Contents

1 Installation 3
2 Usage 5
3 Documentation 7
4 Contributing 45
5 Package index and modules 47
Python Module Index 49
Index 51
The Chemics package is a collection of Python functions for performing calculations in the field of chemical and fluidization engineering. Source code for the package is available on GitHub and contributions from the community are encouraged.
If you don’t have Python installed on your computer, the Anaconda or Miniconda distribution of Python is recommended for scientific computing. After setting up Python, the Chemics package can be downloaded and installed using the pip or conda package managers.

Install Chemics using the pip package manager:

$ pip install chemics

Install Chemics using the conda package manager:

$ conda config --add channels conda-forge
$ conda install chemics
Usage

The example below imports the Chemics package then uses the `{rhog()}` function to calculate the density of a gas based on its molecular weight, pressure, and temperature.

```
import chemics as cm
cm.rhog(28, 170100, 773)
```

The `{ut()}` function calculates the terminal velocity of a particle according to the Haider and Levenspiel 1989 paper as shown below.

```
import chemics as cm

# Parameters
dp = 0.00016  # particle diameter [m]
mu = 1.8e-5   # gas viscosity [kg/(m s)]
phi = 0.67    # particle sphericity [-]
rhog = 1.2    # gas density [kg/m^3]
rhos = 2600   # particle density [kg/m^3]

# Haider and Levenspiel terminal velocity [m/s]
ut_haider = cm.ut_haider(dp, mu, phi, rhog, rhos)
```

Use the `{ChemicalEquation}()` class to get properties of the reactants and products from a given chemical equation.

```
import chemics as cm

ce = cm.ChemicalEquation('2 HCl + 2 Na -> 2 NaCl + H2')

ce.balance  # returns True for balanced equation

ce.rct_properties  # returns a dataframe of the reactant properties

# HCl   Na
```

(continues on next page)
More examples are available in the chemics-examples repository.
Details about the functions available in the Chemics package are provided below. Equations and associated references used to develop the functions are also given.

### 3.1 Atomic elements

Dictionary containing atomic number, name, and atomic weight of elements in the periodic table. Conventional atomic weight is used for atomic weight where applicable. Values taken from IUPAC.

**References**


```python
atomic_elements = {
    'H': {'atomic_number': 1, 'name': 'hydrogen', 'atomic_weight': 1.008},
    'He': {'atomic_number': 2, 'name': 'helium', 'atomic_weight': 4.0026},
    'Li': {'atomic_number': 3, 'name': 'lithium', 'atomic_weight': 6.94},
    'Be': {'atomic_number': 4, 'name': 'beryllium', 'atomic_weight': 9.0122},
    'B': {'atomic_number': 5, 'name': 'boron', 'atomic_weight': 10.81},
    'C': {'atomic_number': 6, 'name': 'carbon', 'atomic_weight': 12.011},
    'N': {'atomic_number': 7, 'name': 'nitrogen', 'atomic_weight': 14.007},
    'O': {'atomic_number': 8, 'name': 'oxygen', 'atomic_weight': 15.999},
    'F': {'atomic_number': 9, 'name': 'fluorine', 'atomic_weight': 18.998},
    'Ne': {'atomic_number': 10, 'name': 'neon', 'atomic_weight': 20.180},
    'Na': {'atomic_number': 11, 'name': 'sodium', 'atomic_weight': 22.990},
    'Mg': {'atomic_number': 12, 'name': 'magnesium', 'atomic_weight': 24.305},
    'Al': {'atomic_number': 13, 'name': 'aluminium', 'atomic_weight': 26.982},
    'Si': {'atomic_number': 14, 'name': 'silicon', 'atomic_weight': 28.085},
    'P': {'atomic_number': 15, 'name': 'phosphorus', 'atomic_weight': 30.974},
    'S': {'atomic_number': 16, 'name': 'sulfur', 'atomic_weight': 32.06},
    'Cl': {'atomic_number': 17, 'name': 'chlorine', 'atomic_weight': 35.45},
}
```
<table>
<thead>
<tr>
<th>Element</th>
<th>Atomic Number</th>
<th>Name</th>
<th>Atomic Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ar</td>
<td>18</td>
<td>argon</td>
<td>39.948</td>
</tr>
<tr>
<td>K</td>
<td>19</td>
<td>potassium</td>
<td>39.098</td>
</tr>
<tr>
<td>Ca</td>
<td>20</td>
<td>calcium</td>
<td>40.078</td>
</tr>
<tr>
<td>Sc</td>
<td>21</td>
<td>scandium</td>
<td>44.956</td>
</tr>
<tr>
<td>Ti</td>
<td>22</td>
<td>titanium</td>
<td>47.867</td>
</tr>
<tr>
<td>V</td>
<td>23</td>
<td>vanadium</td>
<td>50.942</td>
</tr>
<tr>
<td>Cr</td>
<td>24</td>
<td>chromium</td>
<td>51.996</td>
</tr>
<tr>
<td>Mn</td>
<td>25</td>
<td>manganese</td>
<td>54.938</td>
</tr>
<tr>
<td>Fe</td>
<td>26</td>
<td>iron</td>
<td>55.845</td>
</tr>
<tr>
<td>Co</td>
<td>27</td>
<td>cobalt</td>
<td>58.933</td>
</tr>
<tr>
<td>Ni</td>
<td>28</td>
<td>nickel</td>
<td>58.693</td>
</tr>
<tr>
<td>Cu</td>
<td>29</td>
<td>copper</td>
<td>63.546</td>
</tr>
<tr>
<td>Zn</td>
<td>30</td>
<td>zinc</td>
<td>65.380</td>
</tr>
<tr>
<td>Ga</td>
<td>31</td>
<td>gallium</td>
<td>69.723</td>
</tr>
<tr>
<td>Ge</td>
<td>32</td>
<td>germanium</td>
<td>72.630</td>
</tr>
<tr>
<td>As</td>
<td>33</td>
<td>arsenic</td>
<td>74.922</td>
</tr>
<tr>
<td>Se</td>
<td>34</td>
<td>selenium</td>
<td>78.960</td>
</tr>
<tr>
<td>Br</td>
<td>35</td>
<td>bromine</td>
<td>79.904</td>
</tr>
<tr>
<td>Kr</td>
<td>36</td>
<td>krypton</td>
<td>83.798</td>
</tr>
<tr>
<td>Rb</td>
<td>37</td>
<td>rubidium</td>
<td>85.468</td>
</tr>
<tr>
<td>Sr</td>
<td>38</td>
<td>strontium</td>
<td>87.620</td>
</tr>
<tr>
<td>Y</td>
<td>39</td>
<td>yttrium</td>
<td>88.906</td>
</tr>
<tr>
<td>Zr</td>
<td>40</td>
<td>zirconium</td>
<td>91.224</td>
</tr>
<tr>
<td>Nb</td>
<td>41</td>
<td>niobium</td>
<td>92.906</td>
</tr>
<tr>
<td>Mo</td>
<td>42</td>
<td>molybdenum</td>
<td>95.940</td>
</tr>
<tr>
<td>Tc</td>
<td>43</td>
<td>technetium</td>
<td>99.880</td>
</tr>
<tr>
<td>Ru</td>
<td>44</td>
<td>ruthenium</td>
<td>101.07</td>
</tr>
<tr>
<td>Rh</td>
<td>45</td>
<td>rhodium</td>
<td>102.91</td>
</tr>
<tr>
<td>Pd</td>
<td>46</td>
<td>palladium</td>
<td>106.42</td>
</tr>
<tr>
<td>Ag</td>
<td>47</td>
<td>silver</td>
<td>107.87</td>
</tr>
<tr>
<td>Cd</td>
<td>48</td>
<td>cadmium</td>
<td>112.41</td>
</tr>
<tr>
<td>In</td>
<td>49</td>
<td>indium</td>
<td>114.82</td>
</tr>
<tr>
<td>Sn</td>
<td>50</td>
<td>tin</td>
<td>118.71</td>
</tr>
<tr>
<td>Sb</td>
<td>51</td>
<td>antimony</td>
<td>121.76</td>
</tr>
<tr>
<td>Te</td>
<td>52</td>
<td>tellurium</td>
<td>127.60</td>
</tr>
<tr>
<td>I</td>
<td>53</td>
<td>iodine</td>
<td>126.90</td>
</tr>
<tr>
<td>Xe</td>
<td>54</td>
<td>xenon</td>
<td>131.29</td>
</tr>
<tr>
<td>Cs</td>
<td>55</td>
<td>caesium</td>
<td>132.91</td>
</tr>
<tr>
<td>Ba</td>
<td>56</td>
<td>barium</td>
<td>137.33</td>
</tr>
<tr>
<td>La</td>
<td>57</td>
<td>lanthanum</td>
<td>138.91</td>
</tr>
<tr>
<td>Ce</td>
<td>58</td>
<td>cerium</td>
<td>140.12</td>
</tr>
<tr>
<td>Pr</td>
<td>59</td>
<td>praseodymium</td>
<td>140.91</td>
</tr>
<tr>
<td>Nd</td>
<td>60</td>
<td>neodymium</td>
<td>144.24</td>
</tr>
<tr>
<td>Pm</td>
<td>61</td>
<td>promethium</td>
<td>None</td>
</tr>
<tr>
<td>Sm</td>
<td>62</td>
<td>samarium</td>
<td>150.36</td>
</tr>
<tr>
<td>Eu</td>
<td>63</td>
<td>europium</td>
<td>151.96</td>
</tr>
<tr>
<td>Gd</td>
<td>64</td>
<td>gadolinium</td>
<td>157.25</td>
</tr>
<tr>
<td>Tb</td>
<td>65</td>
<td>terbium</td>
<td>158.93</td>
</tr>
<tr>
<td>Dy</td>
<td>66</td>
<td>dysprosium</td>
<td>162.50</td>
</tr>
<tr>
<td>Ho</td>
<td>67</td>
<td>holmium</td>
<td>164.93</td>
</tr>
<tr>
<td>Er</td>
<td>68</td>
<td>erbium</td>
<td>167.26</td>
</tr>
<tr>
<td>Tm</td>
<td>69</td>
<td>thulium</td>
<td>168.93</td>
</tr>
<tr>
<td>Yb</td>
<td>70</td>
<td>ytterbium</td>
<td>173.05</td>
</tr>
<tr>
<td>Lu</td>
<td>71</td>
<td>lutetium</td>
<td>174.97</td>
</tr>
<tr>
<td>Hf</td>
<td>72</td>
<td>hafnium</td>
<td>178.49</td>
</tr>
<tr>
<td>Ta</td>
<td>73</td>
<td>tantalum</td>
<td>180.95</td>
</tr>
<tr>
<td>W</td>
<td>74</td>
<td>tungsten</td>
<td>183.84</td>
</tr>
</tbody>
</table>
3.2 Bed expansion factor

The bed expansion factor can be used to estimate the expanded bed height of a bubbling fluidized bed reactor. Two equations are given in the Souza-Santos book based on the diameter of the bed.

