## Contents

### 1 fluids tutorial

1.1 Importing .......................................................... 3  
1.2 Design philosophy ............................................... 3  
1.3 Dimensionless numbers ........................................ 4  
1.4 Miscellaneous utilities ....................................... 4  
1.5 Friction factors .................................................. 6  
1.6 Pipe schedules .................................................. 7  
1.7 Wire gauges ..................................................... 8  
1.8 Tank geometry ................................................... 8  
1.9 Miscellaneous geometry ...................................... 9  
1.10 Atmospheric properties ..................................... 10  
1.11 Compressor sizing .......................................... 12  
1.12 Gas pipeline sizing .......................................... 14  
1.13 Gas pipeline sizing: Empirical equations ............... 16  
1.14 Drag and terminal velocity ................................ 17  
1.15 Pressure drop through packed beds .................... 18  
1.16 Pressure drop through piping ............................. 19  
1.17 Control valve sizing: Introduction ...................... 21  
1.18 Control valve sizing: Liquid flow ....................... 22  
1.19 Control valve sizing: Gas flow ........................... 23  
1.20 Electric motor sizing ....................................... 24  
1.21 Particle Size Distributions ................................ 26  

### 2 API Reference .................................................. 29  

2.1 Atmospheric properties (fluids.atmosphere) .............. 29  
2.2 Compressible flow and compressor sizing (fluids.compressible) ........................................... 42  
2.3 Control valve sizing and rating (fluids.control_valve) ......................................................... 63  
2.4 Dimensionless numbers (fluids.core) .................... 76  
2.5 Drag and terminal velocity (fluids.drag) ................ 109  
2.6 Filter pressure drop (fluids.filters) ..................... 124  
2.7 Fittings pressure drop (fluids.fittings) .................. 128  
2.8 Orifice plates, flow nozzles, Venturi tubes, cone and wedge meters (fluids.flow_meter) ........ 179  
2.9 Friction factor and pipe roughness (fluids.friction) .... 200  
2.10 Tank and helical coil sizing (fluids.geometry) .......... 244  
2.11 Jet Pump (ejector/eductor) Sizing and Rating (fluids.jet_pump) ........................................... 280  
2.12 Mixing (fluids.mixing) ..................................... 283
Fluids is open-source software for engineers and technicians working in the fields of chemical, mechanical, or civil engineering. It includes modules for piping, fittings, pumps, tanks, compressible flow, open-channel flow, and more.

Module Contents:
1.1 Importing

Fluids can be imported as a standalone library, or all of its functions and classes may be imported with star imports:

```python
>>> import fluids  # Good practice
>>> from fluids import *  # Bad practice but convenient
```

All functions are available from either the main fluids module or the submodule; i.e. both fluids.friction_factor and fluids.friction.friction_factor are valid ways of accessing a function.

1.2 Design philosophy

Like all libraries, this was developed to scratch my own itches. Since its public release it has been found useful by many others, from students across the world to practicing engineers at some of the world’s largest companies.

The bulk of this library’s API is considered stable; enhancements to functions and classes will still happen, and default methods when using a generic correlation interface may change to newer and more accurate correlations as they are published and reviewed.

To the extent possible, correlations are implemented depending on the highest level parameters. The friction_factor correlation does not accept pipe diameter, velocity, viscosity, density, and roughness - it accepts Reynolds number and relative roughness. This makes the API cleaner and encourages modular design.

All functions are designed to accept inputs in base SI units. However, any set of consistent units given to a function will return a consistent result; for instance, a function calculating volume doesn’t care if given an input in inches or meters; the output units will be the cube of those given to it. The user is directed to unit conversion libraries such as pint to perform unit conversions if they prefer not to work in SI units.

The standard math library is used in all functions except where special functions from numpy or scipy are necessary. SciPy is used for root finding, interpolation, scientific constants, ode integration, and its many special mathematical functions not present in the standard math library. No other libraries will become required dependencies; anything else is optional.
To allow use of numpy arrays with fluids, a `vectorized` module is implemented, which wraps all of the fluids functions with `np.vectorize`. Instead of importing from fluids, the user can import from fluids.vectorized:

```python
>>> from fluids.vectorized import *
>>> friction_factor(Re=[100, 1000, 10000], eD=0)
array([0.64 , 0.064 , 0.03088295])
```

It is possible to switch back and forth between the namespaces with a subsequent import:

```python
>>> from fluids import *
```

## 1.3 Dimensionless numbers

More than 30 Dimensionless numbers are available in `fluids.core`:

Calculation of `Reynolds` and `Prandtl` number for water flowing in a 0.01 m diameter pipe at 1.5 m/s:

```python
>>> fluids.core.Reynolds(D=0.01, rho=1000, V=1.5, mu=1E-3)
15000.0
>>> fluids.core.Prandtl(rho=1000, mu=1E-3, Cp=4200, k=0.6)
7.000000000000001
```

Where different parameters may be used with a dimensionless number, either a separate function is created for each or both sets of parameters are can be specified. For example, instead of specifying viscosity and density for the Reynolds number calculation, kinematic viscosity could have been used instead:

```python
>>> Reynolds(D=0.01, V=1.5, nu=1E-6)
15000.0
```

In the case of groups like the Fourier number, used in both heat and mass transfer, two separate functions are available, `Fourier_heat` and `Fourier_mass`. The heat transfer version supports specifying either the density, heat capacity, and thermal conductivity - or just the thermal diffusivity. There is no equivalent set of three parameters for the mass transfer version; it always requires mass diffusivity.

```python
>>> Fourier_heat(t=1.5, L=2, rho=1000., Cp=4000., k=0.6)
5.625e-08
>>> Fourier_heat(1.5, 2, alpha=1E-7)
3.75e-08
>>> Fourier_mass(t=1.5, L=2, D=1E-9)
3.750000000000005e-10
```


## 1.4 Miscellaneous utilities

More than just dimensionless groups are implemented in `fluids.core`. 

Chapter 1. Fluids tutorial
Converters between loss coefficient, L/D equivalent, length of pipe, and pressure drop are available. It is recommended to convert length/diameter equivalents and lengths of pipe at specified friction factors to loss coefficients using the \texttt{K\_from\_L\_equiv} and \texttt{K\_from\_f} functions respectively. They can all be summed easily afterwards.

\begin{verbatim}
>>> K\_from\_f(fd=0.018, L=100., D=.3)
  6.0
>>> K\_from\_L\_equiv(L\_D=240, fd=0.02)
  4.8
\end{verbatim}

Either head loss or pressure drop can be calculated once the total loss coefficient \( K \) is known using \texttt{head\_from\_K} or \texttt{dP\_from\_K} respectively. Head loss does not require knowledge of the fluid’s density, but pressure drop does.

\begin{verbatim}
>>> head\_from\_K(K=(6+4.8), V=3)
  4.955820795072732
>>> dP\_from\_K(K=(6+4.8), rho=1000, V=3)
  48600.0
\end{verbatim}

If a \( K \) value is known and desired to be converted to a L/D ratio or to an equivalent length of pipe, that calculation is available as well with \texttt{L\_from\_K} or \texttt{L\_equiv\_from\_K} respectively:

\begin{verbatim}
>>> L\_from\_K(K=6, fd=0.018, D=.3)
  100.0
>>> L\_equiv\_from\_K(3.6, fd=0.02)
  180.0
\end{verbatim}

Pressure and head are also convertible with the \texttt{head\_from\_P} and \texttt{P\_from\_head} functions:

\begin{verbatim}
>>> head\_from\_P(P=98066.5, rho=1000)
  10.000000000000002
>>> P\_from\_head(head=5., rho=800.)
  39226.6
\end{verbatim}

Also implemented in \texttt{fluids.core} are the following:

\texttt{thermal\_diffusivity}:

\begin{verbatim}
>>> thermal\_diffusivity(k=0.02, rho=1., Cp=1000.)
  2e-05
\end{verbatim}

Speed of sound in an ideal gas \texttt{c\_ideal\_gas}: (requires temperature, isentropic exponent \( Cp/Cv \)):

\begin{verbatim}
>>> c\_ideal\_gas(T=303, k=1.4, MW=28.96)
  348.9820361755092
\end{verbatim}

A converter between dynamic and kinematic viscosity \texttt{nu\_mu\_converter}:

\begin{verbatim}
>>> nu\_mu\_converter(rho=998., nu=1.0E-6)
  0.000998
>>> nu\_mu\_converter(998., mu=0.000998)
  1e-06
\end{verbatim}

Calculation of \texttt{gravity} on earth as a function of height and latitude (input in degrees and height in meters):

\begin{verbatim}
>>> gravity(latitude=55, H=1E6)
  6.729011976863571
\end{verbatim}
1.5 Friction factors

Friction factor is easily calculable with `friction_factor`.

```python
>>> epsilon = 1.5E-6  # m, clean steel
>>> fluids.friction.friction_factor(Re=15000, eD=epsilon/0.01)
0.02808790938573186
```

The transition to laminar flow is implemented abruptly at Re=2040, one of the latest experimental results which is accurate to +/- 10. If the Reynolds number is in the laminar regime, the transition to a different correlation happens automatically and the well-known solution \( \frac{f_d}{Re} = \frac{64}{Re} \) is given.

```python
>>> fluids.friction.friction_factor(Re=150)
0.4266666666666667
```

Friction factor in curved pipes is available as `friction_factor_curved`. The curved friction factor is applicable for helices and coils, and to a lesser extent curved bends.

```python
>>> friction_factor_curved(Re=15000, Di=.01, Dc=2.5, roughness=1.5E-6)
0.02984622907277626
```

The critical Reynolds number for curved pipes is increased compared to straight pipe flow, and is a function of the curvature of the pipe. The preferred method to calculate the transition (used by default for the automatic regime transition) is the method of Schmidt (1967) `helical_transition_Re_Schmidt`.

```python
>>> helical_transition_Re_Schmidt(Di=.01, Dc=2.5)
3948.7442097768603
```

Although roughness is a hard value to know without measuring it for a pipe, several hundred pipe conditions have had their roughness values measured in the literature, and they can be searched through using fuzzy matching and the functions `nearest_material_roughness` and `material_roughness`.

```python
>>> nearest_material_roughness('Used water piping', clean=False)
'Seamless steel tubes, Used water piping'
>>> material_roughness('Seamless steel tubes, Used water piping')
0.0015
```

The `material_roughness` function can also be used directly, but in that case there is no feedback about the material which was found.

```python
>>> material_roughness('glass')
1e-05
```

As fuzzy string matching is a pretty terrible solution, it is encouraged to find the desired string in the actual source code of `fluids`.

There is one more way of obtaining the roughness of a clean pipe, developed by Farshad and Rieke (2006) `roughness_Farshad`. It has been established that in commercial pipe, the larger the diameter, the larger the roughness.

```python
>>> roughness_Farshad('Carbon steel, bare', D=0.05)
3.529128126365038e-05
```

Only the following types of clean, new pipe have data available:

- ‘Plastic coated’
- ‘Carbon steel, honed bare’
- ‘Cr13, electropolished bare’
- ‘Cement lining’
- ‘Carbon steel, bare’
- ‘Fiberglass lining’
- ‘Cr13, bare’

There is also a term called Transmission factor, used in many pipeline applications. It is effectively a variant on friction factor. They can be inter-converted with the `transmission_factor` function.

```python
>>> transmission_factor(fd=0.0185)  # calculate transmission factor
14.704292441876154
>>> transmission_factor(F=20)  # calculate Darcy friction factor
0.01
```

## 1.6 Pipe schedules

ASME/ANSI pipe tables from B36.10M-2004 and B36-19M-2004 are implemented in fluids.piping.

Piping can be looked up based on nominal pipe size, outer diameter, or inner diameter with the `nearest_pipe` function.

```python
>>> nearest_pipe(NPS=2)  # returns NPS, inside diameter, outer diameter, wall thickness
(2, 0.05248, 0.0603, 0.00391)
```

When looking up by actual diameter, the nearest pipe as large or larger then requested is returned:

```python
>>> NPS, Di, Do, t = nearest_pipe(Di=0.5)
>>> Di
0.57504
>>> nearest_pipe(Do=0.5)
(20, 0.47781999999999997, 0.508, 0.01509)
```


```python
>>> nearest_pipe(Do=0.5, schedule='40S')
(20, 0.48894, 0.508, 0.009529999999999999)
>>> nearest_pipe(Do=0.5, schedule='80')
(20, 0.45562, 0.508, 0.02619)
```

If a diameter which is larger than any pipe in the schedule is input, an exception is raised:

```python
>>> nearest_pipe(Do=1)
Traceback (most recent call last):
  File "<stdin>"", line 1, in <module>
  File "fluids/piping.py", line 276, in nearest_pipe
    raise ValueError('Pipe input is larger than max of selected schedule')
ValueError: Pipe input is larger than max of selected schedule
```
1.7 Wire gauges

The construction of mechanical systems often uses the “gauge” systems, a variety of old imperial conversions between plate or wire thickness and a dimensionless number. Conversion from and to the gauge system is done by the `gauge_from_t` and `t_from_gauge` functions.

Looking up the gauge from a wire of known diameter approximately 1.2 mm:

```python
gauge_from_t(0.0012)
```

18

The reverse conversion:

```python
t_from_gauge(18)
```

0.001245

Other schedules are also supported:

- Birmingham Wire Gauge (BWG) ranges from 0.2 (0.5 inch) to 36 (0.004 inch).
- American Wire Gauge (AWG) ranges from 0.167 (0.58 inch) to 51 (0.00099 inch). These are used for electrical wires.
- Steel Wire Gauge (SWG) ranges from 0.143 (0.49 inch) to 51 (0.0044 inch). Also called Washburn & Moen wire gauge, American Steel gauge, Wire Co. gauge, and Roebling wire gauge.
- Music Wire Gauge (MWG) ranges from 0.167 (0.004 inch) to 46 (0.18 inch). Also called Piano Wire Gauge.
- British Standard Wire Gage (BSWG) ranges from 0.143 (0.5 inch) to 51 (0.001 inch). Also called Imperial Wire Gage (IWG).
- Stub’s Steel Wire Gage (SSWG) ranges from 1 (0.227 inch) to 80 (0.013 inch)

```python
t_from_gauge(18, schedule='AWG')
```

0.00102362

1.8 Tank geometry

Sizing of vessels and storage tanks is implemented in an object-oriented way as `TANK` in `fluids.geometry`. All results use the exact equations; all are documented in the many functions in `fluids.geometry`.

```python
T1 = TANK(D=1.2, L=4, horizontal=False)
T1.V_total, T1.A # Total volume of the tank and its surface area
```

(4.523893421169302, 17.34159144781566)

By default, tanks are cylinders without heads. Tank heads can be specified to be conical, ellipsoidal, torispherical, guppy, or spherical. The heads can be specified independently. The diameter and length are not required; the total volume desired can be specified along with the length to diameter ratio.

```python
T1 = TANK(V=10, L_over_D=0.7, sideB='conical', horizontal=False)
T1.L, T1.D
```

(1.7731788548899077, 2.5331126498427254)

Conical, ellipsoidal, guppy and spherical heads are all governed only by one parameter, $a$, the distance the head extends out from the main tank body. Torispherical heads are governed by two parameters $k$ and $f$. If these parameters are not provided, the distance the head extends out will be 25% of the size of the tank’s diameter. For torispherical heads, the distance is similar but more complicated.
Each TANK has __repr__ implemented, to describe the tank when printed.

Torispherical tanks default to the ratios specified as ASME F&D. Other standard ratios can also be used; the documentation for <TANK> lists their values. Here we implement DIN 28011’s ratios.

Partial volume lookups are also useful. This is useful when the height of fluid in the tank is known, but not the volume. The reverse calculation is also implemented, and useful when doing dynamic simulation and to calculate the new height after a specified volume of liquid is removed.

Surface areas of the heads and the main body are available as well as the total surface area of the tank.

1.9 Miscellaneous geometry

In addition to sizing all sorts of tanks, helical coils are supported and so are a number of other simple calculations.

Sphericity is implemented as sphericity, requiring a calculated surface area and volume. For a cube of side length 3, the surface area is 6*a^2=54 and volume a^3=27. Its sphericity is then:

Aspect ratio of is implemented as aspect_ratio; for example, a rectangle 0.2 m by 2 m:

Circularity, a parameter used to characterize 2d images of particles, is implemented as circularity. For a rectangle, one side length = 1, second side length = 100:
1.10 Atmospheric properties

Various main classes are available to model the atmosphere, of varying accuracy. They are the US Standard Atmosphere 1976 (ATMOSPHERE_1976), a basic but very quick model; the NRLMSISE 00 model, substantially more powerful and accurate and still the standard to this day (ATMOSPHERE_NRLMSISE00); and two models for wind speed only, Horizontal Wind Model 1993 (hwm93) and Horizontal Wind Model 2014 (hwm14). The two horizontal wind models are actually fortran codes, and are not compiled automatically on installation. Solar models are earthsun_distance, solar_position, sunrise_sunset and solar_irradiation.

ATMOSPHERE_1976 is the simplest model, and very suitable for basic engineering purposes. It supports atmospheric temperature, density, and pressure as a function of elevation. Optionally, a local temperature difference from earth’s average can be specified to correct the model to local conditions but this is only a crude approximation.

Conditions 5 km into the air:

```python
>>> atm = ATMOSPHERE_1976(Z=5000)
>>> atm.T, atm.P, atm.rho
(255.67554322180348, 54048.28614576141, 0.7364284207799743)
```

The standard also specifies simplistic formulas for calculating the thermal conductivity, viscosity, speed of sound, and gravity at a given elevation:

```python
>>> atm.g, atm.mu, atm.k, atm.v_sonic
(9.791241076982665, 1.628248135362207e-05, 0.0227319027799743, 320.5455196704035)
```

Those property routines are static methods, and can be used without instantiating an atmosphere object:

```python
>>> ATMOSPHERE_1976.gravity(Z=1E5)
9.505238763515356
>>> ATMOSPHERE_1976.sonic_velocity(T=300)
347.22080908230015
>>> ATMOSPHERE_1976.viscosity(T=400)
2.285266457680251e-05
>>> ATMOSPHERE_1976.thermal_conductivity(T=400)
0.033657148617592114
```

ATMOSPHERE_NRLMSISE00 is the recommended model, and calculates atmospheric density, temperature, and pressure as a function of height, latitude/longitude, day of year, and seconds since start of day. The model can also take into account solar and geomagnetic disturbances which effect the atmosphere at very high elevations if more parameters are provided. It is valid up to 1000 km. This model is somewhat slow; it is a Python port of the Fortran version, created by Joshua Milas. It does not support gravity profiles or transport properties, but does calculate the composition of the atmosphere (He, O, N2, O2, Ar, H2, N2 as constituents).

1000 m elevation, 45 degrees latitude and longitude, 150th day of year, 0 seconds in:

```python
>>> atm = ATMOSPHERE_NRLMSISE00(Z=1E3, latitude=45, longitude=45, day=150)
>>> atm.T, atm.P, atm.rho
(285.54408606237405, 90394.4085158811, 1.1019062026405517)
```
The composition of the atmosphere is specified in terms of individual molecules/m^3:

```python
>>> atm.N2_density, atm.O2_density
(1.7909954550444606e+25, 4.8047035072477747e+24)
```

This model uses the ideal gas law to convert particle counts to mass density. Mole fractions of each species are available as well.

```python
>>> atm.components
['N2', 'O2', 'Ar', 'He', 'O', 'H', 'N']

>>> atm.zs
[0.7811046347676225, 0.2095469403691101, 0.009343183088772914, 5.241774494627779e-06, ...
0.0, 0.0, 0.0]
```

The horizontal wind models have almost the same API, and calculate wind speed and direction as a function of elevation, latitude, longitude, day of year and time of day. hwm93 can also take as an argument local geomagnetic conditions and solar activity, but this effect was found to be so negligible it was removed from future versions of the model such as hwm14.

Calculation of wind velocity, meridional (m/sec Northward) and zonal (m/sec Eastward) for 1000 m elevation, 45 degrees latitude and longitude, 150th day of year, 0 seconds in, with both models:

```python
>>> hwm93(Z=1000, latitude=45, longitude=45, day=150)
(-0.0038965975400060415, 3.8324742317199707)

>>> hwm14(Z=1000, latitude=45, longitude=45, day=150)
(-0.9920163154602051, 0.4105832874774933)
```

These wind velocities are only historical normals; conditions may vary year to year.

The solar radiation model is based around the Sun Position Algorithm (SPA) developed by NREL; it can calculate the position of the sun in the sky at any time for any place on Earth, and can calculate how far away the sun is from Earth. The python implementation used is a slightly modified version of the Python implementation written by Tony Lorenzo and released under the BSD 3-clause license. The algorithm is published with the excellent pvlib library for solar energy modelling applications.

To determine the distance of Earth and the sun, use the `earthsun_distance` function which accepts a single datetime object and returns the distance in meters.

```python
>>> from datetime import datetime

>>> earthsun_distance(datetime(2003, 10, 17, 13, 30, 30))
149080606927.64243
```

To determine when the sun rises, sets, and is at solar noon, use the `sunrise_sunset` function, which accepts a datetime instance, a latitude, and a longitude in degrees. Note the datetime for all solar calculations should be in the local time zone - but never in daylight savings time.

```python
>>> sunrise, sunset, transit = sunrise_sunset(datetime(2018, 4, 17, 13, ...
43, 5), 51.0486, -114.07)

>>> sunrise
datetime.datetime(2018, 4, 17, 6, 36, 55, 782660)

>>> sunset
datetime.datetime(2018, 4, 17, 20, 34, 4, 249326)

>>> transit
datetime.datetime(2018, 4, 17, 13, 55, 46, 682665)
```

To determine where in the sky the sun appears at any location and time, use the `solar_position` function, which requires a datetime instance, a latitude, and a longitude.
The function returns several other properties which may be of interest. Its first return value, apparent_zenith, is the zenith which an observer on the ground would see the sun at after accounting for atmospheric refraction. To more accurately calculate the solar position, the temperature and pressure at ground level are required as well - as they impact the refraction as well; these arguments are accepted as well by `solar_position` for more accuracy. When specifying pressure, be sure to use the real pressure of the site - not an adjusted to standard conditions one as reported by weather stations!

```python
>>> solar_position(datetime(2003, 10, 17, 13, 30, 30), 51.0486, -114.07, T=290, P=8.9E4)
60.3701556038549
```

The primary application of sun position is for calculating the amount of sunlight received by an object, via the `solar_irradiation` function. Unlike the previous functions, it requires an installation of `pvlib` to work.

In addition to the arguments previously discussed, the surface_tilt and surface_azimuth of the object are required. The object is assumed to be a plane only - other objects need to be discretized into planes through finite-element calculations. The elevation is required, as well as the average albedo of the ground surrounding the object (not immediately; within several kilometers). The calculation is then straightforward:

```python
>>> solar_irradiation(Z=1100.0, latitude=51.0486, longitude=-114.07,...
... moment=datetime(2018, 4, 15, 13, 43, 5), surface_tilt=41.0,...
... surface_azimuth=180.0, albedo=0.25)
(1065.7622492480543, 945.2657257434173, 120.49652350463705, 95.31534254980346, 25.18118095483359)
```

The first return value is the solar radiation which hits the object, in W/m^2. The next two are the components of the radiation that comes 1) directly from the sun and 2) diffusely, after being reflected from some other object. The final two return values break up the diffuse light into 3) a component reflected only in the sky and clouds and 4) a component caused by earth’s albedo, bounding off the surface, then the sky, before hitting the object.

Note that if not provided, the temperature and pressure of the ground are obtained via the `ATMOSPHERE_NRLMSISE00` class, but this quadruples the time required for the calculation.

### 1.11 Compressor sizing

Both isothermal and isentropic/polytropic compression models are implemented in `fluids.compressible`. Isothermal compression calculates the work required to compress a gas from one pressure to another at a specified temperature. This is the best possible case for compression; all actual compressors require more work to do the compression. By making the compression take a large number of stages and cooling the gas between stages, this can be approached reasonable closely. Integrially geared compressors are often used for this purpose.

The function `isothermal_work_compression` provides this calculation.

```python
>>> isothermal_work_compression(P1=1E5, P2=1E6, T=300)
5743.425357533477
```

Work is calculated on a J/mol basis. If the second pressure is lower than the first, a negative work will result and you are modeling an expander instead of a compressor. Gas compressibility factor can also be specified. The lower the gas’s compressibility factor, the less power required to compress it.
There is only one function implemented to model both isentropic and polytropic compressors, as the only difference is that a polytropic exponent \( n \) is used instead of the gas's isentropic exponent \( \frac{C_p}{C_v} \) \( k \) and the type of efficiency is changed. The model requires initial temperature, inlet and outlet pressure, isentropic exponent or polytropic exponent, and optionally an efficiency.

Compressing air from 1 bar to 10 bar, with inlet temperature of 300 K and efficiency of 78% with the `isentropic_work_compression` function:

```
>>> isentropic_work_compression(P1=1E5, P2=1E6, T1=300, k=1.4, eta=0.78) # work, J/mol
10416.873455626454
```

The model allows for the inlet or outlet pressure or efficiency to be calculated instead of the work:

```
>>> isentropic_work_compression(T1=300, P1=1E5, P2=1E6, k=1.4, W=10416) # Calculate
   efficiency
0.7800654085434559
>>> isentropic_work_compression(T1=300, P1=1E5, k=1.4, W=10416, eta=0.78) # Calculate
   P2
999858.5366533266
>>> isentropic_work_compression(T1=300, P2=1E6, k=1.4, W=10416, eta=0.78) # Calculate
   P1
100014.14833613831
```

The approximate temperature rise can also be calculated with the function `isentropic_T_rise_compression`.

```
>>> T2 = isentropic_T_rise_compression(P1=1E5, P2=1E6, T1=300, k=1.4, eta=0.78)
>>> T2, T2-300
# outlet temperature and temperature rise, K
(657.960664955096, 357.96066495509604)
```

It is more accurate to use an enthalpy-based model which incorporates departure functions.

Polytropic exponents and efficiencies are convertible to isentropic exponents and efficiencies with `isentropic_efficiency` and `polytropic_exponent`. For the above example, with \( k=1.4 \) and \( 'eta_s'=0.78 \):

```
>>> eta_p = isentropic_efficiency(P1=1E5, P2=1E6, k=1.4, eta_s=0.78) # with eta_s
   specified, returns polytropic efficiency
>>> n = polytropic_exponent(k=1.4, eta_p=eta_p)
>>> eta_p, n
(0.8376785349411107, 1.517631868575738)
```

With those results, we can prove the calculation worked by calculating the work required using these polytropic inputs:

```
>>> isentropic_work_compression(P1=1E5, P2=1E6, T1=300, k=n, eta=eta_p)
10416.873455626452
```

The work is the same as calculated with the original inputs. Note that the conversion is specific to three inputs: Inlet pressure; outlet pressure; and isentropic exponent \( k \). If any of those change, then the calculated polytropic exponent and efficiency will be different as well.

To go in the reverse direction, we take the case of isentropic exponent \( k =C_p/C_v=1.4, \eta_p=0.83 \) The power is calculated to be:

We first need to calculate the polytropic exponent from the polytropic efficiency:
Fluids Documentation, Release 0.1

```python
>>> n = polytropic_exponent(k=1.4, eta_p=0.83)
>>> print(n)
1.5249343832

>>> isentropic_work_compression(P1=1E5, P2=1E6, T1=300, k=n, eta=0.83)
10556.494602042329

Converting polytropic efficiency to isentropic efficiency:

```python
>>> eta_s = isentropic_efficiency(P1=1E5, P2=1E6, k=1.4, eta_p=0.83)
>>> print(eta_s)
0.769683649894

Checking the calculated power is the same:

```python
>>> isentropic_work_compression(P1=1E5, P2=1E6, T1=300, k=1.4, eta=eta_s)
10556.494602042327

1.12 Gas pipeline sizing

The standard isothermal compressible gas flow is fully implemented as `isothermal_gas`, and through a variety of numerical and analytical expressions, can solve for any of the following parameters:

- Mass flow rate
- Upstream pressure (numerical)
- Downstream pressure (analytical or numerical if an overflow occurs)
- Diameter of pipe (numerical)
- Length of pipe

Solve for the mass flow rate of gas (kg/s) flowing through a 1 km long 0.5 m inner diameter pipeline, initially at 10 bar with a density of 11.3 kg/m^3 going downstream to a pressure of 9 bar.

