Thermosteam

Yoel Cortes-Pena

Aug 21, 2022

TUTORIAL

Thermosteam is a standalone thermodynamic engine capable of estimating mixture properties, solving thermodynamic phase equilibria, and modeling stoichiometric reactions. Thermosteam builds upon chemicals, the chemical properties component of the Chemical Engineering Design Library, with a robust and flexible framework that facilitates the creation of property packages. The Biorefinery Simulation and Techno-Economic Analysis Modules (BioSTEAM) is dependent on thermosteam for the simulation of unit operations.

CHAPTER

ONE

KEY FEATURES & CAPABILITIES

- **Simple** and straight foward estimation of mixture properties, thermodynamic phase equilibria, and chemical reactions with just a few lines of code.
- **Clear** representation of chemical and phase data within every object using IPython's rich display system.
- **Fast** estimation of thermodynamic equilibrium within hundreds of microseconds through the smart use of cache and Numba Jit compiled functions.
- **Flexible** implemention of thermodynamic models for estimating pure component properties in just a few lines of code.
- **Extendable** framework that allows easy integration of new methods for computing thermodynamic equilibrium coefficients and mixture properties.

1.1 Overview

Thermosteam is an extensive object oriented package for the estimation of thermodynamic equilibrium, mixture properties, and mass and energy balances. The Stream object is the main interface for performing these calculations. Before creating streams, a thermodynamic property package must be defined through a Thermo object, which compiles the working chemicals, mixing rules, and the equlibrium estimation methods. Each Chemical object contains model handles that manages the thermodynamic models and makes sure to use a valid model at given temperatures and pressures whenever estimating properties. The functional algorithms for estimating pure component properties are presented as functors which implicitly store and use fitted coefficients for the estimation of temperature and pressure dependent properties.

1.2 Installation

Get the latest version of Thermosteam from PyPI. If you have an installation of Python with pip, simple install it with:

```
$ pip install thermosteam
```

To get the git version and install it, run:

```
$ git clone --depth 100 git://github.com/BioSTEAMDevelopmentGroup/thermosteam
$ cd thermosteam
$ pip install .
```

We use the *depth* option to clone only the last 100 commits. Thermosteam has a long history, so cloning the whole repository (without using the depth option) may take over 30 min.

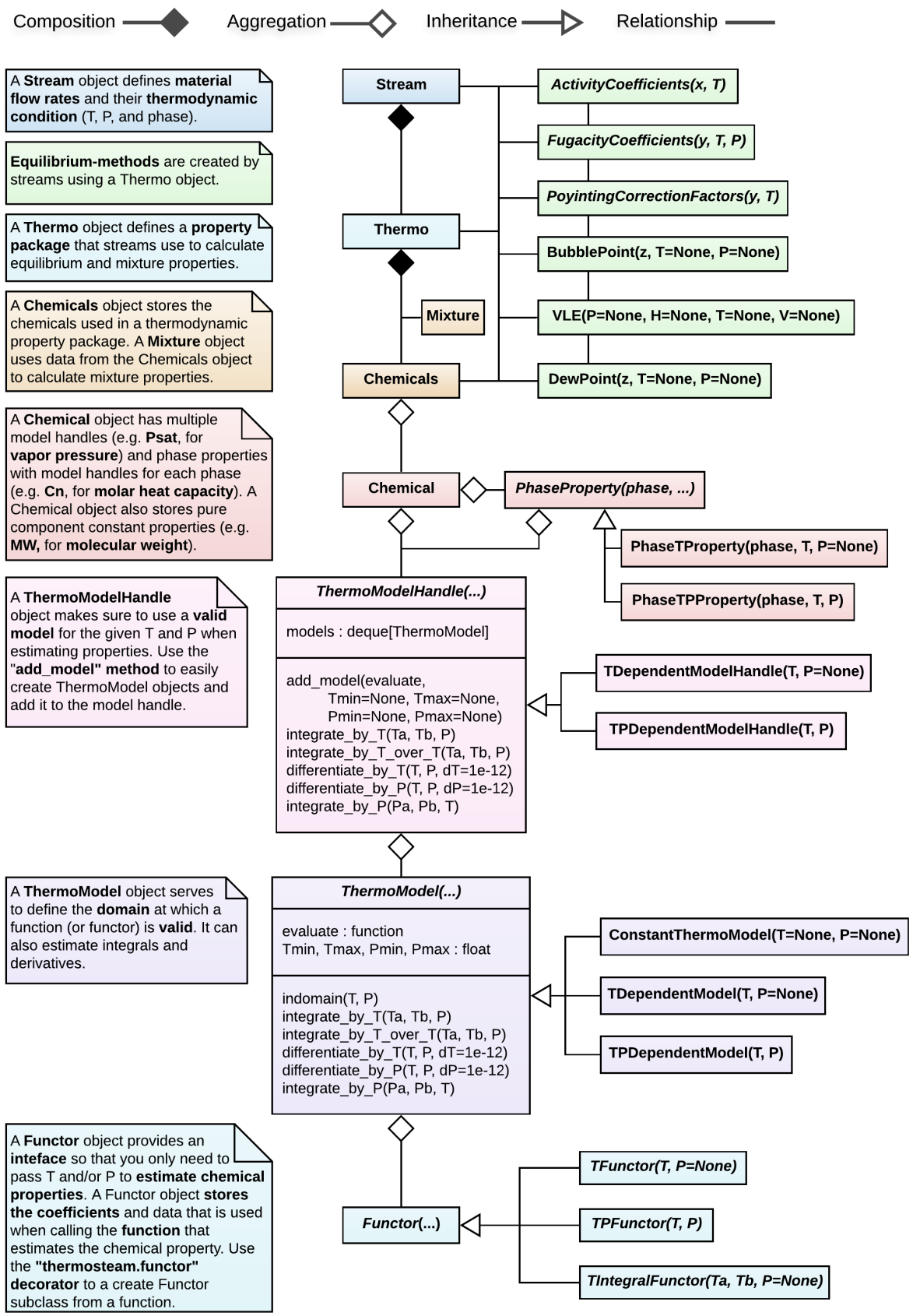


Fig. 1: A functor-oriented class diagram describes the relationship between the core classes of Thermosteam. Overall, the diagram follows the Universal Modeling Language (UML) guidelines except for a few modifications. In particular, the function signature of callable objects is presented after their name. For example: an ActivityCoefficients object returns an array of activity coefficients when called with the liquid molar composition (x) and temperature (T); and a TPFunctor object returns the value of a chemical property when called with the temperature (T), and pressure (P).

1.2.1 Common Issues

- **Cannot install/update Thermosteam:**

  If you are having trouble installing or updating Thermosteam, it may be due to dependency issues. You can bypass these using:

```
$ pip install --user --ignore-installed thermosteam
```

  You can make sure you install the right version by including the version number:

```
$ pip install thermosteam==<version>
```


{

“cells”: [

{

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“# An hour blitz to practical thermodynamics”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Pure component chemical models”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Thermosteam packages chemical and mixture thermodynamic models in a flexible framework that allows users to fully customize and extend the models, as well as create new models. Central to all thermodynamic algorithms is the [Chemical](../Chemical.txt) object, which contains constant chemical properties, as well as thermodynamic and transport properties as a function of temperature and pressure:”

]

}, {

“cell_type”: “code”, “execution_count”: 1, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“Chemical: Water (phase_ref=’l’)n”, “[Names] CAS: 7732-18-5n”, ” InChI: H2O/h1H2n”, ” InChI_key: XLYOFNOQVPJJNP-U…n”, ” common_name: watern”, ” iupac_name: (‘oxidane’,)n”, ” pubchemid: 962n”, ” smiles: On”, ” formula: H2On”, “[Groups] Dortmund: <1H2O>n”, ” UNIFAC: <1H2O>n”, ” PSRK: <1H2O>n”, “[Data] MW: 18.015 g/moln”, ” Tm: 273.15 Kn”, ” Tb: 373.12 Kn”, ” Tt: 273.15 Kn”, ” Tc: 647.14 Kn”, ” Pt: 610.88 Pan”, ” Pc: 2.2048e+07 Pan”, ” Vc: 5.6e-05 m^3/moln”, ” Hf: -2.8582e+05 J/moln”, ” S0: 70 J/K/moln”, ” LHV: 44011 J/moln”, ” HHV: 0 J/moln”, ” Hfus: 6010 J/moln”, ” Sfus: Nonen”, ” omega: 0.344n”, ” dipole: 1.85 Debyen”, ” similarity_variable: 0.16653n”, ” iscyclic_aliphatic: 0n”, ” combustion: {‘H2O’: 1.0}n”

```
            ]
        }
    ], “source”: [
        “import thermosteam as tmon”, “# Initialize chemical with an identifier (e.g. by name, CAS, InChI...)n”, “Water = tmo.Chemical(‘Water’) n”, “Water.show()”
    ]
}, {
    “cell_type”: “markdown”, “metadata”: {}, “source”: [
        “All fields can be easility accessed, for example:”
    ]
}, {
    “cell_type”: “code”, “execution_count”: 2, “metadata”: {}, “outputs”: [
        {
          “data”: {
            “text/plain”: [
              “‘7732-18-5’”
            ]
          }, “execution_count”: 2, “metadata”: {}, “output_type”: “execute_result”
        }
    ], “source”: [
        “# CAS numbern”, “Water.CAS”
    ]
}, {
    “cell_type”: “code”, “execution_count”: 3, “metadata”: {}, “outputs”: [
        {
            “data”: {
              “text/plain”: [
                “18.01528”
              ]
            }, “execution_count”: 3, “metadata”: {}, “output_type”: “execute_result”
        }
    ], “source”: [
        “# Molecular weight (g/mol)n”, “Water.MW”
    ]
}, {
    “cell_type”: “code”, “execution_count”: 4, “metadata”: {}, “outputs”: [
        {
```

“data”: {

“text/plain”: [
“373.124”

]

}, “execution_count”: 4, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Boiling point (K)n”, “Water.Tb”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Temperature (in Kelvin) and pressure (in Pascal) dependent properties can be computed:”

]

}, {

“cell_type”: “code”, “execution_count”: 5, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“101284.55179999319”

]

}, “execution_count”: 5, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Vapor pressure (Pa)n”, “Water.Psat(T=373.15)”

]

}, {

“cell_type”: “code”, “execution_count”: 6, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“0.07197220523022964”

]

}, “execution_count”: 6, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Surface tension (N/m)n”, “Water.sigma(T=298.15)”

```
        ]
    }, {
        “cell_type”: “code”, “execution_count”: 7, “metadata”: {}, “outputs”: [
            {
                “data”: {
                    “text/plain”: [
                        “1.8069204487889095e-05”
                    ]
                }, “execution_count”: 7, “metadata”: {}, “output_type”: “execute_result”
            }
        ], “source”: [
            “# Liquid molar volume (m^3/mol)n”, “Water.V(phase=’l’, T=298.15, P=101325)”
        ]
    }, {
        “cell_type”: “code”, “execution_count”: 8, “metadata”: {}, “outputs”: [
            {
                “data”: {
                    “text/plain”: [
                        “0.023505774739491968”
                    ]
                }, “execution_count”: 8, “metadata”: {}, “output_type”: “execute_result”
            }
        ], “source”: [
            “# Vapor molar volume (m^3/mol)n”, “Water.V(phase=’g’, T=298.15, P=101325)”
        ]
    }, {
        “cell_type”: “markdown”, “metadata”: {}, “source”: [
            “Temperature dependent properties are managed by indexable model handles, which contain many models ordered in decreasing priority:”
        ]
    }, {
        “cell_type”: “code”, “execution_count”: 9, “metadata”: {}, “outputs”: [
            {
                “name”: “stdout”, “output_type”: “stream”, “text”: [
                    “TDependentModelHandle(T, P=None) -> Psat [Pa]n”, “[0] Wagner originaln”, “[1] Antoinen”, “[2] EQ101n”, “[3] Wagnern”, “[4] boiling critical relationn”, “[5] Lee Keslern”, “[6] Ambrose Waltonn”, “[7] Sanjarin”, “[8] Edalatn”
```

```
            ]
        }
    ], "source": [
        "Water.Psat.show()"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Each model is applicable to a certain domain, as given by their *Tmin* and *Tmax*:"
    ]
}, {
    "cell_type": "code", "execution_count": 10, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "TDependentModel(T, P=None) -> Psat [Pa]n", " name: Wagner originaln",
                " Tmin: 275 Kn", " Tmax: 647.35 Kn"
            ]
        }
    ], "source": [
        "Wagner_origial = Water.Psat[0]n", "Wagner_origial.show()"
    ]
}, {
    "cell_type": "code", "execution_count": 11, "metadata": {}, "outputs": [
        {
            "data": {
                "text/plain": [
                    "(647.35, 275.0)"
                ]
            }, "execution_count": 11, "metadata": {}, "output_type": "execute_result"
        }
    ], "source": [
        "# Note that these attributes can be get/set toon", "Wagner_origial.Tmax, Wagner_origial.Tmin"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "When called, the model handle searches through each model until it finds one with an applicable domain. If none are applicable, a domain error is raised:"
    ]
```

}, {

“cell_type”: “code”, “execution_count”: 12, “metadata”: {}, “outputs”: [

{

“ename”: “DomainError”, “evalue”: “Water (CAS: 7732-18-5) has no valid saturated vapor pressure model at T=1000.00 K”, “output_type”: “error”, “traceback”: [

“u001b[1;31m———————————————————————————————————————u001b[0m”, “u001b[1;31mDomainErroru001b[0m Traceback (most recent call last)”, “u001b[1;32m<ipython-input-12-5818a3190dca>u001b[0m in u001b[0;36m<module>u001b[1;34mu001b[0mnu001b[1;32m—–> 1u001b[1;33m u001b[0mWateru001b[0mu001b[1;33m.u001b[0mu001b[0mPsatu001b[0mu001b[1;33m(u0

“u001b[1;32m~\OneDrive\Code\thermosteam\thermosteam\base\thermo_model_handle.pyu001b[0m in u001b[0;36m__call__u001b[1;34m(self, T, P)u001b[0mnu001b[0;32m 284u001b[0m u001b[1;32mifu001b[0m u001b[0mmodelu001b[0mu001b[1;33m.u001b[0mu001b[0mindomainu001b[0mu001b[1;33m(u001b[0mu

u001b[1;32mreturnu001b[0m u001b[0mmodelu001b[0mu001b[1;33m.u001b[0mu001b[0mevaluateu001b[

285u001b[0m raise DomainError(f"{no_valid_model(self._chemical, self._var)} "nu001b[1;32m–> 286u001b[1;33m f"at T={T:.2f} K", chemical=self._chemical)nu001b[0mu001b[0;32m 287u001b[0m u001b[1;33mu001b[0mu001b[0mnu001b[0;32m 288u001b[0m u001b[0mat_Tu001b[0m u001b[1;33m=u001b[0m u001b[0m__call__u001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0mn”, “u001b[1;31mDomainErroru001b[0m: Water (CAS: 7732-18-5) has no valid saturated vapor pressure model at T=1000.00 K”

]

}

], “source”: [

“Water.Psat(1000.0)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Model handles as well as the models themselves have tabulation and plotting methods to help visualize how properties depend on temperature and pressure.”

]

}, {

“cell_type”: “code”, “execution_count”: 13, “metadata”: {}, “outputs”: [

{

“data”: {

“image/png”: “iVBORw0KGgoAAAANSUhEUgAAAYYAAAEKCAYAAAAW8vJGAAAABHNCSVQI

“text/plain”: [

“<Figure size 432x288 with 1 Axes>”

]

}, “metadata”: {

“needs_background”: “light”

}, “output_type”: “display_data”

}

], “source”: [

“import matplotlib.pyplot as pltn”, “Water.Psat.plot_vs_T([Water.Tm, Water.Tb], ‘degC’, ‘atm’, label="Water")n”, “plt.show()”

]

}, {

“cell_type”: “code”, “execution_count”: 14, “metadata”: {}, “outputs”: [

{

“data”: {

“image/png”: “iVBORw0KGgoAAAANSUhEUgAAAYYAAAEKCAYAAAAW8vJGAAAABHNCSVQI

“text/plain”: [

“<Figure size 432x288 with 1 Axes>”

]

}, “metadata”: {

“needs_background”: “light”

}, “output_type”: “display_data”

}

], “source”: [

“# Plot all modelsn”, “Water.Psat.plot_models_vs_T([Water.Tm, Water.Tb], ‘degC’, ‘atm’)n”, “plt.show()”

]

}, {

“cell_type”: “code”, “execution_count”: 15, “metadata”: {}, “outputs”: [

{

“data”: {

“image/png”: “iVBORw0KGgoAAAANSUhEUgAAAYkAAAEKCAYAAAADn+anLAAAABHNCSVQIC

“text/plain”: [

“<Figure size 432x288 with 1 Axes>”

]

}, “metadata”: {

“needs_background”: “light”

}, “output_type”: “display_data”

}

], “source”: [

“# Plot only the ‘Wagner original model’n”, “Water.Psat[0].plot_vs_T(T_units=’degC’, units=’atm’) # Bounds are the model’s Tmin and Tmaxn”, “plt.show()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Manage the model order with the *set_model_priority* and *move_up_model_priority* methods:”

]

}, {

“cell_type”: “code”, “execution_count”: 16, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“TDependentModel(T, P=None) -> Psat [Pa]n”, ” name: Antoinen”, ” Tmin: 273.2 Kn”, ” Tmax: 473.2 Kn”

]

}

], “source”: [

“# Note: In this case, we pass the model name, but itsn”, “# also possible to pass the current index, or the model itself.n”, “Water.Psat.move_up_model_priority(‘Antoine’)n”, “Water.Psat[0].show() # Notice how Antoine is now in the top priority”

]

}, {

“cell_type”: “code”, “execution_count”: 17, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“TDependentModel(T, P=None) -> Psat [Pa]n”, ” name: Wagner originaln”, ” Tmin: 275 Kn”, ” Tmax: 647.35 Kn”

]

}

], “source”: [

“Water.Psat.set_model_priority(‘Wagner original’)n”, “Water.Psat[0].show() # Notice how Wagner original is back on top priority”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“When setting a model priority, the default priority is *0* (or top priority), but you can choose any priority:”

]

}, {

“cell_type”: “code”, “execution_count”: 18, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

```
            "TDependentModel(T, P=None) -> Psat [Pa]n", " name: Antoinen", " Tmin: 273.2 Kn", " Tmax: 473.2 Kn"
           ]
          }
         ], "source": [
          "Water.Psat.set_model_priority('Antoine', 2)n", "Water.Psat[2].show() # Moved Antoine to priority #2"
         ]
        }, {
         "cell_type": "markdown", "metadata": {}, "source": [
          "Thermodynamic properties dependent on the phase are handled by phase properties:"
         ]
        }, {
         "cell_type": "code", "execution_count": 19, "metadata": {}, "outputs": [
          {
           "name": "stdout", "output_type": "stream", "text": [
            "PhaseTPHandle(phase, T, P) -> V [m^3/mol]n"
           ]
          }
         ], "source": [
          "Water.V.show()"
         ]
        }, {
         "cell_type": "markdown", "metadata": {}, "source": [
          "Phase properties contain model handles as attributes:"
         ]
        }, {
         "cell_type": "code", "execution_count": 20, "metadata": {}, "outputs": [
          {
           "name": "stdout", "output_type": "stream", "text": [
            "TPDependentModelHandle(T, P) -> V.l [m^3/mol]n", "[0] volume VDI PPDSn", "[1] Campbell Thodosn", "[2] Yen Woods saturationn", "[3] Rackettn", "[4] Yamada Gunnn", "[5] Bhirud normaln", "[6] Townsend Halesn", "[7] CRC inorganic liquid constantn", "[8] Rackettn", "[9] COSTALDn", "[10] COSTALD compressedn"
           ]
          }
         ], "source": [
          "Water.V.l.show()"
```

```
        ]
    }, {
        “cell_type”: “code”, “execution_count”: 21, “metadata”: {}, “outputs”: [
            {
                “name”: “stdout”, “output_type”: “stream”, “text”: [
                    “TPDependentModelHandle(T, P) -> V.g [m^3/mol]n”, “[0] Tsonopoulos extendedn”, “[1] Tsonopoulosn”, “[2] Abbottn”, “[3] Pitzer Curln”, “[4] CRCVirialn”, “[5] ideal gasn”
                ]
            }
        ], “source”: [
            “Water.V.g.show()”
        ]
    }, {
        “cell_type”: “markdown”, “metadata”: {}, “source”: [
            “A new model can be added easily to a model handle through the *add_model* method, for example:”
        ]
    }, {
        “cell_type”: “code”, “execution_count”: 22, “metadata”: {}, “outputs”: [
            {
                “name”: “stdout”, “output_type”: “stream”, “text”: [
                    “TDependentModel(T) -> Psat [Pa]n”, ” name: User antoine modeln”, ” Tmin: 273.2 Kn”, ” Tmax: 473.2 Kn”
                ]
            }
        ], “source”: [
            “# Set top_priority=True to place model in postion [0]n”, “@Water.Psat.add_model(Tmin=273.20, Tmax=473.20, top_priority=True)n”, “def User_antoine_model(T):n”, ” return 10.0**(10.116 - 1687.537 / (T - 42.98))n”, “Water.Psat[0].show()”
        ]
    }, {
        “cell_type”: “markdown”, “metadata”: {}, “source”: [
            “The *add_model* method is a high level interface that even lets you create a constant model:”
        ]
    }, {
        “cell_type”: “code”, “execution_count”: 23, “metadata”: {}, “outputs”: [
```

```
            {
                "name": "stdout", "output_type": "stream", "text": [
                    "ConstantThermoModel(T=None, P=None) -> V.l [m^3/mol]n", " name: User constantn", " value: 1.687e-05n", " Tmin: 0 Kn", " Tmax: inf Kn", " Pmin: 0 Pan", " Pmax: inf Pan"
                ]
            }
        ], "source": [
            "Water.V.l.add_model(1.687e-05, name='User constant')n", "# Model is appended at the end by defaultn", "Water.V.l[-1].show()"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "Lastly, all default models in thermosteam have functors (i.e. functions with adjustable parameters):"
        ]
    }, {
        "cell_type": "code", "execution_count": 24, "metadata": {}, "outputs": [
            {
                "name": "stdout", "output_type": "stream", "text": [
                    "Functor: Wagner_original(T, P=None) -> Psat [Pa]n", " Tc: 647.35 Kn", " Pc: 2.2122e+07 Pan", " a: -7.7645n", " b: 1.4584n", " c: -2.7758n", " d: -1.233n"
                ]
            }
        ], "source": [
            "# The saturated vapor pressure model from beforen", "Wagner_origial.evaluate.show()"
        ]
    }, {
        "cell_type": "code", "execution_count": 25, "metadata": {}, "outputs": [
            {
                "name": "stdout", "output_type": "stream", "text": [
                    "Functor: Wagner_original(T, P=None) -> Psat [Pa]n", " Tc: 647.35 Kn", " Pc: 2.2064e+07 Pan", " a: -7.7645n", " b: 1.4584n", " c: -2.7758n", " d: -1.233n"
                ]
            }
        ], "source": [
            "Wagner_origial.evaluate.Pc = 22.064e6n", "Wagner_origial.evaluate.show()"
```

```
        ]
    }, {
        “cell_type”: “markdown”, “metadata”: {}, “source”: [
            “## Managing chemical sets”
        ]
    }, {
        “cell_type”: “markdown”, “metadata”: {}, “source”: [
            “Define multiple chemicals as a [Chemicals](../Chemicals.txt) object:”
        ]
    }, {
        “cell_type”: “code”, “execution_count”: 26, “metadata”: {}, “outputs”: [
            {
                “name”: “stdout”, “output_type”: “stream”, “text”: [
                    “Chemicals([Water, Ethanol])n”
                ]
            }
        ], “source”: [
            “chemicals = tmo.Chemicals([‘Water’, ‘Ethanol’])n”, “chemicals”
        ]
    }, {
        “cell_type”: “markdown”, “metadata”: {}, “source”: [
            “The chemicals are attributes:”
        ]
    }, {
        “cell_type”: “code”, “execution_count”: 27, “metadata”: {}, “outputs”: [
            {
                “data”: {
                    “text/plain”: [
                        “(Chemical(‘Water’), Chemical(‘Ethanol’))”
                    ]
                }, “execution_count”: 27, “metadata”: {}, “output_type”: “execute_result”
            }
        ], “source”: [
            “(chemicals.Water, chemicals.Ethanol)”
        ]
    }, {
        “cell_type”: “markdown”, “metadata”: {}, “source”: [
```

```
            "Chemicals are indexable:"
        ]
    }, {
        "cell_type": "code", "execution_count": 28, "metadata": {}, "outputs": [
            {
                "name": "stdout", "output_type": "stream", "text": [
                    "Chemical('Water')n"
                ]
            }
        ], "source": [
            "Water = chemicals['Water']n", "print(repr(Water))"
        ]
    }, {
        "cell_type": "code", "execution_count": 29, "metadata": {}, "outputs": [
            {
                "data": {
                    "text/plain": [
                        "[Chemical('Ethanol'), Chemical('Water')]"
                    ]
                }, "execution_count": 29, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "chemicals['Ethanol', 'Water']"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "Chemicals are also iterable:"
        ]
    }, {
        "cell_type": "code", "execution_count": 30, "metadata": {}, "outputs": [
            {
                "name": "stdout", "output_type": "stream", "text": [
                    "Chemical('Water')n", "Chemical('Ethanol')n"
                ]
            }
        ], "source": [
            "for chemical in chemicals:n", " print(repr(chemical))"
```

```
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "More chemicals can also be appended:"
    ]
}, {
    "cell_type": "code", "execution_count": 31, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "Chemicals([Water, Ethanol, Propanol])n"
            ]
        }
    ], "source": [
        "Propanol = tmo.Chemical('Propanol')n", "chemicals.append(Propanol)n", "chemicals"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "The main benefit of using a Chemicals object, is that they can be compiled and used as part of a thermodynamic property package, as defined through a [Thermo](../Thermo.txt) object:"
    ]
}, {
    "cell_type": "code", "execution_count": 32, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "Thermo(n", " chemicals=CompiledChemicals([Water, Ethanol, Propanol]),n", " mixture=Mixture(n", " rule='ideal mixing', ...n", " include_excess_energies=Falsen", "),n", " Gamma=DortmundActivityCoefficients,n", " Phi=IdealFugacityCoefficients,n", " PCF=IdealPoyintingCorrectionFactorsn", ")n"
            ]
        }
    ], "source": [
        "# A Thermo object is built with an iterable of Chemicals or their IDs.n", "# Default mixture, thermodynamic equilibrium models are selected.n", "thermo = tmo.Thermo(chemicals)n", "thermo.show()"
    ]
}, {
```

"cell_type": "markdown", "metadata": {}, "source": [

"[Creating a thermo property package](./Thermo_property_packages.ipynb), may be a little challenging if some chemicals cannot be found in the database, in which case they can be built from scratch. A complete example on how this can be done is available in another [tutorial](./Thermo_property_packages.ipynb)."

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"### Material and energy balance"

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"A [Stream](../Stream.txt) object is the main interface for estimating thermodynamic properties, vapor-liquid equilibrium, and material and energy balances. First set the thermo property package and we can start creating streams:"

]

}, {

"cell_type": "code", "execution_count": 33, "metadata": {}, "outputs": [

{

"name": "stdout", "output_type": "stream", "text": [

"Stream: s1n", " phase: 'l', T: 298.15 K, P: 101325 Pan", " flow (kg/hr): Water 20n", " Ethanol 20n"

]

}

], "source": [

"tmo.settings.set_thermo(thermo)n", "s1 = tmo.Stream('s1', Water=20, Ethanol=20, units='kg/hr')n", "s1.show(flow='kg/hr')"

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"Create another stream at a higher temperature:"

]

}, {

"cell_type": "code", "execution_count": 34, "metadata": {}, "outputs": [

{

"name": "stdout", "output_type": "stream", "text": [

"Stream: s2n", " phase: 'l', T: 350 K, P: 101325 Pan", " flow (kg/hr): Water 10n"

```
            ]
        }
    ], "source": [
        "s2 = tmo.Stream('s2', Water=10, units='kg/hr', T=350, P=101325)\n",
        "s2.show(flow='kg/hr')"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Mix both stream into a new one:"
    ]
}, {
    "cell_type": "code", "execution_count": 35, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "Stream: s_mix\n", " phase: 'l', T: 310.53 K, P: 101325 Pa\n", " flow (kg/hr): Water 30\n", " Ethanol 20\n"
            ]
        }
    ], "source": [
        "s_mix = tmo.Stream('s_mix')\n", "s_mix.mix_from([s1, s2])\n", "s_mix.show(flow='kg/hr')"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Check the energy balance through enthalpy:"
    ]
}, {
    "cell_type": "code", "execution_count": 36, "metadata": {}, "outputs": [
        {
            "data": {
                "text/plain": [
                    "9.094947017729282e-12"
                ]
            }, "execution_count": 36, "metadata": {}, "output_type": "execute_result"
        }
    ], "source": [
        "s_mix.H - (s1.H + s2.H)"
    ]
```

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Note that the balance is not perfect as the solver stops within a small temperature tolerance. However, the approximation is less than 0.01% off:”

]

}, {

“cell_type”: “code”, “execution_count”: 37, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“0.00%n”

]

}

], “source”: [

“error = s_mix.H - (s1.H + s2.H)n”, “percent_error = 100 * error / (s1.H + s2.H)n”, “print(f"{percent_error:.2%}")”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Split the mixture to two streams by defining the component splits:”

]

}, {

“cell_type”: “code”, “execution_count”: 38, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“Stream: s1n”, ” phase: ‘l’, T: 310.53 K, P: 101325 Pan”, ” flow (kg/hr): Ethanol 20n”, “Stream: s2n”, ” phase: ‘l’, T: 310.53 K, P: 101325 Pan”, ” flow (kg/hr): Water 30n”

]

}

], “source”: [

“# First define an array of component splitsn”, “component_splits = s_mix.chemicals.array([‘Water’, ‘Ethanol’], [0, 1])n”, “s_mix.split_to(s1, s2, component_splits)n”, “s1.T = s2.T = s_mix.T # Take care of energy balancen”, “s1.show(flow=’kg/hr’)n”, “s2.show(flow=’kg/hr’)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Flow rates”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“The most convenient way to get and set flow rates is through the *get_flow* and *set_flow* methods:”

]

}, {

“cell_type”: “code”, “execution_count”: 39, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“1.0”

]

}, “execution_count”: 39, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Set and get flow of a single chemicaln”, “# in gallons per minuten”, “s1.set_flow(1, ‘gpm’, ‘Water’)n”, “s1.get_flow(‘gpm’, ‘Water’)”

]

}, {

“cell_type”: “code”, “execution_count”: 40, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“array([10., 20.])”

]

}, “execution_count”: 40, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Set and get flows of many chemicalsn”, “# in kilograms per hourn”, “s1.set_flow([10, 20], ‘kg/hr’, (‘Ethanol’, ‘Water’))n”, “s1.get_flow(‘kg/hr’, (‘Ethanol’, ‘Water’))”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“It is also possible to index flow rate data using chemical IDs through the *imol*, *imass*, and *ivol* [indexers](../indexer/indexer_module.txt):”

]

}, {

“cell_type”: “code”, “execution_count”: 41, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“ChemicalMolarFlowIndexer (kmol/hr):n”, ” (l) Water 1.11n”, ” Ethanol 0.2171n”

]

}

], “source”: [

“s1.imol.show()”

]

}, {

“cell_type”: “code”, “execution_count”: 42, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [

“1.1101687012358397”

]

}, “execution_count”: 42, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“s1.imol[‘Water’]”

]

}, {

“cell_type”: “code”, “execution_count”: 43, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [

“array([0.217, 1.11])”

]

}, “execution_count”: 43, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“s1.imol[‘Ethanol’, ‘Water’]”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“All flow rates are stored as an array in the *mol* attribute:”

```
        ]
    }, {
        "cell_type": "code", "execution_count": 44, "metadata": {}, "outputs": [
            {
                "data": {
                  "text/plain": [
                    "array([1.11 , 0.217, 0.  ])"
                  ]
                }, "execution_count": 44, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "s1.mol # Molar flow rates [kmol/hr]"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "Mass and volumetric flow rates are available as [property arrays](https://free-properties.readthedocs.io/en/latest/property_array.html):"
        ]
    }, {
        "cell_type": "code", "execution_count": 45, "metadata": {}, "outputs": [
            {
                "data": {
                  "text/plain": [
                    "property_array([<Water: 20 kg/hr>, <Ethanol: 10 kg/hr>,n", " <Propanol: 0 kg/hr>])"
                  ]
                }, "execution_count": 45, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "s1.mass"
        ]
    }, {
        "cell_type": "code", "execution_count": 46, "metadata": {}, "outputs": [
            {
                "data": {
                  "text/plain": [
                    "property_array([<Water: 0.020166 m^3/hr>, <Ethanol: 0.012898 m^3/hr>,n", " <Propanol: 0 m^3/hr>])"
```

```
            ]
        }, “execution_count”: 46, “metadata”: {}, “output_type”: “execute_result”
    }
], “source”: [
    “s1.vol”
]
}, {
“cell_type”: “markdown”, “metadata”: {}, “source”: [
    “These arrays work just like ordinary arrays, but the data is linked to the molar flows:”
]
}, {
“cell_type”: “code”, “execution_count”: 47, “metadata”: {}, “outputs”: [
    {
        “data”: {
          “text/plain”: [
            “<Water: 18.015 kg/hr>”
          ]
        }, “execution_count”: 47, “metadata”: {}, “output_type”: “execute_result”
    }
], “source”: [
    “# Mass flows are always up to date with molar flowsn”, “s1.mol[0] = 1n”, “s1.mass[0]”
]
}, {
“cell_type”: “code”, “execution_count”: 48, “metadata”: {}, “outputs”: [
    {
        “data”: {
          “text/plain”: [
            “2.0”
          ]
        }, “execution_count”: 48, “metadata”: {}, “output_type”: “execute_result”
    }
], “source”: [
    “# Changing mass flows changes molar flowsn”, “s1.mass[0] *= 2n”, “s1.mol[0]”
]
}, {
“cell_type”: “code”, “execution_count”: 49, “metadata”: {}, “outputs”: [
    {
```

“data”: {

“text/plain”: [
“array([38.031, 12. , 2.])”