For \(d_D < 0.0635\) m:

\[
f_{be,exp} = 1 + \frac{1.032 (U - U_m)^{0.57}}{\rho_p^{0.166} \rho_{GA}^{0.083} \phi_D^{0.445}}
\]

3.2. Bed expansion factor
For $d_D \geq 0.0635$ m:

$$f_{bexp} = 1 + \frac{14.314(U - U_{mf})^{0.738} \rho_p^{0.376}}{\rho_{GA}^{0.1206} U_{mf}^{0.937}} d_p^{1.006} \rho_{p}^{0.376}$$

### 3.2.1 Nomenclature

- $d_D$ - Diameter of the bed (m)
- $d_p$ - Diameter of the bed particle (m)
- $f_{bexp}$ - Bed expansion factor (-)
- $\rho_{GA}$ - Gas density (kg/m$^3$)
- $\rho_p$ - Particle density (kg/m$^3$)
- $U$ - Gas superficial velocity (m/s)
- $U_{mf}$ - Minimum fluidization velocity (m/s)

### 3.2.2 Source code

```python
chemics.bed_expansion_factor.fbexp(db, dp, rhog, rhos, umf, us)
```

Bed expansion factor for calculating expanded bed height of a bubbling fluidized bed reactor. See equations 14.7 and 14.8 in Souza-Santos$^1$.

**Parameters**

- `db (float)` – Diameter of the bed [m]
- `dp (float)` – Diameter of the bed particle [m]
- `rhog (float)` – Density of gas [kg/m$^3$]
- `rhos (float)` – Density of bed particle [kg/m$^3$]
- `umf (float)` – Minimum fluidization velocity [m/s]
- `us (float)` – Superficial gas velocity [m/s]

**Returns**

- `fbx (float)` – Bed expansion factor [-]

**Example**

```python
>>> umf = 0.1157
... us = 3.0*umf
... fbexp(0.05232, 0.0004, 0.4413, 2500, 0.1157, us)
1.4864
```

### References

3.3 Biomass composition

Biomass composition can be represented in terms of cellulose, hemicellulose, lignins, and extractives. If experimental data is not available, the components of biomass can be estimated from the ultimate analysis using a characterization method developed by Debiagi, Pecchi, Gentile, Frassoldati, Cuoci, Faravelli, and Ranzi.

Use the `biocomp()` function to calculate biomass composition. Use the `plot_biocomp()` function to create a Matplotlib figure of the biomass composition results.

```python
chemics.biomass_composition.biocomp(yc, yh, yo=None, yh2o=0, yash=0, alpha=0.6, beta=0.8, gamma=0.8, delta=1, epsilon=1, printcomp=False)
```

Determine biomass composition from ultimate analysis mass fractions of C, H, and O. Composition returned as cellulose, hemicellulose, lignins, and extractives based on method discussed in the Debiagi 2015 paper. Parameters

- **`yc`** (`float`): Mass fraction of carbon in biomass, dry ash free basis [-]
- **`yh`** (`float`): Mass fraction of hydrogen in biomass, dry ash free basis [-]
- **`yo`** (`float`, optional): Mass fraction of oxygen in biomass, if not given then value is calculated as difference, dry ash free basis [-]. Default is None.
- **`yh2o`** (`float`, optional): Mass fraction of water in biomass, as received basis [-]. Default is 0.
- **`yash`** (`float`, optional): Mass fraction of ash in biomass, as received basis [-]. Default is 0.
- **`alpha`** (`float`, optional): Splitting parameter as molar ratio of cellulose and hemicellulose contained in reference mixture RM1 [-]. Default is 0.6.
- **`beta`** (`float`, optional): Splitting parameter as molar ratio of lignin LIG-O and lignin LIG-C contained in reference mixture RM2 [-]. Default is 0.8.
- **`gamma`** (`float`, optional): Splitting parameter as molar ratio of lignin LIG-H and lignin LIG-C contained in reference mixture RM3 [-]. Default is 0.8.
- **`delta`** (`float`, optional): Splitting parameter as molar ratio of lignins (LIG-H and LIG-C) and extractive TGL to define reference mixture RM2 [-]. Default is 1.0.
- **`epsilon`** (`float`, optional): Splitting parameter as molar ratio of lignins (LIG-O and LIG-C) and extractive TANN to define reference mixture RM3 [-]. Default is 1.0.
- **`printcomp`** (`bool`, optional): Print composition results if True. Default is False.

Returns

- **`comp`** (`dict`): Dictionary representing reference mixtures and biomass compositions on the basis of mole fractions (x) and mass fractions (y).
  - **`y_rm1`** mass fractions [C, H, O] of reference mixture RM1
  - **`y_rm2`** mass fractions [C, H, O] of reference mixture RM2
  - **`y_rm3`** mass fractions [C, H, O] of reference mixture RM3
  - **`x_daf`** mole fractions [cell, hemi, ligc, ligh, ligo, tann, tgl] of biomass as dry ash-free basis
  - **`x_wet`** mole fractions [cell, hemi, ligc, ligh, ligo, tann, tgl] of biomass as wet basis
  - **`y_daf`** mass fractions [cell, hemi, ligc, ligh, ligo, tann, tgl] of biomass as dry ash-free basis
  - **`y_wet`** mass fractions [cell, hemi, ligc, ligh, ligo, tann, tgl] of biomass as wet basis
  - **`y_wetash`** mass fractions [cell, hemi, ligc, ligh, ligo, tann, tgl] of biomass as wet ash basis

Raises: `ValueError` – When sum of mass fractions is not equal to one.

---

Examples

```python
>>> yc = 0.534
>>> yh = 0.06
>>> bc = biocomp(yc, yh)
>>> bc['y_daf']
array([0.2936, 0.1594, 0.0712, 0.2934, 0.1822, 0., 0.])

>>> yc = 0.500
>>> yh = 0.060
>>> yo = 0.440
>>> yash = 0.15

>>> biocomp(yc, yh, yo, yash=yash, printcomp=True)

<table>
<thead>
<tr>
<th>basis</th>
<th>cell</th>
<th>hemi</th>
<th>liggc</th>
<th>ligh</th>
<th>ligo</th>
<th>tann</th>
<th>tgl</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_daf</td>
<td>0.5016</td>
<td>0.3344</td>
<td>0.0328</td>
<td>0.0614</td>
<td>0.0698</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>x_wet</td>
<td>0.5016</td>
<td>0.3344</td>
<td>0.0328</td>
<td>0.0614</td>
<td>0.0698</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>y_daf</td>
<td>0.4275</td>
<td>0.2322</td>
<td>0.0445</td>
<td>0.1409</td>
<td>0.1549</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>y_wet</td>
<td>0.4275</td>
<td>0.2322</td>
<td>0.0445</td>
<td>0.1409</td>
<td>0.1549</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>y_wetash</td>
<td>0.3634</td>
<td>0.1974</td>
<td>0.0378</td>
<td>0.1197</td>
<td>0.1317</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
```

References

`chemics.biomass_composition.plot_biocomp(ax, yc, yh, y_rm1, y_rm2, y_rm3)`

Plot characterization of biomass sample and calculated reference mixtures as mass fractions, dry ash-free basis.

- **Parameters**
  - `ax (Axes)` – The Matplotlib axes from a figure
  - `yc (float)` – Mass fraction of carbon in biomass, dry ash free basis [-]
  - `yh (float)` – Mass fraction of hydrogen in biomass, dry ash free basis [-]
  - `y_rm1 (array)` – Mass fraction of reference mixture RM1 where `y_rm1 = [yC, yH, yO]`
  - `y_rm2 (array)` – Mass fraction of reference mixture RM2 where `y_rm2 = [yC, yH, yO]`
  - `y_rm3 (array)` – Mass fraction of reference mixture RM3 where `y_rm3 = [yC, yH, yO]`

- **Returns** `ax (Axes)` – The Matplotlib axes for the plot figure

3.4 Bubble rise velocity

The `ubr_kunii()` function is from the Kunii and Levenspiel book while the `ubr_holland()` function is from the Holland and Bragg book.

### 3.4.1 Kunii and Levenspiel

As stated in the Kunii and Levenspiel book, the bubble rise velocity in a dense bed can be calculated as

\[
ubr = 0.711 \left( g b d \right)^{1/2} \quad \text{where} \quad db/dt < 0.125
\]

Wall effects can slow the rise of bubbles when `db/dt > 0.125`; as such, the rise velocity is calculated as

\[
ubr = \left[ 0.711 \left( g b d \right)^{1/2} \right] \times 1.2 \exp \left( -1.49 \frac{db}{dt} \right) \quad \text{where} \quad 0.125 < db/dt < 0.6
\]
For conditions where \( \frac{d_b}{d_t} > 0.6 \), the fluidized bed is considered to be slugging.