```python
>>> isothermal_gas(rho=11.3, fd=0.00185, P1=1E6, P2=9E5, L=1000, D=0.5)
145.4847572636031

The same case, but sizing the pipe to take 100 kg/s of gas:

```python
>>> isothermal_gas(rho=11.3, fd=0.00185, P1=1E6, P2=9E5, L=1000, m=100)
0.42971708911060613

The same case, but determining what the outlet pressure will be if 200 kg/s flow in the 0.5 m diameter pipe:

```python
>>> isothermal_gas(rho=11.3, fd=0.00185, P1=1E6, D=0.5, L=1000, m=200)
784701.0681827427

Determining pipe length from known diameter, pressure drop, and mass flow (possible but not necessarily useful):

```python
>>> isothermal_gas(rho=11.3, fd=0.00185, P1=1E6, P2=9E5, D=0.5, m=150)
937.3258027759333

Not all specified mass flow rates are possible. At a certain downstream pressure, choked flow will develop - that downstream pressure is that at which the mass flow rate reaches a maximum. An exception will be raised if such an input is specified:
The downstream pressure at which choked flow occurs can be calculated directly as well:

```
>>> P_isothermal_critical_flow(P=1E6, fd=0.00185, L=1000., D=0.5)
389699.7317645518
```

A number of limitations exist with respect to the accuracy of this model:

- Density dependence is that of an ideal gas.
- If calculating the pressure drop, the average gas density cannot be known immediately; iteration must be used to correct this.
- The friction factor depends on both the gas density and velocity, so it should be solved for iteratively as well. It changes throughout the pipe as the gas expands and velocity increases.
- The model is not easily adapted to include elevation effects due to the acceleration term included in it.
- As the gas expands, it will change temperature slightly, further altering the density and friction factor.

We can explore how the gas density and friction factor effect the model using the thermo library for chemical properties.

Compute the downstream pressure of 50 kg/s of natural gas flowing in a 0.5 m diameter pipeline for 1 km, roughness = 5E-5 m:

```
>>> from thermo import *
>>> from fluids import *

>>> D = 0.5
>>> L = 1000
>>> epsilon = 5E-5
>>> S1 = Stream('natural gas', P=1E6, m=50)
>>> V = S1.Q/(pi/4*D**2)
>>> Re = S1.Reynolds(D=D, V=V)
>>> fd = friction_factor(Re=Re, eD=epsilon/D)
>>> P2 = isothermal_gas(rho=S1.rho, fd=fd, P1=S1.P, D=D, L=L, m=S1.m)
>>> P2
877424.4964411375
```

In the above example, the friction factor was calculated using the density and velocity of the gas when it enters the stream. However, the average values, at the middle pressure, and more representative. We can iterate to observe the effect of using the average values:
As can be seen, the system converges very quickly. The difference in calculated pressure drop is approximately 1%.

1.13 Gas pipeline sizing: Empirical equations

In addition to the actual model, many common simplifications used in industry are implemented as well. These are equally capable of solving for any of the following inputs:

- Mass flow rate
- Upstream pressure
- Downstream pressure
- Diameter of pipe
- Length of pipe

None of these models include an acceleration term. In addition to reducing their accuracy, it allows all solutions for the above variables to be analytical. These models cannot predict the occurrence of choked flow, and model only turbulent, not laminar, flow. Most of these models do not depend on the gas’s viscosity.

Rather than using mass flow rate, they use specific gravity and volumetric flow rate. The volumetric flow rate is specified with respect to a reference temperature and pressure. The defaults are 288.7 K and 101325 Pa, dating to the old imperial standard of 60°F. The specific gravity is with respect to air at the reference conditions. As the ideal gas law is used in each of these models, in addition to pressure and specific gravity the average temperature in the pipeline is required. Average compressibility factor is an accepted input to all models and corrects the ideal gas law’s ideality.

The full list of approximate models is as follows:

- Panhandle_A
- Panhandle_B
- Weymouth
- Oliphant
- Fritzschke
- Muller
- IGT
• \textit{Spitzglass\_high}
• \textit{Spitzglass\_low}

As an example, calculating flow for a pipe with diameter 0.34 m, upstream pressure 90 bar and downstream pressure 20 bar, 160 km long, 0.693 specific gravity and with an average temperature in the pipeline of 277.15 K:

\begin{verbatim}
>>> Panhandle_A(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, Tavg=277.15)
42.56082051195928
\end{verbatim}

Each model also includes a pipeline efficiency term, ranging from 0 to 1. These are just empirical correction factors. Some of the models were developed with theory and a correction factor applied always; others are more empirical, and have a default correction factor. 0.92 is the default for the Panhandle A/B, Weymouth, and Oliphant models; the rest default to a correction of 1 i.e. no correction at all.

The Muller and IGT models are the most accurate and recent approximations. They both depend on viscosity.

\begin{verbatim}
>>> Muller(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, mu=1E-5, Tavg=277.15)
   60.45796698148659
>>> IGT(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, mu=1E-5, Tavg=277.15)
   48.92351786788815
\end{verbatim}

These empirical models are included because they are mandated in many industrial applications regardless of their accuracy, and correction factors have already been determined.

A great deal of effort was spent converting these models to base SI units and checking the coefficients used in each model with multiple sources. In many cases multiple sets of coefficients are available for a model; the most authoritative or common ones were used in those cases.

\section*{1.14 Drag and terminal velocity}

A number of spherical particle drag correlations are implemented.

In the simplest case, consider a spherical particle of diameter $D=1$ mm, density $=3400$ kg/m$^3$, travelling at 30 m/s in air with viscosity $\mu=1E-5$ Pa*s and density $1.2$ kg/m$^3$.

We calculate the particle Reynolds number:

\begin{verbatim}
>>> Re = Reynolds(V=30, rho=1.2, mu=1E-5, D=1E-3)
>>> Re
   3599.99999999999995
\end{verbatim}

The drag coefficient $C_d$ can be calculated with no other parameters from \textit{drag\_sphere}:

\begin{verbatim}
>>> drag\_sphere(Re)
   0.3914804681941151
\end{verbatim}

The terminal velocity of the particle is easily calculated with the \textit{v\_terminal} function.

\begin{verbatim}
>>> v\_terminal(D=1E-3, rhop=3400, rho=1.2, mu=1E-5)
   8.971223953182939
\end{verbatim}

Very often, we are not interested in just what the velocity of the particle will be at terminal conditions, but on the distance it will travel and the particle will never have time to reach terminal conditions. An integrating function is available to do that. Consider that same particle being shot directly down from a helicopter 100 m high.

The integrating function, \textit{integrate\_drag\_sphere}, performs the integral with respect to time. At one second, we can see the \textit{(velocity, distance travelled)}:
After integrating to 10 seconds, we can see the particle has travelled 97 meters and is almost on the ground.

For this example simply using the terminal velocity would have given an accurate estimation of distance travelled:

Many engineering applications such as direct contact condensers do operate far from terminal velocity however, and this function is useful there.

### 1.15 Pressure drop through packed beds

Twelve different packed bed pressure drop correlations are available. A meta function which allows any of them to be selected and automatically selects the most accurate correlation for the given parameters.

Pressure drop through a packed bed depends on the density, viscosity and velocity of the fluid, as well as the diameter of the particles, the amount of free space in the bed (voidage), and to a lesser amount the ratio of particle to tube diameter and the shape of the particles.

Consider 0.8 mm pebbles with 40% empty space with water flowing through a 2 m column creeping flow at a superficial velocity of 1 mm/s. We can calculate the pressure drop in Pascals using the \( dP_{\text{packed\_bed}} \) function:

The method can be specified manually as well, for example the commonly used Ergun equation:

Incorporation of the tube diameter will add wall effects to the model.

Models can be used directly as well. The length of the column is an optional input; if not provided, the result will be in terms of Pa/m.

If the column diameter was 0.5 m, the flow rate in m^3/s would be:

0.00019634954084936208
The holdup (total volume of the column holding fluid not particles) would be:

```
>>> (pi/4*0.5**2)*(2)*0.4 # A_column*H_column*voidage
0.15707963267948966
```

Not all particles are spherical. There have been correlations published for specific shapes, but what is often performed
is simply an adjustment of particle diameter by its sphericity in the correlation, with the effective \( dp \) used as the product
of the actual \( dp \) and the sphericity of the particle. The less spherical the particles, the higher the pressure drop. This is
supported in all of the correlations.

```
>>> dP_packed_bed(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3, L=2, Dt=0.01, _
˓
→sphericity=0.9)
3050.419598116882
```

While it is easy to measure the volume of particles added to a given column and determine the voidage
experimentally, this does not help in the design process. Several authors have methodically filled columns
with particles of different sizes and created correlations in terms of sphericity and particle to tube diam-
eter ratios. Three such correlations are implemented in fluids, one generally using sphericity, one for
spheres, and one for cylinders (voidage_Benyahia_Oneil, voidage_Benyahia_Oneil_spherical and
voidage_Benyahia_Oneil_cylindrical respectively).

1 mm spheres in a 5 cm diameter tube:

```
>>> voidage_Benyahia_Oneil_spherical(Dp=.001, Dt=.05)
0.3906653157443224
```

1 mm diameter cylinder 5 mm long in a 5 cm diameter tube:

```
>>> V_cyl = V_cylinder(D=0.001, L=0.005)
>>> D_sphere_eq = (6*V_cyl/pi)**(1/3.)
>>> A_cyl = A_cylinder(D=0.001, L=0.005)
>>> sph = sphericity(A=A_cyl, V=V_cyl)
>>> voidage_Benyahia_Oneil_cylindrical(Dpe=D_sphere_eq, Dt=0.05, sphericity=sph)
0.3754895273247688
```

Same calculation, but using the general correlation for all shapes:

```
>>> voidage_Benyahia_Oneil(Dpe=D_sphere_eq, Dt=0.05, sphericity=sph)
0.4425769555048246
```

## 1.16 Pressure drop through piping

It is straightforward to calculate the pressure drop of fluid flowing in a pipeline with any number of fittings using the
fluids library’s `fluids.fittings` submodule.

15 m of piping, with a sharp entrance and sharp exit, two 30 degree miter bends, one rounded bend 45 degrees, 1
sharp contraction to half the pipe diameter and 1 sharp expansion back to the normal pipe diameter (water, \( V=3 \) m/s,
\( D_i=0.05 \), roughness 0.01 mm):

```
>>> Re = Reynolds(V=3, D=0.05, rho=1000, mu=1E-3)
>>> fd = friction_factor(Re, eD=1E-5/0.05)
>>> K = K_from_f(fd, L=15, D=0.05)
>>> K += entrance_sharp()
>>> K += exit_normal()
>>> K += 2*bend_miter(angle=30)
```
If the diameter of the piping varies, not all of the loss coefficients will be with respect to the same diameter. Each loss coefficient must be converted to one standard diameter before the total pressure drop can be calculated. The following example is solved with the optional `pint` unit compatibility module.

40 m piping, beveled entrance (10 mm length, 30 degrees, into 5 cm ID pipe), then a 30 degree miter bend, a sharp contraction to half the pipe diameter (5 m long), a 30 degree miter bend, a rounded 45 degree bend, a sharp expansion to 4 cm ID pipe (15 more meters), and a sharp exit:

```python
>>> from fluids.units import *
>>> from math import *

>>> material = nearest_material_roughness('steel', clean=True)

>>> epsilon = material_roughness(material)

>>> Q = .01*u.m**3/u.s

>>> rho = 1000*u.kg/u.m**3

>>> mu = 1E-4*u.Pa*u.s

>>> D1 = 5*u.cm

>>> D2 = 2.5*u.cm

>>> D3 = 4*u.cm

>>> L1 = 20*u.m

>>> L2 = 5*u.m

>>> L3 = 15*u.m

>>> V1 = Q/(pi/4*D1**2)

>>> Re = Reynolds(V=V1, D=D1, rho=rho, mu=mu)

>>> fd = friction_factor(Re, eD=epsilon/D1)

>>> K += entrance_beveled(Di=D1, l=10*u.mm, angle=30*u.degrees)

>>> K += bend_miter(angle=30*u.degrees)

>>> K += contraction_sharp(Di1=D1, Di2=D2)

>>> V2 = Q/(pi/4*D2**2)

>>> Re2 = Reynolds(V=V2, D=D2, rho=rho, mu=mu)

>>> fd2 = friction_factor(Re2, eD=epsilon/D2)

>>> K += change_K_basis(K_from_f(fd=fd2, L=L2, D=D2), D1=D2, D2=D1)

>>> K += change_K_basis(K1=bend_miter(angle=30*u.degrees), D1=D2, D2=D1)

>>> K += change_K_basis(K1=bend_rounded(Di=D2, angle=45*u.degrees, fd=fd2), D1=D2, D2=D1)

>>> V3 = Q/(pi/4*D3**2)

>>> Re3 = Reynolds(V=V3, D=D3, rho=rho, mu=mu)

>>> fd3 = friction_factor(Re3, eD=epsilon/D3)

>>> K += change_K_basis(K_from_f(fd=fd3, L=L3, D=D3), D1=D3, D2=D1)

>>> K += diffuser_sharp(D11=D2, D21=D3)

>>> dP_from_K(K, rho=rho, V=V1)

<Quantity(608471.881547, 'pascal')>

37920.51140146369

**Chapter 1. Fluids Tutorial**

1.17 Control valve sizing: Introduction

The now internationally-standardized methods (IEC 60534) for sizing liquid and gas valves have been implemented. Conversion factors among the different types of valve coefficients are implemented as well.

There are two forms of loss coefficient used for valves, an imperial and a metric variable called “valve flow coefficient”. Both can be converted to the standard dimensionless loss coefficient.

If one knows the actual loss coefficient of a valve, the valve flow coefficient can be calculated in either metric or imperial forms as follows. The flow coefficients are specific to the diameter of the valve. Kv, Cv, and K values can be converted easily with the functions `K_to_Kv`, `K_to_Cv`, `Cv_to_K`, `Kv_to_K`, `Cv_to_Kv`, and `Kv_to_Cv`.

```python
>>> from fluids import *
>>> K_to_Kv(K=16, D=0.016)
2.56
>>> K_to_Cv(K=16, D=0.016)
2.9596140245853606
```

If Kv or Cv are known, they can be converted to each other with the proportionality constant 1.156, which is derived from a unit conversion only. This conversion does not require valve diameter.

```python
>>> Cv_to_Kv(12)
10.379731865307619
>>> Kv_to_Cv(10.37)
11.988748998027418
```

If a Cv or Kv is obtained from a valve datasheet, it can be converted into a standard loss coefficient as follows.

```python
>>> Kv_to_K(Kv=2.56, D=0.016)
16.000000000000004
>>> Cv_to_K(Cv=3, D=0.016)
15.57211586581753
```

For a valve with a specified Kv and pressure drop, the flow rate can be calculated easily for the case of non-choked non-compressible flow (neglecting other friction losses), as illustrated in the example below for a 5 cm valve with a pressure drop 370 kPa and density of 870 kg/m^3:

```python
>>> Kv = 72.5
>>> D = 0.05
>>> dP = 370E3
>>> K = Kv_to_K(D=D, Kv=Kv)
>>> rho = 870
>>> V = (dP/((.5*rho*K))**0.5 # dP = K*0.5*rho*V^2
>>> A = pi/4*D**2
>>> Q = V*A
>>> Q
0.04151682468778643
```

Alternatively, the required Kv can be calculated from an assumed diameter and allowable pressure drop:

```python
>>> Q = .05
>>> D = 0.05
>>> dP = 370E3
>>> rho = 870
>>> A = pi/4*D**2
>>> V = Q/A
>>> K = dP/((.5*rho*V**2)
```

(continues on next page)
The approach documented above is not an adequate procedure for sizing valves however because chocked flow, compressible flow, the effect of inlet and outlet reducers, the effect of viscosity and the effect of laminar/turbulent flow all have large influences on the performance of control valves.

Historically, valve manufacturers had their own standards for sizing valves, but these have been standardized today into the IEC 60534 methods.

### 1.18 Control valve sizing: Liquid flow

To rigorously size a control valve for liquid flow, the inlet pressure, allowable pressure drop, and desired flow rate must first be known. These need to be determined taking into account the entire pipe network and the various operating conditions it needs to support; sizing the valves can be performed afterward and only if no valve with the desired performance is available does the network need to be redesigned.

To illustrate sizing a valve, we borrow an example from Emerson’s Control Valve Handbook, 4th edition (2005). It involves a flow of 800 gpm of liquid propane. The inlet and outlet pipe size is 8 inches, but the size of the valve itself is unknown. The desired pressure drop is 25 psi.

Converting this problem to SI units and using the thermo library to calculate the necessary properties of the fluid, we calculate the necessary Kv of the valve based on an assumed valve size of 3 inches:

```python
>>> from scipy.constants import *
>>> from fluids.control_valve import size_control_valve_l
>>> from thermo.chemical import Chemical

>>> P1 = 300*psi + psi # to Pa
>>> P2 = 275*psi + psi # to Pa
>>> T = 273.15 + 21 # to K
>>> propane = Chemical('propane', P=(P1+P2)/2, T=T)
>>> rho = propane.rho
>>> Psat = propane.Psat
>>> Pc = propane.Pc
>>> mu = propane.mu
>>> Q = 800*gallon/minute # to m^3/s
>>> D1 = D2 = 8*inch # to m
>>> d = 3*inch # to m
```

The standard specifies two more parameters specific to a valve:

- FL, Liquid pressure recovery factor of a control valve without attached fittings
- Fd, Valve style modifier

Both of these are factors between 0 and 1. In the Emerson handbook, they are not considered in the sizing procedure and set to 1. These factors are also a function of the diameter of the valve and are normally tabulated next to the values of Cv or Kv for a valve. Now using `size_control_valve_l` to solve for the flow coefficient:

```python
>>> Kv = size_control_valve_l(rho, Psat, Pc, mu, P1, P2, Q, D1, D2, d, FL=1, Fd=1)
>>> Kv
109.27127420992377
```

The handbook states the Cv of the valve is 121; we convert Kv to Cv:
The example in the book calculated \( C_v = 125.7 \), but doesn’t actually use the full calculation method. Either way, the valve will not carry the desired flow rate; we need to try a larger valve size. The 4 inch size is tried next in the example, which has a known \( C_v \) of 203.

The calculated \( C_v \) is well under the valve’s maximum \( C_v \); we can select it.

This model requires a vapor pressure and a critical pressure of the fluid as inputs. There is no clarification in the standard about how to handle mixtures, which do not have these values. It is reasonable to calculate vapor pressure as the bubble pressure, and the mixture’s critical pressure through a mole-weighted average.

For actual values of \( C_v \), Fl, Fd, and available diameters, an excellent resource is the Fisher Catalog 12.

### 1.19 Control valve sizing: Gas flow

To rigorously size a control valve for gas flow, the inlet pressure, allowable pressure drop, and desired flow rate must first be known. These need to be determined taking into account the entire pipe network and the various operating conditions it needs to support; sizing the valves can be performed afterward and only if no valve with the desired performance is available does the network need to be redesigned.

To illustrate sizing a valve, we borrow an example from Emerson’s Control Valve Handbook, 4th edition (2005). It involves a flow of 6 million ft\(^3\)/hour of natural gas. The inlet and outlet pipe size is 8 inches, but the size of the valve itself is unknown. The desired pressure drop is 150 psi.

Converting this problem to SI units and using the thermo library to calculate the necessary properties of the fluid, we calculate the necessary \( Kv \) of the valve based on an assumed valve size of 8 inches.

```python
>>> from scipy.constants import *
>>> from fluids.control_valve import size_control_valve_g
>>> from thermo.chemical import Chemical

>>> P1 = 214.7*psi
>>> P2 = 64.7*psi
>>> T = 16 + 273.15
>>> natural_gas = Mixture('natural gas', T=T, P=(P1+P2)/2)
>>> Z = natural_gas.Z
>>> MW = natural_gas.MW
>>> mu = natural_gas.mu
>>> gamma = natural_gas.isentropic_exponent
>>> Q = 6E6*foot**3/hour
>>> D1 = D2 = d = 8*inch  # 8-inch Fisher Design V250
```

The standard specifies three more parameters specific to a valve:

- FL, Liquid pressure recovery factor of a control valve without attached fittings
- Fd, Valve style modifier
- xT, Pressure difference ratio factor of a valve without fittings at choked flow

All three of these are factors between 0 and 1. In the Emerson handbook, FL and Fd are not considered in the sizing procedure and set to 1. \( xT \) is specified as 0.137 at full opening. These factors are also a function of the diameter of the
valve and are normally tabulated next to the values of \( Cv \) or \( Kv \) for a valve. Now using `size_control_valve_g` to solve for the flow coefficient:

```
>>> Kv = size_control_valve_g(T, MW, mu, gamma, Z, P1, P2, Q, D1, D2, d, FL=1, Fd=1, xT=.137)
>>> Kv_to_Cv(Kv)
1560.9362792230884
```

The 8-inch valve is rated with \( Cv = 2190 \). The valve is adequate to provide the desired flow because the rated \( Cv \) is higher. The calculated value in their example is 1515, differing slightly due to the properties used.

The example next goes on to determine the actual opening position the valve should be set at to provide the required flow. Their conclusion is approximately 75% open; we can do better using a numerical solver. The values of opening at different positions are obtained in this example from the valve’s datasheet.

Loading the data and creating interpolation functions so FL, Fd, and xT are all smooth functions:

```
>>> from scipy.interpolate import interp1d
>>> from scipy.optimize import newton

>>> openings = [.2, .3, .4, .5, .6, .7, .8, .9]
>>> Fds = [0.59, 0.75, 0.85, 0.92, 0.96, 0.98, 0.99, 0.99]
>>> Fls = [0.9, 0.9, 0.9, 0.85, 0.78, 0.68, 0.57, 0.45]
>>> xTs = [0.92, 0.81, 0.85, 0.63, 0.58, 0.48, 0.29, 0.14]
>>> Kvs = [24.1, 79.4, 153, 266, 413, 623, 1060, 1890]

>>> Fd_interp = interp1d(openings, Fds, kind='cubic')
>>> Fl_interp = interp1d(openings, Fls, kind='cubic')
>>> xT_interp = interp1d(openings, xTs, kind='cubic')
>>> Kv_interp = interp1d(openings, Kvs, kind='cubic')
```

Creating and solving the objective function:

```
>>> def to_solve(opening):
...     Fd = float(Fd_interp(opening))
...     Fl = float(Fl_interp(opening))
...     xT = float(xT_interp(opening))
...     Kv_lookup = float(Kv_interp(opening))
...     Kv_calc = size_control_valve_g(T, MW, mu, gamma, Z, P1, P2, Q, D1, D2, d, FL=Fl, Fd=Fd, xT=xT)
...     return Kv_calc - Kv_lookup

>>> newton(to_solve, .8)  # initial guess of 80%
0.7495168349025819
```

We see the valve should indeed be set to almost exactly 75% open to provide the desired flow.

### 1.20 Electric motor sizing

Motors are available in standard sizes, mostly as designated by the National Electrical Manufacturers Association (NEMA). To easily determine what the power of a motor will actually be once purchased, `motor_round_size` implements rounding up of a motor power to the nearest size. NEMA standard motors are specified in terms of horsepower.

```
>>> motor_round_size(1E5)  # 100 kW motor; 11.8% larger than desired
111854.98073734052
>>> from scipy.constants import hp
>>> motor_round_size(1E5)/hp  # convert to hp
150.0
```
Motors are designed to generate a certain amount of power, but they themselves are not 100% efficient at doing this and require more power due to efficiency losses. Many minimum values for motor efficiency are standardized. The Canadian standard for this is implemented in fluids as `CSA_motor_efficiency`.

```python
>>> CSA_motor_efficiency(P=5*hp)
0.855
```

Most motors are not enclosed (the default assumption), but those that are closed are more efficient.

```python
>>> CSA_motor_efficiency(P=5*hp, closed=True)
0.875
```

The number of poles in a motor also affects its efficiency:

```python
>>> CSA_motor_efficiency(P=5*hp, poles=6)
0.875
```

There is also a schedule of higher efficiency values standardized as well, normally available at somewhat higher cost:

```python
>>> CSA_motor_efficiency(P=5*hp, closed=True, poles=6, high_efficiency=True)
0.895
```

A motor will spin at more or less its design frequency, depending on its type. However, if it does not meet sufficient resistance, it will not be using its design power. This is good and bad - less power is used, but as a motor drops under 50% of its design power, its efficiency becomes terrible. The function `motor_efficiency_underloaded` has been written based on generic performance curves to estimate the underloaded efficiency of a motor. Just how bad efficiency drops off depends on the design power of a motor - higher power motors do better operating at low loads than small motors.

```python
>>> motor_efficiency_underloaded(P=1E3, load=.9)
1
>>> motor_efficiency_underloaded(P=1E3, load=.2)
0.6639347559654663
```

This needs to be applied on top of the normal motor efficiency; for example, that 1 kW motor at 20% load would have a net efficiency of:

```python
>>> motor_efficiency_underloaded(P=1E3, load=.2)*CSA_motor_efficiency(P=1E3)
0.5329404286134798
```

Many motors have Variable Frequency Drives (VFDs) which allow them to vary the speed of their rotation. The VFD is another source of inefficiency, but by allowing the pump or other piece of equipment to vary its speed, a system may be designed to be less energy intensive. For example, rather than running a pump at a certain high frequency and controlling the flow with a large control valve, the flow rate can be controlled with the VFD directly.

The efficiency of a VFD depends on the maximum power it needs to be able to generate, and the power it is actually generating at an instant (load). A table of typical modern VFD efficiencies is implemented in fluids as `VFD_efficiency`.

```python
>>> VFD_efficiency(1E5)  # 100 kW
0.97
>>> VFD_efficiency(5E3, load=.2)  # 5 kW, 20% load
0.8562
```
1.21 Particle Size Distributions

Fluids has means for calculating, fitting, and manipulating particle size distributions through the `fluids.particle_size_distribution` module. In addition to discrete and continuous distributions, there are also means to create interpolating distributions from discrete distributions, and to use any of SciPy’s statistical distributions or a custom distribution for calculations.

The following example particle size distribution shows some calculations. Note there is one more diameter point than number point - this is how the input should be given when the analysis provides classes and each bin has a range of sizes representing it. Also supported is providing as many diameter values as fraction values.

```python
>>> ds = [240, 360, 450, 562.5, 703, 878, 1097, 1371, 2141, 2676, 3345, 4181, 5226, 6532]
>>> numbers = [65, 119, 232, 410, 629, 849, 990, 981, 825, 579, 297, 111, 21, 1]
>>> psd = ParticleSizeDistribution(ds=ds, fractions=numbers, order=0)
>>> psd
<Particle Size Distribution, points=14, D[3, 3]=2450.886241 m>
```

In the above example, the analysis available was the number of particles counted in each bin. This is equivalent to having normalized the numbers into fractions; they are normalized inside the `ParticleSizeDistribution` class.

If masses in each of the different bins had been known instead, then the same constructor would be given except with `order=3`, representing a mass or volume distribution (they are the same thing for distributions with the same density for all particles).

If the data is available as a cumulative distribution, simple add the flag `cdf=True` and it will be interpreted correctly.

Important statistical parameters describing the distribution can be calculated with the methods `fluids.particle_size_distribution.ParticleSizeDistribution.mean_size` or `fluids.particle_size_distribution.ParticleSizeDistribution.mean_size_ISO`. The following example shows calculation of the size-weighted mean diameter; arithmetic mean diameter; Sauter mean diameter; and De Brouckere diameter.

```python
>>> psd.mean_size(2, 1)
1857.788857205553
>>> psd.mean_size(1, 0)
1459.3725650679328
>>> psd.mean_size(1, 2)
1857.7888572055529
>>> psd.mean_size(1, 3)
2053.2703977309357
```

An interpolated distribution exists underneath the discrete data to allow useful properties to be calculated, such as the D10 or D90:

```python
>>> psd.dn(0.1), psd.dn(0.9)
(1437.071392769334, 3911.479636364713)
```

Or probability density functions:

```python
>>> psd.pdf(1000)
0.00010632384327525043
>>> psd.cdf(5000)
0.9897400734854198
```
Statistical distributions implemented are *PSDLognormal*, *PSDGatesGaudinSchuhman*, and *PSDRosinRammler*. Discrete and continuous distributions share most methods.

```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6)
>>> psd.pdf(1e-6)  # probability density function
4487.892155358317
>>> psd.cdf(7e-6)  # cumulative distribution function
0.749508691386811
>>> psd.dn(0.1)   # At what diameter is this fraction of particles smaller than?
2.63417591480183e-06
>>> psd.mean_size(3, 2)
4.412484512922977e-06
>>> ds = psd.ds_discrete(pts=1000)  # Compare calculations with the discrete distribution
>>> fractions = psd.fractions_discrete(ds)
>>> ParticleSizeDistribution(ds=ds, fractions=fractions, order=3).mean_size(3, 2)
4.425743630583137e-06
```

It is straightforward to calculate descriptions of the distribution using the available routines:

**Volume specific surface area:**

```python
>>> psd.vssa
1359778.1436801916
```

**Span (D90 - D10):**

```python
>>> psd.dn(.9) - psd.dn(0.1)
6.85534945193373e-06
```

**Relative span (D90 - D10)/D50:**

```python
>>> (psd.dn(.9) - psd.dn(0.1))/psd.dn(0.5)
1.3710691890386744
```

**Percentile ratios, D75/D25 and D90/D10:**

```python
>>> psd.dn(0.75)/psd.dn(0.25)
1.9630310841582574
>>> psd.dn(0.9)/psd.dn(0.1)
3.60224479279158
```
2.1 Atmospheric properties (fluids.atmosphere)

This module contains models of earth’s atmosphere. Models are empirical and based on extensive research, primarily by NASA.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker or contact the author at Caleb.Andrew.Bell@gmail.com.

- Atmospheres
- Solar Radiation and Position
- Wind Models (requires Fortran compiler!)

2.1.1 Atmospheres

```python
class fluids.atmosphere.ATMOSPHERE_1976(Z, dT=0.0)
    Bases: object

US Standard Atmosphere 1976 class, which calculates T, P, rho, v_sonic, mu, k, and g as a function of altitude above sea level. Designed to provide reasonable results up to an elevation of 86,000 m (0.4 Pa). The model is also valid under sea level, to -610 meters.

Parameters

- Z [float] Elevation, [m]
- dT [float, optional] Temperature difference from standard conditions used in determining the properties of the atmosphere, [K]
```
Notes

Up to 32 km, the International Standard Atmosphere (ISA) and World Meteorological Organization (WMO) standard atmosphere are identical.

This is a revision of the US 1962 atmosphere.

References

[Rf2dc4e0525c7-1], [Rf2dc4e0525c7-2], [Rf2dc4e0525c7-3]

Examples

```python
>>> five_km = ATMOSPHERE_1976(5000)
>>> five_km.P, five_km.rho, five_km.mu
(54048.28614576141, 0.7364284207799743, 1.628248135362207e-05)
```

Attributes

- `T` [float] Temperature of atmosphere at specified conditions, [K]
- `P` [float] Pressure of atmosphere at specified conditions, [Pa]
- `rho` [float] Mass density of atmosphere at specified conditions [kg/m^3]
- `H` [float] Geopotential height, [m]
- `g` [float] Acceleration due to gravity, [m/s^2]
- `mu` [float] Viscosity of atmosphere at specified conditions, [Pa*s]
- `k` [float] Thermal conductivity of atmosphere at specified conditions, [W/m/K]
- `v_sonic` [float] Speed of sound of atmosphere at specified conditions, [m/s]

Methods

- `gravity(Z)` Method defined in the US Standard Atmosphere 1976 for calculating the gravitational acceleration above earth as a function of elevation only.
- `pressure_integral(T1, P1, dH)` Method to compute an integral of the pressure differential of an elevation difference with a base elevation defined by temperature `T1` and pressure `P1`.
- `sonic_velocity(T)` Method defined in the US Standard Atmosphere 1976 for calculating the speed of sound in air as a function of `T` only.
- `thermal_conductivity(T)` Method defined in the US Standard Atmosphere 1976 for calculating thermal conductivity of air as a function of `T` only.

Continued on next page
Table 1 – continued from previous page

| viscosity(T) | Method defined in the US Standard Atmosphere 1976 for calculating viscosity of air as a function of T only. |

\[ R = 8314.32 \]

**static density** \((T, P)\)

Method defined in the US Standard Atmosphere 1976 for calculating density of air as a function of \(T\) and \(P\). MW is defined as 28.9644 g/mol, and \(R\) as 8314.32 J/kmol/K

\[
\rho_g = \frac{P \cdot MW}{T \cdot R \cdot 1000} 
\]

**Parameters**
- \(T\) [float] Temperature, [K]
- \(P\) [float] Pressure, [Pa]

**Returns**
- rho [float] Mass density, [kg/m^3]

**static gravity** \((Z)\)

Method defined in the US Standard Atmosphere 1976 for calculating the gravitational acceleration above earth as a function of elevation only.

\[
g = g_0 \left( \frac{r_0}{r_0 + Z} \right)^2 
\]

**Parameters**
- \(Z\) [float] Elevation above sea level, [m]

**Returns**
- g [float] Acceleration due to gravity, [m/s^2]

**static pressure_integral** \((T1, P1, dH)\)

Method to compute an integral of the pressure differential of an elevation difference with a base elevation defined by temperature \(T1\) and pressure \(P1\). This is similar to subtracting the pressures at two different elevations, except it allows for local conditions (temperature and pressure) to be taken into account. This is useful for e.g. evaluating the pressure difference between the top and bottom of a natural draft cooling tower.

**Parameters**
- \(T1\) [float] Temperature at the lower elevation condition, [K]
- \(P1\) [float] Pressure at the lower elevation condition, [Pa]
- \(dH\) [float] Elevation difference for which to evaluate the pressure difference, [m]

**Returns**
- delta_P [float] Pressure difference between the elevations, [Pa]

**static sonic_velocity** \((T)\)

Method defined in the US Standard Atmosphere 1976 for calculating the speed of sound in air as a function of \(T\) only.

\[
c = \left( \frac{\gamma RT}{MW} \right)^{0.5} 
\]
Parameters
T [float] Temperature, [K]

Returns
c [float] Speed of sound, [m/s]

\textbf{static thermal_conductivity}(T)
Method defined in the US Standard Atmosphere 1976 for calculating thermal conductivity of air as a function of \( T \) only.

\[ k_g = \frac{2.64638 \times 10^{-3} T^{1.5}}{T + 245.4 \cdot 10^{-12}/T} \]

Parameters
T [float] Temperature, [K]

Returns
kg [float] Thermal conductivity, [W/m/K]

\textbf{static viscosity}(T)
Method defined in the US Standard Atmosphere 1976 for calculating viscosity of air as a function of \( T \) only.

\[ \mu_g = \frac{1.458 \times 10^{-6} T^{1.5}}{T + 110.4} \]

Parameters
T [float] Temperature, [K]

Returns
mug [float] Viscosity, [Pa*s]

\textbf{class} fluids.atmosphere.ATMOSPHERE_NRLMSISE00(Z, \text{ latitude}=0.0, \text{ longitude}=0.0, \text{ day}=0, \text{ seconds}=0.0, \text{ f107}=150.0, \text{ f107_avg}=150.0, \text{ geomagnetic_disturbance_indices}=\text{None})

\textbf{Bases: object}

NRLMSISE 00 model for calculating temperature and density of gases in the atmosphere, from ground level to 1000 km, as a function of time of year, longitude and latitude, solar activity and earth’s geomagnetic disturbance.