]

}, “execution_count”: 49, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Property arrays act just like normal arraysn”, “s1.mass + 2 # A new array is created”

]

}, {

“cell_type”: “code”, “execution_count”: 50, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“15.34352”

]

}, “execution_count”: 50, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Array methods are also the samen”, “s1.mass.mean()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Thermal condition”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Temperature and pressure can be get and set through the *T* and *P* attributes:”

]

}, {

“cell_type”: “code”, “execution_count”: 51, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“Stream: s1n”, ” phase: ‘l’, T: 400 K, P: 202650 Pan”, ” flow (kmol/hr): Water 2n”, ” Ethanol 0.217n”

]

}

], “source”: [

“s1.T = 400.n”, “s1.P = 2 * 101325.n”, “s1.show()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“The phase may also be changed (‘s’ for solid, ‘l’ for liquid, and ‘g’ for gas):”

]

}, {

“cell_type”: “code”, “execution_count”: 52, “metadata”: {}, “outputs”: [], “source”: [

“s1.phase = ‘g’”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Notice that VLE is not enforced, but it is possible to perform. For now, just check that the dew point is lower than the actual temperature to assert it must be gas:”

]

}, {

“cell_type”: “code”, “execution_count”: 53, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [

“DewPointValues(T=390.84, P=202650, IDs=(‘Water’, ‘Ethanol’), z=[0.902 0.098], x=[0.991 0.009])”

]

}, “execution_count”: 53, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“dp = s1.dew_point_at_P() # Dew point at constant pressuren”, “dp”

]

}, {

“cell_type”: “code”, “execution_count”: 54, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [

“True”

]

}, “execution_count”: 54, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“dp.T < s1.T”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“It is also possible to get and set in other units of measure:”

]

}, {

“cell_type”: “code”, “execution_count”: 55, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“1.0”

]

}, “execution_count”: 55, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“s1.set_property(‘P’, 1, ‘atm’)n”, “s1.get_property(‘P’, ‘atm’)”

]

}, {

“cell_type”: “code”, “execution_count”: 56, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“256.99999999999994”

]

}, “execution_count”: 56, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“s1.set_property(‘T’, 125, ‘degC’)n”, “s1.get_property(‘T’, ‘degF’)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Enthalpy can also be set. An energy balance is made to solve for temperature at isobaric conditions:”

```
    ]
}, {
    "cell_type": "code", "execution_count": 57, "metadata": {}, "outputs": [
        {
            "data": {
                "text/plain": [
                    "130.80215821464316"
                ]
            }, "execution_count": 57, "metadata": {}, "output_type": "execute_result"
        }
    ], "source": [
        "s1.H = s1.H + 500n", "s1.get_property('T', 'degC') # Temperature should go up"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "### Thermal properties"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Thermodynamic properties are pressure, temperature and phase dependent. In the following examples, let's just use water as it is easier to check properties:"
    ]
}, {
    "cell_type": "code", "execution_count": 58, "metadata": {}, "outputs": [
        {
            "data": {
                "text/plain": [
                    "997.0156689562489"
                ]
            }, "execution_count": 58, "metadata": {}, "output_type": "execute_result"
        }
    ], "source": [
        "s_water = tmo.Stream('s_water', Water=1, units='kg/hr')n", "s_water.rho # Density [kg/m^3]"
    ]
}, {
    "cell_type": "code", "execution_count": 59, "metadata": {}, "outputs": [
```

```
            {
                "data": {
                  "text/plain": [
                    "971.4430230945908"
                  ]
                }, "execution_count": 59, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "s_water.T = 350n", "s_water.rho # Density changes"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "Get properties in different units:"
        ]
    }, {
        "cell_type": "code", "execution_count": 60, "metadata": {}, "outputs": [
            {
                "data": {
                  "text/plain": [
                    "0.06324769600985489"
                  ]
                }, "execution_count": 60, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "s_water.get_property('sigma', 'N/m') # Surface tension"
        ]
    }, {
        "cell_type": "code", "execution_count": 61, "metadata": {}, "outputs": [
            {
                "data": {
                  "text/plain": [
                    "0.01854486528979459"
                  ]
                }, "execution_count": 61, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "s_water.get_property('V', 'm3/kmol') # Molar volume"
```

```
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "### Flow properties"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "Several flow properties are available, such as net material and energy flow rates:"
        ]
    }, {
        "cell_type": "code", "execution_count": 62, "metadata": {}, "outputs": [
            {
                "data": {
                  "text/plain": [
                    "0.05550843506179199"
                  ]
                }, "execution_count": 62, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "# Net molar flow rate [kmol/hr]n", "s_water.F_mol"
        ]
    }, {
        "cell_type": "code", "execution_count": 63, "metadata": {}, "outputs": [
            {
                "data": {
                  "text/plain": [
                    "1.0"
                  ]
                }, "execution_count": 63, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "# Net mass flow rate [kg/hr]n", "s_water.F_mass"
        ]
    }, {
        "cell_type": "code", "execution_count": 64, "metadata": {}, "outputs": [
            {
                "data": {
```

“text/plain”: [
“0.0010293964506682433”
]
}, “execution_count”: 64, “metadata”: {}, “output_type”: “execute_result”
}
], “source”: [
“# Net volumetric flow rate [m3/hr]n”, “s_water.F_vol”
]
}, {
“cell_type”: “code”, “execution_count”: 65, “metadata”: {}, “outputs”: [
{
“data”: {
“text/plain”: [
“216.85387645424356”
]
}, “execution_count”: 65, “metadata”: {}, “output_type”: “execute_result”
}
], “source”: [
“# Enthalpy flow rate [kJ/hr]n”, “s_water.H”
]
}, {
“cell_type”: “code”, “execution_count”: 66, “metadata”: {}, “outputs”: [
{
“data”: {
“text/plain”: [
“4.556131378540336”
]
}, “execution_count”: 66, “metadata”: {}, “output_type”: “execute_result”
}
], “source”: [
“# Entropy flow rate [kJ/hr]n”, “s_water.S”
]
}, {
“cell_type”: “code”, “execution_count”: 67, “metadata”: {}, “outputs”: [
{
“data”: {

“text/plain”: [
“4.197680667946338”

]

}, “execution_count”: 67, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Capacity flow rate [J/K]n”, “s_water.C”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Thermodynamic equilibrium”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Before moving into performing vapor-liquid and liquid-liquid equilibrium calculations, it may be useful to have a look at the phase envelopes to understand chemical interactions and ultimately how they separate between phases.”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Plot the binary phase evelope of two chemicals in vapor-liquid equilibrium at constant pressure:”

]

}, {

“cell_type”: “code”, “execution_count”: 68, “metadata”: {}, “outputs”: [

{

“data”: {
“image/png”: “iVBORw0KGgoAAAANSUhEUgAAAZ4AAAEFCAYAAADT3YGPAAAABHNCSVQIC
“text/plain”: [

“<Figure size 432x288 with 3 Axes>”

]

}, “metadata”: {

“needs_background”: “light”

}, “output_type”: “display_data”

}

], “source”: [

“eq = tmo.equilibrium # Thermosteam’s equilibrium modulen”, “eq.plot_vle_binary_phase_envelope([‘Ethanol’, ‘Water’], P=101325)n”, “plt.show()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Plot the ternary phase diagram of three chemicals in liquid-liquid equilibrium at constant pressure:”

]

}, {

“cell_type”: “code”, “execution_count”: 69, “metadata”: {}, “outputs”: [

{

“data”: {

“image/png”: “iVBORw0KGgoAAAANSUhEUgAAAbQAAAEXCAYAAADFvLEGAAAABHNCSVQI

“text/plain”: [

“<Figure size 432x288 with 1 Axes>”

]

}, “metadata”: {}, “output_type”: “display_data”

}

], “source”: [

“# This one will take like 30 secondsn”, “# Thermosteam’s LLE algorithm is stochastic,n”, “# so its much slower than the VLE algorithm.n”, “# You’ll need to "pip install python-ternary" to run this linen”, “eq.plot_lle_ternary_diagram(‘Water’, ‘Ethanol’, ‘EthylAcetate’, T=298.15)n”, “plt.show()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Vapor-liquid equilibrium”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Vapor-liquid equilibrium can be performed by setting 2 degrees of freedom from the following list: *T* (Temperature; in K), *P* (Pressure; in Pa), *V* (Vapor fraction), and *H* (Enthalpy; in kJ/hr).n”, “n”, “For example, set vapor fraction and pressure:”

]

}, {

“cell_type”: “code”, “execution_count”: 70, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“MultiStream: s_eqn”, ” phases: (‘g’, ‘l’), T: 353.88 K, P: 101325 Pan”, ” composition: (g) Water 0.3861n”, ” Ethanol 0.6139n”, ” ——- 10 kmol/hrn”, ” (l) Water 0.6139n”, ” Ethanol 0.3861n”, ” ——- 10 kmol/hrn”

```
            ]
        }
    ], "source": [
        "s_eq = tmo.Stream('s_eq', Water=10, Ethanol=10)\n", "s_eq.vle(V=0.5, P=101325)\n", "s_eq.show(composition=True)"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Note that the stream is a now a MultiStream to manage multiple phases. Each phase can be accessed separately too:"
    ]
}, {
    "cell_type": "code", "execution_count": 71, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "Stream: \n", " phase: 'l', T: 353.88 K, P: 101325 Pa\n", " flow (kmol/hr): Water 6.14\n", " Ethanol 3.86\n"
            ]
        }
    ], "source": [
        "s_eq['l'].show()"
    ]
}, {
    "cell_type": "code", "execution_count": 72, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "Stream: \n", " phase: 'g', T: 353.88 K, P: 101325 Pa\n", " flow (kmol/hr): Water 3.86\n", " Ethanol 6.14\n"
            ]
        }
    ], "source": [
        "s_eq['g'].show()"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Note that the phase of these substreams cannot be changed:"
    ]
}, {
```

```
"cell_type": "code", "execution_count": 73, "metadata": {}, "outputs": [
    {
        "ename": "AttributeError", "evalue": "phase is locked", "output_type": "error",
        "traceback": [
            "u001b[1;31m---------------------------------------------------------------------u001b[0m",
            "u001b[1;31mAttributeErroru001b[0m Traceback (most recent call
            last)", "u001b[1;32m<ipython-input-73-ed0136a78442>u001b[0m
            in u001b[0;36m<module>u001b[1;34mu001b[0mnu001b[1;32m---->
            1u001b[1;33m u001b[0ms_equ001b[0mu001b[1;33m[u001b[0mu001b[1;34m'g'u001b[0mu001b[1;33m]u
            u001b[1;33m=u001b[0m u001b[1;34m'l'u001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0
            "u001b[1;32m~\OneDrive\Code\thermosteam\thermosteam\_stream.pyu001b[0m
            in u001b[0;36mphaseu001b[1;34m(self, phase)u001b[0mnu001b[0;32m
            589u001b[0m u001b[1;33m@u001b[0mu001b[0mphaseu001b[0mu001b[1;33m.u001b[0mu001b[0msetter
            590u001b[0m u001b[1;32mdefu001b[0m u001b[0mphaseu001b[0mu001b[1;33m(u001b[0mu001b[0mselfu
            u001b[0mphaseu001b[0mu001b[1;33m)u001b[0mu001b[1;33m:u001b[0mu001b[1;33mu001b[0mu001b[1
            591u001b[1;33m u001b[0mselfu001b[0mu001b[1;33m.u001b[0mu001b[0m_imolu001b[0mu001b[1;33m.
            u001b[1;33m=u001b[0m u001b[0mphaseu001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0
            592u001b[0m u001b[1;33mu001b[0mu001b[0mnu001b[0;32m
            593u001b[0m u001b[1;33m@u001b[0mu001b[0mpropertyu001b[0mu001b[1;33mu001b[0mu001b[1;33m
            "u001b[1;32m~\OneDrive\Code\thermosteam\thermosteam\indexer.pyu001b[0m
            in u001b[0;36mphaseu001b[1;34m(self, phase)u001b[0mnu001b[0;32m
            223u001b[0m u001b[1;33m@u001b[0mu001b[0mphaseu001b[0mu001b[1;33m.u001b[0mu001b[0msetter
            224u001b[0m u001b[1;32mdefu001b[0m u001b[0mphaseu001b[0mu001b[1;33m(u001b[0mu001b[0mselfu
            u001b[0mphaseu001b[0mu001b[1;33m)u001b[0mu001b[1;33m:u001b[0mu001b[1;33mu001b[0mu001b[1
            225u001b[1;33m u001b[0mselfu001b[0mu001b[1;33m.u001b[0mu001b[0m_phaseu001b[0mu001b[1;33m
            u001b[1;33m=u001b[0m u001b[0mphaseu001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0
            226u001b[0m u001b[1;33mu001b[0mu001b[0mnu001b[0;32m
            227u001b[0m u001b[1;32mdefu001b[0m u001b[0m__format__u001b[0mu001b[1;33m(u001b[0mu001b[0
            u001b[0mtabsu001b[0mu001b[1;33m=u001b[0mu001b[1;34m""u001b[0mu001b[1;33m)u001b[0mu001b[
            "u001b[1;32m~\OneDrive\Code\thermosteam\thermosteam\_phase.pyu001b[0m
            in u001b[0;36m__setattr__u001b[1;34m(self, name,
            value)u001b[0mnu001b[0;32m 68u001b[0m u001b[1;32mdefu001b[0m
            u001b[0m__setattr__u001b[0mu001b[1;33m(u001b[0mu001b[0mselfu001b[0mu001b[1;33m,u001b[0m
            u001b[0mnameu001b[0mu001b[1;33m,u001b[0m
            u001b[0mvalueu001b[0mu001b[1;33m)u001b[0mu001b[1;33m:u001b[0mu001b[1;33mu001b[0mu001b[1
            69u001b[0m u001b[1;32mifu001b[0m u001b[0mvalueu001b[0m
            u001b[1;33m!=u001b[0m u001b[0mselfu001b[0mu001b[1;33m.u001b[0mu001b[0mphaseu001b[0mu001b
            70u001b[1;33m u001b[1;32mraiseu001b[0m
            u001b[0mAttributeErroru001b[0mu001b[1;33m(u001b[0mu001b[1;34m'phase
            is locked'u001b[0mu001b[1;33m)u001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0mnu00
            71u001b[0m u001b[1;33mu001b[0mu001b[0mnu001b[0;32m
            72u001b[0m u001b[0mNoPhaseu001b[0m u001b[1;33m=u001b[0m
            u001b[0mLockedPhaseu001b[0mu001b[1;33m(u001b[0mu001b[1;32mNoneu001b[0mu001b[1;33m)u001
            "u001b[1;31mAttributeErroru001b[0m: phase is locked"
        ]
    }
], "source": [
    "s_eq['g'].phase = 'l'"
]
}, {
```

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Again, the most convenient way to get and set flow rates is through the *get_flow* and *set_flow* methods:”

]

}, {

“cell_type”: “code”, “execution_count”: 74, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“1.0”

]

}, “execution_count”: 74, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Set flow of liquid watern”, “s_eq.set_flow(1, ‘gpm’, (‘l’, ‘Water’))n”, “s_eq.get_flow(‘gpm’, (‘l’, ‘Water’))”

]

}, {

“cell_type”: “code”, “execution_count”: 75, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“array([10., 20.])”

]

}, “execution_count”: 75, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Set multiple liquid flowsn”, “key = (‘l’, (‘Ethanol’, ‘Water’))n”, “s_eq.set_flow([10, 20], ‘kg/hr’, key)n”, “s_eq.get_flow(‘kg/hr’, key)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Chemical flows across all phases can be retrieved if no phase is given:”

]

}, {

“cell_type”: “code”, “execution_count”: 76, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“array([89.565, 292.793])”
]
}, “execution_count”: 76, “metadata”: {}, “output_type”: “execute_result”
}
], “source”: [
“# Get water and ethanol flows summed across all phasesn”, “s_eq.get_flow(‘kg/hr’, (‘Water’, ‘Ethanol’))”
]
}, {
“cell_type”: “markdown”, “metadata”: {}, “source”: [
“However, setting chemical data of MultiStream objects requires the phase to be specified:”
]
}, {
“cell_type”: “code”, “execution_count”: 77, “metadata”: {}, “outputs”: [
{
“ename”: “IndexError”, “evalue”: “multiple phases present; must include phase key to set chemical data”, “output_type”: “error”, “traceback”: [
“u001b[1;31m——u001b[0m”,
“u001b[1;31mIndexErroru001b[0m Traceback (most recent call last)”, “u001b[1;32m<ipython-input-77-d6cf98178f52>u001b[0m in u001b[0;36m<module>u001b[1;34mu001b[0mnu001b[1;32m—–> 1u001b[1;33m u001b[0ms_equ001b[0mu001b[1;33m.u001b[0mu001b[0mset_flowu001b[0mu001b[1;33m u001b[1;36m20u001b[0mu001b[1;33m]u001b[0mu001b[1;33m,u001b[0m u001b[1;34m’kg/hr’u001b[0mu001b[1;33m,u001b[0m u001b[1;33m(u001b[0mu001b[1;34m’Water’u001b[0mu001b[1;33m,u001b[0m u001b[1;34m’Ethanol’u001b[0mu001b[1;33m)u001b[0mu001b[1;33m)u001b[0mu001b[1;33mu001b[0mu
“u001b[1;32m~\OneDrive\Code\thermosteam\thermosteam_multi_stream.pyu001b[0m in u001b[0;36mset_flowu001b[1;34m(self, data, units, key)u001b[0mnu001b[0;32m 308u001b[0m u001b[0mnameu001b[0mu001b[1;33m,u001b[0m u001b[0mfactoru001b[0m u001b[1;33m=u001b[0m u001b[0mselfu001b[0mu001b[1;33m.u001b[0mu001b[0m_get_flow_name_and_factoru001b[0mu001b[1;
309u001b[0m u001b[0mindexeru001b[0m u001b[1;33m=u001b[0m u001b[0mgetattru001b[0mu001b[1;33m(u001b[0mu001b[0mselfu001b[0mu001b[1;33m,u001b[0m u001b[1;34m’i’u001b[0m u001b[1;33m+u001b[0m u001b[0mnameu001b[0mu001b[1;33m)u001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0
310u001b[1;33m u001b[0mindexeru001b[0mu001b[1;33m[u001b[0mu001b[0mkeyu001b[0mu001b[1;33m u001b[1;33m=u001b[0m u001b[0mnpu001b[0mu001b[1;33m.u001b[0mu001b[0masarrayu001b[0mu001b u001b[0mdtypeu001b[0mu001b[1;33m=u001b[0mu001b[0mfloatu001b[0mu001b[1;33m)u001b[0m u001b[1;33m/u001b[0m u001b[0mfactoru001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0
311u001b[0m u001b[1;33mu001b[0mu001b[0mnu001b[0;32m 312u001b[0m u001b[1;31m### Stream data ###u001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0mn”,
“u001b[1;32m~\OneDrive\Code\thermosteam\thermosteam\indexer.pyu001b[0m in u001b[0;36m__setitem__u001b[1;34m(self,

key, data)u001b[0mnu001b[0;32m 484u001b[0m
u001b[0mindexu001b[0m u001b[1;33m=u001b[0m
u001b[0mselfu001b[0mu001b[1;33m.u001b[0mu001b[0mget_indexu001b[0mu001b[1;33m(u001b[0mu00
485u001b[0m u001b[1;32mifu001b[0m u001b[0misau001b[0mu001b[1;33m(u001b[0mu001b[0mindexu0
u001b[0mChemicalIndexu001b[0mu001b[1;33m)u001b[0mu001b[1;33m:u001b[0mu001b[1;33mu001b[0
486u001b[1;33m raise IndexError("multiple phases present;
must include phase key "nu001b[0mu001b[0;32m 487u001b[0m
"to set chemical data") nu001b[0;32m 488u001b[0m
u001b[0mselfu001b[0mu001b[1;33m.u001b[0mu001b[0m_datau001b[0mu001b[1;33m[u001b[0mu001b[0
u001b[1;33m=u001b[0m u001b[0mdatau001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0m
“u001b[1;31mIndexErroru001b[0m: multiple phases present; must include
phase key to set chemical data”

]

}

], “source”: [

“s_eq.set_flow([10, 20], ‘kg/hr’, (‘Water’, ‘Ethanol’))”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Similar to Stream objects, all flow rates can be accessed through the *imol*, *imass*, and *ivol* attributes:”

]

}, {

“cell_type”: “code”, “execution_count”: 78, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“MolarFlowIndexer (kmol/hr):n”, ” (g) Water 3.861n”, ” Ethanol 6.139n”, ”
(l) Water 1.11n”, ” Ethanol 0.2171n”

]

}

], “source”: [

“s_eq.imol # Molar flow rates”

]

}, {

“cell_type”: “code”, “execution_count”: 79, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“1.1101687012358397”

]

}, “execution_count”: 79, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Index a single chemical in the liquid phasen”, “s_eq.imol[‘l’, ‘Water’]”

]

}, {

“cell_type”: “code”, “execution_count”: 80, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“array([0.217, 1.11])”

]

}, “execution_count”: 80, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Index multiple chemicals in the liquid phasen”, “s_eq.imol[‘l’, (‘Ethanol’, ‘Water’)]”

]

}, {

“cell_type”: “code”, “execution_count”: 81, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“array([3.861, 6.139, 0.])”

]

}, “execution_count”: 81, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Index the vapor phasen”, “s_eq.imol[‘g’]”

]

}, {

“cell_type”: “code”, “execution_count”: 82, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“array([6.356, 4.972])”

]

}, “execution_count”: 82, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Index flow of chemicals summed across all phasesn”, “s_eq.imol[‘Ethanol’, ‘Water’]”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Because multiple phases are present, overall chemical flows in MultiStream objects cannot be set like in Stream objects:”

]

}, {

“cell_type”: “code”, “execution_count”: 83, “metadata”: {}, “outputs”: [

{

“ename”: “IndexError”, “evalue”: “multiple phases present; must include phase key to set chemical data”, “output_type”: “error”, “traceback”: [

“u001b[1;31m——u001b[0m”, “u001b[1;31mIndexErroru001b[0m Traceback (most recent call last)”, “u001b[1;32m<ipython-input-83-fcb482ddb0a2>u001b[0m in u001b[0;36m<module>u001b[1;34mu001b[0mnu001b[1;32m—–> 1u001b[1;33m u001b[0ms_equ001b[0mu001b[1;33m.u001b[0mu001b[0mimolu001b[0mu001b[1;33m[u0 u001b[1;34m’Water’u001b[0mu001b[1;33m]u001b[0m u001b[1;33m=u001b[0m u001b[1;33m[u001b[0mu001b[1;36m1u001b[0mu001b[1;33m,u001b[0m u001b[1;36m0u001b[0mu001b[1;33m]u001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0m “u001b[1;32m~\OneDrive\Code\thermosteam\thermosteam\indexer.pyu001b[0m in u001b[0;36m__setitem__u001b[1;34m(self, key, data)u001b[0mnu001b[0;32m 484u001b[0m u001b[0mindexu001b[0m u001b[1;33m=u001b[0m u001b[0mselfu001b[0mu001b[1;33m.u001b[0mu001b[0mget_indexu001b[0mu001b[1;33m(u001b[0mu00 485u001b[0m u001b[1;32mifu001b[0m u001b[0misau001b[0mu001b[1;33m(u001b[0mu001b[0mindexu0 u001b[0mChemicalIndexu001b[0mu001b[1;33m)u001b[0mu001b[1;33m:u001b[0mu001b[1;33mu001b[0 486u001b[1;33m raise IndexError("multiple phases present; must include phase key "nu001b[0mu001b[0;32m 487u001b[0m "to set chemical data") nu001b[0;32m 488u001b[0m u001b[0mselfu001b[0mu001b[1;33m.u001b[0mu001b[0m_datau001b[0mu001b[1;33m[u001b[0mu001b[0 u001b[1;33m=u001b[0m u001b[0mdatau001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0m “u001b[1;31mIndexErroru001b[0m: multiple phases present; must include phase key to set chemical data”

]

}

], “source”: [

“s_eq.imol[‘Ethanol’, ‘Water’] = [1, 0]”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Chemical flows must be set by phase:”

]

}, {

“cell_type”: “code”, “execution_count”: 84, “metadata”: {}, “outputs”: [], “source”: [

“s_eq.imol[‘l’, (‘Ethanol’, ‘Water’)] = [1, 0]”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“One main difference between a [MultiStream](../MultiStream.txt) object and a [Stream](../Stream.txt) object is that the *mol* attribute no longer stores any data, it simply returns the total flow rate of each chemical. Setting an element of the array raises an error to prevent the wrong assumption that the data is linked:”

]

}, {

“cell_type”: “code”, “execution_count”: 85, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [

“array([3.861, 7.139, 0.])”

]

}, “execution_count”: 85, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“s_eq.mol”

]

}, {

“cell_type”: “code”, “execution_count”: 86, “metadata”: {}, “outputs”: [

{

“ename”: “ValueError”, “evalue”: “assignment destination is read-only”, “output_type”: “error”, “traceback”: [

“u001b[1;31m———u001b[0m”, “u001b[1;31mValueErroru001b[0m Traceback (most recent call last)”, “u001b[1;32m<ipython-input-86-632093460ce3>u001b[0m in u001b[0;36m<module>u001b[1;34mu001b[0mnu001b[1;32m—–> 1u001b[1;33m u001b[0ms_equ001b[0mu001b[1;33m.u001b[0mu001b[0mmolu001b[0mu001b[1;33m[u00 u001b[1;33m=u001b[0m u001b[1;36m1u001b[0mu001b[1;33mu001b[0mu001b[1;33mu001b[0mu001b[0m “u001b[1;31mValueErroru001b[0m: assignment destination is read-only”

]

}

], “source”: [

“s_eq.mol[0] = 1”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Note that for both Stream and MultiStream objects, *get_flow*, *imol*, and *mol* return chemical flows across all phases when given only chemical IDs.”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Liquid-liquid equilibrium”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Liquid-liquid equilibrium (LLE) only requires the temperature. Pressure is not a significant variable as liquid fungacity coefficients are not a strong function of pressure. “

]

}, {

“cell_type”: “code”, “execution_count”: 87, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“MultiStream: liquid_mixturen”, ” phases: (‘L’, ‘l’), T: 300 K, P: 101325 Pan”, ” flow (kmol/hr): (L) Water 98.54n”, ” Butanol 1.209n”, ” Octane 0.001977n”, ” (l) Water 1.458n”, ” Butanol 3.791n”, ” Octane 100n”

]

}

], “source”: [

“tmo.settings.set_thermo([‘Water’, ‘Butanol’, ‘Octane’])n”, “liquid_mixture = tmo.Stream(‘liquid_mixture’, Water=100, Octane=100, Butanol=5)n”, “liquid_mixture.lle(T=300)n”, “liquid_mixture.show()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Compared to VLE, LLE is several orders of magnitude times slower. This is because differential evolution, a purely stochastic method, is used to find the solution that globally minimizes the gibb’s free energy of both phases. For now, the LLE algorithm may not present completely accurate results and is subject to change in the future.”

]

}

], “metadata”: {

“kernelspec”: {
“display_name”: “Python 3”, “language”: “python”, “name”: “python3”

}, “language_info”: {

“codemirror_mode”: {
“name”: “ipython”, “version”: 3

}, “file_extension”: “.py”, “mimetype”: “text/x-python”, “name”: “python”, “nbconvert_exporter”: “python”, “pygments_lexer”: “ipython3”, “version”: “3.7.6”

}

}, “nbformat”: 4, “nbformat_minor”: 2

}

{

“cells”: [

{
“cell_type”: “markdown”, “metadata”: {}, “source”: [

“# Thermodynamic equilibrium”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“It is not necessary to use a Stream object to use thermodynamic equilibrium methods. In fact, thermosteam makes it just as easy to compute vapor-liquid equilibrium, bubble and dew points, and fugacities. “

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Fugacities”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“The easiest way to calculate fugacities is through LiquidFugacities and GasFugacities objects:”

]

}, {

“cell_type”: “code”, “execution_count”: 1, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“array([43274.119, 58056.67])”

]

}, “execution_count”: 1, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“import thermosteam as tmon”, “import numpy as npn”, “chemicals = tmo.Chemicals([‘Water’, ‘Ethanol’])n”, “tmo.settings.set_thermo(chemicals)n”, “n”, “# Create a LiquidFugacities objectn”, “F_l = tmo.equilibrium.LiquidFugacities(chemicals)n”, “n”, “# Compute liquid fugacitiesn”, “liquid_molar_composition = np.array([0.72, 0.28])n”, “f_l = F_l(x=liquid_molar_composition, T=355)n”, “f_l “

]

}, {

“cell_type”: “code”, “execution_count”: 2, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [

“array([43569.75, 57755.25])”

]

}, “execution_count”: 2, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Create a GasFugacities objectn”, “F_g = tmo.equilibrium.GasFugacities(chemicals)n”, “n”, “# Compute gas fugacitiesn”, “gas_molar_composition = np.array([0.43, 0.57])n”, “f_g = F_g(y=gas_molar_composition, T=355, P=101325)n”, “f_g”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Bubble and dew points”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Similarly bubble and dew point can be calculated through BubblePoint and DewPoint objects:”

]

}, {

“cell_type”: “code”, “execution_count”: 3, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [

“BubblePointValues(T=355.00, P=109755, IDs=(‘Water’, ‘Ethanol’), z=[0.5 0.5], y=[0.343 0.657])”

```
                ]
            }, “execution_count”: 3, “metadata”: {}, “output_type”: “execute_result”
        }
    ], “source”: [
        “# Create a BubblePoint objectn”, “BP = tmo.equilibrium.BubblePoint(chemicals)n”, “molar_composition = np.array([0.5, 0.5])n”, “n”, “# Solve bubble point at constant temperaturen”, “bp = BP(z=molar_composition, T=355)n”, “bp”
    ]
}, {
    “cell_type”: “code”, “execution_count”: 4, “metadata”: {}, “outputs”: [
        {
            “data”: {
                “text/plain”: [
                    “(355,n”, ” 109755.45319869411,n”, ” (‘Water’, ‘Ethanol’),n”, ” array([0.5, 0.5]),n”, ” array([0.343, 0.657]))”
                ]
            }, “execution_count”: 4, “metadata”: {}, “output_type”: “execute_result”
        }
    ], “source”: [
        “# Note that the result is a BubblePointValues object which contain all results as attributesn”, “(bp.T, bp.P, bp.IDs, bp.z, bp.y)”
    ]
}, {
    “cell_type”: “code”, “execution_count”: 5, “metadata”: {}, “outputs”: [
        {
            “data”: {
                “text/plain”: [
                    “BubblePointValues(T=371.78, P=202650, IDs=(‘Water’, ‘Ethanol’), z=[0.5 0.5], y=[0.35 0.65])”
                ]
            }, “execution_count”: 5, “metadata”: {}, “output_type”: “execute_result”
        }
    ], “source”: [
        “# Solve bubble point at constant pressuren”, “BP(z=molar_composition, P=2*101325)”
    ]
}, {
    “cell_type”: “code”, “execution_count”: 6, “metadata”: {}, “outputs”: [
        {
```

“data”: {

“text/plain”: [
“DewPointValues(T=355.00, P=91970, IDs=(‘Water’, ‘Ethanol’), z=[0.5 0.5], x=[0.851 0.149])”

]

}, “execution_count”: 6, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Create a DewPoint objectn”, “DP = tmo.equilibrium.DewPoint(chemicals)n”, “n”, “# Solve for dew point at constant temperautren”, “dp = DP(z=molar_composition, T=355)n”, “dp”

]

}, {

“cell_type”: “code”, “execution_count”: 7, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“(355,n”, ” 91970.1496840849,n”, ” (‘Water’, ‘Ethanol’),n”, ” array([0.5, 0.5]),n”, ” array([0.851, 0.149]))”

]

}, “execution_count”: 7, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Note that the result is a DewPointValues object which contain all results as attributesn”, “(dp.T, dp.P, dp.IDs, dp.z, dp.x)”

]

}, {

“cell_type”: “code”, “execution_count”: 8, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“DewPointValues(T=376.26, P=202648, IDs=(‘Water’, ‘Ethanol’), z=[0.5 0.5], x=[0.832 0.168])”

]

}, “execution_count”: 8, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“# Solve for dew point at constant pressuren”, “DP(z=molar_composition, P=2*101324)”

```
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "### Vapor liquid equilibrium"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Vapor-liquid equilibrium can be calculated through a VLE object:"
    ]
}, {
    "cell_type": "code", "execution_count": 9, "metadata": {}, "outputs": [
        {
            "data": {
                "text/plain": [
                    "VLE(imol=MaterialIndexer(n",
                    "  g=[('Water', 0.5), ('Ethanol', 0.5)],n",
                    "  l=[('Water', 0.5), ('Ethanol', 0.5)]),n",
                    "  thermal_condition=ThermalCondition(T=298.15, P=101325))"
                ]
            }, "execution_count": 9, "metadata": {}, "output_type": "execute_result"
        }
    ], "source": [
        "# First create a material indexer for the VLE object to manage material datan",
        "imol = tmo.indexer.MaterialIndexer(l=[('Water', 0.5), ('Ethanol', 0.5)],n",
        "  g=[('Water', 0.5), ('Ethanol', 0.5)])n",
        "n",
        "# Create a VLE objectn",
        "vle = tmo.equilibrium.VLE(imol)n",
        "vle"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "You can call the VLE object by setting 2 degrees of freedom from the following list: n",
        "n",
        "* T - Temperature [K]n",
        "* P - Pressure [Pa]n",
        "* V - Molar vapor fractionn",
        "* H - Enthalpy [kJ/hr]n",
        "* y - Binary molar vapor compositionn",
        "* x - Binary molar liquid compositionn",
        "n",
        "Here are some examples:"
    ]
}, {
    "cell_type": "code", "execution_count": 10, "metadata": {}, "outputs": [
        {
            "data": {
                "text/plain": [
                    "VLE(imol=MaterialIndexer(n",
                    "  g=[('Water', 0.6417), ('Ethanol',
```

```
                0.8607)],n", " l=[('Water', 0.3583), ('Ethanol', 0.1393)]),n", " thermal_condition=ThermalCondition(T=355.00, P=101325))"

            ]

        }, "execution_count": 10, "metadata": {}, "output_type": "execute_result"

    }

], "source": [

    "vle(T=355, P=101325)n", "vle"

]

}, {

    "cell_type": "code", "execution_count": 11, "metadata": {}, "outputs": [

    {

        "data": {

            "text/plain": [
                "VLE(imol=MaterialIndexer(n", " g=[('Water', 0.6161), ('Ethanol', 0.8266)],n", " l=[('Water', 0.3839), ('Ethanol', 0.1734)]),n", " thermal_condition=ThermalCondition(T=373.69, P=202650))"

            ]

        }, "execution_count": 11, "metadata": {}, "output_type": "execute_result"

    }

], "source": [

    "mixture_enthalpy = vle.mixture.xH(imol, *vle.thermal_condition)n", "vle(H=mixture_enthalpy, P=202650)n", "vle"

]

}, {

    "cell_type": "code", "execution_count": 12, "metadata": {}, "outputs": [

    {

        "data": {

            "text/plain": [
                "VLE(imol=MaterialIndexer(n", " g=[('Water', 0.3861), ('Ethanol', 0.6139)],n", " l=[('Water', 0.6139), ('Ethanol', 0.3861)]),n", " thermal_condition=ThermalCondition(T=353.88, P=101325))"

            ]

        }, "execution_count": 12, "metadata": {}, "output_type": "execute_result"

    }

], "source": [

    "vle(V=0.5, P=101325)n", "vle"

]

}, {

    "cell_type": "code", "execution_count": 13, "metadata": {}, "outputs": [
```

```
            {
                "data": {
                  "text/plain": [
                    "VLE(imol=MaterialIndexer(n",  " g=[('Water',  0.3886),  ('Ethanol',
                    0.6114)],n",  " l=[('Water',  0.6114),  ('Ethanol',  0.3886)]),n",  " ther-
                    mal_condition=ThermalCondition(T=360.00, P=128136))"
                  ]
                }, "execution_count": 13, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "vle(V=0.5, T=360)n", "vle"
        ]
    }, {
        "cell_type": "code", "execution_count": 14, "metadata": {}, "outputs": [
            {
                "data": {
                  "text/plain": [
                    "VLE(imol=MaterialIndexer(n",  " g=[('Water',  0.8356),  ('Ethanol',
                    0.9589)],n",  " l=[('Water',  0.1644),  ('Ethanol',  0.04109)]),n",  " ther-
                    mal_condition=ThermalCondition(T=356.25, P=128136))"
                  ]
                }, "execution_count": 14, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "vle(x=np.array([0.8, 0.2]), P=101325)n", "vle"
        ]
    }, {
        "cell_type": "code", "execution_count": 15, "metadata": {}, "outputs": [
            {
                "data": {
                  "text/plain": [
                    "VLE(imol=MaterialIndexer(n",  " g=[('Water',  0.4691),  ('Ethanol',
                    0.7036)],n",  " l=[('Water',  0.5309),  ('Ethanol',  0.2964)]),n",  " ther-
                    mal_condition=ThermalCondition(T=356.25, P=126727))"
                  ]
                }, "execution_count": 15, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "vle(y=np.array([0.4, 0.6]), T=360)n", "vle"
```