**Nomenclature**

- \( d_b \) - Diameter of sphere having same volume as spherical cap bubble, effective bubble diameter (m)
- \( d_t \) - Bed or tube diameter (m)
- \( g \) - Acceleration due to gravity, 9.81 m/s\(^2\)
- \( u_{br} \) - Rise velocity of single bubbles in a fluidized bed (m/s)

## 3.4.2 Holland and Bragg

In order to predict the performance of two-phase flow in vertical tubes and fluidised beds, it is first necessary to study the rate of rise of single bubbles.

Peebles and Garber (1953) developed a number of models based on empirical data and theoretical equations (Stoke’s Law) to fit four flow regimes of bubbles. Later Willis (1969) added a fifth region to describe terminal rise velocity from potential flow theory. A summary of the theory can be found in Holland and Bragg (1995) p234.

**Region 1**

Bubbles behave as buoyant solid spheres, rising vertically.

\[
Re_b \leq 2
\]

\[
u_b = \frac{2R_b^2(\rho_l - \rho_g)g}{9\mu_l}
\]

**Region 2**

Bubbles raise as spheres, but the drag coefficient is slightly less than that of a solid of the same volume.

\[
2 < Re_b \leq 4.02G_1^{-0.214}
\]

\[
u_b = 0.33g^{0.76} \left( \frac{\rho_l}{\mu_l} \right)^{0.52} R_b^{1.28}
\]

**Region 3**

Bubbles are flattened and rise in zig-zag pattern.

\[
4.02G_1^{-0.214} < Re_b \leq 3.10G_1^{-0.25}
\]

\[
u_b = 1.35 \left( \frac{\sigma}{\rho_l R_b} \right)^{0.5}
\]

**Region 4**

Bubbles rise vertically adopting a mushroom-cap shape.

\[
3.10G_1^{-0.25} < Re_b \leq 2.3 \sqrt{\frac{\sigma g}{\rho_l}}
\]

\[
u_b = 1.18 \left( \frac{\sigma g}{\rho_l} \right)^{0.25}
\]

**Region 5**
Large spherical-cap bubbles.

\[ Re_b > 2.3 \sqrt{\frac{\sigma}{g \rho_l}} \]

\[ u_b = \sqrt{g R_b} \]

**Dimensionless Groups:**

\[ Re_b = \frac{2 \rho_l u_b R_b}{\mu_l} \]

\[ G_1 = \frac{g \mu_l^4}{\rho_l \sigma^3} \]

**Nomenclature**

- \( u_b \): The rise velocity (m/s)
- \( R_b \): The radius of a sphere having the same volume as the bubble (m)
- \( D_e \): The equivalent diameter where \( D_e = 2R_b \) (m)
- \( \sigma \): The surface tension (dynes/cm)
- \( \rho_l/g \): The density of the liquid or gas respectively (kg/m\(^3\))
- \( \mu_l \): The viscosity of the liquid (centiPoise)
- \( g \): Gravitational constant = 9.81 (m/s\(^2\))
- \( Re_b \): Reynolds Number of the bubble (dimensionless)
- \( G_1 \): Morton Number (dimensionless)

**3.4.3 References**


**3.4.4 Source code**

```python
chemics.bubble_rise_velocity.ubr_holland(db, rho_l, rho_g, sig, mu_l)
```

Bubble rise velocity in a liquid for a range of applicability. This velocity is related to the terminal rise velocity correlations for a single bubble in a liquid as given in Table 7.1 in the Holland and Bragg book\(^1\).

**Parameters**

- \( db \) (float) – Diameter of sphere having same volume as spherical cap bubble, referred to as the effective bubble diameter [m]
- \( rho_l \) (float) – Density of the liquid [kg/m\(^3\)]
- \( rho_g \) (float) – Density of the gas [kg/m\(^3\)]
- \( sig \) (float) – Surface tension [dynes/cm]

• \texttt{mu\_l} (\texttt{float}) – Dynamic viscosity of the liquid [centiPoise]

Returns \texttt{ub} (\texttt{float}) – Bubble (terminal) rise velocity [m/s]

Example

```python
>>> ubr_holland(0.00003, 998, 0, 72.75, 1.21)
0.0004046
```

References

\texttt{chemics.bubble\_rise\_velocity.ubr\_kunii} (\texttt{db, dt})
Rise velocity of single bubbles in a fluidized bed from Equations 5.3 and 5.4 in Kunii and Levenspiel book\(^2\).

Parameters

• \texttt{db} (\texttt{float}) – Diameter of sphere having same volume as spherical cap bubble, referred to as the effective bubble diameter [m]
• \texttt{dt} (\texttt{float}) – Bed or tube diameter [m]

Returns \texttt{ubr} (\texttt{float}) – Rise velocity of single bubbles in fluidized bed [m/s]

Example

```python
>>> ubr_kunii(0.05, 0.6)
0.4979
```

References

3.5 Chemical equation

Use the \texttt{ChemicalEquation} class to determine properties of the reactants and products in a given chemical equation.

\texttt{class} \texttt{chemics.chemical\_equation.ChemicalEquation} (\texttt{eq, names=None})
Properties of the reactants and products in a given chemical equation.

Parameters

• \texttt{eq} (\texttt{str}) – Chemical equation such as A + B -> C + D
• \texttt{names} (\texttt{dict, optional}) – Names of unique species and their corresponding chemical formula.

Variables

• \texttt{eq} (\texttt{str}) – Chemical equation such as A + B -> C + D
• \texttt{names} (\texttt{dict, optional}) – Names of unique species and their corresponding chemical formula.
• \texttt{rct\_properties} (\texttt{dataframe}) – Number of moles, chemical species, molecular weight, mass, mole fraction, and mass fraction for each reactant.

• **rct_elements** *(dict)* – Total number of atomic elements on reactant side of the equation.
  • **rct_moles** *(float)* – Total number of moles on reactant side of the equation.
  • **rct_mass** *(float)* – Total mass of the reactants.

• **prod_properties** *(dataframe)* – Number of moles, chemical species, molecular weight, mass, mole fraction, and mass fraction for each product.

• **prod_elements** *(dict)* – Total number of atomic elements on product side of the equation.
  • **prod_moles** *(float)* – Total number of moles on product side of the equation.
  • **prod_mass** *(float)* – Total mass of the products.

• **balance** *(bool)* – Balance between atomic elements in reactants and products.

**Examples**

```python
>>> ce = ChemicalEquation('2 HCl + 2 Na -> 2 NaCl + H2')
>>> ce.balance
True

>>> ce.rct_properties
       HCl   Na
    moles  2     2
species HCl   Na
molwt  36.458 22.99
mass   72.916 45.98
mofrac  0.5  0.5
massfrac 0.613275 0.386725

>>> ce.rct_properties.loc['massfrac']
HCl  0.613275
Na  0.386725

>>> ce.rct_elements
{'H': 2.0, 'Cl': 2.0, 'Na': 2.0}

>>> ce.rct_moles
4.0

>>> ce.rct_mass
118.896
```

### 3.6 Choking velocity

Choking velocity is when the air flow is insufficient to support the flow of solid particles up a vertical pipe.
3.6.1 Source code

chemics.choking_velocity.uch_bifan(ar, dp, gs, rhog)
Choking velocity from Bi and Fan\(^1\), also see Zhang 2015 paper.

Parameters

• \( \text{ar} \) (float) – Archimedes number [-]
• \( \text{dp} \) (float) – Diameter of particle [m]
• \( \text{gs} \) (float) – Solids flux [kg/(s m^2)]
• \( \text{rhog} \) (float) – Density of gas [kg/m^3]

Returns \( \text{uch} \) (float) – Choking velocity [m/s]

References

chemics.choking_velocity.uch_leung(us, ut)
Choking velocity based on Equation 6 from Leung\(^2\). All parameters must be in units of ft/s or in units of m/s.

Parameters

• \( \text{us} \) (float) – Solids velocity [ft/s or m/s]
• \( \text{ut} \) (float) – Terminal velocity of a single particle [ft/s or m/s]

Returns \( \text{uch} \) (float) – Choking velocity [ft/s or m/s]

References

chemics.choking_velocity.uch_matsen(gs, rhos, ut)
Choking velocity from the Matsen paper\(^3\).

Parameters

• \( \text{gs} \) (float) – Solids flux [kg/(s m^2)]
• \( \text{rhos} \) (float) – Density of particle [kg/m^3]
• \( \text{ut} \) (float) – Terminal velocity [m/s]

Returns \( \text{uch} \) (float) – Choking velocity [m/s]

References

chemics.choking_velocity.uch_psri(x, dp, d_pipe, gs, rhog, rhos, ut)
Choking velocity from PSRI 2016 notebook. Uses MKS units for input parameters but returns Uch in ft/s. Function requires SciPy fsolve to solve for Uch.

Parameters

• \( \text{x} \) (float) – Solve for Uch with SciPy fsolve
• \( \text{dp} \) (float) – Particle size [m]

---


• **d_pipe** *(float)* – Pipe diameter [m]
• **gs** *(float)* – Solids mass flux [kg/(s m^2)]
• **rhog** *(float)* – Density of gas [kg/m^3]
• **rhos** *(float)* – Density of particle [kg/m^3]
• **ut** *(float)* – Terminal velocity [m/s]

**Returns** **uch** *(float)* – Choking velocity [ft/s]

**References**

Notebook from PSRI 2016 workshop, pages J-20 and J-72.