NRLMSISE stands for the \textit{US Naval Research Laboratory Mass Spectrometer and Incoherent Scatter Radar Exosphere} model, released in 2001; see [R4bc72d62ed0b-1] for details.

Parameters
Z [float] Elevation, [m]

\textbf{latitude} [float, optional] Latitude, between -90 and 90 [degrees]

\textbf{longitude} [float, optional] Longitude, between -180 and 180 or 0 and 360, [degrees]

\textbf{day} [float, optional] Day of year, 0-366 [day]

\textbf{seconds} [float, optional] Seconds since start of day, in UT1 time; using UTC provides no loss in accuracy [s]

\textbf{f107} [float, optional] Daily average 10.7 cm solar flux measurement of the strength of solar emissions on the 100 MHz band centered on 2800 MHz, averaged hourly; in sfu units, which are multiples of \( 10^{\sim}22 \) W/m\(^2\)/Hz; use 150 as a default [\( 10\sim^{22} \) W/m\(^2\)/Hz]
**f107_avg** [float, optional] 81-day sfu average; centered on specified day if possible, otherwise use the previous days \([10^{-22} \text{ W/m}^2/\text{Hz}]\)

**geomagnetic_disturbance_indices** [list of float, optional] List of the 7 following Ap indexes also known as planetary magnetic indexes. Has a negligible effect on the calculation. 4 is the default value often used for each of these values.

- Average daily Ap.
- 3-hour average Ap centered on the current time.
- 3-hour average Ap before the current time.
- 6-hour average Ap before the current time.
- 9-hour average Ap before the current time.
- Average Ap from 12 to 33 hours before the current time, based on eight 3-hour average Ap values.
- Average Ap from 36 to 57 hours before the current time, based on eight 3-hour average Ap values.

**Notes**

No full description has been published of this model; it has been defined by its implementation only. It was written in FORTRAN, and is accessible at ftp://hanna.ccmc.gsfc.nasa.gov/pub/modelweb/atmospheric/msis/nrlmsise00/

A C port of the model by Dominik Brodowski has become popular, and is available on his website: http://www.brodo.de/space/nrlmsise/.

In 2013 Joshua Milas ported the C port to Python. This is an interface to his excellent port. It is a 1000-sloc model, and has been rigorously tested against the C version, and the online calculation tool available at [R4bc72d62ed0b-3] for parametric inputs of latitude, longitude, altitude, time of day and day of year.

This model is based on measurements other than gravity; it does not provide a calculation method for \(g\). It does not provide transport properties.

This model takes on the order of \(~2\) ms.

**References**

[R4bc72d62ed0b-1], [R4bc72d62ed0b-2], [R4bc72d62ed0b-3]

**Examples**

```python
>>> atmosphere = ATMOSPHERE_NRLMSISE00(1E3, 45, 45, 150)
>>> atmosphere.T, atmosphere.rho
(285.54408606237405, 1.1019062026405517)
```

**Attributes**

- **rho** [float] Mass density \([\text{kg/m}^3]\)
- **T** [float] Temperature, [K]
- **P** [float] Pressure, calculated with ideal gas law [Pa]

2.1. Atmospheric properties (fluids.atmosphere)
**He_density** [float] Density of helium atoms [count/m³]

**O_density** [float] Density of monatomic oxygen [count/m³]

**N2_density** [float] Density of nitrogen molecules [count/m³]

**O2_density** [float] Density of oxygen molecules [count/m³]

**Ar_density** [float] Density of Argon atoms [count/m³]

**H_density** [float] Density of hydrogen atoms [count/m³]

**N_density** [float] Density of monatomic nitrogen [count/m³]

**O_anomalous_density** [float] Density of anomalous oxygen; see [R4bc72d62ed0b-1] for details [count/m³]

**particle_density** [float] Total density of molecules [count/m³]

**components** [list[str]] List of species making up the atmosphere [-]

**zs** [list[float]] Mole fractions of each molecule in the atmosphere, in order of *components* [-]

\[
MWs = [28.0134, 31.9988, 39.948, 4.002602, 15.9994, 1.00794, 14.0067]
\]

\[
attrs = ['N2_density', 'O2_density', 'Ar_density', 'He_density', 'O_density', 'H_density', 'N_density']
\]

\[
components = ['N2', 'O2', 'Ar', 'He', 'O', 'H', 'N']
\]

**fluids.atmosphere.airmass (func, angle, H_max=86400.0, R_planet=6371229.0, RI=1.000276)**

Calculates mass of air per square meter in the atmosphere using a provided atmospheric model. The lowest air mass is calculated straight up; as the angle is lowered to nearer and nearer the horizon, the air mass increases, and can approach 40x or more the minimum airmass.

\[
m(\gamma) = \int_0^\infty \rho \left\{ 1 - [1 + 2(RI - 1)(1 - \rho/\rho_0)] \left[ \frac{\cos \gamma}{(1 + h/R)} \right]^2 \right\}^{-1/2} dH
\]

**Parameters**

**func** [float] Function which returns the density of the atmosphere as a function of elevation

**angle** [float] Degrees above the horizon (90 = straight up), [degrees]

**H_max** [float, optional] Maximum height to compute the integration up to before the contribution of density becomes negligible, [m]

**R_planet** [float, optional] The radius of the planet for which the integration is being performed, [m]

**RI** [float, optional] The refractive index of the atmosphere (air on earth at 0.7 um as default) assumed a constant, [-]

**Returns**

**m** [float] Mass of air per square meter in the atmosphere, [kg/m²]

**Notes**

Numerical integration via SciPy’s *quad* is used to perform the calculation.

**References**

[1]
2.1.2 Solar Radiation and Position

```python
>>> airmass(lambda Z : ATMOSPHERE_1976(Z).rho, 90)
10356.127665863998
```

Calculate the position of the sun in the sky. It is defined in terms of two angles - the zenith and the azimuth. The azimuth tells where a sundial would see the sun as coming from; the zenith tells how high in the sky it is. The solar elevation angle is returned for convinience; it is the complimentary angle of the zenith.

The sun’s refraction changes how high it appears as though the sun is; so values are returned with an optional conversion to the apparent angle. This impacts only the zenith/elevation.

Uses the Reda and Andreas (2004) model described in [1], originally incorporated into the excellent `pvlib` library.

**Parameters**

- `moment` [datetime] Time and date for the calculation, in local UTC time (not daylight savings time), [-]
- `latitude` [float] Latitude, between -90 and 90 [degrees]
- `longitude` [float] Longitude, between -180 and 180, [degrees]
- `Z` [float, optional] Elevation above sea level for the solar position calculation, [m]
- `T` [float, optional] Temperature of atmosphere at ground level, [K]
- `P` [float, optional] Pressure of atmosphere at ground level, [Pa]
- `atmos_refract` [float, optional] Atmospheric refractivity, [degrees]

**Returns**

- `apparent_zenith` [float] Zenith of the sun as observed from the ground based after accounting for atmospheric refraction, [degrees]
- `zenith` [float] Actual zenith of the sun (ignores atmospheric refraction), [degrees]
- `apparent_altitude` [float] Altitude of the sun as observed from the ground based after accounting for atmospheric refraction, [degrees]
- `altitude` [float] Actual altitude of the sun (ignores atmospheric refraction), [degrees]
- `azimuth` [float] The azimuth of the sun, [degrees]
- `equation_of_time` [float] Equation of time - the number of seconds to be added to the day’s mean solar time to obtain the apparent solar noon time, [seconds]

**Notes**

If you were standing at the same longitude of the sun such that it was no further east or west than you were, the amount of angle it was south or north of you is the **zenith**. If it were directly overhead it would be 0°; a little north or south and it would be a little positive; near sunset or sunrise, near 90°; and at night, between 90° and 180°.

The **solar altitude angle** is defined as 90° - zenith. Note the elevation angle is just another name for the altitude angle.
The *azimuth* the angle in degrees that the sun is East of the North angle. It is positive North eastwards 0° to 360°. Other conventions may be used.

Note that due to differences in atmospheric refractivity, estimation of sunset and sunrise are accurate to no more than one minute. Refraction conditions truly vary across the atmosphere; so characterizing it by an average value is limiting as well.

**References**

[1], [2]

**Examples**

```python
>>> solar_position(datetime(2003, 10, 17, 13, 30, 30), 45, 45)
[140.8367913391112, 140.8367913391112, -50.83679133911118, -50.83679133911118, 329.9096671679604, 878.4902950980904]
```

Sunrise occurs when the zenith is 90 degrees (Calgary, AB):

```python
>>> solar_position(datetime(2018, 4, 15, 6, 43, 5), 51.0486, -114.07)
[90.00054676987014]
```

Sunrise also occurs when the zenith is 90 degrees (13.5 hours later):

```python
>>> solar_position(datetime(2018, 4, 15, 20, 30, 28), 51.0486, -114.07)
[89.9995695661236, 90.54103812161853, 0.00043043387640950836, -0.5410381216185247, 286.8313781904518, 6.631429525878048]
```

```python
fluids.atmosphere.solar_irradiation(latitude, longitude, Z, moment, surface_tilt, surface_azimuth, T=None, P=None, solar_constant=1366.1, atmos_refract=0.5667, albedo=0.25, linke_turbidity=None, extraradiation_method='spencer', air_mass_model='kastenyoung1989', cache=None)
```

Calculates the amount of solar radiation and radiation reflected back the atmosphere which hits a surface at a specified tilt, and facing a specified azimuth.

This function is a wrapper for the incredibly comprehensive pvlib library, and requires it to be installed.

**Parameters**

- **latitude** [float] Latitude, between -90 and 90 [degrees]
- **longitude** [float] Longitude, between -180 and 180, [degrees]
- **Z** [float, optional] Elevation above sea level for the position, [m]
- **moment** [datetime] Time and date for the calculation, in local UTC time (not daylight savings time), [-]
- **surface_tilt** [float] The angle above the horizontal of the object being hit by radiation, [degrees]
- **surface_azimuth** [float] The angle the object is facing (positive North eastwards 0° to 360°), [degrees]
- **T** [float, optional] Temperature of atmosphere at ground level, [K]
- **P** [float, optional] Pressure of atmosphere at ground level, [Pa]
**solar_constant** [float, optional] The amount of solar radiation which reaches earth’s disk (at a standardized distance of 1 AU); this constant is independent of activity or conditions on earth, but will vary throughout the sun’s lifetime and may increase or decrease slightly due to solar activity, [W/m^2]

**atmos_refract** [float, optional] Atmospheric refractivity at sunrise/sunset (0.5667 deg is an often used value; this varies substantially and has an impact of a few minutes on when sunrise and sunset is), [degrees]

**albedo** [float, optional] The average amount of reflection of the terrain surrounding the object at quite a distance; this impacts how much sunlight reflected off the ground, gest reflected back off clouds, [-]

**linke_turbidity** [float, optional] The amount of pollution/water in the sky versus a perfect clear sky; If not specified, this will be retrieved from a historical grid; typical values are 3 for cloudy, and 7 for severe pollution around a city, [-]

**extraradiation_method** [str, optional] The specified method to calculate the effect of earth’s position on the amount of radiation which reaches earth according to the methods available in the *pvlib* library, [-]

**airmass_model** [str, optional] The specified method to calculate the amount of air the sunlight needs to travel through to reach the earth according to the methods available in the *pvlib* library, [-]

**cache** [dict, optional] Dictionary to to check for values to use to skip some calculations; *apparent_zenith*, *zenith*, *azimuth* supported, [-]

**Returns**

- **poa_global** [float] The total irradiance in the plane of the surface, [W/m^2]
- **poa_direct** [float] The total beam irradiance in the plane of the surface, [W/m^2]
- **poa_diffuse** [float] The total diffuse irradiance in the plane of the surface, [W/m^2]
- **poa_sky_diffuse** [float] The sky component of the diffuse irradiance, excluding the impact from the ground, [W/m^2]
- **poa_ground_diffuse** [float] The ground-sky diffuse irradiance component, [W/m^2]

**Notes**

The retrieval of **linke_turbidity** requires the pytables library (and Pandas); if it is not installed, specify a value of **linke_turbidity** to avoid the dependency.

There is some redundancy of the calculated results, according to the following relations. The total irradiance is normally that desired for engineering calculations.

\[
\text{poa_diffuse} = \text{poa_ground_diffuse} + \text{poa_sky_diffuse} \\
\text{poa_global} = \text{poa_direct} + \text{poa_diffuse}
\]

For a surface such as a pipe or vessel, an approach would be to split it into a number of rectangles and sum up the radiation absorbed by each.

This calculation is fairly slow.

**References**

[1]
Examples

```python
>>> solar_irradiation(Z=1100.0, latitude=51.0486, longitude=-114.07,
... moment=datetime(2018, 4, 15, 13, 43, 5), surface_tilt=41.0,
... surface_azimuth=180.0)
(1065.7621896280812, 945.2656564506323, 120.49653317744884, 95.31535344213178, 25.18117975317063)
```

```python
>>> cache = {'apparent_zenith': 41.099082295767545, 'zenith': 41.11285376417578,
˓→'azimuth': 182.5631874250523}
>>> solar_irradiation(Z=1100.0, latitude=51.0486, longitude=-114.07,
... moment=datetime(2018, 4, 15, 13, 43, 5), surface_tilt=41.0,
... surface_azimuth=180.0, cache=cache)
(1042.5677703677097, 918.2377548545295, 124.33001551318027, 99.6228657378363, 24.70714977534396)
```

At night, there is no solar radiation and this function returns zeros:

```python
>>> solar_irradiation(Z=1100.0, latitude=51.0486, longitude=-114.07,
... moment=datetime(2018, 4, 15, 2, 43, 5), surface_tilt=41.0,
... surface_azimuth=180.0)
(0.0, -0.0, 0.0, 0.0, 0.0)
```

```python
def fluids.atmosphere.sunrise_sunset(moment, latitude, longitude)
Calculates the times at which the sun is at sunset; sunrise; and halfway between sunrise and sunset (transit).
```

Uses the Reda and Andreas (2004) model described in [1], originally incorporated into the excellent pvlib library

Parameters

- **moment** [datetime] Date for the calculation; needs to contain only the year, month, and day, [-]
- **latitude** [float] Latitude, between -90 and 90 [degrees]
- **longitude** [float] Longitude, between -180 and 180, [degrees]

Returns

- **sunrise** [datetime] The time at the specified day when the sun rises **IN UTC**, [-]
- **sunset** [datetime] The time at the specified day when the sun sets **IN UTC**, [-]
- **transit** [datetime] The time at the specified day when the sun is at solar noon - halfway between sunrise and sunset **IN UTC**, [-]

Notes

This function takes on the order of 2 ms per calculation.

The reason the function cannot return the time correct the local timezone is that the function does not know the timezone at the specified lat/long.

References

[1]
Examples

```python
>>> sunrise, sunset, transit = sunrise_sunset(datetime(2018, 4, 17),
...   51.0486, -114.07)
>>> sunrise
datetime.datetime(2018, 4, 17, 12, 36, 55, 782660)
>>> sunset
datetime.datetime(2018, 4, 18, 2, 34, 4, 249326)
>>> transit
datetime.datetime(2018, 4, 17, 19, 35, 46, 686265)
```

```python
fluids.atmosphere.earthsun_distance(moment)
Calculates the distance between the earth and the sun as a function of date and time. Uses the Reda and Andreas
(2004) model described in [1], originally incorporated into the excellent pvlib library

Parameters

- **moment** [datetime] Time and date for the calculation, in UTC time (or GMT, which is almost
  the same thing); not local time, [-]

Returns

- **distance** [float] Distance between the center of the earth and the center of the sun, [m]

Notes

This function is quite accurate. The difference comes from the impact of the moon.
Note this function is not continuous; the sun-earth distance is not sufficiently accurately modeled for the change
to be continuous throughout each day.

References

[1]

Examples

```python
>>> earthsun_distance(datetime(2003, 10, 17, 13, 30, 30))
149090925951.18338
```
The distance at perihelion, which occurs at 4:21 according to this algorithm. The real value is 04:38 (January
2nd).

```python
>>> earthsun_distance(datetime(2013, 1, 2, 4, 21, 50))
147098089490.67123
```
The distance at aphelion, which occurs at 14:44 according to this algorithm. The real value is dead on - 14:44
(July 5).
2.1.3 Wind Models (requires Fortran compiler!)

```
fluids.atmosphere.hwm93(Z, latitude=0, longitude=0, day=0, seconds=0, f107=150.0,
f107_avg=150.0, geomagnetic_disturbance_index=4)
```

Horizontal Wind Model 1993, for calculating wind velocity in the atmosphere as a function of time of year, longitude and latitude, solar activity and earth’s geomagnetic disturbance.

The model is described across the publications [1], [2], and [3].

**Parameters**

- **Z** [float] Elevation, [m]
- **latitude** [float, optional] Latitude, between -90 and 90 [degrees]
- **longitude** [float, optional] Longitude, between -180 and 180 or 0 and 360, [degrees]
- **day** [float, optional] Day of year, 0-366 [day]
- **seconds** [float, optional] Seconds since start of day, in UT1 time; using UTC provides no loss in accuracy [s]
- **f107** [float, optional] Daily average 10.7 cm solar flux measurement of the strength of solar emissions on the 100 MHz band centered on 2800 MHz, averaged hourly; in sfu units, which are multiples of 10^-22 W/m^2/Hz; use 150 as a default [W/m^2/Hz]
- **f107_avg** [float, optional] 81-day sfu average; centered on specified day if possible, otherwise use the previous days [W/m^2/Hz]
- **geomagnetic_disturbance_index** [float, optional] Average daily Ap or also known as planetary magnetic index.

**Returns**

- **v_north** [float] Wind velocity, meridional (Northward) [m/s]
- **v_east** [float] Wind velocity, zonal (Eastward) [m/s]

**Notes**

No full description has been published of this model; it has been defined by its implementation only. It was written in FORTRAN, and is accessible at ftp://hanna.ccmc.gsfc.nasa.gov/pub/modelweb/atmospheric/hwm93/.

F2PY auto-compilation support is not yet currently supported. To compile this file, run the following command in a shell after navigating to $FLUIDSPATH/fluids/optional/.

```
f2py -c hwm93.pyf hwm93.for –f77flags=”-std=legacy”
```

If the module is not compiled, an import error will be raised.

**References**

[1], [2], [3]

**Examples**

```
>>> hwm93(5E5, 45, 50, 365)
(-73.00312042236328, 0.1485661268234253)
```
Horizontal Wind Model 2014, for calculating wind velocity in the atmosphere as a function of time of year, longitude and latitude, and earth’s geomagnetic disturbance. The model is described in [1].

The model no longer accounts for solar flux.

**Parameters**

- **Z** [float] Elevation, [m]
- **latitude** [float, optional] Latitude, between -90 and 90 [degrees]
- **longitude** [float, optional] Longitude, between -180 and 180 or 0 and 360, [degrees]
- **day** [float, optional] Day of year, 0-366 [day]
- **seconds** [float, optional] Seconds since start of day, in UT1 time; using UTC provides no loss in accuracy [s]
- **geomagnetic_disturbance_index** [float, optional] Average daily Ap or also known as planetary magnetic index.

**Returns**

- **v_north** [float] Wind velocity, meridional (Northward) [m/s]
- **v_east** [float] Wind velocity, zonal (Eastward) [m/s]

**Notes**

No full description has been published of this model; it has been defined by its implementation only. It was written in FORTRAN, and is accessible at [http://onlinelibrary.wiley.com/store/10.1002/2014EA000089/asset/supinfo/ess224-sup-0002-supinfo.tgz?v=1&s=2a957ba70b7cf9dd0612d9430076297c3634ea75](http://onlinelibrary.wiley.com/store/10.1002/2014EA000089/asset/supinfo/ess224-sup-0002-supinfo.tgz?v=1&s=2a957ba70b7cf9dd0612d9430076297c3634ea75).

F2PY auto-compilation support is not yet currently supported. To compile this file, run the following command in a shell after navigating to $FLUIDSPATH/fluids/optional/.

```
f2py -c hwm14.pyf hwm14.f90
```

The fortran .pyf signature file is included with this project, but it can also be re-created with the command:

```
f2py -m hwm14 -h hwm14.pyf hwm14.f90
```

If the module is not compiled, an import error will be raised.

No patches were necessary to either the generated pyf or hwm14.f90 file, as the authors of [1] have made it F2PY compatible.

Developed using 73 million data points taken by 44 instruments over 60 years.

**References**

[1]

**Examples**

```
>>> hwm14(5E5, 45, 50, 365)
(-38.64341354370117, 12.871272087097168)
```
### 2.2 Compressible flow and compressor sizing (fluids.compressible)

**fluids.compressible.Panhandle_A**

```
fluids.compressible.Panhandle_A(SG, Tavg, L=None, D=None, P1=None, P2=None, Q=None,
Ts=288.7, Ps=101325.0, Zavg=1, E=0.92)
```

Calculation function for dealing with flow of a compressible gas in a pipeline with the Panhandle A formula. Can calculate any of the following, given all other inputs:

- Flow rate
- Upstream pressure
- Downstream pressure
- Diameter of pipe
- Length of pipe

A variety of different constants and expressions have been presented for the Panhandle A equation. Here, a new form is developed with all units in base SI, based on the work of [1].

\[
Q = 158.02053E \left( \frac{T_s}{P_s} \right)^{1.0788} \left[ \frac{P_2^2 - P_1^2}{L \cdot SG^{0.8539} T_{avg} Z_{avg}} \right]^{0.5394} D^{2.6182}
\]

**Parameters**

- **SG** [float] Specific gravity of fluid with respect to air at the reference temperature and pressure 
  
  $T_s$ and $P_s$, [-]

- **Tavg** [float] Average temperature of the fluid in the pipeline, [K]

- **L** [float, optional] Length of pipe, [m]

- **D** [float, optional] Diameter of pipe, [m]

- **P1** [float, optional] Inlet pressure to pipe, [Pa]

- **P2** [float, optional] Outlet pressure from pipe, [Pa]

- **Q** [float, optional] Flow rate of gas through pipe, [m^3/s]

- **Ts** [float, optional] Reference temperature for the specific gravity of the gas, [K]

- **Ps** [float, optional] Reference pressure for the specific gravity of the gas, [Pa]

- **Zavg** [float, optional] Average compressibility factor for gas, [-]

- **E** [float, optional] Pipeline efficiency, a correction factor between 0 and 1

**Returns**

- **Q, P1, P2, D, or L** [float] The missing input which was solved for [base SI]

**Notes**

[1]'s original constant was 4.5965E-3, and it has units of km (length), kPa, mm (diameter), and flowrate in m^3/day.

The form in [2] has the same exponents as used here, units of mm (diameter), kPa, km (length), and flow in m^3/hour; its leading constant is 1.9152E-4.

The GPSA [3] has a leading constant of 0.191, a bracketed power of 0.5392, a specific gravity power of 0.853, and otherwise the same constants. It is in units of mm (diameter) and kPa and m^3/day; length is stated to be in km, but according to the errata is in m.
[4] has a leading constant of 1.198E7, a specific gravity of power of 0.8541, and a power of diameter which is under the root of 4.854 and is otherwise the same. It has units of kPa and m^3/day, but is otherwise in base SI units.

[5] has a leading constant of 99.5211, but its reference correction has no exponent; other exponents are the same as here. It is entirely in base SI units.

[6] has pressures in psi, diameter in inches, length in miles, Q in ft^3/day, T in degrees Rankine, and a constant of 435.87. Its reference condition power is 1.07881, and it has a specific gravity correction outside any other term with a power of 0.4604.

References

[1], [2], [3], [4], [5], [6]

Examples

```python
>>> Panhandle_A(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, Tavg=277.15)
42.56082051195928
```

**fluids.compressible.Panhandle_B(SG, Tavg, L=None, P1=None, P2=None, Q=None, Ts=288.7, Ps=101325.0, Zavg=1, E=0.92)**

Calculation function for dealing with flow of a compressible gas in a pipeline with the Panhandle B formula. Can calculate any of the following, given all other inputs:

- Flow rate
- Upstream pressure
- Downstream pressure
- Diameter of pipe
- Length of pipe

A variety of different constants and expressions have been presented for the Panhandle B equation. Here, a new form is developed with all units in base SI, based on the work of [1].

\[
Q = 152.88116E \left( \frac{T_s}{P_s} \right)^{1.02} \left[ \frac{P_1^2 - P_2^2}{L \cdot SG^{0.961} T_{avg} Z_{avg}} \right]^{0.51} D^{2.53}
\]

Parameters

- **SG** [float] Specific gravity of fluid with respect to air at the reference temperature and pressure
- **Tavg** [float] Average temperature of the fluid in the pipeline, [K]
- **L** [float, optional] Length of pipe, [m]
- **D** [float, optional] Diameter of pipe, [m]
- **P1** [float, optional] Inlet pressure to pipe, [Pa]
- **P2** [float, optional] Outlet pressure from pipe, [Pa]
- **Q** [float, optional] Flow rate of gas through pipe, [m^3/s]
- **Ts** [float, optional] Flow rate of gas through pipe, [m^3/s]
- **Ps** [float, optional] Reference temperature for the specific gravity of the gas, [K]
- **Ps** [float, optional] Reference pressure for the specific gravity of the gas, [Pa]
**Zavg** [float, optional] Average compressibility factor for gas, [-]

**E** [float, optional] Pipeline efficiency, a correction factor between 0 and 1

**Returns**

Q, P1, P2, D, or L [float] The missing input which was solved for [base SI]

**Notes**

[1]’s original constant was 1.002E-2, and it has units of km (length), kPa, mm (diameter), and flowrate in m³/day.

The form in [2] has the same exponents as used here, units of mm (diameter), kPa, km (length), and flow in m³/hour; its leading constant is 4.1749E-4.

The GPSA [3] has a leading constant of 0.339, and otherwise the same constants. It is in units of mm (diameter) and kPa and m³/day; length is stated to be in km, but according to the errata is in m.

[4] has a leading constant of 1.264E7, a diameter power of 4.961 which is also under the 0.51 power, and is otherwise the same. It has units of kPa and m³/day, but is otherwise in base SI units.

[5] has a leading constant of 135.8699, but its reference correction has no exponent and its specific gravity has a power of 0.9608; the other exponents are the same as here. It is entirely in base SI units.

[6] has pressures in psi, diameter in inches, length in miles, Q in ft³/day, T in degrees Rankine, and a constant of 737 with the exponents the same as here.

**References**

[1], [2], [3], [4], [5], [6]

**Examples**

```
>>> Panhandle_B(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, Tavg=277.15)
42.3566178004172
```

**fluids.compressible.Weymouth** *(SG, Tavg, L=None, D=None, P1=None, P2=None, Q=None, Ts=288.7, Ps=101325.0, Zavg=1, E=0.92)*

Calculation function for dealing with flow of a compressible gas in a pipeline with the Weymouth formula. Can calculate any of the following, given all other inputs:

- Flow rate
- Upstream pressure
- Downstream pressure
- Diameter of pipe
- Length of pipe

A variety of different constants and expressions have been presented for the Weymouth equation. Here, a new form is developed with all units in base SI, based on the work of [1].

\[
Q = 137.32958 E \frac{T_s}{P_s} \left[ \frac{P_1^2 - P_2^2}{L \cdot SG \cdot T_{avg} Z_{avg}} \right]^{0.5} D^{2.667}
\]

**Parameters**
SG  [float] Specific gravity of fluid with respect to air at the reference temperature and pressure $Ts$ and $Ps$, [-]

$T_{avg}$  [float] Average temperature of the fluid in the pipeline, [K]

$L$  [float, optional] Length of pipe, [m]

$D$  [float, optional] Diameter of pipe, [m]

$P_1$  [float, optional] Inlet pressure to pipe, [Pa]

$P_2$  [float, optional] Outlet pressure from pipe, [Pa]

$Q$  [float, optional] Flow rate of gas through pipe, [m$^3$/s]

$T_s$  [float, optional] Reference temperature for the specific gravity of the gas, [K]

$P_s$  [float, optional] Reference pressure for the specific gravity of the gas, [Pa]

$Z_{avg}$  [float, optional] Average compressibility factor for gas, [-]

$E$  [float, optional] Pipeline efficiency, a correction factor between 0 and 1

Returns

$Q, P_1, P_2, D, \text{ or } L$  [float] The missing input which was solved for [base SI]

Notes

[1]’s original constant was $3.7435E-3$, and it has units of km (length), kPa, mm (diameter), and flowrate in m$^3$/day.

The form in [2] has the same exponents as used here, units of mm (diameter), kPa, km (length), and flow in m$^3$/hour; its leading constant is $1.5598E-4$.

The GPSA [3] has a leading constant of $0.1182$, and otherwise the same constants. It is in units of mm (diameter) and kPa and m$^3$/day; length is stated to be in km, but according to the errata is in m.

[4] has a leading constant of $1.162E7$, a diameter power of $5.333$ which is also under the $0.50$ power, and is otherwise the same. It has units of kPa and m$^3$/day, but is otherwise in base SI units.

[5] has a leading constant of $137.2364$; the other exponents are the same as here. It is entirely in base SI units.

[6] has pressures in psi, diameter in inches, length in miles, $Q$ in ft$^3$/hour, $T$ in degrees Rankine, and a constant of $18.062$ with the exponents the same as here.

References

[1], [2], [3], [4], [5], [6]

Examples

```python
>>> Weymouth(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, Tavg=277.15)
32.07729055913029
```

fluids.compressible.Spitzglass_high(SG, Tavg, L=None, D=None, P1=None, P2=None, Q=None, Ts=288.7, Ps=101325.0, Zavg=1, E=1.0)

Calculation function for dealing with flow of a compressible gas in a pipeline with the Spitzglass (high pressure drop) formula. Can calculate any of the following, given all other inputs:

- Flow rate
• Upstream pressure
• Downstream pressure
• Diameter of pipe (numerical solution)
• Length of pipe

A variety of different constants and expressions have been presented for the Spitzglass (high pressure drop) formula. Here, the form as in [1] is used but with a more precise metric conversion from inches to m.

\[ Q = 125.1060E \left( \frac{T_s}{P_s} \right) \left[ \frac{P_1^2 - P_2^2}{L \cdot SG \cdot T_{avg} \cdot Z_{avg} \cdot (1 + 0.09144/D + \frac{150}{127}D)} \right]^{0.5} D^{2.5} \]

Parameters

**SG** [float] Specific gravity of fluid with respect to air at the reference temperature and pressure $T_s$ and $P_s$, [-]

**Tavg** [float] Average temperature of the fluid in the pipeline, [K]

**L** [float, optional] Length of pipe, [m]

**D** [float, optional] Diameter of pipe, [m]

**P1** [float, optional] Inlet pressure to pipe, [Pa]

**P2** [float, optional] Outlet pressure from pipe, [Pa]

**Q** [float, optional] Flow rate of gas through pipe, [m^3/s]

**Ts** [float, optional] Reference temperature for the specific gravity of the gas, [K]

**Ps** [float, optional] Reference pressure for the specific gravity of the gas, [Pa]

**Zavg** [float, optional] Average compressibility factor for gas, [-]

**E** [float, optional] Pipeline efficiency, a correction factor between 0 and 1

Returns

**Q, P1, P2, D, or L** [float] The missing input which was solved for [base SI]

Notes

This equation is often presented without any correction for reference conditions for specific gravity.

This model is also presented in [2] with a leading constant of 1.0815E-2, the same exponents as used here, units of mm (diameter), kPa, km (length), and flow in m^3/hour.

References

[1], [2]

Examples