]
}, {
"cell_type": "markdown", "metadata": {}, "source": [
"Note that some compositions are infeasible; so it is not advised to pass x or y unless you know what you're doing:"
]
}, {
"cell_type": "code", "execution_count": 16, "metadata": {}, "outputs": [], "source": [
"vle(x=np.array([0.2, 0.8]), P=101325)"
]
}, {
"cell_type": "markdown", "metadata": {}, "source": [
"### Liquid-liquid equilibrium"
]
}, {
"cell_type": "markdown", "metadata": {}, "source": [
"Liquid-liquid equilibrium can be calculated through a LLE object:"
]
}, {
"cell_type": "code", "execution_count": 17, "metadata": {}, "outputs": [
{
"data": {
"text/plain": [
"LLE(imol=MolarFlowIndexer(n", " L=[('Water', 290.7), ('Octane', 0.02063), ('Butanol', 4.3)],n", " l=[('Water', 13.35), ('Octane', 99.98), ('Butanol', 25.7)]),n", " thermal_condition=ThermalCondition(T=360.00, P=101325))"
]
}, "execution_count": 17, "metadata": {}, "output_type": "execute_result"
}
], "source": [
"tmo.settings.set_thermo(['Water', 'Octane', 'Butanol'])n", "imol = tmo.indexer.MolarFlowIndexer(n", " l=[('Water', 304), ('Butanol', 30)],n", " L=[('Octane', 100)])n", "lle = tmo.equilibrium.LLE(imol)n", "lle(T=360)n", "lle"
]
}, {
"cell_type": "markdown", "metadata": {}, "source": [
"Pressure is not a significant factor in liquid-liquid equilibrium, so only temperature is needed."

```
            ]
        }
    ], "metadata": {
        "kernelspec": {
            "display_name": "Python 3", "language": "python", "name": "python3"
        }, "language_info": {
            "codemirror_mode": {
                "name": "ipython", "version": 3
            }, "file_extension": ".py", "mimetype": "text/x-python", "name": "python", "nbconvert_exporter": "python", "pygments_lexer": "ipython3", "version": "3.7.6"
        }
    }, "nbformat": 4, "nbformat_minor": 2
}
{
    "cells": [
        {
            "cell_type": "markdown", "metadata": {}, "source": [
                "# Thermo property packages"
            ]
        }, {
            "cell_type": "markdown", "metadata": {}, "source": [
                "A [Thermo](../Thermo.txt) object defines a thermodynamic property package. To build a *Thermo* object, we must first define all the chemicals involed. In the following example, we create a property package that [BioSTEAM](https://biosteam.readthedocs.io/en/latest/) can use to model the co-production of biodiesel and ethanol from lipid-cane [[1-2]](#References).n", "n", "![](./lipidcane_areas.png "Lipid-cane diagram")"
            ]
        }, {
            "cell_type": "markdown", "metadata": {}, "source": [
                "### Creating chemicals"
            ]
        }, {
            "cell_type": "markdown", "metadata": {}, "source": [
                "We can first start by defining the common chemicals already in the data base:"
            ]
        }, {
            "cell_type": "code", "execution_count": 1, "metadata": {}, "outputs": [], "source": [
```

"import thermosteam as tmon", "Biodiesel = tmo.Chemical('Biodiesel',n", " search_ID='Methyl oleate')n", "lipidcane_chemicals = tmo.Chemicals(n", " ['Water', 'Methanol', 'Ethanol', 'Glycerol',n", " 'Glucose', 'Sucrose', 'H3PO4', 'P4O10', 'CO2',n", " 'Octane', 'O2', Biodiesel])n", "n", "(Water, Methanol, Ethanol,n", " Glycerol, Glucose, Sucrose,n", " H3PO4, P4O10, CO2, Octane, O2, Biodiesel) = lipidcane_chemicals"

], }, {

"cell_type": "markdown", "metadata": {}, "source": [

"We can assume that CO2 and O2 will always remain a gas in the process by setting the state:"

], }, {

"cell_type": "code", "execution_count": 2, "metadata": {}, "outputs": [], "source": [

"O2.at_state(phase='g')n", "CO2.at_state(phase='g')"

], }, {

"cell_type": "markdown", "metadata": {}, "source": [

"Similarly, we can assume glucose, sucrose, and phosphoric acid all remain as solids:"

], }, {

"cell_type": "code", "execution_count": 3, "metadata": {}, "outputs": [], "source": [

"H3PO4.at_state(phase='s')n", "P4O10.at_state(phase='s')n", "Glucose.at_state(phase='s')n", "Sucrose.at_state(phase='s')"

], }, {

"cell_type": "markdown", "metadata": {}, "source": [

"Now we can define the solids in the process (both soluble and insoluble). We can use the *Chemical.blank* method to create a "blank" *Chemical* object and add the thermo models ourselves:"

], }, {

"cell_type": "code", "execution_count": 4, "metadata": {}, "outputs": [], "source": [

"def create_new_chemical(ID, phase='s', **constants):n", " # Create a new solid chemical without any datan", " solid = tmo.Chemical.blank(ID, phase=phase, **constants)n", " n", " # Add chemical to the Chemicals objectn", " lipidcane_chemicals.append(solid)n", " n", " return solidn", "n", "# Cellulose and hemicellulose are modeledn", "# as their monomer minus on H2O.n", "Cellulose = create_new_chemical('Cellulose',n", " formula="C6H10O5", # Hydrolyzed glucose monomern", " Hf=-975708.8)n", "Hemicellulose = create_new_chemical('Hemicellulose',n", " formula="C5H8O5",

```
        # Hydrolyzed xylose monomern", " Hf=-761906.4)n", "Flocculant = create_new_chemical('Flocculant', MW=1.)n", "# Lignin is modeled as vanillin.n", "Lignin = create_new_chemical('Lignin',n", " formula='C8H8O3', # Vanillinn", " Hf=-452909.632)n", "Ash = create_new_chemical('Ash', MW=1.)n", "Solids = create_new_chemical('Solids', MW=1.)n", "DryYeast = create_new_chemical('DryYeast', MW=1., CAS='Yeast')n", "NaOCH3 = create_new_chemical('NaOCH3', formula='NaOCH3')n", "CaO = create_new_chemical('CaO', formula='CaO')n", "HCl = create_new_chemical('HCl', formula='HCl')n", "NaOH = create_new_chemical('NaOH', formula='NaOH')"
      ]
    }, {
      "cell_type": "markdown", "metadata": {}, "source": [
        "Note that we are still missing the lipid, modeled as Triolein. However, Triolein is not in the data bank, so let's start making it from scratch:"
      ]
    }, {
      "cell_type": "code", "execution_count": 5, "metadata": {}, "outputs": [], "source": [
        "Lipid = create_new_chemical(n", " 'Lipid',n", " phase='l',n", " Hf=-2193.7e3,n", " formula = 'C57H104O6',n", ")"
      ]
    }, {
      "cell_type": "markdown", "metadata": {}, "source": [
        "Instead of creating new models based on external sources, here we will approximate Triolein with the properties of Tripalmitin (which does exist in the data bank):"
      ]
    }, {
      "cell_type": "code", "execution_count": 6, "metadata": {}, "outputs": [], "source": [
        "Tripalmitin = tmo.Chemical('Tripalmitin').at_state(phase='l', copy=True)n", "Lipid.copy_models_from(Tripalmitin, ['V', 'sigma', 'kappa', 'Cn'])"
      ]
    }, {
      "cell_type": "markdown", "metadata": {}, "source": [
        "All what is left is to fill the chemical properties. This done through the *add_model* method of the chemical model handles. Let's begin with the solids using data from [[2-4]](#References):"
      ]
    }, {
      "cell_type": "code", "execution_count": 7, "metadata": {}, "outputs": [
        {
          "data": {
```

“text/plain”: [
“1.1”

]

}, “execution_count”: 7, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“from thermosteam import functional as fnn”, “n”, “# Assume a constant volume for lipidn”, “lipid_molar_volume = fn.rho_to_V(rho=900, MW=Lipid.MW)n”, “Lipid.V.add_model(lipid_molar_volume)n”, “n”, “# Insolubles occupy a significant volumen”, “insoluble_solids = (Ash, Cellulose, Hemicellulose,n”, ” Flocculant, Lignin, Solids, DryYeast, P4O10)n”, “for chemical in insoluble_solids:n”, ” V = fn.rho_to_V(rho=1540, MW=chemical.MW)n”, ” chemical.V.add_model(V, top_priority=True)n”, “n”, “# Solubles don’t occupy much volumen”, “soluble_solids = (CaO, HCl, NaOH, H3PO4, Sucrose, Glucose) n”, “for chemical in soluble_solids:n”, ” V = fn.rho_to_V(rho=1e5, MW=chemical.MW)n”, ” chemical.V.add_model(V, top_priority=True)n”, “n”, “n”, “# Assume sodium methoxide has some of the same properities as methanoln”, “LiquidMethanol = Methanol.at_state(phase=’l’, copy=True)n”, “NaOCH3.copy_models_from(LiquidMethanol, [‘V’, ‘sigma’,’kappa’, ‘Cn’])n”, “n”, “# Add constant models for molar heat capacity of solidsn”, “Ash.Cn.add_model(0.09 * 4.184 * Ash.MW) n”, “CaO.Cn.add_model(1.02388 * CaO.MW) n”, “Cellulose.Cn.add_model(1.364 * Cellulose.MW) n”, “Hemicellulose.Cn.add_model(1.364 * Hemicellulose.MW)n”, “Flocculant.Cn.add_model(4.184 * Flocculant.MW)n”, “Lignin.Cn.add_model(1.364 * Lignin.MW)n”, “Solids.Cn.add_model(1.100 * Solids.MW)n”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“We don’t care much about the rest of the properties (e.g. thermal conductivity), so we can default them to the values of water:”

]

}, {

“cell_type”: “code”, “execution_count”: 8, “metadata”: {}, “outputs”: [], “source”: [

“for chemical in lipidcane_chemicals: chemical.default()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Finalize the chemicals by compiling:”

]

}, {

“cell_type”: “code”, “execution_count”: 9, “metadata”: {}, “outputs”: [], “source”: [

“lipidcane_chemicals.compile()”

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"This enables methods such as *<CompiledChemicals>.array* to create chemical data ordered according to the IDs, as well as *<CompiledChemicals>.get_index* to get the index of a chemical:"

]

}, {

"cell_type": "code", "execution_count": 10, "metadata": {}, "outputs": [

{

"data": {

"text/plain": [

"array([2., 0., 2., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,n", " 0., 0., 0., 0., 0., 0., 0.])"

]

}, "execution_count": 10, "metadata": {}, "output_type": "execute_result"

}

], "source": [

"lipidcane_chemicals.array(['Water', 'Ethanol'], [2, 2])"

]

}, {

"cell_type": "code", "execution_count": 11, "metadata": {}, "outputs": [

{

"data": {

"text/plain": [

"[0, 2]"

]

}, "execution_count": 11, "metadata": {}, "output_type": "execute_result"

}

], "source": [

"lipidcane_chemicals.get_index(('Water', 'Ethanol'))"

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"After compiling, it is possible to set synonyms for indexing:"

]

}, {

"cell_type": "code", "execution_count": 12, "metadata": {}, "outputs": [

```
        {
            "name": "stdout", "output_type": "stream", "text": [
                "Watern"
            ]
        }
    ], "source": [
        "lipidcane_chemicals.set_synonym('Water', 'H2O')n",
        "print(lipidcane_chemicals.H2O)"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "### Mixture objects"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Before creating a *Thermo* object, we must define the mixing rules to calculate mixture properties. A *Mixture* object is able to calculate mixture properties through functors. In this example we will use a function to create a *Mixture* object with ideal mixing rules:"
    ]
}, {
    "cell_type": "code", "execution_count": 13, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "Mixture(n", " rule='ideal mixing', ...n", " include_excess_energies=Falsen", ")n"
            ]
        }
    ], "source": [
        "mixture = tmo.mixture.ideal_mixture(lipidcane_chemicals,n", " include_excess_energies=False)n", "mixture.show()"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "You can use the mixture for estimating mixture properties:"
    ]
}, {
    "cell_type": "code", "execution_count": 14, "metadata": {}, "outputs": [
        {
```

“data”: {

“text/plain”: [
“694.8277782515652”

]

}, “execution_count”: 14, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“array = lipidcane_chemicals.arrayn”, “mol = array([‘Water’, ‘Ethanol’], [2, 2])n”, “mixture.H(‘l’, mol, 300, 101325)”

]

}, {

“cell_type”: “code”, “execution_count”: 15, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“376.3037481383151”

]

}, “execution_count”: 15, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“mol = array([‘Water’, ‘Ethanol’], [2, 2])n”, “mixture.Cn(‘l’, mol, 300)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“You can also estimate multi-phase mixture properties through methods that start with "x" (e.g. *xCn*):”

]

}, {

“cell_type”: “code”, “execution_count”: 16, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“574.6441052202636”

]

}, “execution_count”: 16, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

"mol_liquid = array(['Water', 'Ethanol'], [2, 2])n", "mol_vapor = array(['Water', 'Ethanol'], [2, 2])n", "phase_data = [('l', mol_liquid), ('g', mol_vapor)]n", "mixture.xCn(phase_data, T=300)"

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"Note: To implement a your own Mixture object, you can request help through https://github.com/BioSTEAMDevelopmentGroup/thermosteam."

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"### Thermo objects"

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"Once the chemicals and mixture objects are finalized, we can compile them into a Thermo object:"

]

}, {

"cell_type": "code", "execution_count": 17, "metadata": {}, "outputs": [

{

"name": "stdout", "output_type": "stream", "text": [

"Thermo(n", " chemicals=CompiledChemicals([Water, Methanol, Ethanol, Glycerol, Glucose, Sucrose, H3PO4, P4O10, CO2, Octane, O2, Biodiesel, Cellulose, Hemicellulose, Flocculant, Lignin, Ash, Solids, DryYeast, NaOCH3, CaO, HCl, NaOH, Lipid]),n", " mixture=Mixture(n", " rule='ideal mixing', ...n", " include_excess_energies=Falsen", "),n", " Gamma=DortmundActivityCoefficients,n", " Phi=IdealFugacityCoefficients,n", " PCF=IdealPoyintingCorrectionFactorsn", ")n"

]

}

], "source": [

"thermo = tmo.Thermo(lipidcane_chemicals, mixture)n", "thermo.show()"

]

}, {

"cell_type": "markdown", "metadata": {}, "source": [

"Note that a Thermo object contains *ActivityCoefficients*, *FugacityCoefficients*, and *PoyintingCorrectionFactors* subclasses to define fugacity estimation methods. By default, the Dortmund modified UNIFAC method for estimating activities is selected, while ideal values for (vapor phase) fugacity coefficients and poyinting correction

```
            factors are selected. Additionally, a *Thermo* object defaults to ideal mixing rules for estimating mixture properties, and neglects excess energies in the calculation of enthalpy and entropy:"

        ]

    }, {

        "cell_type": "code", "execution_count": 18, "metadata": {}, "outputs": [

            {
                "name": "stdout", "output_type": "stream", "text": [

                    "Mixture(n", " rule='ideal mixing', ...n", " include_excess_energies=Falsen", ")n"

                ]

            }

        ], "source": [

            "thermo = tmo.Thermo(lipidcane_chemicals)n", "thermo.mixture.show()"

        ]

    }, {

        "cell_type": "markdown", "metadata": {}, "source": [

            "### References"

        ]

    }, {

        "cell_type": "markdown", "metadata": {}, "source": [

            "<a id='References'></a>n", "n", "1. Huang, H., Long, S., & Singh, V. (2016) "Techno-economic analysis of biodiesel and ethanol co-production from lipid-producing sugarcane" Biofuels, Bioproducts and Biorefining, 10(3), 299–315. https://doi.org/10.1002/bbb.1640n", "n", "2. Cortes-Peña, Y.; Kumar, D.; Singh, V.; Guest, J. S. BioSTEAM: A Fast and Flexible Platform for the Design, Simulation, and Techno-Economic Analysis of Biorefineries under Uncertainty. ACS Sustainable Chem. Eng. 2020. https://doi.org/10.1021/acssuschemeng.9b07040.n", "n", "3. Hatakeyama, T., Nakamura, K., & Hatakeyama, H. (1982). Studies on heat capacity of cellulose and lignin by differential scanning calorimetry. Polymer, 23(12), 1801–1804. https://doi.org/10.1016/0032-3861(82)90125-2n", "n", "4. Thybring, E. E. (2014). Explaining the heat capacity of wood constituents by molecular vibrations. Journal of Materials Science, 49(3), 1317–1327. https://doi.org/10.1007/s10853-013-7815-6n"

        ]

    }

], "metadata": {

    "kernelspec": {
        "display_name": "Python 3", "language": "python", "name": "python3"

    }, "language_info": {

        "codemirror_mode": {
            "name": "ipython", "version": 3
```

}, “file_extension”: “.py”, “mimetype”: “text/x-python”, “name”: “python”, “nbconvert_exporter”: “python”, “pygments_lexer”: “ipython3”, “version”: “3.7.6”

}

}, “nbformat”: 4, “nbformat_minor”: 2

}

{

“cells”: [

{

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“# Stoichiometric reactions”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Thermosteam provides array based objects that can model stoichiometric reactions given a conversion.”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Single reaction”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Create a Reaction object based on the transesterification reaction:n”, “n”, “**|Reaction|Reactant|% Converted|**n”, “**|---|—|---|**n”, “**|Lipid + 3Methanol -> 3Biodiesel + Glycerol|Lipid|90|**n”

]

}, {

“cell_type”: “code”, “execution_count”: 1, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“Reaction (by mol):n”, ” stoichiometry reactant X[%]n”, ” 3 Methanol + Lipid -> 3 Biodiesel + 1 Glycerol Lipid 90.00n”

]

}

], “source”: [

“import thermosteam as tmon”, “from biorefineries import lipidcane as lcn”, “n”, “tmo.settings.set_thermo(lc.chemicals)n”, “transesterification = tmo.Reaction(n”, ” ‘Lipid + Methanol -> Biodiesel + Glycerol’, # Reactionn”, ” correct_atomic_balance=True, # Corrects stoichiometric coefficients by atomic

```
            balancen", " reactant='Lipid', # Limiting reactantn", " X=0.9, # Conversionn", ")n",
            "transesterification.show()"
        ]
    }, {
        "cell_type": "code", "execution_count": 2, "metadata": {}, "outputs": [
            {
                "name": "stdout", "output_type": "stream", "text": [
                    "CompiledChemicals([Water, Ethanol, Glucose, Sucrose, H3PO4, P4O10,
                    CO2, Octane, O2, CH4, Ash, Cellulose, Hemicellulose, Flocculant, Lignin,
                    Solids, Yeast, CaO, Biodiesel, Methanol, Glycerol, HCl, NaOH, NaOCH3,
                    Lipid])n"
                ]
            }
        ], "source": [
            "transesterification.chemicals"
        ]
    }, {
        "cell_type": "code", "execution_count": 3, "metadata": {}, "outputs": [
            {
                "data": {
                    "text/plain": [
                        "array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,n", " 0., 0., 0., 0., 0., 3., -3.,
                        1., 0., 0., 0., -1.])"
                    ]
                }, "execution_count": 3, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
            "transesterification.stoichiometry"
        ]
    }, {
        "cell_type": "code", "execution_count": 4, "metadata": {}, "outputs": [
            {
                "data": {
                    "text/plain": [
                        "'Lipid'"
                    ]
                }, "execution_count": 4, "metadata": {}, "output_type": "execute_result"
            }
        ], "source": [
```

“transesterification.reactant”

]

}, {

“cell_type”: “code”, “execution_count”: 5, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“0.9”

]

}, “execution_count”: 5, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“transesterification.X”

]

}, {

“cell_type”: “code”, “execution_count”: 6, “metadata”: {}, “outputs”: [

{

“data”: {

“text/plain”: [
“51641.999999999585”

]

}, “execution_count”: 6, “metadata”: {}, “output_type”: “execute_result”

}

], “source”: [

“transesterification.dH # Heat of reaction J / mol-reactant (accounts for conversion); latent heats are not included”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“When a Reaction object is called with a stream, it updates the material data to reflect the reaction:”

]

}, {

“cell_type”: “code”, “execution_count”: 7, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

```
                "BEFORE REACTIONn", "Stream: s1n", " phase: 'l', T: 298.15 K, P: 101325 Pan", " flow (kmol/hr): Methanol 600n", " Lipid 100n", "AFTER REACTIONn", "Stream: s1n", " phase: 'l', T: 298.15 K, P: 101325 Pan", " flow (kmol/hr): Biodiesel 270n", " Methanol 330n", " Glycerol 90n", " Lipid 10n"

            ]

        }

    ], "source": [

        "feed = tmo.Stream(Lipid=100, Methanol=600)n", "print('BEFORE REACTION')n", "feed.show(N=100)n", "n", "# React feed molar flow raten", "transesterification(feed)n", "n", "print('AFTER REACTION')n", "feed.show(N=100)"

    ]

}, {

    "cell_type": "markdown", "metadata": {}, "source": [

        "Let's change the basis of the reaction:"

    ]

}, {

    "cell_type": "code", "execution_count": 8, "metadata": {}, "outputs": [

        {

            "name": "stdout", "output_type": "stream", "text": [

                "Reaction (by wt):n", " stoichiometry reactant X[%]n", " 0.109 Methanol + Lipid -> 1 Biodiesel + 0.104 Glycerol Lipid 90.00n"

            ]

        }

    ], "source": [

        "transesterification.basis = 'wt'n", "transesterification.show()"

    ]

}, {

    "cell_type": "markdown", "metadata": {}, "source": [

        "Notice that the stoichiometry also adjusted. If we react a stream, we should see the same result, regardless of basis:"

    ]

}, {

    "cell_type": "code", "execution_count": 9, "metadata": {}, "outputs": [

        {

            "name": "stdout", "output_type": "stream", "text": [

                "BEFORE REACTIONn", "Stream: s2n", " phase: 'l', T: 298.15 K, P: 101325 Pan", " flow (kmol/hr): Methanol 600n", " Lipid 100n", "AFTER REACTIONn", "Stream: s2n", " phase: 'l', T: 298.15 K, P: 101325 Pan", " flow (kmol/hr): Biodiesel 270n", " Methanol 330n", " Glycerol 90n", " Lipid 10n"
```

```
            ]
        }
    ], "source": [
        "feed = tmo.Stream(Lipid=100, Methanol=600)n", "print('BEFORE REACTION')n", "feed.show(N=100)n", "n", "# React feed molar flow raten", "transesterification(feed)n", "n", "print('AFTER REACTION')n", "feed.show(N=100)"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "The heat of reaction is now in units of J/kg-reactant:"
    ]
}, {
    "cell_type": "code", "execution_count": 10, "metadata": {}, "outputs": [
        {
            "data": {
              "text/plain": [
                "58.32406836499681"
              ]
            }, "execution_count": 10, "metadata": {}, "output_type": "execute_result"
        }
    ], "source": [
        "transesterification.dH # Accounts for conversion too"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Note that these reactions are carried out isothermally, but it is also possible to do so adiabatically:"
    ]
}, {
    "cell_type": "code", "execution_count": 11, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
              "BEFORE REACTIONn", "Stream: s3n", " phase: 'l', T: 298.15 K, P: 101325 Pan", " flow (kmol/hr): Water 5.55n", " Glucose 0.0555n", "AFTER REACTIONn", "Stream: s3n", " phase: 'l', T: 306.19 K, P: 101325 Pan", " flow (kmol/hr): Water 5.55n", " Ethanol 0.0999n", " Glucose 0.00555n", " CO2 0.0999n"
            ]
        }
```

], “source”: [

“feed = tmo.Stream(Glucose=10, H2O=100, units=’kg/hr’)n”, “print(‘BEFORE REACTION’)n”, “feed.show(N=100)n”, “n”, “# React feed adiabatically (temperature should increase)n”, “fermentation = tmo.Reaction(‘Glucose + O2 -> Ethanol + CO2’,n”, ” reactant=’Glucose’, X=0.9,n”, ” correct_atomic_balance=True)n”, “fermentation.adiabatic_reaction(feed)n”, “n”, “print(‘AFTER REACTION’)n”, “feed.show(N=100)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Phases changes can be included in the reaction: “

]

}, {

“cell_type”: “code”, “execution_count”: 12, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“Reaction (by mol):n”, ” stoichiometry reactant X[%]n”, ” Glucose,l -> 2 CO2,g + 2 Ethanol,l Glucose,l 90.00n”

]

}

], “source”: [

“fermentation = tmo.Reaction(‘Glucose,l -> Ethanol,l + CO2,g’,n”, ” reactant=’Glucose’, X=0.9,n”, ” correct_atomic_balance=True)n”, “fermentation.show()”

]

}, {

“cell_type”: “code”, “execution_count”: 13, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“BEFORE ADIABATIC REACTIONn”, “MultiStream: s4n”, ” phases: (‘g’, ‘l’), T: 298.15 K, P: 101325 Pan”, ” flow (kmol/hr): (l) Water 5.551n”, ” Glucose 0.05551n”, “AFTER ADIABATIC REACTIONn”, “MultiStream: s4n”, ” phases: (‘g’, ‘l’), T: 306.19 K, P: 101325 Pan”, ” flow (kmol/hr): (g) CO2 0.09991n”, ” (l) Water 5.551n”, ” Ethanol 0.09991n”, ” Glucose 0.005551n”

]

}

], “source”: [

“feed = tmo.Stream(Glucose=10, H2O=100, units=’kg/hr’)n”, “feed.phases = ‘gl’n”, “print(‘BEFORE ADIABATIC REACTION’)n”, “feed.show(N=100)n”, “n”, “# React feed adiabatically (temperature should increase)n”, “fermentation.adiabatic_reaction(feed)n”, “n”, “print(‘AFTER ADIABATIC REACTION’)n”, “feed.show(N=100)”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Lastly, when working with positive ions, simply pass a dictionary of stoichiometric coefficients instead of the equation:”

]

}, {

“cell_type”: “code”, “execution_count”: 14, “metadata”: {}, “outputs”: [

{

“name”: “stdout”, “output_type”: “stream”, “text”: [

“Reaction (by mol):n”, ” stoichiometry reactant X[%]n”, ” NaCl -> Na+ + Cl- NaCl 100.00n”

]

}

], “source”: [

“# First let’s define a new set of chemicalsn”, “chemicals = NaCl, SodiumIon, ChlorideIon = tmo.Chemicals([‘NaCl’, ‘Na+’, ‘Cl-‘])n”, “n”, “# We set the state to completely ignore other possible phasesn”, “NaCl.at_state(‘s’)n”, “SodiumIon.at_state(‘l’)n”, “ChlorideIon.at_state(‘l’)n”, “n”, “# Molar volume doesn’t matter in this scenario, but itsn”, “# required to compile the chemicals. We can assume n”, “# a very low volume since its in solution.n”, “NaCl.V.add_model(1e-6)n”, “SodiumIon.V.add_model(1e-6)n”, “ChlorideIon.V.add_model(1e-6)n”, “n”, “# We can pass a Chemicals object to not have to override n”, “# the lipidcane chemicals we set earlier.n”, “dissociation = tmo.Reaction({‘NaCl’:-1, ‘Na+’:1, ‘Cl-’: 1},n”, ” reactant=’NaCl’, X=1.,n”, ” chemicals=chemicals)n”, “dissociation.show()”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“### Parallel reactions”

]

}, {

“cell_type”: “markdown”, “metadata”: {}, “source”: [

“Model the pretreatment hydrolysis reactions and assumed conversions from Humbird et. al. as shown in the follwing table [[1]](#References):n”, “n”, “**|Reaction|Reactant|% Converted|**n”, “**|---|—|---|**n”, “**|(Glucan)n + n H2O→ n Glucose|Glucan|9.9|**n”, “**|(Glucan)n + n H2O → n Glucose Oligomer|Glucan|0.3|**n”, “**|(Glucan)n → n HMF + 2n H2O|Glucan|0.3|**n”, “**|Sucrose → HMF + Glucose + 2 H2O|Sucrose|100.0|**n”, “**|(Xylan)n + n H2O→ n Xylose|Xylan|90.0|**n”, “**|(Xylan)n + m H2O → m Xylose Oligomer|Xylan|2.4|**n”, “**|(Xylan)n → n Furfural + 2n H2O|Xylan|5.0|**n”, “**|Acetate → Acetic Acid|Acetate|100.0|**n”, “**|(Lignin)n → n Soluble Lignin|Lignin|5.0|**n”

]

}, {

```
    "cell_type": "markdown", "metadata": {}, "source": [
        "Create a ParallelReaction from Reaction objects:"
    ]
}, {
    "cell_type": "code", "execution_count": 15, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "ParallelReaction (by mol):n", "index stoichiometry reactant X[%]n", "[0] Water + Glucan -> Glucose Glucan 9.90n", "[1] Water + Glucan -> GlucoseOligomer Glucan 0.30n", "[2] Glucan -> 2 Water + HMF Glucan 0.30n", "[3] Sucrose -> 2 Water + HMF + Glucose Sucrose 0.30n", "[4] Water + Xylan -> Xylose Xylan 90.00n", "[5] Water + Xylan -> XyloseOligomer Xylan 0.24n", "[6] Xylan -> 2 Water + Furfural Xylan 0.50n", "[7] Acetate -> AceticAcid Acetate 100.00n", "[8] Lignin -> SolubleLignin Lignin 0.50n"
            ]
        }
    ], "source": [
        "from biorefineries import cornstover as csn", "n", "# Set chemicals as defined in [1-4]n", "tmo.settings.set_thermo(cs.chemicals)n", "n", "# Create reactionsn", "pretreatment_parallel_rxn = tmo.ParallelReaction([n", " # Reaction definition Reactant Conversionn", " tmo.Reaction('Glucan + H2O -> Glucose', 'Glucan', 0.0990),n", " tmo.Reaction('Glucan + H2O -> GlucoseOligomer', 'Glucan', 0.0030),n", " tmo.Reaction('Glucan -> HMF + 2 H2O', 'Glucan', 0.0030),n", " tmo.Reaction('Sucrose -> HMF + Glucose + 2H2O', 'Sucrose', 0.0030),n", " tmo.Reaction('Xylan + H2O -> Xylose', 'Xylan', 0.9000),n", " tmo.Reaction('Xylan + H2O -> XyloseOligomer', 'Xylan', 0.0024),n", " tmo.Reaction('Xylan -> Furfural + 2 H2O', 'Xylan', 0.0050),n", " tmo.Reaction('Acetate -> AceticAcid', 'Acetate', 1.0000),n", " tmo.Reaction('Lignin -> SolubleLignin', 'Lignin', 0.0050)])n", "n", "pretreatment_parallel_rxn.show()"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Model the reaction:"
    ]
}, {
    "cell_type": "code", "execution_count": 16, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "BEFORE REACTIONn", "Stream: s5n", " phase: 'l', T: 298.15 K, P: 101325 Pan", " flow (kmol/hr): Water 2.07e+05n", " Ethanol 18n", " Furfural 172n", " H2SO4 1.84e+03n", " Sucrose 1.87n", " Extract 67.8n", " Acetate 25.1n", " Ash 4.11e+03n", " Lignin 1.31e+04n", " Protein 108n", " Glucan 180n", " Xylan 123n", " Arabinan 9.02n", " Mannan 3.08n", "AFTER REACTIONNn", "Stream: s5n", " phase: 'l', T: 298.15 K, P:
```

```
                101325 Pan", " flow (kmol/hr): Water 2.07e+05n", " Ethanol 18n", " AceticAcid 25.1n", " Furfural 173n", " H2SO4 1.84e+03n", " HMF 0.546n", " Glucose 17.8n", " Xylose 111n", " Sucrose 1.86n", " Extract 67.8n", " Ash 4.11e+03n", " Lignin 1.3e+04n", " SolubleLignin 65.5n", " GlucoseOligomer 0.54n", " XyloseOligomer 0.295n", " Protein 108n", " Glucan 161n", " Xylan 11.4n", " Arabinan 9.02n", " Mannan 3.08n"
            ]
        }
    ], "source": [
        "feed = tmo.Stream(H2O=2.07e+05,n", " Ethanol=18,n", " H2SO4=1.84e+03,n", " Sucrose=1.87,n", " Extract=67.8,n", " Acetate=25.1,n", " Ash=4.11e+03,n", " Lignin=1.31e+04,n", " Protein=108,n", " Glucan=180,n", " Xylan=123,n", " Arabinan=9.02,n", " Mannan=3.08,n", " Furfural=172)n", "print('BEFORE REACTION')n", "feed.show(N=100)n", "n", "# React feed molar flow raten", "pretreatment_parallel_rxn(feed)n", "n", "print('AFTER REACTION')n", "feed.show(N=100)"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "### Reactions in series"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "SeriesReaction objects work the same way, but in series:"
    ]
}, {
    "cell_type": "code", "execution_count": 17, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "SeriesReaction (by mol):n", "index stoichiometry reactant X[%]n", "[0] Water + Glucan -> Glucose Glucan 9.90n", "[1] Water + Glucan -> GlucoseOligomer Glucan 0.30n", "[2] Glucan -> 2 Water + HMF Glucan 0.30n", "[3] Sucrose -> 2 Water + HMF + Glucose Sucrose 0.30n", "[4] Water + Xylan -> Xylose Xylan 90.00n", "[5] Water + Xylan -> XyloseOligomer Xylan 0.24n", "[6] Xylan -> 2 Water + Furfural Xylan 0.50n", "[7] Acetate -> AceticAcid Acetate 100.00n", "[8] Lignin -> SolubleLignin Lignin 0.50n"
            ]
        }
    ], "source": [
        "pretreatment_series_rxn = tmo.SeriesReaction(pretreatment_parallel_rxn)n", "pretreatment_series_rxn.show()"
    ]
```

```
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Net conversion in parallel:"
    ]
}, {
    "cell_type": "code", "execution_count": 18, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "ChemicalIndexer:n", " Sucrose 0.003n", " Acetate 1n", " Lignin 0.005n",
                " Glucan 0.105n", " Xylan 0.9074n"
            ]
        }
    ], "source": [
        "pretreatment_parallel_rxn.X_net"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Net conversion in series:"
    ]
}, {
    "cell_type": "code", "execution_count": 19, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "ChemicalIndexer:n", " Sucrose 0.003n", " Acetate 1n", " Lignin 0.005n",
                " Glucan 0.1044n", " Xylan 0.9007n"
            ]
        }
    ], "source": [
        "# Notice how the conversion isn", "# slightly lower for some reactantsn", "pretreat-
        ment_series_rxn.X_net"
    ]
}, {
    "cell_type": "code", "execution_count": 20, "metadata": {}, "outputs": [
        {
            "name": "stdout", "output_type": "stream", "text": [
                "BEFORE REACTIONn", "Stream: s6n", " phase: 'l', T: 298.15 K, P:
                101325 Pan", " flow (kmol/hr): Water 2.07e+05n", " Ethanol 18n", " Fur-
                fural 172n", " H2SO4 1.84e+03n", " Sucrose 1.87n", " Extract 67.8n", "
                Acetate 25.1n", " Ash 4.11e+03n", " Lignin 1.31e+04n", " Protein 108n",
```

```
                    " Glucan 180n", " Xylan 123n", " Arabinan 9.02n", " Mannan 3.08n",
                    "AFTER REACTIONn", "Stream: s6n", " phase: 'l', T: 298.15 K, P:
                    101325 Pan", " flow (kmol/hr): Water 2.07e+05n", " Ethanol 18n", " Aceti-
                    cAcid 25.1n", " Furfural 172n", " H2SO4 1.84e+03n", " HMF 0.491n",
                    " Glucose 17.8n", " Xylose 111n", " Sucrose 1.86n", " Extract 67.8n", "
                    Ash 4.11e+03n", " Lignin 1.3e+04n", " SolubleLignin 65.5n", " Glucose-
                    Oligomer 0.487n", " XyloseOligomer 0.0295n", " Protein 108n", " Glucan
                    161n", " Xylan 12.2n", " Arabinan 9.02n", " Mannan 3.08n"
                ]
            }
        ], "source": [
            "feed = tmo.Stream(H2O=2.07e+05,n", " Ethanol=18,n", " H2SO4=1.84e+03,n",
            " Sucrose=1.87,n", " Extract=67.8,n", " Acetate=25.1,n", " Ash=4.11e+03,n", "
            Lignin=1.31e+04,n", " Protein=108,n", " Glucan=180,n", " Xylan=123,n", " Ara-
            binan=9.02,n", " Mannan=3.08,n", " Furfural=172)n", "print('BEFORE REAC-
            TION')n", "feed.show(N=100)n", "n", "# React feed molar flow raten", "pretreat-
            ment_series_rxn(feed)n", "n", "print('AFTER REACTION')n", "feed.show(N=100)"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "### Indexing reactions"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "Both SeriesReaction, and ParallelReaction objects are indexable:"
        ]
    }, {
        "cell_type": "code", "execution_count": 21, "metadata": {}, "outputs": [
            {
                "name": "stdout", "output_type": "stream", "text": [
                    "ParallelReaction (by mol):n", "index stoichiometry reactant X[%]n", "[0]
                    Water + Glucan -> Glucose Glucan 9.90n", "[1] Water + Glucan -> Glucose-
                    Oligomer Glucan 0.30n"
                ]
            }
        ], "source": [
            "# Index a slicen", "pretreatment_parallel_rxn[0:2].show()"
        ]
    }, {
        "cell_type": "code", "execution_count": 22, "metadata": {}, "outputs": [
```

```
            {
                "name": "stdout", "output_type": "stream", "text": [
                    "ReactionItem (by mol):n", " stoichiometry reactant X[%]n", " Water + Glucan -> Glucose Glucan 9.90n"
                ]
            }
        ], "source": [
            "# Index an itemn", "pretreatment_parallel_rxn[0].show()"
        ]
    }, {
        "cell_type": "code", "execution_count": 23, "metadata": {}, "outputs": [], "source": [
            "# Change conversion through the itemn", "pretreatment_parallel_rxn[0].X = 0.10"
        ]
    }, {
        "cell_type": "code", "execution_count": 24, "metadata": {}, "outputs": [
            {
                "name": "stdout", "output_type": "stream", "text": [
                    "ParallelReaction (by mol):n", "index stoichiometry reactant X[%]n", "[0] Water + Glucan -> Glucose Glucan 10.00n", "[1] Water + Glucan -> GlucoseOligomer Glucan 0.30n", "[2] Glucan -> 2 Water + HMF Glucan 0.30n", "[3] Sucrose -> 2 Water + HMF + Glucose Sucrose 0.30n", "[4] Water + Xylan -> Xylose Xylan 90.00n", "[5] Water + Xylan -> XyloseOligomer Xylan 0.24n", "[6] Xylan -> 2 Water + Furfural Xylan 0.50n", "[7] Acetate -> AceticAcid Acetate 100.00n", "[8] Lignin -> SolubleLignin Lignin 0.50n"
                ]
            }
        ], "source": [
            "pretreatment_parallel_rxn.show()"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "Notice how changing conversion of a ReactionItem object changes the conversion in the ParallelReaction object."
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "### Referencesn", "n", "<a id='References'></a>n", "n", "1. Humbird, D., Davis, R., Tao, L., Kinchin, C., Hsu, D., Aden, A., Dudgeon, D. (2011). Process Design and Economics for Biochemical Conversion of Lignocellulosic Biomass to Ethanol: Dilute-Acid Pretreatment and Enzymatic Hydrolysis of Corn Stover (No. NREL/TP-5100-47764, 1013269). https://doi.org/10.2172/1013269n", "n", "2. Hatakeyama,
```

T., Nakamura, K., & Hatakeyama, H. (1982). Studies on heat capacity of cellulose and lignin by differential scanning calorimetry. Polymer, 23(12), 1801–1804. https://doi.org/10.1016/0032-3861(82)90125-2n", "n", "3. Thybring, E. E. (2014). Explaining the heat capacity of wood constituents by molecular vibrations. Journal of Materials Science, 49(3), 1317–1327. https://doi.org/10.1007/s10853-013-7815-6n", "n", "4. Murphy W. K., and K. R. Masters. (1978). Gross heat of combustion of northern red oak (Quercus rubra) chemical components. Wood Sci. 10:139-141."