Chemics.choking_velocity.uch_punwani(xy, d_pipe, gs, rhog, rhos, ut)

Provide functions to solve for voidage and choking velocity based on equations from Punwani 1976. Requires SciPy fsolve function.

**Parameters**

• **xy** – Solves for ep and uch using SciPy fsolve where ep = voidage [-] and uch = choking velocity [m/s]
• **d_pipe** *(float)* – Internal pipe diameter [m]
• **gs** *(float)* – Solids flux [kg/(s m^2)]
• **rhog** *(float)* – Density of gas, must be in units of lb/ft^3
• **rhos** *(float)* – Density of particle [kg/m^3]
• **ut** *(float)* – Terminal velocity [m/s]

**Returns** **f1, f2** *(functions)* – Functions to solve for ep and uch

**References**


Chemics.choking_velocity.uch_yang(xy, d_pipe, gs, rhos, ut)

Provide functions to solve for voidage and choking velocity based on equations from Yang 1975. Requires SciPy fsolve function.

**Parameters**

• **xy** – Solve for ep and uch using SciPy fsolve where ep = voidage [-] and uch = choking velocity [m/s]
• **d_pipe** *(float)* – Internal pipe diameter [m]
• **gs** *(float)* – Solids flux [kg/(s m^2)]
• **rhos** *(float)* – Density of particle [kg/m^3]
• **ut** *(float)* – Terminal velocity [m/s]

**Returns** **f1, f2** *(functions)* – Functions to solve for ep and uch
References


chemics.choking_velocity.uch_yousfi(dp, gs, rhog, mu, ut)
Choking velocity from Yousfi 1974, also see Zhang 2015 paper.

Parameters
• dp (float) – Particle diameter [ft or m]
• gs (float) – Solids flux [lb/(hr ft^2) or kg/(s m^2)]
• rhog (float) – Gas density [lb/ft^3 or kg/m^3]
• mu (float) – Gas viscosity [lb/(ft s) or kg/(s m)]
• ut (float) – Terminal velocity of particle [ft/s or m/s]

Returns uch (float) – Choking velocity [ft/s or m/s]

References


chemics.choking_velocity.uch_zhang(ar, dp, gs, rhog)
Choking velocity from Zhang 2015.

Parameters
• ar (float) – Archimedes number [-]
• dp (float) – Diameter of particle [m]
• gs (float) – Solids flux [kg/(s m^2)]
• rhog (float) – Density of gas [kg/m^3]

Returns
• uch (float) – Choking velocity [m/s]

Reference
• ———

3.7 Conversions

Helper functions for unit conversion.

chemics.conversions.massfrac_to_molefrac(y, mw)
Convert from mass fractions to mole fractions. Calculation assumes a total mass of 100 g.

\[ m_i = y_i \times 100 \]
\[ x_i = \frac{m_i}{\sum m_i} \]
where $m$ is mass [g], $y$ is mass fraction [-], $x$ is mole fraction [-], and $MW$ is molecular weight [g/mol] of each component.

**Parameters**
- $y$ (list, tuple or array) – Mass fraction of each component [-]
- $mw$ (list, tuple or array) – Molecular weight of each component [g/mol]

**Returns**
- $x$ (array) – Mole fractions of each component [-]

**Example**
```python
given:
y = [0.36, 0.16, 0.20, 0.28]
... mw = [12.011, 1.008, 15.999, 14.007]
... massfrac_to_molefrac(y, mw)
yield:
[0.136 0.718 0.057 0.09]
```

**chemics.conversions.molefrac_to_massfrac($x$, $mw$)**

Convert from mole fractions to mass fractions. Calculation assumes total moles is 100. 

$$n_i = x_i \times 100$$
$$y_i = \frac{n_i \times MW_i}{\sum n_i \times MW_i}$$

where $n$ is moles [mol], $x$ is mole fraction [-], $y$ is mass fraction [-], and $MW$ is molecular weight [g/mol] of each component.

**Parameters**
- $x$ (list, tuple, or array) – Mole fraction of each component [-]
- $mw$ (list, tuple, or array) – Molecular weight of each component [g/mol]

**Returns**
- $y$ (array) – Mass fraction of each component [-]

**Example**
```python
given:
x = [0.36, 0.16, 0.20, 0.28]
>> mw = [12.011, 1.008, 15.999, 14.007]
... molefrac_to_massfrac(x, mw)
yield:
[0.373 0.014 0.276 0.338]
```

**chemics.conversions.slm_to_lpm($slm$, $pgas$, $tgas$)**

Convert volumetric gas flow from standard liters per minute (SLM or SLPM) to liters per minute (LPM) where STP defined as 273.25 K and 101,325 Pa.

$$1LPM = 1SLPM \times \frac{T_{gas}}{273.15K} \times \frac{14.696 psi}{P_{gas}}$$

**Parameters**
- $slm$ (float) – Volumetric gas flow in standard liters per minute [SLM]
- $pgas$ (float) – Absolute gas pressure [kPa]
- $tgas$ (float) – Gas temperature [K]

**Returns**
- $lpm$ (float) – Volumetric gas flow in liters per minute [LPM]
3.8 Devolatilization time

A correlation to estimate the conversion time of cylindrical beech wood particles in a sand bed fluidized by nitrogen gas was developed by Colomba Di Blasi and Carmen Branca. The correlation is an empirical power-law relation corresponding to a conversion of 95% (also referred to as a devolatilization time of 95%). It was developed from experiments conducted at fast pyrolysis conditions where bed temperatures varied from 712-1107 K and particle diameters ranged from 2-10 mm. According to the Di Blasi and Carmen article, similar correlations exist for coal particles.

\[ t_v = 0.8 e^{1525/T_r} d^{1.2} \]

3.8.1 Nomenclature

- \( t_v \) - conversion time, devolatilization time of 95% (s)
- \( T_r \) - Temperature of fluidized bed (K)
- \( d \) - Diameter of wood particle (mm)

3.8.2 Source code

```python
chemics.devol_time.devol_time(dp, tbed)
```

Correlation for 95% devolatilization time for wood particles. Refer to Equation 2 in the Di Blasi\(^1\) article which is based on beech wood cylinders in a fluidized sand bed.

Parameters
- \( dp \) (float or array) – Diameter of biomass particle [mm]
- \( tbed \) (float or array) – Bed temperature [K]

Returns \( tv \) (float or array) – Devolatilization time for 95% conversion [s]

Example

```python
>>> devol_time(2.0, 773.15)
13.2114
```

3.9 Dimensionless numbers

Functions are provided for the following dimensionless numbers:

- Archimedes number (Ar)
- Biot number (Bi)
- Prandtl number (Pr)
- Pyrolysis number I (Py I)
- Pyrolysis number II (Py II)
- Reynolds number (Re)

3.9.1 Source code

```python
chemics.dimensionless_numbers.archimedes(dp, rhog, rhos, mu)
Calculate the dimensionless Archimedes number.

\[ \text{Ar} = \frac{dp^3 \rho \rho_s (\rho_s - \rho_g) g}{\mu^2} \]

Parameters

- `dp (float)` – Particle diameter [m]
- `rhog (float)` – Gas density [kg/m^3]
- `rhos (float)` – Solid density [kg/m^3]
- `mu (float)` – Dynamic viscosity [kg/(ms)]

Returns `ar (float)` – Archimedes number [-]

Example

```python
>>> archimedes(0.001, 910, 2500, 0.001307)
8309.1452
```

References

• \( k \) (\textit{float}) – Thermal conductivity [W/(mK)]

Returns \( b_i \) (\textit{float}) – Biot number [-]

**Example**

```python
>>> biot(4.63, 0.001, 3.84)
0.0012057
```

**References**


```python
chemics.dimensionless_numbers.prandtl(cp=None, mu=None, k=None, nu=None, alpha=None)
```

Calculate the dimensionless Prandtl number for a fluid or gas.

\[
Pr = \frac{cp\mu}{k} = \frac{\nu}{\alpha}
\]

Parameters

• \( cp \) (\textit{float}) – Specific heat [J/(kgK)]
• \( mu \) (\textit{float}) – Dynamic viscosity [kg/(ms)]
• \( k \) (\textit{float}) – Thermal conductivity [W/(mK)]
• \( nu \) (\textit{float}, optional) – Kinematic viscosity [m²/s]
• \( alpha \) (\textit{float}, optional) – Thermal diffusivity [m²/s]

Returns \( pr \) (\textit{float}) – Prandtl number [-]

**Examples**

```python
>>> prandtl(cp=4188, mu=0.001307, k=0.5674)
9.647
```

```python
>>> prandtl(nu=1.5064e-5, alpha=2.1002e-5)
0.71726
```

**Raises** \texttt{ValueError} – Must provide (cp, mu, k) or (nu, alpha)

**References**


```python
chemics.dimensionless_numbers.pyroI(k, kr, rho, cp, r)
```

Calculate the pyrolysis number Py I for a biomass particle.

\[
Py^I = \frac{k}{\rho \, C_p \, R^2 \, K}
\]

Parameters

• \( k \) (\textit{float}) – Thermal conductivity of the biomass particle [W/mK]
• **kr** (*float*) – Rate constant [1/s]
• **rho** (*float*) – Density of the biomass particle [kg/m$^3$]
• **cp** (*float*) – Heat capacity of the biomass particle [J/kgK]
• **r** (*float*) – Radius or characteristic length of the biomass particle [m]

**Returns** **pyI** (*float*) – Pyrolysis number Py I [-]

**Example**

