CONTENTS

1 Quick Installation Guide 1
1.1 Installing TC-Toolbox for MATLAB® 1
   1.1.1 Automatic Installation of TC-Toolbox 1
   1.1.2 Manual Installation of TC-Toolbox 2
      1.1.2.1 Multiple Windows Users on Same Machine 2
      1.1.2.2 Installer Cannot Find the MATLAB® Installation Directory 2
   1.1.3 Check the Installation 2
1.2 Uninstalling TC-Toolbox for MATLAB® 2

2 Architecture overview 3
2.1 TCToolbox 3
2.2 SystemBuilder and System 4
2.3 Calculation 5
   2.3.1 Single Equilibrium Calculations 5
   2.3.2 Batch Equilibrium Calculations 6
   2.3.3 Precipitation Calculations 7
   2.3.4 Scheil Calculations 7
   2.3.5 Property Diagram Calculations 8
   2.3.6 Phase Diagram Calculations 8
   2.3.7 Diffusion Calculations 9
   2.3.8 Property Model Calculations 9
2.4 Result 10

3 Best Practices 13
3.1 Using Tab-Completion and the Integrated documentation 13
3.2 Re-use of the Single Equilibrium Calculation State 14
3.3 Re-use and Saving Results 15
3.4 Using the TCToolbox class efficiently 16
3.5 Parallel Calculations 16
3.6 Handling Calculation Engine Crashes 18

4 API Reference 19
4.1 Calculations 19
   4.1.1 Package “single_equilibrium” 19
   4.1.2 Package “batch_equilibrium” 28
   4.1.3 Package “precipitation” 33
   4.1.4 Package “scheil” 52
   4.1.5 Package “step_or_map_diagrams” 64
   4.1.6 Package “diffusion” 85
   4.1.7 Package “propertymodel” 176
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Root Package</td>
<td>181</td>
</tr>
<tr>
<td>4.3</td>
<td>Package “system”</td>
<td>202</td>
</tr>
<tr>
<td>5</td>
<td>Troubleshooting</td>
<td>215</td>
</tr>
<tr>
<td>5.1</td>
<td>Diagnostics Script</td>
<td>215</td>
</tr>
</tbody>
</table>
CHAPTER ONE

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\2021b\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

If the Thermo-Calc installer cannot find the MATLAB® installation directory, follow 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\2021b\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.
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:**

```matlab
import tc_toolbox.*

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

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

```matlab
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()`.
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:

```matlab
import tc_toolbox.*

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

%... 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:

```matlab
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:

```matlab
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:

```matlab
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:

```matlab
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:

```matlab
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(),...
      → fraction_of_all_solid_phases());
```

2.3. Calculation
2.3.5 Property Diagram Calculations

For the property diagram (step) calculation, everything that you can configure in the Equilibrium Calculator when choosing Property diagram 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:

```matlab
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:

```matlab
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"), 0.01)
  .calculate()
```

(continues on next page)
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:

```matlab
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:

```matlab
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")

2.4 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
\% \rightarrow state)

gibbs = single_eq_result.get_value_of("G")
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.

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.
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**.

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:

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

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

```matlab
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)
```

3.5 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.

```matlab
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
```
Example:

```matlab
tc_toolbox = TCToolbox();
sys = sys.select_database_and_elements("FEDEMO", elements).get_system();

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

3.5. Parallel Calculations
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",
        "MFEDMO", ["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
```
4.1 Calculations

4.1.1 Package “single_equilibrium”

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.AbstractCalculation`.

**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()**
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()`.

**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_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 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 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.

---

**4.1. Calculations**
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)*  
Runs the batch equilibrium calculation. The calculated `BatchEquilibriumResult` can then be queried for the values of the quantities specified.