]

}

], "metadata": {

"kernelspec": {
"display_name": "Python 3", "language": "python", "name": "python3"

}, "language_info": {

"codemirror_mode": {
"name": "ipython", "version": 3

}, "file_extension": ".py", "mimetype": "text/x-python", "name": "python", "nbconvert_exporter": "python", "pygments_lexer": "ipython3", "version": "3.7.6"

}

}, "nbformat": 4, "nbformat_minor": 2

}

1.3 Chemical

class thermosteam.**Chemical**(*ID*, *cache=None*, ***, *search_ID=None*, *eos=None*, *phase_ref=None*, *CAS=None*, *default=False*, *phase=None*, *search_db=True*, *V=None*, *Cn=None*, *mu=None*, *Cp=None*, *rho=None*, *sigma=None*, *kappa=None*, *epsilon=None*, *Psat=None*, *Hvap=None*, ***data*)

Creates a Chemical object which contains constant chemical properties, as well as thermodynamic and transport properties as a function of temperature and pressure.

Parameters

- **ID** (*str*) –

  One of the following [-]:

  - Name, in IUPAC form or common form or a synonym registered in PubChem
  - InChI name, prefixed by 'InChI=1S/' or 'InChI=1/'
  - InChI key, prefixed by 'InChIKey='
  - PubChem CID, prefixed by 'PubChem='
  - SMILES (prefix with 'SMILES=' to ensure smiles parsing)
  - CAS number

- **cache** (*optional*) – Whether or not to use cached chemicals and cache new chemicals.

- **search_ID** (*str, optional*) – ID to search through database. Pass this key-word argument when you'd like to give a custom name to the chemical, but cannot find the chemical with that name.
- **eos** (*GCEOS subclass, optional*) – Equation of state class for solving thermodynamic properties. Defaults to Peng-Robinson.
- **phase_ref** (*{'s', 'l', 'g'}, optional*) – Reference phase of chemical.
- **CAS** (*str, optional*) – CAS number of chemical.
- **phase** (*{'s', 'l' or 'g'}, optional*) – Phase to set state of chemical.
- **search_db=True** (*bool, optional*) – Whether to search the data base for chemical.
- **Cn** (*float or function(T), optional*) – Molar heat capacity model [J/mol] as a function of temperature [K].
- **sigma** (*float or function(T), optional*) – Surface tension model [N/m] as a function of temperature [K].
- **epsilon** (*float or function(T), optional*) – Relative permitivity model [-] as a function of temperature [K].
- **Psat** (*float or function(T), optional*) – Vapor pressure model [N/m] as a function of temperature [K].
- **Hvap** (*float or function(T), optional*) – Heat of vaporization model [J/mol] as a function of temperature [K].
- **V** (*float or function(T, P), optional*) – Molar volume model [mol/m3] as a function of temperature [K] and pressure [Pa].
- **mu** (*float or function(T, P), optional*) – Dynamic viscosity model [Pa*s] as a function of temperature [K] and pressure [Pa].
- **kappa** (*float or function(T, P), optional*) – Thermal conductivity model [W/m/K] as a function of temperature [K] and pressure [Pa].
- **Cp** (*float, optional*) – Constant heat capacity model [J/g].
- **rho** (*float, optional*) – Constant density model [kg/m3].
- **default=False** (*bool, optional*) – Whether to default any missing chemical properties such as molar volume, heat capacity, surface tension, thermal conductivity, and molecular weight to that of water (on a weight basis).
- ****data** (*float or str*) – User data (e.g. Tb, formula, etc.).

Examples

Chemical objects contain pure component properties:

```
>>> import thermosteam as tmo
>>> # Initialize with an identifier
>>> # (e.g. by name, CAS, InChI...)
>>> Water = tmo.Chemical('Water')
>>> Water.show()
Chemical: Water (phase_ref='l')
[Names]  CAS: 7732-18-5
         InChI: H2O/h1H2
```

(continues on next page)

(continued from previous page)

```
         InChI_key: XLYOFNOQVPJJNP-U...
         common_name: water
         iupac_name: ('oxidane',)
         pubchemid: 962
         smiles: O
         formula: H2O
[Groups] Dortmund: <1H2O>
         UNIFAC: <1H2O>
         PSRK: <1H2O>
[Data]   MW: 18.015 g/mol
         Tm: 273.15 K
         Tb: 373.12 K
         Tt: 273.15 K
         Tc: 647.14 K
         Pt: 610.88 Pa
         Pc: 2.2048e+07 Pa
         Vc: 5.6e-05 m^3/mol
         Hf: -2.8582e+05 J/mol
         S0: 70 J/K/mol
         LHV: 44011 J/mol
         HHV: 0 J/mol
         Hfus: 6010 J/mol
         Sfus: None
         omega: 0.344
         dipole: 1.85 Debye
         similarity_variable: 0.16653
         iscyclic_aliphatic: 0
         combustion: {'H2O': 1.0}
```

All fields shown are accessible:

```
>>> Water.CAS
'7732-18-5'
```

Functional group identifiers (e.g. *Dortmund*, *UNIFAC*, *PSRK*) allow for the estimation of activity coefficients through group contribution methods. In other words, these attributes define the functional groups for thermodynamic equilibrium calculations:

```
>>> Water.Dortmund
<DortmundGroupCounts: 1H2O>
```

Temperature (in Kelvin) and pressure (in Pascal) dependent properties can be computed:

```
>>> # Vapor pressure (Pa)
>>> Water.Psat(T=373.15)
101284.55...
>>> # Surface tension (N/m)
>>> Water.sigma(T=298.15)
0.07197220523...
>>> # Molar volume (m^3/mol)
>>> Water.V(phase='l', T=298.15, P=101325)
1.80692...e-05
```

Note that the reference state of all chemicals is 25 degC and 1 atm:

```
>>> (Water.T_ref, Water.P_ref)
(298.15, 101325.0)
>>> # Enthalpy at reference conditions (J/mol; without excess energies)
>>> Water.H(T=298.15, phase='l')
0.0
```

Constant pure component properties are also available:

```
>>> # Molecular weight (g/mol)
>>> Water.MW
18.01528
>>> # Boiling point (K)
>>> Water.Tb
373.124
```

Temperature dependent properties are managed by model handles:

```
>>> Water.Psat.show()
TDependentModelHandle(T, P=None) -> Psat [Pa]
[0] Wagner original
[1] Antoine
[2] EQ101
[3] Wagner
[4] boiling critical relation
[5] Lee Kesler
[6] Ambrose Walton
[7] Sanjari
[8] Edalat
```

Phase dependent properties have attributes with model handles for each phase:

```
>>> Water.V
<PhaseTPHandle(phase, T, P) -> V [m^3/mol]>
>>> (Water.V.l, Water.V.g)
(<TPDependentModelHandle(T, P) -> V.l [m^3/mol]>, <TPDependentModelHandle(T, P) ->␣
↪V.g [m^3/mol]>)
```

A model handle contains a series of models applicable to a certain domain:

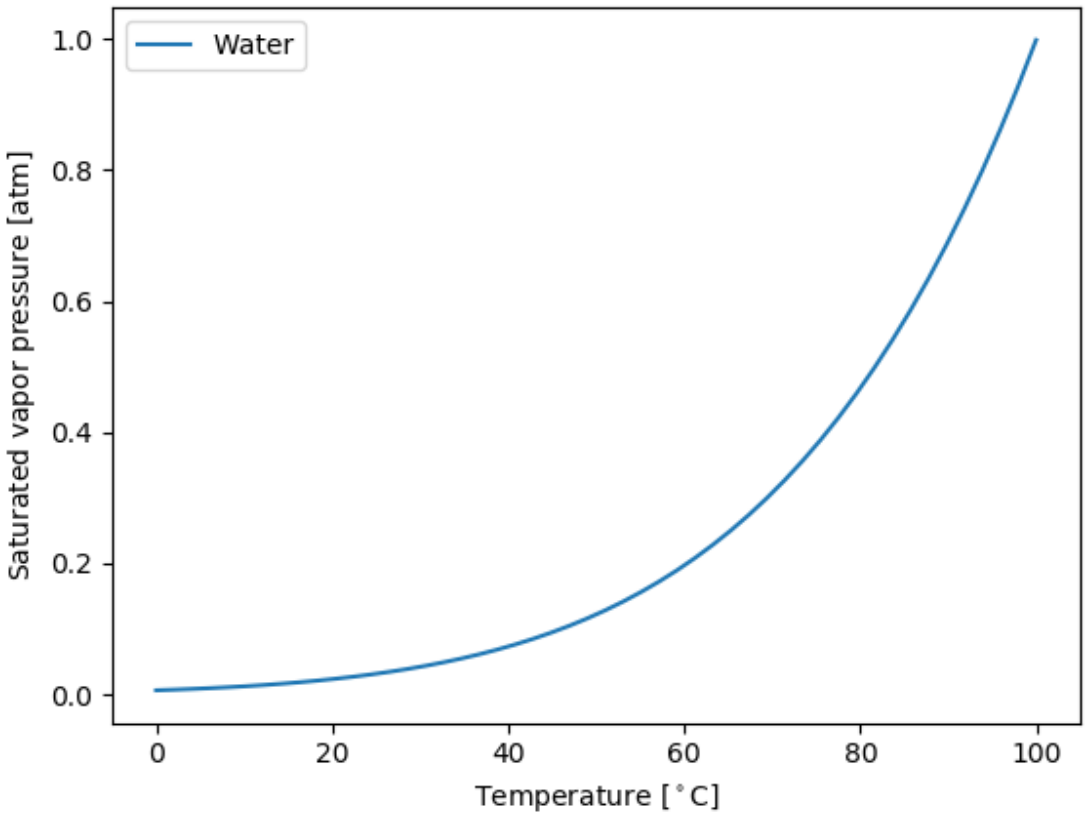
```
>>> Water.Psat[0].show()
TDependentModel(T, P=None) -> Psat [Pa]
 name: Wagner original
 Tmin: 275 K
 Tmax: 647.35 K
```

When called, the model handle searches through each model until it finds one with an applicable domain. If none are applicable, a value error is raised:

```
>>> Water.Psat(373.15)
101284.55179999319
>>> # Water.Psat(1000.0) ->
>>> # ValueError: Water has no valid saturated vapor pressure model at T=1000.00 K
```

Model handles as well as the models themselves have tabulation and plotting methods to help visualize how properties depend on temperature and pressure.

```
>>> # import matplotlib.pyplot as plt
>>> # Water.Psat.plot_vs_T([Water.Tm, Water.Tb], 'degC', 'atm', label="Water")
>>> # plt.show()
```

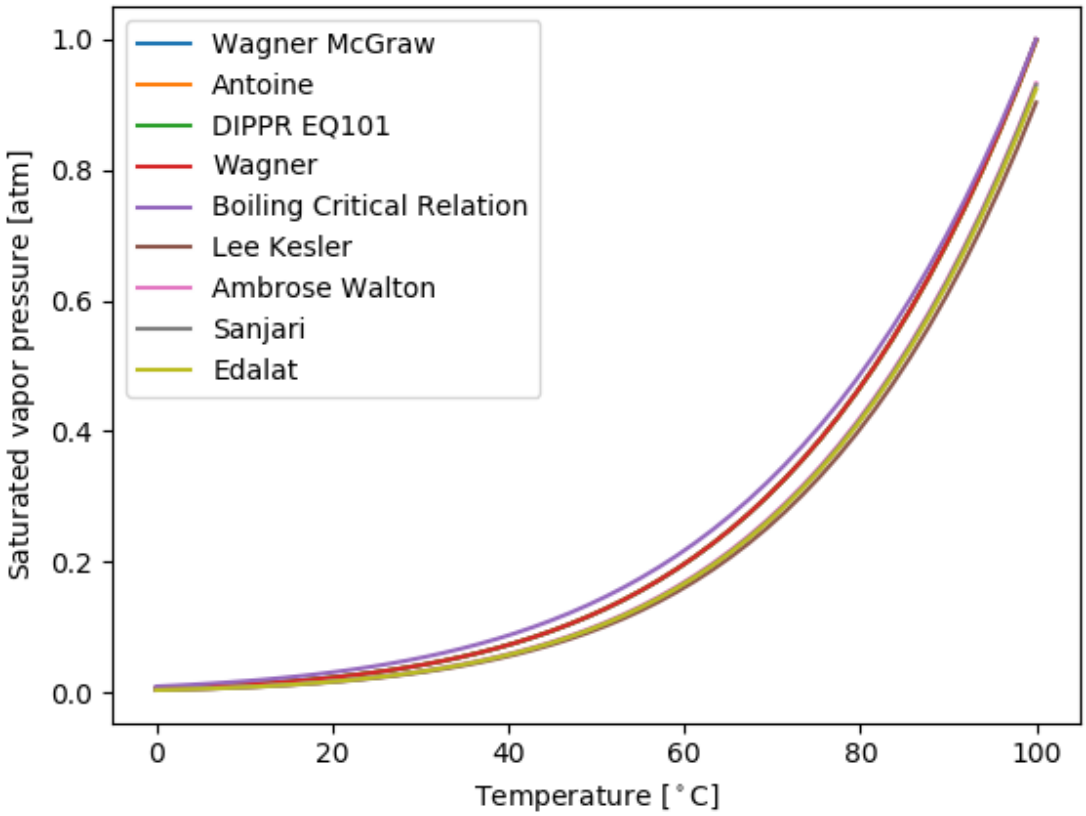



```
>>> # Plot all models
```

```
>>> # Water.Psat.plot_models_vs_T([Water.Tm, Water.Tb], 'degC', 'atm')
>>> # plt.show()
```

Each model may contain either a function or a functor (a function with stored data) to compute the property:

```
>>> functor = Water.Psat[0].evaluate
>>> functor.show()
Functor: Wagner_original(T, P=None) -> Psat [Pa]
 Tc: 647.35 K
 Pc: 2.2122e+07 Pa
 a: -7.7645
 b: 1.4584
 c: -2.7758
 d: -1.233
```


Wagner McGraw
Antoine
DIPPR EQ101
Wagner
Boiling Critical Relation
Lee Kesler
Ambrose Walton
Sanjari
Edalat
Saturated vapor pressure [atm]
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Temperature [°C]

Note: All chemical property functors are available in the thermosteam.properties subpackage. You can also use help(<functor>) for further information on the math and equations used in the functor.

A new model can be added easily to a model handle through the *add_model* method, for example:

```
>>> # Set top_priority=True to place model in postion [0]
>>> @Water.Psat.add_model(Tmin=273.20, Tmax=473.20, top_priority=True)
... def User_antoine_model(T):
...     return 10.0**(10.116 -  1687.537 / (T - 42.98))
>>> Water.Psat.show()
TDependentModelHandle(T, P=None) -> Psat [Pa]
[0] User antoine model
[1] Wagner original
[2] Antoine
[3] EQ101
[4] Wagner
[5] boiling critical relation
[6] Lee Kesler
[7] Ambrose Walton
[8] Sanjari
[9] Edalat
```

The *add_model* method is a high level interface that even lets you create a constant model:

```
>>> value = Water.V.l.add_model(1.687e-05, name='User constant')
... # Model is appended at the end by default
>>> added_model = Water.V.l[-1]
>>> added_model.show()
ConstantThermoModel(T=None, P=None) -> V.l [m^3/mol]
 name: User constant
 value: 1.687e-05
 Tmin: 0 K
 Tmax: inf K
 Pmin: 0 Pa
 Pmax: inf Pa
```

Note: Because no bounds were given, the model assumes it is valid across all temperatures and pressures.

Manage the model order with the *set_model_priority* and *move_up_model_priority* methods:

```
>>> # Note: In this case, we pass the model name, but its
>>> # also possible to pass the current index, or the model itself.
>>> Water.Psat.move_up_model_priority('Wagner original')
>>> Water.Psat.show()
TDependentModelHandle(T, P=None) -> Psat [Pa]
[0] Wagner original
[1] Antoine
[2] EQ101
[3] Wagner
[4] boiling critical relation
[5] Lee Kesler
```

(continues on next page)

(continued from previous page)

```
[6] Ambrose Walton
[7] Sanjari
[8] Edalat
[9] User antoine model
```

```
>>> Water.Psat.set_model_priority('Antoine')
>>> Water.Psat.show()
TDependentModelHandle(T, P=None) -> Psat [Pa]
[0] Antoine
[1] Wagner original
[2] EQ101
[3] Wagner
[4] boiling critical relation
[5] Lee Kesler
[6] Ambrose Walton
[7] Sanjari
[8] Edalat
[9] User antoine model
```

The default priority is *0* (or top priority), but you can choose any priority:

```
>>> Water.Psat.set_model_priority('Antoine', 1)
>>> Water.Psat.show()
TDependentModelHandle(T, P=None) -> Psat [Pa]
[0] Wagner original
[1] Antoine
[2] EQ101
[3] Wagner
[4] boiling critical relation
[5] Lee Kesler
[6] Ambrose Walton
[7] Sanjari
[8] Edalat
[9] User antoine model
```

Note: All models are stored as a deque in the *models* attribute (e.g. Water.Psat.models).

---

mu(*phase*, *T*, *P*)

Dynamic viscosity [Pa*s].

kappa(*phase*, *T*, *P*)

Thermal conductivity [W/m/K].

V(*phase*, *T*, *P*)

Molar volume [m^3/mol].

Cn(*phase*, *T*)

Molar heat capacity [J/mol/K].

Psat(*T*)

Vapor pressure [Pa].

Hvap(*T*)

Heat of vaporization [J/mol]

sigma(*T*)

Surface tension [N/m].

epsilon(*T*)

Relative permitivity [-]

S(*phase*, *T*, *P*)

Entropy [J/mol].

H(*phase*, *T*)

Enthalpy [J/mol].

S_excess(*T*, *P*)

Excess entropy [J/mol].

H_excess(*T*, *P*)

Excess enthalpy [J/mol].

T_ref = 298.15

[float] Reference temperature in Kelvin

P_ref = 101325.0

[float] Reference pressure in Pascal

H_ref = 0.0

[float] Reference enthalpy in J/mol

chemical_cache = {}

dict[str, Chemical] Cached chemicals

cache = False

[bool] Wheather or not to search cache by default

classmethod new(*ID*, *CAS*, *eos=<class 'thermosteam.eos.PR'>*, *phase_ref=None*, *phase=None*, ***data*)

Create a new chemical from data without searching through the data base, and load all possible models from given data.

classmethod blank(*ID*, *CAS=None*, *phase_ref=None*, *phase=None*, *formula=None*, ***data*)

Return a new Chemical object without any thermodynamic models or data (unless provided).

Parameters

- **ID** (*str*) – Chemical identifier.
- **CAS** (*str, optional*) – CAS number. If none provided, it defaults to the *ID*.
- **phase_ref** (*str, optional*) – Phase at the reference state (T=298.15, P=101325).
- **phase** (*str, optional*) – Phase to set state as a single phase chemical.
- ****data** – Any data to fill chemical with.

Examples

```
>>> from thermosteam import Chemical
>>> Substance = Chemical.blank('Substance')
>>> Substance.show()
Chemical: Substance (phase_ref=None)
[Names]  CAS: Substance
         InChI: None
         InChI_key: None
         common_name: None
         iupac_name: None
         pubchemid: None
         smiles: None
         formula: None
[Groups] Dortmund: <Empty>
         UNIFAC: <Empty>
         PSRK: <Empty>
[Data]   MW: None
         Tm: None
         Tb: None
         Tt: None
         Tc: None
         Pt: None
         Pc: None
         Vc: None
         Hf: None
         S0: None
         LHV: None
         HHV: None
         Hfus: None
         Sfus: None
         omega: None
         dipole: None
         similarity_variable: None
         iscyclic_aliphatic: None
         combustion: None
```

copy(*ID*, *CAS=None*, ***data*)

Return a copy of the chemical with a new ID.

Examples

```
>>> import thermosteam as tmo
>>> Glucose = tmo.Chemical('Glucose')
>>> Mannose = Glucose.copy('Mannose')
>>> Mannose.show()
Chemical: Mannose (phase_ref='l')
[Names]  CAS: Mannose
         InChI: C6H12O6/c7-1-3(9)5(1...
         InChI_key: GZCGUPFRVQAUEE-S...
         common_name: D-Glucose
         iupac_name: ('(2R,3S,4R,5R)...
```

(continues on next page)

(continued from previous page)

```
         pubchemid: 1.0753e+05
         smiles: O=C[C@H](O)[C@@H](O...
         formula: C6H12O6
[Groups] Dortmund: {2: 1, 3: 4, 14: ...
         UNIFAC: {2: 1, 3: 4, 14: 5,...
         PSRK: {2: 1, 3: 4, 14: 5, 2...
[Data]   MW: 180.16 g/mol
         Tm: None
         Tb: 616.29 K
         Tt: None
         Tc: 755 K
         Pt: None
         Pc: 4.82e+06 Pa
         Vc: 0.000414 m^3/mol
         Hf: -1.2711e+06 J/mol
         S0: 0 J/K/mol
         LHV: -2.5406e+06 J/mol
         HHV: -2.8047e+06 J/mol
         Hfus: 0 J/mol
         Sfus: None
         omega: 2.387
         dipole: None
         similarity_variable: 0.13322
         iscyclic_aliphatic: 0
         combustion: {'CO2': 6, 'O2'...
```

property phase_ref

{'s', 'l', 'g'} Phase at 298 K and 101325 Pa.

property ID

[str] Identification of chemical.

property CAS

[str] CAS number of chemical.

property InChI

[str] IUPAC International Chemical Identifier.

property InChI_key

[str] IUPAC International Chemical Identifier shorthand.

property common_name

[str] Common name identifier.

property iupac_name

[str] Standard name as defined by IUPAC.

property pubchemid

[str] Chemical identifier as defined by PubMed.

property smiles

[str] Chemical SMILES formula.

property formula

[str] Chemical atomic formula.

property Dortmund

[DortmundGroupCounts] Dictionary-like object with functional group numerical identifiers as keys and the number of groups as values.

property UNIFAC

[UNIFACGroupCounts] Dictionary-like object with functional group numerical identifiers as keys and the number of groups as values.

property PSRK

[PSRKGroupCounts] Dictionary-like object with functional group numerical identifiers as keys and the number of groups as values.

property eos

[object] Instance for solving equations of state.

property eos_1atm

[object] Instance for solving equations of state at 1 atm.

property mu

Dynamic viscosity [Pa*s].

property kappa

Thermal conductivity [W/m/K].

property V

Molar volume [m^3/mol].

property Cn

Molar heat capacity [J/mol/K].

property Psat

Vapor pressure [Pa].

property Hvap

Heat of vaporization [J/mol].

property sigma

Surface tension [N/m].

property epsilon

Relative permitivity [-].

property S

Entropy [J/mol].

property H

Enthalpy [J/mol].

property S_excess

Excess entropy [J/mol].

property H_excess

Excess enthalpy [J/mol].

property MW

Molecular weight [g/mol].

property Tm

Normal melting temperature [K].

property Tb

Normal boiling temperature [K].

property Pt

Triple point pressure [Pa].

property Tt

Triple point temperature [K].

property Tc

Critical point temperature [K].

property Pc

Critical point pressure [Pa].

property Vc

Critical point molar volume [m^3/mol].

property Hfus

Heat of fusion [J/mol].

property Sfus

Entropy of fusion [J/mol].

property omega

Accentric factor [-].

property dipole

Dipole moment [Debye].

property similarity_variable

Similarity variable [-].

property iscyclic_aliphatic

Whether the chemical is cyclic-aliphatic.

property Hf

Heat of formation at reference phase and temperature [J/mol].

property LHV

Lower heating value [J/mol].

property HHV

Higher heating value [J/mol].

property combustion

dict[str, int] Combustion reaction.

property Stiel_Polar

[float] Stiel Polar factor for computing surface tension.

property Zc

[float] Compressibility factor.

property has_hydroxyl

[bool] Whether or not chemical contains a hydroxyl functional group, as determined by the Dortmund/UNIFAC/PSRK functional groups.

property atoms

int] Atom-count pairs based on formula.

Type
dict[str

get_combustion_reaction(*chemicals=None*)

Return a Reaction object defining the combustion of this chemical. If no combustion stoichiometry is available, return None.

get_phase(*T=298.15*, *P=101325.0*)

Return phase of chemical at given thermal condition.

Examples

```
>>> from thermosteam import Chemical
>>> Water = Chemical('Water', cache=True)
>>> Water.get_phase(T=400, P=101325)
'g'
```

Tsat(*P*, *Tguess=None*, *Tmin=None*, *Tmax=None*)

Return the saturated temperature (in Kelvin) given the pressure (in Pascal).

reset(*CAS*, *eos=<class 'thermosteam.eos.PR'>*, *phase_ref=None*, *smiles=None*, *InChI=None*, *InChI_key=None*, *pubchemid=None*, *iupac_name=None*, *common_name=None*, *formula=None*, *MW=None*, *Tm=None*, *Tb=None*, *Tc=None*, *Pc=None*, *Vc=None*, *omega=None*, *Tt=None*, *Pt=None*, *Hf=None*, *S0=None*, *LHV=None*, *combustion=None*, *HHV=None*, *Hfus=None*, *dipole=None*, *similarity_variable=None*, *iscyclic_aliphatic=None*, ***, *metadata=None*, *phase=None*)

Reset all chemical properties.

Parameters

- **CAS** (*str*) – CAS number of chemical to load.
- **eos** (*optional*) – Equation of state. The default is Peng Robinson.
- **phase_ref** (*str, optional*) – Phase reference. Defaults to the phase at 298.15 K and 101325 Pa.

reset_combustion_data(*method='Stoichiometry'*)

Reset combustion data (LHV, HHV, and combution attributes) based on the molecular formula and the heat of formation.

reset_free_energies()

Reset the *H*, *S*, *H_excess*, and *S_excess* functors.

get_key_property_names()

Return the attribute names of key properties required to model a process.

default(*properties=None*)

Default all *properties* with the chemical properties of water. If no *properties* given, all essential chemical properties that are missing are defaulted. *properties* which are still missing are returned as set.

Parameters
properties (*Iterable[str], optional*) – Names of chemical properties to default.

Returns
missing_properties – Names of chemical properties that are still missing.

Return type
set[str]

Examples

```
>>> from thermosteam import Chemical
>>> Substance = Chemical.blank('Substance')
>>> missing_properties = Substance.default()
>>> sorted(missing_properties)
['Dortmund', 'Hfus', 'Hvap', 'PSRK', 'Pc', 'Psat', 'Pt', 'Sfus', 'Tb', 'Tc',
↪'Tm', 'Tt', 'UNIFAC', 'V', 'Vc', 'dipole', 'iscyclic_aliphatic', 'omega',
↪'similarity_variable']
```

Note that missing properties does not include essential properties volume, heat capacity, and conductivity.

get_missing_properties(*properties=None*)

Return a list all missing thermodynamic properties.

Examples

```
>>> from thermosteam import Chemical
>>> Substance = Chemical.blank('Substance', phase_ref='l')
>>> sorted(Substance.get_missing_properties())
['Cn', 'Dortmund', 'H', 'HHV', 'H_excess', 'Hf', 'Hfus', 'Hvap', 'LHV', 'MW',
↪'PSRK', 'Pc', 'Psat', 'Pt', 'S', 'S_excess', 'Sfus', 'Tb', 'Tc', 'Tm', 'Tt',
↪'UNIFAC', 'V', 'Vc', 'combustion', 'dipole', 'epsilon', 'iscyclic_aliphatic',
↪ 'kappa', 'mu', 'omega', 'sigma', 'similarity_variable']
```

copy_models_from(*other*, *names=None*)

Copy models from other.

property locked_state

[str] Constant phase of chemical.

at_state(*phase*, *copy=False*)

Set the state of chemical.

Examples

```
>>> from thermosteam import Chemical
>>> N2 = Chemical('N2')
>>> N2.at_state(phase='g')
>>> N2.H(298.15) # No longer a function of phase
0.0
```

show()

Print all specifications

1.4 Chemicals

class thermosteam.**Chemicals**(*chemicals*, *cache=False*)

Create a Chemicals object that contains Chemical objects as attributes.

Parameters

- **chemicals** (*Iterable[str or Chemical]*) –

  Strings should be one of the following [-]:

  - Name, in IUPAC form or common form or a synonym registered in PubChem
  - InChI name, prefixed by 'InChI=1S/' or 'InChI=1/'
  - InChI key, prefixed by 'InChIKey='
  - PubChem CID, prefixed by 'PubChem='
  - SMILES (prefix with 'SMILES=' to ensure smiles parsing)
  - CAS number

- **cache** (*bool, optional*) – Wheather or not to use cached chemicals.

Examples

Create a Chemicals object from chemical identifiers:

```
>>> from thermosteam import Chemicals
>>> chemicals = Chemicals(['Water', 'Ethanol'], cache=True)
>>> chemicals
Chemicals([Water, Ethanol])
```

All chemicals are stored as attributes:

```
>>> chemicals.Water, chemicals.Ethanol
(Chemical('Water'), Chemical('Ethanol'))
```

Chemicals can also be accessed as items:

```
>>> chemicals = Chemicals(['Water', 'Ethanol', 'Propane'], cache=True)
>>> chemicals['Ethanol']
Chemical('Ethanol')
>>> chemicals['Propane', 'Water']
[Chemical('Propane'), Chemical('Water')]
```

A Chemicals object can be extended with more chemicals:

```
>>> from thermosteam import Chemical
>>> Methanol = Chemical('Methanol')
>>> chemicals.append(Methanol)
>>> chemicals
Chemicals([Water, Ethanol, Propane, Methanol])
>>> new_chemicals = Chemicals(['Hexane', 'Octanol'], cache=True)
>>> chemicals.extend(new_chemicals)
>>> chemicals
Chemicals([Water, Ethanol, Propane, Methanol, Hexane, Octanol])
```

Chemical objects cannot be repeated:

```
>>> chemicals.append(chemicals.Water)
Traceback (most recent call last):
ValueError: Water already defined in chemicals
>>> chemicals.extend(chemicals['Ethanol', 'Octanol'])
Traceback (most recent call last):
ValueError: Ethanol already defined in chemicals
```

A Chemicals object can only contain Chemical objects:

```
>>> chemicals.append(10)
Traceback (most recent call last):
TypeError: only 'Chemical' objects can be appended, not 'int'
```

You can check whether a Chemicals object contains a given chemical:

```
>>> 'Water' in chemicals
True
>>> chemicals.Water in chemicals
True
>>> 'Butane' in chemicals
False
```

An attempt to access a non-existent chemical raises an UndefinedChemical error:

```
>>> chemicals['Butane']
Traceback (most recent call last):
UndefinedChemical: 'Butane'
```

copy()

Return a copy.

append(*chemical*)

Append a Chemical.

extend(*chemicals*)

Extend with more Chemical objects.

subgroup(*IDs*)

Create a new subgroup of chemicals.

Parameters

IDs (*Iterable[str]*) – Chemical identifiers.

Examples

```
>>> chemicals = Chemicals(['Water', 'Ethanol', 'Propane'])
>>> chemicals.subgroup(['Propane', 'Water'])
Chemicals([Propane, Water])
```

compile(*skip_checks=False*)

Cast as a CompiledChemicals object.

Parameters

skip_checks (*bool, optional*) – Whether to skip checks for missing or invalid properties.

> **Warning:** If checks are skipped, certain features in thermosteam (e.g. phase equilibrium) cannot be guaranteed to function properly.

Examples

Compile ethanol and water chemicals:

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol'])
>>> chemicals.compile()
>>> chemicals
CompiledChemicals([Water, Ethanol])
```

Attempt to compile chemicals with missing properties:

```
>>> Substance = tmo.Chemical('Substance', search_db=False)
>>> chemicals = tmo.Chemicals([Substance])
>>> chemicals.compile()
Traceback (most recent call last):
RuntimeError: Substance is missing key thermodynamic properties
(V, S, H, Cn, Psat, Tb and Hvap); use the `<Chemical>.get_missing_properties()`
to check all missing properties
```

Compile chemicals with missing properties (skipping checks) and note how certain features do not work:

```
>>> chemicals.compile(skip_checks=True)
>>> tmo.settings.set_thermo(chemicals)
>>> s = tmo.Stream('s', Substance=10)
>>> s.rho
Traceback (most recent call last):
DomainError: Substance (CAS: Substance) has no valid liquid molar
volume model at T=298.15 K and P=101325 Pa
```

class thermosteam.**CompiledChemicals**(*chemicals*, *cache=None*)

Create a CompiledChemicals object that contains Chemical objects as attributes.

Parameters

- **chemicals** (*Iterable[str or Chemical]*) –

  Strings should be one of the following [-]:

  - Name, in IUPAC form or common form or a synonym registered in PubChem
  - InChI name, prefixed by 'InChI=1S/' or 'InChI=1/'
  - InChI key, prefixed by 'InChIKey='
  - PubChem CID, prefixed by 'PubChem='
  - SMILES (prefix with 'SMILES=' to ensure smiles parsing)
  - CAS number

- **cache** (*optional*) – Whether or not to use cached chemicals.

tuple

All compiled chemicals.

Type
tuple[*Chemical*]

size

Number of chemicals.

Type
int

IDs

IDs of all chemicals.

Type
tuple[str]

CASs

CASs of all chemicals

Type
tuple[str]

MW

MWs of all chemicals.

Type
1d ndarray

Hf

Heats of formation of all chemicals.

Type
1d ndarray

Hc

Heats of combustion of all chemicals.

Type
1d ndarray

vle_chemicals

Chemicals that may have vapor and liquid phases.

Type
tuple[*Chemical*]

lle_chemicals

Chemicals that may have two liquid phases.

Type
tuple[*Chemical*]

heavy_chemicals

Chemicals that are only present in liquid or solid phases.

Type
tuple[*Chemical*]

light_chemicals

IDs of chemicals that are only present in gas phases.

Type

tuple[*Chemical*]

Examples

Create a CompiledChemicals object from chemical identifiers

```
>>> from thermosteam import CompiledChemicals, Chemical
>>> chemicals = CompiledChemicals(['Water', 'Ethanol'], cache=True)
>>> chemicals
CompiledChemicals([Water, Ethanol])
```

All chemicals are stored as attributes:

```
>>> chemicals.Water, chemicals.Ethanol
(Chemical('Water'), Chemical('Ethanol'))
```

Note that because they are compiled, the append and extend methods do not work:

```
>>> Propane = Chemical('Propane', cache=True)
>>> chemicals.append(Propane)
Traceback (most recent call last):
TypeError: 'CompiledChemicals' object is read-only
```

You can check whether a Chemicals object contains a given chemical:

```
>>> 'Water' in chemicals
True
>>> chemicals.Water in chemicals
True
>>> 'Butane' in chemicals
False
```

compile()

Do nothing, CompiledChemicals objects are already compiled.

refresh_constants()

Refresh constant arrays according to their chemical values, including the molecular weight, heats of formation, and heats of combustion.

get_combustion_reactions()

Return a ParallelReactions object with all defined combustion reactions.

Examples

```
>>> chemicals = CompiledChemicals(['H2O', 'Methanol', 'Ethanol', 'CO2', 'O2'],␣
↪cache=True)
>>> rxns = chemicals.get_combustion_reactions()
>>> rxns.show()
ParallelReaction (by mol):
index  stoichiometry                     reactant    X[%]
[0]    Methanol + 1.5 O2 -> 2 H2O + CO2  Methanol  100.00
[1]    Ethanol + 3 O2 -> 3 H2O + 2 CO2   Ethanol   100.00
```

property formula_array

An array describing the formulas of all chemicals. Each column is a chemical and each row an element. Rows are ordered by atomic number.

Examples

```
>>> chemicals = CompiledChemicals(['Water', 'Ethanol', 'Propane'], cache=True)
>>> chemicals.formula_array
array([[2., 6., 8.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 2., 3.],
       [0., 0., 0.],
       [1., 1., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
```

(continues on next page)

(continued from previous page)

```
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
```

(continues on next page)

(continued from previous page)

```
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

`subgroup`(*IDs*)

Create a new subgroup of chemicals.