```python
>>> pyroI(k=0.12, kr=1.38556, rho=540, cp=3092.871049, r=0.0001847)
1.5198
```

**References**


---

**chemics.dimensionless_numbers**.**pyroII** (*h, kr, rho, cp, r*)

Calculate the pyrolysis number Py II for a biomass particle.

\[ Py^{II} = \frac{h}{\rho C_p R K} \]

**Parameters**

• **h** (*float*) – Convective heat transfer coefficient [W/m$^2$K]
• **kr** (*float*) – Rate constant [1/s]
• **rho** (*float*) – Density of the biomass particle [kg/m$^3$]
• **cp** (*float*) – Heat capacity of the biomass particle [J/kgK]
• **r** (*float*) – Radius or characteristic length of the biomass particle [m]

**Returns** **pyII** (*float*) – Pyrolysis number Py II [-]

**Example**

```python
>>> pyroII(h=862.6129, kr=1.38556, rho=540, cp=3092.871049, r=0.0001847)
2.018038
```

**References**


---

**chemics.dimensionless_numbers**.**reynolds** (*u, d, rho=None, mu=None, nu=None*)

Calculate the dimensionless Reynolds number for a fluid or gas flow.

\[ Re = \frac{\rho u d}{\mu} = \frac{u d}{\nu} \]

**Parameters**
• \( u (\text{float}) \) – Flow speed [m/s]
• \( d (\text{float}) \) – Characteristic length or dimension [m]
• \( \rho (\text{float}, \text{optional}) \) – Density of the fluid or gas [kg/m³]
• \( \mu (\text{float}, \text{optional}) \) – Dynamic viscosity of the fluid or gas [kg/(ms)]
• \( \nu (\text{float}, \text{optional}) \) – Kinematic viscosity of the fluid or gas [m²/s]

Returns \( \text{re} (\text{float}) \) – Reynolds number [-]

Examples

```python
>>> reynolds(2.6, 0.025, rho=910, mu=0.38)
155.65789
>>> reynolds(0.25, 0.102, nu=1.4e-6)
18214.2857
```

Raises \text{ValueError} – Must provide \((u, d, \rho, \mu)\) or \((u, d, \nu)\)

References


### 3.10 Gas density

The density of a gas can be calculated using its molecular weight, pressure, and temperature as shown below:

\[
\rho_{\text{gas}} = \frac{P \cdot MW}{R \cdot T}
\]

#### 3.10.1 Nomenclature

- \( \rho_{\text{gas}} \) - Density of gas (kg/m³)
- \( P \) - Pressure of the gas (Pa)
- \( MW \) - Molecular weight of the gas (g/mol)
- \( R \) - Gas constant as 8.3145 [(m³ Pa)/(K mol)]
- \( T \) - Temperature of the gas (K)

#### 3.10.2 Source code

```python
chemics.gas_density.rhog(mw, p, t)
```

Calculate gas density from molecular weight, pressure, and temperature.

Parameters

- \( mw (\text{float}) \) – Molecular weight of gas [g/mol]
- \( p (\text{float}) \) – Pressure of the gas [Pa]
- \texttt{tk} (\texttt{float}) – Temperature of the gas [K]

\textbf{Returns} \texttt{rho} (\texttt{float}) – Density of gas [kg/m$^3$]

\textbf{Examples}

```
>>> rhog(28, 101325, 773)
0.4414
```

### 3.11 Gas thermal conductivity

Thermal conductivity of gas as a function of temperature can be calculated from the following equation:

\[k_{\text{gas}} = A + B T + C T^2 + D T^3\]

where coefficients A, B, C, and D are obtained from tables in the Yaws’ Handbook for inorganic and organic compounds. The \texttt{k\_gas\_inorganic} function in the Chemics package supports 259 gas species while the \texttt{k\_gas\_organic} function supports 6,914 species. At a minimum, the functions require formula name and temperature as input parameters; however, the CAS number is also needed as an input if more than one form of a species exists.

#### 3.11.1 Nomenclature

- \(k_{\text{gas}}\) - Thermal conductivity of gas [W/(m K)]
- \(A, B, C, D\) - Coefficients from Yaws’ Handbook (-)
- \(T\) - Temperature (K)

#### 3.11.2 Source code

```python
chemics.gas_thermal_conductivity.k_gas_inorganic(formula, temp, full=False)
```

Thermal conductivity of gas as a function of temperature. Applicable to gas comprised of inorganic compounds. Results based on coefficients from Yaws’ Critical Property Data for Chemical Engineers and Chemists$^1$.

\textbf{Parameters}

- \texttt{formula (string)} – Molecular formula of the gas.
- \texttt{temp (float)} – Temperature of the gas [K]
- \texttt{full (bool, optional)} – When set to \texttt{False} (default) just thermal conductivity is returned. When set to \texttt{True} then return thermal conductivity and other information.

\textbf{Returns}

- \texttt{k\_gas (float)} – Thermal conductivity of gas [W/(m K)]
- \texttt{k\_gas, cas, tmin, tmax, a, b, c, d (tuple)} – Additional values are only returned when keyword \texttt{full=True}.

\begin{flushright}
\footnotesize$^1$ Carl L. Yaws. Table 84. Thermal Conductivity of Gas – Inorganic Compounds in Yaws’ Critical Property Data for Chemical Engineers and Chemists. Published by Knovel, 2014.
\end{flushright}
cas - CAS number [-]
tmin, tmax - Temperature range at which results are applicable [K]
a, b, c, d - Values for regression coefficients [-]

**Raises**
- `ValueError` – If gas formula is not found in csv data file.
- `ValueError` – If gas temperature is not in range between tmin and tmax.

**Examples**

```python
>>> k_gas_inorganic('N2', 773)
0.0535
```

```python
>>> k_gas_inorganic('N2', 773, full=True)
(0.0535, '7727-37-9', 63.15, 1500.0, -0.0002267, 0.0001027, -6.0151e-08, 2.2331e-11)
```

**References**

chemics.gas_thermal_conductivity.

`k_gas_organic(formula, temp, cas=None, full=False)`

Thermal conductivity of gas as a function of temperature. Applicable to gas comprised of organic compounds. Results based on coefficients from Yaws’ Critical Property Data for Chemical Engineers and Chemists\(^2\).

**Parameters**

- `formula (string)` – Molecular formula of the gas.
- `temp (float)` – Temperature of the gas [K]
- `cas (string)` – CAS number of the gas, required for some species [-]
- `full (bool, optional)` – When set to `False` (default) just thermal conductivity is returned. When set to `True` then return thermal conductivity and other information.

**Returns**

- `k_gas (float)` – Thermal conductivity of gas [W/(m K)]
- `k_gas, cas, tmin, tmax, a, b, c, d (tuple)` – Additional values are only returned when keyword `full=True`.

```python
k_gas - Thermal conductivity of gas [W/(m K)]
cas - CAS number [-]
tmin, tmax - Temperature range at which results are applicable [K]
a, b, c, d - Values for regression coefficients [-]
```

** Raises**

- `ValueError` – If gas formula is not available in CSV data file.

---

• ValueError – If multiple substances have same formula. Require CAS number.
• ValueError – If gas temperature is not in range between Tmin and Tmax.

Examples

```python
>>> k_gas_organic('CO', 801)
0.05722

>>> k_gas_organic('C18H38O', 920, cas='593-32-8')
0.04174
```

References

3.12 Gas viscosity

Gas viscosity as a function of temperature can be calculated from the following equation:

\[ \mu_{gas} = A + BT + CT^2 + DT^3 \]

where coefficients A, B, C, and D are obtained from tables in the Yaws’ Handbook for inorganic and organic compounds. The Chemics function supports 372 inorganic gas species and 7,031 organic gas species. At a minimum, the function requires the formula name and temperature as input parameters; however, the CAS number may be needed as an input if more than one form of a species exists.

Viscosity of a gas mixture can be estimated as a weighted mean using the mixture function. Note that for a gas mixture, the sum of the components must equal 1.

3.12.1 Nomenclature

\[ \mu_{gas} \] - Viscosity of gas (\( \mu P \))  
\[ A, B, C, D \] - Coefficients from Yaws’ Handbook ( - )  
\[ T \] - Temperature (K)

3.12.2 Source code

`chemics.gas_viscosity.mu_gas(formula, temp, cas=None, full=False)`  
Viscosity of gas as a function of temperature. Results calculated from coefficients in Yaws’ Critical Property Data for Chemical Engineers and Chemists\(^1\). CAS (Chemical Abstracts Service) number may be required for some species.

**Parameters**

- `formula` (str) – Molecular formula of the gas.
- `temp` (float) – Temperature of the gas [K]
- `cas` (str, optional) – CAS number of the gas, required for some species [-]

---

\(^1\) Carl L. Yaws. Viscosity Gas Tables 80 and 81 in Yaws’ Critical Property Data for Chemical Engineers and Chemists. Published by Knovel, 2014.
• **full** *(bool, optional)* – When set to `False` (default) just gas viscosity is returned. When set to `True` then return gas viscosity and other information.

**Returns**

• **mu_gas** *(float)* – Viscosity of gas [micropoise]

• **mu_gas, cas, tmin, tmax, a, b, c, d** *(tuple)* – Additional values are only returned when keyword `full=True`.