Example:

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

**Parameters** **logging_frequency** – Determines how often logging should be done.  
**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:
```python
>>> [[('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:
```python
>>> [[(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:

```matlab
>>> 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.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_grain_aspect_ratio (grain_aspect_ratio)
Enter a numerical value. Default: 1.0.

Parameters grain_aspect_ratio – The grain aspect ratio [-]

set_grain_radius (grain_radius)
Sets grain radius / size. Default: 1.0E-4 m

Parameters grain_radius – The grain radius / size [m]

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 [-]
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

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

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

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

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

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

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

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

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

4.1. Calculations
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: \( \text{interfacial_energy_estimation_prefactor} = 2.5 \Rightarrow 2.5 \times \text{calculated interfacial energy} \)

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_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

```
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 ()
```
Runs the CCT diagram calculation.

**Returns** A PrecipitationCalculationTTToCCTResult 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 an 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**
get_critical_radius_of

- **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

- **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_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

- **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_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

- **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 (time [s], normalized driving force)

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_matrix_composition_in_mole_fraction_of

- **time** – The time [s]
- **element_name** – The element

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

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

- **time** – The time [s]
- **element_name** – The element

Returns A tuple of two lists of floats (time [s], weight 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

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_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_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_for_particle_length_of(precipitate_id, time)`
Returns the number density distribution of a precipitate in dependency of its mean particle length at a certain time.

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_number_density_distribution_for_radius_of(precipitate_id, time)`
Returns the number density distribution of a precipitate in dependency of its mean radius at a certain time.

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_number_density_of**(*precipitate_id*)

Returns the particle number density 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_for_particle_length_of**(*precipitate_id*, *time*)

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

Parameters

- **time** – The time [s]
- **precipitate_id** – The id of a precipitate can either be 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 dependency of its mean radius at a certain time.

Parameters

- **time** – The time [s]
- **precipitate_id** – The id of a precipitate can either be 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.
\texttt{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 \texttt{load\_result\_from\_disk()}

\textbf{Parameters} \texttt{path} – the path to the folder you want the result to be saved in. It can be relative or absolute.

\textbf{Returns} this \texttt{PrecipitationCalculationResult} object

class +tc\_toolbox.+precipitation.PrecipitationCalculationTTToCCTResult (back)

Result of a TTT or CCT precipitation calculation.

\texttt{PrecipitationCalculationTTToCCTResult (back)}

Call base constructor: \texttt{tc\_toolbox\_precipitation\_PrecipitationCalculationResult}. get\_result\_for\_precipitate (precipitate\_id)

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

\textbf{Parameters} \texttt{precipitate\_id} – The id of a precipitate can either be the phase name or an alias

\textbf{Returns} A tuple of two lists of floats (time [s], temp [K])

\texttt{invalidate ()}

Invalidates the object and frees the disk space used by it. \textit{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.

\texttt{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 \texttt{load\_result\_from\_disk()}

\textbf{Parameters} \texttt{path} – the path to the folder you want the result to be saved in. It can be relative or absolute.

\textbf{Returns} this \texttt{PrecipitationCalculationResult} object.

class +tc\_toolbox.+precipitation.PrecipitationIsoThermalCalculation (back)

Configuration for an isothermal precipitation calculation.

\texttt{PrecipitationIsoThermalCalculation (back)}

Call base constructor: \texttt{tc\_toolbox\_AbstractCalculation}. calculate ()

Runs the isothermal precipitation calculation.

\textbf{Returns} A \texttt{PrecipitationCalculationSingleResult} which later can be used to get specific values from the calculated result

\texttt{get\_configuration\_as\_string ()}

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

\begin{quote}
\textbf{Warning:} The structure of the calculator objects is an implementation detail and might change between releases without notice. \textbf{Therefore do not rely on the internal object structure.}
\end{quote}

\texttt{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 \texttt{with\_system\_modifications()}. Note: Parameters can only be read from unencrypted (i.e. user) databases loaded as *\texttt{.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 ()
Runs the non-isothermal precipitation calculation.

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 PrecipitationNonIsoThermalCalculation 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()
Runs the TTT diagram calculation.

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 $c \times \text{cooling rate}^{-n}$ 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 $c \times \text{cooling rate}^{-n}$ 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: $c \times \text{cooling rate}^{-n}$ 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 \( \text{Scanning speed} \times \cos(\text{angle}) \) with \( \text{Scanning speed} \) and \( \text{angle} \) 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 \( c \times \text{cooling_rate}^{(-n)} \).

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 \( c \times \text{cooling_rate}^{(-n)} \).

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: \( c \ast \text{cooling_rate}^{(-n)} \) 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` object

**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` 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: \( c \times \text{cooling_rate}^{-n} \) 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.

:returns: 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()**

Runs the Scheil calculation.

**Warning:** Scheil calculations do not support the GAS phase being selected, this means the **GAS phase must always be deselected in the system** if it is present in the database.

**Returns** A **ScheilCalculationResult** which later can be used to get specific values from the simulation.

**disable_global_minimization()**

Disables global minimization.

**Default:** Disabled

**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:** Disabled

**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()
Invalidate 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_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 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_back_diffusion(scheil_back_diffusion)
Enables 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 performed.
Parameters **scheil_back_diffusion** – an instance of a *ScheilBackDiffusion* class, where the options for back diffusion can be specified.

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_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 \( \text{Scanning speed} \times \cos(\text{angle}) \) with \( \text{Scanning speed} \) and \( \text{angle} \) 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`.

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 \( \text{Scanning speed} \times \cos(\text{angle}) \) with \( \text{Scanning speed} \) and \( \text{angle} \) 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`. 
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_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_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_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_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 solidification 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

