Reaktoro Documentation

Allan Leal

Feb 08, 2019
## Getting Started

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Overview</td>
<td>3</td>
</tr>
<tr>
<td>2 Installation</td>
<td>7</td>
</tr>
<tr>
<td>3 Tutorials: Chemical Equilibrium</td>
<td>11</td>
</tr>
<tr>
<td>4 Tutorials: Chemical Kinetics</td>
<td>19</td>
</tr>
<tr>
<td>5 Tutorials: Chemical Transport</td>
<td>21</td>
</tr>
<tr>
<td>6 Thermodynamic Databases</td>
<td>23</td>
</tr>
<tr>
<td>7 Thermodynamic Backends</td>
<td>27</td>
</tr>
<tr>
<td>8 Acknowledgements</td>
<td>33</td>
</tr>
<tr>
<td>9 License</td>
<td>35</td>
</tr>
</tbody>
</table>
Reaktoro is a computational framework developed in C++ and Python that implements numerical methods for modeling chemically reactive processes governed by either chemical equilibrium, chemical kinetics, or a combination of both.

Note: This guide is under active development. You may find some pages incomplete at the moment. 17.12.2018
Below is a quick overview of the chemical equilibrium and kinetics capabilities of Reaktoro. Read this entire guide to learn more about what you can do with Reaktoro!

### 1.1 Chemical Equilibrium

Here is a simple C++ code using Reaktoro to perform a multiphase chemical equilibrium calculation:

```cpp
#include <Reaktoro/Reaktoro.hpp>
using namespace Reaktoro;

int main()
{
    ChemicalEditor editor;
    editor.addAqueousPhase("H2O NaCl CaCO3 CO2");
    editor.addGaseousPhase("CO2(g)");
    editor.addMineralPhase("Calcite");

    ChemicalSystem system(editor);

    EquilibriumProblem problem(system);
    problem.add("H2O", 1, "kg");
    problem.add("CO2", 1, "mol");
    problem.add("NaCl", 0.7, "mol");
    problem.add("CaCO3", 1, "g");

    ChemicalState state = equilibrate(problem);

    state.output("result.txt");
}
```

This calculation could also be performed using Reaktoro’s Python interface:
from reaktoro import *

editor = ChemicalEditor()
editor.addAqueousPhase('H2O NaCl CaCO3 CO2')
editor.addGaseousPhase('CO2(g)')
editor.addMineralPhase('Calcite')

system = ChemicalSystem(editor)

problem = EquilibriumProblem(system)
problem.add('H2O', 1, 'kg')
problem.add('CO2', 1, 'mol')
problem.add('NaCl', 0.7, 'mol')
problem.add('CaCO3', 1, 'g')

state = equilibrate(problem)
state.output('result.txt')

1.2 Chemical Kinetics

Reaktoro can also perform chemical kinetic calculations with both equilibrium-controlled and kinetically-controlled reactions. The C++ example below demonstrate this for a simple mineral dissolution modeling, in which CaCO3(s, calcite) reacts with a carbonated aqueous solution:

```cpp
#include <Reaktoro/Reaktoro.hpp>
using namespace Reaktoro;

int main()
{
    ChemicalEditor editor;
    editor.addAqueousPhase("H2O CO2 CaCO3");
    editor.addMineralPhase("Calcite");

    MineralReaction reaction = editor.addMineralReaction("Calcite");
    reaction.setEquation("Calcite = Ca++ + CO3--");
    reaction.addMechanism("logk = -5.81 mol/(m2*s); Ea = 23.5 kJ/mol");
    reaction.addMechanism("logk = -0.30 mol/(m2*s); Ea = 14.4 kJ/mol; a[H+] = 1.0");
    reaction.setSpecificSurfaceArea(10, "cm2/g");

    ChemicalSystem system(editor);
    ReactionSystem reactions(editor);

    Partition partition(system);
    partition.setKineticSpecies("Calcite");

    EquilibriumProblem problem(system);
    problem.setPartition(partition);
    problem.setTemperature(60, "celsius");
    problem.setPressure(100, "bar");
    problem.add("H2O", 1, "kg");
    problem.add("CO2", 0.1, "mol");

    ChemicalState initialstate = equilibrate(problem);
```

(continues on next page)
When the application is executed, the following figures are produced:

In the example above, the mineral reaction is specified to be under kinetic control and the aqueous species in chemical equilibrium at all times. As the mineral dissolves, it perturbs the chemical equilibrium state of the aqueous species. By assuming the aqueous species to be always in equilibrium, it is like if they were capable of reacting instantaneously to a new state of equilibrium. In general, the aqueous species react among themselves at much faster rates than mineral dissolution reactions, and thus this partial equilibrium assumption is plausible, and fairly accurate in most cases.
There are many ways for installing Reaktoro, from a simple single-line terminal command to building the entire project from sources. All current possibilities are detailed below.

2.1 Installation using Conda

Reaktoro can be easily installed using Conda, a powerful package manager used to simplify Reaktoro’s installation and the management of its external software dependencies. Once you install Conda, and append the necessary channels, you’ll be able to install Reaktoro by just executing the following command in your terminal:

```
conda install reaktoro
```

Follow the Conda installation steps shown next before you execute this command!