Parameters

`IDs` (*`Iterable[str]`*) – Chemical identifiers.

Examples

```
>>> chemicals = CompiledChemicals(['Water', 'Ethanol', 'Propane'], cache=True)
>>> chemicals.subgroup(['Propane', 'Water'])
CompiledChemicals([Propane, Water])
```

`get_synonyms`(*ID*)

Get all synonyms of a chemical.

Parameters

`ID` (*`str`*) – Chemical identifier.

Examples

Get all synonyms of water:

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water'], cache=True)
>>> chemicals.get_synonyms('Water')
['7732-18-5', 'Water']
```

set_synonym(*ID*, *synonym*)

Set a new synonym for a chemical.

Parameters

- **ID** (*str*) – Chemical identifier.
- **synonym** (*str*) – New identifier for chemical.

Examples

Set new synonym for water:

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Ethanol'], cache=True)
>>> chemicals.set_synonym('Water', 'H2O')
>>> chemicals.H2O is chemicals.Water
True
```

Note that you cannot use one synonym for two chemicals:

```
>>> chemicals.set_synonym('Ethanol', 'H2O')
Traceback (most recent call last):
ValueError: synonym 'H2O' already in use by Chemical('Water')
```

zeros()

Return an array of zeros with entries that correspond to the orded chemical IDs.

Examples

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Ethanol'], cache=True)
>>> chemicals.zeros()
array([0., 0.])
```

ones()

Return an array of ones with entries that correspond to the orded chemical IDs.

Examples

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Ethanol'], cache=True)
>>> chemicals.ones()
array([1., 1.])
```

kwarray(*ID_data*)

Return an array with entries that correspond to the orded chemical IDs.

Parameters

ID_data (*dict*) – ID-data pairs.

Examples

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Ethanol'], cache=True)
>>> chemicals.kwarray(dict(Water=2))
array([2., 0.])
```

array(*IDs*, *data*)

Return an array with entries that correspond to the ordered chemical IDs.

Parameters

- **IDs** (*iterable*) – Compound IDs.
- **data** (*array_like*) – Data corresponding to IDs.

Examples

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Ethanol'], cache=True)
>>> chemicals.array(['Water'], [2])
array([2., 0.])
```

iarray(*IDs*, *data*)

Return a chemical indexer.

Parameters

- **IDs** (*iterable*) – Chemical IDs.
- **data** (*array_like*) – Data corresponding to IDs.

Examples

Create a chemical indexer from chemical IDs and data:

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Methanol', 'Ethanol'], cache=True)
>>> chemical_indexer = chemicals.iarray(['Water', 'Ethanol'], [2., 1.])
>>> chemical_indexer.show()
ChemicalIndexer:
 Water    2
 Ethanol  1
```

Note that indexers allow for computationally efficient indexing using identifiers:

```
>>> chemical_indexer['Ethanol', 'Water']
array([1., 2.])
>>> chemical_indexer['Ethanol']
1.0
```

ikwarray(*ID_data*)

Return a chemical indexer.

Parameters

ID_data (*Dict[str: float]*) – Chemical ID-value pairs.

Examples

Create a chemical indexer from chemical IDs and data:

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Methanol', 'Ethanol'], cache=True)
>>> chemical_indexer = chemicals.ikwarray(dict(Water=2., Ethanol=1.))
>>> chemical_indexer.show()
ChemicalIndexer:
 Water    2
 Ethanol  1
```

Note that indexers allow for computationally efficient indexing using identifiers:

```
>>> chemical_indexer['Ethanol', 'Water']
array([1., 2.])
>>> chemical_indexer['Ethanol']
1.0
```

isplit(*split*, *order=None*)

Create a chemical indexer that represents chemical splits.

Parameters

- **split** (*Should be one of the following*) –
  - [float] Split fraction
  - [array_like] Componentwise split
  - [dict] ID-split pairs

- **order=None** (*Iterable[str], options*) – Chemical order of split. Defaults to biosteam.settings.chemicals.IDs

Examples

From a dictionary:

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Methanol', 'Ethanol'], cache=True)
>>> chemical_indexer = chemicals.isplit(dict(Water=0.5, Ethanol=1.))
>>> chemical_indexer.show()
ChemicalIndexer:
 Water    0.5
 Ethanol  1
```

From iterable given the order:

```
>>> chemical_indexer = chemicals.isplit([0.5, 1], ['Water', 'Ethanol'])
>>> chemical_indexer.show()
ChemicalIndexer:
 Water    0.5
 Ethanol  1
```

From a fraction:

```
>>> chemical_indexer = chemicals.isplit(0.75)
>>> chemical_indexer.show()
ChemicalIndexer:
 Water     0.75
 Methanol  0.75
 Ethanol   0.75
```

From an iterable (assuming same order as the Chemicals object):

```
>>> chemical_indexer = chemicals.isplit([0.5, 0, 1])
>>> chemical_indexer.show()
ChemicalIndexer:
 Water    0.5
 Ethanol  1
```

index(*ID*)

Return index of specified chemical.

Parameters

ID (*str*) – Chemical identifier.

Examples

Index by ID:

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Ethanol'])
>>> chemicals.index('Water')
0
```

Indices by CAS number:

```
>>> chemicals.index('7732-18-5')
0
```

indices(*IDs*)

Return indices of specified chemicals.

Parameters

IDs (*iterable*) – Chemical indentifiers.

Examples

Indices by ID:

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Ethanol'])
>>> chemicals.indices(['Water', 'Ethanol'])
[0, 1]
```

Indices by CAS number:

```
>>> chemicals.indices(['7732-18-5', '64-17-5'])
[0, 1]
```

get_index(*IDs*)

Return index/indices of specified chemicals.

Parameters

IDs (*iterable[str] or str*) – Chemical identifiers.

Notes

CAS numbers are also supported.

Examples

Get multiple indices with a tuple of IDs:

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Ethanol'], cache=True)
>>> IDs = ('Water', 'Ethanol')
>>> chemicals.get_index(IDs)
[0, 1]
```

Get a single index with a string:

```
>>> chemicals.get_index('Ethanol')
1
```

An Ellipsis returns a slice:

```
>>> chemicals.get_index(...)
slice(None, None, None)
```

Anything else returns an error:

```
>>> chemicals.get_index(['Water', 'Ethanol'])
Traceback (most recent call last):
TypeError: only strings, tuples, and ellipsis are valid index keys
```

get_vle_indices(*nonzeros*)

Return indices of species in vapor-liquid equilibrium given an array dictating whether or not the chemicals are present.

Examples

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Methanol', 'Ethanol'])
>>> data = chemicals.kwarray(dict(Water=2., Ethanol=1.))
>>> chemicals.get_vle_indices(data!=0)
[0, 2]
```

get_lle_indices(*nonzeros*)

Return indices of species in liquid-liquid equilibrium given an array dictating whether or not the chemicals are present.

Examples

```
>>> from thermosteam import CompiledChemicals
>>> chemicals = CompiledChemicals(['Water', 'Methanol', 'Ethanol'])
>>> data = chemicals.kwarray(dict(Water=2., Ethanol=1.))
>>> chemicals.get_lle_indices(data!=0)
[0, 2]
```

append(**args*, ***kwargs*)

Append a Chemical.

extend(**args*, ***kwargs*)

Extend with more Chemical objects.

1.5 Thermo

class thermosteam.**Thermo**(*chemicals*, *mixture=None*, *Gamma=<class 'thermosteam.equilibrium.activity_coefficients.DortmundActivityCoefficients'>*, *Phi=<class 'thermosteam.equilibrium.fugacity_coefficients.IdealFugacityCoefficients'>*, *PCF=<class 'thermosteam.equilibrium.poynting_correction_factors.IdealPoyntingCorrectionFactors'>*, *cache=None*)

Create a Thermo object that defines a thermodynamic property package

Parameters

- **chemicals** (*Chemicals or Iterable[str]*) – Pure component chemical data.
- **mixture** (*Mixture, optional*) – Calculates mixture properties.
- **Gamma** (*ActivityCoefficients subclass, optional*) – Class for computing activity coefficients.
- **Phi** (*FugacityCoefficients subclass, optional*) – Class for computing fugacity coefficients.
- **PCF** (*PoyntingCorrectionFactor subclass, optional*) – Class for computing poynting correction factors.
- **cache** (*optional*) – Whether or not to use cached chemicals.

Examples

Create a property package for water and ethanol:

```
>>> import thermosteam as tmo
>>> thermo = tmo.Thermo(['Ethanol', 'Water'], cache=True)
>>> thermo
Thermo(chemicals=CompiledChemicals([Ethanol, Water]), mixture=Mixture(rule='ideal␣
↪mixing', ..., include_excess_energies=False), Gamma=DortmundActivityCoefficients,␣
↪Phi=IdealFugacityCoefficients, PCF=IdealPoyntingCorrectionFactors)
>>> thermo.show() # May be easier to read
Thermo(
    chemicals=CompiledChemicals([Ethanol, Water]),
    mixture=Mixture(
        rule='ideal mixing', ...
        include_excess_energies=False
    ),
    Gamma=DortmundActivityCoefficients,
    Phi=IdealFugacityCoefficients,
    PCF=IdealPoyntingCorrectionFactors
)
```

Note that the Dortmund-UNIFAC is the default activity coefficient model. The ideal-equilibrium property package (which assumes a value of 1 for all activity coefficients) is also available:

```
>>> ideal = thermo.ideal()
>>> ideal.show()
Thermo(
    chemicals=CompiledChemicals([Ethanol, Water]),
    mixture=Mixture(
        rule='ideal mixing', ...
        include_excess_energies=False
    ),
    Gamma=IdealActivityCoefficients,
    Phi=IdealFugacityCoefficients,
    PCF=IdealPoyintingCorrectionFactors
)
```

Thermodynamic equilibrium results are affected by the choice of property package:

```
>>> # Ideal
>>> tmo.settings.set_thermo(ideal)
>>> stream = tmo.Stream('stream', Water=100, Ethanol=100)
>>> stream.vle(T=361, P=101325)
>>> stream.show()
MultiStream: stream
 phases: ('g', 'l'), T: 361 K, P: 101325 Pa
 flow (kmol/hr): (g) Ethanol  32.2
                     Water    17.3
                 (l) Ethanol  67.8
                     Water    82.7
>>> # Modified Roult's law:
>>> tmo.settings.set_thermo(thermo)
>>> stream = tmo.Stream('stream', Water=100, Ethanol=100)
>>> stream.vle(T=360, P=101325)
>>> stream.show()
MultiStream: stream
 phases: ('g', 'l'), T: 360 K, P: 101325 Pa
 flow (kmol/hr): (g) Ethanol  100
                     Water    100
```

Thermodynamic property packages are pickleable:

```
>>> tmo.utils.save(thermo, "Ethanol-Water Property Package")
>>> thermo = tmo.utils.load("Ethanol-Water Property Package")
>>> thermo.show()
Thermo(
    chemicals=CompiledChemicals([Ethanol, Water]),
    mixture=Mixture(
        rule='ideal mixing', ...
        include_excess_energies=False
    ),
    Gamma=DortmundActivityCoefficients,
    Phi=IdealFugacityCoefficients,
    PCF=IdealPoyintingCorrectionFactors
)
```

chemicals

Pure component chemical data.

Type
Chemicals or Iterable[str]

mixture

Calculates mixture properties.

Type
Mixture, optional

Gamma

Class for computing activity coefficients.

Type
ActivityCoefficients subclass, optional

Phi

Class for computing fugacity coefficients.

Type
FugacityCoefficients subclass, optional

PCF

Class for computing poynting correction factors.

Type
PoyntingCorrectionFactor subclass, optional

ideal()

Ideal thermodynamic property package.

as_chemical(*chemical*)

Return chemical as a Chemical object.

Parameters
chemical (*str or* Chemical) – Name of chemical being retrieved.

Examples

```
>>> import thermosteam as tmo
>>> thermo = tmo.Thermo(['Ethanol', 'Water'], cache=True)
>>> thermo.as_chemical('Water') is thermo.chemicals.Water
True
>>> thermo.as_chemical('Octanol') # Chemical not defined, so it will be created
Chemical('Octanol')
```

subgroup(*IDs*)

Create a Thermo object from a subset of chemicals.

Parameters
IDs (*Iterable[str]*) – Names of chemicals.

Examples

```
>>> import thermosteam as tmo
>>> thermo = tmo.Thermo(['Ethanol', 'Water'], cache=True)
>>> thermo_water = thermo.subgroup(['Water'])
>>> thermo_water.show()
Thermo(
    chemicals=CompiledChemicals([Water]),
    mixture=Mixture(
        rule='ideal mixing', ...
        include_excess_energies=False
    ),
    Gamma=DortmundActivityCoefficients,
    Phi=IdealFugacityCoefficients,
    PCF=IdealPoyintingCorrectionFactors
)
```

1.6 Stream

class thermosteam.**Stream**(*ID=''*, *flow=()*, *phase='l'*, *T=298.15*, *P=101325.0*, *units='kmol/hr'*, *price=0.0*, *thermo=None*, ***chemical_flows*)

Create a Stream object that defines material flow rates along with its thermodynamic state. Thermodynamic and transport properties of a stream are available as properties, while thermodynamic equilbrium (e.g. VLE, and bubble and dew points) are available as methods.

Parameters

- **ID=''** (*str*) – A unique identification. If ID is None, stream will not be registered. If no ID is given, stream will be registered with a unique ID.
- **flow=()** (*Iterable[float]*) – All flow rates corresponding to chemical *IDs*.
- **phase='l'** (*'l', 'g', or 's'*) – Either gas (g), liquid (l), or solid (s).
- **T=298.15** (*float*) – Temperature [K].
- **P=101325** (*float*) – Pressure [Pa].
- **units='kmol/hr'** (*str*) – Flow rate units of measure (only mass, molar, and volumetric flow rates are valid).
- **price=0** (*float*) – Price per unit mass [USD/kg].
- **thermo=None** (Thermo) – Thermo object to initialize input and output streams. Defaults to *biosteam.settings.get_thermo()*.
- ****chemical_flows** (*float*) – ID - flow pairs.

Examples

Before creating a stream, first set the chemicals:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
```

Create a stream, defining the thermodynamic condition and flow rates:

```
>>> s1 = tmo.Stream(ID='s1',
...                 Water=20, Ethanol=10, units='kg/hr',
...                 T=298.15, P=101325, phase='l')
>>> s1.show(flow='kg/hr') # Use the show method to select units of display
Stream: s1
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    20
               Ethanol  10
>>> s1.show(composition=True, flow='kg/hr') # Its also possible to show by␣
↪composition
Stream: s1
 phase: 'l', T: 298.15 K, P: 101325 Pa
 composition: Water    0.667
              Ethanol  0.333
              -------  30 kg/hr
```

All flow rates are stored as an array in the *mol* attribute:

```
>>> s1.mol # Molar flow rates [kmol/hr]
array([1.11 , 0.217])
```

Mass and volumetric flow rates are available as property arrays:

```
>>> s1.mass
property_array([<Water: 20 kg/hr>, <Ethanol: 10 kg/hr>])
>>> s1.vol
property_array([<Water: 0.02006 m^3/hr>, <Ethanol: 0.012722 m^3/hr>])
```

These arrays work just like ordinary arrays, but the data is linked to the molar flows:

```
>>> # Mass flows are always up to date with molar flows
>>> s1.mol[0] = 1
>>> s1.mass[0]
<Water: 18.015 kg/hr>
>>> # Changing mass flows changes molar flows
>>> s1.mass[0] *= 2
>>> s1.mol[0]
2.0
>>> # Property arrays act just like normal arrays
>>> s1.mass + 2
array([38.031, 12.   ])
```

The temperature, pressure and phase are attributes as well:

```
>>> (s1.T, s1.P, s1.phase)
(298.15, 101325.0, 'l')
```

The most convinient way to get and set flow rates is through the *get_flow* and *set_flow* methods:

```
>>> # Set flow
>>> s1.set_flow(1, 'gpm', 'Water')
>>> s1.get_flow('gpm', 'Water')
1.0
>>> # Set multiple flows
>>> s1.set_flow([10, 20], 'kg/hr', ('Ethanol', 'Water'))
>>> s1.get_flow('kg/hr', ('Ethanol', 'Water'))
array([10., 20.])
```

It is also possible to index using IDs through the *imol*, *imass*, and *ivol* indexers:

```
>>> s1.imol.show()
ChemicalMolarFlowIndexer (kmol/hr):
 (l) Water    1.11
     Ethanol  0.2171
>>> s1.imol['Water']
1.1101687012358397
>>> s1.imol['Ethanol', 'Water']
array([0.217, 1.11 ])
```

Thermodynamic properties are available as stream properties:

```
>>> s1.H # Enthalpy (kJ/hr)
0.0
```

Note that the reference enthalpy is 0.0 at the reference temperature of 298.15 K, and pressure of 101325 Pa. Retrive the enthalpy at a 10 degC above the reference.

```
>>> s1.T += 10
>>> s1.H
1083.467954...
```

Other thermodynamic properties are temperature and pressure dependent as well:

```
>>> s1.rho # Density [kg/m3]
908.8914226...
```

It may be more convinient to get properties with different units:

```
>>> s1.get_property('rho', 'g/cm3')
0.90889142...
```

It is also possible to set some of the properties in different units:

```
>>> s1.set_property('T', 40, 'degC')
>>> s1.T
313.15
```

Bubble point and dew point computations can be performed through stream methods:

```
>>> bp = s1.bubble_point_at_P() # Bubble point at constant pressure
>>> bp
```

(continues on next page)

(continued from previous page)

```
BubblePointValues(T=357.09, P=101325, IDs=('Water', 'Ethanol'), z=[0.836 0.164],␣
↪y=[0.49 0.51])
```

The bubble point results contain all results as attributes:

```
>>> bp.T # Temperature [K]
357.088...
>>> bp.y # Vapor composition
array([0.49, 0.51])
```

Vapor-liquid equilibrium can be performed by setting 2 degrees of freedom from the following list: T [Temperature; in K], P [Pressure; in Pa], V [Vapor fraction], H [Enthalpy; in kJ/hr].

Set vapor fraction and pressure of the stream:

```
>>> s1.vle(P=101325, V=0.5)
>>> s1.show()
MultiStream: s1
 phases: ('g', 'l'), T: 364.8 K, P: 101325 Pa
 flow (kmol/hr): (g) Water    0.472
                     Ethanol  0.192
                 (l) Water    0.638
                     Ethanol  0.0255
```

Note that the stream is a now a MultiStream object to manage multiple phases. Each phase can be accessed separately too:

```
>>> s1['l'].show()
Stream:
 phase: 'l', T: 364.8 K, P: 101325 Pa
 flow (kmol/hr): Water    0.638
                 Ethanol  0.0255
```

```
>>> s1['g'].show()
Stream:
 phase: 'g', T: 364.8 K, P: 101325 Pa
 flow (kmol/hr): Water    0.472
                 Ethanol  0.192
```

We can convert a MultiStream object back to a Stream object by setting the phase:

```
>>> s1.phase = 'l'
>>> s1.show(flow='kg/hr')
Stream: s1
 phase: 'l', T: 364.8 K, P: 101325 Pa
 flow (kg/hr): Water    20
               Ethanol  10
```

display_units = DisplayUnits(T='K', P='Pa', flow='kmol/hr', composition=False, N=7)

[DisplayUnits] Units of measure for IPython display (class attribute)

as_stream()

Does nothing.

property price

[float] Price of stream per unit mass [USD/kg.

isempty()

Return whether or not stream is empty.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water'], cache=True)
>>> stream = tmo.Stream()
>>> stream.isempty()
True
```

property vapor_fraction

Molar vapor fraction.

property liquid_fraction

Molar liquid fraction.

property solid_fraction

Molar solid fraction.

isfeed()

Return whether stream has a sink but no source.

isproduct()

Return whether stream has a source but no sink.

property main_chemical

[str] ID of chemical with the largest mol fraction in stream.

disconnect()

Disconnect stream from unit operations.

get_atomic_flow(*symbol*)

Return flow rate of atom in kmol / hr given the atomic symbol.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water'], cache=True)
>>> stream = tmo.Stream(Water=1)
>>> stream.get_atomic_flow('H') # kmol/hr of H
2.0
>>> stream.get_atomic_flow('O') # kmol/hr of O
1.0
```

get_atomic_flows()

Return dictionary of atomic flow rates in kmol / hr.

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water'], cache=True)
>>> stream = tmo.Stream(Water=1)
>>> stream.get_atomic_flows()
{'H': 2.0, 'O': 1.0}
```

get_flow(*units*, *key=Ellipsis*)

Return an flow rates in requested units.

Parameters

- **units** (*str*) – Units of measure.
- **key** (*tuple[str] or str, optional*) – Chemical identifiers.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s1.get_flow('kg/hr', 'Water')
20.0
```

set_flow(*data*, *units*, *key=Ellipsis*)

Set flow rates in given units.

Parameters

- **data** (*1d ndarray or float*) – Flow rate data.
- **units** (*str*) – Units of measure.
- **key** (*Iterable[str] or str, optional*) – Chemical identifiers.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream(ID='s1', Water=20, Ethanol=10, units='kg/hr')
>>> s1.set_flow(10, 'kg/hr', 'Water')
>>> s1.get_flow('kg/hr', 'Water')
10.0
```

get_total_flow(*units*)

Get total flow rate in given units.

Parameters

units (*str*) – Units of measure.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s1.get_total_flow('kg/hr')
30.0
```

set_total_flow(*value*, *units*)

Set total flow rate in given units keeping the composition constant.

Parameters

- **value** (*float*) – New total flow rate.
- **units** (*str*) – Units of measure.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s1.set_total_flow(1.0,'kg/hr')
>>> s1.get_total_flow('kg/hr')
0.9999999999999999
```

get_property(*name*, *units*)

Return property in requested units.

Parameters

- **name** (*str*) – Name of stream property.
- **units** (*str*) – Units of measure.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s1.get_property('sigma', 'N/m') # Surface tension
0.063780393
```

set_property(*name*, *value*, *units*)

Set property in given units.

Parameters

- **name** (*str*) – Name of stream property.
- **value** (*str*) – New value of stream property.
- **units** (*str*) – Units of measure.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s1.set_property('P', 2, 'atm')
>>> s1.P
202650.0
```

property source

[Unit] Outlet location.

property sink

[Unit] Inlet location.

property thermal_condition

[ThermalCondition] Contains the temperature and pressure conditions of the stream.

property T

[float] Temperature in Kelvin.

property P

[float] Pressure in Pascal.

property phase

Phase of stream.

property mol

[array] Molar flow rates in kmol/hr.

property mass

[property_array] Mass flow rates in kg/hr.

property vol

[property_array] Volumetric flow rates in m3/hr.

property imol

[Indexer] Flow rate indexer with data in kmol/hr.

property imass

[Indexer] Flow rate indexer with data in kg/hr.

property ivol

[Indexer] Flow rate indexer with data in m3/hr.

property cost

[float] Total cost of stream in USD/hr.

property F_mol

[float] Total molar flow rate in kmol/hr.

property F_mass

[float] Total mass flow rate in kg/hr.

property F_vol

[float] Total volumetric flow rate in m3/hr.

property H

[float] Enthalpy flow rate in kJ/hr.

property S

[float] Absolute entropy flow rate in kJ/hr.

property Hnet

[float] Total enthalpy flow rate (including heats of formation) in kJ/hr.

property Hf

[float] Enthalpy of formation flow rate in kJ/hr.

property LHV

[float] Lower heating value flow rate in kJ/hr.

property HHV

[float] Higher heating value flow rate in kJ/hr.

property Hvap

[float] Enthalpy of vaporization flow rate in kJ/hr.

property C

[float] Heat capacity flow rate in kJ/K/hr.

property z_mol

[1d array] Molar composition.

property z_mass

[1d array] Mass composition.

property z_vol

[1d array] Volumetric composition.

property MW

[float] Overall molecular weight.

property V

[float] Molar volume [m^3/mol].

property kappa

[float] Thermal conductivity [W/m/k].

property Cn

[float] Molar heat capacity [J/mol/K].

property mu

[float] Hydrolic viscosity [Pa*s].

property sigma

[float] Surface tension [N/m].

property epsilon

[float] Relative permittivity [-].

property Cp

[float] Heat capacity [J/g/K].

property alpha

[float] Thermal diffusivity [m^2/s].

property rho

[float] Density [kg/m^3].

property nu

[float] Kinematic viscosity [-].

property Pr

[float] Prandtl number [-].

property available_chemicals

list[Chemical] All chemicals with nonzero flow.

in_thermal_equilibrium(*other*)

Return whether or not stream is in thermal equilibrium with another stream.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> stream = Stream(Water=1, T=300)
>>> other = Stream(Water=1, T=300)
>>> stream.in_thermal_equilibrium(other)
True
```

classmethod sum(*streams*, *ID=None*, *thermo=None*)

Return a new Stream object that represents the sum of all given streams.

Examples

Sum two streams:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s_sum = tmo.Stream.sum([s1, s1], 's_sum')
>>> s_sum.show(flow='kg/hr')
Stream: s_sum
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    40
               Ethanol  20
```

Sum two streams with new property package:

```
>>> thermo = tmo.Thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s_sum = tmo.Stream.sum([s1, s1], 's_sum', thermo)
>>> s_sum.show(flow='kg/hr')
Stream: s_sum
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    40
               Ethanol  20
```

mix_from(*others*)

Mix all other streams into this one, ignoring its initial contents.

Examples

Mix two streams with the same thermodynamic property package:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s2 = s1.copy('s2')
>>> s1.mix_from([s1, s2])
>>> s1.show(flow='kg/hr')
Stream: s1
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    40
               Ethanol  20
```

It's also possible to mix streams with different property packages so long as all chemicals are defined in the mixed stream's property package:

```
>>> tmo.settings.set_thermo(['Water'], cache=True)
>>> s1 = tmo.Stream('s1', Water=40, units='kg/hr')
>>> tmo.settings.set_thermo(['Ethanol'], cache=True)
>>> s2 = tmo.Stream('s2', Ethanol=20, units='kg/hr')
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s_mix = tmo.Stream('s_mix')
>>> s_mix.mix_from([s1, s2])
>>> s_mix.show(flow='kg/hr')
Stream: s_mix
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    40
               Ethanol  20
```

split_to(*s1*, *s2*, *split*)

Split molar flow rate from this stream to two others given the split fraction or an array of split fractions.

Examples

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> s = tmo.Stream('s', Water=20, Ethanol=10, units='kg/hr')
>>> s1 = tmo.Stream('s1')
>>> s2 = tmo.Stream('s2')
>>> split = chemicals.kwarray(dict(Water=0.5, Ethanol=0.1))
>>> s.split_to(s1, s2, split)
>>> s1.show(flow='kg/hr')
Stream: s1
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    10
               Ethanol  1
```

```
>>> s2.show(flow='kg/hr')
Stream: s2
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    10
               Ethanol  9
```

link_with(*other*, *flow=True*, *phase=True*, *TP=True*)

Link with another stream.

Parameters

- **other** (Stream) –
- **flow** (*bool, defaults to True*) – Whether to link the flow rate data.
- **phase** (*bool, defaults to True*) – Whether to link the phase.
- **TP** (*bool, defaults to True*) – Whether to link the temperature and pressure.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s2 = tmo.Stream('s2')
>>> s2.link_with(s1)
>>> s1.mol is s2.mol
True
>>> s2.thermal_condition is s1.thermal_condition
True
>>> s1.phase = 'g'
>>> s2.phase
'g'
```

unlink()

Unlink stream from other streams.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s2 = tmo.Stream('s2')
>>> s2.link_with(s1)
>>> s1.unlink()
>>> s2.mol is s1.mol
False
```

copy_like(*other*)

Copy all conditions of another stream.

Examples

Copy data from another stream with the same property package:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s2 = tmo.Stream('s2', Water=2, units='kg/hr')
>>> s1.copy_like(s2)
>>> s1.show(flow='kg/hr')
Stream: s1
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water  2
```

Copy data from another stream with a different property package:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> tmo.settings.set_thermo(['Water'], cache=True)
>>> s2 = tmo.Stream('s2', Water=2, units='kg/hr')
>>> s1.copy_like(s2)
>>> s1.show(flow='kg/hr')
Stream: s1
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water  2
```

copy_thermal_condition(*other*)

Copy thermal conditions (T and P) of another stream.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=2, units='kg/hr')
>>> s2 = tmo.Stream('s2', Water=1, units='kg/hr', T=300.00)
>>> s1.copy_thermal_condition(s2)
>>> s1.show(flow='kg/hr')
Stream: s1
 phase: 'l', T: 300 K, P: 101325 Pa
 flow (kg/hr): Water  2
```

copy_flow(*stream*, *IDs=Ellipsis*, *, *remove=False*, *exclude=False*)

Copy flow rates of stream to self.

Parameters

- **stream** (Stream) – Flow rates will be copied from here.
- **IDs=...** (*Iterable[str], defaults to all chemicals.*) – Chemical IDs.
- **remove=False** (*bool, optional*) – If True, copied chemicals will be removed from *stream*.
- **exclude=False** (*bool, optional*) – If True, exclude designated chemicals when copying.

Examples

Initialize streams:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s2 = tmo.Stream('s2')
```

Copy all flows:

```
>>> s2.copy_flow(s1)
>>> s2.show(flow='kg/hr')
Stream: s2
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    20
               Ethanol  10
```

Reset and copy just water flow:

```
>>> s2.empty()
>>> s2.copy_flow(s1, 'Water')
>>> s2.show(flow='kg/hr')
Stream: s2
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water  20
```

Reset and copy all flows except water:

```
>>> s2.empty()
>>> s2.copy_flow(s1, 'Water', exclude=True)
>>> s2.show(flow='kg/hr')
Stream: s2
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Ethanol  10
```

Cut and paste flows:

```
>>> s2.copy_flow(s1, remove=True)
>>> s2.show(flow='kg/hr')
Stream: s2
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    20
               Ethanol  10
```

```
>>> s1.show()
Stream: s1
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow: 0
```

Its also possible to copy flows from a multistream:

```
>>> s1.phases = ('g', 'l')
>>> s1.imol['g', 'Water'] = 10
```

(continues on next page)

(continued from previous page)

```
>>> s2.copy_flow(s1, remove=True)
>>> s2.show()
Stream: s2
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water  10
>>> s1.show()
MultiStream: s1
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow: 0
```

copy(*ID=None*)

Return a copy of the stream.

Examples

Create a copy of a new stream:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s1_copy = s1.copy('s1_copy')
>>> s1_copy.show(flow='kg/hr')
Stream: s1_copy
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water    20
               Ethanol  10
```

Warning: Prices are not copied.

flow_proxy(*ID=None*)

Return a new stream that shares flow rate data with this one.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s2 = s1.flow_proxy()
>>> s2.mol is s1.mol
True
```

proxy(*ID=None*)

Return a new stream that shares all data with this one.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s2 = s1.proxy()
>>> s2.imol is s1.imol and s2.thermal_condition is s1.thermal_condition
True
```

empty()

Empty stream flow rates.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s1.empty()
>>> s1.F_mol
0.0
```

property vle

[VLE] An object that can perform vapor-liquid equilibrium on the stream.

property lle

[LLE] An object that can perform liquid-liquid equilibrium on the stream.

property sle

[SLE] An object that can perform solid-liquid equilibrium on the stream.

property vle_chemicals

list[Chemical] Chemicals cabable of liquid-liquid equilibrium.

property lle_chemicals

list[Chemical] Chemicals cabable of vapor-liquid equilibrium.

get_bubble_point(*IDs=None*)

Return a BubblePoint object capable of computing bubble points.

Parameters

IDs (*Iterable[str], optional*) – Chemicals that participate in equilibrium. Defaults to all chemicals in equilibrium.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, T=350, units='kg/hr')
>>> s1.get_bubble_point()
BubblePoint([Water, Ethanol])
```

get_dew_point(*IDs=None*)

Return a DewPoint object capable of computing dew points.

Parameters

IDs (*Iterable[str], optional*) – Chemicals that participate in equilibrium. Defaults to all chemicals in equilibrium.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, T=350, units='kg/hr')
>>> s1.get_dew_point()
DewPoint([Water, Ethanol])
```

bubble_point_at_T(*T=None*, *IDs=None*)

Return a BubblePointResults object with all data on the bubble point at constant temperature.

Parameters

IDs (*Iterable[str], optional*) – Chemicals that participate in equilibrium. Defaults to all chemicals in equilibrium.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, T=350, units='kg/hr')
>>> s1.bubble_point_at_T()
BubblePointValues(T=350.00, P=76622, IDs=('Water', 'Ethanol'), z=[0.836 0.164],
↪ y=[0.486 0.514])
```

bubble_point_at_P(*P=None*, *IDs=None*)

Return a BubblePointResults object with all data on the bubble point at constant pressure.

Parameters

IDs (*Iterable[str], optional*) – Chemicals that participate in equilibrium. Defaults to all chemicals in equilibrium.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, T=350, units='kg/hr')
>>> s1.bubble_point_at_P()
BubblePointValues(T=357.09, P=101325, IDs=('Water', 'Ethanol'), z=[0.836 0.
↪164], y=[0.49 0.51])
```

dew_point_at_T(*T=None*, *IDs=None*)

Return a DewPointResults object with all data on the dew point at constant temperature.

Parameters
IDs (*Iterable[str], optional*) – Chemicals that participate in equilibrium. Defaults to all chemicals in equilibrium.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, T=350, units='kg/hr')
>>> s1.dew_point_at_T()
DewPointValues(T=350.00, P=48991, IDs=('Water', 'Ethanol'), z=[0.836 0.164],␣
↪x=[0.984 0.016])
```

dew_point_at_P(*P=None*, *IDs=None*)

Return a DewPointResults object with all data on the dew point at constant pressure.

Parameters
IDs (*Iterable[str], optional*) – Chemicals that participate in equilibrium. Defaults to all chemicals in equilibrium.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, T=350, units='kg/hr')
>>> s1.dew_point_at_P()
DewPointValues(T=368.66, P=101325, IDs=('Water', 'Ethanol'), z=[0.836 0.164],␣
↪x=[0.984 0.016])
```

get_normalized_mol(*IDs*)

Return normalized molar fractions of given chemicals. The sum of the result is always 1.

Parameters
IDs (*tuple[str]*) – IDs of chemicals to be normalized.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, Methanol=10, units='kmol/hr')
>>> s1.get_normalized_mol(('Water', 'Ethanol'))
array([0.667, 0.333])
```

get_normalized_mass(*IDs*)

Return normalized mass fractions of given chemicals. The sum of the result is always 1.