```matlab
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

```matlab
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”

```matlab
class +tc_toolbox.+step_or_map_diagrams.AbstractAxisType
```
The abstract base class for all axis types.

```matlab
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()`.

```matlab
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
```

```matlab
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
```

```matlab
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
Configuration for a phase diagram calculation.

Note: Specify the conditions, the calculation is performed with calculate().

PhaseDiagramCalculation
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)
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.

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))
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 ThermodynamicCalculation 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 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**\( (\text{phase}, \text{amount}) \)

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**\( (\text{phase}) \)

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**\( (\text{axis}) \)

Sets the first calculation axis.

**Parameters** **axis** – The axis

**Returns** This `PhaseDiagramCalculation` object

**with_options**\( (\text{options}) \)

Sets the simulation options.

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

**Returns** This `PhaseDiagramCalculation` object

**with_reference_state**\( (\text{component, phase, temperature, pressure}) \)

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`
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 dependence 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:** It’s 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*I.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:** It’s 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()**
Validates 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.

**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** phaseNameStyle - The phase name style

**Returns** This PhaseDiagramResult object

**PhaseDiagramResultValues(back)**
Represents the data of a phase diagram.

**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)
Configuration for a property diagram calculation.

Note: Specify the conditions, the calculation is performed with calculate().

PropertyDiagramCalculation(back)
Call base constructor: tc_toolbox.AbstractCalculation.

calculate (keep_previous_results)
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.

Returns A new PropertyDiagramResult object which later can be used to get specific values from the calculated result

disable_global_minimization()
Disables global minimization.
**disable_step_separate_phases()**
Disables \textit{step separate phases}. This is the \textbf{default} setting.

Returns This \textit{PropertyDiagramCalculation} object

**enable_global_minimization()**
Enables \textit{global minimization}.

Default: Enabled

Returns This \textit{PropertyDiagramCalculation} object

**enable_step_separate_phases()**
Enables \textit{step separate phases}.

Default: By default separate phase stepping is \textit{disabled}

\textbf{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 \textit{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.

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

**get_gibbs_energy_addition_for(\textit{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 \textit{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 \textit{with_system_modifications()}. 

\textbf{Note:} Parameters can only be read from unencrypted (i.e. \textit{user}) databases loaded as *.tdb-file.
Returns The system data

invalidate() Invalidate 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: It's 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()**
Invalidate 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:

\[ J = f(T,P,TIME) \ast (ACTIVITY^N \ast g(T,P,TIME)) \]

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.


Constructs an instance of \textit{ActivityFluxFunction}.

\texttt{static activity\_flux\_function} ()
Factory method that creates a \texttt{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:
\[ J = f(T,P,\text{TIME}) \times \text{(ACTIVITY}^N - g(T,P,\text{TIME}) \text{)} \]

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.


Returns A new \textit{ActivityFluxFunction} object

\texttt{static closed\_system} ()
Factory method that creates a \texttt{new} closed-system boundary condition.

Returns A new \textit{ClosedSystem} object

\texttt{static fix\_flux\_value} ()
Factory method that creates a \texttt{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: \( J(T,P,\text{TIME}) \).

Returns A new \textit{FixFluxValue} object

\texttt{static fixed\_compositions} \( (\text{unit\_enum} \) )
Factory method that creates a \texttt{new} fixed-composition boundary condition.

Parameters \texttt{unit\_enum} – The composition unit

Returns A new \textit{FixedCompositions} object

\texttt{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:

\[ J = f(T,P,TIME) \times (ACTIVITY^n - g(T,P,TIME)) \]

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