2.1.1 Installing Conda

Conda can be installed by installing either Anaconda or Miniconda. We recommend the installation of Miniconda, unless you already have Anaconda installed or you think you need 1,400+ software packages that ship with it! Miniconda is just a tiny subset of Anaconda containing only our needed conda application and its dependencies.

Download the Miniconda Installer by clicking on the image below:

Fig. 1: Reaktoro is a proud user of Conda, a powerful and modern package manager!

You’ll be given options to install Miniconda for Windows, Mac OS X, and Linux (32-bit or 64-bit) using either Python 3.7 or Python 2.7. We recommend a 64-bit installer of Miniconda with Python 3.7.

2.1.2 Adding conda-forge channels

The Reaktoro pre-built package is hosted on conda-forge. After installing Miniconda, go to a terminal and execute:
conda config --append channels conda-forge
conda config --append channels conda-forge/label/gcc7
to add the conda-forge channels required to find the Reaktoro package.
All should now be set to install Reaktoro using:
conda install reaktoro
Go to Reporting a failed installation if this does not work for you.

2.2 Installation using CMake

Reaktoro has several software and library dependencies that need to be pre-installed for its successful compilation and installation using CMake. To greatly simplify the building process of Reaktoro for Windows, Mac OS X, and Linux, you’ll need Conda. Follow the Conda installation steps in the previous section, in which a Miniconda installer is used.
After installing Miniconda, go to a terminal and execute:
conda install -n base conda-devenv
This installs conda-devenv, a conda tool with convenient functionalities to define and initialize conda environments.

2.2.1 Downloading Reaktoro from GitHub

We need now to download the source code of Reaktoro, which is hosted on GitHub. This can be done by either executing the following git command from the terminal (if you already have git installed!):
git clone https://github.com/reaktoro/reaktoro.git
or by directly downloading reaktoro-master.zip, the latest version of Reaktoro’s source code in a zip file.

Note: If you use the direct download option above, please unzip the downloaded file in a directory of your choice. We assume the unzipped folder is named reaktoro for the next installation steps, and not reaktoro-master.

2.2.2 Creating a conda environment for Reaktoro

The next step is to create a conda environment that contains all the software and library dependencies needed to build Reaktoro. In the root of the reaktoro directory, execute:
conda devenv
This command will create the conda environment called reaktoro, which can take a few minutes to complete for the first time.

Attention: You only need to execute conda devenv again when the list of external dependencies changes or some configuration in the conda environment reaktoro is altered.
2.2.3 Activating the conda environment for Reaktoro

The next step is to activate the conda environment `reaktoro` that `conda-devenv` created for us:

```
conda activate reaktoro
```

**Attention:** You need to activate the `reaktoro` conda environment whenever you use Reaktoro from C++ or Python! This is because conda will adjust some environment variables in your system (e.g., `PYTHONPATH`, `LD_LIBRARY_PATH`, `PATH`) so that Reaktoro’s libraries, executables, and Python packages can be found. Activating the `reaktoro` conda environment is the simplest way to get these environment variables set correctly.

2.2.4 Building and installing Reaktoro with CMake

You can now build and install Reaktoro by executing the following from the root of the reaktoro source directory:

```
cmake -P install
```

Assuming the conda environment `reaktoro` is active, this command will first build Reaktoro and then install its header files, libraries, executables, Python package in your local `miniconda` directory:

**Linux** /home/user/miniconda3/envs/reaktoro/

**Windows** C:\miniconda3\envs\reaktoro\

Alternatively, to build and install Reaktoro in a more traditional way, execute the following from the root directory of Reaktoro’s source code:

```
mkdir build
cd build
cmake ..
cmake --build . --target install
```

The following is also possible with CMake v3.13 or newer:

```
cmake -S . -B build
cmake --build build/ --target install
```

**Tip:** Compiling the Reaktoro C++ library and the Reaktoro Python module should take a few minutes for the first time. However, if you activate the reaktoro conda environment, `ccache` will be used to significantly speed up future compilations automatically for you!

2.2.5 Installing Reaktoro in a custom directory

To install Reaktoro in a different directory, say, `/home/user/other`, use:

```
2.2. Installation using CMake
9
```
You’ll need, however, to set the environment variables PYTHONPATH, LD_LIBRARY_PATH, and PATH yourself. For example, in Linux:

```
export PATH=$PATH:/home/user/other/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/user/other/lib{64}
export PYTHONPATH=$PYTHONPATH:/home/user/other/lib{64}/pythonX.Y/site-packages
```

where lib{64} is either lib or lib64, and pythonX.Y where pythonX.Y depends on the python version used to compile Reaktoro’s Python package (e.g., python3.6, python3.7).

### 2.2.6 Checking for a successful installation

Check if Reaktoro was installed correctly by executing:

```
python -c "import reaktoro; print(reaktoro.__path__[0])"
```

This should print the path to the installed python package reaktoro. For example:

```
/home/user/miniconda3/envs/reaktoro/lib/pythonX.Y/site-packages/reaktoro
```

where pythonX.Y above depends on the python version used.

**Attention:** Make sure you have the conda environment reaktoro active! Otherwise the checking above might not work without further actions (e.g., changing the PYTHONPATH environment variable).