```
mu_gas - Viscosity of gas [micropoise]
cas - CAS number [-]
tmin, tmax - Temperature range at which results are applicable [K]
a, b, c, d - Values for regression coefficients [-]
```

**Raises**

• `ValueError` – If gas formula is not found in csv data file.

• `ValueError` – If gas temperature is not in range between tmin and tmax.

**Examples**

```
>>> mu_gas('CH4', 810)
234.21

>>> mu_gas('C2Cl2F4', 900, cas='374-07-2')
314.90

>>> mu_gas('H2', 404)
113.18

>>> mu_gas('N2', 773)
363.82

>>> mu_gas('N2', 773, full=True)
(363.82, '7727-37-9', 63.15, 1970.0, 4.46, 0.63, -0.00026, 5.41e-08)
```

**References**

`chemics.gas_viscosity.mu_graham(mus, xs)`

Calculate viscosity of a gas mixture using Graham’s method\(^2\). Formula presented here is based on Equation 1 from the Davidson report\(^3\).

\[
\mu_{mix} = \sum (x_i \cdot \mu_i)
\]

where \(\mu_{mix}\) is viscosity of the gas mixture, \(x_i\) is mole fraction [-] of each component, and \(\mu_i\) is gas viscosity of each component.


3.12. Gas viscosity
Parameters

- **mus** *(list, tuple, or array)* – Viscosity of each gas component.
- **xs** *(list, tuple, or array)* – Mole fraction of each gas component [-]

**Returns** **mu** *(float)* – Gas viscosity of the mixture. Units are same as input viscosity.

**Raises** **ValueError** – If sum of mole fractions does not equal 1.0

**Example**

```python
>>> mu_h2 = cm.mu_gas('H2', 773.15)
... mu_n2 = cm.mu_gas('N2', 773.15)
... mu_graham([mu_h2, mu_n2], [0.85, 0.15])
207.37
```

**References**

chemics.gas_viscosity.mu_herning *(mus, mws, xs)*

Calculate viscosity of a gas mixture using the approach by Herning and Zipperer\(^4\). Formula presented here is based on Equation 1 from the Davidson report\(^5\).

\[
\mu_{\text{mix}} = \frac{\sum (\mu_i \cdot x_i \cdot \sqrt{MW_i})}{\sum (x_i \cdot \sqrt{MW_i})}
\]

where \(\mu_{\text{mix}}\) is viscosity of the gas mixture, \(x_i\) is mole fraction [-] of each component, \(\mu_i\) is gas viscosity of each component, and \(MW_i\) is the molecular weight [g/mol] of each component.

**Parameters**

- **mus** *(list, tuple, or array)* – Viscosity of each gas component.
- **mws** *(list, tuple, or array)* – Molecular weight of each gas component [g/mol]
- **xs** *(list, tuple, or array)* – Mole fraction of each gas component [-]

**Returns** **mu** *(float)* – Gas viscosity of the mixture. Units are same as input viscosity.

**Raises** **ValueError** – If sum of mole fractions does not equal 1.0

**Example**

```python
>>> mu_h2 = cm.mu_gas('H2', 773.15)
... mu_n2 = cm.mu_gas('N2', 773.15)
... mw_h2 = cm.mw('H2')
... mw_n2 = cm.mw('N2')
... mu_herning([mu_h2, mu_n2], [mw_h2, mw_n2], [0.85, 0.15])
252.81
```


3.13 Geldart particle classification

The behavior of solids fluidized by gases can be classified into four groups using the density difference and mean particle diameter. This particle classification was introduced by D. Geldart in 1973.

3.13.1 Source code

```python
def geldart_chart(dp, rhog, rhos, dpmin=None, dpmax=None):
    # Create a Matplotlib figure of the Geldart chart⁠. Data points for drawing the demarcation lines and regions were
digitized from Figure 2 in⁠. 

    Parameters
    
    - dp (float) – Sauter mean particle diameter [µm]
    - rhog (float) – Gas density [g/cm³]
    - rhos (float) – Particle density [g/cm³]
    - dpmin (float, optional) – Minimum particle size from particle size distribution [µm]
    - dpmax (float, optional) – Max particle size from particle size distribution [µm]

    Returns  Matplotlib figure.

    Note:  Particle diameter must be in microns (µm) and density in g/cm³.
```

Examples

For a single particle size

```python
>>> geldart_figure(300, 0.1, 2.5)
```

For a particle size with min and max sizes

```python
>>> geldart_figure(300, 0.1, 2.5, 100, 500)
```

References

3.14 Minimum fluidization velocity

For a bed of particles, the minimum fluidization velocity is the gas velocity at which the drag force of the upward moving gas equals the weight of the particles. As discussed in Chapter 3 of the Kunii and Levenspiel book, the

---

minimum fluidization velocity \( (u_{mf}) \) can be calculated from the equation shown below. This formula is based on the Ergun pressure drop equation for a bed of particles.

\[
\frac{1.75}{\epsilon_{mf}^{3/2}} \left( \frac{d_p u_{mf} \rho_g}{\mu} \right)^2 + \frac{150(1 - \epsilon_{mf})}{\epsilon_{mf}^{3/2} \phi^2} \left( \frac{d_p u_{mf} \rho_g}{\mu} \right) = \frac{d_p \rho_g (\rho_s - \rho_g) g}{\mu^2}
\]

The above equation can be written in terms of the Reynolds and Archimedes numbers as follows

\[
\frac{1.75}{\epsilon_{mf}^{3/2}} Re_{p,mf}^2 + \frac{150(1 - \epsilon_{mf})}{\epsilon_{mf}^{3/2} \phi^2} Re_{p,mf} = Ar
\]

where

\[
Ar = \frac{d_p \rho_g (\rho_s - \rho_g) g}{\mu^2}
\]

\[
Re_{p,mf} = \frac{d_p u_{mf} \rho_g}{\mu}
\]

Kunii and Levenspiel further simplify the equation to the following form

\[
K_1 Re_{p,mf}^2 + K_2 Re_{p,mf} = Ar
\]

where

\[
K_1 = \frac{1.75}{\epsilon_{mf}^{3/2}}
\]

\[
K_2 = \frac{150(1 - \epsilon_{mf})}{\epsilon_{mf}^{3/2} \phi^2}
\]

Solving for the Reynolds number provides

\[
Re_{p,mf} = (a^2 + bAr)^{1/2} - a
\]

where

\[
a = \frac{K_2}{2K_1}
\]

\[
b = \frac{1}{K_1}
\]

Finally, the minimum fluidization velocity can be calculated from the above Reynolds number as

\[
u_{mf} = \frac{Re_{p,mf} \mu}{d_p \rho_g}
\]

For very small particles where \( Re < 20 \), the above equation can be simplified to

\[
u_{mf} = \frac{d_p^2 (\rho_s - \rho_g) g}{150 \mu} \frac{\epsilon_{mf}^{3/2} \phi^2}{1 - \epsilon_{mf}}
\]

and for large particles where \( Re > 1000 \), the following equation can be used

\[
u_{mf}^2 = \frac{d_p^2 (\rho_s - \rho_g) g}{175 \rho_g} \frac{\epsilon_{mf}^{3/2}}{\phi}
\]

When void fraction and sphericity are not known, values for \( a \) and \( b \) from Table 4 in Chapter 3 of Kunii and Levenspiel can be used to estimate \( u_{mf} \).
### 3.14.1 Nomenclature

- \( a, b \) - dimensionless constants (-)
- \( Ar \) - Archimedes number (-)
- \( d_p \) - Particle diameter (m)
- \( \epsilon_{mf} \) - Bed void fraction at minimum fluidizing conditions (-)
- \( g \) - Acceleration due to gravity, 9.81 m/s²
- \( K_1, K_2 \) - dimensionless constants (-)
- \( \mu \) - Gas viscosity (kg/(m s))
- \( \phi \) - Sphericity of a particle (-)
- \( \rho_g \) - Gas density (kg/m³)
- \( \rho_s \) - Solid particle density (kg/m³)

### 3.14.2 Source code

```python
chemics.minimum_fluidization_velocity.umf_coeff(dp, mu, rhog, rhos, coeff='wenyu')
```

Determine minimum fluidization velocity using experimental coefficients from Wen and Yu, Richardson, Saxena and Vogel, Babu, Grace, and Chitester. This approach can be used when bed void fraction and particle sphericity are not known. Refer to Equation 25 and Table 4 in Chapter 3 of Kunii and Levenspiel¹.

#### Parameters
- \( dp \) (float) – Diameter of bed particle [m]
- \( mu \) (float) – Viscosity of gas [kg/(m s)]
- \( rhog \) (float) – Density of gas [kg/m³]
- \( rhos \) (float) – Density of bed particle [kg/m³]
- \( coeff \) (string) – Keyword to determine which coefficients to use for umf calculation. Valid options are ‘wenyu’, ‘rich’, ‘sax’, ‘babu’, ‘grace’, and ‘chit’. Default coefficients are set to ‘wenyu’.

#### Returns
- \( umf \) (float) – Minimum fluidization velocity [m/s]

#### Example

```python
>>> umf_coeff(0.0005, 3.6e-5, 0.44, 2500, 'rich')
0.1192
```

### References

- chemics.minimum_fluidization_velocity.umf_ergun(dp, ep, mu, phi, rhog, rhos)
  Determine minimum fluidization velocity from particle and gas properties. This approach is based on the Ergun pressure drop equation for a bed of particles. Refer to Equations 18 and 19 in Chapter 3 of Kunii and Levenspiel²,

#### Parameters
- \( dp \) (float) – Diameter of bed particle [m]
- \( ep \) (float) – Void fraction of the bed [-]

---
• \( \mu (\text{float}) \) – Viscosity of gas [kg/ms]
• \( \phi (\text{float}) \) – Sphericity of bed particle [-]
• \( \rho_{\text{g}} (\text{float}) \) – Density of gas [kg/m\(^3\)]
• \( \rho_{\text{s}} (\text{float}) \) – Density of bed particle [kg/m\(^3\)]

Returns \( \text{umf (float)} \) – Minimum fluidization velocity [m/s]

Example