```python
def get_type() -> str:
    """The type of the solver."""
    return "Classic"
```

Returns  The type

```python
def static homogenization() -> str:
    """Factory method that creates a new homogenization solver."
    return "Homogenization"
```

**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

```python
def set_flux_balance_equation_accuracy(accuracy: float) -> str:
    """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"
    return "Automatic"
```

Parameters  accuracy – The required accuracy

Returns  A new `AutomaticSolver` object

```python
def set_tieline_search_variable_to_activity() -> str:
    """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"
    return "Automatic"
```

Returns  A new `AutomaticSolver` object

```python
def set_tieline_search_variable_to_potential() -> str:
    """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"
    return "Automatic"
```

Returns  A new `AutomaticSolver` object

```python
class +tc_toolbox.+diffusion.BoundaryCondition
    Contains factory methods for the different boundary conditions available.
```

```python
def static activity_flux_function() -> str:
    """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:
    \[ J = f(T,P,TIM) \times (ACTIVITY^N - g(T,P,TIM)) \]
    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."
    return "Activity"
```

Chapter 4. API Reference
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.


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: \( J(T,P,\text{TIME}) \).

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

Returns A new ClassicSolver object

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:

\[ J = f(T,P,TIME) \times (ACTIVITY^N - g(T,P,TIME)) \]

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

```java
static closed_system()
    Factory method that creates a new closed-system boundary condition.
    Returns A new *ClosedSystem* object
```

```java
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
```

```java
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
```

```java
get_type()
    Convenience method for getting the type of the boundary condition.
    Returns The type of the boundary condition
```

```java
static mixed_zero_flux_and_activity()
    Factory method that creates a new mixed zero-flux and activity boundary condition
    Returns A new *MixedZeroFluxAndActivity* object
```

```java
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
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.

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 ()
Runs the diffusion calculation.

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 \( (\text{back}) \)
Call base constructor: `tc_toolbox.AbstractResult`.

**get_mass_fraction_at_lower_interface** \( (\text{region}, \text{component}) \)
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** \( (\text{region}, \text{component}) \)
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** \( (\text{component}, \text{time}) \)
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** \( (\text{phase}, \text{time}) \)
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 the 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 un-
til 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()
Runs the diffusion calculation.

4.1. 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.
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 ex-
plicitly 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()

Runs the diffusion calculation.

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”.

4.1. Calculations
**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 a 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.

**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 param-
eters 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

```matlab
get_lower_geometrical_factor()
Returns the lower geometrical factor (for the left half).

Returns The lower geometrical factor
```

```matlab
get_no_of_points()
Returns number of grid points.

Returns The number of grid points
```

```matlab
get_type()
Type of the grid.

Returns The type of the grid
```

```matlab
get_upper_geometrical_factor()
Returns the upper geometrical factor (for the right half).

Returns The upper geometrical factor
```

```matlab
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
```

```matlab
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
```

```matlab
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
```

```matlab
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
```

```matlab
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 function** *(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 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:

\[ J = f(T,P,TIME) \times (ACTIVITY^N - g(T,P,TIME)) \]

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.

**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**

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**

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 \( J(T,P,TIME) \)
• \texttt{tc.time} – The max-time for which the flux function is used.

\begin{verbatim}
class +tc_toolbox.+diffusion.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:
  \[ J = f(T,P,\text{TIME}) \times (ACTIVITY^N - g(T,P,\text{TIME})) \]
  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 corre-
  sponding 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 con-

  Returns A new \texttt{ActivityFluxFunction} object

static closed_system()
  Factory method that creates a new closed-system boundary condition.

  Returns A new \texttt{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 vol-
  ume for the independent components. May be a function of time, temperature and pressure: \( J(T,P,\text{TIME}) \).

  Returns A new \texttt{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 \texttt{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 \texttt{MixedZeroFluxAndActivity} object
\end{verbatim}
`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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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.

4.1. Calculations 119
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

```java
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

```java
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

```java
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

```java
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

```java
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

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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

```matlab
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

```matlab
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.