Parameters
IDs (*tuple[str]*) – IDs of chemicals to be normalized.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, Methanol=10, units='kg/hr')
>>> s1.get_normalized_mass(('Water', 'Ethanol'))
array([0.667, 0.333])
```

get_normalized_vol(*IDs*)

Return normalized mass fractions of given chemicals. The sum of the result is always 1.

Parameters

IDs (*tuple[str]*) – IDs of chemicals to be normalized.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, Methanol=10, units='m3/hr')
>>> s1.get_normalized_vol(('Water', 'Ethanol'))
array([0.667, 0.333])
```

get_molar_composition(*IDs*)

Return molar composition of given chemicals.

Parameters

IDs (*tuple[str]*) – IDs of chemicals.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, Methanol=10, units='kmol/hr')
>>> s1.get_molar_composition(('Water', 'Ethanol'))
array([0.5 , 0.25])
```

get_mass_composition(*IDs*)

Return mass composition of given chemicals.

Parameters

IDs (*tuple[str]*) – IDs of chemicals.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, Methanol=10, units='kg/hr')
>>> s1.get_mass_composition(('Water', 'Ethanol'))
array([0.5 , 0.25])
```

get_volumetric_composition(*IDs*)

Return volumetric composition of given chemicals.

Parameters

IDs (*tuple[str]*) – IDs of chemicals.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, Methanol=10, units='m3/hr')
>>> s1.get_volumetric_composition(('Water', 'Ethanol'))
array([0.5 , 0.25])
```

property ID

Unique identification (str). If set as '', it will choose a default ID.

get_concentration(*IDs*)

Return concentration of given chemicals in kmol/m3.

Parameters

IDs (*tuple[str]*) – IDs of chemicals.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, Methanol=10, units='m3/hr')
>>> s1.get_concentration(('Water', 'Ethanol'))
array([27.671,  4.266])
```

property P_vapor

Vapor pressure of liquid.

receive_vent(*other*, *accumulate=False*, *fraction=1.0*)

Receive vapors from another stream as if in equilibrium.

Notes

Disregards energy balance. Assumes liquid composition will not change significantly.

Examples

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol', 'Methanol', tmo.Chemical('N2
↪', phase='g')], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> s1 = tmo.Stream('s1', N2=10, units='m3/hr', phase='g', T=330)
>>> s2 = tmo.Stream('s2', Water=10, Ethanol=2, T=330)
>>> s1.receive_vent(s2, accumulate=True)
```

(continues on next page)

(continued from previous page)

```
>>> s1.show(flow='kmol/hr')
Stream: s1
 phase: 'g', T: 330 K, P: 101325 Pa
 flow (kmol/hr): Water    0.0557
                 Ethanol  0.0616
                 N2       0.369
```

property link

[Stream] Data on the thermal condition and material flow rates may be shared with this stream.

property phases

tuple[str] All phases present.

show(*T=None*, *P=None*, *flow=None*, *composition=None*, *N=None*)

Print all specifications.

Parameters

- **T** (*str, optional*) – Temperature units.
- **P** (*str, optional*) – Pressure units.
- **flow** (*str, optional*) – Flow rate units.
- **composition** (*bool, optional*) – Whether to show composition.
- **N** (*int, optional*) – Number of compounds to display.

Notes

Default values are stored in *Stream.display_units*.

print(*units=None*)

Print in a format that you can use recreate the stream.

Parameters

units (*str, optional*) – Units of measure for material flow rates. Defaults to 'kmol/hr'

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream(ID='s1',
...                 Water=20, Ethanol=10, units='kg/hr',
...                 T=298.15, P=101325, phase='l')
>>> s1.print(units='kg/hr')
Stream(ID='s1', phase='l', T=298.15, P=101325, Water=20, Ethanol=10, units='kg/
↪hr')
>>> s1.print() # Units default to kmol/hr
Stream(ID='s1', phase='l', T=298.15, P=101325, Water=1.11, Ethanol=0.2171,␣
↪units='kmol/hr')
```

1.7 MultiStream

A MultiStream object represents a material flow with multiple phases in a chemical process.

class thermosteam.**MultiStream**(*ID=''*, *flow=()*, *T=298.15*, *P=101325.0*, *phases=('g', 'l')*, *units=None*, *thermo=None*, *price=0*, ***phase_flows*)

Create a MultiStream object that defines material flow rates for multiple phases along with its thermodynamic state. Thermodynamic and transport properties of a stream are available as properties, while thermodynamic equilbrium (e.g. VLE, and bubble and dew points) are available as methods.

Parameters

- **ID=''** (*str*) – A unique identification. If ID is None, stream will not be registered. If no ID is given, stream will be registered with a unique ID.
- **flow=()** (*2d array*) – All flow rates corresponding to *phases* by row and chemical IDs by column.
- **thermo=None** (Thermo) – Thermodynamic equilibrium package. Defaults to *thermosteam.settings.get_thermo()*.
- **units='kmol/hr'** (*str*) – Flow rate units of measure (only mass, molar, and volumetric flow rates are valid).
- **phases** (*tuple['g', 'l', 's', 'G', 'L', 'S']*) – Tuple denoting the phases present. Defaults to ('g', 'l').
- **T=298.15** (*float*) – Temperature [K].
- **P=101325** (*float*) – Pressure [Pa].
- **price=0** (*float*) – Price per unit mass [USD/kg].
- ****phase_flow** (*tuple[str, float]*) – phase-(ID, flow) pairs.

Examples

Before creating streams, first set the chemicals:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
```

Create a multi phase stream, defining the thermodynamic condition and flow rates:

```
>>> s1 = tmo.MultiStream(ID='s1',T=298.15, P=101325,
...                      l=[('Water', 20), ('Ethanol', 10)], units='kg/hr')
>>> s1.show(flow='kg/hr') # Use the show method to select units of display
MultiStream: s1
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kg/hr): (l) Water    20
                   Ethanol  10
```

The temperature and pressure are stored as attributes:

```
>>> (s1.T, s1.P)
(298.15, 101325.0)
```

Unlike Stream objects, the *mol* attribute does not store data, it simply returns the total flow rate of each chemical. Setting an element of the array raises an error to prevent the wrong assumption that the data is linked:

```
>>> s1.mol
array([1.11 , 0.217])
>>> s1.mol[0] = 1
Traceback (most recent call last):
ValueError: assignment destination is read-only
```

All flow rates are stored in the *imol* attribute:

```
>>> s1.imol.show() # Molar flow rates [kmol/hr]
MolarFlowIndexer (kmol/hr):
 (l) Water     1.11
     Ethanol   0.2171
>>> # Index a single chemical in the liquid phase
>>> s1.imol['l', 'Water']
1.1101687012358397
>>> # Index multiple chemicals in the liquid phase
>>> s1.imol['l', ('Ethanol', 'Water')]
array([0.217, 1.11 ])
>>> # Index the vapor phase
>>> s1.imol['g']
array([0., 0.])
>>> # Index flow of chemicals summed across all phases
>>> s1.imol['Ethanol', 'Water']
array([0.217, 1.11 ])
```

Note that overall chemical flows in MultiStream objects cannot be set like with Stream objects:

```
>>> # s1.imol['Ethanol', 'Water'] = [1, 0]
Traceback (most recent call last):
IndexError: multiple phases present; must include phase key to set chemical data
```

Chemical flows must be set by phase:

```
>>> s1.imol['l', ('Ethanol', 'Water')] = [1, 0]
```

The most convinient way to get and set flow rates is through the *get_flow* and *set_flow* methods:

```
>>> # Set flow
>>> key = ('l', 'Water')
>>> s1.set_flow(1, 'gpm', key)
>>> s1.get_flow('gpm', key)
1.0
>>> # Set multiple flows
>>> key = ('l', ('Ethanol', 'Water'))
>>> s1.set_flow([10, 20], 'kg/hr', key)
>>> s1.get_flow('kg/hr', key)
array([10., 20.])
```

Chemical flows across all phases can be retrieved if no phase is given:

```
>>> s1.get_flow('kg/hr', ('Ethanol', 'Water'))
array([10., 20.])
```

However, setting chemical data requires the phase to be specified:

```
>>> s1.set_flow([10, 20], 'kg/hr', ('Ethanol', 'Water'))
Traceback (most recent call last):
IndexError: multiple phases present; must include phase key to set chemical data
```

Note that for both Stream and MultiStream objects, *mol*, *imol*, and *get_flow* return chemical flows across all phases when given only chemical IDs.

Vapor-liquid equilibrium can be performed by setting 2 degrees of freedom from the following list:

- T - Temperature [K]
- P - Pressure [Pa]
- V - Vapor fraction
- H - Enthalpy [kJ/hr]

```
>>> s1.vle(P=101325, T=365)
```

Each phase can be accessed separately too:

```
>>> s1['l'].show()
Stream:
 phase: 'l', T: 365 K, P: 101325 Pa
 flow (kmol/hr): Water    0.619
                 Ethanol  0.0238
>>> s1['g'].show()
Stream:
 phase: 'g', T: 365 K, P: 101325 Pa
 flow (kmol/hr): Water    0.491
                 Ethanol  0.193
```

Note that the phase cannot be changed:

```
>>> s1['g'].phase = 'l'
Traceback (most recent call last):
AttributeError: phase is locked
```

get_flow(*units*, *key=Ellipsis*)

Return an array of flow rates in requested units.

Parameters

- **units** (*str*) – Units of measure.
- **key** (*tuple(phase, IDs), phase, or IDs*) –
  - phase: str, ellipsis, or missing.
  - IDs: str, tuple(str), ellipisis, or missing.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10)], units='kg/hr
↪')
>>> s1.get_flow('kg/hr', ('l', 'Water'))
20.0
```

set_flow(*data*, *units*, *key=Ellipsis*)

Set flow rates in given units.

Parameters

- **data** (*1d ndarray or float*) – Flow rate data.
- **units** (*str*) – Units of measure.
- **key** (*tuple(phase, IDs), phase, or IDs*) –
  - phase: str, ellipsis, or missing.
  - IDs: str, tuple(str), ellipisis, or missing.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10)], units='kg/hr
↪')
>>> s1.set_flow(10, 'kg/hr', ('l', 'Water'))
>>> s1.get_flow('kg/hr', ('l', 'Water'))
10.0
```

property phases

tuple[str] All phases avaiable.

property mol

[Array] Chemical molar flow rates (total of all phases).

property mass

[Array] Chemical mass flow rates (total of all phases).

property vol

[Array] Chemical volumetric flow rates (total of all phases).

property H

[float] Enthalpy flow rate in kJ/hr.

property S

[float] Absolute entropy flow rate in kJ/hr.

property C

[float] Heat capacity flow rate in kJ/hr.

property F_vol

[float] Total volumetric flow rate in m3/hr.

property Hvap

[float] Enthalpy of vaporization flow rate in kJ/hr.

property vapor_fraction

Molar vapor fraction.

property liquid_fraction

Molar liquid fraction.

property solid_fraction

Molar solid fraction.

property V

[float] Molar volume [m^3/mol].

property kappa

[float] Thermal conductivity [W/m/k].

property Cn

[float] Molar heat capacity [J/mol/K].

property mu

[float] Hydrolic viscosity [Pa*s].

property sigma

[float] Surface tension [N/m].

property epsilon

[float] Relative permittivity [-].

copy_flow(*other*, *phase=Ellipsis*, *IDs=Ellipsis*, *, *remove=False*, *exclude=False*)

Copy flow rates of another stream to self.

Parameters

- **other** (Stream) – Flow rates will be copied from here.
- **phase** (*str or Ellipsis*) –
- **IDs=None** (*iterable[str], defaults to all chemicals.*) – Chemical IDs.
- **remove=False** (*bool, optional*) – If True, copied chemicals will be removed from *stream*.
- **exclude=False** (*bool, optional*) – If True, exclude designated chemicals when copying.

Notes

Works just like <Stream>.copy_flow, but the phase must be specified.

Examples

Initialize streams:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10)], units='kg/hr
↪')
>>> s2 = tmo.MultiStream('s2')
```

Copy all flows:

```
>>> s2.copy_flow(s1)
>>> s2.show(flow='kg/hr')
MultiStream: s2
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kg/hr): (l) Water    20
                   Ethanol  10
```

Reset and copy just water flow:

```
>>> s2.empty()
>>> s2.copy_flow(s1, IDs='Water')
>>> s2.show(flow='kg/hr')
MultiStream: s2
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kg/hr): (l) Water  20
```

Reset and copy all flows except water:

```
>>> s2.empty()
>>> s2.copy_flow(s1, IDs='Water', exclude=True)
>>> s2.show(flow='kg/hr')
MultiStream: s2
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kg/hr): (l) Ethanol  10
```

Cut and paste flows:

```
>>> s2.copy_flow(s1, remove=True)
>>> s2.show(flow='kg/hr')
MultiStream: s2
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kg/hr): (l) Water    20
                   Ethanol  10
>>> s1.show()
MultiStream: s1
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow: 0
```

The other stream can also be a single phase stream (doesn't have to be a MultiStream object):

Initialize streams:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.Stream('s1', Water=20, Ethanol=10, units='kg/hr')
>>> s2 = tmo.MultiStream('s2')
```

Copy all flows:

```
>>> s2.copy_flow(s1)
>>> s2.show(flow='kg/hr')
MultiStream: s2
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kg/hr): (l) Water    20
                   Ethanol  10
```

Reset and copy just water flow:

```
>>> s2.empty()
>>> s2.copy_flow(s1, IDs='Water')
>>> s2.show(flow='kg/hr')
MultiStream: s2
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kg/hr): (l) Water  20
```

Reset and copy all flows except water:

```
>>> s2.empty()
>>> s2.copy_flow(s1, IDs='Water', exclude=True)
>>> s2.show(flow='kg/hr')
MultiStream: s2
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kg/hr): (l) Ethanol  10
```

Cut and paste flows:

```
>>> s2.copy_flow(s1, remove=True)
>>> s2.show(flow='kg/hr')
MultiStream: s2
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kg/hr): (l) Water    20
                   Ethanol  10
>>> s1.show()
Stream: s1
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow: 0
```

get_normalized_mol(*IDs*)

Return normalized molar fractions of given chemicals. The sum of the result is always 1.

Parameters

IDs (*tuple[str]*) – IDs of chemicals to be normalized.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10), ('Methanol',␣
↪10)], units='kmol/hr')
>>> s1.get_normalized_mol(('Water', 'Ethanol'))
array([0.667, 0.333])
```

get_normalized_vol(*IDs*)

Return normalized mass fractions of given chemicals. The sum of the result is always 1.

Parameters

IDs (*tuple[str]*) – IDs of chemicals to be normalized.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10), ('Methanol',␣
↪10)], units='m3/hr')
>>> s1.get_normalized_vol(('Water', 'Ethanol'))
array([0.667, 0.333])
```

get_normalized_mass(*IDs*)

Return normalized mass fractions of given chemicals. The sum of the result is always 1.

Parameters

IDs (*tuple[str]*) – IDs of chemicals to be normalized.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10), ('Methanol',␣
↪10)], units='kg/hr')
>>> s1.get_normalized_mass(('Water', 'Ethanol'))
array([0.667, 0.333])
```

get_molar_composition(*IDs*)

Return molar composition of given chemicals.

Parameters

IDs (*tuple[str]*) – IDs of chemicals.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10), ('Methanol',␣
↪10)], units='kmol/hr')
>>> s1.get_molar_composition(('Water', 'Ethanol'))
array([0.5 , 0.25])
```

get_mass_composition(*IDs*)

Return mass composition of given chemicals.

Parameters

IDs (*tuple[str]*) – IDs of chemicals.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10), ('Methanol',␣
↪10)], units='kg/hr')
>>> s1.get_mass_composition(('Water', 'Ethanol'))
array([0.5 , 0.25])
```

get_volumetric_composition(*IDs*)

Return volumetric composition of given chemicals.

Parameters

IDs (*tuple[str]*) – IDs of chemicals.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10), ('Methanol',␣
↪10)], units='m3/hr')
>>> s1.get_volumetric_composition(('Water', 'Ethanol'))
array([0.5 , 0.25])
```

get_concentration(*phase*, *IDs*)

Return concentration of given chemicals in kmol/m3.

Parameters

IDs (*tuple[str]*) – IDs of chemicals.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Methanol'], cache=True)
>>> s1 = tmo.MultiStream('s1', l=[('Water', 20), ('Ethanol', 10), ('Methanol',␣
↪10)], units='m3/hr')
>>> s1.get_concentration('l', ('Water', 'Ethanol'))
array([27.671,  4.266])
```

property vle

[VLE] An object that can perform vapor-liquid equilibrium on the stream.

property lle

[LLE] An object that can perform liquid-liquid equilibrium on the stream.

property sle

[SLE] An object that can perform solid-liquid equilibrium on the stream.

as_stream()

Convert MultiStream to Stream.

property phase

Phase of stream.

print()

Print in a format that you can use recreate the stream.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> s1 = tmo.MultiStream(ID='s1',T=298.15, P=101325,
...                      l=[('Water', 20), ('Ethanol', 10)], units='kg/hr')
>>> s1.print()
MultiStream(ID='s1', phases=('g', 'l'), T=298.15, P=101325, l=[('Water', 1.11),
↪ ('Ethanol', 0.2171)])
```

1.8 ThermalCondition

class thermosteam.**ThermalCondition**(*T*, *P*)

Create a ThermalCondition object that contains temperature and pressure values.

Parameters

- **T** (*float*) – Temperature [K].
- **P** (*float*) – Pressure [Pa].

property T

[float] Temperature in Kelvin.

property P

[float] Pressure in Pascal.

in_equilibrium(*other*)

Return whether thermal condition is in equilibrium with another (i. e. same temperature and pressure).

copy()

Return a copy.

copy_like(*other*)

Copy the specifications of another ThermalCondition object.

1.9 ThermoData

class thermosteam.**ThermoData**(*data: dict*)

Create a ThermoData object for creating thermodynamic property packages and streams.

Parameters

data (*dict*) –

Examples

```
>>> import thermosteam as tmo
>>> data = {
... 'Chemicals': {
...     'Water': {},
...     'Ethanol': {},
...     'O2': {'phase': 'gas'},
...     'Cellulose': {
...          'search_db': False,
...          'phase': 'solid',
...          'formula': 'C6H10O5',
...          'Hf': -975708.8,
...          'default': True
...          },
...     'Octane': {}
...     },
... 'Synonyms': {
...     'Water': 'H2O',
...     'Ethanol': [
...         'CH3CH2OH',
...         'EthylAlcohol',
...         ]
...     },
... 'Streams': {
...     'process_water': {
...         'Water': 500,
...         'units': 'kg/hr',
...         'price': 0.00035,
...         },
...     'gasoline': {
...         'Octane': 400,
...         'units': 'kg/hr',
...         'price': 0.756,
```

(continues on next page)

(continued from previous page)

```
...         },
...     }
... }
>>> thermo_data = tmo.ThermoData(data)
>>> chemicals = thermo_data.create_chemicals()
>>> chemicals
CompiledChemicals([Water, Ethanol, O2, Cellulose, Octane])
>>> tmo.settings.set_thermo(chemicals)
>>> thermo_data.create_streams()
[<Stream: process_water>, <Stream: gasoline>]
```

It is also possible to create a ThermoData object from json or yaml files For example, lets say we have a yaml file that looks like this:

```
# File name: example_chemicals.yaml
Chemicals:
  Water:
  Ethanol:
  O2:
    phase: gas
  Cellulose:
    search_db: False
    phase: solid
    formula: C6H10O5
    Hf: -975708.8
    default: True
  Octane:
Synonyms:
  Water: H2O
  Ethanol:
    - CH3CH2OH
    - EthylAlcohol
```

Then we could create the chemicals in just a few lines:

```
>>> # thermo_data = tmo.ThermoData.from_yaml('example_chemicals.yaml')
>>> # thermo_data.create_chemicals()
>>> # CompiledChemicals([Water, Ethanol, O2, Cellulose, Octane])
```

classmethod from_yaml(*file*)

Create a ThermoData object from a yaml file given its directory.

classmethod from_json(*file*)

Create a ThermoData object from a json file given its directory.

create_stream(*ID*)

Create stream from data.

Parameters

ID=None (*str*) – ID of stream to create.

create_streams(*IDs=None*)

Create streams from data.

Parameters
IDs=None (*Iterable[str], optional*) – IDs of streams to create. Defaults to all streams.

create_chemicals(*IDs=None*)

Create chemicals from data.

Parameters
IDs=None (*Iterable[str] or str, optional*) – IDs of chemicals to create. Defaults to all chemicals.

set_synonyms(*chemicals*)

Set synonyms to chemicals.

Parameters
chemicals (CompiledChemicals) –

1.10 functor

thermosteam.**functor**(*f=None*, *var=None*, *units=None*)

Decorate a function of temperature, or both temperature and pressure to have an attribute, *functor*, that serves to create its functor counterpart.

Parameters

- **f** (function(T, *args) or function(T, P, *args)) – Function that calculates a thermodynamic property based on temperature, or both temperature and pressure.
- **var** (*str, optional*) – Name of variable returned (useful for bookkeeping).
- **units** (*dict, optional*) – Units of measure for functor signature.

Returns
f – Same function, but with an attribute *functor* that can create its *Functor* counterpart.

Return type
function(T, *args) or function(T, P, *args)

Notes

The functor decorator checks the signature of the function to find the names of the parameters that should be stored as data.

Examples

Create a functor of temperature that returns the vapor pressure in Pascal:

```
>>> # Describe the return value with `var`.
>>> # Thermosteam's chemical units of measure are always assumed.
>>> import thermosteam as tmo
>>> @tmo.functor(var='Psat')
... def Antoine(T, a, b, c):
...     return 10.0**(a - b / (T + c))
>>> Antoine(T=373.15, a=10.116, b=1687.5, c=-42.98) # functional
```

(continues on next page)

(continued from previous page)

```
101157.148
>>> f = Antoine.functor(a=10.116, b=1687.5, c=-42.98) # as functor object
>>> f.show()
Functor: Antoine(T, P=None) -> Psat [Pa]
 a: 10.116
 b: 1687.5
 c: -42.98
>>> f(T=373.15)
101157.148
```

All functors are saved in the *functors* module:

```
>>> tmo.functors.Antoine
<class 'thermosteam.functors.Antoine'>
```

1.11 exceptions

exception thermosteam.exceptions.**UndefinedChemical**(*ID*)

Exception regarding undefined chemicals.

exception thermosteam.exceptions.**UndefinedPhase**(*phase*)

Exception regarding undefined phases.

exception thermosteam.exceptions.**DimensionError**

ValueError regarding wrong dimensions.

exception thermosteam.exceptions.**InfeasibleRegion**(*region*)

Runtime error regarding infeasible processes.

exception thermosteam.exceptions.**DomainError**(*msg*, ***data*)

ValueError regarding an attempt to evaluate a model out of its domain.

exception thermosteam.exceptions.**InvalidMethod**(*method*)

ValueError regarding an attempt to evaluate an invalid method.

1.12 functional

thermosteam.functional.**normalize**(*array*, *minimum=1e-16*)

Return a normalized array to a magnitude of 1. If magnitude is zero, all fractions will have equal value.

thermosteam.functional.**mixing_simple**(*z*, *y*)

Return a weighted average of *y* given the weights, *z*.

Examples

```
>>> import numpy as np
>>> mixing_simple(np.array([0.1, 0.9]), np.array([0.01, 0.02]))
0.019000000000000003
```

thermosteam.functional.**mixing_logarithmic**(*z*, *y*)

Return the logarithmic weighted average *y* given weights, *z*.

$$y = \sum_i z_i \cdot \log(y_i)$$

Notes

Does not work on negative values.

Examples

```
>>> import numpy as np
>>> mixing_logarithmic(np.array([0.1, 0.9]), np.array([0.01, 0.02]))
0.01866065983073615
```

thermosteam.functional.**mu_to_nu**(*mu*, *rho*)

Return the kinematic viscosity (nu) given the dynamic viscosity (mu) and density (rho).

$$\nu = \frac{\mu}{\rho}$$

Examples

```
>>> mu_to_nu(0.000998, 998.)
1.0e-06
```

thermosteam.functional.**V_to_rho**(*V*, *MW*)

Return the density (rho) in kg/m^3 given the molar volume (V) in m^3/mol and molecular weight (MW) in g/mol.

$$\rho = \frac{MW}{1000 \cdot V}$$

Parameters

- **V** (*float*) – Molar volume, [m^3/mol]
- **MW** (*float*) – Molecular weight, [g/mol]

Returns

rho – Density, [kg/m^3]

Return type

float

Examples

```
>>> V_to_rho(0.000132, 86.18)
652.878...
```

thermosteam.functional.**rho_to_V**(*rho*, *MW*)

Return the molar volume (V) in m^3/mol given the density (rho) in kg/m^3 and molecular weight (MW) in g/mol.

$$V = \left(\frac{1000\rho}{MW}\right)^{-1}$$

Parameters

- **rho** (*float*) – Density, [kg/m^3]
- **MW** (*float*) – Molecular weight, [g/mol]

Returns

V – Molar volume, [m^3/mol]

Return type

float

Examples

```
>>> rho_to_V(652.9, 86.18)
0.0001319957...
```

1.13 equilibrium

Use the equilibrium subpackage to perform equilibrium calculations.

1.13.1 VLE

class thermosteam.equilibrium.**VLE**(*imol=None*, *thermal_condition=None*, *thermo=None*, *bubble_point_cache=None*, *dew_point_cache=None*)

Create a VLE object that performs vapor-liquid equilibrium when called.

Parameters

- **imol=None** (MaterialIndexer, *optional*) – Molar chemical phase data is stored here.
- **thermal_condition=None** (ThermalCondition, *optional*) – Temperature and pressure results are stored here.
- **thermo=None** (Thermo, *optional*) – Themodynamic property package for equilibrium calculations. Defaults to *thermosteam.settings.get_thermo()*.
- **bubble_point_cache=None** (*thermosteam.utils.Cache*, *optional*) – Cache to retrieve bubble point object.
- **dew_point_cache=None** (*thermosteam.utils.Cache*, *optional*) – Cache to retrieve dew point object

Examples

First create a VLE object:

```
>>> from thermosteam import indexer, equilibrium, settings
>>> settings.set_thermo(['Water', 'Ethanol', 'Methanol', 'Propanol'], cache=True)
>>> imol = indexer.MolarFlowIndexer(
...             l=[('Water', 304), ('Ethanol', 30)],
...             g=[('Methanol', 40), ('Propanol', 1)])
>>> vle = equilibrium.VLE(imol)
>>> vle
VLE(imol=MolarFlowIndexer(
        g=[('Methanol', 40), ('Propanol', 1)],
        l=[('Water', 304), ('Ethanol', 30)]),
    thermal_condition=ThermalCondition(T=298.15, P=101325))
```

Equilibrium given vapor fraction and pressure:

```
>>> vle(V=0.5, P=101325)
>>> vle
VLE(imol=MolarFlowIndexer(
        g=[('Water', 126.7), ('Ethanol', 26.4), ('Methanol', 33.49), ('Propanol', 0.
↪896)],
        l=[('Water', 177.3), ('Ethanol', 3.598), ('Methanol', 6.509), ('Propanol',␣
↪0.104)]),
    thermal_condition=ThermalCondition(T=363.88, P=101325))
```

Equilibrium given temperature and pressure:

```
>>> vle(T=363.88, P=101325)
>>> vle
VLE(imol=MolarFlowIndexer(
        g=[('Water', 126.7), ('Ethanol', 26.4), ('Methanol', 33.49), ('Propanol', 0.
↪8959)],
        l=[('Water', 177.3), ('Ethanol', 3.601), ('Methanol', 6.513), ('Propanol',␣
↪0.1041)]),
    thermal_condition=ThermalCondition(T=363.88, P=101325))
```

Equilibrium given enthalpy and pressure:

```
>>> H = vle.thermo.mixture.xH(vle.imol, T=363.88, P=101325)
>>> vle(H=H, P=101325)
>>> vle
VLE(imol=MolarFlowIndexer(
        g=[('Water', 126.7), ('Ethanol', 26.4), ('Methanol', 33.49), ('Propanol', 0.
↪8959)],
        l=[('Water', 177.3), ('Ethanol', 3.601), ('Methanol', 6.513), ('Propanol',␣
↪0.1041)]),
    thermal_condition=ThermalCondition(T=363.88, P=101325))
```

Equilibrium given vapor fraction and temperature:

```
>>> vle(V=0.5, T=363.88)
>>> vle
```

(continues on next page)

(continued from previous page)

```
VLE(imol=MolarFlowIndexer(
        g=[('Water', 126.7), ('Ethanol', 26.4), ('Methanol', 33.49), ('Propanol', 0.
↪896)],
        l=[('Water', 177.3), ('Ethanol', 3.598), ('Methanol', 6.509), ('Propanol',␣
↪0.104)]),
    thermal_condition=ThermalCondition(T=363.88, P=101317))
```

Equilibrium given enthalpy and temperature:

```
>>> vle(H=H, T=363.88)
>>> vle
VLE(imol=MolarFlowIndexer(
        g=[('Water', 126.7), ('Ethanol', 26.4), ('Methanol', 33.49), ('Propanol', 0.
↪8959)],
        l=[('Water', 177.3), ('Ethanol', 3.601), ('Methanol', 6.513), ('Propanol',␣
↪0.1041)]),
    thermal_condition=ThermalCondition(T=363.88, P=101325))
```

__call__(*P=None*, *H=None*, *T=None*, *V=None*, *x=None*, *y=None*)

Perform vapor-liquid equilibrium.

Parameters

- **P=None** (*float*) – Operating pressure [Pa].
- **H=None** (*float*) – Enthalpy [kJ/hr].
- **T=None** (*float*) – Operating temperature [K].
- **V=None** (*float*) – Molar vapor fraction.
- **x=None** (*float*) – Molar composition of liquid (for binary mixtures).
- **y=None** (*float*) – Molar composition of vapor (for binary mixtures).

Notes

You may only specify two of the following parameters: P, H, T, V, x, and y. Additionally, If x or y is specified, the other parameter must be either P or T (e.g., x and V is invalid).

property imol

[MaterialIndexer] Chemical phase data.

property thermal_condition

[ThermalCondition] Temperature and pressure data.

1.13.2 LLE

class thermosteam.equilibrium.**LLE**(*imol=None*, *thermal_condition=None*, *thermo=None*, *composition_cache_tolerance=1e-06*, *temperature_cache_tolerance=1e-06*)

Create a LLE object that performs liquid-liquid equilibrium when called. Differential evolution is used to find the solution that globally minimizes the gibb's free energy of both phases.

Parameters

- **imol=None** (MaterialIndexer, *optional*) – Molar chemical phase data is stored here.
- **thermal_condition=None** (ThermalCondition, *optional*) – The temperature and pressure used in calculations are stored here.
- **thermo=None** (Thermo, *optional*) – Themodynamic property package for equilibrium calculations. Defaults to *thermosteam.settings.get_thermo()*.

Examples

```
>>> from thermosteam import indexer, equilibrium, settings
>>> settings.set_thermo(['Water', 'Ethanol', 'Octane', 'Hexane'], cache=True)
>>> imol = indexer.MolarFlowIndexer(
...             l=[('Water', 304), ('Ethanol', 30)],
...             L=[('Octane', 40), ('Hexane', 1)])
>>> lle = equilibrium.LLE(imol)
>>> lle(T=360)
>>> lle
LLE(imol=MolarFlowIndexer(
        L=[('Water', 2.67), ('Ethanol', 2.28), ('Octane', 39.9), ('Hexane', 0.988)],
        l=[('Water', 301.), ('Ethanol', 27.7), ('Octane', 0.0788), ('Hexane', 0.
↪0115)]),
    thermal_condition=ThermalCondition(T=360.00, P=101325))
```

__call__(*T*, *P=None*, *top_chemical=None*)

Perform liquid-liquid equilibrium.

Parameters

- **T** (*float*) – Operating temperature [K].
- **P** (*float, optional*) – Operating pressure [Pa].
- **top_chemical** (*str, optional*) – Identifier of chemical that will be favored in the "liquid" phase.

1.13.3 BubblePoint

class thermosteam.equilibrium.**BubblePoint**(*chemicals=()*, *thermo=None*)

Create a BubblePoint object that returns bubble point values when called with a composition and either a temperture (T) or pressure (P).

Parameters

- **chemicals=()** (*Iterable[Chemical], optional*) –
- **thermo=None** (*Thermo, optional*) –

Examples

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> BP = tmo.equilibrium.BubblePoint(chemicals)
>>> molar_composition = (0.5, 0.5)
>>> # Solve bubble point at constant temperature
>>> bp = BP(z=molar_composition, T=355)
>>> bp
BubblePointValues(T=355.00, P=109755, IDs=('Water', 'Ethanol'), z=[0.5 0.5], y=[0.
↪343 0.657])
>>> # Note that the result is a BubblePointValues object which contain all results␣
↪as attibutes
>>> (bp.T, round(bp.P), bp.IDs, bp.z, bp.y)
(355, 109755, ('Water', 'Ethanol'), array([0.5, 0.5]), array([0.343, 0.657]))
>>> # Solve bubble point at constant pressure
>>> BP(z=molar_composition, P=101325)
BubblePointValues(T=352.95, P=101325, IDs=('Water', 'Ethanol'), z=[0.5 0.5], y=[0.
↪342 0.658])
```

__call__(*z*, *, *T=None*, *P=None*)

Call self as a function.

solve_Ty(*z*, *P*)

Bubble point at given composition and pressure.

Parameters

- **z** (*ndarray*) – Molar composotion.
- **P** (*float*) – Pressure [Pa].

Returns

- **T** (*float*) – Bubble point temperature [K].
- **y** (*ndarray*) – Vapor phase molar composition.

Examples

```
>>> import thermosteam as tmo
>>> import numpy as np
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> BP = tmo.equilibrium.BubblePoint(chemicals)
>>> BP.solve_Ty(z=np.array([0.6, 0.4]), P=101325)
(353.7543, array([0.381, 0.619]))
```

solve_Py(*z*, *T*)

Bubble point at given composition and temperature.

Parameters

- **z** (*ndarray*) – Molar composotion.
- **T** (*float*) – Temperature [K].

Returns

- **P** (*float*) – Bubble point pressure [Pa].
- **y** (*ndarray*) – Vapor phase molar composition.

Examples

```
>>> import thermosteam as tmo
>>> import numpy as np
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> BP = tmo.equilibrium.BubblePoint(chemicals)
>>> BP.solve_Py(z=np.array([0.703, 0.297]), T=352.28)
(91830.9798, array([0.419, 0.581]))
```

1.13.4 DewPoint

class thermosteam.equilibrium.**DewPoint**(*chemicals=()*, *thermo=None*)

Create a DewPoint object that returns dew point values when called with a composition and either a temperture (T) or pressure (P).