If you get instead something like:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'reaktoro'
```

then the installation was not successful or it was installed in a custom path that is not yet given in the PYTHONPATH environment variable.

### 2.3 Reporting a failed installation

A failed installation can be a frustrating emotion, but we will be happy to help you fixing your installation issue. However, please do make sure you followed exactly the steps given before. If you are sure that you followed every single instruction and the installation still fails, please go to:

Reaktoro’s GitHub Issues

and let us know!
Chemical equilibrium calculations are essential for many chemical reaction modeling problems. Below is a list of tutorials demonstrating how Reaktoro’s computational chemical equilibrium capabilities using Gibbs energy minimization algorithms can be applied to solve different modeling problem.

### 3.1 Solubility of CO₂ in NaCl brines

In this tutorial, we show how Reaktoro can be used to compute the solubility of CO₂ in a 1 molal NaCl brine at temperature 60 °C and pressure 100 bar. We show no magical function to perform such calculation in a single line of code, but instead a sequence of steps using Reaktoro’s components (classes, methods) to enrich your understanding of how Reaktoro can be used for solving this and many other different chemical reaction modeling problems.

To calculate the solubility of CO₂ in a NaCl brine, we need two phases in our chemical system: an aqueous phase to represent our NaCl brine, and a gaseous phase to represent our CO₂ gas. Next, we formulate and solve a chemical equilibrium problem, in which 10 mol of CO₂ is mixed with 1 kg of H₂O and 1 mol of NaCl, at 60°C and 100 bar. The solution to this problem is a chemical equilibrium state from which we can inspect how much CO₂ exist in the gaseous phase and compare this with the initial amount of CO₂ we used to mix with H₂O and NaCl.

**Note:** Our 1 molal NaCl brine is here represented by the mixture of 1 kg of H₂O and 1 mol of NaCl.

**Note:** Given that the mole mass of CO₂ is roughly 44 g/mol, 10 mol of CO₂ is approximately 440 g! Mixing this amount of CO₂ with 1 kg of H₂O will most likely result in a chemical equilibrium state in which the aqueous phase is saturated with CO₂ and the remaining CO₂ exists as a gas (or super-critical fluid depending on the temperature and pressure). If we choose a small amount for CO₂, we can end up with an equilibrium state in which all CO₂ has been dissolved in the aqueous phase, and we will then not be able to determine its solubility.

We present below the Python script that performs a multi-phase, multi-species chemical equilibrium calculation using Reaktoro to determine the solubility of CO₂ in a 1 molal NaCl brine at temperature 60 °C and pressure 100 bar.
# Step 1: Import the reaktoro Python package
from reaktoro import *

# Step 2: Initialize a thermodynamic database
db = Database('supcrt98.xml')

# Step 3: Define the chemical system
editor = ChemicalEditor(db)
editor.addAqueousPhase('H2O Na Cl C')
editor.addGaseousPhase(['CO2(g)'])

# Step 4: Construct the chemical system
system = ChemicalSystem(editor)

# Step 5: Define the chemical equilibrium problem
problem = EquilibriumProblem(system)
problem.setTemperature(60, 'celsius')
problem.setPressure(100, 'bar')
problem.add('H2O', 1.0, 'kg')
problem.add('NaCl', 1.0, 'mol')
problem.add('CO2', 10.0, 'mol')

# Step 6: Calculate the chemical equilibrium state
state = equilibrate(problem)

# Step 7: Output the calculated chemical state to a file
state.output('result.txt')

# Step 8: Print the amounts of some aqueous species
print('Amount of CO2(aq):', state.speciesAmount('CO2(aq)'))
print('Amount of HCO3-:', state.speciesAmount('HCO3-'))
print('Amount of CO3--:', state.speciesAmount('CO3--'))

# Step 9: Print the amounts of element C in both aqueous and gaseous phases
print('Amount of C in aqueous phase:', state.elementAmountInPhase('C', 'Aqueous'))
print('Amount of C in gaseous phase:', state.elementAmountInPhase('C', 'Gaseous'))

You find next a step-by-step explanation of the above script.

## 3.1.1 Importing the reaktoro Python package

# Step 1: Import the reaktoro Python package
from reaktoro import *

Using Reaktoro in Python requires first an import of the python package reaktoro. From this point on, we are able to use the library components of Reaktoro (classes, methods, constants), which are needed to define our chemical system and chemical reaction modeling problems.

**Note:** To simplify the tutorials, we use from reaktoro import *, which imports all components of the reaktoro package into the default Python namespace, which can potentially create name conflicts. For your applications, consider instead using import reaktoro as rkt, and then refer to Reaktoro's classes and methods as rkt.Database, rkt.ChemicalSystem, rkt.equilibrate, and so forth.
3.1.2 Initializing a thermodynamic database

Thermodynamic databases are essential for modeling chemically reactive systems using Reaktoro. We need a database from where we collect data of substances that will compose our phases of interest in a multi-phase chemical system. A thermodynamic database also contains model parameters for the evaluation of standard thermodynamic properties of species and/or reactions (e.g., standard Gibbs energies, equilibrium constants).

In this step:

```
# Step 2: Initialize a thermodynamic database
db = Database('supcrt98.xml')
```

we initialize a `Database` object with the `supcrt98.xml` database file. This database was generated from the original SUPCRT92 database file `slop98.dat`. You are welcome to inspect these files and learn more about the chemical species available in them. You can also read more about the available thermodynamic databases supported in Reaktoro at [Thermodynamic Databases](#).

**Note:** All SUPCRT92 thermodynamic databases have been embedded into Reaktoro. Thus, you don’t actually need to have a database file named `supcrt98.xml` in a local directory when initializing the `Database` object. If you want to use a customized database file, however, also named `supcrt98.xml`, then your local file will be used instead.

**Tip:** If you are using a customized version of a thermodynamic database, consider changing its name (e.g., `custom-supcrt98.xml`) to avoid an accidental use of an embedded database. This can happen if you do not give a correct path to your custom database file.

3.1.3 Defining the chemical system

Reaktoro is a general-purpose chemical solver that avoids as much as possible presuming specific assumptions about your problems. Thus, you need to specify how your chemical system should be defined. This encompass the specification of all phases in the system as well as the chemical species that compose each phase. By using the `ChemicalEditor` class, you can conveniently achieve this as shown below:

```
# Step 3: Define the chemical system
editor = ChemicalEditor(db)
editor.addAqueousPhase('H O Na Cl C')
editor.addGaseousPhase(['CO2(g)'])
```

In this step, we create an object of class `ChemicalEditor` and specify that an aqueous phase and a gaseous phase should be considered in the chemical system. The aqueous phase is defined such that its species are all those aqueous species in the selected thermodynamic database that can be created by combining the chemical elements H, O, Na, Cl, and C. The gaseous phase is defined with only one gaseous species: CO$_2$(g).

**Note:** The selected elements H, O, Na, Cl, and C in the definition of the aqueous phase and the choice of CO$_2$(g) as the single gaseous species composing the gaseous phase consistently represent our intentions of calculating the solubility of CO$_2$ in a NaCl brine. If we decide to do a similar computation, but with a NaCl-MgCl$_2$-CaCl$_2$ brine, then that original list of elements would need to be incremented with Mg and Ca.

**Note:** An automatic search for chemical species can result in a large number of species in the phase, potentially causing the chemical reaction calculations to be more computationally expensive. If you are using Reaktoro in demanding applications (e.g., as a chemical solver in a reactive transport simulator), you might want to manually specify

---

3.1. Solubility of CO$_2$ in NaCl brines
the chemical species of each phase in your chemical system. This can be achieved by providing a list of species names as shown below:

```python
editor.addAqueousPhase([ 'H2O(l)', 'H+', 'OH-', 'Na+', 'Cl-', 'HCO3-', 'CO3--', 'CO2(aq)' ])
```

This is exactly what we did for the definition of the gaseous phase. If we had done instead:

```python
editor.addGaseousPhase('CO')
```

then other gases would be considered, such as CO(g) and O2(g), which are not of interest in our modeling problem.

**Caution:** If you manually specify the chemical species in a phase, you need to make sure that they exist in the thermodynamic database with exact same name! Replacing 'CO2(g)' above with 'CO2' will cause an error if the database has no gaseous species with such name.

### 3.1.4 Constructing the chemical system

```python
# Step 4: Construct the chemical system
system = ChemicalSystem(editor)
```

This step is where we create an object of class `ChemicalSystem` using the chemical system definition details stored in the object `editor`.

**Note:** `ChemicalSystem` is perhaps the main class in Reaktoro. An object of this class stores the phases, species and elements in our defined chemical system, as well as provides the means to compute many types of thermodynamic properties, such as standard thermodynamic properties (e.g., standard Gibbs energies, standard enthalpies, and standard volumes of species), and thermo-chemical properties (e.g., activity and activity coefficients of species; density, enthalpy and internal energy of phases). As you learn more about other Reaktoro’s classes, you will note that an object of class `ChemicalSystem` is almost always needed for their initialization!

### 3.1.5 Defining the chemical equilibrium problem

We have now defined and constructed our chemical system of interest, enabling us to move on to the next step in Reaktoro’s modeling workflow: defining our chemical reaction problems. Below we create an equilibrium problem with our prescribed equilibrium conditions for temperature, pressure, and amounts of elements that are consistent with our intention of calculating the solubility of CO2 at 60 °C and 100 bar in a 1 molal NaCl brine.

```python
# Step 5: Define the chemical equilibrium problem
problem = EquilibriumProblem(system)
problem.setTemperature(60, 'celsius')
```

(continues on next page)
Reaktoro Documentation

(continued from previous page)

```cpp
problem.setPressure(100, 'bar')
problem.add('H2O', 1.0, 'kg')
problem.add('NaCl', 1.0, 'mol')
problem.add('CO2', 10.0, 'mol')
```

**Note:** Did you pay attention we said prescribed equilibrium conditions for the *amounts of elements*? Since we actually provided the amounts of substances H$_2$O, NaCl, and CO$_2$ in the above code, this statement seems a little bit confusing at least. Here is what happens behind the scenes: Reaktoro parses these chemical formulas and determines the elements and their coefficients. Once this is done, the amount of each element stored inside the object `problem` is incremented according to the given amount of substance and its coefficient in the formula. The amounts of elements you provide are then used as constraints for the Gibbs energy minimization calculation when computing the state of chemical equilibrium (i.e., when we try to find the amounts of all species in the system that correspond to a state of minimum Gibbs energy and at the same time satisfying the *element amounts constraints*).

**Danger:** Now that you know that an equivalent chemical equilibrium problem could be defined with:

```cpp
problem.add('H' 111.0, 'mol')
problem.add('O' 75.5, 'mol')
problem.add('Na' 1.0, 'mol')
problem.add('Cl' 1.0, 'mol')
problem.add('C' 10.0, 'mol')
```

assuming that 1 kg of H$_2$O is roughly 55.5 mol, you might want to *adventure* in manually specifying different values for the amounts of elements. Just be extra careful with the values you provide as this could accidentally result in an *infeasible* chemical equilibrium state.

Here is a simple example of element amount conditions that result in an infeasible equilibrium state. Consider a chemical system containing only a gaseous phase with gases H$_2$O(g) and CO$_2$(g). Find non-negative amounts for H$_2$O(g) and CO$_2$(g) when the given amounts of elements are: 2 mol of H, 1 mol of O, and 1 mol of C.

**Tip:** The substance formulas given in the `EquilibriumProblem.add` method can, but do not need to, correspond to names of chemical species in the thermodynamic database. Even unusual, if not strange, substance formulas, such as HCl$_3$(NaO)$_4$C$_13$, would be understood by that method. *We do not promise, however, that you will obtain a feasible chemical equilibrium state with unrealistic conditions!*

**Note:** In Reaktoro, the word *element* is used as a synonym of components of chemical species, and not necessarily chemical elements. Electric charge is, for example, an element, even though it is not a not a chemical element. Thus, we say the ionic species CO$_3^{2-}$ is composed of elements C, O, and Z, with coefficients 1, 3, and -2 respectively, where Z is the symbol we use to denote the electric charge element.

3.1. Solubility of CO$_2$ in NaCl brines
**Warning:** Prefer the use of neutral substances when using the method `EquilibriumProblem.add`, unless you definitely need to add a charged, ionic species in the recipe. The following code:

```cpp
problem.add('H+' 0.1, 'mmol')
```

will increment not only the amount of element H by 0.1 mmol, but also the electric charge element Z. As a result, the composition of the aqueous phase at equilibrium will not be electrically neutral, which might not be your intention. The following:

```cpp
problem.add('H+' 0.1, 'mmol')
problem.add('Cl-' 0.1, 'mmol')
```

would result in the amount of element Z equal to zero.

### 3.1.6 Calculating the chemical equilibrium state

```cpp
# Step 6: Calculate the chemical equilibrium state
state = equilibrate(problem)
```

In this step, we use the `equilibrate` function to calculate the chemical equilibrium state of the system with the given equilibrium conditions stored in the object `problem`. For this calculation, Reaktoro uses an efficient **Gibbs energy minimization** computation to determine the species amounts that correspond to a state of minimum Gibbs energy in the system, while satisfying the prescribed amount conditions for temperature, pressure, and element amounts. The result is stored in the object `state`, of class `ChemicalState`.

**Attention:** In the vast majority of cases, you’ll only have one object of class `ChemicalSystem` in your code and one or more objects of class `ChemicalState` describing different states of your chemical system! Reaktoro differentiates these two independent concepts: chemical system definition and chemical system state.

**Tip:** The method `equilibrate` is a convenient function in Reaktoro. Consider using the class `EquilibriumSolver` for more advanced requirements. For example, if you have to perform many equilibrium calculations in sequence. The `equilibrate` method has a computational overhead because every call creates a new object of class `EquilibriumSolver`. Preferably, this object should be created only once, and then used subsequently for all other equilibrium calculations. Here is a demonstration:

```cpp
solver = EquilibriumSolver(system)  # Our chemical equilibrium solver
state = ChemicalState(system)  # Our chemical state
solver.solve(state, problem)  # Initial equilibrium calculation
state.output('state-initial.txt')  # Output the initial equilibrium state
problem.add('NaCl', 0.1, 'mol')  # Increment the amount of NaCl
solver.solve(state, problem)  # Subsequent equilibrium calculation
state.output('state-modified.txt')  # Output the modified equilibrium state
```

### 3.1.7 Outputting the calculated chemical state to a file

We have performed our chemical equilibrium calculation and now we want to inspect the computed compositional state and its thermodynamic properties, which can be done by outputting the chemical state to a file, here named
result.txt.

```python
# Step 7: Output the calculated chemical state to a file
state.output('result.txt')
```

### 3.1.8 Printing the amounts of some aqueous species

Here is just a small demonstration of getting species amount information from a `ChemicalState` object using the method `ChemicalState.speciesAmount` to extract the amount of a few chemical species. Please inspect the API of `ChemicalState` class to learn more about its methods.

```python
# Step 8: Print the amounts of some aqueous species
print('Amount of CO2(aq):', state.speciesAmount('CO2(aq)'))
print('Amount of HCO3-:', state.speciesAmount('HCO3-'))
print('Amount of CO3--:', state.speciesAmount('CO3--'))
```

### 3.1.9 Printing the amounts of element C in both aqueous and gaseous phases

Finally, we print the amounts of element C in both aqueous and gaseous phases:

```python
# Step 9: Print the amounts of element C in both aqueous and gaseous phases
print('Amount of C in aqueous phase:', state.elementAmountInPhase('C', 'Aqueous'))
print('Amount of C in gaseous phase:', state.elementAmountInPhase('C', 'Gaseous'))
```

In this specific case in which there were no initial element C in the aqueous phase, the value corresponding to the amount of element C in the aqueous phase is our solubility of CO2 in the NaCl brine with the previously prescribed conditions.

**Tip:** If we had used 2 kg of H$_2$O, we would have needed to divide the calculated amount of element C in the aqueous phase by 2 to obtain the solubility in molal (mol per kg of H$_2$O).

---

### 3.1. Solubility of CO$_2$ in NaCl brines
Chemical equilibrium calculations alone are sometimes not sufficient to understand a chemically reactive process. This happens when we need to understand how the composition of the chemical system changes with time as a result of chemical reactions. For this, chemical kinetics is imperative.

Reaktoro can perform chemical kinetics calculations combined with chemical equilibrium (i.e., part of the chemical system evolves under kinetics, while the other is continuously in equilibrium at all times). This mode of calculation is particularly useful for simulating chemically reactive systems in which some reactions have rates that are many orders of magnitude higher than others (and thus can be assumed in instantaneous equilibrium at any time).

Geochemical systems involving aqueous species and minerals are examples in which this combined chemical kinetics and equilibrium approach is valuable.
Chemical equilibrium and kinetics calculations are both space independent. If you need to model transport processes (e.g., advection, diffusion) combined with chemical reactive processes, then chemical transport (or reactive transport) simulations are what you need.
Thermodynamic databases allow us to define and model a chemically reactive system by providing the means for the computation of necessary thermodynamic properties (e.g., standard Gibbs energies of species, equilibrium constants of reactions). In such databases, we find a collection of chemical species and/or reactions and their accompanying data, which include substance’s name and chemical formula, reaction equations, thermodynamic data and/or model parameters.

**Attention:** There are many thermodynamic databases available in the literature and they are in general very different from each other. At the moment, there is no standard on how thermodynamic databases should be formatted. Some databases are based on chemical reactions and their equilibrium constants (e.g., PHREEQC databases), while others are based on substances and their model parameters for evaluation of their standard thermodynamic properties at temperature and pressure of interest (e.g., SUPCRT92 databases).

Reaktoro currently supports the following thermodynamic databases:

- SUPCRT92 databases;
- PHREEQC databases; and
- GEMS databases.

### 6.1 SUPCRT92 Databases

**SUPCRT92 Database Files**

- supcr98.xml
- supcrt98-organics.xml
- supcrt07.xml
The SUPCRT92 thermodynamic databases supported in Reaktoro are presented next. They contain parameters for the calculation of standard thermodynamic properties of aqueous species, gases, and minerals for temperatures 0-1000 °C and pressures 1-5000 bar. The standard properties of aqueous species are calculated using the revised Helgeson-Kirkham-Flowers (HKF) equations of state and, for the gases and minerals, a thermodynamic model based on Maier–Kelly heat capacity polynomial equation.

**Note:** The thermodynamic databases supcrt98.xml and supcrt07.xml, in XML format, were derived, respectively, from the original SUPCRT92 database files slop98.dat and slop07.dat. In the process, **all organic aqueous species were removed**! If you need them in your modeling problem, you should then use instead supcrt98-organics.xml and supcrt07-organics.xml.

**Tip:** If your problem requires an aqueous phase without organic species and you are using an automatic initialization scheme for its construction (e.g., creating an aqueous phase with all species in the database whose elements are H, O, C, or Ca), then make sure you are using one of the SUPCRT92 databases **without organic species**! Otherwise you might end up with an aqueous phase containing an extremely long list of organic species that will only serve to decrease the performance of the calculations.

### 6.2 PHREEQC Databases

Reaktoro can use PHREEQC as a **thermodynamic backend**, which permits us to take advantage of the rich collection of PHREEQC thermodynamic databases that are listed next.

**PHREEQC Database Files**

Amm.dat  
frezchem.dat  
iso.dat  
llnl.dat  
minteq.dat  
minteq.v4.dat  
phreeqc.dat  
pitzer.dat  
sit.dat  
wateq4f.dat

### 6.3 GEMS Databases

Reaktoro can also use GEMS as a **thermodynamic backend** and take advantage of its databases.
Todo: Write about GEMS databases.
There are excellent and mature open-source software for modeling chemical systems for which development and maintenance has been ongoing for several years to a few decades. Wouldn’t it be great if Reaktoro could use the best these software can offer in terms of thermodynamic databases and activity models as thermodynamic backends, while relying on Reaktoro’s numerical algorithms for the intense chemical reaction calculations? Well, it turns out this is already supported for two widely used geochemical modeling codes: PHREEQC and GEMS.

Fig. 1: PHREEQC is a computer program for speciation, batch-reaction, one-dimensional transport, and inverse geochemical calculations developed by USGS, United States.

Note: Neither PHREEQC nor GEMS are used in Reaktoro for solving the underlying mathematical problems for chemical equilibrium and kinetics. These backends act as providers of thermodynamic properties of species, phases, and reactions (e.g., activity coefficients, activities, standard chemical potentials, phase molar volume and enthalpy, equilibrium constant of reaction). Whenever Reaktoro’s numerical algorithms need these properties, the thermodynamic backend is invoked to retrieve them in a very efficient way by directly using its API. It seems complicated, but rest assure, the usage is rather simple as you’ll see below!

### 7.1 PHREEQC Backend

The code below demonstrates the combined use of Reaktoro and PHREEQC to perform a chemical equilibrium calculation in which PHREEQC thermodynamic data and activity models are used together with Reaktoro’s Gibbs energy minimization algorithm.