**Parameters** `excluded_phases` – The excluded phases

```matlab
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

```matlab
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.

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

```java
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

```java
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

```java
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

```java
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 \texttt{HashinShtrikmanBoundPrescribedExcludedPhase} object

\textbf{static inverse\_rule\_of\_mixtures} ()
Factory method that creates a new homogenization function of the type \texttt{InverseRuleOfMixtures}.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

Returns A new \texttt{InverseRuleOfMixtures} object

\textbf{static inverse\_rule\_of\_mixtures\_excluded\_phase} (\texttt{excluded\_phases})
Factory method that creates a new homogenization function of the type \texttt{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 \texttt{excluded\_phases} – The excluded phases

Returns A new \texttt{InverseRuleOfMixturesExcludedPhase} object

\textbf{static labyrinth\_factor\_f} (\texttt{matrix\_phase})
Factory method that creates a new homogenization function of the type \texttt{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 \texttt{matrix\_phase} – The matrix phase

Returns A new \texttt{LabyrinthFactorF} object

\textbf{static labyrinth\_factor\_f2} (\texttt{matrix\_phase})
Factory method that creates a new homogenization function of the type \texttt{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 \texttt{matrix\_phase} – The matrix phase

Returns A new \texttt{LabyrinthFactorF2} object

\textbf{static rule\_of\_mixtures} ()
Factory method that creates a new homogenization function of the type \texttt{RuleOfMixtures}.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

Returns A new \texttt{RuleOfMixtures} object

\textbf{static rule\_of\_mixtures\_excluded\_phase} (\texttt{excluded\_phases})
Factory method that creates a new homogenization function of the type \texttt{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 \texttt{excluded\_phases} – The excluded phases

Returns A new \texttt{RuleOfMixturesExcludedPhase} object

\textbf{class +tc\_toolbox\.+diffusion\.+HashinShtrikmanBoundMajorityExcludedPhase} (\texttt{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 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

**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

```java
static hashin_shtrikman_bound_majority() (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

```java
static hashin_shtrikman_bound_majority_excluded_phase(excluded_phases) (hashin_shtrikman_bound_majority_excluded_phase)
```

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

```java
static hashin_shtrikman_bound_prescribed(matrix_phase) (hashin_shtrikman_bound_prescribed)
```

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

```java
static hashin_shtrikman_bound_prescribed_excluded_phase(matrix_phase, excluded_phases) (hashin_shtrikman_bound_prescribed_excluded_phase)
```

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. 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

```java
static inverse_rule_of_mixtures() (inverse_rule_of_mixtures)
```

Factory method that creates a new homogenization function of the type `InverseRuleOfMixtures`. 

---

Chapter 4. API Reference
Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

**Returns** A new `InverseRuleOfMixtures` object

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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
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

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

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

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

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 \texttt{InverseRuleOfMixtures} object

\textbf{static inverse\_rule\_of\_mixtures\_excluded\_phase} (\texttt{excluded\_phases})

Factory method that creates a new homogenization function of the type \texttt{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** \texttt{excluded\_phases} – The excluded phases

Returns A new \texttt{InverseRuleOfMixturesExcludedPhase} object

\textbf{static labyrinth\_factor\_f} (\texttt{matrix\_phase})

Factory method that creates a new homogenization function of the type \texttt{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** \texttt{matrix\_phase} – The matrix phase

Returns A new \texttt{LabyrinthFactorF} object

\textbf{static labyrinth\_factor\_f2} (\texttt{matrix\_phase})

Factory method that creates a new homogenization function of the type \texttt{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** \texttt{matrix\_phase} – The matrix phase

Returns A new \texttt{LabyrinthFactorF2} object

\textbf{static rule\_of\_mixtures} ()

Factory method that creates a new homogenization function of the type \texttt{RuleOfMixtures}.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

Returns A new \texttt{RuleOfMixtures} object

\textbf{static rule\_of\_mixtures\_excluded\_phase} (\texttt{excluded\_phases})

Factory method that creates a new homogenization function of the type \texttt{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** \texttt{excluded\_phases} – The excluded phases

Returns A new \texttt{RuleOfMixturesExcludedPhase} object

\texttt{class +tc\_toolbox.\+diffusion.HomogenizationFunction}

\textit{Homogenization function} used for the \textit{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:** \texttt{RULE\_OF\_MIXTURES} (i.e. upper Wiener bounds)

\texttt{class +tc\_toolbox.\+diffusion.HomogenizationFunctions}

\textbf{static general\_lower\_hashin\_shtrikman} ()

Factory method that creates a new homogenization function of the type \texttt{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