```python
>>> Spitzglass_high(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, Tavg=277.15)
29.42670246281681
```
fluids.compressible.Spitzglass_low(SG, Tavg, L=None, D=None, P1=None, P2=None, Q=None, Ts=288.7, Ps=101325.0, Zavg=1, E=1.0)

Calculation function for dealing with flow of a compressible gas in a pipeline with the Spitzglass (low pressure drop) formula. Can calculate any of the following, given all other inputs:

- Flow rate
- Upstream pressure
- Downstream pressure
- Diameter of pipe (numerical solution)
- Length of pipe

A variety of different constants and expressions have been presented for the Spitzglass (low pressure drop) formula. Here, the form as in [1] is used but with a more precise metric conversion from inches to m.

\[
Q = 125.1060E \left( \frac{T_s}{P_s} \right) \left[ \frac{2(P_1 - P_2)(P_s + 1210)}{L \cdot SG \cdot T_{avg}Z_{avg}(1 + 0.09144/D + \frac{150}{127}D)} \right]^{0.5} D^{2.5}
\]

**Parameters**

- **SG** [float] Specific gravity of fluid with respect to air at the reference temperature and pressure
- **Tavg** [float] Average temperature of the fluid in the pipeline, [K]
- **L** [float, optional] Length of pipe, [m]
- **D** [float, optional] Diameter of pipe, [m]
- **P1** [float, optional] Inlet pressure to pipe, [Pa]
- **P2** [float, optional] Outlet pressure from pipe, [Pa]
- **Q** [float, optional] Flow rate of gas through pipe, [m^3/s]
- **Ts** [float, optional] Reference temperature for the specific gravity of the gas, [K]
- **Ps** [float, optional] Reference pressure for the specific gravity of the gas, [Pa]
- **Zavg** [float, optional] Average compressibility factor for gas, [-]
- **E** [float, optional] Pipeline efficiency, a correction factor between 0 and 1

**Returns**

- **Q, P1, P2, D, or L** [float] The missing input which was solved for [base SI]

**Notes**

This equation is often presented without any correction for reference conditions for specific gravity.

This model is also presented in [2] with a leading constant of 5.69E-2, the same exponents as used here, units of mm (diameter), kPa, km (length), and flow in m^3/hour. However, it is believed to contain a typo, and gives results <1/3 of the correct values. It is also present in [2] in imperial form; this is believed correct, but makes a slight assumption not done in [1].

This model is present in [3] without reference corrections. The 1210 constant in [1] is an approximation necessary for the reference correction to function without a square of the pressure difference. The GPSA version is as follows, and matches this formulation very closely:

\[
Q = 0.821 \left[ \frac{(P_1 - P_2)D^5}{L \cdot SG(1 + 91.44/D + 0.0018D)} \right]^{0.5}
\]
The model is also shown in [4], with diameter in inches, length in feet, flow in MMSCFD, pressure drop in inH2O, and a rounded leading constant of 0.09; this makes its predictions several percent higher than the model here.

References

[1], [2], [3], [4]

Examples

```python
>>> Spitzglass_low(D=0.154051, P1=6720.3199, P2=0, L=54.864, SG=0.6, Tavg=288.7)
0.9488775242530617
```

`fluids.compressible.Oliphant(SG, Tavg, L=None, D=None, P1=None, P2=None, Q=None, Ts=288.7, Ps=101325.0, Zavg=1, E=0.92)`

Calculation function for dealing with flow of a compressible gas in a pipeline with the Oliphant formula. Can calculate any of the following, given all other inputs:

- Flow rate
- Upstream pressure
- Downstream pressure
- Diameter of pipe (numerical solution)
- Length of pipe

This model is a more complete conversion to metric of the Imperial version presented in [1].

\[ Q = 84.5872 \left( D^{2.5} + 0.20915D^3 \right) \frac{T_s}{P_s} \left( \frac{P_2^2 - P_1^2}{L \cdot SG \cdot T_{avg}} \right)^{0.5} \]

Parameters

- **SG** [float] Specific gravity of fluid with respect to air at the reference temperature and pressure \( T_s \) and \( P_s \), [-]
- **Tavg** [float] Average temperature of the fluid in the pipeline, [K]
- **L** [float, optional] Length of pipe, [m]
- **D** [float, optional] Diameter of pipe, [m]
- **P1** [float, optional] Inlet pressure to pipe, [Pa]
- **P2** [float, optional] Outlet pressure from pipe, [Pa]
- **Q** [float, optional] Flow rate of gas through pipe, [m³/s]
- **Ts** [float, optional] Reference temperature for the specific gravity of the gas, [K]
- **Ps** [float, optional] Reference pressure for the specific gravity of the gas, [Pa]
- **Zavg** [float, optional] Average compressibility factor for gas, [-]
- **E** [float, optional] Pipeline efficiency, a correction factor between 0 and 1

Returns

- **Q, P1, P2, D, or L** [float] The missing input which was solved for [base SI]
Notes

Recommended in [1] for use between vacuum and 100 psi.

The model is simplified by grouping constants here; however, it is presented in the imperial unit set inches (diameter), miles (length), psi, Rankine, and MMSCFD in [1]:

\[
Q = 42(24) \left( D^{2.5} + \frac{D^3}{30} \right) \left( \frac{14.4}{P_s} \right) \left( \frac{T_s}{520} \right) \left( \frac{0.6}{SG} \right) \left( \frac{520}{T_{avg}} \right) \left( \frac{P_1^2 - P_2^2}{L} \right)^0.5
\]

References

[1], [2]

Examples

```python
>>> Oliphant(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, Tavg=277.15)
28.851535408143057
```

`fluids.compressible.Fritzsche(SG, Tavg, L=None, D=None, P1=None, P2=None, Q=None, Ts=288.7, Ps=101325.0, Zavg=1, E=1)`

Calculation function for dealing with flow of a compressible gas in a pipeline with the Fritzsche formula. Can calculate any of the following, given all other inputs:

- Flow rate
- Upstream pressure
- Downstream pressure
- Diameter of pipe
- Length of pipe

A variety of different constants and expressions have been presented for the Fritzsche formula. Here, the form as in [1] is used but with all inputs in base SI units:

\[
Q = 93.500 \frac{T_s}{P_s} \left( \frac{P_1^2 - P_2^2}{L \cdot SG^{0.8587} \cdot T_{avg}} \right)^{0.538} D^{2.69}
\]

Parameters

- SG [float] Specific gravity of fluid with respect to air at the reference temperature and pressure $T_s$ and $P_s$, [-]
- Tavg [float] Average temperature of the fluid in the pipeline, [K]
- L [float, optional] Length of pipe, [m]
- D [float, optional] Diameter of pipe, [m]
- P1 [float, optional] Inlet pressure to pipe, [Pa]
- P2 [float, optional] Outlet pressure from pipe, [Pa]
- Q [float, optional] Flow rate of gas through pipe, [m^3/s]
- Ts [float, optional] Reference temperature for the specific gravity of the gas, [K]
- Ps [float, optional] Reference pressure for the specific gravity of the gas, [Pa]
Zavg [float, optional] Average compressibility factor for gas, [-]
E [float, optional] Pipeline efficiency, a correction factor between 0 and 1

Returns
Q, P1, P2, D, or L [float] The missing input which was solved for [base SI]

Notes
This model is also presented in [1] with a leading constant of 2.827, the same exponents as used here, units of mm (diameter), kPa, km (length), and flow in m^3/hour.
This model is shown in base SI units in [2], and with a leading constant of 94.2565, a diameter power of 2.6911, main group power of 0.5382 and a specific gravity power of 0.858. The difference is very small.

References
[1], [2]

Examples

```python
>>> Fritzsche(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, Tavg=277.15)
39.421535157535565
```

```
fluids.compressible.Muller(SG, Tavg, mu, L=None, D=None, P1=None, P2=None, Q=None, Ts=288.7, Ps=101325.0, Zavg=1, E=1)
```

Calculation function for dealing with flow of a compressible gas in a pipeline with the Muller formula. Can calculate any of the following, given all other inputs:
- Flow rate
- Upstream pressure
- Downstream pressure
- Diameter of pipe
- Length of pipe

A variety of different constants and expressions have been presented for the Muller formula. Here, the form as in [1] is used but with all inputs in base SI units.

\[
Q = 15.7743 \frac{T_s}{P_s} E \left( \frac{P_1^2 - P_2^2}{L \cdot Z_{\text{avg}} \cdot T_{\text{avg}}} \right)^{0.575} \left( \frac{D^{2.725}}{\mu^{0.15} SG^{0.425}} \right)
\]

Parameters

- **SG** [float] Specific gravity of fluid with respect to air at the reference temperature and pressure Ts and Ps, [-]
- **Tavg** [float] Average temperature of the fluid in the pipeline, [K]
- **mu** [float] Average viscosity of the fluid in the pipeline, [Pa*s]
- **L** [float, optional] Length of pipe, [m]
- **D** [float, optional] Diameter of pipe, [m]
- **P1** [float, optional] Inlet pressure to pipe, [Pa]
P2 [float, optional] Outlet pressure from pipe, [Pa]
Q [float, optional] Flow rate of gas through pipe, [m^3/s]
Ts [float, optional] Reference temperature for the specific gravity of the gas, [K]
Ps [float, optional] Reference pressure for the specific gravity of the gas, [Pa]
Zavg [float, optional] Average compressibility factor for gas, [-]
E [float, optional] Pipeline efficiency, a correction factor between 0 and 1

Returns
Q, P1, P2, D, or L [float] The missing input which was solved for [base SI]

Notes
This model is presented in [1] with a leading constant of 0.4937, the same exponents as used here, units of inches (diameter), psi, feet (length), Rankine, pound/(foot*second) for viscosity, and 1000 ft^3/hour.

This model is also presented in [2] in both SI and imperial form. The SI form was incorrectly converted and yields much higher flow rates. The imperial version has a leading constant of 85.7368, the same powers as used here except with rounded values of powers of viscosity (0.2609) and specific gravity (0.7391) rearranged to be inside the bracketed group; its units are inches (diameter), psi, miles (length), Rankine, pound/(foot*second) for viscosity, and ft^3/day.

This model is shown in base SI units in [3], and with a leading constant of 15.7650, a diameter power of 2.724, main group power of 0.5747, a specific gravity power of 0.74, and a viscosity power of 0.1494.

References
[1], [2], [3]

Examples
>>> Muller(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, mu=1E-5, ... Tavg=277.15)
60.45796698148659

fluids.compressible.IGT (SG, Tavg, mu, L=None, D=None, P1=None, P2=None, Q=None, Ts=288.7, Ps=101325.0, Zavg=1, E=1)
Calculation function for dealing with flow of a compressible gas in a pipeline with the IGT formula. Can calculate any of the following, given all other inputs:
- Flow rate
- Upstream pressure
- Downstream pressure
- Diameter of pipe
- Length of pipe

A variety of different constants and expressions have been presented for the IGT formula. Here, the form as in [1] is used but with all inputs in base SI units.

\[
Q = 24.6241 \frac{T_s}{P_s} E \left( \frac{P_1^2 - P_2^2}{L \cdot Z_{avg} \cdot T_{avg}} \right)^{5/9} \left( \frac{D^{8/3}}{\mu^{1/9} SG^{4/9}} \right)
\]
Parameters

\begin{itemize}
  \item **SG** [float] Specific gravity of fluid with respect to air at the reference temperature and pressure \(Ts\) and \(Ps\), [-]
  \item **Tavg** [float] Average temperature of the fluid in the pipeline, [K]
  \item **mu** [float] Average viscosity of the fluid in the pipeline, [Pa*s]
  \item **L** [float, optional] Length of pipe, [m]
  \item **D** [float, optional] Diameter of pipe, [m]
  \item **P1** [float, optional] Inlet pressure to pipe, [Pa]
  \item **P2** [float, optional] Outlet pressure from pipe, [Pa]
  \item **Q** [float, optional] Flow rate of gas through pipe, [m^3/s]
  \item **Ts** [float, optional] Reference temperature for the specific gravity of the gas, [K]
  \item **Ps** [float, optional] Reference pressure for the specific gravity of the gas, [Pa]
  \item **Zavg** [float, optional] Average compressibility factor for gas, [-]
  \item **E** [float, optional] Pipeline efficiency, a correction factor between 0 and 1
\end{itemize}

Returns

\begin{itemize}
  \item **Q, P1, P2, D, or L** [float] The missing input which was solved for [base SI]
\end{itemize}

Notes

This model is presented in [1] with a leading constant of 0.6643, the same exponents as used here, units of inches (diameter), psi, feet (length), Rankine, pound/(foot*second) for viscosity, and 1000 ft^3/hour.

This model is also presented in [2] in both SI and imperial form. Both forms are correct. The imperial version has a leading constant of 136.9, the same powers as used here except with rounded values of powers of viscosity (0.2) and specific gravity (0.8) rearranged to be inside the bracketed group; its units are inches (diameter), psi, miles (length), Rankine, pound/(foot*second) for viscosity, and ft^3/day.

This model is shown in base SI units in [3], and with a leading constant of 24.6145, and the same powers as used here.

References

[1], [2], [3]

Examples

\begin{verbatim}
>>> IGT(D=0.340, P1=90E5, P2=20E5, L=160E3, SG=0.693, mu=1E-5, Tavg=277.15)
48.92351786788815
\end{verbatim}

\begin{verbatim}
fluids.compressible.isothermal_gas(rho, fd, P1=None, P2=None, L=None, D=None, m=None)
\end{verbatim}

Calculation function for dealing with flow of a compressible gas in a pipeline for the complete isothermal flow equation. Can calculate any of the following, given all other inputs:

- Mass flow rate
- Upstream pressure (numerical)
• Downstream pressure (analytical or numerical if an overflow occurs)
• Diameter of pipe (numerical)
• Length of pipe

A variety of forms of this equation have been presented, differing in their use of the ideal gas law and choice of gas constant. The form here uses density explicitly, allowing for non-ideal values to be used.

\[ \dot{m}^2 = \frac{\left( \frac{\pi D^2}{4} \right)^2 \rho_{avg} (P_1^2 - P_2^2)}{P_1 \left( f_d \frac{L}{D} \right) + 2 \ln \frac{P_1}{P_2}} \]

Parameters

\[ \text{rho} \] [float] Average density of gas in pipe, [kg/m^3]
\[ \text{fd} \] [float] Darcy friction factor for flow in pipe [-]
\[ P_1 \] [float, optional] Inlet pressure to pipe, [Pa]
\[ P_2 \] [float, optional] Outlet pressure from pipe, [Pa]
\[ L \] [float, optional] Length of pipe, [m]
\[ D \] [float, optional] Diameter of pipe, [m]
\[ \text{m} \] [float, optional] Mass flow rate of gas through pipe, [kg/s]

Returns

\[ \text{m, P1, P2, D, or L} \] [float] The missing input which was solved for [base SI]

Notes

The solution for \( P_2 \) has the following closed form, derived using Maple:

\[ P_2 = P_1 \left( 0.5 \cdot \frac{1}{m^2} \left( -Cm^2 + \text{lambertW} \left( -\frac{BP_1}{m^2} e^{\frac{-Cm^2 + BP_1}{m^2}} \right) m^2 + BP_1 \right) \right)^{-1} \]

\[ B = \frac{\pi^2 D^4}{4} \rho_{avg} \]
\[ C = f_d \frac{L}{D} \]

A wide range of conditions are impossible due to choked flow. See \textit{P_isothermal_critical_flow} for details. An exception is raised when they occur.

The 2 multiplied by the logarithm is often shown as a power of the pressure ratio; this is only the case when the pressure ratio is raised to the power of 2 before its logarithm is taken.

A number of limitations exist for this model:

• Density dependence is that of an ideal gas.
• If calculating the pressure drop, the average gas density cannot be known immediately; iteration must be used to correct this.
• The friction factor depends on both the gas density and velocity, so it should be solved for iteratively as well. It changes throughout the pipe as the gas expands and velocity increases.
• The model is not easily adapted to include elevation effects due to the acceleration term included in it.
• As the gas expands, it will change temperature slightly, further altering the density and friction factor.

There are many commercial packages which perform the actual direct integration of the flow, such as OLGA Dynamic Multiphase Flow Simulator, or ASPEN Hydraulics.

This expression has also been presented with the ideal gas assumption directly incorporated into it [4] (note R is the specific gas constant, in units of J/kg/K):

\[
\dot{m}^2 = \frac{(\frac{\pi D^2}{4})^2 (P_1^2 - P_2^2)}{RT \left( f_d \frac{P_1}{P} + 2 \ln \frac{P_1}{P_2} \right)}
\]

References
[1], [2], [3], [4]

Examples

```python
>>> isothermal_gas(rho=11.3, fd=0.00185, P1=1E6, P2=9E5, L=1000, D=0.5)
145.4847572636031
```

`fluids.compressible.isothermal_work_compression(P1, P2, T, Z=1)`

Calculates the work of compression or expansion of a gas going through an isothermal process.

\[
W = zRT \ln \left( \frac{P_2}{P_1} \right)
\]

Parameters

- **P1** [float] Inlet pressure, [Pa]
- **P2** [float] Outlet pressure, [Pa]
- **T** [float] Temperature of the gas going through an isothermal process, [K]
- **Z** [float] Constant compressibility factor of the gas, [-]

Returns

- **W** [float] Work performed per mole of gas compressed/expanded [J/mol]

Notes

The full derivation with all forms is as follows:

\[
W = \int_{P_1}^{P_2} V dP = zRT \int_{P_1}^{P_2} \frac{1}{P} dP
\]

\[
W = zRT \ln \left( \frac{P_2}{P_1} \right) = P_1 V_1 \ln \left( \frac{P_2}{P_1} \right) = P_2 V_2 \ln \left( \frac{P_2}{P_1} \right)
\]

The substitutions are according to the ideal gas law with compressibility:

The work of compression/expansion is the change in enthalpy of the gas. Returns negative values for expansion and positive values for compression.

An average compressibility factor can be used where Z changes. For further accuracy, this expression can be used repeatedly with small changes in pressure and the work from each step summed.
This is the best possible case for compression; all actual compressors require more work to do the compression. By making the compression take a large number of stages and cooling the gas between stages, this can be approached reasonable closely. Integrally geared compressors are often used for this purpose.

References

[1]

Examples

```python
gas = 1E5, 1E6, 300
5743.427304244769
```

```python
fluids.compressible.polytropic_exponent(k=1.4, eta_p=0.78)
1.5780346820809246
```

**fluids.compressible.polytropic_exponent** *(k, n=None, eta_p=None)*

Calculates one of:

- Polytropic exponent from polytropic efficiency
- Polytropic efficiency from the polytropic exponent

\[
\eta_p = \left( \frac{n}{k} \right) = \frac{n(k - 1)}{k(n - 1)}
\]

Parameters

- **k** [float] Isentropic exponent of the gas (Cp/Cv) [-]
- **n** [float, optional] Polytropic exponent of the process [-]
- **eta_p** [float, optional] Polytropic efficiency of the process, [-]

Returns

- **n** or **eta_p** [float] Polytropic exponent or polytropic efficiency, depending on input, [-]

References

[1]

Examples

```python
>>> polytropic_exponent(l.4, eta_p=0.78)
1.5780346820809246
```

**fluids.compressible.isentropic_work_compression** *(T1, k, Z=1, P1=None, P2=None, W=None, eta=None)*

Calculation function for dealing with compressing or expanding a gas going through an isentropic, adiabatic process assuming constant Cp and Cv. The polytropic model is the same equation; just provide \( n \) instead of \( k \) and use a polytropic efficiency for \( \eta \) instead of a isentropic efficiency. Can calculate any of the following, given all the other inputs:

- W, Work of compression
• P2, Pressure after compression
• P1, Pressure before compression
• \( \eta \), isentropic efficiency of compression

\[
W = \left( \frac{k}{k-1} \right) ZRT_1 \left[ \left( \frac{P_2}{P_1} \right)^{(k-1)/k} - 1 \right] / \eta_{\text{isentropic}}
\]

Parameters

T1 [float] Initial temperature of the gas, [K]
k [float] Isentropic exponent of the gas (\( \text{Cp}/\text{Cv} \)) or polytropic exponent \( n \) to use this as a polytropic model instead [-]
Z [float, optional] Constant compressibility factor of the gas, [-]
P1 [float, optional] Inlet pressure, [Pa]
P2 [float, optional] Outlet pressure, [Pa]
W [float, optional] Work performed per mole of gas compressed/expanded [J/mol]
eta [float, optional] Isentropic efficiency of the process or polytropic efficiency of the process to use this as a polytropic model instead [-]

Returns

W, P1, P2, or eta [float] The missing input which was solved for [base SI]

Notes

For the same compression ratio, this is always of larger magnitude than the isothermal case.
The full derivation is as follows:
For constant-heat capacity “isentropic” fluid,

\[
V = \frac{P_1^{1/k} V_1}{P_1^{1/k}}
\]

\[
W = \int_{P_1}^{P_2} V dP = \int_{P_1}^{P_2} \frac{P_1^{1/k} V_1}{P_1^{1/k}} dP
\]

\[
W = \frac{P_1^{1/k} V_1}{1 - \frac{1}{k}} \left[ P_2^{1-1/k} - P_1^{1-1/k} \right]
\]

After performing the integration and substantial mathematical manipulation we can obtain:

\[
W = \left( \frac{k}{k-1} \right) P_1 V_1 \left[ \left( \frac{P_2}{P_1} \right)^{(k-1)/k} - 1 \right]
\]

Using \( PV = ZRT \):

\[
W = \left( \frac{k}{k-1} \right) ZRT_1 \left[ \left( \frac{P_2}{P_1} \right)^{(k-1)/k} - 1 \right]
\]

The work of compression/expansion is the change in enthalpy of the gas. Returns negative values for expansion and positive values for compression.
An average compressibility factor should be used as Z changes. For further accuracy, this expression can be used repeatedly with small changes in pressure and new values of isentropic exponent, and the work from each step summed.

For the polytropic case this is not necessary, as \( \eta \) corrects for the simplification.

**References**

[1]

**Examples**

```python
>>> isentropic_work_compression(P1=1E5, P2=1E6, T1=300, k=1.4, eta=0.78)
10416.876986384483
```

`fluids.compressible.isentropic_efficiency(P1, P2, k, eta_s=None, eta_p=None)`

Calculates either isentropic or polytropic efficiency from the other type of efficiency.

\[
\eta_s = \frac{(P_2/P_1)^{(k-1)/k} - 1}{(P_2/P_1)^{\frac{k-1}{\eta_p}} - 1}
\]

\[
\eta_p = \frac{(k - 1) \log \left( \frac{P_2}{P_1} \right)}{k \log \left( \frac{1}{\eta_s} \left( \frac{P_2}{P_1} \frac{1}{\eta_s} \left( \frac{P_2}{P_1} \right)^{\frac{1}{k-1}} - 1 \right) \right)}
\]

**Parameters**

- `P1` [float] Initial pressure of gas [Pa]
- `P2` [float] Final pressure of gas [Pa]
- `k` [float] Isentropic exponent of the gas (Cp/Cv) [-]
- `eta_s` [float, optional] Isentropic (adiabatic) efficiency of the process, [-]
- `eta_p` [float, optional] Polytropic efficiency of the process, [-]

**Returns**

- `eta_s` or `eta_p` [float] Isentropic or polytropic efficiency, depending on input, [-]

**Notes**

The form for obtained \( \eta_p \) from \( \eta_s \) was derived with SymPy.

**References**

[1]
Examples

```python
>>> isentropic_efficiency(1E5, 1E6, 1.4, eta_p=0.78)
0.7027614191263858
```

`fluids.compressible.isentropic_T_rise_compression(T1, P1, P2, k, eta=1)`
Calculates the increase in temperature of a fluid which is compressed or expanded under isentropic, adiabatic conditions assuming constant Cp and Cv. The polytropic model is the same equation; just provide \( n \) instead of \( k \) and use a polytropic efficiency for \( \eta \) instead of a isentropic efficiency.

\[
T_2 = T_1 + \frac{\Delta T_s}{\eta_s} = T_1 \left\{ 1 + \frac{1}{\eta_s} \left[ \left( \frac{P_2}{P_1} \right)^{(k-1)/k} - 1 \right] \right\}
\]

**Parameters**
- **T1** [float] Initial temperature of gas [K]
- **P1** [float] Initial pressure of gas [Pa]
- **P2** [float] Final pressure of gas [Pa]
- **k** [float] Isentropic exponent of the gas (Cp/Cv) or polytropic exponent \( n \) to use this as a polytropic model instead [-]
- **eta** [float] Isentropic efficiency of the process or polytropic efficiency of the process to use this as a polytropic model instead [-]

**Returns**
- **T2** [float] Final temperature of gas [K]

**Notes**
For the ideal case (\('\eta'=1\)), the model simplifies to:

\[
\frac{T_2}{T_1} = \left( \frac{P_2}{P_1} \right)^{(k-1)/k}
\]

**References**
[1], [2]

**Examples**

```python
>>> isentropic_T_rise_compression(286.8, 54050, 432400, 1.4)
519.5230938217768
```

`fluids.compressible.T_critical_flow(T, k)`
Calculates critical flow temperature \( T_{cf} \) for a fluid with the given isentropic coefficient. \( T_{cf} \) is in a flow (with \( Ma=1 \)) whose stagnation conditions are known. Normally used with converging/diverging nozzles.

\[
\frac{T^*}{T_0} = \frac{2}{k + 1}
\]

**Parameters**
Fluids Documentation, Release 0.1

T  [float] Stagnation temperature of a fluid with Ma=1 [K]
k  [float] Isentropic coefficient []

Returns
Tcf  [float] Critical flow temperature at Ma=1 [K]

Notes
Assumes isentropic flow.

References
[1]

Examples
Example 12.4 in [1]:

```python
>>> T_critical_flow(473, 1.289)
413.2809086937528
```

fluids.compressible.P_critical_flow(P, k)
Calculates critical flow pressure Pcf for a fluid with the given isentropic coefficient. Pcf is in a flow (with Ma=1) whose stagnation conditions are known. Normally used with converging/diverging nozzles.

\[
\frac{P^*}{P_0} = \left( \frac{2}{k + 1} \right)^{k/(k-1)}
\]

Parameters
P  [float] Stagnation pressure of a fluid with Ma=1 [Pa]
k  [float] Isentropic coefficient []

Returns
Pcf  [float] Critical flow pressure at Ma=1 [Pa]

Notes
Assumes isentropic flow.

References
[1]

Examples
Example 12.4 in [1]:

```python
>>> P_critical_flow(1400000, 1.289)
766812.9022792266
```
fluids.compressible.P_isothermal_critical_flow(P, fd, D, L)
Calculates critical flow pressure Pcf for a fluid flowing isothermally and suffering pressure drop caused by a
pipe’s friction factor.

\[ P_2 = P_1 e^{\frac{D}{L} \left( \text{LambertW} \left( \frac{1}{e^{D-LfD}} \right) + 1 \right)} + LfD \]

**Parameters**
- P [float] Inlet pressure [Pa]
- fd [float] Darcy friction factor for flow in pipe [-]
- D [float] Diameter of pipe, [m]
- L [float] Length of pipe, [m]

**Returns**
- Pcf [float] Critical flow pressure of a compressible gas flowing from P1 to Pcf in a tube of
  length L and friction factor fd [Pa]

**Notes**
Assumes isothermal flow. Developed based on the isothermal_gas model, using SymPy.
The isothermal gas model is solved for maximum mass flow rate; any pressure drop under it is impossible due
to the formation of a shock wave.

**References**
[1]

**Examples**

```python
>>> P_isothermal_critical_flow(P=1E6, fd=0.00185, L=1000., D=0.5)
389699.7317645518
```

fluids.compressible.is_critical_flow(P1, P2, k)
Determines if a flow of a fluid driven by pressure gradient P1 - P2 is critical, for a fluid with the given isentropic
coefficient. This function calculates critical flow pressure, and checks if this is larger than P2. If so, the flow is
critical and choked.

**Parameters**
- P1 [float] Higher, source pressure [Pa]
- P2 [float] Lower, downstream pressure [Pa]
- k [float] Isentropic coefficient []

**Returns**
- flowtype [bool] True if the flow is choked; otherwise False

**Notes**
Assumes isentropic flow. Uses P_critical_flow function.
References

[1]

Examples

Examples 1-2 from API 520.

```python
>>> is_critical_flow(670E3, 532E3, 1.11)
False
>>> is_critical_flow(670E3, 101E3, 1.11)
True
```

**fluids.compressible.stagnation_energy(V)**

Calculates the increase in enthalpy $dH$ which is provided by a fluid’s velocity $V$.

$$\Delta H = \frac{V^2}{2}$$

Parameters

- $V$ [float] Velocity [m/s]

Returns

- $dH$ [float] Increase in enthalpy [J/kg]

Notes

The units work out. This term is pretty small, but not trivial.

References

[1]

Examples

```python
>>> stagnation_energy(125)
7812.5
```

**fluids.compressible.P_stagnation(P, T, Tst, k)**

Calculates stagnation flow pressure $P_{st}$ for a fluid with the given isentropic coefficient and specified stagnation temperature and normal temperature. Normally used with converging/diverging nozzles.

$$\frac{P_0}{P} = \left(\frac{T_0}{T}\right)^{\frac{k}{k-1}}$$

Parameters

- $P$ [float] Normal pressure of a fluid [Pa]
- $T$ [float] Normal temperature of a fluid [K]
- $Tst$ [float] Stagnation temperature of a fluid moving at a certain velocity [K]
- $k$ [float] Isentropic coefficient []
Fluids Documentation, Release 0.1

Returns

Pst [float] Stagnation pressure of a fluid moving at a certain velocity [Pa]

Notes

Assumes isentropic flow.

References

[1]

Examples

Example 12-1 in [1].

```python
>>> P_stagnation(54050., 255.7, 286.8, 1.4)
80772.80495900588
```

fluids.compressible.T_stagnation(T, P, Pst, k)

Calculates stagnation flow temperature $Tst$ for a fluid with the given isentropic coefficient and specified stagnation pressure and normal pressure. Normally used with converging/diverging nozzles.

$$T = T_0 \left( \frac{P}{P_0} \right)^{\frac{k-1}{k}}$$

Parameters

- T [float] Normal temperature of a fluid [K]
- P [float] Normal pressure of a fluid [Pa]
- Pst [float] Stagnation pressure of a fluid moving at a certain velocity [Pa]
- k [float] Isentropic coefficient []

Returns

- Tst [float] Stagnation temperature of a fluid moving at a certain velocity [K]

Notes

Assumes isentropic flow.

References

[1]

Examples

Example 12-1 in [1].