Parameters

- **chemicals=None** (*Iterable[Chemical], optional*) –
- **thermo=None** (*Thermo, optional*) –

Examples

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> DP = tmo.equilibrium.DewPoint(chemicals)
>>> # Solve for dew point at constant temperautre
>>> molar_composition = (0.5, 0.5)
>>> dp = DP(z=molar_composition, T=355)
>>> dp
DewPointValues(T=355.00, P=91970, IDs=('Water', 'Ethanol'), z=[0.5 0.5], x=[0.851 0.
↪149])
>>> # Note that the result is a DewPointValues object which contain all results as␣
↪attibutes
>>> (dp.T, round(dp.P), dp.IDs, dp.z, dp.x)
(355, 91970, ('Water', 'Ethanol'), array([0.5, 0.5]), array([0.851, 0.149]))
>>> # Solve for dew point at constant pressure
>>> DP(z=molar_composition, P=2*101324)
DewPointValues(T=376.26, P=202648, IDs=('Water', 'Ethanol'), z=[0.5 0.5], x=[0.832␣
↪0.168])
```

__call__(*z*, *, *T=None*, *P=None*)

Call self as a function.

solve_Tx(*z*, *P*)

Dew point given composition and pressure.

Parameters

- **z** (*ndarray*) – Molar composition.
- **P** (*float*) – Pressure [Pa].

Returns

- **T** (*float*) – Dew point temperature [K].
- **x** (*numpy.ndarray*) – Liquid phase molar composition.

Examples

```
>>> import thermosteam as tmo
>>> import numpy as np
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> DP = tmo.equilibrium.DewPoint(chemicals)
>>> DP.solve_Tx(z=np.array([0.5, 0.5]), P=101325)
(357.451847, array([0.849, 0.151]))
```

solve_Px(*z*, *T*)

Dew point given composition and temperature.

Parameters

- **z** (*ndarray*) – Molar composition.
- **T** (*float*) – Temperature (K).

Returns

- **P** (*float*) – Dew point pressure (Pa).
- **x** (*ndarray*) – Liquid phase molar composition.

Examples

```
>>> import thermosteam as tmo
>>> import numpy as np
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> DP = tmo.equilibrium.DewPoint(chemicals)
>>> DP.solve_Px(z=np.array([0.5, 0.5]), T=352.28)
(82444.29876, array([0.853, 0.147]))
```

1.13.5 activity_coefficients

class thermosteam.equilibrium.activity_coefficients.**ActivityCoefficients**

Abstract class for the estimation of activity coefficients. Non-abstract subclasses should implement the following methods:

__init__(self, chemicals: Iterable[Chemicals]):
Should use pure component data from chemicals to setup future calculations of activity coefficients.

__call__(self, x: 1d array, T: float):
Should accept an array of liquid molar compositions x, and temperature T (in Kelvin), and return an array of activity coefficients. Note that the molar compositions must be in the same order as the chemicals defined when creating the ActivityCoefficients object.

property chemicals

tuple[Chemical] All chemicals involved in the calculation of activity coefficients.

class thermosteam.equilibrium.activity_coefficients.**IdealActivityCoefficients**(*chemicals*)

Create an IdealActivityCoefficients object that estimates all activity coefficients to be 1 when called with a composition and a temperature (K).

Parameters
chemicals (*Iterable[Chemical]*) –

__call__(*xs*, *T*)

Call self as a function.

class thermosteam.equilibrium.activity_coefficients.**GroupActivityCoefficients**(*chemicals*)

Abstract class for the estimation of activity coefficients using group contribution methods.

Parameters
chemicals (*Iterable[Chemical]*) –

activity_coefficients(*x*, *T*)

Return activity coefficients of chemicals with defined functional groups.

Parameters

- **x** (*array_like*) – Molar fractions
- **T** (*float*) – Temperature [K]

__call__(*x*, *T*)

Return activity coefficients.

Parameters

- **x** (*array_like*) – Molar fractions
- **T** (*float*) – Temperature [K]

Notes

Activity coefficients of chemicals with missing groups are default to 1.

class thermosteam.equilibrium.activity_coefficients.**DortmundActivityCoefficients**(*chemicals*)

Create a DortmundActivityCoefficients that estimates activity coefficients using the Dortmund UNIFAC group contribution method when called with a composition and a temperature (K).

Parameters

chemicals (*Iterable[Chemical]*) –

Examples

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['Water', 'Ethanol'], cache=True)
>>> Gamma = tmo.equilibrium.DortmundActivityCoefficients(chemicals)
>>> composition = [0.5, 0.5]
>>> T = 350.
>>> Gamma(composition, T)
array([1.475, 1.242])
```

```
>>> chemicals = tmo.Chemicals(['Dodecane', 'Tridecane'], cache=True)
>>> Gamma = tmo.equilibrium.DortmundActivityCoefficients(chemicals)
>>> # Note how both hydrocarbons have similar lenghts and structure,
>>> # so activities should be very close
>>> Gamma([0.5, 0.5], 350.)
array([1., 1.])
```

```
>>> chemicals = tmo.Chemicals(['Water', 'O2'], cache=True)
>>> Gamma = tmo.equilibrium.DortmundActivityCoefficients(chemicals)
>>> # The following warning is issued because O2 does not have Dortmund groups
>>> # RuntimeWarning: O2 has no defined Dortmund groups;
>>> # functional group interactions are ignored
>>> Gamma([0.5, 0.5], 350.)
array([1., 1.])
```

class thermosteam.equilibrium.activity_coefficients.**UNIFACActivityCoefficients**(*chemicals*)

Create a UNIFACActivityCoefficients that estimates activity coefficients using the UNIFAC group contribution method when called with a composition and a temperature (K).

Parameters

chemicals (*Iterable[Chemical]*) –

1.13.6 fugacity_coefficients

class thermosteam.equilibrium.fugacity_coefficients.**FugacityCoefficients**(*chemicals*)

Abstract class for the estimation of fugacity coefficients. Non-abstract subclasses should implement the following methods:

__init__(self, chemicals: Iterable[Chemicals]):
Should use pure component data from chemicals to setup future calculations of fugacity coefficients.

__call__(self, y: 1d array, T: float, P:float):
Should accept an array of vapor molar compositions *y*, temperature *T* (in Kelvin), and pressure *P* (in Pascal), and return an array of fugacity coefficients. Note that the molar compositions must be in the same order as the chemicals defined when creating the FugacityCoefficients object.

class thermosteam.equilibrium.fugacity_coefficients.**IdealFugacityCoefficients**(*chemicals*)

Create an IdealFugacityCoefficients object that estimates all fugacity coefficients to be 1 when called with composition, temperature (K), and pressure (Pa).

Parameters
chemicals (*Iterable[Chemical]*) –

property chemicals

tuple[Chemical] All chemicals involved in the calculation of fugacity coefficients.

__call__(*y*, *T*, *P*)

Call self as a function.

1.13.7 poynting_correction_factors

class thermosteam.equilibrium.poynting_correction_factors.**PoyntingCorrectionFactors**(*chemicals*)

Abstract class for the estimation of Poynting correction factors. Non-abstract subclasses should implement the following methods:

__init__(self, chemicals: Iterable[Chemicals]):
Should use pure component data from chemicals to setup future calculations of Poynting correction factors.

__call__(self, y: 1d array, T: float):
Should accept an array of vapor molar compositions *y*, and temperature *T* (in Kelvin), and return an array of Poynting correction factors. Note that the molar compositions must be in the same order as the chemicals defined when creating the PoyntingCorrectionFactors object.

class thermosteam.equilibrium.poynting_correction_factors.**IdealPoyntingCorrectionFactors**(*chemicals*)

Create an IdealPoyntingCorrectionFactor object that estimates all poynting correction factors to be 1 when called with composition and temperature (K).

Parameters
chemicals (*Iterable[Chemical]*) –

__call__(*y*, *T*)

Call self as a function.

1.13.8 plot_equilibrium

thermosteam.equilibrium.plot_equilibrium.**plot_vle_binary_phase_envelope**(*chemicals*, *T=None*, *P=None*, *color=None*, *thermo=None*)

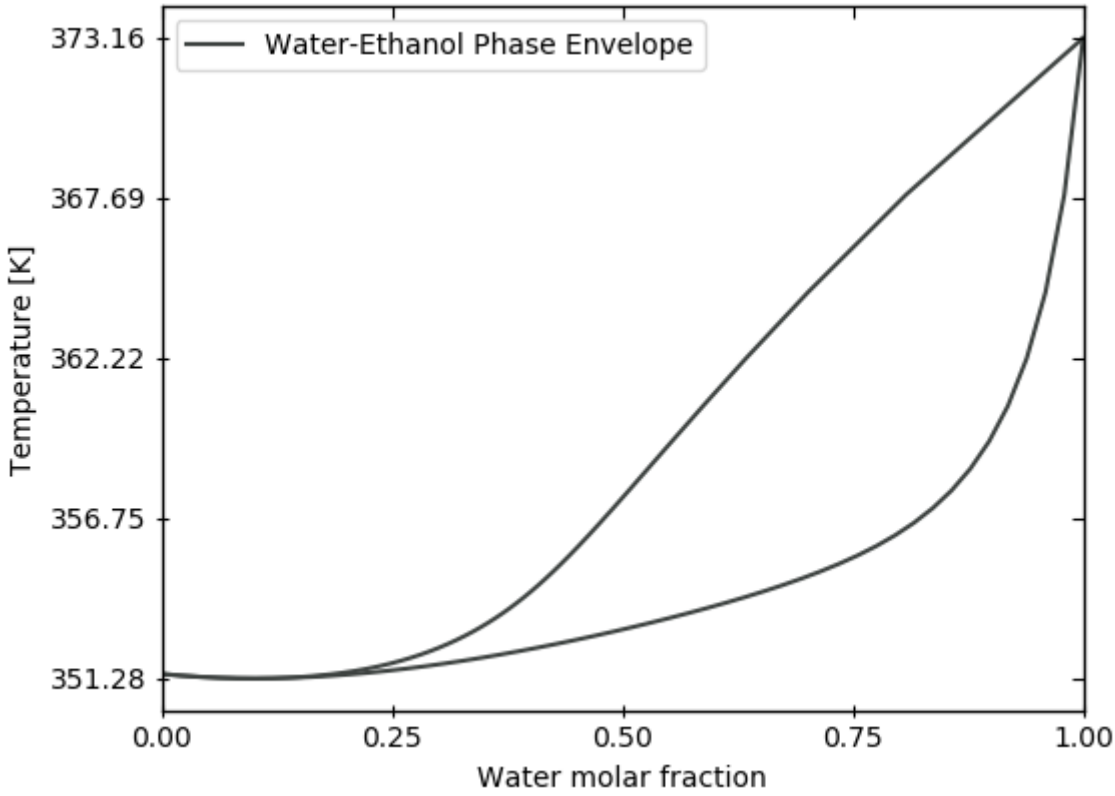
Plot the binary phase envelope of two chemicals at a given temperature or pressure.

Parameters

- **chemicals** (*Iterable[Chemical or str]*) – Chemicals in equilibrium.
- **T** (*float, optional*) – Temperature [K].
- **P** (*float, optional*) – Pressure [Pa].
- **color** (*str, optional*) – Color of line plot.
- **thermo** (Thermo, *optional*) – Thermodynamic property package.

Examples

```
>>> # from thermosteam import equilibrium as eq
>>> # eq.plot_vle_binary_phase_envelope(['Ethanol', 'Water'], P=101325)
```



thermosteam.equilibrium.plot_equilibrium.**plot_lle_ternary_diagram**(*carrier*, *solvent*, *solute*, *T*, *P=101325*, *thermo=None*, *color=None*, *tie_line_points=None*, *tie_color=None*, *N_tie_lines=15*, *N_equilibrium_grids=15*)

Plot the ternary phase diagram of chemicals in liquid-liquid equilibrium.

Parameters

- **carrier** (Chemical) –
- **solvent** (Chemical) –
- **solute** (Chemical) –
- **T** (*float, optional*) – Temperature [K].
- **P** (*float, optional*) – Pressure [Pa]. Defaults to 101325.
- **thermo** (Thermo, *optional*) – Thermodynamic property package.
- **color** (*str, optional*) – Color of equilibrium line.
- **tie_line_points** (*1d array(size=3), optional*) – Additional composition points to create tie lines.
- **tie_color** (*str, optional*) – Color of tie lines.
- **N_tie_lines** (*int, optional*) – Number of tie lines. The default is 15.
- **N_equilibrium_grids** (*int, optional*) – Number of solute composition points to plot. The default is 15.

Examples

```
>>> # from thermosteam import equilibrium as eq
>>> # eq.plot_lle_ternary_diagram('Water', 'Ethanol', 'EthylAcetate', T=298.15)
```

1.14 reaction

Use the reaction subpackage to perform reactions on chemical arrays.

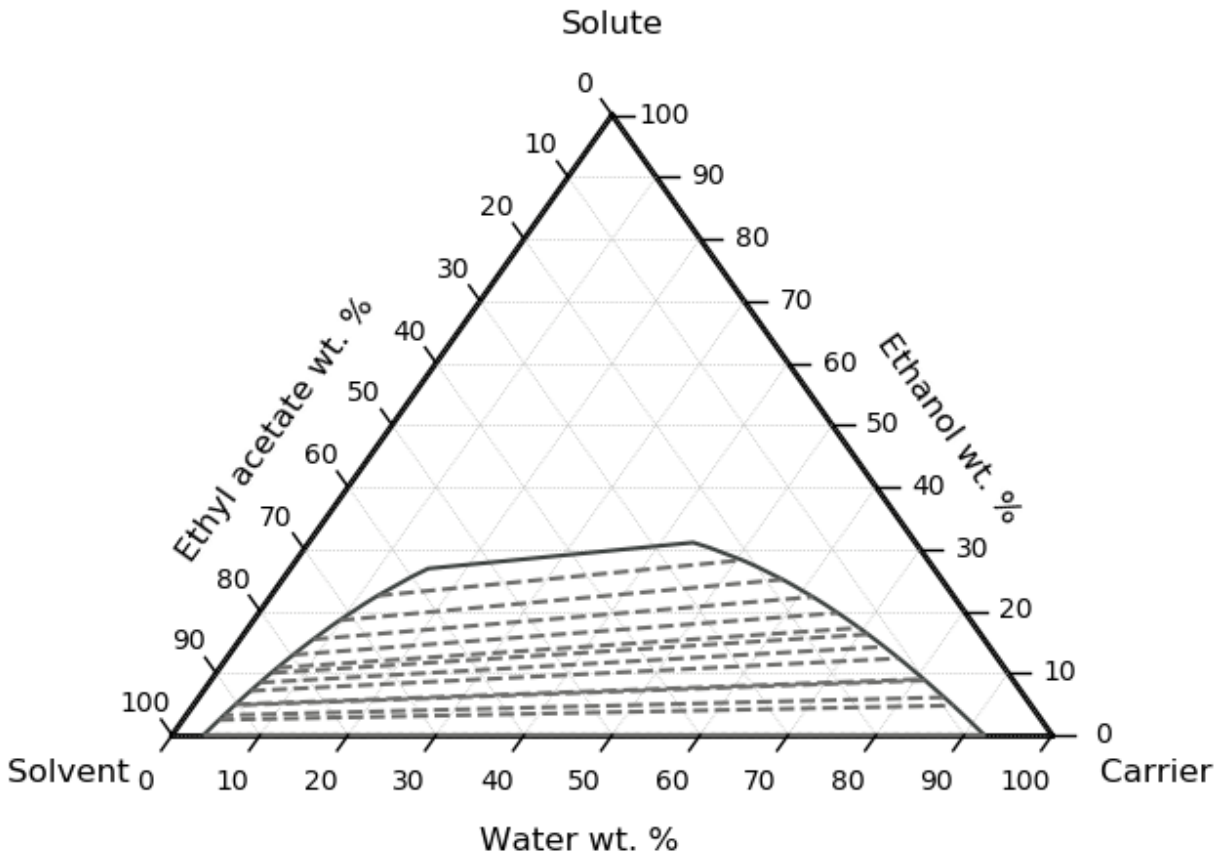
1.14.1 Reaction

class thermosteam.reaction.**Reaction**(*reaction*, *reactant*, *X*, *chemicals=None*, *basis='mol'*, ***, *phases=None*, *check_mass_balance=False*, *check_atomic_balance=False*, *correct_atomic_balance=False*, *correct_mass_balance=False*)

Create a Reaction object which defines a stoichiometric reaction and conversion. A Reaction object is capable of reacting the material flow rates of a *thermosteam.Stream* object.

Parameters

- **reaction** (*dict or str*) – A dictionary of stoichiometric coefficients or a stoichiometric equation written as: i1 R1 + … + in Rn -> j1 P1 + … + jm Pm


Solute
Ethyl acetate wt. %
Ethanol wt. %
Solvent
Carrier
Water wt. %
0
10
20
30
40
50
60
70
80
90
100

- **reactant** (*str*) – ID of reactant compound.
- **X** (*float*) – Reactant conversion (fraction).
- **chemicals=None** (Chemicals, *defaults to settings.chemicals.*) – Chemicals corresponing to each entry in the stoichiometry array.
- **basis='mol'** (*{'mol', 'wt'}*) – Basis of reaction.
- **check_mass_balance=False** (*bool*) – Whether to check if mass is not created or destroyed.
- **correct_mass_balance=False** (*bool*) – Whether to make sure mass is not created or destroyed by varying the reactant stoichiometric coefficient.
- **check_atomic_balance=False** (*bool*) – Whether to check if stoichiometric balance by atoms cancel out.
- **correct_atomic_balance=False** (*bool*) – Whether to correct the stoichiometry according to the atomic balance.

Notes

A reaction object can react either a stream or an array. When a stream is passed, it reacts either the mol or mass flow rate according to the basis of the reaction object. When an array is passed, the array elements are reacted regardless of what basis they are associated with.

Warning: Negative conversions and conversions above 1.0 are fair game (allowed), but may lead to odd/infeasible values when reacting a stream.

Examples

Electrolysis of water to molecular hydrogen and oxygen:

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['H2O', 'H2', 'O2'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> reaction = tmo.Reaction('2H2O,l -> 2H2,g + O2,g', reactant='H2O', X=0.7)
>>> reaction.show() # Note that the default basis is by 'mol'
Reaction (by mol):
 stoichiometry             reactant    X[%]
 H2O,l -> H2,g + 0.5 O2,g  H2O,l       70.00
>>> reaction.reactant # The reactant is a tuple of phase and chemical ID
('l', 'H2O')
>>> feed = tmo.Stream('feed', H2O=100)
>>> feed.phases = ('g', 'l') # Gas and liquid phases must be available
>>> reaction(feed) # Call to run reaction on molar flow
>>> feed.show() # Notice how 70% of water was converted to product
MultiStream: feed
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): (g) H2   70
                     O2   35
                 (l) H2O  30
```

Let's change to a per 'wt' basis:

```
>>> reaction.basis = 'wt'
>>> reaction.show()
Reaction (by wt):
 stoichiometry                      reactant    X[%]
 H2O,l -> 0.112 H2,g + 0.888 O2,g   H2O,l       70.00
```

Although we changed the basis, the end result is the same if we pass a stream:

```
>>> feed = tmo.Stream('feed', H2O=100)
>>> feed.phases = ('g', 'l')
>>> reaction(feed) # Call to run reaction on mass flow
>>> feed.show() # Notice how 70% of water was converted to product
MultiStream: feed
 phases: ('g', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): (g) H2   70
                     O2   35
                 (l) H2O  30
```

If chemicals phases are not specified, Reaction objects can react a any single phase Stream object (regardless of phase):

```
>>> reaction = tmo.Reaction('2H2O -> 2H2 + O2', reactant='H2O', X=0.7)
>>> feed = tmo.Stream('feed', H2O=100, phase='g')
>>> reaction(feed)
>>> feed.show()
Stream: feed
 phase: 'g', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): H2O  30
                 H2   70
                 O2   35
```

Alternatively, it's also possible to react an array (instead of a stream):

```
>>> import numpy as np
>>> array = np.array([100., 0. , 0.])
>>> reaction(array)
>>> array
array([30., 70., 35.])
```

Reaction objects with the same reactant can be added together:

```
>>> tmo.settings.set_thermo(['Glucose', 'Ethanol', 'H2O', 'O2', 'CO2'])
>>> fermentation = tmo.Reaction('Glucose + O2 -> Ethanol + CO2', reactant='Glucose',
↪ X=0.7)
>>> combustion = tmo.Reaction('Glucose + O2 -> H2O + CO2', reactant='Glucose', X=0.
↪2)
>>> mixed_reaction = fermentation + combustion
>>> mixed_reaction.show()
Reaction (by mol):
 stoichiometry                                  reactant    X[%]
 Glucose + O2 -> 0.778 Ethanol + 0.222 H2O + CO2  Glucose    90.00
```

Note how conversions are added and the stoichiometry rescales to a per reactant basis. Conversly, reaction objects may be substracted as well:

```
>>> combustion = mixed_reaction - fermentation
>>> combustion.show()
Reaction (by mol):
 stoichiometry                reactant    X[%]
 Glucose + O2 -> H2O + CO2  Glucose    20.00
```

When a Reaction object is multiplied (or divided), a new Reaction object with the conversion multiplied (or divided) is returned:

```
>>> combustion_multiplied = 2 * combustion
>>> combustion_multiplied.show()
Reaction (by mol):
 stoichiometry                reactant    X[%]
 Glucose + O2 -> H2O + CO2  Glucose    40.00
>>> fermentation_divided = fermentation / 2
>>> fermentation_divided.show()
Reaction (by mol):
 stoichiometry                  reactant    X[%]
 Glucose + O2 -> Ethanol + CO2  Glucose    35.00
```

copy(*basis=None*)

Return copy of Reaction object.

force_reaction(*material*)

React material ignoring feasibility checks.

product_yield(*product*, *basis=None*)

Return yield of product per reactant.

adiabatic_reaction(*stream*)

React stream material adiabatically, accounting for the change in enthalpy due to the heat of reaction.

Examples

Note how the stream temperature changed after the reaction due to the heat of reaction:

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['H2', 'O2', 'H2O'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> reaction = tmo.Reaction('2H2 + O2 -> 2H2O', reactant='H2', X=0.7)
>>> s1 = tmo.Stream('s1', H2=10, O2=20, H2O=1000, T=373.15, phase='g')
>>> s2 = tmo.Stream('s2')
>>> s2.copy_like(s1) # s1 and s2 are the same
>>> s1.show() # Before reaction
Stream: s1
 phase: 'g', T: 373.15 K, P: 101325 Pa
 flow (kmol/hr): H2   10
                 O2   20
                 H2O  1e+03
```

```
>>> reaction.show()
Reaction (by mol):
```

(continues on next page)

(continued from previous page)

```
 stoichiometry       reactant    X[%]
 H2 + 0.5 O2 -> H2O  H2         70.00
```

```
>>> reaction(s1)
>>> s1.show() # After isothermal reaction
Stream: s1
 phase: 'g', T: 373.15 K, P: 101325 Pa
 flow (kmol/hr): H2   3
                 O2   16.5
                 H2O  1.01e+03
```

```
>>> reaction.adiabatic_reaction(s2)
>>> s2.show() # After adiabatic reaction
Stream: s2
 phase: 'g', T: 421.6 K, P: 101325 Pa
 flow (kmol/hr): H2   3
                 O2   16.5
                 H2O  1.01e+03
```

property dH

Heat of reaction at given conversion. Units are in either J/mol-reactant or J/g-reactant; depending on basis.

> **Warning:** Latents heats of vaporization are not accounted for; only heats of formation are included in this term. Note that heats of vaporization are temperature dependent and cannot be calculated using a Reaction object.

property X

[float] Reaction converion as a fraction.

property stoichiometry

[array] Stoichiometry coefficients.

property istoichiometry

[ChemicalIndexer] Stoichiometry coefficients.

property reactant

[str] Reactant associated to conversion.

property MWs

[1d array] Molecular weights of all chemicals [g/mol].

property basis

{'mol', 'wt'} Basis of reaction

check_mass_balance(*tol=0.001*)

Check that stoichiometric mass balance is correct.

check_atomic_balance(*tol=0.001*)

Check that stoichiometric atomic balance is correct.

correct_mass_balance(*variable=None*)

Make sure mass is not created or destroyed by varying the reactant stoichiometric coefficient.

correct_atomic_balance(*constants=None*)

Correct stoichiometry coffecients to satisfy atomic balance.

Parameters

constants (*str, optional*) – IDs of chemicals for which stoichiometric coefficients are held constant.

Examples

Balance glucose fermentation to ethanol:

```
>>> import thermosteam as tmo
>>> from biorefineries import lipidcane as lc
>>> tmo.settings.set_thermo(lc.chemicals)
>>> fermentation = tmo.Reaction('Glucose + O2 -> Ethanol + CO2',
...                             reactant='Glucose',  X=0.9)
>>> fermentation.correct_atomic_balance()
>>> fermentation.show()
Reaction (by mol):
 stoichiometry                reactant    X[%]
 Glucose -> 2 Ethanol + 2 CO2 Glucose    90.00
```

Balance methane combustion:

```
>>> combustion = tmo.Reaction('CH4 + O2 -> Water + CO2',
...                           reactant='CH4', X=1)
>>> combustion.correct_atomic_balance()
>>> combustion.show()
Reaction (by mol):
 stoichiometry               reactant    X[%]
 2 O2 + CH4 -> 2 Water + CO2 CH4       100.00
```

Balance electrolysis of water (with chemical phases specified):

```
>>> electrolysis = tmo.Reaction('H2O,l -> H2,g + O2,g',
...                             chemicals=tmo.Chemicals(['H2O', 'H2', 'O2']),
...                             reactant='H2O', X=1)
>>> electrolysis.correct_atomic_balance()
>>> electrolysis.show()
Reaction (by mol):
 stoichiometry            reactant    X[%]
 H2O,l -> H2,g + 0.5 O2,g H2O,l     100.00
```

Note that if the reaction is underspecified, there are infinite ways to balance the reaction and a runtime error is raised:

```
>>> rxn_underspecified = tmo.Reaction('CH4 + Glucose + O2 -> Water + CO2',
...                                   reactant='CH4', X=1)
>>> rxn_underspecified.correct_atomic_balance()
Traceback (most recent call last):
RuntimeError: reaction stoichiometry is underspecified; pass the
`constants` argument to the `<Reaction>.correct_atomic_balance` method
to specify which stoichiometric coefficients to hold constant
```

Chemical coefficients can be held constant to prevent this error:

```
>>> rxn_underspecified = tmo.Reaction('CH4 + Glucose + O2 -> Water + CO2',
...                                   reactant='CH4', X=1)
>>> rxn_underspecified.correct_atomic_balance(['Glucose', 'CH4'])
>>> rxn_underspecified.show()
Reaction (by mol):
 stoichiometry                          reactant    X[%]
 Glucose + 8 O2 + CH4 -> 8 Water + 7 CO2  CH4       100.00
```

1.14.2 ParallelReaction

class thermosteam.reaction.**ParallelReaction**(*reactions*)

Create a ParallelReaction object from Reaction objects. When called, it returns the change in material due to all parallel reactions.

Parameters

reactions (*Iterable[Reaction]*) –

Examples

Run two reactions in parallel:

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['H2', 'Ethanol', 'CH4', 'O2', 'CO2', 'H2O'],␣
↪cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> kwargs = dict(phases='lg', correct_atomic_balance=True)
>>> reaction = tmo.ParallelReaction([
...     #            Reaction definition                    Reactant            ␣
↪Conversion
...     tmo.Reaction('H2,g + O2,g -> 2H2O,g',               reactant='H2',       X=0.
↪7, **kwargs),
...     tmo.Reaction('Ethanol,l + O2,g -> CO2,g + 2H2O,g',  reactant='Ethanol',  X=0.
↪1, **kwargs)
... ])
>>> reaction.reactants # Note that reactants are tuples of phase and ID pairs.
(('g', 'H2'), ('l', 'Ethanol'))
```

```
>>> reaction.show()
ParallelReaction (by mol):
index  stoichiometry                          reactant     X[%]
[0]    H2,g + 0.5 O2,g -> H2O,g               H2,g        70.00
[1]    3 O2,g + Ethanol,l -> 2 CO2,g + 3 H2O,g  Ethanol,l   10.00
```

```
>>> s1 = tmo.MultiStream('s1', T=373.15,
...                      l=[('Ethanol', 10)],
...                      g=[('H2', 10), ('CH4', 5), ('O2', 100), ('H2O', 10)])
>>> s1.show() # Before reaction
MultiStream: s1
 phases: ('g', 'l'), T: 373.15 K, P: 101325 Pa
```

(continues on next page)

(continued from previous page)

```
 flow (kmol/hr): (g) H2       10
                     CH4      5
                     O2       100
                     H2O      10
                 (l) Ethanol  10
```

```
>>> reaction(s1)
>>> s1.show() # After isothermal reaction
MultiStream: s1
 phases: ('g', 'l'), T: 373.15 K, P: 101325 Pa
 flow (kmol/hr): (g) H2       3
                     CH4      5
                     O2       93.5
                     CO2      2
                     H2O      20
                 (l) Ethanol  9
```

Reaction items are accessible:

```
>>> reaction[0].show()
ReactionItem (by mol):
 stoichiometry             reactant    X[%]
 H2,g + 0.5 O2,g -> H2O,g  H2,g       70.00
```

Note that changing the conversion of a reaction item changes the conversion of its parent reaction set:

```
>>> reaction[0].X = 0.5
>>> reaction.show()
ParallelReaction (by mol):
index  stoichiometry                              reactant     X[%]
[0]    H2,g + 0.5 O2,g -> H2O,g                   H2,g        50.00
[1]    3 O2,g + Ethanol,l -> 2 CO2,g + 3 H2O,g    Ethanol,l   10.00
```

Reactions subsets can be made as well:

```
>>> reaction[:1].show()
ParallelReaction (by mol):
index  stoichiometry             reactant    X[%]
[0]    H2,g + 0.5 O2,g -> H2O,g  H2,g       50.00
```

Get net reaction conversion of reactants as a material indexer:

```
>>> mi = reaction.X_net
>>> mi.show()
MaterialIndexer:
 (g) H2        0.5
 (l) Ethanol   0.1
>>> mi['g', 'H2']
0.5
```

If no phases are specified for a reaction set, the *X_net* property returns a ChemicalIndexer:

```
>>> kwargs = dict(correct_atomic_balance=True)
>>> reaction = tmo.ParallelReaction([
...     #            Reaction definition            Reactant             Conversion
...     tmo.Reaction('H2 + O2 -> 2H2O',             reactant='H2',       X=0.7,␣
↪**kwargs),
...     tmo.Reaction('Ethanol + O2 -> CO2 + 2H2O',  reactant='Ethanol',  X=0.1,␣
↪**kwargs)
... ])
>>> ci = reaction.X_net
>>> ci.show()
ChemicalIndexer:
 H2       0.7
 Ethanol  0.1
>>> ci['H2']
0.7
```

force_reaction(*material*)

React material ignoring feasibility checks.

adiabatic_reaction(*stream*)

React stream material adiabatically, accounting for the change in enthalpy due to the heat of reaction.

Examples

Note how the stream temperature changed after the reaction due to the heat of reaction:

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['H2', 'CH4', 'O2', 'CO2', 'H2O'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> reaction = tmo.ParallelReaction([
...     #            Reaction definition          Reactant    Conversion
...     tmo.Reaction('2H2 + O2 -> 2H2O',        reactant='H2',  X=0.7),
...     tmo.Reaction('CH4 + O2 -> CO2 + 2H2O',  reactant='CH4', X=0.1)
... ])
>>> s1 = tmo.Stream('s1', H2=10, CH4=5, O2=100, H2O=100, T=373.15, phase='g')
>>> s2 = tmo.Stream('s2')
>>> s1.show() # Before reaction
Stream: s1
 phase: 'g', T: 373.15 K, P: 101325 Pa
 flow (kmol/hr): H2   10
                 CH4  5
                 O2   100
                 H2O  100
```

```
>>> reaction.show()
ParallelReaction (by mol):
index  stoichiometry            reactant    X[%]
[0]    H2 + 0.5 O2 -> H2O       H2         70.00
[1]    CH4 + O2 -> CO2 + 2 H2O  CH4        10.00
```

```
>>> reaction.adiabatic_reaction(s1)
>>> s1.show() # After adiabatic reaction
```

(continues on next page)

(continued from previous page)

```
Stream: s1
 phase: 'g', T: 666.21 K, P: 101325 Pa
 flow (kmol/hr): H2   3
                 CH4  4.5
                 O2   96
                 CO2  0.5
                 H2O  108
```

reduce()

Return a new Parallel reaction object that combines reaction with the same reactant together, reducing the number of reactions.

property X_net

[ChemicalIndexer] Net reaction conversion of reactants.

1.14.3 SeriesReaction

class thermosteam.reaction.**SeriesReaction**(*reactions*)

Create a ParallelReaction object from Reaction objects. When called, it returns the change in material due to all reactions in series.

Parameters

reactions (*Iterable[Reaction]*) –

force_reaction(*material*)

React material ignoring feasibility checks.

adiabatic_reaction(*stream*)

React stream material adiabatically, accounting for the change in enthalpy due to the heat of reaction.

Examples

Note how the stream temperature changed after the reaction due to the heat of reaction:

```
>>> import thermosteam as tmo
>>> chemicals = tmo.Chemicals(['CH4', 'CO','O2', 'CO2', 'H2O'], cache=True)
>>> tmo.settings.set_thermo(chemicals)
>>> reaction = tmo.SeriesReaction([
...     #            Reaction definition                Reactant      ␣
↪Conversion
...     tmo.Reaction('2CH4 + 3O2 -> 2CO + 4H2O',       reactant='CH4',    X=0.
↪7),
...     tmo.Reaction('2CO + O2 -> 2CO2',               reactant='CO',     X=0.
↪1)
...     ])
>>> s1 = tmo.Stream('s1', CH4=5, O2=100, H2O=100, T=373.15, phase='g')
>>> s1.show() # Before reaction
Stream: s1
 phase: 'g', T: 373.15 K, P: 101325 Pa
 flow (kmol/hr): CH4  5
                 O2   100
                 H2O  100
```

```
>>> reaction.show()
SeriesReaction (by mol):
index  stoichiometry               reactant    X[%]
[0]    CH4 + 1.5 O2 -> CO + 2 H2O  CH4        70.00
[1]    CO + 0.5 O2 -> CO2          CO         10.00
```

```
>>> reaction.adiabatic_reaction(s1)
>>> s1.show() # After adiabatic reaction
Stream: s1
 phase: 'g', T: 649.84 K, P: 101325 Pa
 flow (kmol/hr): CH4  1.5
                 CO   3.15
                 O2   94.6
                 CO2  0.35
                 H2O  107
```

property X_net

[ChemicalIndexer] Net reaction conversion of reactants.

1.15 indexer

Use the indexer subpackage for fast indexing of chemical arrays.

1.15.1 Indexer

class thermosteam.indexer.**Indexer**

Abstract class for fast indexing.

1.15.2 ChemicalIndexer

class thermosteam.indexer.**ChemicalIndexer**(*phase=LockedPhase(None)*, *units=None*, *chemicals=None*, ***ID_data*)

Create a ChemicalIndexer that can index a single-phase, 1d-array given chemical IDs.

Parameters

- **phase** (*[str or PhaseContainer] {'s', 'l', 'g', 'S', 'L', 'G'}*) – Phase of data.
- **units** (*str*) – Units of measure of input data.
- **chemicals** (Chemicals) – Required to define the chemicals that are present.
- ****ID_data** (*float*) – ID-value pairs

Notes

A ChemicalIndexer does not have any units defined. To use units of measure, use the *ChemicalMolarIndexer*, *ChemicalMassIndexer*, or *ChemicalVolumetricIndexer*.

show(*N=None*)

Print all specifications.

Parameters

N (*int, optional*) – Number of compounds to display.

1.15.3 MaterialIndexer

class thermosteam.indexer.**MaterialIndexer**(*phases=None*, *units=None*, *chemicals=None*, ***phase_data*)

Create a MaterialIndexer that can index a multi-phase, 2d-array given the phase and chemical IDs.

Parameters

- **phases** (*tuple['s', 'l', 'g', 'S', 'L', 'G']*) – Phases of data rows.
- **units** (*str*) – Units of measure of input data.
- **chemicals** (Chemicals) – Required to define the chemicals that are present.
- ****phase_data** (*tuple[str, float]*) – phase-(ID, value) pairs

Notes

A MaterialIndexer does not have any units defined. To use units of measure, use the *MolarIndexer*, *MassIndexer*, or *VolumetricIndexer*.

iter_composition()

Iterate over phase-composition pairs.

show(*N=None*)

Print all specifications.

Parameters

N (*int, optional*) – Number of compounds to display.

1.15.4 ChemicalMolarFlowIndexer

class thermosteam.indexer.**ChemicalMolarFlowIndexer**(*phase=LockedPhase(None)*, *units=None*, *chemicals=None*, ***ID_data*)

Create a ChemicalMolarFlowIndexer that can index a single-phase, 1d-array given chemical IDs.

Parameters

- **phase** (*[str or PhaseContainer] {'s', 'l', 'g', 'S', 'L', 'G'}*) – Phase of data.
- **units** (*str*) – Units of measure of input data.
- **chemicals** (Chemicals) – Required to define the chemicals that are present.
- ****ID_data** (*float*) – ID-value pairs

by_mass()

Return a ChemicalMassFlowIndexer that references this object's molar data.

by_volume(*TP*)

Return a ChemicalVolumetricFlowIndexer that references this object's molar data.

Parameters

TP (ThermalCondition) –

1.15.5 ChemicalMassFlowIndexer

class thermosteam.indexer.**ChemicalMassFlowIndexer**(*phase=LockedPhase(None)*, *units=None*, *chemicals=None*, ***ID_data*)

Create a ChemicalMassFlowIndexer that can index a single-phase, 1d-array given chemical IDs.