```matlab
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

```matlab
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

```matlab
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

```matlab
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

```matlab
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

```matlab
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

```matlab
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`.

```matlab
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

```matlab
get_type()
```
The type of solver.

Returns The type

```matlab
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_homogenization_function(homogenization_function_enum)
Sets the homogenization function used by the homogenization model. Default is RULE_OF_MIXTURES.

Parameters homogenization_function_enum – The homogenization function used by the homogenization model

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

```csharp
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

```csharp
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

```csharp
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

```csharp
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

```csharp
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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

---

4.1. Calculations
**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

```java
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

```java
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

```java
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

```java
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

```java
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

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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 \textit{LabyrinthFactorF2} object

\texttt{static rule_of_mixtures()}

Factory method that creates a new homogenization function of the type \textit{RuleOfMixtures}.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

Returns A new \textit{RuleOfMixtures} object

\texttt{static rule_of_mixtures_excluded_phase(excluded_phases)}

Factory method that creates a new homogenization function of the type \textit{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 \texttt{excluded_phases} – The excluded phases

Returns A new \textit{RuleOfMixturesExcludedPhase} object

class +tc_toolbox.+diffusion.LinearGrid\texttt{(no_of_points)}

Represents an equally spaced grid.

\texttt{LinearGrid(no_of_points)}

Creates an equally spaced grid.

Parameters \texttt{no_of_points} – The number of points

static \texttt{double_geometric(no_of_points, lower_geometrical_factor, upper_geometrical_factor)}

Factory method that creates a new double geometric grid.

\textbf{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

\begin{itemize}
  \item \texttt{no_of_points} – The number of points
  \item \texttt{lower_geometrical_factor} – The geometrical factor for the left half
  \item \texttt{upper_geometrical_factor} – The geometrical factor for the right half
\end{itemize}

Returns A new \textit{DoubleGeometricGrid} object

\texttt{static geometric(no_of_points, geometrical_factor)}

Factory method that creates a new geometric grid.

\textbf{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

\begin{itemize}
  \item \texttt{no_of_points} – The number of points
  \item \texttt{geometrical_factor} – The geometrical factor
\end{itemize}
Returns A new GeometricGrid object

get_no_of_points()
Returns the number of grid points.

Returns The number of grid points

getype()
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:

\[ J = f(T, P, \text{TIME}) \times (\text{ACTIVITY}^N - g(T, P, \text{TIME})) \]

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.

4.1. Calculations  161

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 Fix FluxValue 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 Mixed Zero Flux And Activity 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)
• $\sinh(X)$, $\cosh(X)$, $\tanh(X)$, $\text{asinh}(X)$, $\text{acosh}(X)$, $\text{atanh}(X)$

• $\text{sign}(X)$

• $\text{erf}(X)$ 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.

```matlab
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

```matlab
class +tc_toolbox.+diffusion.options
```

General simulation conditions for the diffusion calculations.

**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 than 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 \( DGM(\text{ph}) \)

**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 \texttt{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 \texttt{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 \texttt{with_point_by_point_grid_containing_compositions()} needs to be applied.

**Note:** The composition profiles need to be defined separately using \texttt{with_composition_profile()}, additionally the region width needs to be specified using \texttt{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, ...) \texttt{with_grid()} should be used instead.

**Note:** If a point-by-point grid is used, \texttt{with_grid()}, \texttt{with_composition_profile()} and \texttt{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

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

```latex
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

```latex
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

```latex
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

```latex
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

```latex
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

```java
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

```java
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** `matrix_phase` – The matrix phase

**Returns** A new `HashinShtrikmanBoundPrescribed` object

```java
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 `HashinShtrikmanBoundPrescribedExcludedPhase` object

```java
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. 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

```java
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

```java
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

```java
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

```java
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

```java
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].