```python
>>> T_stagnation(286.8, 54050, 54050*8, 1.4)
519.5230938217768
```
Fluids Documentation, Release 0.1

**fluids.compressible.T_stagnation_ideal** (*T, V, Cp*)
Calculates the ideal stagnation temperature *T* calculated assuming the fluid has a constant heat capacity *Cp* and with a specified velocity *V* and temperature *T*.

\[ T^*_f = T + \frac{V^2}{2C_p} \]

**Parameters**
- *T* [float] Temperature [K]
- *V* [float] Velocity [m/s]
- *Cp* [float] Ideal heat capacity [J/kg/K]

**Returns**
- *Tst* [float] Stagnation temperature [J/kg]

**References**
[1]

**Examples**
Example 12-1 in [1].

```python
>>> T_stagnation_ideal(255.7, 250, 1005.)
286.79452736318405
```

2.3 Control valve sizing and rating (fluids.control_valve)

**fluids.control_valve.size_control_valve_l** (*rho, Psat, Pc, mu, P1, P2, Q, D1=None, D2=None, d=None, FL=0.9, Fd=1, allow_choked=True, allow_laminar=True, full_output=False*)
Calculates flow coefficient of a control valve passing a liquid according to IEC 60534. Uses a large number of inputs in SI units. Note the return value is not standard SI. All parameters are required. This sizing model does not officially apply to liquid mixtures, slurries, non-Newtonian fluids, or liquid-solid conveyance systems. For details of the calculations, consult [1].

**Parameters**
- *rho* [float] Density of the liquid at the inlet [kg/m^3]
- *Psat* [float] Saturation pressure of the fluid at inlet temperature [Pa]
- *Pc* [float] Critical pressure of the fluid [Pa]
- *mu* [float] Viscosity of the fluid [Pa*s]
- *P1* [float] Inlet pressure of the fluid before valves and reducers [Pa]
- *P2* [float] Outlet pressure of the fluid after valves and reducers [Pa]
- *Q* [float] Volumetric flow rate of the fluid [m^3/s]
- *D1* [float, optional] Diameter of the pipe before the valve [m]
**Fluids Documentation, Release 0.1**

**size_control_valve_l**(rho=965.4, Psat=70.1E3, Pc=22120E3, mu=3.1472E-4, T=30, MW=28.97, gamma=1.4, Z=1, P1=680E3, P2=220E3, Q=0.1, D1=0.15, D2=0.15, d=0.15, FL=0.9, Fd=0.46)

164.9954763704956

**size_control_valve_l**(rho=965.4, Psat=70.1E3, Pc=22120E3, mu=3.1472E-4, T=30, MW=28.97, gamma=1.4, Z=1, P1=680E3, P2=220E3, Q=0.1, D1=0.1, D2=0.1, d=0.1, FL=0.6, Fd=0.98)

238.05817216710483

Calculates flow coefficient of a control valve passing a gas according to IEC 60534. Uses a large number of
inputs in SI units. Note the return value is not standard SI. All parameters are required. For details of the calculations, consult [1]. Note the inlet gas flow conditions.

**Parameters**

- **T** [float] Temperature of the gas at the inlet [K]
- **MW** [float] Molecular weight of the gas [g/mol]
- **mu** [float] Viscosity of the fluid at inlet conditions [Pa*s]
- **gamma** [float] Specific heat capacity ratio [-]
- **Z** [float] Compressibility factor at inlet conditions, [-]
- **P1** [float] Inlet pressure of the gas before valves and reducers [Pa]
- **P2** [float] Outlet pressure of the gas after valves and reducers [Pa]
- **Q** [float] Volumetric flow rate of the gas at 273.15 K and 1 atm specifically [m³/s]
- **D1** [float, optional] Diameter of the pipe before the valve [m]
- **D2** [float, optional] Diameter of the pipe after the valve [m]
- **d** [float, optional] Diameter of the valve [m]
- **FL** [float, optional] Liquid pressure recovery factor of a control valve without attached fittings (normally 0.8-0.9 at full open and decreasing as opened further to below 0.5; use default very cautiously!) [-]
- **Fd** [float, optional] Valve style modifier (0.1 to 1; varies tremendously depending on the type of valve and position; do not use the default at all!) [-]
- **xT** [float, optional] Pressure difference ratio factor of a valve without fittings at choked flow (increasing to 0.9 or higher as the valve is closed further and decreasing to 0.1 or lower as the valve is opened further; use default very cautiously!) [-]

**Returns**

- **Kv** [float] Metric Kv valve flow coefficient (flow rate of water at a pressure drop of 1 bar) [m³/hr]

**Notes**

It is possible to use this model without any diameters specified; in that case, turbulent flow is assumed. Choked flow can still be modeled. This is not recommended. All three diameters need to be None for this to work. FL and Fd are not used by the models when the diameters are not specified, but xT definitely is used by the model.

**References**

[1]
Examples

From [1], matching example 3 for non-choked gas flow with attached fittings and a rotary, eccentric plug, flow-to-open control valve:

```python
>>> size_control_valve_g(T=433., MW=44.01, mu=1.4665E-4, gamma=1.30,
... Z=0.988, P1=680E3, P2=310E3, Q=38/36., D1=0.08, D2=0.1, d=0.05,
... FL=0.85, Fd=0.42, xT=0.60)
72.58664545391052
```

From [1], roughly matching example 4 for a small flow trim sized tapered needle plug valve. Difference is 3% and explained by the difference in algorithms used.

```python
>>> size_control_valve_g(T=320., MW=39.95, mu=5.625E-5, gamma=1.67, Z=1.0,
... P1=2.8E5, P2=1.3E5, Q=0.46/3600., D1=0.015, D2=0.015, d=0.015, FL=0.98,
... Fd=0.07, xT=0.8)
0.016498765335995726
```

`fluids.control_valve.cavitation_index(P1, P2, Psat)`

Calculates the cavitation index of a valve with upstream and downstream absolute pressures \(P_1\) and \(P_2\) for a fluid with a vapor pressure \(P_{sat}\).

\[
\sigma = \frac{P_1 - P_{sat}}{P_1 - P_2}
\]

**Parameters**

- **P1** [float] Absolute pressure upstream of the valve [Pa]
- **P2** [float] Absolute pressure downstream of the valve [Pa]
- **Psat** [float] Saturation pressure of the liquid at inlet temperature [Pa]

**Returns**

- **sigma** [float] Cavitation index of the valve [-]

**Notes**

Larger values are safer. Models for adjusting cavitation indexes provided by the manufacturer to the user’s conditions are available, making use of scaling the pressure differences and size differences.

Values can be calculated for incipient cavitation, constant cavitation, maximum vibration cavitation, incipient damage, and choking cavitation.

Has also been defined as:

\[
\sigma = \frac{P_2 - P_{sat}}{P_1 - P_2}
\]

Another definition and notation series is:

\[
K = xF = \frac{1}{\sigma} = \frac{P_1 - P_2}{P_1 - P_{sat}}
\]

**References**

[1]
Examples

>>> cavitation_index(1E6, 8E5, 2E5)
4.0

```python
fluids.control_valve.FF_critical_pressure_ratio_l(Psat, Pc)
Calculates FF, the liquid critical pressure ratio factor, for use in IEC 60534 liquid valve sizing calculations.

\[ F_F = 0.96 - 0.28 \sqrt{\frac{P_{sat}}{P_c}} \]

Parameters

Psat [float] Saturation pressure of the liquid at inlet temperature [Pa]
Pc [float] Critical pressure of the liquid [Pa]

Returns

FF [float] Liquid critical pressure ratio factor [-]

References

[1]

Examples

From [1], matching example.

```python
>>> FF_critical_pressure_ratio_l(70100.0, 22120000.0)
0.9442375225233299
```

```python
fluids.control_valve.is_choked_turbulent_l(dP, P1, Psat, FF, FL=None, FLP=None, FP=None)
Calculates if a liquid flow in IEC 60534 calculations is critical or not, for use in IEC 60534 liquid valve sizing calculations. Either FL may be provided or FLP and FP, depending on the calculation process.

\[ \Delta P > F_L^2(P_1 - FF P_{sat}) \]

\[ \Delta P \geq \left( \frac{FLP}{FP} \right)^2 (P_1 - FF P_{sat}) \]

Parameters

dP [float] Differential pressure across the valve, with reducer/expanders [Pa]
P1 [float] Pressure of the fluid before the valve and reducers/expanders [Pa]
Psat [float] Saturation pressure of the fluid at inlet temperature [Pa]
FF [float] Liquid critical pressure ratio factor [-]
FL [float, optional] Liquid pressure recovery factor of a control valve without attached fittings [-]
FLP [float, optional] Combined liquid pressure recovery factor with piping geometry factor, for a control valve with attached fittings [-]
FP [float, optional] Piping geometry factor [-]"
Returns

**choked**  [bool] Whether or not the flow is choked [-]

References

[1]

Examples

```python
>>> is_choked_turbulent_l(460.0, 680.0, 70.1, 0.94, 0.9)
False
>>> is_choked_turbulent_l(460.0, 680.0, 70.1, 0.94, 0.6)
True
```

**fluids.control_valve.is_choked_turbulent_g**(*x, Fgamma, xT=None, xTP=None*)

Calculates if a gas flow in IEC 60534 calculations is critical or not, for use in IEC 60534 gas valve sizing calculations. Either xT or xTP must be provided, depending on the calculation process.

\[
x \geq F_\gamma x_T
\]

\[
x \geq F_\gamma x_{TP}
\]

Parameters

- **x**  [float] Differential pressure over inlet pressure, [-]
- **Fgamma**  [float] Specific heat ratio factor [-]
- **xT**  [float, optional] Pressure difference ratio factor of a valve without fittings at choked flow [-]
- **xTP**  [float] Pressure difference ratio factor of a valve with fittings at choked flow [-]

Returns

**choked**  [bool] Whether or not the flow is choked [-]

References

[1]

Examples

Example 3, compressible flow, non-choked with attached fittings:

```python
>>> is_choked_turbulent_g(0.544, 0.929, 0.6)
False
>>> is_choked_turbulent_g(0.544, 0.929, xTP=0.625)
False
```

**fluids.control_valve.Reynolds_valve**(*nu, Q, D1, FL, Fd, C*)

Calculates Reynolds number of a control valve for a liquid or gas flowing through it at a specified Q, for a specified D1, FL, Fd, C, and with kinematic viscosity *nu* according to IEC 60534 calculations.

\[
Re_v = \frac{N_4 F_d Q}{\nu \sqrt{C_F L}} \left( \frac{F_d^2 C^2}{N_2 D^4} + 1 \right)^{1/4}
\]
Parameters

- **nu** [float] Kinematic viscosity, [m^2/s]
- **Q** [float] Volumetric flow rate of the fluid [m^3/s]
- **D1** [float] Diameter of the pipe before the valve [m]
- **FL** [float, optional] Liquid pressure recovery factor of a control valve without attached fittings []
- **Fd** [float] Valve style modifier [-]
- **C** [float] Metric Kv valve flow coefficient (flow rate of water at a pressure drop of 1 bar) [m^3/hr]

Returns

- **Rev** [float] Valve reynolds number [-]

References

[1]

Examples

```python
>>> Reynolds_valve(3.26e-07, 360, 150.0, 0.9, 0.46, 165)
2966984.7525455453
```

`fluids.control_valve.loss_coefficient_piping(d, D1=None, D2=None)`

Calculates the sum of loss coefficients from possible inlet/outlet reducers/expanders around a control valve according to IEC 60534 calculations.

\[
\Sigma \xi = \xi_1 + \xi_2 + \xi_{B1} - \xi_{B2}
\]

\[
\xi_1 = 0.5 \left[ 1 - \left( \frac{d}{D_1} \right)^2 \right]^2
\]

\[
\xi_2 = 1.0 \left[ 1 - \left( \frac{d}{D_2} \right)^2 \right]^2
\]

\[
\xi_{B1} = 1 - \left( \frac{d}{D_1} \right)^4
\]

\[
\xi_{B2} = 1 - \left( \frac{d}{D_2} \right)^4
\]

Parameters

- **d** [float] Diameter of the valve [m]
- **D1** [float] Diameter of the pipe before the valve [m]
- **D2** [float] Diameter of the pipe after the valve [m]

Returns

- **loss** [float] Sum of the four loss coefficients [-]
Examples

In example 3, non-choked compressible flow with fittings:

```python
>>> loss_coefficient_piping(0.05, 0.08, 0.1)
0.6580810546875
```

`fluids.control_valve.Reynolds_factor(FL, C, d, Rev, full_trim=True)`
Calculates the Reynolds number factor $FR$ for a valve with a Reynolds number $Rev$, diameter $d$, flow coefficient $C$, liquid pressure recovery factor $FL$, and with either full or reduced trim, all according to IEC 60534 calculations.

If full trim:

$$FR_{1a} = 1 + \left( \frac{0.33F_{L}^{0.5}}{n_{1}^{0.25}} \right) \log_{10} \left( \frac{Re_{v}}{10000} \right)$$

$$FR_{2} = \min \left( \frac{0.026}{F_{L}} \sqrt{n_{1}Re_{v}}, 1 \right)$$

$$n_{1} = \frac{N_{2}}{(\frac{C}{d})^{2}}$$

$$FR = FR_{2} \text{ if } Rev < 10 \text{ else } \min(FR_{1a}, FR_{2})$$

Otherwise:

$$FR_{3a} = 1 + \left( \frac{0.33F_{L}^{0.5}}{n_{2}^{0.25}} \right) \log_{10} \left( \frac{Re_{v}}{10000} \right)$$

$$FR_{4} = \frac{0.026}{F_{L}} \sqrt{n_{2}Re_{v}}$$

$$n_{2} = 1 + N_{32} \left( \frac{C}{d} \right)^{2/3}$$

$$FR = FR_{4} \text{ if } Rev < 10 \text{ else } \min(FR_{3a}, FR_{4})$$

Parameters

- `FL` [float] Liquid pressure recovery factor of a control valve without attached fittings []
- `C` [float] Metric Kv valve flow coefficient (flow rate of water at a pressure drop of 1 bar) [m³/hr]
- `d` [float] Diameter of the valve [m]
- `Rev` [float] Valve reynolds number [-]
- `full_trim` [bool] Whether or not the valve has full trim

Returns

- `FR` [float] Reynolds number factor for laminar or transitional flow []

References

[1]
Examples

In Example 4, compressible flow with small flow trim sized for gas flow (Cv in the problem was converted to 
Kv here to make FR match with N32, N2):

```python
>>> Reynolds_factor(FL=0.98, C=0.015483, d=15., Rev=1202., full_trim=False)
0.7148753122302025
```

fluids.control_valve.Cv_char_quick_opening(opening)

fluids.control_valve.Cv_char_linear(opening)

fluids.control_valve.Cv_char_equal_percentage(opening)

fluids.control_valve.convert_flow_coefficient(flow_coefficient, old_scale, new_scale)

Convert from one flow coefficient scale to another; supports the Kv Cv, and Av scales.

Other scales are Qn and Cg, but clear definitions have yet to be found.

Parameters

- **flow_coefficient** [float] Value of the flow coefficient to be converted, expressed in the original
  scale.
- **old_scale** [str] String specifying the original scale; one of ‘Av’, ‘Cv’, or ‘Kv’, [-]
- **new_scale** [str] String specifying the new scale; one of ‘Av’, ‘Cv’, or ‘Kv’, [-]

Returns

- **converted_flow_coefficient** [float] Flow coefficient converted to the specified scale.

Notes

Qn is a scale based on a flow of air in units of L/minute as air travels through a valve and loses one bar of
pressure (initially 7 bar absolute, to 6 bar absolute). No consistent conversion factors have been found and those
from theory do not match what have been found. Some uses of Qn use its flow rate as in normal (STP reference
conditions) flow rate of air; others use something like the 7 bar absolute condition.

Examples

```python
>>> convert_flow_coefficient(10, 'Kv', 'Av')
0.0002776532068951358
```

fluids.control_valve.control_valve_choke_P_l(Psat, Pc, FL, P1=None, P2=None, disp=True)

Calculates either the upstream or downstream pressure at which choked flow through a liquid control valve
occurs, given either a set upstream or downstream pressure. Implements an analytical solution of the needed
equations from the full function size_control_valve_l. For some pressures, no choked flow is possible;
for choked flow to occur the direction of flow must be reversed. If disp is True, an exception will be raised for
these conditions.

\[
P_1 = \frac{F_F P_{sat}^2 - P_2}{F_L^2 - 1}
\]

\[
P_2 = F_F F_L P_{sat} - F_L^2 P_1 + P_1
\]

Parameters
Psat [float] Saturation pressure of the liquid at inlet temperature [Pa]
Pc [float] Critical pressure of the liquid [Pa]
FL [float, optional] Liquid pressure recovery factor of a control valve without attached fittings [-]
P1 [float, optional] Absolute pressure upstream of the valve [Pa]
P2 [float, optional] Absolute pressure downstream of the valve [Pa]
disp [bool, optional] Whether or not to raise an exception on flow reversal, [-]

Returns

P_choke [float] Pressure at which a choke occurs in the liquid valve [Pa]

Notes

Extremely cheap to compute.

Examples

```python
>>> control_valve_choke_P_l(69682.89291024722, 22048320.0, 0.6, 680000.0)
458887.5306077305
>>> control_valve_choke_P_l(69682.89291024722, 22048320.0, 0.6, P2=458887.5306077305)
680000.0
```

`fluids.control_valve.control_valve_choke_P_g(xT, gamma, P1=None, P2=None)`

Calculates either the upstream or downstream pressure at which choked flow through a gas control valve occurs, given either a set upstream or downstream pressure. Implements an analytical solution of the needed equations from the full function `size_control_valve_g`. A singularity arises as `xT` goes to 1 and `gamma` goes to 1.4.

\[
P_1 = -\frac{7P_2}{5\gamma x_T - 1}
\]

\[
P_2 = \frac{P_1}{T} (-5\gamma x_T + 7)
\]

Parameters

xT [float, optional] Pressure difference ratio factor of a valve without fittings at choked flow [-]
gamma [float] Specific heat capacity ratio [-]
P1 [float, optional] Absolute pressure upstream of the valve [Pa]
P2 [float, optional] Absolute pressure downstream of the valve [Pa]

Returns

P_choke [float] Pressure at which a choke occurs in the gas valve [Pa]

Notes

Extremely cheap to compute.
Examples

```python
>>> control_valve_choke_P_g(1, 1.3, 1E5)
7142.857142857143
>>> control_valve_choke_P_g(1, 1.3, P2=7142.857142857143)
100000.0
```

```python
fluids.control_valve.control_valve_noise_l_2015(m, P1, P2, Psat, rho, c, Kv, d, Di, FL, Fd, t_pipe, rho_pipe=7800.0, c_pipe=5000.0, rho_air=1.2, c_air=343.0, xFz=None, An=-4.6)
```

Calculates the sound made by a liquid flowing through a control valve according to the standard IEC 60534-8-4 (2015) [1].

**Parameters**

- **m** [float] Mass flow rate of liquid through the control valve, [kg/s]
- **P1** [float] Inlet pressure of the fluid before valves and reducers [Pa]
- **P2** [float] Outlet pressure of the fluid after valves and reducers [Pa]
- **Psat** [float] Saturation pressure of the fluid at inlet temperature [Pa]
- **rho** [float] Density of the liquid at the inlet [kg/m^3]
- **c** [float] Speed of sound of the liquid at the inlet conditions [m/s]
- **Kv** [float] Metric Kv valve flow coefficient (flow rate of water at a pressure drop of 1 bar) [m^3/hr]
- **d** [float] Diameter of the valve [m]
- **Di** [float] Internal diameter of the pipe before and after the valve [m]
- **FL** [float, optional] Liquid pressure recovery factor of a control valve without attached fittings (normally 0.8-0.9 at full open and decreasing as opened further to below 0.5) [-]
- **Fd** [float, optional] Valve style modifier [-]
- **t_pipe** [float] Wall thickness of the pipe after the valve, [m]
- **rho_pipe** [float, optional] Density of the pipe wall material at flowing conditions, [kg/m^3]
- **c_pipe** [float, optional] Speed of sound of the pipe wall material at flowing conditions, [m/s]
- **rho_air** [float, optional] Density of the air surrounding the valve and pipe wall, [kg/m^3]
- **c_air** [float, optional] Speed of sound of the air surrounding the valve and pipe wall, [m/s]
- **xFz** [float, optional] If specified, this value xFz is used instead of estimated; the calculation is sensitive to this value, [-]
- **An** [float, optional] Valve correction factor for acoustic efficiency

**Returns**

- **LpAe1m** [float] A weighted sound pressure level 1 m from the pipe wall, 1 m distance downstream of the valve (at reference sound pressure level 2E-5), [dBA]

2.3. Control valve sizing and rating (fluids.control_valve) 73
Notes

For formulas see [1]. This takes on the order of 100 us to compute. This model can also tell if noise is being produced in a valve just due to turbulent flow, or cavitation. For values of \( An \), see [1]; it is normally -4.6 for globe valves, -4.3 for butterfly valves, and -4.0 for expanders.

This model was checked against three examples in [1]; they match to all given decimals.

A formula is given in [1] for multihole trim valves to estimate \( xF_z \) as well; this is not implemented here and \( xF_z \) must be calculated by the user separately. The formula is

\[
x_Fz = \left( 4.5 + 1650 \frac{N_0 d_H^2}{F_L} \right)^{-1/2}
\]

Where \( N_0 \) is the number of open channels and \( d_H \) is the multihole trim hole diameter.

References

[1]

Examples

```python
>>> control_valve_noise_l_2015(m=40, P1=1E6, P2=6.5E5, Psat=2.32E3, ...
... rho=997, c=1400, Kv=77.848, d=0.1, Di=0.1071, FL=0.92, Fd=0.42, ...
... t_pipe=0.0036, rho_pipe=7800.0, c_pipe=5000.0, rho_air=1.293,
... c_air=343.0, An=-4.6)
81.58200097996539
```

Calculates the sound made by a gas flowing through a control valve according to the standard IEC 60534-8-3 (2011) [1].

Parameters

- \( m \) [float] Mass flow rate of gas through the control valve, [kg/s]
- \( P1 \) [float] Inlet pressure of the gas before valves and reducers [Pa]
- \( P2 \) [float] Outlet pressure of the gas after valves and reducers [Pa]
- \( T1 \) [float] Inlet gas temperature, [K]
- \( \rho \) [float] Density of the gas at the inlet [kg/m\(^3\)]
- \( \gamma \) [float] Specific heat capacity ratio [-]
- \( MW \) [float] Molecular weight of the gas [g/mol]
- \( Kv \) [float] Metric \( Kv \) valve flow coefficient (flow rate of water at a pressure drop of 1 bar) [m\(^3\)/hr]
- \( d \) [float] Diameter of the valve [m]
- \( Di \) [float] Internal diameter of the pipe before and after the valve [m]
t_pipe [float] Wall thickness of the pipe after the valve, [m]

Fd [float] Valve style modifier (0.1 to 1; varies tremendously depending on the type of valve and position; do not use the default at all!) [-]

FL [float] Liquid pressure recovery factor of a control valve without attached fittings (normally 0.8-0.9 at full open and decreasing as opened further to below 0.5; use default very cautiously!) [-]

FLP [float, optional] Combined liquid pressure recovery factor with piping geometry factor, for a control valve with attached fittings [-]

FP [float, optional] Piping geometry factor [-]

rho_pipe [float, optional] Density of the pipe wall material at flowing conditions, [kg/m^3]

c_pipe [float, optional] Speed of sound of the pipe wall material at flowing conditions, [m/s]

P_air [float, optional] Pressure of the air surrounding the valve and pipe wall, [Pa]

rho_air [float, optional] Density of the air surrounding the valve and pipe wall, [kg/m^3]

c_air [float, optional] Speed of sound of the air surrounding the valve and pipe wall, [m/s]

An [float, optional] Valve correction factor for acoustic efficiency

Stp [float, optional] Strouhal number at the peak fp; between 0.1 and 0.3 typically, [-]

T2 [float, optional] Outlet gas temperature; assumed T1 if not provided (a PH flash should be used to obtain this if possible), [K]

beta [float, optional] Valve outlet / expander inlet contraction coefficient, [-]

Returns

LpAe1m [float] A weighted sound pressure level 1 m from the pipe wall, 1 m distance downstream of the valve (at reference sound pressure level 2E-5), [dBA]

Notes

For formulas see [1]. This takes on the order of 100 us to compute. For values of An, see [1]. This model was checked against six examples in [1]; they match to all given decimals. Several additional formulas are given for multihole trim valves, control valves with two or more fixed area stages, and multipath, multistage trim valves.

References

[1]

Examples

```python
>>> control_valve_noise_g_2011(m=2.22, P1=1E6, P2=7.2E5, T1=450, rho=5.3, 
... gamma=1.22, MW=19.8, Kv=77.85, d=0.1, Di=0.2031, FL=None, FLP=0.792, 
... FP=0.98, Fd=0.296, t_pipe=0.008, rho_pipe=8000.0, c_pipe=5000.0, 
... rho_air=1.293, c_air=343.0, An=-3.8, Stp=0.2)
91.67702674629604
```
2.4 Dimensionless numbers (fluids.core)

**fluids.core.Reynolds** *(V, D, rho=None, mu=None, nu=None)*
Calculates Reynolds number or *Re* for a fluid with the given properties for the specified velocity and diameter.

\[
Re = \frac{D \cdot V}{\nu} = \frac{\rho V D}{\mu}
\]

Inputs either of any of the following sets:
- V, D, density *rho* and kinematic viscosity *mu*
- V, D, and dynamic viscosity *nu*

**Parameters**
- V [float] Velocity [m/s]
- D [float] Diameter [m]
- rho [float, optional] Density, [kg/m³]
- mu [float, optional] Dynamic viscosity, [Pa*s]
- nu [float, optional] Kinematic viscosity, [m²/s]

**Returns**
- Re [float] Reynolds number []

**Notes**

\[
Re = \frac{\text{Momentum}}{\text{Viscosity}}
\]

An error is raised if none of the required input sets are provided.

**References**

[1], [2]

**Examples**

```python
>>> Reynolds(2.5, 0.25, 1.1613, 1.9E-5)
38200.65789473684
>>> Reynolds(2.5, 0.25, nu=1.636e-05)
38202.93398533008
```

**fluids.core.Prandtl** *(Cp=None, k=None, mu=None, nu=None, rho=None, alpha=None)*
Calculates Prandtl number or *Pr* for a fluid with the given parameters.

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

Inputs can be any of the following sets:
- Heat capacity, dynamic viscosity, and thermal conductivity
- Thermal diffusivity and kinematic viscosity
• Heat capacity, kinematic viscosity, thermal conductivity, and density

Parameters

\( C_p \) [float] Heat capacity, [J/kg/K]
\( k \) [float] Thermal conductivity, [W/m/K]
\( \mu \) [float, optional] Dynamic viscosity, [Pa*s]
\( \nu \) [float, optional] Kinematic viscosity, [m^2/s]
\( \rho \) [float] Density, [kg/m^3]
\( \alpha \) [float] Thermal diffusivity, [m^2/s]

Returns

\( Pr \) [float] Prandtl number []

Notes

\[ Pr = \frac{\text{kinematic viscosity}}{\text{thermal diffusivity}} = \frac{\text{momentum diffusivity}}{\text{thermal diffusivity}} \]

An error is raised if none of the required input sets are provided.

References

[1], [2], [3]

Examples

```python
>>> Prandtl(Cp=1637., k=0.010, mu=4.61E-6)
0.754657
>>> Prandtl(Cp=1637., k=0.010, nu=6.4E-7, rho=7.1)
0.7438528
>>> Prandtl(nu=6.3E-7, alpha=9E-7)
0.7000000000000001
```

\texttt{fluids.core.Grashof} \( (L, \beta, T1, T2=0, \rho=\text{None}, \mu=\text{None}, nu=\text{None}, g=9.80665) \)

Calculates Grashof number or \( Gr \) for a fluid with the given properties, temperature difference, and characteristic length.

\[ Gr = \frac{g\beta(T_s - T_\infty) L^3}{\nu^2} = \frac{g\beta(T_s - T_\infty) L^3 \rho^2}{\mu^2} \]

Inputs either of any of the following sets:

• L, beta, T1 and T2, and density rho and kinematic viscosity mu
• L, beta, T1 and T2, and dynamic viscosity nu

Parameters

\( L \) [float] Characteristic length [m]
\( \beta \) [float] Volumetric thermal expansion coefficient [1/K]
\( T1 \) [float] Temperature 1, usually a film temperature [K]
**T2** [float, optional] Temperature 2, usually a bulk temperature (or 0 if only a difference is provided to the function) [K]

**rho** [float, optional] Density, [kg/m^3]

**mu** [float, optional] Dynamic viscosity, [Pa*s]

**nu** [float, optional] Kinematic viscosity, [m^2/s]

**g** [float, optional] Acceleration due to gravity, [m/s^2]

**Returns**

Gr [float] Grashof number []

**Notes**

\[
Gr = \frac{\text{Buoyancy forces}}{\text{Viscous forces}}
\]

An error is raised if none of the required input sets are provided. Used in free convection problems only.

**References**

[1], [2]

**Examples**

Example 4 of [1], p. 1-21 (matches):

```python
>>> Grashof(L=0.9144, beta=0.000933, T1=178.2, rho=1.1613, mu=1.9E-5)
4656936556.178915
>>> Grashof(L=0.9144, beta=0.000933, T1=378.2, T2=200, nu=1.636e-05)
4657491516.530312
```

**fluids.core.Nusselt** *(h, L, k)*

Calculates Nusselt number \(Nu\) for a heat transfer coefficient \(h\), characteristic length \(L\), and thermal conductivity \(k\).

\[
Nu = \frac{hL}{k}
\]

**Parameters**

- **h** [float] Heat transfer coefficient, [W/m^2/K]
- **L** [float] Characteristic length, no typical definition [m]
- **k** [float] Thermal conductivity of fluid [W/m/K]

**Returns**

- **Nu** [float] Nusselt number, [-]

**Notes**

Do not confuse \(k\), the thermal conductivity of the fluid, with that of within a solid object associated with!

\[
Nu = \frac{\text{Convective heat transfer}}{\text{Conductive heat transfer}}
\]
References

[1], [2]

Examples

```python
>>> Nusselt(1000., 1.2, 300.)
4.0
>>> Nusselt(10000., .01, 4000.)
0.025
```

`fluids.core.Sherwood(K, L, D)`
Calculates Sherwood number $Sh$ for a mass transfer coefficient $K$, characteristic length $L$, and diffusivity $D$.

\[ Sh = \frac{KL}{D} \]

Parameters

- $K$ [float] Mass transfer coefficient, [m/s]
- $L$ [float] Characteristic length, no typical definition [m]
- $D$ [float] Diffusivity of a species [m/s²]

Returns

- $Sh$ [float] Sherwood number, [-]

Notes

\[ Sh = \frac{\text{Mass transfer by convection}}{\text{Mass transfer by diffusion}} = \frac{K}{D/L} \]

References

[1]

Examples

```python
>>> Sherwood(1000., 1.2, 300.)
4.0
```

`fluids.core.Rayleigh(Pr, Gr)`
Calculates Rayleigh number or $Ra$ using Prandtl number $Pr$ and Grashof number $Gr$ for a fluid with the given properties, temperature difference, and characteristic length used to calculate $Gr$ and $Pr$.

\[ Ra = PrGr \]

Parameters

- $Pr$ [float] Prandtl number
- $Gr$ [float] Grashof number

Returns

- $Ra$ [float] Rayleigh number
Notes

Used in free convection problems only.

References

[1], [2]

Examples

```python
>>> Rayleigh(1.2, 4.6E9)
5520000000.0
```

Fluids Documentation: `fluids.core.Schmidt(D, mu=None, nu=None, rho=None)`
Calculates Schmidt number or Sc for a fluid with the given parameters.

\[
Sc = \frac{\mu}{D \rho} = \frac{\nu}{D}
\]

Inputs can be any of the following sets:

- Diffusivity, dynamic viscosity, and density
- Diffusivity and kinematic viscosity

Parameters

- **D** [float] Diffusivity of a species, [m^2/s]
- **mu** [float, optional] Dynamic viscosity, [Pa*s]
- **nu** [float, optional] Kinematic viscosity, [m^2/s]
- **rho** [float, optional] Density, [kg/m^3]

Returns

- **Sc** [float] Schmidt number []

Notes

\[
Sc = \frac{\text{kinematic viscosity}}{\text{molecular diffusivity}} = \frac{\text{viscous diffusivity}}{\text{species diffusivity}}
\]

An error is raised if none of the required input sets are provided.

References

[1], [2]
Examples

```python
>>> Schmidt(D=2E-6, mu=4.61E-6, rho=800)
0.00288125
>>> Schmidt(D=1E-9, nu=6E-7)
599.9999999999999
```

```
fluids.core.Peclet_heat(V, L, rho=None, Cp=None, k=None, alpha=None)
Calculates heat transfer Peclet number or Pe for a specified velocity V, characteristic length L, and specified properties for the given fluid.

\[ P_e = \frac{V L \rho C_p}{k} = \frac{L V}{\alpha} \]

Inputs either of any of the following sets:
- V, L, density rho, heat capacity Cp, and thermal conductivity k
- V, L, and thermal diffusivity alpha

Parameters
- V [float] Velocity [m/s]
- L [float] Characteristic length [m]
- rho [float, optional] Density, \([\text{kg/m}^3]\)
- Cp [float, optional] Heat capacity, \([\text{J/kg/K}]\)
- k [float, optional] Thermal conductivity, \([\text{W/m/K}]\)
- alpha [float, optional] Thermal diffusivity, \([\text{m}^2/\text{s}]\)

Returns
- Pe [float] Peclet number (heat) []

Notes

\[ P_e = \frac{\text{Bulk heat transfer}}{\text{Conduction heat transfer}} \]

An error is raised if none of the required input sets are provided.