```python
>>> umf_ergun(0.0005, 0.46, 3.6e-5, 0.86, 0.44, 2500)
0.1488
```

References

chemics.minimum_fluidization_velocity.\textbf{umf_reynolds}(dp, ep, \mu, \phi, re, \rho_{\text{g}}, \rho_{\text{s}})

Calculate minimum fluidization velocity for very small particles where Reynolds number < 20 and for very large particles where Reynolds number > 1000. See Equations 21 and 22 in Chapter 3 of Kunii and Levenspiel\(^3\).

Parameters

• \( dp (\text{float}) \) – Diameter of bed particle [m]
• \( ep (\text{float}) \) – Void fraction [-]
• \( \mu (\text{float}) \) – Viscosity of gas [kg/ms]
• \( \phi (\text{float}) \) – Sphericity of bed particle [-]
• \( re (\text{float}) \) – Reynolds number where Re < 20 or Re > 1000 [-]
• \( \rho_{\text{g}} (\text{float}) \) – Density of gas [kg/m\(^3\)]
• \( \rho_{\text{s}} (\text{float}) \) – Density of bed particle [kg/m\(^3\)]

Returns \( \text{umf (float)} \) – Minimum fluidization velocity [m/s]

Example

For small Reynolds number where \( Re = 19 \)

```python
>>> umf_reynolds(0.0005, 0.46, 3.6e-5, 0.86, 19, 0.44, 2500)
0.1513
```

For large Reynolds number where \( Re = 1001 \)

```python
>>> umf_reynolds(0.0005, 0.46, 3.6e-5, 0.86, 1001, 0.44, 2500)
1.1545
```

References

3.15 Molecular weight

Functions to calculate molecular weight of an element, compound, or gas mixture. Note that mole fractions must sum to one otherwise an error is raised.

chemics.molecular_weight.mw(formula)

Tokenize a molecular formula to determine total molecular weight. Calculation is based on atomic weight values from IUPAC\(^1\).

Parameters

- **formula** (str) – Molecular formula or element.

Returns

- **mw** (float) – Molecular weight of the formula or element [g/mol]

Examples

```python
>>> mw('C')
12.011

>>> mw('CH4')
16.04

>>> mw('(NH4)2SO4')
132.13
```

References

chemics.molecular_weight.mw_mix(mws, xs)

Molecular weight of a gas mixture calculated as a weighted mean.

Parameters

- **mws** (list, tuple, or array) – Molecular weight of each gas component [g/mol]
- **xs** (list, tuple, or array) – Mole fraction of each gas component [-]

Returns

- **mw_mix** (float) – Molecular weight of a gas mixture [g/mol]

Raises

- ValueError – If sum of mole fractions does not equal 1.0

Examples

```python
>>> mw_h2 = cm.mw('H2')
... mw_n2 = cm.mw('N2')
... mw_mix([mw_h2, mw_n2], [0.8, 0.2])
7.2156
```

---

3.16 Proximate analysis bases

Proximate analysis provides the percentage composition of a material in terms of fixed carbon (FC), volatile matter (VM), ash, and moisture. It can be represented on an as-received basis (% ar), dry basis (% dry), and dry-ash free basis (% daf).

3.16.1 Source code

`chemics.proximate_bases.proximate_bases(fc, vm, ash, moisture, disp=False)`

Convert proximate analysis from as-received basis (% ar) to dry basis (% dry) and dry-ash free basis (% daf).

Parameters

- `fc (float)` – Percent fixed carbon
- `vm (float)` – Percent volatile matter
- `ash (float)` – Percent ash
- `moisture (float)` – Percent moisture
- `disp (bool, optional)` – Print results to console, default is `False`

Returns `bases (dict)` – Proximate analysis bases calculated from given as-received basis. Bases are represented by keys in the dictionary where ‘ar’ is as-received basis with values of `[FC, VM, ash, moisture]`, ‘dry’ is dry basis with values of `[FC, VM, ash]`, and ‘daf’ is dry ash-free basis with values of `[FC, VM]`.

Raises `ValueError` – If the proximate analysis sum is not 100.

Example

```python
>>> proximate_bases(16.92, 76.40, 0.64, 6.04)
{  
    'ar': [16.92, 76.4, 0.64, 6.04],
    'dry': [18.00, 81.31, 0.68],
    'daf': [18.13, 81.86]
}
```

3.17 Terminal velocity

An individual particle can be carried by a stream of gas when the gas velocity exceeds the terminal velocity $u_t$ of the particle. However, in fluidized bed reactors, entrainment of particles out of the bed may require a gas velocity many times higher than the terminal velocity.
chemics终端速度.ut (cd, dp, rhog, rhos)

Calculate terminal velocity of a single particle based on Equation 28 on page 80 in the Kunii and Levenspiel book\(^1\) where \(C_D\) is an experimentally determined drag coefficient.

\[
u_t = \left( \frac{4d_p(\rho_s - \rho_g)g}{3\rho_g C_D} \right)^{1/2}
\]

**Parameters**

- **cd** (float) – Drag coefficient [-]
- **dp** (float) – Diameter of particle [m]
- **rhog** (float) – Density of gas [kg/m\(^3\)]
- **rhos** (float) – Density of solid [kg/m\(^3\)]

**Returns** \(ut\) (float) – Terminal velocity [m/s]

**Example**

```python
>>> ut(11.6867, 0.00016, 1.2, 2600)
0.6227
```

**References**

chemics终端速度.ut_ganser (dp, mu, phi, rhog, rhos)

Estimate terminal velocity of a non-spherical particle based on the Ganser drag coefficient\(^2\). According to the Chhabra paper\(^3\), the Ganser drag correlation is applicable for sphericity values from 0.09 to 1.

\[
C_d = \frac{24}{Re K_1} \left( 1 + 0.1118(Re K_1 K_2)^{0.6567} \right) + \frac{0.4305 K_2}{1 + \frac{4.905}{Re K_1 K_2}}
\]

\[
K_1 = \left( \frac{1}{3} + \frac{2}{3} \phi \right)
\]

\[
K_2 = 10^{1.8148(-\log \phi)^{0.5743}}
\]

where \(K_1\) is Stokes’ shape factor and \(K_2\) is Newton’s shape factor. The Cui 2007 and Chhabra 1999 papers leave out the \(-2.25 \ast d_v/D\) term in the shape factor equations.

**Parameters**

- **dp** (float) – Diameter of the particle [m]
- **mu** (float) – Viscosity of gas [kg/(m s)]
- **phi** (float) – Sphericity of the particle [-]
- **rhog** (float) – Density of the gas [kg/m\(^3\)]
- **rhos** (float) – Density of the particle [kg/m\(^3\)]

**Returns** \(ut\) (float) – Terminal velocity of non-spherical particle [m/s]

---

Example

```python
>>> ut_ganser(0.00016, 1.8e-5, 0.67, 1.2, 2600)
0.6230
```

Note: Drag coefficient is referenced from Equation 18 in Ganser, Equation 6 in Chhabra, and as Equation 2 in Cui\(^4\).

References

chemics.terminal_velocity.ut_haider(dp, mu, phi, rhog, rhos)

Calculate terminal velocity of a particle as discussed in the Haider and Levenspiel\(^5\).

To determine the terminal velocity for a range of particle sphericities, Haider and Levenspiel first define two dimensionless quantities

\[
d_* = d_p \left[ \frac{g \rho_g (\rho_s - \rho_g)}{\mu^2} \right]^{1/3} 
\]

\[
u_* = \left[ \frac{18}{d_*^{2/3}} + \frac{2.3348 - 1.7439 \phi}{d_*^{0.5}} \right]^{-1}
\]

where \(0.5 \leq \phi \leq 1\) and particle diameter \(d_p\) is an equivalent spherical diameter, the diameter of a sphere having the same volume as the particle. The relationship between \(u_*\) and \(u_t\) is given by

\[
u_* = u_t \left[ \frac{\rho_g}{g \mu (\rho_s - \rho_g)} \right]^{1/3}
\]

The terminal velocity of the particle can finally be determined by rearranging the above equation such that

\[
u_t = u_* \left[ \frac{g \mu (\rho_s - \rho_g)}{\rho_g^2} \right]^{1/3}
\]

Parameters

- dp (float) – Diameter of particle [m]
- mu (float) – Viscosity of gas [kg/(m s)]
- phi (float) – Sphericity of particle [-]
- rhog (float) – Density of gas [kg/m\(^3\)]
- rhos (float) – Density of particle [kg/m\(^3\)]

Returns ut (float) – Terminal velocity of a particle [m/s]

Example

```python
>>> ut_haider(0.00016, 1.8e-5, 0.67, 1.2, 2600)
0.8857
```


3.18 Transport disengaging height

chemics.transport_disengaging_height.tdh_chan(ug)
Calculate transport disengaging height (TDH) based on the Chan and Knowlton correlation\(^1\). This function is based on the equation presented in the Cahyadi\(^2\) review article

\[
TDH = 0.85 U_g^{1.2} (7.33 - 1.2 \log U_g)
\]

where \( U_g \) is superficial gas velocity. According to Table 1 in the Cahyadi review paper, this correlation is relevant for Geldart A and B sand particles with a density of 2595 kg/m\(^3\), mean particle size of 37-420 \( \mu \)m, and an inner diameter fluidizing column of 0.3 m.

**Parameters**
- \( u_g (float) \) – Superficial gas velocity [m/s]

**Returns**
- \( tdh (float) \) – Transport disengaging height [m]

**Example**

```python
>>> tdh_chan(0.3)
1.7587
```

**References**

3.19 Transport velocity

In circulating fluidized bed reactors, particle velocities should exceed the transport velocity to ensure circulation of solids in the system. According to Zhang et al., the transport velocity can be calculated from the following equation

\[ U_{TR} = \left( \frac{\mu}{\rho_g d_p} \right) (3.23 + 0.23 Ar) \]

where the Archimedes number is defined as

\[ Ar = \frac{d^3 \rho_s (\rho_s - \rho_g) g}{\mu^2} \]

3.19.1 Nomenclature

- \( Ar \) - Archimedes number (-)
- \( d_p \) - Particle diameter (m)
- \( g \) - Acceleration due to gravity, 9.81 m/s\(^2\)
- \( \mu \) - Gas viscosity (kg/(m s))
- \( \rho_g \) - Gas density (kg/m\(^3\))
- \( \rho_s \) - Solid particle density (kg/m\(^3\))

3.19.2 Source code