```java
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

```java
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

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 smallest 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)
Registers a POLY 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 remove_all_poly_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 POLY-commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten parenthesis, ...).

Parameters poly_command – The POLY Console Mode command

Returns This PropertyModelCalculation object

calculate ()
Runs the Property Model calculation.

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.

```matlab
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

```matlab
remove_all_conditions()
```
Removes all set classic POLY conditions.

**Note:** This does not affect the compositions set by `set_composition()`.

**Returns**  
This `PropertyModelCalculation` object

```matlab
remove_all_poly_commands()
```
Removes all previously added POLY Console Mode commands.

**Returns**  
This `PropertyModelCalculation` object

```matlab
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

```matlab
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

```matlab
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(\textit{unit\_enum})

Sets the composition unit.

\textbf{Default}: Mole percent (CompositionUnit.MOLE\_PERCENT).

\textbf{Parameters} \textit{unit\_enum} – The new composition unit

\textbf{Returns} This \textit{PropertyModelCalculation} object

set_condition(\textit{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 \textit{set\_dependent\_element()}.

\textbf{Default if not specified}: pressure \( P = 1e5 \) Pa, system size \( N = 1 \), Temperature \( T = 1000 \) K

\textbf{Note}: It should not be necessary for most users to use this method, try to use \textit{set\_composition()} instead.

\textbf{Warning}: It is not possible to mix POLY-commands and compositions using \textit{set\_composition()}.

\textbf{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, ...).

\textbf{Parameters}

\begin{itemize}
  \item \textit{classic\_condition} – The classic POLY condition (for example: \( X(CR) \))
  \item \textit{value} – The value of the condition
\end{itemize}

\textbf{Returns} This \textit{PropertyModelCalculation} object

set_dependent_element(\textit{dependent\_element\_name})

Sets the dependent element manually.

\textbf{Note}: It should not be necessary for most users to use this method. Setting the dependent element manually is only necessary and allowed if \textit{set\_condition()} is used.

\textbf{Parameters} \textit{dependent\_element\_name} – The name of the dependent element

\textbf{Returns} This \textit{PropertyModelCalculation} object

set_model_parameter(\textit{model\_parameter\_id, value})

Resets an optimizable model parameter. The id can be obtained with \textit{get\_model\_parameters()}.

\textbf{Parameters}

\begin{itemize}
  \item \textit{model\_parameter\_id} – The model parameter id
  \item \textit{value} – The new value of the parameter
\end{itemize}

\textbf{Returns} This \textit{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.

**Parameters**

- result_quantity_id – The result quantity id

**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_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.
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.CompositionUnit

The composition unit.
class +tc_toolbox.Constants

    ALL_COMPONENTS = "'\*'"
    ALL_PHASES = "'\*'"
    CURRENT_TEMPERATURE = '-1.0'
    SER = "'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 [m²/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 [m²/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 [m²/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 \texttt{component} – The name of the component

Returns A new \texttt{TotalMoleFractionOfComponent} object.

\texttt{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

\begin{itemize}
  \item \texttt{phase} – The name of the phase
  \item \texttt{component} – The name of the component
\end{itemize}

Returns A new \texttt{TotalMoleFractionOfComponentInPhase} object.

\texttt{static total\_volume\_fraction\_of\_phase (phase)}

Creates a quantity representing the total volume fraction of a phase.

Parameters \texttt{phase} – The name of the phase.

Returns A new \texttt{TotalVolumeFractionOfPhase} object.

\texttt{static tracer\_diffusion\_coefficient (phase, diffusing\_element)}

Creates a quantity representing tracer diffusion coefficient of a phase [m$^2$/s].

Parameters

\begin{itemize}
  \item \texttt{phase} – The name of the phase
  \item \texttt{diffusing\_element} – The diffusing element
\end{itemize}

Returns A new \texttt{TracerDiffusionCoefficient} object.

\texttt{static u\_fraction\_of\_a\_component (component)}

Creates a quantity representing the u-fraction of a component.

Parameters \texttt{component} – The name of the component

Returns A new \texttt{UFractionOfAComponent} object.

\texttt{static user\_defined\_function (expression)}

Creates a quantity representing a user-defined function.

Parameters \texttt{expression} – The function expression

Returns A new \texttt{Function} object

\texttt{static velocity\_of\_lower\_boundary\_of\_region (region)}

Creates a quantity representing the velocity of lower boundary of a region [m/s].

Parameters \texttt{region} – The name of the region

Returns A new \texttt{VelocityOfLowerBoundaryOfRegion} object.

\texttt{static velocity\_of\_upper\_boundary\_of\_region (region)}

Creates a quantity representing the velocity of upper boundary of a region [m/s].