References

[1], [2]

Examples

```python
>>> Peclet_heat(1.5, 2, 1000., 4000., 0.6)
20000000.0
>>> Peclet_heat(1.5, 2, alpha=1E-7)
30000000.0
```
fluids.core.Peclet_mass(V, L, D)
Calculates mass transfer Peclet number or \( Pe \) for a specified velocity \( V \), characteristic length \( L \), and diffusion coefficient \( D \).

\[
P_e = \frac{LV}{D}
\]

Parameters
- \( V \) [float] Velocity [m/s]
- \( L \) [float] Characteristic length [m]
- \( D \) [float] Diffusivity of a species, [m^2/s]

Returns
- \( Pe \) [float] Peclet number (mass) []

Notes
\[
P_e = \frac{\text{Advective transport rate}}{\text{Diffusive transport rate}}
\]

References
[1]

Examples
```python
>>> Peclet_mass(1.5, 2, 1E-9)
3000000000.0
```

fluids.core.Fourier_heat(t, L, rho=None, Cp=None, k=None, alpha=None)
Calculates heat transfer Fourier number or \( Fo \) for a specified time \( t \), characteristic length \( L \), and specified properties for the given fluid.

\[
F_o = \frac{kt}{C_p\rho L^2} = \frac{\alpha t}{L^2}
\]

Inputs either of any of the following sets:
- \( t, L, \) density \( rho \), heat capacity \( Cp \), and thermal conductivity \( k \)
- \( t, L, \) and thermal diffusivity \( alpha \)

Parameters
- \( t \) [float] time [s]
- \( L \) [float] Characteristic length [m]
- \( rho \) [float, optional] Density, [kg/m^3]
- \( Cp \) [float, optional] Heat capacity, [J/kg/K]
- \( k \) [float, optional] Thermal conductivity, [W/m/K]
- \( alpha \) [float, optional] Thermal diffusivity, [m^2/s]
Returns

\( F_o \) [float] Fourier number (heat) []

Notes

\[
F_o = \frac{\text{Heat conduction rate}}{\text{Rate of thermal energy storage in a solid}}
\]

An error is raised if none of the required input sets are provided.

References

[1], [2]

Examples

```python
>>> Fourier_heat(t=1.5, L=2, rho=1000., Cp=4000., k=0.6)
5.625e-08
>>> Fourier_heat(1.5, 2, alpha=1E-7)
3.75e-08
```

\texttt{fluids.core.Fourier\_mass}(t, \( L \), \( D \))

Calculates mass transfer Fourier number or \( F_o \) for a specified time \( t \), characteristic length \( L \), and diffusion coefficient \( D \).

\[
F_o = \frac{D t}{L^2}
\]

Parameters

- \( t \) [float] time [s]
- \( L \) [float] Characteristic length [m]
- \( D \) [float] Diffusivity of a species, [m^2/s]

Returns

\( F_o \) [float] Fourier number (mass) []

Notes

\[
F_o = \frac{\text{Diffusive transport rate}}{\text{Storage rate}}
\]

References

[1]
Examples

```python
>>> Fourier_mass(t=1.5, L=2, D=1E-9)
3.7500000000000005e-10
```

```
fluids.core.Graetz_heat(V, D, x, rho=None, Cp=None, k=None, alpha=None)
Calculates Graetz number or Gz for a specified velocity V, diameter D, axial distance x, and specified properties for the given fluid.

\[
Gz = \frac{V D^2 \cdot C_p \rho}{x \cdot k} = \frac{V D^2}{x \alpha}
\]

Inputs either of any of the following sets:
- V, D, x, density rho, heat capacity Cp, and thermal conductivity k
- V, D, x, and thermal diffusivity alpha

Parameters
- V [float] Velocity, [m/s]
- D [float] Diameter [m]
- x [float] Axial distance [m]
- rho [float, optional] Density, [kg/m^3]
- Cp [float, optional] Heat capacity, [J/kg/K]
- k [float, optional] Thermal conductivity, [W/m/K]
- alpha [float, optional] Thermal diffusivity, [m^2/s]

Returns
- Gz [float] Graetz number []

Notes

\[
Gz = \frac{\text{Time for radial heat diffusion in a fluid by conduction}}{\text{Time taken by fluid to reach distance x}} = \frac{D}{x Re Pr}
\]

An error is raised if none of the required input sets are provided.

References

[1]

Examples

```python
>>> Graetz_heat(1.5, 0.25, 5, 800., 2200., 0.6)
55000.0
>>> Graetz_heat(1.5, 0.25, 5, alpha=1E-7)
187500.0
```
Fluids Documentation, Release 0.1

**fluids.core.Lewis** *(D=None, alpha=None, Cp=None, k=None, rho=None)*

Calculates Lewis number or $Le$ for a fluid with the given parameters.

\[ Le = \frac{k}{\rho C_p D} = \frac{\alpha}{D} \]

Inputs can be either of the following sets:
- Diffusivity and Thermal diffusivity
- Diffusivity, heat capacity, thermal conductivity, and density

**Parameters**
- **D** [float] Diffusivity of a species, [m^2/s]
- **alpha** [float, optional] Thermal diffusivity, [m^2/s]
- **Cp** [float, optional] Heat capacity, [J/kg/K]
- **k** [float, optional] Thermal conductivity, [W/m/K]
- **rho** [float, optional] Density, [kg/m^3]

**Returns**
- **Le** [float] Lewis number

**Notes**

\[ Le = \frac{\text{Thermal diffusivity}}{\text{Mass diffusivity}} = \frac{Sc}{Pr} \]

An error is raised if none of the required input sets are provided.

**References**
[1], [2], [3]

**Examples**

```python
>>> Lewis(D=22.6E-6, alpha=19.1E-6)
0.8451327433628318
>>> Lewis(D=22.6E-6, rho=800., k=.2, Cp=2200)
0.00502815768302494
```

**fluids.core.Weber** *(V, L, rho, sigma)*

Calculates Weber number, $We$, for a fluid with the given density, surface tension, velocity, and geometric parameter (usually diameter of bubble).

\[ We = \frac{V^2 L \rho}{\sigma} \]

**Parameters**
- **V** [float] Velocity of fluid, [m/s]
- **L** [float] Characteristic length, typically bubble diameter [m]
- **rho** [float] Density of fluid, [kg/m^3]

2.4. Dimensionless numbers (fluids.core) 85
**sigma** [float] Surface tension, [N/m]

**Returns**

**We** [float] Weber number []

**Notes**

Used in bubble calculations.

\[ We = \frac{\text{inertial force}}{\text{surface tension force}} \]

**References**

[1], [2], [3]

**Examples**

```python
>>> Weber(V=0.18, L=0.001, rho=900., sigma=0.01)
2.916
```

**fluids.core.Mach(V,c)**

Calculates Mach number or Ma for a fluid of velocity V with speed of sound c.

\[ Ma = \frac{V}{c} \]

**Parameters**

V [float] Velocity of fluid, [m/s]

c [float] Speed of sound in fluid, [m/s]

**Returns**

Ma [float] Mach number []

**Notes**

Used in compressible flow calculations.

\[ Ma = \frac{\text{fluid velocity}}{\text{sonic velocity}} \]

**References**

[1], [2]
Examples

```python
>>> Mach(33., 330)
0.1
```

`fluids.core.Knudsen(path, L)`
Calculates Knudsen number or $Kn$ for a fluid with mean free path $path$ and for a characteristic length $L$.

$$Kn = \frac{\lambda}{L}$$

**Parameters**
- `path` [float] Mean free path between molecular collisions, [m]
- `L` [float] Characteristic length, [m]

**Returns**
- `Kn` [float] Knudsen number []

**Notes**
Used in mass transfer calculations.

$$Kn = \frac{\text{Mean free path length}}{\text{Characteristic length}}$$

**References**
[1], [2]

**Examples**

```python
>>> Knudsen(1e-10, .001)
1e-07
```

`fluids.core.Bond(rhol, rhog, sigma, L)`
Calculates Bond number, $Bo$ also known as Eotvos number, for a fluid with the given liquid and gas densities, surface tension, and geometric parameter (usually length).

$$Bo = \frac{g(\rho_l - \rho_g)L^2}{\sigma}$$

**Parameters**
- `rhol` [float] Density of liquid, [kg/m^3]
- `rhog` [float] Density of gas, [kg/m^3]
- `sigma` [float] Surface tension, [N/m]
- `L` [float] Characteristic length, [m]

**Returns**
- `Bo` [float] Bond number []

2.4. Dimensionless numbers (fluids.core)
fluids.core.Dean(Re, Di, D)
Calculates Dean number, $De$, for a fluid with the Reynolds number $Re$, inner diameter $Di$, and a secondary diameter $D$. $D$ may be the diameter of curvature, the diameter of a spiral, or some other dimension.

$$De = \sqrt[2]{\frac{D_i}{D} Re} = \sqrt[2]{\frac{D_i \rho l D}{\mu}}$$

Parameters

- **Re** [float] Reynolds number []
- **Di** [float] Inner diameter []
- **D** [float] Diameter of curvature or outer spiral or other dimension []

Returns

- **De** [float] Dean number [-]

Notes

Used in flow in curved geometry.

$$De = \sqrt{\frac{\text{centripetal forces} \cdot \text{inertial forces}}{\text{viscous forces}}}$$

References

[1]

Examples

```python
>>> Bond(1000., 1.2, .0589, 2)
665187.2339558573
```

fluids.core.Morton(rhol, rhog, mul, sigma, g=9.80665)
Calculates Morton number or $Mo$ for a liquid and vapor with the specified properties, under the influence of gravitational force $g$.

$$Mo = \frac{g \mu l^4 (\rho_l - \rho_g)}{\rho_l^2 \sigma^3}$$

Parameters

- **rhol** [float] Density of liquid phase, [kg/m^3]
- **rhog** [float] Density of gas phase, [kg/m^3]
**mul** [float] Viscosity of liquid phase, [Pa*s]  
**sigma** [float] Surface tension between liquid-gas phase, [N/m]  
**g** [float, optional] Acceleration due to gravity, [m/s^2]  

**Returns**  
**Mo** [float] Morton number, [-]  

**Notes**  
Used in modeling bubbles in liquid.  

**References**  
[1], [2]  

**Examples**  
```python  
>>> Morton(1077.0, 76.5, 4.27E-3, 0.023)  
2.3118310430743e-07  
```

`fluids.core.Froude(V, L, g=9.80665, squared=False)`  
Calculates Froude number \( Fr \) for velocity \( V \) and geometric length \( L \). If desired, gravity can be specified as well. Normally the function returns the result of the equation below; Froude number is also often said to be defined as the square of the equation below.  
\[
Fr = \frac{V}{\sqrt{gL}}
\]

**Parameters**  
\( V \) [float] Velocity of the particle or fluid, [m/s]  
\( L \) [float] Characteristic length, no typical definition [m]  
\( g \) [float, optional] Acceleration due to gravity, [m/s^2]  
  
**squared** [bool, optional] Whether to return the squared form of Froude number  

**Returns**  
**Fr** [float] Froude number, [-]  

**Notes**  
Many alternate definitions including density ratios have been used.  
\[
Fr = \frac{\text{Inertial Force}}{\text{Gravity Force}}
\]

**References**  
[1], [2]
Examples

```python
>>> Froude(1.83, L=2., g=1.63)
1.0135432593877318
>>> Froude(1.83, L=2., squared=True)
0.17074638128208924
```

```python
fluids.core.Froude_densimetric(V, L, rho1, rho2, heavy=True, g=9.80665)
```

Calculates the densimetric Froude number \( \text{Fr}_{\text{den}} \) for velocity \( V \) geometric length \( L \), heavier fluid density \( \rho_1 \), and lighter fluid density \( \rho_2 \). If desired, gravity can be specified as well. Depending on the application, this dimensionless number may be defined with the heavy phase or the light phase density in the numerator of the square root. For some applications, both need to be calculated. The default is to calculate with the heavy liquid density on top; set \( \text{heavy} \) to False to reverse this.

\[
Fr = \frac{V}{\sqrt{gL}} \sqrt{\frac{\rho_1 \rho_2}{\rho_1 - \rho_2}}
\]

Parameters

- \( V \) [float] Velocity of the specified phase, [m/s]
- \( L \) [float] Characteristic length, no typical definition [m]
- \( \rho_1 \) [float] Density of the heavier phase, [kg/m^3]
- \( \rho_2 \) [float] Density of the lighter phase, [kg/m^3]
- \( \text{heavy} \) [bool, optional] Whether or not the density used in the numerator is the heavy phase or the light phase, [-]
- \( g \) [float, optional] Acceleration due to gravity, [m/s^2]

Returns

- \( \text{Fr}_{\text{den}} \) [float] Densimetric Froude number, [-]

Notes

Many alternate definitions including density ratios have been used.

\[
Fr = \frac{\text{Inertial Force}}{\text{Gravity Force}}
\]

Where the gravity force is reduced by the relative densities of one fluid in another.

Note that an Exception will be raised if \( \rho_1 > \rho_2 \), as the square root becomes negative.

References

[1]

Examples

```python
>>> Froude_densimetric(1.83, L=2., rho1=800, rho2=1.2, g=9.81)
0.4134543386272418
>>> Froude_densimetric(1.83, L=2., rho1=800, rho2=1.2, g=9.81, heavy=False)
0.016013017679205096
```
fluids.core.Strouhal \((f, L, V)\)
Calculates Strouhal number \(St\) for a characteristic frequency \(f\), characteristic length \(L\), and velocity \(V\).

\[
St = \frac{fL}{V}
\]

**Parameters**
- \(f\) [float] Characteristic frequency, usually that of vortex shedding, [Hz]
- \(L\) [float] Characteristic length, [m]
- \(V\) [float] Velocity of the fluid, [m/s]

**Returns**
- \(St\) [float] Strouhal number, [-]

**Notes**
Sometimes abbreviated to \(S\) or \(Sr\).

\[
St = \frac{\text{Characteristic flow time}}{\text{Period of oscillation}}
\]

**References**
[1], [2]

**Examples**

```python
>>> Strouhal(8, 2., 4.)
4.0
```

fluids.core.Biot \((h, L, k)\)
Calculates Biot number \(Br\) for heat transfer coefficient \(h\), geometric length \(L\), and thermal conductivity \(k\).

\[
Bi = \frac{hL}{k}
\]

**Parameters**
- \(h\) [float] Heat transfer coefficient, [W/m^2/K]
- \(L\) [float] Characteristic length, no typical definition [m]
- \(k\) [float] Thermal conductivity, within the object [W/m/K]

**Returns**
- \(Bi\) [float] Biot number, [-]

**Notes**
Do not confuse \(k\), the thermal conductivity within the object, with that of the medium \(h\) is calculated with!

\[
Bi = \frac{\text{Surface thermal resistance}}{\text{Internal thermal resistance}}
\]
fluids.core.Stanton(h, V, rho, Cp)
Calculates Stanton number or $St$ for a specified heat transfer coefficient $h$, velocity $V$, density $rho$, and heat capacity $Cp$.

$$St = \frac{h}{V \rho C_p}$$

Parameters
- $h$ [float] Heat transfer coefficient, [W/m^2/K]
- $V$ [float] Velocity, [m/s]
- $rho$ [float] Density, [kg/m^3]
- $Cp$ [float] Heat capacity, [J/kg/K]

Returns
- $St$ [float] Stanton number []

Notes

$$St = \frac{Heat\ transfer\ coefficient}{Thermal\ capacity}$$

References
[1],[2]

Examples

```python
>>> Stanton(5000, 5, 800, 2000.)
0.000625
```
Fluids Documentation, Release 0.1

**rho** [float] Density of the fluid, [kg/m^3]

**V** [float] Velocity of fluid, [m/s]

**Returns**

**Eu** [float] Euler number []

**Notes**

Used in pressure drop calculations. Rarely, this number is divided by two. Named after Leonhard Euler applied calculus to fluid dynamics.

\[
Eu = \frac{\text{Pressure drop}}{\frac{1}{2} \cdot \text{velocity head}}
\]

**References**

[1], [2]

**Examples**

```python
>>> Euler(1E5, 1000., 4)
6.25
```

**fluids.core.Cavitation** *(P, Psat, rho, V)*

Calculates Cavitation number or *Ca* for a fluid of velocity *V* with a pressure *P*, vapor pressure *Psat*, and density *rho*.

\[
Ca = \sigma_c = \sigma = \frac{P - P_{sat}}{\frac{1}{2} \rho V^2}
\]

**Parameters**

- **P** [float] Internal pressure of the fluid, [Pa]
- **Psat** [float] Vapor pressure of the fluid, [Pa]
- **rho** [float] Density of the fluid, [kg/m^3]
- **V** [float] Velocity of fluid, [m/s]

**Returns**

**Ca** [float] Cavitation number []

**Notes**

Used in determining if a flow through a restriction will cavitate. Sometimes, the multiplication by 2 will be omitted;

\[
Ca = \frac{\text{Pressure - Vapor pressure}}{\text{Inertial pressure}}
\]

**References**

[1], [2]
Examples

```python
>>> Cavitation(2E5, 1E4, 1000, 10)
3.8
```

`fluids.core.Eckert(V, Cp, dT)`
Calculates Eckert number or $Ec$ for a fluid of velocity $V$ with a heat capacity $Cp$, between two temperature given as $dT$.

$$Ec = \frac{V^2}{Cp \Delta T}$$

Parameters

- $V$ [float] Velocity of fluid, [m/s]
- $Cp$ [float] Heat capacity of the fluid, [J/kg/K]
- $dT$ [float] Temperature difference, [K]

Returns

- $Ec$ [float] Eckert number []

Notes

Used in certain heat transfer calculations. Fairly rare.

$$Ec = \frac{\text{Kinetic energy}}{\text{Enthalpy difference}}$$

References

[1]

Examples

```python
>>> Eckert(10, 2000., 25.)
0.002
```

`fluids.core.Jakob(Cp, Hvap, Te)`
Calculates Jakob number or $Ja$ for a boiling fluid with sensible heat capacity $Cp$, enthalpy of vaporization $Hvap$, and boiling at $Te$ degrees above its saturation boiling point.

$$Ja = \frac{Cp \Delta T_e}{\Delta H_{vap}}$$

Parameters

- $Cp$ [float] Heat capacity of the fluid, [J/kg/K]
- $Hvap$ [float] Enthalpy of vaporization of the fluid at its saturation temperature [J/kg]
- $Te$ [float] Temperature difference above the fluid’s saturation boiling temperature, [K]

Returns

- $Ja$ [float] Jakob number []
Notes

Used in boiling heat transfer analysis. Fairly rare.

\[ Ja = \frac{\Delta\text{Sensible heat}}{\Delta\text{Latent heat}} \]

References

[1], [2]

Examples

```python
>>> Jakob(4000., 2E6, 10.)
0.02
```

*fluids.core.*`Power_number(P, L, N, rho)`

Calculates power number, Po, for an agitator applying a specified power P with a characteristic length L, rotational speed N, to a fluid with a specified density rho.

\[ P_o = \frac{P}{\rho N^3 D^5} \]

Parameters

- `P` [float] Power applied, [W]
- `L` [float] Characteristic length, typically agitator diameter [m]
- `N` [float] Speed [revolutions/second]
- `rho` [float] Density of fluid, [kg/m^3]

Returns

- `Po` [float] Power number []

Notes

Used in mixing calculations.

\[ P_o = \frac{\text{Power}}{\text{Rotational inertia}} \]

References

[1], [2]

Examples

```python
>>> Power_number(P=180, L=0.01, N=2.5, rho=800.)
144000000.0
```
**fluids.core.Stokes_number** (V, Dp, D, rhop, mu)

Calculates Stokes Number for a given characteristic velocity V, particle diameter Dp, characteristic diameter D, particle density rhop, and fluid viscosity mu.

\[
Stk = \frac{\rho_p V D_p^2}{18 \mu_f D}
\]

**Parameters**
- V [float] Characteristic velocity (often superficial), [m/s]
- Dp [float] Particle diameter, [m]
- D [float] Characteristic diameter (ex demister wire diameter or cyclone diameter), [m]
- rhop [float] Particle density, [kg/m^3]
- mu [float] Fluid viscosity, [Pa*s]

**Returns**
- Stk [float] Stokes numer, [-]

**Notes**
Used in droplet impaction or collection studies.

**References**
[1], [2]

**Examples**
```python
>>> Stokes_number(V=0.9, Dp=1E-5, D=1E-3, rhop=1000, mu=1E-5)
0.5
```

**fluids.core.Drag** (F, A, V, rho)

Calculates drag coefficient Cd for a given drag force F, projected area A, characteristic velocity V, and density rho.

\[
C_D = \frac{F_d}{A \cdot \frac{1}{2} \rho V^2}
\]

**Parameters**
- F [float] Drag force, [N]
- A [float] Projected area, [m^2]
- V [float] Characteristic velocity, [m/s]
- rho [float] Density, [kg/m^3]

**Returns**
- Cd [float] Drag coefficient, [-]

---

96 Chapter 2. API Reference
**Notes**

Used in flow around objects, or objects flowing within a fluid.

\[
C_D = \frac{\text{Drag forces}}{\text{Projected area} \cdot \text{Velocity head}}
\]

**References**

[1], [2]

**Examples**

```python
>>> Drag(1000, 0.0001, 5, 2000)
400.0
```

**fluids.core.Capillary(V, mu, sigma)**

Calculates Capillary number \(Ca\) for a characteristic velocity \(V\), viscosity \(mu\), and surface tension \(sigma\).

\[
Ca = \frac{V \mu}{\sigma}
\]

**Parameters**

- \(V\) [float] Characteristic velocity, [m/s]
- \(mu\) [float] Dynamic viscosity, [Pa*s]
- \(sigma\) [float] Surface tension, [N/m]

**Returns**

- \(Ca\) [float] Capillary number, [-]

**Notes**

Used in porous media calculations and film flow calculations. Surface tension may be gas-liquid, or liquid-liquid.

\[
Ca = \frac{\text{Viscous forces}}{\text{Surface forces}}
\]

**References**

[1], [2]

**Examples**

```python
>>> Capillary(1.2, 0.01, .1)
0.12
```

**fluids.core.Bejan_L(dP, L, mu, alpha)**

Calculates Bejan number of a length or \(Be\_L\) for a fluid with the given parameters flowing over a characteristic length \(L\) and experiencing a pressure drop \(dP\).

\[
Be_L = \frac{\Delta P L^2}{\mu \alpha}
\]
Parameters

\(dP\) [float] Pressure drop, [Pa]
\(L\) [float] Characteristic length, [m]
\(\mu\) [float, optional] Dynamic viscosity, [Pa*s]
\(\alpha\) [float] Thermal diffusivity, [m^2/s]

Returns

\(Be_L\) [float] Bejan number with respect to length [

Notes

Termed a dimensionless number by someone in 1988.

References

[1], [2]

Examples

```
>>> Bejan_L(1E4, 1, 1E-3, 1E-6)
10000000000000.0
```

```
fluids.core.Bejan_p(dP, K, \mu, \alpha)
```

Calculates Bejan number of a permeability or \(Be_p\) for a fluid with the given parameters and a permeability \(K\) experiencing a pressure drop \(dP\).

\[
Be_p = \frac{\Delta PK}{\mu \alpha}
\]

Parameters

\(dP\) [float] Pressure drop, [Pa]
\(K\) [float] Permeability, [m^2]
\(\mu\) [float, optional] Dynamic viscosity, [Pa*s]
\(\alpha\) [float] Thermal diffusivity, [m^2/s]

Returns

\(Be_p\) [float] Bejan number with respect to pore characteristics [

Notes

Termed a dimensionless number by someone in 1988.

References

[1], [2]
Examples

```python
>>> Bejan_p(1E4, 1, 1E-3, 1E-6)
10000000000000.0
```

`fluids.core.Boiling(G, q, Hvap)`
Calculates Boiling number or $B_g$ using heat flux, two-phase mass flux, and heat of vaporization of the fluid flowing. Used in two-phase heat transfer calculations.

$$B_g = \frac{q}{G \Delta H_{\text{vap}}}$$

Parameters

- **G** [float] Two-phase mass flux in a channel (combined liquid and vapor) [kg/m$^2$/s]
- **q** [float] Heat flux [W/m$^2$]
- **Hvap** [float] Heat of vaporization of the fluid [J/kg]

Returns

- **B_g** [float] Boiling number [-]

Notes

Most often uses the symbol $B_o$ instead of $B_g$, but this conflicts with Bond number.

$$B_g = \frac{\text{mass liquid evaporated}}{\text{area heat transfer surface}} \div \frac{\text{mass flow rate fluid}}{\text{flow cross sectional area}}$$

First defined in [4], though not named.

References

[1], [2], [3], [4]

Examples

```python
>>> Boiling(300, 3000, 800000)
1.25e-05
```

`fluids.core.Confinement(D, rhol, rhog, sigma, g=9.80665)`
Calculates Confinement number or $C_o$ for a fluid in a channel of diameter $D$ with liquid and gas densities $\rho_{\text{liq}}$ and $\rho_{\text{gas}}$ and surface tension $\sigma$, under the influence of gravitational force $g$.

$$C_o = \left[ \frac{\sigma}{g(\rho_{\text{liq}} - \rho_{\text{gas}})} \right]^{0.5} \div \frac{D}{D}$$

Parameters

- **D** [float] Diameter of channel, [m]
- **rhol** [float] Density of liquid phase, [kg/m$^3$]
- **rhog** [float] Density of gas phase, [kg/m$^3$]
- **sigma** [float] Surface tension between liquid-gas phase, [N/m]
Fluids Documentation, Release 0.1

g [float, optional] Acceleration due to gravity, [m/s^2]

Returns

Co [float] Confinement number [-]

Notes

Used in two-phase pressure drop and heat transfer correlations. First used in [1] according to [3].

\[ Co = \frac{\text{surface tension force}}{\text{buoyancy force}} \div \frac{\text{Channel area}}{\text{Channel area}} \]

References

[1], [2], [3]

Examples

```python
>>> Confinement(0.001, 1077, 76.5, 4.27E-3)
0.6596978265315191
```

fluids.core.Archimedes(L, rhof, rhop, mu, g=9.80665)

Calculates Archimedes number, \( Ar \), for a fluid and particle with the given densities, characteristic length, viscosity, and gravity (usually diameter of particle).

\[ Ar = \frac{L^3 \rho_f (\rho_p - \rho_f) g}{\mu^2} \]

Parameters

- L [float] Characteristic length, typically particle diameter [m]
- rhof [float] Density of fluid, [kg/m^3]
- rhop [float] Density of particle, [kg/m^3]
- mu [float] Viscosity of fluid, [N/m]
- g [float, optional] Acceleration due to gravity, [m/s^2]

Returns

- Ar [float] Archimedes number []

Notes

Used in fluid-particle interaction calculations.

\[ Ar = \frac{\text{Gravitational force}}{\text{Viscous force}} \]

References

[1], [2]
Examples

```python
>>> Archimedes(0.002, 2., 3000, 1E-3)
470.4053872
```

**fluids.core.Ohnesorge** *(L, rho, mu, sigma)*

Calculates Ohnesorge number, *Oh*, for a fluid with the given characteristic length, density, viscosity, and surface tension.

\[
Oh = \frac{\mu}{\sqrt{\rho \sigma L}}
\]

**Parameters**

- **L** [float] Characteristic length [m]
- **rho** [float] Density of fluid, [kg/m^3]
- **mu** [float] Viscosity of fluid, [Pa*s]
- **sigma** [float] Surface tension, [N/m]

**Returns**

- **Oh** [float] Ohnesorge number

**Notes**

Often used in spray calculations. Sometimes given the symbol Z.

\[
Oh = \frac{\sqrt{\frac{We}{Re}}}{\sqrt{\text{Inertia} \cdot \text{Surface tension}}}
\]

**References**

[1]

**Examples**

```python
>>> Ohnesorge(1E-4, 1000., 1E-3, 1E-1)
0.01
```

**fluids.core.Suratman** *(L, rho, mu, sigma)*

Calculates Suratman number, *Su*, for a fluid with the given characteristic length, density, viscosity, and surface tension.

\[
Su = \frac{\rho \sigma L}{\mu^2}
\]

**Parameters**

- **L** [float] Characteristic length [m]
- **rho** [float] Density of fluid, [kg/m^3]
- **mu** [float] Viscosity of fluid, [Pa*s]
- **sigma** [float] Surface tension, [N/m]
Returns

Su [float] Suratman number []

Notes

Also known as Laplace number. Used in two-phase flow, especially the bubbly-slug regime. No confusion regarding the definition of this group has been observed.

\[ Su = \frac{Re^2}{We} = \frac{\text{Inertia} \cdot \text{Surface tension}}{(\text{viscous forces})^2} \]

The oldest reference to this group found by the author is in 1963, from [2].

References

[1], [2]

Examples

```python
>>> Suratman(1E-4, 1000., 1E-3, 1E-1)
10000.0
```

`fluids.core.Hagen(Re, fd)`
Calculates Hagen number, \( Hg \), for a fluid with the given Reynolds number and friction factor.

\[ Hg = \frac{f_d Re^2}{2} = \frac{1}{\rho} \frac{\Delta P D^3}{\Delta z \nu^2} = \frac{\rho \Delta P D^3}{\mu^2 \Delta z} \]

Parameters

- Re [float] Reynolds number [-]
- fd [float, optional] Darcy friction factor, [-]

Returns

- Hg [float] Hagen number, [-]

Notes

Introduced in [1]; further use of it is mostly of the correlations introduced in [1]. Notable for use in correlations, because it does not have any dependence on velocity. This expression is useful when designing backwards with a pressure drop spec already known.

References

[1], [2], [3]
Examples

Example from [3]:

```python
>>> Hagen(Re=2610, fd=1.935235)
6591507.17175
```

```python
fluids.core.thermal_diffusivity(k, rho, Cp)
Calculates thermal diffusivity or alpha for a fluid with the given parameters.

\[ \alpha = \frac{k}{\rho C_p} \]

Parameters

- `k` [float] Thermal conductivity, [W/m/K]
- `rho` [float] Density, [kg/m^3]
- `Cp` [float] Heat capacity, [J/kg/K]

Returns

- `alpha` [float] Thermal diffusivity, [m^2/s]

References

[1]

Examples

```python
>>> thermal_diffusivity(k=0.02, rho=1., Cp=1000.)
2e-05
```

```python
fluids.core.c_ideal_gas(T, k, MW)
Calculates speed of sound `c` in an ideal gas at temperature `T`.

\[ c = \sqrt{k R_{specific} T} \]

Parameters

- `T` [float] Temperature of fluid, [K]
- `k` [float] Isentropic exponent of fluid, [-]
- `MW` [float] Molecular weight of fluid, [g/mol]

Returns

- `c` [float] Speed of sound in fluid, [m/s]

Notes

Used in compressible flow calculations. Note that the gas constant used is the specific gas constant:

\[ R_{specific} = \frac{R}{MW} \]
**References**

[1], [2]

**Examples**

```python
>>> c_ideal_gas(T=303, k=1.4, MW=28.96)
348.9820953185441
```

`fluids.core.relative_roughness(D, roughness=1.52e-06)`

Calculates relative roughness $eD$ using a diameter and the roughness of the material of the wall. Default roughness is that of steel.

$$
eD = \frac{\varepsilon}{D}
$$

**Parameters**

- $D$ [float] Diameter of pipe, [m]
- `roughness` [float, optional] Roughness of pipe wall [m]

**Returns**

- $eD$ [float] Relative Roughness, [-]

**References**

[1], [2]

**Examples**

```python
>>> relative_roughness(0.5, 1E-4)
0.0002
```

`fluids.core.nu_mu_converter(rho, mu=None, nu=None)`

Calculates either kinematic or dynamic viscosity, depending on inputs. Used when one type of viscosity is known as well as density, to obtain the other type. Raises an error if both types of viscosity or neither type of viscosity is provided.

$$
\nu = \frac{\mu}{\rho}
$$

$$
\mu = \nu \rho
$$

**Parameters**

- `rho` [float] Density, [kg/m^3]
- `mu` [float, optional] Dynamic viscosity, [Pa*s]
- `nu` [float, optional] Kinematic viscosity, [m^2/s]

**Returns**

- `mu` or `nu` [float] Dynamic viscosity, Pa*s or Kinematic viscosity, m^2/s
References

[1]

Examples

```python
>>> nu_mu_converter(998., nu=1.0E-6)
0.000998
```

```python
fluids.core.gravity(latitude, H)
Calculates local acceleration due to gravity $g$ according to [1]. Uses latitude and height to calculate $g$.

$$g = 9.780356(1 + 0.0052885 \sin^2 \phi - 0.00000592 \phi^2) - 3.086 \times 10^{-6} H$$

Parameters

latitude [float] Degrees, [degrees]

H [float] Height above earth’s surface [m]

Returns

$g$ [float] Acceleration due to gravity, [m/s^2]

Notes

Better models, such as EGM2008 exist.