```python
chemics.transport_velocity.utr(dp, mu, rhog, rhos)
```

Determine the transport velocity of particles in a circulating fluidized bed riser. Based on Equation 2 in article by Zhang et al.\(^1\).

**Parameters**

- `dp` (float) – Diameter of particle [m]
- `mu` (float) – Viscosity of gas [kg/(m s)]
- `rhog` (float) – Density of gas [kg/m\(^3\)]
- `rhos` (float) – Density of solid particle [kg/m\(^3\)]

**Returns**

- `utr` (float) – Transport velocity [m/s]

**Example**

```python
>>> utr(0.0005, 3.6e-5, 0.44, 1630)
26.0617
```

---

3.20 Ultimate analysis bases

Ultimate analysis provides the percentage composition of a material in terms of carbon (C), hydrogen (H), oxygen (O), nitrogen (N), sulfur (S), ash, and moisture. It can be represented on an as-received basis (% ar), dry basis (% dry), and dry-ash free basis (% daf).

3.20.1 Source code

chemics.ultimate_bases.ultimate_bases(c, h, o, n, s, ash, moisture, disp=False)

Convert ultimate analysis from as-received basis (% ar) to dry basis (% dry) and dry ash-free (% daf) basis.

Parameters

- c (float) – Percent carbon
- h (float) – Percent hydrogen
- o (float) – Percent oxygen
- n (float) – Percent nitrogen
- s (float) – Percent sulfur
- ash (float) – Percent ash
- moisture (float) – Percent moisture
- disp (bool, optional) – Display results to console, default is False

Returns bases (dict) – Ultimate analysis bases calculated from given as-received basis. Bases are represented by keys in the dictionary where ‘ar’ is as-received basis with values that represent [C, H, O, N, S, ash, moisture], ‘dry’ is dry basis with values that represent [C, H, O, N, S, ash], and ‘daf’ is dry ash-free basis with values of [C, H, O, N].

Raises ValueError – If the ultimate analysis sum is not 100.

Example

```python
>>> ultimate_bases(49.52, 5.28, 38.35, 0.15, 0.02, 0.64, 6.04)
{'ar': [49.52, 5.28, 38.35, 0.15, 0.02, 0.64, 6.04],
 'dry': [52.70, 5.61, 40.81, 0.15, 0.02, 0.68],
 'daf': [53.06, 5.65, 41.09, 0.16, 0.02]}
```

3.21 Wood heat capacity

chemics.wood_heat_capacity.cp_wood(x, tk)

Heat capacity of wood based on moisture content and temperature

\[ c_{p,x} = \left( c_{p0} + c_{pw} \frac{x}{100} \right) / \left( 1 + \frac{x}{100} \right) + A_c \]
where \( c_{p,x} \) is heat capacity of wet wood [kJ/(kg K)], \( c_{p0} \) is heat capacity of dry wood [kJ/(kg K)], \( c_{pw} \) is heat capacity of water as 4.18 kJ/(kg K), \( x \) is moisture content [%], and \( A_c \) is an adjustment factor that accounts for the additional energy in the wood–water bond\(^1\).

The \( c_{p0} \) term is determined from

\[
c_{p0} = 0.1031 + 0.003867 T
\]

where \( T \) is temperature [K]. The \( A_c \) term is calculated from

\[
A_c = x(b_1 + b_2 T + b_3 x)
\]

with \( b_1 = -0.06191 \), \( b_2 = 2.36 \times 10^{-4} \), and \( b_3 = -1.33 \times 10^{-4} \).

**Parameters**

- \( x \) (float) – Moisture content [%]
- \( t_k \) (float) – Temperature [K]

**Returns**

- \( c_p \) (float) – Heat capacity of wood [kJ/(kg K)]

**Example**

```python
>>> cp_wood(12, 340)
1.91
```

**References**

### 3.22 Wood thermal conductivity

\( \texttt{chemics.wood_thermal_conductivity.k_wood(gb, so, x)} \)

Thermal conductivity of wood based on moisture content, volumetric shrinkage, and basic specific gravity

\[
k = G_x (B + Cx) + A
\]

where \( k \) is thermal conductivity [W/(mK)] of wood, \( G_x \) is specific gravity [-] based on volume at moisture content \( x \) [%] and \( A, B, C \) are constants.

The \( G_x \) term is determined from

\[
G_x = \frac{G_b}{1 - S_x/100}
\]

where \( G_b \) is basic specific gravity [-] and \( S_x \) is volumetric shrinkage [%] from green condition to moisture content \( x \).

The \( S_x \) term is calculated from

\[
S_x = S_o \left(1 - \frac{x}{MC_{fs}}\right)
\]

where \( S_o \) is volumetric shrinkage [%] from Table 4-3\(^1\) and \( MC_{fs} \) is the fiber saturation point assumed to be 30% moisture content.

---

Parameters

- \( g_b (\text{float}) \) – Basic specific gravity [-]
- \( s_o (\text{float}) \) – Volumetric shrinkage [%]
- \( x(\text{float}) \) – Moisture content [%]

Returns \( k (\text{float}) \) – Thermal conductivity [W/(mK)]

Example

```python
>>> k_wood(0.54, 12.3, 10)
0.1567
```

References
See the CONTRIBUTING document on GitHub for guidelines on contributing to the Chemics package.
Package index and modules

- genindex
- modindex
- search
Python Module Index

C
chemics.atomic_elements, 7
chemics.bed_expansion_factor, 10
chemics.biomass_composition, 10
chemics.bubble_rise_velocity, 14
chemics.chemical_equation, 15
chemics.choking_velocity, 17
chemics.conversions, 19
chemics.devol_time, 21
chemics.dimensionless_numbers, 22
chemics.gas_density, 25
chemics.gas_thermal_conductivity, 26
chemics.gas_viscosity, 28
chemics.geldart, 31
chemics.minimum_fluidization_velocity, 33
chemics.molecular_weight, 35
chemics.proximate_bases, 36
chemics.terminal_velocity, 36
chemics.transport_disengaging_height, 39
chemics.transport_velocity, 40
chemics.ultimate_bases, 41
chemics.wood_heat_capacity, 41
chemics.wood_thermal_conductivity, 42
Index

A
archimedes() (in module chemics.dimensionless_numbers), 22

B
biocomp() (in module chemics.biomass_composition), 11
biot() (in module chemics.dimensionless_numbers), 22

C
ChemicalEquation (class in chemics.chemical_equation), 15
chemics.atomic_elements (module), 7
chemics.bed_expansion_factor (module), 10
chemics.biomass_composition (module), 10
chemics.bubble_rise_velocity (module), 14
chemics.chemical_equation (module), 15
chemics.choking_velocity (module), 17
chemics.conversions (module), 19
chemics.devol_time (module), 21
chemics.dimensionless_numbers (module), 22
chemics.gas_density (module), 25
gas_thermal_conductivity (module), 26
chemics.gas_viscosity (module), 28
chemics.geldart (module), 31
chemics.minimum_fluidization_velocity (module), 33
chemics.molecular_weight (module), 35
chemics.proximate_bases (module), 36
chemics.terminal_velocity (module), 36
chemics.transport_disengaging_height (module), 39
gas_thermal_conductivity (module), 26
chemics.ultimate_bases (module), 41
chemics.wood_heat_capacity (module), 41
chemics.wood_thermal_conductivity (module), 42
cp_wood() (in module chemics.wood_heat_capacity), 41

D
devol_time() (in module chemics.devol_time), 21

F
fbexp() (in module chemics.bed_expansion_factor), 10

G
geldart_chart() (in module chemics.geldart), 31

K
k_gas_inorganic() (in module chemics.gas_thermal_conductivity), 26
k_gas_organic() (in module chemics.gas_thermal_conductivity), 27
k_wood() (in module chemics.wood_thermal_conductivity), 42

M
massfrac_to_molefrac() (in module chemics.conversions), 19
molefrac_to_massfrac() (in module chemics.conversions), 20
mu_gas() (in module chemics.gas_viscosity), 28
mu_graham() (in module chemics.gas_viscosity), 29
mu_herning() (in module chemics.gas_viscosity), 30
mw() (in module chemics.molecular_weight), 35
mw_mix() (in module chemics.molecular_weight), 35

P
plot_biocomp() (in module chemics.biomass_composition), 12
prandtl() (in module chemics.dimensionless_numbers), 23
proximate_bases() (in module chemics.proximate_bases), 36
Chemics Documentation, Release 20.9

pyroI() (in module chemics.dimensionless_numbers), 23
pyroII() (in module chemics.dimensionless_numbers), 24
R
reynolds() (in module chemics.dimensionless_numbers), 24
rhog() (in module chemics.gas_density), 25
S
slm_to_lpm() (in module chemics.conversions), 20
T
tdh_chan() (in module chemics.transport_disengaging_height), 39
tdh_horio() (in module chemics.transport_disengaging_height), 39
U
ubr_holland() (in module chemics.bubble_rise_velocity), 14
ubr_kunii() (in module chemics.bubble_rise_velocity), 15
uch_bifan() (in module chemics.choking_velocity), 17
uch_leung() (in module chemics.choking_velocity), 17
uch_matsen() (in module chemics.choking_velocity), 17
uch_psri() (in module chemics.choking_velocity), 17
uch_punwani() (in module chemics.choking_velocity), 18
uch_yang() (in module chemics.choking_velocity), 18
uch_yousfi() (in module chemics.choking_velocity), 19
uch_zhang() (in module chemics.choking_velocity), 19
ultimate_bases() (in module chemics.ultimate_bases), 41
ut() (in module chemics.terminal_velocity), 36
Index