```python
from reaktoro import *

# This string defines a PHREEQC script problem.
# This problem was taken from the official PHREEQC example named ex1.
ex1 = r'''(continues on next page)'''
```

(continues on next page)
Fig. 2: GEMS is a Gibbs energy minimization software for geochemical modeling developed at Paul Scherrer Institute, Switzerland.

```
TITLE Example 1.--Add uranium and speciate seawater.
SOLUTION 1 SEAWATER FROM NORDSTROM AND OTHERS (1979)
  units ppm
  pH  8.22
  pe  8.451
  density 1.023
  temp 25.0
  redox O(0)/O(-2)
  Ca  412.3
  Mg 1291.8
  Na 10768.0
  K  399.1
  Fe  0.002
  Mn 0.0002 pe
  Si  4.28
  Cl  19353.0
  Alkalinity 141.682 as HCO3
  S(6)  2712.0
  N(5) 0.29 gfw  62.0
  N(-3) 0.03 as NH4
  U  3.3 ppb N(5)/N(-3)
  O(0) 1.0 O2(g) -0.7
SOLUTION_MASTER_SPECIES
  U    U+4  0.0  238.0290  238.0290
  U(4) U+4  0.0  238.0290
  U(5) UO2+ 0.0  238.0290
  U(6) UO2+2 0.0  238.0290
SOLUTION_SPECIES
  #primary master species for U
  #is also secondary master species for U(4)
```
\[
\begin{align*}
U^+4 &= U^4 \\
\log_k &= 0.0 \\
U^+4 + 4 H_2O &= U(OH)^4 + 4 H^+ \\
\log_k &= -8.538 \\
delta_h &= 24.760 \text{ kcal} \\
U^+4 + 5 H_2O &= U(OH)^5^- + 5 H^+ \\
\log_k &= -13.147 \\
delta_h &= 27.580 \text{ kcal} \\
\text{#secondary master species for U(5)} \\
U^+4 + 2 H_2O &= UO_2^+ + 4 H^+ + e^- \\
\log_k &= -6.432 \\
delta_h &= 31.130 \text{ kcal} \\
\text{#secondary master species for U(6)} \\
U^+4 + 2 H_2O &= UO_2^+2 + 4 H^+ + 2 e^- \\
\log_k &= -9.217 \\
delta_h &= 34.430 \text{ kcal} \\
UO_2^+2 + H_2O &= UO_2OH^+ + H^+ \\
\log_k &= -5.782 \\
delta_h &= 11.015 \text{ kcal} \\
2UO_2^+2 + 2H_2O &= (UO_2)^2(OH)^2+2 + 2H^+ \\
\log_k &= -5.626 \\
delta_h &= -36.04 \text{ kcal} \\
3UO_2^+2 + 5H_2O &= (UO_2)^3(OH)^5+ + 5H^+ \\
\log_k &= -15.641 \\
delta_h &= -44.27 \text{ kcal} \\
UO_2^+2 + CO_3^-2 &= UO_2CO_3 \\
\log_k &= 10.064 \\
delta_h &= 0.84 \text{ kcal} \\
UO_2^+2 + 2CO_3^-2 &= UO_2(CO_3)^2^-2 \\
\log_k &= 16.977 \\
delta_h &= 3.48 \text{ kcal} \\
UO_2^+2 + 3CO_3^-2 &= UO_2(CO_3)^3^-4 \\
\log_k &= 21.397 \\
delta_h &= -8.78 \text{ kcal} \\
\text{PHASES} \\
\text{Uraninite} \\
UO_2 + 4 H^+ &= U^+4 + 2 H_2O \\
\log_k &= -3.490 \\
delta_h &= -18.630 \text{ kcal} \\
\text{END} \\
\end{align*}
\]

# Initialize a Phreeqc instance with the official phreeqc.dat database file
phreeqc = Phreeqc('../..../databases/phreeqc/phreeqc.dat')

# Execute a PHREEQC script defining a geochemical problem.
# Here this script is actually embedded into a string named `ex1`.
# However, `ex1` could also be a string containing the path to a script file.
# Method execute will automatically identify when the contents are embedded in
# the string and when the string is actually a path to a script file.
phreeqc.execute('ex1')

# Initialize a ChemicalSystem instance using the current state of the Phreeqc
# instance. This will allow the use of both PHREEQC thermodynamic data and
# PHREEQC activity models in the subsequent equilibrium calculations using
# Reaktoro's algorithms.
system = ChemicalSystem(phreeqc)
# Initialize an ChemicalState instance using the current state of the Phreeqc instance.
state = phreeqc.state(system)

# Output the equilibrium state calculated by PHREEQC to a file.
state.output('state-phreeqc.txt')

# Define an equilibrium problem in which the current state is mixed with 1 mmol of HCl.
problem = EquilibriumProblem(system)
problem.setTemperature(state.temperature())
problem.setPressure(state.pressure())
problem.add(state)
problem.add('HCl', 1.0, 'mmol')

# Set Hessian of Gibbs energy to an approximation, since PHREEQC does not compute molar derivatives of activities
options = EquilibriumOptions()
options.hessian = GibbsHessian.Approximation

# Calculate the new equilibrium state of the system. This will use both PHREEQC thermodynamic data and PHREEQC activity models.
state = ChemicalState(system)
equilibrate(state, problem, options)

# Print the new equilibrium state and check with pH is more acidic now.
state.output('state-phreeqc-updated.txt')

Python and C++ files for this demo:

demo-backends-phreeqc.py
demo-backends-phreeqc.cpp

## 7.2 GEMS Backend

Similarly, the code below briefly demonstrates how Reaktoro and GEMS can be used together. You’ll need first to prepare your chemical system definition using GEM-Selektor, the graphical user interface of GEMS. In this step, you’ll be able to select which GEMS’ supported thermodynamic database you want to use as well as the activity models for each phase (aqueous, gaseous, solid solutions). Next, export the GEMS project files to disk, and use it in Reaktoro as shown below.