References

[1]

Examples

```python
>>> gravity(55, 1E4)
9.784151976863571
```

```python
fluids.core.K_from_f(fd, L, D)
Calculates loss coefficient, $K$, for a given section of pipe at a specified friction factor.

$$K = f_d L / D$$

Parameters

fd [float] friction factor of pipe, []

L [float] Length of pipe, [m]

D [float] Inner diameter of pipe, [m]

Returns

$K$ [float] Loss coefficient, []
Notes

For fittings with a specified L/D ratio, use D = 1 and set L to specified L/D ratio.

Examples

```python
>>> K_from_f(fd=0.018, L=100., D=.3)
6.0
```

```python
fluids.core.K_from_L_equiv(L_D=240, fd=0.015)
```

Calculates loss coefficient, for a given equivalent length (L/D).

\[ K = f_d \frac{L}{D} \]

Parameters

- **L_D** [float] Length over diameter, []
- **fd** [float, optional] Darcy friction factor, [-]

Returns

- **K** [float] Loss coefficient, []

Notes

Almost identical to `K_from_f`, but with a default friction factor for fully turbulent flow in steel pipes.

Examples

```python
>>> K_from_L_equiv(240)
3.6
```

```python
fluids.core.L_equiv_from_K(K=3.6, fd=0.015)
```

Calculates equivalent length of pipe (L/D), for a given loss coefficient.

\[ \frac{L}{D} = \frac{K}{f_d} \]

Parameters

- **K** [float] Loss coefficient, [-]
- **fd** [float, optional] Darcy friction factor, [-]

Returns

- **L_D** [float] Length over diameter, [-]

Notes

Assumes a default friction factor for fully turbulent flow in steel pipes.
Examples

```python
>>> L_equiv_from_K(3.6)
240.00000000000003
```

`fluids.core.L_from_K(K, D, fd=0.015)`
Calculates the length of straight pipe at a specified friction factor required to produce a given loss coefficient \( K \).

\[
L = \frac{K D}{f_d}
\]

Parameters

- **K** [float] Loss coefficient, \([\text{]}\]
- **D** [float] Inner diameter of pipe, \([\text{m}]\)
- **fd** [float] friction factor of pipe, \([\text{]}\)

Returns

- **L** [float] Length of pipe, \([\text{m}]\)

Examples

```python
>>> L_from_K(K=6, D=.3, fd=0.018)
100.0
```

`fluids.core.dP_from_K(K, rho, V)`
Calculates pressure drop, for a given loss coefficient, at a specified density and velocity.

\[
dP = 0.5K \rho V^2
\]

Parameters

- **K** [float] Loss coefficient, \([\text{]}\]
- **rho** [float] Density of fluid, \([\text{kg/m}^3]\)
- **V** [float] Velocity of fluid in pipe, \([\text{m/s}]\)

Returns

- **dP** [float] Pressure drop, \([\text{Pa}]\)

Notes

Loss coefficient \( K \) is usually the sum of several factors, including the friction factor.

Examples

```python
>>> dP_from_K(K=10, rho=1000, V=3)
45000.0
```

`fluids.core.head_from_K(K, V, g=9.80665)`
Calculates head loss, for a given loss coefficient, at a specified velocity.

\[
\text{head} = \frac{KV^2}{2g}
\]
Parameters

- **K** [float] Loss coefficient, []
- **V** [float] Velocity of fluid in pipe, [m/s]
- **g** [float, optional] Acceleration due to gravity, [m/s²]

Returns

- **head** [float] Head loss, [m]

Notes

Loss coefficient *K* is usually the sum of several factors, including the friction factor.

Examples

```python
>>> head_from_K(K=10, V=1.5)
1.1471807396001694
```

`fluids.core.head_from_P(P, rho, g=9.80665)`
Calculates head for a fluid of specified density at specified pressure.

\[
\text{head} = \frac{P}{\rho g}
\]

Parameters

- **P** [float] Pressure fluid in pipe, [Pa]
- **rho** [float] Density of fluid, [kg/m³]
- **g** [float, optional] Acceleration due to gravity, [m/s²]

Returns

- **head** [float] Head, [m]

Notes

By definition. Head varies with location, inversely proportional to the increase in gravitational constant.

Examples

```python
>>> head_from_P(P=98066.5, rho=1000)
10.000000000000002
```

`fluids.core.P_from_head(head, rho, g=9.80665)`
Calculates head for a fluid of specified density at specified pressure.

\[
P = \rho g \cdot \text{head}
\]

Parameters

- **head** [float] Head, [m]
- **rho** [float] Density of fluid, [kg/m³]
**g** [float, optional] Acceleration due to gravity, [m/s^2]

**Returns**

**P** [float] Pressure fluid in pipe, [Pa]

**Examples**

```python
>>> P_from_head(head=5., rho=800.)
39226.6
```

**fluids.core.Eotvos** *(rhol, rhog, sigma, L)*

Calculates Bond number, *Bo* also known as Eotvos number, for a fluid with the given liquid and gas densities, surface tension, and geometric parameter (usually length).

\[
Bo = \frac{g(\rho_l - \rho_g)L^2}{\sigma}
\]

**Parameters**

- **rhol** [float] Density of liquid, [kg/m^3]
- **rhog** [float] Density of gas, [kg/m^3]
- **sigma** [float] Surface tension, [N/m]
- **L** [float] Characteristic length, [m]

**Returns**

**Bo** [float] Bond number []

**References**

[1]

**Examples**

```python
>>> Bond(1000., 1.2, .0589, 2)
665187.2339558573
```

### 2.5 Drag and terminal velocity (fluids.drag)

**fluids.drag.drag_sphere** *(Re, Method=None, AvailableMethods=False)*

This function handles calculation of drag coefficient on spheres. Twenty methods are available, all requiring only the Reynolds number of the sphere. Most methods are valid from Re=0 to Re=200,000. A correlation will be automatically selected if none is specified. The full list of correlations valid for a given Reynolds number can be obtained with the `AvailableMethods` flag.

If no correlation is selected, the following rules are used:

- If Re < 0.01, use Stoke’s solution.
- If 0.01 <= Re < 0.1, linearly combine ‘Barati’ with Stokes’s solution such that at Re = 0.1 the solution is ‘Barati’, and at Re = 0.01 the solution is ‘Stokes’.
• If 0.1 <= Re <= ~212963, use the ‘Barati’ solution.
• If ~212963 < Re <= 1E6, use the ‘Barati_high’ solution.
• For Re > 1E6, raises an exception; no valid results have been found.

Parameters

Re [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

Cd [float] Drag coefficient [-]

methods [list, only returned if AvailableMethods == True] List of methods which can be used to calculate Cd with the given Re

Other Parameters

Method [string, optional] A string of the function name to use, as in the dictionary drag_sphere_correlations

AvailableMethods [bool, optional] If True, function will consider which methods which can be used to calculate Cd with the given Re

Examples

```python
>>> drag_sphere(200)
0.7682237950389874
```

`fluids.drag.v_terminal(D, rhop, rho, mu, Method=None)`

Calculates terminal velocity of a falling sphere using any drag coefficient method supported by `drag_sphere`. The laminar solution for Re < 0.01 is first tried; if the resulting terminal velocity does not put it in the laminar regime, a numerical solution is used.

\[
v_t = \sqrt{\frac{4gd_p(\rho_p - \rho_f)}{3C_D\rho_f}}
\]

Parameters

D [float] Diameter of the sphere, [m]

rhop [float] Particle density, [kg/m^3]

rho [float] Density of the surrounding fluid, [kg/m^3]

mu [float] Viscosity of the surrounding fluid [Pa*s]

Method [string, optional] A string of the function name to use, as in the dictionary drag_sphere_correlations

Returns

v_t [float] Terminal velocity of falling sphere [m/s]

Notes

As there are no correlations implemented for Re > 1E6, an error will be raised if the numerical solver seeks a solution above that limit.
The laminar solution is given in [1] and is:

\[ v_t = \frac{gD^2(\rho_p - \rho_f)}{18\mu_f} \]

References

[1], [2]

Examples

```python
>>> v_terminal(D=70E-6, rhop=2600., rho=1000., mu=1E-3)
0.004142497244531304
```

Example 7-1 in GPSA handbook, 13th edition:

```python
>>> from scipy.constants import *

>>> v_terminal(D=150E-6, rhop=31.2*lb/foot**3, rho=2.07*lb/foot**3, mu=1.2e-05)/foot
0.4491992020345101
```

The answer reported there is 0.46 ft/sec.

`fluids.drag.integrate_drag_sphere` *(D, rhop, rho, mu, t, V=0, Method=None, distance=False)*

Integrates the velocity and distance traveled by a particle moving at a speed which will converge to its terminal velocity.

Performs an integration of the following expression for acceleration:

\[ a = \frac{g(\rho_p - \rho_f)}{\rho_p} - \frac{3C_D\rho_f u^2}{4D\rho_p} \]

Parameters

- **D** [float] Diameter of the sphere, [m]
- **rhop** [float] Particle density, [kg/m^3]
- **rho** [float] Density of the surrounding fluid, [kg/m^3]
- **mu** [float] Viscosity of the surrounding fluid [Pa*s]
- **t** [float] Time to integrate the particle to, [s]
- **V** [float] Initial velocity of the particle, [m/s]
- **Method** [string, optional] A string of the function name to use, as in the dictionary `drag_sphere_correlations`
- **distance** [bool, optional] Whether or not to calculate the distance traveled and return it as well

Returns

- **v** [float] Velocity of falling sphere after time \( t \) [m/s]
- **x** [float, returned only if `distance == True`] Distance traveled by the falling sphere in time \( t \), [m]
Notes

This can be relatively slow as drag correlations can be complex.

There are analytical solutions available for the Stokes law regime \((\text{Re} < 0.3)\). They were obtained from Wolfram Alpha. [1] was not used in the derivation, but also describes the derivation fully.

\[
V(t) = \frac{\exp(-at)(V_0a + b(\exp(at) - 1))}{a}
\]

\[
x(t) = \frac{\exp(-at)[V_0a(\exp(at) - 1) + b \exp(at)(at - 1) + b]}{a^2}
\]

\[
a = \frac{18 \mu_f}{D^2 \rho_p}
\]

\[
b = \frac{g(\rho_p - \rho_f)}{\rho_p}
\]

The analytical solution will automatically be used if the initial and terminal velocity is shown to show the particle’s behavior to be laminar. Note that this behavior requires that the terminal velocity of the particle be solved for - this adds slight (1%) overhead for the cases where particles are not laminar.

References

[1]

Examples

```python
>>> integrate_drag_sphere(D=0.001, rhop=2200., rho=1.2, mu=1.78E-5, t=0.5,
... V=30, distance=True)
(9.686465044053476, 7.8294546436299175)
```

`fluids.drag.time_v_terminal_Stokes(D, rhop, rho, mu, V0, tol=1e-14)`

Calculates the time required for a particle in Stoke’s regime only to reach terminal velocity (approximately). An infinitely long period is required theoretically, but with floating points, it is possible to calculate the time required to come within a specified \(\text{tol}\) of that terminal velocity.

\[
t_{\text{term}} = -\frac{1}{18\mu} \ln \left( \frac{D^2 g \rho_p - D^2 g \rho_p + 18 \mu V_{\text{term}}}{D^2 g \rho_p - D^2 g \rho_p + 18 \mu V_0} \right) D^2 \rho_p
\]

Parameters

- \(D\) [float] Diameter of the sphere, [m]
- \(\text{rhop}\) [float] Particle density, [kg/m^3]
- \(\text{rho}\) [float] Density of the surrounding fluid, [kg/m^3]
- \(\text{mu}\) [float] Viscosity of the surrounding fluid [Pa*s]
- \(V0\) [float] Initial velocity of the particle, [m/s]
- \(\text{tol}\) [float, optional] How closely to approach the terminal velocity - the target velocity is the terminal velocity multiplied by 1 (+/-) this, depending on if the particle is accelerating or decelerating, [-]

Returns

- \(t\) [float] Time for the particle to reach the terminal velocity to within the specified or an achievable tolerance, [s]
Notes

The symbolic solution was obtained via Wolfram Alpha.

If a solution cannot be obtained due to floating point error at very high tolerance, an exception is raised - but first, the tolerance is doubled, up to fifty times in an attempt to obtain the highest possible precision while still giving an answer. If at any point the tolerance is larger than 1%, an exception is also raised.

Examples

```python
>>> time_v_terminal_Stokes(D=1e-7, rhop=2200., rho=1.2, mu=1.78E-5, V0=1)
3.188003113787153e-06

>>> time_v_terminal_Stokes(D=1e-2, rhop=2200., rho=1.2, mu=1.78E-5, V0=1, ...
... tol=1e-30)
24800.636391802
```

`fluids.drag.Stokes(Re)`

Calculates drag coefficient of a smooth sphere using Stoke’s law.

\[ C_D = \frac{24}{Re} \]

Parameters

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

- **Cd** [float] Drag coefficient [-]

Notes

Range is Re <= 0.3

References

[1]

Examples

```python
>>> Stokes(0.1)
240.0
```

`fluids.drag.Barati(Re)`

Calculates drag coefficient of a smooth sphere using the method in [1].

\[ C_D = 5.4856 \times 10^9 \tanh(4.3774 \times 10^{-9}/Re) + 0.0709 \tanh(700.6574/Re) + 0.3894 \tanh(74.1539/Re) - 0.1198 \tanh(7429.0843/Re) + 1.7174 \tanh[9.9851/(Re + 2.3384)] + 0.4744 \]

Parameters

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

- **Cd** [float] Drag coefficient [-]
Notes

Range is $Re \leq 2E5$

References

[1]

Examples

Matching example in [1], in a table of calculated values.

```python
>>> Barati(200.)
0.7682237950389874
```

`fluids.drag.Barati_high(Re)`

Calculates drag coefficient of a smooth sphere using the method in [1].

\[
C_D = 8 \times 10^{-6} \left[ (Re/6530)^2 + \tanh(Re) - 8 \ln(Re)/\ln(10) \right] - 0.4119 \exp(-2.08 \times 10^{43}/[Re + Re^2]^4) - 2.1344 \exp(-\{\ln(Re^2 + 10.7563)/\ln(10)\}^2 + 9.9867) / Re + 2.4795 - 8.5 \times 10^{-3} \{2 \ln[\tanh(\tanh(Re))] / \ln(10) - 28.25.7162 \} / Re + 2.4795
\]

Parameters

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

- **Cd** [float] Drag coefficient [-]

Notes

Range is $Re \leq 1E6$ This model is the wider-range model the authors developed. At sufficiently low diameters or $Re$ values, drag is no longer a phenomena.

References

[1]

Examples

Matching example in [1], in a table of calculated values.

```python
>>> Barati_high(200.)
0.7730544082789523
```

`fluids.drag.Rouse(Re)`

Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \frac{24}{Re} + \frac{3}{Re^{0.5}} + 0.34
\]

Parameters

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]
Fluids Documentation, Release 0.1

Returns

Cd [float] Drag coefficient [-]

Notes

Range is Re <= 2E5

References

[1], [2]

Examples

```python
>>> Rouse(200.)
0.6721320343559642
```

`fluids.drag.Engelund_Hansen(Re)`

Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[ C_D = \frac{24}{Re} + 1.5 \]

Parameters

Re [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

Cd [float] Drag coefficient [-]

Notes

Range is Re <= 2E5

References

[1], [2]

Examples

```python
>>> Engelund_Hansen(200.)
1.62
```

`fluids.drag.Clift_Gauvin(Re)`

Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[ C_D = \frac{24}{Re} (1 + 0.152Re^{0.677}) + \frac{0.417}{1 + 5070Re^{-0.94}} \]

Parameters
**Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

**Returns**

**Cd** [float] Drag coefficient [-]

**Notes**

Range is Re <= 2E5

**References**

[1], [2]

**Examples**

```python
>>> Clift_Gauvin(200.)
0.7905400398000133
```

`fluids.drag.Morsi_Alexander(Re)`

Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \begin{cases}
\frac{24}{Re} + \frac{0.9903}{Re^{0.8889}} + 3.69 & \text{if } Re < 0.1 \\
\frac{29.1667}{Re} - \frac{3.8889}{Re^{2}} + 1.2220 & \text{if } 0.1 < Re < 1 \\
\frac{46.5}{Re} - \frac{116.67}{Re^{2}} + 0.6167 & \text{if } 1 < Re < 10 \\
\frac{98.33}{Re} - \frac{2778}{Re^{2}} + 0.3644 & \text{if } 10 < Re < 100 \\
\frac{146.62}{Re} - \frac{4.75 \times 10^{4}}{Re^{2}} + 0.3570 & \text{if } 100 < Re < 5000 \\
\frac{290.5460}{Re} - \frac{57.87 \times 10^{4}}{Re^{2}} + 0.46 & \text{if } 5000 < Re < 10000 \\
\frac{-1662.5}{Re} + \frac{5.4167 \times 10^{6}}{Re^{2}} + 0.5191 & \text{if } 10000 < Re < 50000
\end{cases}
\]

**Parameters**

**Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

**Returns**

**Cd** [float] Drag coefficient [-]

**Notes**

Range is Re <= 2E5. Original was reviewed, and confirmed to contain the cited equations.

**References**

[1], [2]

**Examples**
fluids.drag.Graf(Re)
Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \frac{24}{Re} + \frac{7.3}{1 + Re^{0.5}} + 0.25
\]

Parameters
- Re [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns
- Cd [float] Drag coefficient [-]

Notes
Range is Re <= 2E5

References
[1], [2]

Examples

```python
>>> Graf(200.)
0.8520984424785725
```

fluids.drag.Flemmer_Banks(Re)
Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \frac{24}{Re^{10^E}}
E = 0.383Re^{0.356} - 0.207Re^{0.396} - \frac{0.143}{1 + (\log_{10} Re)^2}
\]

Parameters
- Re [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns
- Cd [float] Drag coefficient [-]

Notes
Range is Re <= 2E5

References
[1], [2]
Fluids Documentation, Release 0.1

Examples

```python
>>> Flemmer_Banks(200.)
0.7849169609270039
```

`fluids.drag.Khan_Richardson(Re)`
Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[ C_D = (2.49Re^{-0.328} + 0.34Re^{0.067})^{3.18} \]

Parameters

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity. [-]

Returns

- **Cd** [float] Drag coefficient [-]

Notes

Range is Re <= 2E5

References

[1],[2]

Examples

```python
>>> Khan_Richardson(200.)
0.7747572379211097
```

`fluids.drag.Swamee_Ojha(Re)`
Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[ C_D = 0.5 \left\{ 16 \left[ \frac{24}{Re} \right]^{1.6} + \left( \frac{130}{Re} \right)^{0.72} \right\}^{2.5} + \left[ \left( \frac{40000}{Re} \right)^2 + 1 \right]^{-0.25} \right\}^{0.25} \]

Parameters

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity. [-]

Returns

- **Cd** [float] Drag coefficient [-]

Notes

Range is Re <= 1.5E5

References

[1],[2]
Examples

```python
>>> Swamee_Ojha(200.)
0.8490012397545713
```

**fluids.drag.Yen** *(Re)*

Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \frac{24}{Re} \left( 1 + 0.15 \sqrt{Re} + 0.017 Re \right) - \frac{0.208}{1 + 10^4 Re^{-0.5}}
\]

**Parameters**

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

**Returns**

- **Cd** [float] Drag coefficient [-]

**Notes**

Range is Re <= 2E5

**References**

[1], [2]

Examples

```python
>>> Yen(200.)
0.7822647002187014
```

**fluids.drag.Haider_Levenspiel** *(Re)*

Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \frac{24}{Re} \left( 1 + 0.1806 Re^{0.6459} \right) + \left( \frac{0.4251}{1 + 0.85 Re^{0.35}} \right)
\]

**Parameters**

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

**Returns**

- **Cd** [float] Drag coefficient [-]

**Notes**

Range is Re <= 2E5 An improved version of this correlation is in Brown and Lawler.

**References**

[1], [2]
Examples

```python
>>> Haider_Levenspiel(200.)
0.7959551680251666
```

`fluids.drag.Cheng(Re)`
Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \frac{24}{Re} \left(1 + 0.27Re^{0.43} + 0.47[1 - \exp(-0.04Re^{0.38})]\right)
\]

Parameters

- Re [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

- Cd [float] Drag coefficient [-]

Notes

Range is Re \( \leq 2 \times 10^5 \)

References

[1],[2]

Examples

```python
>>> Cheng(200.)
0.7939143028294227
```

`fluids.drag.Terfous(Re)`
Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = 2.689 + \frac{21.683}{Re} + \frac{0.131}{Re^2} - \frac{10.616}{Re^{0.1}} + \frac{12.216}{Re^{0.2}}
\]

Parameters

- Re [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

- Cd [float] Drag coefficient [-]

Notes

Range is \( 0.1 < Re \leq 5 \times 10^4 \)

References

[1],[2]
Examples

```python
>>> Terfous(200.)
0.7814651149769638
```

`fluids.drag.Mikhailov_Freire(Re)`
Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \frac{3808[(1617933/2030) + (178861/1063)Re + (1219/1084)Re^2]}{681Re[(77531/422) + (13529/976)Re - (1/71154)Re^2]}
\]

Parameters

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

- **Cd** [float] Drag coefficient [-]

Notes

Range is Re \(\leq 118300\)

References

[1], [2]

Examples

```python
>>> Mikhailov_Freire(200.)
0.7514111388018659
```

`fluids.drag.Clift(Re)`
Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \begin{cases} 
\frac{24}{Re^3} + \frac{3}{Re} & \text{if } Re < 0.01 \\
\frac{24}{Re^3}(1 + 0.1315Re^{0.82 - 0.05\log Re}) & \text{if } 0.01 < Re < 20 \\
\frac{24}{Re^3}(1 + 0.1935Re^{0.6305}) & \text{if } 20 < Re < 260 \\
10^{1.6435 - 1.1242\log Re + 0.1558[\log Re]^2} & \text{if } 260 < Re < 1500 \\
10^{-2.4571 + 2.5558\log Re - 0.9295[\log Re]^2 + 0.1049[\log Re]^3} & \text{if } 1500 < Re < 12000 \\
10^{-1.9181 + 0.6370\log Re - 0.0636[\log Re]^2} & \text{if } 12000 < Re < 44000 \\
10^{-4.3390 + 1.5809\log Re - 0.1546[\log Re]^2} & \text{if } 44000 < Re < 338000 \\
9.78 - 5.3\log Re & \text{if } 338000 < Re < 400000 \\
0.19\log Re - 0.49 & \text{if } 400000 < Re < 1000000 
\end{cases}
\]

Parameters

- **Re** [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

- **Cd** [float] Drag coefficient [-]
Notes

Range is \( Re \leq 1E6 \).

References

[1], [2]

Examples

```python
>>> Ceylan(200)
0.7756342422322543
```

```python
fluids.drag.Ceylan(Re)
```

Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = 1 - 0.5 \exp(0.182) + 10.11Re^{-2/3} \exp(0.952Re^{-1/4}) - 0.03859Re^{-4/3} \exp(1.30Re^{-1/2}) + 0.037 \times 10^{-4} Re \exp(-0.116 \times 10^{-10} Re^2)
\]

Parameters

- \( Re \) [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

- \( Cd \) [float] Drag coefficient [-]

Notes

Range is 0.1 < \( Re \) <= 1E6 Original article reviewed.

References

[1], [2]

Examples

```python
>>> Ceylan(200.)
0.7816735980280175
```

```python
fluids.drag.Almedeij(Re)
```

Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \left\{ \frac{1}{(\phi_1 + \phi_2)^{-1} + (\phi_3)^{-1} + \phi_4} \right\}^{0.1}
\]

\[
\phi_1 = (24Re^{-1})^{10} + (21Re^{-0.67})^{10} + (4Re^{-0.33})^{10} + 0.4^{10}
\]

\[
\phi_2 = (0.148Re^{0.11})^{-10} + (0.5)^{-10}
\]

\[
\phi_3 = (1.57 \times 10^8 Re^{-1.625})^{10}
\]

\[
\phi_4 = (6 \times 10^{-17} Re^{2.63})^{-10} + (0.2)^{-10}
\]
Parameters

\( \text{Re} \) [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

\( \text{Cd} \) [float] Drag coefficient [-]

Notes

Range is \( \text{Re} \leq 1 \times 10^6 \). Original work has been reviewed.

References

[1], [2]

Examples

```python
>>> Almedej(200.)
0.7114768646813396
```

\texttt{fluids.drag.Morrison(Re)}

Calculates drag coefficient of a smooth sphere using the method in [1] as described in [2].

\[
C_D = \frac{24}{Re} + \frac{2.6Re/5}{1 + \left(\frac{Re}{5}\right)^{1.52}} + \frac{0.411 \left(\frac{Re}{263000}\right)^{-7.94}}{1 + \left(\frac{Re}{263000}\right)^{-8}} + \frac{Re^{0.8}}{461000}
\]

Parameters

\( \text{Re} \) [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]

Returns

\( \text{Cd} \) [float] Drag coefficient [-]

Notes

Range is \( \text{Re} \leq 1 \times 10^6 \).

References

[1], [2]

Examples

```python
>>> Morrison(200.)
0.767731559965325
```
**fluids.drag.** `Song_Xu(Re, sphericity=1.0, S=1.0)`

Calculates drag coefficient of a particle using the method in [1]. Developed with data for spheres, cubes, and cylinders. Claims 3.52% relative error for 0.001 < Re < 100 based on 336 tests data.

\[
C_d = \frac{24}{Re^{0.65}S^{0.3}} (1 + 0.35Re)^{0.44}
\]

**Parameters**

- `Re` [float] Particle Reynolds number of the sphere using the surrounding fluid density and viscosity, [-]
- `sphericity` [float, optional] Sphericity of the particle
- `S` [float, optional] Ratio of equivalent sphere area and the projected area in the particle settling direction [-]

**Returns**

- `Cd` [float] Drag coefficient of particle [-]

**Notes**

Notable as its experimental data and analysis is included in their supporting material.

**References**

[1]

**Examples**

```python
>>> Song_Xu(30.)
2.3431335190092444
```

### 2.6 Filter pressure drop (fluids.filters)

**fluids.filters.** `round_edge_screen(alpha, Re, angle=0)`

Returns the loss coefficient for a round edged wire screen or bar screen, as shown in [1]. Angle of inclination may be specified as well.

**Parameters**

- `alpha` [float] Fraction of screen open to flow [-]
- `Re` [float] Reynolds number of flow through screen with D = space between rods, []
- `angle` [float, optional] Angle of inclination, with 0 being straight and 90 being parallel to flow [degrees]

**Returns**

- `K` [float] Loss coefficient [-]
Notes

Linear interpolation between a table of values. Re table extends from 20 to 400, with constant values outside of
the table. This behavior should be adequate. alpha should be between 0.05 and 0.8. If angle is over 85 degrees,
the value at 85 degrees is used.

The velocity the loss coefficient relates to is the approach velocity before the screen.

References

[1]

Examples

```python
>>> round_edge_screen(0.5, 100)
2.0999999999999996
>>> round_edge_screen(0.5, 100, 45)
1.05
```

`fluids.filters.round_edge_open_mesh(alpha, subtype='diamond pattern wire', angle=0)`

Returns the loss coefficient for a round edged open net/screen made of one of the following patterns, according
to [1]:

‘round bar screen’:

\[ K = 0.95(1 - \alpha) + 0.2(1 - \alpha)^2 \]

‘diamond pattern wire’:

\[ K = 0.67(1 - \alpha) + 1.3(1 - \alpha)^2 \]

‘knotted net’:

\[ K = 0.70(1 - \alpha) + 4.9(1 - \alpha)^2 \]

‘knotless net’:

\[ K = 0.72(1 - \alpha) + 2.1(1 - \alpha)^2 \]

Parameters

- `alpha` [float] Fraction of net/screen open to flow [-]
- `angle` [float, optional] Angle of inclination, with 0 being straight and 90 being parallel to flow [degrees]

Returns

- `K` [float] Loss coefficient [-]

Notes

`alpha` should be between 0.85 and 1 for these correlations. Flow should be turbulent, with Re > 500.

The velocity the loss coefficient relates to is the approach velocity before the mesh.
Fluids Documentation, Release 0.1

References

[1]

Examples