Parameters \texttt{region} – The name of the region

Returns A new \texttt{VelocityOfUpperBoundaryOfRegion} object.

\texttt{static width\_of\_region (region)}

Creates a quantity representing the width of a region [m].

Parameters \texttt{region} – The name of the region

Returns A new \texttt{WidthOfRegion} object.
**Class: +tc_toolbox.IndependentVariable**

Factory class providing quantities used for defining the independent variable in general diffusion result querying.

**static distance(\texttt{region})**

Creates an independent variable representing the distance [m].

**Returns** A new \texttt{Distance} object

**static time()**

Creates an independent variable representing the time [s].

**Returns** A new \texttt{Time} object

**Class: +tc_toolbox.InterfacePosition**

The position of an interface relative to its region. Only used for diffusion simulations.

**Class: +tc_toolbox.PhaseParameter(\texttt{parameter\_name})**

Database phase parameter expression used by \texttt{SystemModifications.set()}. Parameters \texttt{parameter\_name} – The phase parameter name

**PhaseParameter(\texttt{parameter\_name})**

Constructs an instance of \texttt{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 \texttt{PhaseParameter} object

**remove\_interval\_with\_upper\_limit(\texttt{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 \texttt{PhaseParameter} object

**set\_expression\_with\_upper\_limit(\texttt{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 \texttt{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.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 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_solid_phase(phase)
  Creates a quantity representing the average density of a solid phase [g/cm^3].

  Parameters phase – The name of the phase or ALL_PHASES to choose all solid phases

  Returns A new DensityOfSolidPhase 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 [-].
    
    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.

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.
class +tc_toolbox.SystemData

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

Constructs an instance of SystemData.

get_phase_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

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
\texttt{set\_interval(interval)}

Adds/overwrites a temperature interval.

\textbf{Note:} The lower temperature limit is either defined by the lower temperature limit given with \texttt{SystemFunction.set\_lower\_temperature\_limit()} or by the upper temperature limit of the adjacent interval.

\textbf{Note:} If there is an existing interval with exactly the same \texttt{upper\_temperature\_limit}, that interval is overwritten, otherwise the interval is added.

\textbf{Returns} This \texttt{SystemFunction} object

\texttt{set\_lower\_temperature\_limit(lower\_temperature\_limit)}

Sets the lower temperature limit of the system function.

\textbf{Default:} 298.15 K

\textbf{Parameters} \texttt{lower\_temperature\_limit} – The lower limit in K

\textbf{Returns} This \texttt{SystemFunction} object

\textbf{class} \texttt{+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 \texttt{+tc\_toolbox.abstract\_base.AbstractCalculation.with\_system\_modifications()} on a calculator object.

\textbf{SystemModifications()}

Constructs an instance of \texttt{SystemModifications}.

\texttt{run\_ges\_command(ges\_command)}

Sends a GES-command. This is actually applied when running `with\_system\_modifications` on a calculator object.

Example: \texttt{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 \textit{Fe} as major constituent on first sublattice and \textit{C} as major constituent on second sublattice.

\textbf{Note:} For details about the syntax search the Thermo-Calc help for GES (the name for the Gibbs Energy System module in Console Mode).

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

\textbf{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, ...).
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

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.
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 SystemBuilder 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

4.3. Package “system” 205
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.

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

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.
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.


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_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.
  
Warning: Scheil calculations do not support the GAS phase being selected, this means the `GAS`
  phase must always be deselected in the system if it is present in the database

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

**get_system_for_scheil_calculations()**

Creates a new System object without gas phases being selected, that is the basis for all calculation types, but its particularly useful for Scheil solidification calculations, where the model does not allow that a gas phase is selected in the system.

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.

**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
CHAPTER FIVE

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

toolbox_version = "2021b";

clc

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")

disp('Url of license server: (if license server is NO-NET, you need a local license file)')
```

(continues on next page)
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', 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')