Parameters

- **phase** (*[str or PhaseContainer] {'s', 'l', 'g', 'S', 'L', 'G'}*) – Phase of data.
- **units** (*str*) – Units of measure of input data.
- **chemicals** (Chemicals) – Required to define the chemicals that are present.
- ****ID_data** (*float*) – ID-value pairs

1.15.6 ChemicalVolumetricFlowIndexer

class thermosteam.indexer.**ChemicalVolumetricFlowIndexer**(*phase=LockedPhase(None)*, *units=None*, *chemicals=None*, ***ID_data*)

Create a ChemicalVolumetricFlowIndexer that can index a single-phase, 1d-array given chemical IDs.

Parameters

- **phase** (*[str or PhaseContainer] {'s', 'l', 'g', 'S', 'L', 'G'}*) – Phase of data.
- **units** (*str*) – Units of measure of input data.
- **chemicals** (Chemicals) – Required to define the chemicals that are present.
- ****ID_data** (*float*) – ID-value pairs

1.15.7 MolarFlowIndexer

class thermosteam.indexer.**MolarFlowIndexer**(*phases=None*, *units=None*, *chemicals=None*, ***phase_data*)

Create a MolarFlowIndexer that can index a multi-phase, 2d-array given the phase and chemical IDs.

Parameters

- **phases** (*tuple['s', 'l', 'g', 'S', 'L', 'G']*) – Phases of data rows.
- **units** (*str*) – Units of measure of input data.
- **chemicals** (Chemicals) – Required to define the chemicals that are present.
- ****phase_data** (*tuple[str, float]*) – phase-(ID, value) pairs

by_mass()

Return a MassFlowIndexer that references this object's molar data.

by_volume(*TP*)

Return a VolumetricFlowIndexer that references this object's molar data.

Parameters

TP (ThermalCondition) –

1.15.8 MassFlowIndexer

class thermosteam.indexer.**MassFlowIndexer**(*phases=None*, *units=None*, *chemicals=None*, ***phase_data*)

Create a MassFlowIndexer that can index a multi-phase, 2d-array given the phase and chemical IDs.

Parameters

- **phases** (*tuple['s', 'l', 'g', 'S', 'L', 'G']*) – Phases of data rows.
- **units** (*str*) – Units of measure of input data.
- **chemicals** (Chemicals) – Required to define the chemicals that are present.
- ****phase_data** (*tuple[str, float]*) – phase-(ID, value) pairs

1.15.9 VolumetricFlowIndexer

class thermosteam.indexer.**VolumetricFlowIndexer**(*phases=None*, *units=None*, *chemicals=None*, ***phase_data*)

Create a VolumetricFlowIndexer that can index a multi-phase, 2d-array given the phase and chemical IDs.

Parameters

- **phases** (*tuple['s', 'l', 'g', 'S', 'L', 'G']*) – Phases of data rows.
- **units** (*str*) – Units of measure of input data.
- **chemicals** (Chemicals) – Required to define the chemicals that are present.
- ****phase_data** (*tuple[str, float]*) – phase-(ID, value) pairs

1.16 mixture

1.16.1 mixture_builders

All Mixture object builders.

thermosteam.mixture.mixture_builders.**ideal_mixture**(*chemicals*, *include_excess_energies=False*)

Create a Mixture object that computes mixture properties using ideal mixing rules.

Parameters

- **chemicals** (Chemicals) – For retrieving pure component chemical data.
- **include_excess_energies=False** (*bool*) – Whether to include excess energies in enthalpy and entropy calculations.

See also:

Mixture, *IdealMixtureModel*

Examples

```
>>> from thermosteam import Chemicals
>>> from thermosteam.mixture import ideal_mixture
>>> chemicals = Chemicals(['Water', 'Ethanol'])
>>> ideal_mixture_model = ideal_mixture(chemicals)
>>> ideal_mixture_model.Hvap([0.2, 0.8], 350)
39601.089191849824
```

1.16.2 IdealMixtureModel

class thermosteam.mixture.**IdealMixtureModel**(*models*, *var*)

Create an IdealMixtureModel object that calculates mixture properties based on the molar weighted sum of pure chemical properties.

Parameters

- **models** (*Iterable[function(T, P)]*) – Chemical property functions of temperature and pressure.
- **var** (*str*) – Description of thermodynamic variable returned.

Notes

Mixture objects can contain IdealMixtureModel objects to establish as mixture model for thermodynamic properties.

See also:

Mixture, *ideal_mixture()*

Examples

```
>>> from thermosteam.mixture import IdealMixtureModel
>>> from thermosteam import Chemicals
>>> chemicals = Chemicals(['Water', 'Ethanol'])
>>> models = [i.Psat for i in chemicals]
>>> mixture_model = IdealMixtureModel(models, 'Psat')
>>> mixture_model
<IdealMixtureModel(mol, T, P=None) -> Psat [Pa]>
>>> mixture_model([0.2, 0.8], 350)
84902.48775
```

1.16.3 Mixture

class thermosteam.mixture.**Mixture**(*rule*, *Cn*, *H*, *S*, *H_excess*, *S_excess*, *mu*, *V*, *kappa*, *Hvap*, *sigma*, *epsilon*, *include_excess_energies=False*)

Create an Mixture object for estimating mixture properties.

Parameters

- **rule** (*str*) – Description of mixing rules used.
- **Cn** (*function(phase, mol, T)*) – Molar heat capacity mixture model [J/mol/K].
- **H** (*function(phase, mol, T)*) – Enthalpy mixture model [J/mol].
- **S** (*function(phase, mol, T, P)*) – Entropy mixture model [J/mol].
- **H_excess** (*function(phase, mol, T, P)*) – Excess enthalpy mixture model [J/mol].
- **S_excess** (*function(phase, mol, T, P)*) – Excess entropy mixture model [J/mol].
- **mu** (*function(phase, mol, T, P)*) – Dynamic viscosity mixture model [Pa*s].
- **V** (*function(phase, mol, T, P)*) – Molar volume mixture model [m^3/mol].
- **kappa** (*function(phase, mol, T, P)*) – Thermal conductivity mixture model [W/m/K].
- **Hvap** (*function(mol, T)*) – Heat of vaporization mixture model [J/mol]
- **sigma** (*function(mol, T, P)*) – Surface tension mixture model [N/m].
- **epsilon** (*function(mol, T, P)*) – Relative permitivity mixture model [-]
- **include_excess_energies=False** (*bool*) – Whether to include excess energies in enthalpy and entropy calculations.

Notes

Although the mixture models are on a molar basis, this is only if the molar data is normalized before the calculation (i.e. the *mol* parameter is normalized before being passed to the model).

See also:

IdealMixtureModel, *ideal_mixture()*

rule

Description of mixing rules used.

Type
str

include_excess_energies

Whether to include excess energies in enthalpy and entropy calculations.

Type
bool

Cn(*phase*, *mol*, *T*)

Mixture molar heat capacity [J/mol/K].

mu(*phase*, *mol*, *T*, *P*)

Mixture dynamic viscosity [Pa*s].

V(*phase*, *mol*, *T*, *P*)
Mixture molar volume [m^3/mol].

kappa(*phase*, *mol*, *T*, *P*)
Mixture thermal conductivity [W/m/K].

Hvap(*mol*, *T*, *P*)
Mixture heat of vaporization [J/mol]

sigma(*mol*, *T*, *P*)
Mixture surface tension [N/m].

epsilon(*mol*, *T*, *P*)
Mixture relative permitivity [-].

H(*phase*, *mol*, *T*, *P*)
Return enthalpy [J/mol].

S(*phase*, *mol*, *T*, *P*)
Return entropy in [J/mol].

solve_T(*phase*, *mol*, *H*, *T_guess*, *P*)
Solve for temperature in Kelvin.

xsolve_T(*phase_mol*, *H*, *T_guess*, *P*)
Solve for temperature in Kelvin.

xCn(*phase_mol*, *T*)
Multi-phase mixture heat capacity [J/mol/K].

xH(*phase_mol*, *T*, *P*)
Multi-phase mixture enthalpy [J/mol].

xS(*phase_mol*, *T*, *P*)
Multi-phase mixture entropy [J/mol].

xV(*phase_mol*, *T*, *P*)
Multi-phase mixture molar volume [mol/m^3].

xmu(*phase_mol*, *T*, *P*)
Multi-phase mixture hydrolic [Pa*s].

xkappa(*phase_mol*, *T*, *P*)
Multi-phase mixture thermal conductivity [W/m/K].

1.17 separations

This module contains functions for modeling separations in unit operations.

thermosteam.separations.**split**(*feed*, *top*, *bottom*, *split*)
Run splitter mass and energy balance with mixing all input streams.

Parameters

- **feed** (Stream) – Inlet fluid.
- **top** (Stream) – Top inlet fluid.

- **bottom** (Stream) – Bottom inlet fluid
- **split** (*array_like*) – Component-wise split of feed to the top stream.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> feed = tmo.Stream(Water=35, Ethanol=10)
>>> split = 0.8
>>> effluent_a = tmo.Stream('effluent_a')
>>> effluent_b = tmo.Stream('effluent_b')
>>> tmo.separations.split(feed, effluent_a, effluent_b, split)
>>> effluent_a.show()
Stream: effluent_a
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water    28
                 Ethanol  8
>>> effluent_b.show()
Stream: effluent_b
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water    7
                 Ethanol  2
```

thermosteam.separations.**mix_and_split**(*ins*, *top*, *bottom*, *split*)

Run splitter mass and energy balance with mixing all input streams.

Parameters

- **ins** (*Iterable[Stream]*) – All inlet fluids.
- **top** (Stream) – Top inlet fluid.
- **bottom** (Stream) – Bottom inlet fluid
- **split** (*array_like*) – Component-wise split of feed to the top stream.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> feed_a = tmo.Stream(Water=20, Ethanol=5)
>>> feed_b = tmo.Stream(Water=15, Ethanol=5)
>>> split = 0.8
>>> effluent_a = tmo.Stream('effluent_a')
>>> effluent_b = tmo.Stream('effluent_b')
>>> tmo.separations.mix_and_split([feed_a, feed_b], effluent_a, effluent_b, split)
>>> effluent_a.show()
Stream: effluent_a
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water    28
                 Ethanol  8
>>> effluent_b.show()
Stream: effluent_b
```

(continues on next page)

(continued from previous page)

```
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water    7
                 Ethanol  2
```

thermosteam.separations.**adjust_moisture_content**(*retentate*, *permeate*, *moisture_content*)

Remove water from permate to adjust retentate moisture content.

Parameters

- **retentate** (Stream) –
- **permeate** (Stream) –
- **moisture_content** (*float*) – Fraction of water in retentate.

Examples

```
>>> import thermosteam as tmo
>>> Solids = tmo.Chemical('Solids', default=True, search_db=False, phase='s')
>>> tmo.settings.set_thermo(['Water', Solids])
>>> retentate = tmo.Stream('retentate', Solids=20, units='kg/hr')
>>> permeate = tmo.Stream('permeate', Water=50, Solids=0.1, units='kg/hr')
>>> moisture_content = 0.5
>>> tmo.separations.adjust_moisture_content(retentate, permeate, moisture_content)
>>> retentate.show(flow='kg/hr')
Stream: retentate
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water   20
               Solids  20
>>> permeate.show(flow='kg/hr')
Stream: permeate
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water   30
               Solids  0.1
```

Note that if not enough water is available, an InfeasibleRegion error is raised:

```
>>> retentate.imol['Water'] = permeate.imol['Water'] = 0
>>> tmo.separations.adjust_moisture_content(retentate, permeate, moisture_content)
Traceback (most recent call last):
InfeasibleRegion: not enough water; permeate moisture content is infeasible
```

thermosteam.separations.**mix_and_split_with_moisture_content**(*ins*, *retentate*, *permeate*, *split*, *moisture_content*)

Run splitter mass and energy balance with mixing all input streams and and ensuring retentate moisture content.

Parameters

- **ins** (*Iterable[Stream]*) – Inlet fluids with solids.
- **retentate** (Stream) –
- **permeate** (Stream) –
- **split** (*array_like*) – Component splits to the retentate.

- **moisture_content** (*float*) – Fraction of water in retentate.

Examples

```
>>> import thermosteam as tmo
>>> Solids = tmo.Chemical('Solids', default=True, search_db=False, phase='s')
>>> tmo.settings.set_thermo(['Water', Solids])
>>> feed = tmo.Stream('feed', Water=100, Solids=10, units='kg/hr')
>>> wash_water = tmo.Stream('wash_water', Water=10, units='kg/hr')
>>> retentate = tmo.Stream('retentate')
>>> permeate = tmo.Stream('permeate')
>>> split = [0., 1.]
>>> moisture_content = 0.5
>>> tmo.separations.mix_and_split_with_moisture_content(
...     [feed, wash_water], retentate, permeate, split, moisture_content
... )
>>> retentate.show(flow='kg/hr')
Stream: retentate
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water   10
               Solids  10
>>> permeate.show(flow='kg/hr')
Stream: permeate
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kg/hr): Water  100
```

thermosteam.separations.**partition_coefficients**(*IDs*, *top*, *bottom*)

Return partition coefficients given streams in equilibrium.

Parameters

- **top** (Stream) – Vapor fluid.
- **bottom** (Stream) – Liquid fluid.
- **IDs** (*tuple[str]*) – IDs of chemicals in equilibrium.

Returns

K – Patition coefficients in mol fraction in top stream over mol fraction in bottom stream.

Return type

1d array

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', tmo.Chemical('O2', phase='g')],␣
↪cache=True)
>>> s = tmo.Stream('s', Water=20, Ethanol=20, O2=0.1)
>>> s.vle(V=0.5, P=101325)
>>> tmo.separations.partition_coefficients(('Water', 'Ethanol'), s['g'], s['l'])
array([0.629, 1.59 ])
```

thermosteam.separations.**vle_partition_coefficients**(*top*, *bottom*)

Return VLE partition coefficients given vapor and liquid streams in equilibrium.

Parameters

- **top** (Stream) – Vapor fluid.
- **bottom** (Stream) – Liquid fluid.

Returns

- **IDs** (*tuple[str]*) – IDs for chemicals in vapor-liquid equilibrium.
- **K** (*1d array*) – Patition coefficients in mol fraction in vapor over mol fraction in liquid.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', tmo.Chemical('O2', phase='g')],␣
↪cache=True)
>>> s = tmo.Stream('s', Water=20, Ethanol=20, O2=0.1)
>>> s.vle(V=0.5, P=101325)
>>> IDs, K = tmo.separations.vle_partition_coefficients(s['g'], s['l'])
>>> IDs
('Water', 'Ethanol')
>>> K
array([0.629, 1.59 ])
```

thermosteam.separations.**lle_partition_coefficients**(*top*, *bottom*)

Return LLE partition coefficients given two liquid streams in equilibrium.

Parameters

- **top** (Stream) – Liquid fluid.
- **bottom** (Stream) – Other liquid fluid.

Returns

- **IDs** (*tuple[str]*) – IDs for chemicals in liquid-liquid equilibrium.
- **K** (*1d array*) – Patition coefficients in mol fraction in top liquid over mol fraction in bottom liquid.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Octanol'], cache=True)
>>> s = tmo.Stream('s', Water=20, Octanol=20, Ethanol=1)
>>> s.lle(T=298.15, P=101325)
>>> IDs, K = tmo.separations.lle_partition_coefficients(s['l'], s['L'])
>>> IDs
('Water', 'Ethanol', 'Octanol')
>>> K
array([6.82e+00, 2.38e-01, 3.00e-04])
```

thermosteam.separations.**partition**(*feed*, *top*, *bottom*, *IDs*, *K*, *phi=None*)

Run equilibrium of feed to top and bottom streams given partition coeffiecients and return the phase fraction.

Parameters

- **feed** (Stream) – Mixed feed.
- **top** (Stream) – Top fluid.
- **bottom** (Stream) – Bottom fluid.
- **IDs** (*tuple[str]*) – IDs of chemicals in equilibrium.
- **K** (*1d array*) – Partition coefficeints corresponding to IDs.
- **phi** (*float, optional*) – Guess phase fraction in top phase.

Returns

phi – Phase fraction in top phase.

Return type

float

Notes

Chemicals not in equilibrium end up in the top phase.

Examples

```
>>> import numpy as np
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', tmo.Chemical('O2', phase='g')],␣
↪cache=True)
>>> IDs = ('Water', 'Ethanol')
>>> K = np.array([0.629, 1.59])
>>> feed = tmo.Stream('feed', Water=20, Ethanol=20, O2=0.1)
>>> top = tmo.Stream('top')
>>> bottom = tmo.Stream('bottom')
>>> tmo.separations.partition(feed, top, bottom, IDs, K)
0.5002512677600628
>>> top.show()
Stream: top
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water    7.73
                 Ethanol  12.3
                 O2       0.1
>>> bottom.show()
Stream: bottom
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water    12.3
                 Ethanol  7.72
```

thermosteam.separations.**lle**(*feed*, *top*, *bottom*, *top_chemical=None*, *efficiency=1.0*, *multi_stream=None*)

Run LLE mass and energy balance.

Parameters

- **feed** (Stream) – Mixed feed.
- **top** (Stream) – Top fluid.
- **bottom** (Stream) – Bottom fluid.
- **top_chemical** (*str, optional*) – Identifier of chemical that will be favored in the top fluid.
- **efficiency=1.** (*float,*) – Fraction of feed in liquid-liquid equilibrium. The rest of the feed is divided equally between phases
- **multi_stream** (MultiStream, *optional*) – Data from feed is passed to this stream to perform liquid-liquid equilibrium.

Examples

Perform liquid-liquid equilibrium around water and octanol and split the phases:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Octanol'], cache=True)
>>> feed = tmo.Stream('feed', Water=20, Octanol=20, Ethanol=1)
>>> top = tmo.Stream('top')
>>> bottom = tmo.Stream('bottom')
>>> tmo.separations.lle(feed, top, bottom)
>>> top.show()
Stream: top
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water    3.55
                 Ethanol  0.861
                 Octanol  20
>>> bottom.show()
Stream: bottom
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water    16.5
                 Ethanol  0.139
                 Octanol  0.00409
```

Assume that 1% of the feed is not in equilibrium (possibly due to poor mixing):

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol', 'Octanol'], cache=True)
>>> feed = tmo.Stream('feed', Water=20, Octanol=20, Ethanol=1)
>>> top = tmo.Stream('top')
>>> bottom = tmo.Stream('bottom')
>>> ms = tmo.MultiStream('ms', phases='lL') # Store flow rate data here as well
>>> tmo.separations.lle(feed, top, bottom, efficiency=0.99, multi_stream=ms)
>>> ms.show()
MultiStream: ms
 phases: ('L', 'l'), T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): (L) Water    3.55
                     Ethanol  0.861
                     Octanol  20
                 (l) Water    16.5
                     Ethanol  0.139
                     Octanol  0.00408
```

thermosteam.separations.**vle**(*feed*, *vap*, *liq*, *T=None*, *P=None*, *V=None*, *Q=None*, *x=None*, *y=None*, *multi_stream=None*)

Run VLE mass and energy balance.

Parameters

- **feed** (Stream) – Mixed feed.
- **vap** (Stream) – Vapor fluid.
- **liq** (Stream) – Liquid fluid.
- **P=None** (*float*) – Operating pressure [Pa].
- **Q=None** (*float*) – Duty [kJ/hr].
- **T=None** (*float*) – Operating temperature [K].
- **V=None** (*float*) – Molar vapor fraction.
- **x=None** (*float*) – Molar composition of liquid (for binary mixtures).
- **y=None** (*float*) – Molar composition of vapor (for binary mixtures).
- **multi_stream** (MultiStream, *optional*) – Data from feed is passed to this stream to perform vapor-liquid equilibrium.

Examples

Perform vapor-liquid equilibrium on water and ethanol and split phases to vapor and liquid streams:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> feed = tmo.Stream('feed', Water=20, Ethanol=20)
>>> vapor = tmo.Stream('top')
>>> liquid = tmo.Stream('bottom')
>>> tmo.separations.vle(feed, vapor, liquid, V=0.5, P=101325)
>>> vapor.show()
Stream: top
 phase: 'g', T: 353.88 K, P: 101325 Pa
 flow (kmol/hr): Water    7.72
                 Ethanol  12.3
>>> liquid.show()
Stream: bottom
 phase: 'l', T: 353.88 K, P: 101325 Pa
 flow (kmol/hr): Water    12.3
                 Ethanol  7.72
```

It is also possible to save flow rate data in a multi-stream as well:

```
>>> ms = tmo.MultiStream('ms', phases='lg')
>>> tmo.separations.vle(feed, vapor, liquid, V=0.5, P=101325, multi_stream=ms)
>>> ms.show()
MultiStream: ms
 phases: ('g', 'l'), T: 353.88 K, P: 101325 Pa
 flow (kmol/hr): (g) Water    7.72
                     Ethanol  12.3
```

(continues on next page)

(continued from previous page)

```
                (l) Water    12.3
                    Ethanol  7.72
```

thermosteam.separations.**material_balance**(*chemical_IDs*, *variable_inlets*, *constant_inlets=()*, *constant_outlets=()*, *is_exact=True*, *balance='flow'*)

Solve stream mass balance by iteration.

Parameters

- **chemical_IDs** (*tuple[str]*) – Chemicals that will be used to solve mass balance linear equations. The number of chemicals must be same as the number of input streams varied.
- **variable_inlets** (*Iterable[Stream]*) – Inlet streams that can vary in net flow rate to accomodate for the mass balance.
- **constant_inlets** (*Iterable[Stream], optional*) – Inlet streams that cannot vary in flow rates.
- **constant_outlets** (*Iterable[Stream], optional*) – Outlet streams that cannot vary in flow rates.
- **is_exact=True** (*bool, optional*) – True if exact flow rate solution is required for the specified IDs.
- **balance='flow'** (*{'flow', 'composition'}, optional*) –
  - 'flow': Satisfy output flow rates
  - 'composition': Satisfy net output molar composition

Examples

Vary inlet flow rates to satisfy outlet flow rates:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> in_a = tmo.Stream('in_a', Water=1)
>>> in_b = tmo.Stream('in_b', Ethanol=1)
>>> variable_inlets = [in_a, in_b]
>>> in_c = tmo.Stream('in_c', Water=100)
>>> constant_inlets = [in_c]
>>> out_a = tmo.Stream('out_a', Water=200, Ethanol=2)
>>> out_b = tmo.Stream('out_b', Ethanol=100)
>>> constant_outlets = [out_a, out_b]
>>> chemical_IDs = ('Water', 'Ethanol')
>>> tmo.separations.material_balance(chemical_IDs, variable_inlets, constant_inlets,
↪ constant_outlets)
>>> tmo.Stream.sum([in_a, in_b, in_c]).mol - tmo.Stream.sum([out_a, out_b]).mol #␣
↪Molar flow rates entering and leaving are equal
array([0., 0.])
```

Vary inlet flow rates to satisfy outlet composition:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> in_a = tmo.Stream('in_a', Water=1)
```

(continues on next page)

(continued from previous page)

```
>>> in_b = tmo.Stream('in_b', Ethanol=1)
>>> variable_inlets = [in_a, in_b]
>>> in_c = tmo.Stream('in_c', Water=100)
>>> constant_inlets = [in_c]
>>> out_a = tmo.Stream('out_a', Water=200, Ethanol=2)
>>> out_b = tmo.Stream('out_b', Ethanol=100)
>>> constant_outlets = [out_a, out_b]
>>> chemical_IDs = ('Water', 'Ethanol')
>>> tmo.separations.material_balance(chemical_IDs, variable_inlets, constant_inlets,
 ↪ constant_outlets, balance='composition')
>>> tmo.Stream.sum([in_a, in_b, in_c]).z_mol - tmo.Stream.sum([out_a, out_b]).z_mol
 ↪# Molar composition entering and leaving are equal
array([0., 0.])
```

class thermosteam.separations.**MultiStageLLE**(*N_stages*, *feed*, *solvent*, *carrier_chemical=None*, *thermo=None*, *partition_data=None*)

Create a MultiStageLLE object that models a counter-current system of mixer-settlers for liquid-liquid extraction.

Parameters

- **N_stages** (*int*) – Number of stages.
- **feed** (Stream) – Feed with solute.
- **solvent** (Stream) – Solvent to contact feed and recover solute.
- **carrier_chemical** (*str*) – Name of main chemical in the feed (which is not selectively extracted by the solvent).
- **partition_data** (*{'IDs': tuple[str], 'K': 1d array}, optional*) – IDs of chemicals in equilibrium and partition coefficients (molar composition ratio of the raffinate over the extract). If given, The mixer-settlers will be modeled with these constants. Otherwise, partition coefficients are computed based on temperature and composition.

Examples

Simulate 2-stage extraction of methanol from water using octanol:

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Methanol', 'Octanol'], cache=True)
>>> N_stages = 2
>>> feed = tmo.Stream('feed', Water=500, Methanol=50)
>>> solvent = tmo.Stream('solvent', Octanol=500)
>>> stages = tmo.separations.MultiStageLLE(N_stages, feed, solvent)
>>> stages.simulate_multi_stage_lle_without_side_draws()
>>> stages.raffinate.show()
Stream:
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water     413
                 Methanol  8.4
                 Octanol   0.1
>>> stages.extract.show()
Stream:
 phase: 'L', T: 298.15 K, P: 101325 Pa
```

(continues on next page)

(continued from previous page)

```
 flow (kmol/hr): Water     87.
                 Methanol  41.
                 Octanol   500
```

Simulate 10-stage extraction with user defined partition coefficients:

```
>>> import numpy as np
>>> tmo.settings.set_thermo(['Water', 'Methanol', 'Octanol'])
>>> N_stages = 10
>>> feed = tmo.Stream('feed', Water=5000, Methanol=500)
>>> solvent = tmo.Stream('solvent', Octanol=5000)
>>> stages = tmo.separations.MultiStageLLE(N_stages, feed, solvent,
...     partition_data={
...         'K': np.array([6.894, 0.7244, 3.381e-04]),
...         'IDs': ('Water', 'Methanol', 'Octanol'),
...         'phi': 0.4100271108219455 # Initial phase fraction guess. This is␣
↪optional.
...     }
... )
>>> stages.simulate_multi_stage_lle_without_side_draws()
>>> stages.raffinate.show()
Stream:
 phase: 'l', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water     4.1e+03
                 Methanol  1.2
                 Octanol   1.5
>>> stages.extract.show()
Stream:
 phase: 'L', T: 298.15 K, P: 101325 Pa
 flow (kmol/hr): Water     871
                 Methanol  499
                 Octanol   5e+03
```

thermosteam.separations.**single_component_flow_rates_for_multi_stage_lle_without_side_draws**(*N_stages*, *phase_ratios*, *partition_coefficients*, *feed*, *solvent*)

Solve flow rates for a single component across a multi stage liquid-liquid extraction operation without side draws.

Parameters

- **N_stages** (*int*) – Number of stages.
- **phase_ratios** (*1d array*) – Phase ratios by stage. The phase ratio for a given stage is defined as F_l / F_L; where F_l and F_L are the flow rates of phase l (raffinate) and L (extract) leaving the stage respectively.
- **partition_coefficients** (*1d array*) – Partition coefficients by stage. The partition coefficient for a given stage is defined as x_l / x_L; where x_l and x_L are the fraction of the component in phase l (raffinate) and L (extract) leaving the stage.

- **feed** (*float*) – Component flow rate in feed entering stage 1.
- **solvent** (*float*) – Component flow rate in solvent entering stage N.

Returns
extract_flow_rates – Extract component flow rates by stage.

Return type
1d array

thermosteam.separations.**flow_rates_for_multi_stage_extration_without_side_draws**(*N_stages*, *phase_fractions*, *partition_coefficients*, *feed*, *solvent*)

Solve flow rates for a single component across a multi stage liquid-liquid extraction without side draws.

Parameters

- **N_stages** (*int*) – Number of stages.
- **phase_fractions** (*1d array*) – Phase fractions by stage. The phase fraction for a given stage is defined as F_l / (F_l + F_L); where F_l and F_L are the flow rates of phase l (raffinate) and L (extract) leaving the stage respectively.
- **partition_coefficients** (*Iterable[1d array]*) – Partition coefficients with components by row and stages by column. The partition coefficient for a component in a given stage is defined as x_l / x_L; where x_l and x_L are the fraction of the component in phase l (raffinate) and L (extract) leaving the stage.
- **feed** (*Iterable[float]*) – Flow rates of all components in feed entering stage 1.
- **solvent** (*Iterable[float]*) – Flow rates of all components in solvent entering stage N.

Returns
extract_flow_rates – Extract flow rates with stages by row and components by column.

Return type
2d array

thermosteam.separations.**chemical_splits**(*a*, *b=None*, *mixed=None*)

Return a ChemicalIndexer with splits for all chemicals to stream *a*.

Examples

```
>>> import thermosteam as tmo
>>> tmo.settings.set_thermo(['Water', 'Ethanol'], cache=True)
>>> stream = tmo.Stream('stream', Water=10, Ethanol=10)
>>> stream.vle(V=0.5, P=101325)
>>> isplits = tmo.separations.chemical_splits(stream['g'], stream['l'])
>>> isplits.show()
ChemicalIndexer:
 Water    0.3861
 Ethanol  0.6139
>>> isplits = tmo.separations.chemical_splits(stream['g'], mixed=stream)
>>> isplits.show()
```

(continues on next page)

(continued from previous page)

```
ChemicalIndexer:
 Water    0.3861
 Ethanol  0.6139
```

1.18 Thermosteam 0.22

1.18.1 0.22.5

- Activity coefficient calculations for chemicals with missing functional groups models are default to 1 and a RuntimeWarning is issued. Previously, an RuntimeError was raised instead.

1.18.2 0.22.4

Bug fixes:

- Corrected bug in the gas enthalpy of chemicals, closing enthalpy balances.

1.18.3 0.22.3

Bug fixes:

- *thermosteam.reaction.Reaction* math operations (including divisions) are now working correctly. Please see examples in documentation to learn how to add, multiply, and divide reactions.

1.18.4 0.22.2

New features:

- *thermosteam.reaction.Reaction*, *thermosteam.reaction.ParallelReaction*, and *thermosteam.reaction.SeriesReaction* can now handle chemical reactions with phases changed. For example, Reaction('H2O,l -> H2,g + O2,g', reactant='H2O') will be able to react a MultiStream and add products (i.e. H2, O2) to the gas phase.

1.18.5 0.22.1

New features:

- *thermosteam.Stream.S* now returns absolute entropy instead of relative entropy.
- *thermosteam.reaction.Reaction.correct_atomic_balance()* can now balance equations where the number of atoms balanced is greater than the number varied chemical coefficients.
- *thermosteam.reaction.Reaction* objects can now be imported directly from the thermosteam module (e.g. "from thermosteam import Reaction").

1.19 Contributing to Thermosteam

1.19.1 General Process

Here's the short summary of how to contribute using git bash:

1. If you are a first-time contributor:
    - Go to https://github.com/BioSTEAMDevelopmentGroup/thermosteam and click the "fork" button to create your own copy of the project.
    - Clone the project to your local computer:

```
git clone --depth 100 https://github.com/your-username/thermosteam.git
```

    - Change the directory:

```
cd thermosteam
```

    - Add the upstream repository:

```
git remote add upstream https://github.com/BioSTEAMDevelopmentGroup/
↪thermosteam.git
```

    - Now, git remote -v will show two remote repositories named "upstream" (which refers to the thermosteam repository), and "origin" (which refers to your personal fork).
2. Develop your contribution:
    - Pull the latest changes from upstream:

```
git checkout master
git pull upstream master
```

    - Create a branch for the feature you want to work on. Since the branch name will appear in the merge message, use a sensible name such as "Chemical-properties-enhancement":

```
git checkout -b Chemical-properties-enhancement
```

    - Commit locally as you progress (git add and git commit) Use a properly formatted commit message, write tests that fail before your change and pass afterward, run all the tests locally. Be sure to document any changed behavior in docstrings, keeping to the NumPy docstring standard.
3. To submit your contribution:
    - Push your changes back to your fork on GitHub:

```
git push origin Chemical-properties-enhancement
```

    - Enter your GitHub username and password (repeat contributors or advanced users can remove this step by connecting to GitHub with SSH).
    - Go to GitHub. The new branch will show up with a green Pull Request button. Make sure the title and message are clear, concise, and self- explanatory. Then click the button to submit it.
    - If your commit introduces a new feature or changes functionality, post in https://github.com/BioSTEAMDevelopmentGroup/thermosteam/issues to explain your changes. For bug fixes, documentation updates, etc., this is generally not necessary, though if you do not get any reaction, do feel free to ask for a review.

1.19.2 Testing

First install the developer version of thermosteam:

```
$ cd thermosteam
$ pip install -e .[dev]
```

This installs pytest and other dependencies you need to run the tests locally. You can run tests by going to your local thermosteam directory and running the following:

```
$ pytest
 ========================== test session starts ===========================
 platform win32 -- Python 3.7.6, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
 rootdir: C:\...\thermosteam, configfile: pytest.ini
 plugins: hypothesis-5.5.4, arraydiff-0.3, astropy-header-0.1.2, cov-2.10.1,
 doctestplus-0.5.0, openfiles-0.4.0, remotedata-0.3.2
 collected 157 items

 tests\test_biorefineries.py .......                                    [  4%]
 tests\test_chemical.py .                                               [  5%]
 tests\test_stream.py ..                                                [  6%]
 thermosteam\_chemical.py .......                                       [ 10%]
 thermosteam\_chemicals.py ......................                       [ 24%]
 thermosteam\_multi_stream.py ............                              [ 31%]
 thermosteam\_stream.py ....................................            [ 56%]
 thermosteam\_thermo.py ...                                             [ 57%]
 thermosteam\_thermo_data.py .                                          [ 58%]
 thermosteam\eos.py ...................                                 [ 70%]
 thermosteam\functional.py .....                                        [ 73%]
 thermosteam\separations.py ............                                [ 82%]
 thermosteam\units_of_measure.py ....                                   [ 84%]
 thermosteam\base\functor.py .                                          [ 85%]
 thermosteam\chemicals\reaction.py ..                                   [ 86%]
 thermosteam\equilibrium\bubble_point.py ...                            [ 88%]
 thermosteam\equilibrium\dew_point.py ...                               [ 90%]
 thermosteam\equilibrium\lle.py .                                       [ 91%]
 thermosteam\equilibrium\sle.py .                                       [ 91%]
 thermosteam\equilibrium\vle.py .                                       [ 92%]
 thermosteam\mixture\ideal_mixture_model.py .                           [ 92%]
 thermosteam\mixture\mixture_builders.py .                              [ 93%]
 thermosteam\reaction\_reaction.py .....                                [ 96%]
 thermosteam\utils\representation.py .....                              [100%]

 ========================== 157 passed in 36.52s ==========================
```

This runs all the doctests in thermosteam, which covers most of the API. If any test is marked with a letter F, that test has failed. Pytest will point you to the location of the error, the values that were expected, and the values that were generated.

Note: Some parts in thermosteam do not have tests yet. Any contributions towards rigorous testing is welcome!

1.19.3 Documentation

Concise and thorough documentation is required for any contribution. Make sure to:

- Use NumPy style docstrings.
- Document all functions and classes.
- Document short functions in one line if possible.
- Mention and reference any equations or methods used and make sure to include the chapter and page number if it is a book or a long document.
- Preview the docs before making a pull request (open your cmd/terminal in the "docs" folder, run "make html", and open "docs/_build/html/index.html").

1.19.4 Best practices

Please refer to the following guides for best practices to make software designs more understandable, flexible, and maintainable:

- PEP 8 style guide.
- PEP 257 docstring guide.
- Zen of Python philosophy.
- SOLID programing principles.

CHAPTER

TWO

RELATED PROJECTS

There are many third party open-source Python libraries that may provide additional resources for designing and modeling chemical processes:

- chemicals: Chemical properties component of Chemical Engineering Design Library.
- fluids: Fluid dynamics component of Chemical Engineering Design Library (ChEDL)
- chempy: A package useful for chemistry written in Python.
- thermochem: Useful Python modules for Thermodynamics and Thermochemistry.
- chemics: A Python package for chemical reactor engineering.
- ase: The Atomic Simulation Environment.
- pMuTT: The Python Multiscale Thermochemistry Toolbox.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X

Z