```python
from reaktoro import *

# **Note:**
# This demo should be executed in the same directory where the script is located.
# Example:
# ```
# cd Reaktoro/demos/python
# python demo-equilibrium-gems.py
# ```
# Use an exported project file from GEMS to initialize a Gems object,
gems = Gems("../resources/gems/CalciteBC-dat.lst")
``` (continues on next page)
# and then use it to construct the ChemicalSystem object.
system = ChemicalSystem(gems)

# Create a ChemicalState object that contains the temperature, pressure,
# and amounts of species stored in the exported GEMS file.
state = gems.state(system)

# Output the equilibrium state calculated by GEMS to a file.
state.output("state-gems.txt")

# Perturb the equilibrium state calculated by GEMS
state.setSpeciesAmount("CO2", 0.1)

# and then equilibrate the modified chemical state using Reaktoro's methods.
equilibrate(state)

# Output the updated equilibrium state to a file.
state.output("state-gems-updated.txt")

Python and C++ files for this demo:

demo-backends-gems.py
demo-backends-gems.cpp

### 7.3 What about more thermodynamic backends?

Are there other chemical reaction modeling software that you think could be integrated with Reaktoro as thermodynamic backends? Let us know by creating a new issue at Reaktoro’s GitHub Issues.

**Attention:** It would be great if you could contribute to expanding the list of supported Reaktoro's thermodynamic backends. Contributions can be made in several forms, ranging from direct code contribution to financing a project in which one or more experts will implement this.
CHAPTER 8

Acknowledgements

Reaktoro would not exist without its past and current supporters and contributors. Therefore, we thank on this page everyone that is contributing or has contributed to the project with either financial, scientific, and/or coding contributions.

8.1 Academic and Research Institutions

Reaktoro is thankful to the following academic and research institutions for their substantial past or current financial support:

- Imperial College London, UK (2011-2014) Qatar Carbonates and Carbon Storage Research Centre (QCCSRC)
- Paul Scherrer Institute, Switzerland (2014-2015) Laboratory for Waste Management
- ETH Zurich, Switzerland (2015-present) Geothermal Energy and Geofluids Group

8.2 Industries

Reaktoro is proud of its past and current financial support from industries. We would like to thank:

- Shell (2011-2014, 2019-present) Shell Technology Centre Amsterdam
- Qatar Petroleum and Qatar Science & Technology Park (2011-2014) Qatar Carbonates and Carbon Storage Research Centre (QCCSRC)

Reaktoro is also especially thankful for software development contributions from the following companies:

- ESSS (2019-current) Engineering Simulation and Scientific Software
8.3 Government Institutions

Distributing Reaktoro as an open-source software requires public funding. The following is a list of research grants obtained from government institutions for further development and maintenance of Reaktoro:


8.4 Scientific Contributors

Reaktoro’s advanced numerical methods and modeling capabilities would not be possible without the numerous current and past scientific discussions with the following people:

- Martin Blunt, Prof. (Imperial College London, UK) (2011-2014)
- Tara LaForce, Dr. (CSIRO, Australia) (2011-2014)
- Dmitrii Kulik, Dr. (Paul Scherrer Institute, Switzerland) (2014-present)
- Georg Kosakowski, PD Dr. (Paul Scherrer Institute, Switzerland) (2014-present)
- Nikolaos Prasianakis, Dr. (Paul Scherrer Institute, Switzerland) (2014-present)
- Dan Miron, Dr. (Paul Scherrer Institute, Switzerland) (2016-present)
- Martin Saar, Prof. (ETH Zurich, Switzerland) (2015-present)
- William Smith, Prof. (University of Guelph, Canada) (2016-present)

8.5 Development Contributors

The development of Reaktoro is and has been helped by several individuals, with contributions that range from documentation and fixing bugs to implementation of modern software engineering practices into the project. We thank them below:

- Dan Miron, Dr. (Paul Scherrer Institute, Switzerland) (2016-present)
- Till Ehrengruber (ETH Zurich, Switzerland) (2016-present)
- Alex Hobé (ETH Zurich, Switzerland) (2016-2017)
- Edson Tadeu M. Manoel (ESSS, Brazil) (2018-present)
- Alexandre Barbosa (ESSS, Brazil) (2018-present)
- Bruno Oliveira (ESSS, Brazil) (2018-present)
- Armando A. S. C. Neto (ESSS, Brazil) (2018-present)
- Svetlana Matculevich, Dr. (ETH Zurich, Switzerland) (2018-present)

8.6 Have we forgotten you?

You might have already contributed to Reaktoro and your name (or organization’s name) is still not listed here. Please get in touch and we’ll be happy to correct this.
CHAPTER 9

License

LGPL v2.1
Copyright © 2014-2018 Allan Leal and Reaktoro contributors

Reaktoro is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

Reaktoro is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.