```python
>>> round_edge_open_mesh(0.96, angle=33.)
0.02031327712601458
```

```
fluids.filters.square_edge_screen(alpha)
```

Returns the loss coefficient for a square wire screen or square bar screen or perforated plate with squared edges, as shown in [1].

Parameters

- **alpha** [float] Fraction of screen open to flow [-]

Returns

- **K** [float] Loss coefficient [-]

Notes

Linear interpolation between a table of values. The velocity the loss coefficient relates to is the approach velocity before the screen.

References

[1]

Examples

```python
>>> square_edge_screen(0.99)
0.008000000000000007
```

```
fluids.filters.square_edge_grill(alpha, l=None, Dh=None, fd=None)
```

Returns the loss coefficient for a square grill or square bar screen or perforated plate with squared edges of thickness l, as shown in [1].

for $Dh < l < 50D$

$$K = \frac{0.5(1 - \alpha) + (1 - \alpha^2)}{\alpha^2}$$

else:

$$K = \frac{0.5(1 - \alpha) + (1 - \alpha^2) + f l / D}{\alpha^2}$$

Parameters

- **alpha** [float] Fraction of grill open to flow [-]
- **l** [float, optional] Thickness of the grill or plate [m]
- **Dh** [float, optional] Hydraulic diameter of gap in grill, [m]
Fluids Documentation, Release 0.1

**fd** [float, optional] Darcy friction factor [-]

**Returns**

**K** [float] Loss coefficient [-]

**Notes**

If l, Dh, or fd is not provided, the first expression is used instead. The alteration of the expression to include friction factor is there if the grill is long enough to have considerable friction along the surface of the grill.

The velocity the loss coefficient relates to is the approach velocity before the grill.

**References**

[1]

**Examples**

```python
grip_area_grill(.45)
grip_area_grill(.45, l=.15, Dh=.002, fd=.0185)
```

**fluids.filters.round_edge_grill** *(alpha, l=None, Dh=None, fd=None)*

Returns the loss coefficient for a rounded square grill or square bar screen or perforated plate with rounded edges of thickness l, as shown in [1].

for Dh < l < 50D

\[
K = lookup(alpha)
\]

else:

\[
K = lookup(alpha) + \frac{f l}{\alpha^2 D}
\]

**Parameters**

alpha [float] Fraction of grill open to flow [-]

l [float, optional] Thickness of the grill or plate [m]

Dh [float, optional] Hydraulic diameter of gap in grill, [m]

fd [float, optional] Darcy friction factor [-]

**Returns**

**K** [float] Loss coefficient [-]

**Notes**

If l, Dh, or fd is not provided, the first expression is used instead. The alteration of the expression to include friction factor is there if the grill is long enough to have considerable friction along the surface of the grill. alpha must be between 0.3 and 0.7.

The velocity the loss coefficient relates to is the approach velocity before the grill.
2.7 Fittings pressure drop (fluids.fittings)

`fluids.fittings.contraction_sharp(Di1, Di2)`

Returns loss coefficient for any sharp edged pipe contraction as shown in [1].

\[
K = 0.0696(1 - \beta^5)\lambda^2 + (\lambda - 1)^2
\]

\[
\lambda = 1 + 0.622(1 - 0.215\beta^2 - 0.785\beta^5)
\]

\[
\beta = \frac{d_2}{d_1}
\]

Parameters

- **Di1** [float] Inside diameter of original pipe, [m]
- **Di2** [float] Inside diameter of following pipe, [m]

Returns

- **K** [float] Loss coefficient in terms of the following pipe [-]

Notes

A value of 0.506 or simply 0.5 is often used.
References

[1]

Examples

```python
>>> contraction_sharp(Di1=1, Di2=0.4)
0.5301269161591805
```

```python
fluids.fittings.contraction_round(Di1, Di2, rc, method='Rennels')
```

Returns loss coefficient for any any round edged pipe contraction. This calculation has three methods available. The ‘Miller’ [2] method is a bivariate spline digitization of a graph; the ‘Idelchik’ [3] method is an interpolation using a formula and a table of values.

The most conservative formulation is that of Rennels; with fairly similar. The ‘Idelchik’ method is more conservative and less complex; it offers a straight-line curve where the others curves are curved.

The Rennels [1] formulas are:

\[
K = 0.0696 \left(1 - 0.569 \frac{r}{d_2}\right) \left(1 - \sqrt[4]{\frac{r}{d_2}} \beta\right) (1 - \beta^5)\lambda^2 + (\lambda - 1)^2 \\
\lambda = 1 + 0.622 \left(1 - 0.30 \sqrt[4]{\frac{r}{d_2}} - 0.70 \frac{r}{d_2}\right)^4 (1 - 0.215\beta^2 - 0.785\beta^5) \\
\beta = \frac{d_2}{d_1}
\]

Parameters

- **Di1** [float] Inside diameter of original pipe, [m]
- **Di2** [float] Inside diameter of following pipe, [m]
- **rc** [float] Radius of curvature of the contraction, [m]
- **method** [str] The calculation method to use; one of ‘Rennels’, ‘Miller’, or ‘Idelchik’, [-]

Returns

- **K** [float] Loss coefficient in terms of the following pipe [-]

Notes

Rounding radius larger than 0.14Di2 prevents flow separation from the wall. Further increase in rounding radius continues to reduce loss coefficient.
Comparison of available methods for rounded pipe contractions

- Rennels, ratio = 0.1
- Miller, ratio = 0.1
- Idelchik, ratio = 0.1
- Rennels, ratio = 0.06
- Miller, ratio = 0.06
- Idelchik, ratio = 0.06
- Rennels, ratio = 0.04
- Miller, ratio = 0.04
- Idelchik, ratio = 0.04
- Rennels, ratio = 0.02
- Miller, ratio = 0.02
- Idelchik, ratio = 0.02
- Rennels, ratio = 0
- Miller, ratio = 0
- Idelchik, ratio = 0
References

[1], [2], [3]

Examples

```python
>>> contraction_round(Di1=1, Di2=0.4, rc=0.04)
0.178332490866574
```

**fluids.fittings.contraction_round_Miller** *(Di1, Di2, rc)*

Returns loss coefficient for any round edged pipe contraction using the method of Miller [1]. This method uses a spline fit to a graph with area ratios 0 to 1, and radius ratios (rc/Di2) from 0.1 to 0.

**Parameters**

- **Di1** [float] Inside diameter of original pipe, [m]
- **Di2** [float] Inside diameter of following pipe, [m]
- **rc** [float] Radius of curvature of the contraction, [m]

**Returns**

- **K** [float] Loss coefficient in terms of the following pipe, [-]

**Notes**

This method normally gives lower losses than the Rennels formulation.

References

[1]

Examples

```python
>>> contraction_round_Miller(Di1=1, Di2=0.4, rc=0.04)
0.08565953051298639
```

**fluids.fittings.contraction_conical** *(Di1, Di2, fd=None, l=None, angle=None, Re=None, roughness=0.0, method='Rennels')*

Returns the loss coefficient for any conical pipe contraction. This calculation has five methods available. The ‘Idelchik’ [2] and ‘Blevins’ [3] methods use interpolation among tables of values; ‘Miller’ uses a 2d spline representation of a graph; and the ‘Rennels’ [1], ‘Crane’ [4], and ‘Swamee’ [5] methods use formulas for their calculations.

The ‘Rennels’ [1] formulas are:

\[
K_2 = K_{fr,2} + K_{conv,2}
\]

\[
K_{fr,2} = \frac{f_d(1 - \beta^4)}{8 \sin(\theta/2)}
\]

\[
K_{conv,2} = 0.0696[1 + C_B(\sin(\alpha/2) - 1)](1 - \beta^5)\lambda^2 + (\lambda - 1)^2
\]

\[
\lambda = 1 + 0.622(\alpha/180)^{0.8}(1 - 0.215\beta^2 - 0.785\beta^5)
\]
The ‘Swamee’ [5] formula is:

\[ K = 0.315 \theta^{1/3} \]

Parameters

- `Di1` [float] Inside pipe diameter of the larger, upstream, pipe, [m]
- `Di2` [float] Inside pipe diameter of the smaller, downstream, pipe, [m]
- `fd` [float, optional] Darcy friction factor; used only in the Rennels method and will be calculated if not given, [-]
- `l` [float, optional] Length of the contraction, optional [m]
- `angle` [float, optional] Angle of contraction (180 = sharp, 0 = infinitely long contraction), optional [degrees]
- `Re` [float, optional] Reynolds number of the pipe (used in Rennels method only if no friction factor given), [m]
- `roughness` [float, optional] Roughness of bend wall (used in Rennel method if no friction factor given), [m]

Returns

- `K` [float] Loss coefficient in terms of the following pipe [-]

Notes

Cheap and has substantial impact on pressure drop.

The ‘Idelchik’ method includes two tabular interpolations; its friction term is limited to angles between 2 and 20 degrees and area ratios 0.05 to 0.6, while its main term is limited to length over diameter ratios 0.025 to 0.6. This seems to give it high results for angles < 25 degrees.

The ‘Blevins’ method is based on Idelchik data; it should not be used, because its data jumps around and its data is limited to area ratios .1 to 0.83, and length over diameter ratios 0 to 0.6. The ‘Miller’ method jumps around as well. Unlike most of Miller’s method, there is no correction for Reynolds number.

There is quite a bit of variance in the predictions of the methods, as demonstrated by the following figure.
Comparison of available methods for conical pipe contractions
Area ratio (x) vs. Loss coefficient (y)

Rennels
Idelchik
Crane
Swamee
Blevins
Miller

2.7. Fittings pressure drop (fluids.fittings)
References

[1], [2], [3], [4], [5], [6]

Examples

```python
>>> contraction_conical(Di1=0.1, Di2=0.04, l=0.04, Re=1E6)
0.15639885880609544
```

```
fluids.fittings.contraction_conical_Crane(Di1, Di2, l=None, angle=None)
Returns loss coefficient for a conical pipe contraction as shown in Crane TP 410M [1] between 0 and 180 degrees.
If $\theta < 45^\circ$:

$$K_2 = 0.8 \sin \frac{\theta}{2} (1 - \beta^2)$$

otherwise:

$$K_2 = 0.5 \sqrt{\sin \frac{\theta}{2} (1 - \beta^2)}$$

$$\beta = \frac{d_2}{d_1}$$

Parameters

- `Di1` [float] Inside pipe diameter of the larger, upstream, pipe, [m]
- `Di2` [float] Inside pipe diameter of the smaller, downstream, pipe, [m]
- `l` [float] Length of the contraction, optional [m]
- `angle` [float] Angle of contraction, optional [degrees]

Returns

- `K` [float] Loss coefficient in terms of the following (smaller) pipe [-]

Notes

Cheap and has substantial impact on pressure drop. Note that the nomenclature in [1] is somewhat different - the smaller pipe is called 1, and the larger pipe is called 2; and so the beta ratio is reversed, and the fourth power of beta used in their equation is not necessary.

References

[1]

Examples

```python
>>> contraction_conical_Crane(Di1=0.0779, Di2=0.0525, l=0)
0.2729017979998056
```
**Fluids Documentation, Release 0.1**

**contraction_beveled** *(Di1, Di2, l=None, angle=None)*

Returns loss coefficient for any sharp beveled pipe contraction as shown in [1].

\[
K = 0.0696 \left[ 1 + C_B (\sin(\alpha/2) - 1) \right] (1 - \beta^5)^2 + (\lambda - 1)^2
\]

\[
\lambda = 1 + 0.622 \left[ 1 + C_B \left( \left( \frac{\alpha}{180} \right)^{0.8} - 1 \right) \right] (1 - 0.215\beta^2 - 0.785\beta^5)
\]

\[
C_B = \frac{l}{d_2} \frac{2\beta \tan(\alpha/2)}{1 - \beta}
\]

\[
\beta = d_2/d_1
\]

**Parameters**

- **Di1** [float] Inside diameter of original pipe, [m]
- **Di2** [float] Inside diameter of following pipe, [m]
- **l** [float] Length of the bevel along the pipe axis, [m]
- **angle** [float] Angle of bevel, [degrees]

**Returns**

- **K** [float] Loss coefficient in terms of the following pipe [-]

**References**

[1]

**Examples**

```python
>>> contraction_beveled(Di1=0.5, Di2=0.1, l=.7*.1, angle=120)
0.40946469413070485
```

**diffuser_sharp** *(Di1, Di2)*

Returns loss coefficient for any sudden pipe diameter expansion as shown in [1] and in other sources.

\[
K_1 = (1 - \beta^2)^2
\]

**Parameters**

- **Di1** [float] Inside diameter of original pipe (smaller), [m]
- **Di2** [float] Inside diameter of following pipe (larger), [m]

**Returns**

- **K** [float] Loss coefficient [-]

2.7. Fittings pressure drop (fluids.fittings)
Notes
Highly accurate.

References
[1]

Examples

```python
>>> diffuser_sharp(Di1=.5, Di2=1)
0.5625
```

`fluids.fittings.diffuser_conical(Di1, Di2, l=None, angle=None, fd=None, Re=None, roughness=0.0, method='Rennels')`

Returns the loss coefficient for any conical pipe diffuser. This calculation has four methods available.

The ‘Rennels’ [1] formulas are as follows (three different formulas are used, depending on the angle and the ratio of diameters):

For 0 to 20 degrees, all aspect ratios:

\[
K_1 = 8.30[\tan(\alpha/2)]^{1.75}(1 - \beta^2)^2 + \frac{f(1 - \beta^4)}{8 \sin(\alpha/2)}
\]

For 20 to 60 degrees, beta < 0.5:

\[
K_1 = \left\{ 1.366 \sin \left[ \frac{2\pi (\alpha - 15^\circ)}{180} \right]^{0.5} - 0.170 - 3.28(0.0625 - \beta^4) \right\} \left( 1 - \beta^2 \right)^2 + \frac{f(1 - \beta^4)}{8 \sin(\alpha/2)}
\]

For 20 to 60 degrees, beta >= 0.5:

\[
K_1 = \left\{ 1.366 \sin \left[ \frac{2\pi (\alpha - 15^\circ)}{180} \right]^{0.5} - 0.170 \right\} \left( 1 - \beta^2 \right)^2 + \frac{f(1 - \beta^4)}{8 \sin(\alpha/2)}
\]

For 60 to 180 degrees, beta < 0.5:

\[
K_1 = \left[ 1.205 - 3.28(0.0625 - \beta^4) - 12.8\beta^6 \right] \left( 1 - \beta^2 \right)^2
\]

For 60 to 180 degrees, beta >= 0.5:

\[
K_1 = \left[ 1.205 - 0.20 \sqrt{\frac{\alpha - 60^\circ}{120^\circ}} \right] \left( 1 - \beta^2 \right)^2
\]

The Swamee [5] formula is:

\[
K = \left\{ \frac{0.25}{\beta^3} \left[ 1 + 0.6 \left( \frac{\pi - \theta}{\theta} \right) \right]^{0.533\theta - 2.6} \right\}^{-0.5}
\]

Parameters

- **Di1** [float] Inside diameter of original pipe (smaller), [m]
**Di2** [float] Inside diameter of following pipe (larger), [m]

**l** [float, optional] Length of the contraction along the pipe axis, optional, [m]

**angle** [float, optional] Angle of contraction, [degrees]

**fd** [float, optional] Darcy friction factor [-]

**Re** [float, optional] Reynolds number of the pipe (used in Rennels method only if no friction factor given), [m]

**roughness** [float, optional] Roughness of bend wall (used in Rennel method if no friction factor given), [m]

**method** [str] The method to use for the calculation; one of ‘Rennels’, ‘Crane’, ‘Miller’, ‘Swamee’, or ‘Idelchik’ [-]

**Returns**

**K** [float] Loss coefficient with respect to smaller, upstream diameter [-]

**Notes**

The Miller method changes around quite a bit.

There is quite a bit of variance in the predictions of the methods, as demonstrated by the following figure.

**References**

[1], [2], [3], [4], [5]

**Examples**

```python
>>> diffuser_conical(Di1=1/3., Di2=1.0, angle=50.0, Re=1E6)
0.8027721093415322
```

`fluids.fittings.diffuser_conical_staged(Di1, Di2, DEs, ls, fd=None, method='Rennels')`

Returns loss coefficient for any series of staged conical pipe expansions as shown in [1]. Five different formulas are used, depending on the angle and the ratio of diameters. This function calls `diffuser_conical`.

**Parameters**

- **Di1** [float] Inside diameter of original pipe (smaller), [m]
- **Di2** [float] Inside diameter of following pipe (larger), [m]
- **DEs** [array] Diameters of intermediate sections, [m]
- **ls** [array] Lengths of the various sections, [m]
Comparison of available methods for conical pipe diffusers
Area ratio (x) vs. Loss coefficient (y)
fd [float] Darcy friction factor [-]

method [str] The method to use for the calculation; one of ‘Rennels’, ‘Crane’, ‘Miller’, ‘Swamee’, or ‘Idelchik’ [-]

Returns

K [float] Loss coefficient [-]

Notes

Only lengths of sections currently allowed. This could be changed to understand angles also.

Formula doesn’t make much sense, as observed by the example comparing a series of conical sections. Use only for small numbers of segments of highly differing angles.

References

[1]

Examples

```python
diffuser_conical(Di1=1., Di2=10., l=9, fd=0.01)
```

```
0.973137914861591
```

fluids.fittings.diffuser_conical(Di1, Di2, l)

Returns loss coefficient for any curved wall pipe expansion as shown in [1].

\[
K_1 = \phi (1.43 - 1.3\beta^2)(1 - \beta^2)^2
\]

\[
\phi = 1.01 - 0.624 \frac{l}{d_1} + 0.30 \left( \frac{l}{d_1} \right)^2 - 0.074 \left( \frac{l}{d_1} \right)^3 + 0.0070 \left( \frac{l}{d_1} \right)^4
\]

Parameters

Di1 [float] Inside diameter of original pipe (smaller), [m]

Di2 [float] Inside diameter of following pipe (larger), [m]

l [float] Length of the curve along the pipe axis, [m]

Returns

K [float] Loss coefficient [-]
Notes

Beta^2 should be between 0.1 and 0.9. A small mismatch between tabulated values of this function in table 11.3 is observed with the equation presented.

References

[1]

Examples

```python
>>> diffuser_curved(Di1=.25**0.5, Di2=1., l=2.)
0.2299781250000002
```

```python
fluids.fittings.diffuser_pipe_reducer(Di1, Di2, l, fd1, fd2=None)
```

Returns loss coefficient for any pipe reducer pipe expansion as shown in [1]. This is an approximate formula.

\[
K_f = f_1 \frac{0.20l}{d_1} + \frac{f_1(1 - \beta)}{8 \sin(\alpha/2)} + f_2 \frac{0.20l}{d_2} \beta^4
\]

\[
\alpha = 2 \tan^{-1} \left( \frac{d_1 - d_2}{1.20l} \right)
\]

Parameters

- **Di1** [float] Inside diameter of original pipe (smaller), [m]
- **Di2** [float] Inside diameter of following pipe (larger), [m]
- **l** [float] Length of the pipe reducer along the pipe axis, [m]
- **fd1** [float] Darcy friction factor at inlet diameter [-]
- **fd2** [float] Darcy friction factor at outlet diameter, optional [-]

Returns

- **K** [float] Loss coefficient [-]

Notes

Industry lack of standardization prevents better formulas from being developed. Add 15% if the reducer is eccentric. Friction factor at outlet will be assumed the same as at inlet if not specified.

Doubt about the validity of this equation is raised.

References

[1]
Examples

```python
>>> diffuser_pipe_reducer(Di1=.5, Di2=.75, l=1.5, fd1=0.07)
0.06873244301714816
```

`fluids.fittings.entrance_sharp(method=’Rennels’)`

Returns loss coefficient for a sharp entrance to a pipe. Six sources are available; four of them recommending $K = 0.5$, the most recent ‘Rennels’, method recommending $K = 0.57$, and the ‘Miller’ method recommending $\sim 0.51$ as read from a graph.

![Diagram of entrance_sharp function](image)

**Parameters**


**Returns**

- **$K$** [float] Loss coefficient [-]

**Notes**

0.5 is the result for ‘Swamee’, ‘Blevins’, ‘Idelchik’, and ‘Crane’; ‘Miller’ returns 0.5093, and ‘Rennels’ returns 0.57.

**References**

[1], [2], [3], [4], [5], [6]

**Examples**

```python
>>> entrance_sharp()
0.57
```

`fluids.fittings.entrance_distance(Di, t=None, l=None, method=’Rennels’)`

Returns the loss coefficient for a sharp entrance to a pipe at a distance from the wall of a reservoir. This calculation has five methods available; all but ‘Idelchik’ require the pipe to be at least $Di/2$ into the reservoir.

The most conservative formulation is that of Rennels; with Miller being almost identical until $t/Di$ reaches 0.05, when it continues settling to $K = 0.53$ compared to $K = 0.57$ for ‘Rennels’. ‘Idelchik’ is offset lower by about 0.03 and settles to 0.50. The ‘Harris’ method is a straight interpolation from experimental results with smoothing, and it is the lowest at all points. The ‘Crane’ [6] method returns 0.78 for all cases.

The Rennels [1] formula is:

$$K = 1.12 - 22.5 \frac{t}{d} + 216 \left( \frac{t}{d} \right)^2 + 80 \left( \frac{t}{d} \right)^3$$
Fluids Documentation, Release 0.1

Parameters

Di [float] Inside diameter of pipe, [m]

\( t \) [float, optional] Thickness of pipe wall, used in all but ‘Crane’ method, [m]

\( l \) [float, optional] The distance the pipe extends into the reservoir; used only in the ‘Idelchik’ method, defaults to \( Di \), [m]


Returns

\( K \) [float] Loss coefficient [-]

Notes

This type of inlet is also known as a Borda’s mouthpiece. It is not of practical interest according to [1].

The ‘Idelchik’ [3] data is recommended in [5]; it also provides rounded values for the ‘Harris’ method.

References

[1], [2], [3], [4], [5], [6]

Examples

```python
>>> entrance_distance(Di=0.1, t=0.0005)
1.0154100000000001
>>> entrance_distance(Di=0.1, t=0.0005, method='Idelchik')
0.9249999999999999
>>> entrance_distance(Di=0.1, t=0.0005, l=.02, method='Idelchik')
0.8474999999999999
```

`fluids.fittings.entrance_angled(angle, method='Idelchik')`

Returns loss coefficient for a sharp, angled entrance to a pipe flush with the wall of a reservoir. First published in [2], it has been recommended in [3] as well as in [1].

\[
K = 0.57 + 0.30 \cos(\theta) + 0.20 \cos(\theta)^2
\]

Parameters

\( angle \) [float] Angle of inclination (90° = straight, 0° = parallel to pipe wall), [degrees]
Comparison of available methods for re-entrant entrances

- Rennels
- Miller
- Idelchik
- Harris
- Crane

2.7. Fittings pressure drop (fluids.fittings)
method [str, optional] The method to use; only ‘Idelchik’ is supported

Returns

K [float] Loss coefficient [-]

Notes

Not reliable for angles under 20 degrees. Loss coefficient is the same for an upward or downward angled inlet.

References

[1], [2], [3]

Examples

```python
>>> entrance_angled(30)
0.9798076211353315
```

`fluids.fittings.entrance_rounded(Di, rc, method='Rennels')`

Returns loss coefficient for a rounded entrance to a pipe flush with the wall of a reservoir. This calculation has six methods available.

The most conservative formulation is that of Rennels; with the Swimmee correlation being 0.02-0.07 lower. They were published in 2012 and 2008 respectively, and for this reason could be regarded as more reliable.

The Idel’chik correlation appears based on the Hamilton data; and the Miller correlation as well, except a little more conservative. The Crane model trends similarly but only has a few points. The Harris data set is the lowest.

The Rennels [1] formulas are:

\[
K = 0.0696 \left(1 - 0.569 \frac{r}{d}\right) \lambda^2 + (\lambda - 1)^2
\]

\[
\lambda = 1 + 0.622 \left(1 - 0.30 \sqrt{\frac{r}{d}} - 0.70 \frac{r}{d}\right)^4
\]

The Swamee [5] formula is:

\[
K = 0.5 \left[1 + 36 \left(\frac{r}{D}\right)^{1.2}\right]^{-1}
\]

Parameters

Di [float] Inside diameter of pipe, [m]
rc [float] Radius of curvature of the entrance, [m]

**Returns**

K [float] Loss coefficient [-]

**Notes**

For generously rounded entrance (rc/Di >= 1), the loss coefficient converges to 0.03 in the Rennels method.

The Rennels formulation was derived primarily from data and theoretical analysis from different flow scenarios than a rounded pipe entrance; the only available data in [2] is quite old and [1] casts doubt on it.

The Hamilton data set is available in [1] and [6].

**Comparison of available methods for rounded flush entrances to pipes**

![Comparison of available methods for rounded flush entrances to pipes](image)

**References**

[1], [2], [3], [4], [5], [6], [7], [8]

**Examples**

Point from Diagram 9.2 in [1], which was used to confirm the Rennels model implementation:
fluids.fittings.entrance_beveled(Di, l, angle, method='Rennels')

Returns loss coefficient for a beveled or chamfered entrance to a pipe flush with the wall of a reservoir. This calculation has two methods available.

The ‘Rennels’ and ‘Idelchik’ methods have similar trends, but the ‘Rennels’ formulation is centered around a straight loss coefficient of 0.57, so it is normally at least 0.07 higher.

The Rennels [1] formulas are:

\[
K = 0.0696 \left( 1 - \frac{1}{4} \right)^{\frac{2}{3}} + \frac{\lambda - 1}{\lambda^2}
\]

\[
\lambda = 1 + 0.622 \left[ 1 - 1.5 C_b \left( \frac{l}{d} \right)^{1/4} \right]
\]

\[
C_b = \left( 1 - \frac{\theta}{90} \right) \left( \frac{\theta}{90} \right)^{1/4}
\]

Parameters

- **Di** [float] Inside diameter of pipe, [m]
- **l** [float] Length of bevel measured parallel to the pipe length, [m]
- **angle** [float] Angle of bevel with respect to the pipe length, [degrees]
- **method** [str, optional] One of ‘Rennels’, or ‘Idelchik’, [-]

Returns

- **K** [float] Loss coefficient [-]

Notes

A cheap way of getting a lower pressure drop. Little credible data is available.

The table of data in [2] uses the angle for both bevels, so it runs from 0 to 180 degrees; this function follows the convention in [1] which uses only one angle, with the angle varying from 0 to 90 degrees.

References

[1], [2]
Comparison of available methods for beveled entrances

2.7. Fittings pressure drop (fluids.fittings)
Fluids Documentation, Release 0.1

Examples

```python
>>> entrance_beveled(Di=0.1, l=0.003, angle=45)
0.45086864221916984
>>> entrance_beveled(Di=0.1, l=0.003, angle=45, method='Idelchik')
0.3995000000000001
```

`fluids.fittings.entrance_beveled_orifice(Di, do, l, angle)`

Returns loss coefficient for a beveled or chamfered orifice entrance to a pipe flush with the wall of a reservoir, as shown in [1].

\[
K = 0.0696 \left( 1 - C_b \frac{l}{d_o} \right) \lambda^2 + \left( \lambda - \left( \frac{d_o}{D_i} \right)^2 \right)^2
\]

\[
\lambda = 1 + 0.622 \left[ 1 - C_b \left( \frac{l}{d_o} \right)^{\frac{1-(l/d_o)^{0.25}}{2}} \right]
\]

\[
C_b = \left( 1 - \frac{\Psi}{90} \right) \left( \frac{\Psi}{90} \right)^{\frac{1}{1 + l/d_o}}
\]

Parameters

- **Di** [float] Inside diameter of pipe, [m]
- **do** [float] Inside diameter of orifice, [m]
- **l** [float] Length of bevel measured parallel to the pipe length, [m]
- **angle** [float] Angle of bevel with respect to the pipe length, [degrees]

Returns

- **K** [float] Loss coefficient [-]

References

[1]

Examples

```python
>>> entrance_beveled_orifice(Di=0.1, do=.07, l=0.003, angle=45)
1.2987552913818574
```
**entrance_distance_45_Miller** 

`entrance_distance_45_Miller(Di, Di0)`

Returns loss coefficient for a sharp entrance to a pipe at a distance from the wall of a reservoir with an initial 45 degree slope conical section of diameter `Di0` added to reduce the overall loss coefficient.

This method is as shown in Miller’s Internal Flow Systems [1]. This method is a curve fit to a graph in [1] which was digitized.

**Parameters**

- `Di` [float] Inside diameter of pipe, [m]
- `Di0` [float] Initial inner diameter of the welded conical section of the entrance of the distant (re-entrant) pipe, [m]

**Returns**

- `K` [float] Loss coefficient with respect to the main pipe diameter `Di`, [-]

**Notes**

The graph predicts an almost constant loss coefficient once the thickness of pipe wall to pipe diameter ratio becomes ~0.02.

**References**

[1]

**Examples**

```python
>>> entrance_distance_45_Miller(Di=0.1, Di0=0.14)
0.24407641818143339
```

**exit_normal**

Returns loss coefficient for any exit to a pipe as shown in [1] and in other sources.

\[ K = 1 \]

**Returns**

- `K` [float] Loss coefficient [-]
Notes

It has been found on occasion that \( K = 2.0 \) for laminar flow, and ranges from about 1.04 to 1.10 for turbulent flow.

References

[1]

Examples

```python
>>> exit_normal()
1.0
```

Fluids Documentation, Release 0.1

 fluids.fittings.bend_rounded(Di, angle, fd=None, rc=None, bend_diameters=5.0, Re=None, roughness=0.0, L_unimpeded=None, method='Rennels')

Returns loss coefficient for rounded bend in a pipe of diameter \( Di \), \( angle \), with a specified either radius of curvature \( rc \) or curvature defined by \( bend_diameters \), Reynolds number \( Re \) and optionally pipe roughness, unimpeded length downstream, and with the specified method. This calculation has five methods available.

It is hard to describe one method as more conservative than another as depending on the conditions, the relative results change significantly.

The ‘Miller’ method is the most complicated and slowest method; the ‘Ito’ method comprehensive as well and a source of original data, and the primary basis for the ‘Rennels’ method. The ‘Swamee’ method is very simple and generally does not match the other methods. The ‘Crane’ method may match or not match other methods depending on the inputs.

The Rennels [1] formula is:

\[
K = f \frac{r}{d} + (0.10 + 2.4f) \sin(\alpha/2) + \frac{6.6f(\sqrt{\sin(\alpha/2)} + \sin(\alpha/2))}{(r/d)^{4/7}}
\]

The Swamee [5] formula is:

\[
K = \left[ 0.0733 + 0.923 \left( \frac{d}{rc} \right)^{3.5} \right] \theta^{0.5}
\]

Parameters

\( Di \) [float] Inside diameter of pipe, [m]
\( angle \) [float] Angle of bend, [degrees]
fd [float, optional] Darcy friction factor; used only in Rennels method; calculated if not provided from Reynolds number, diameter, and roughness [-]

rc [float, optional] Radius of curvature of the entrance, optional [m]

bend_diameters [float, optional (used if rc not provided)] Number of diameters of pipe making up the bend radius [-]

Re [float, optional] Reynolds number of the pipe (used in Miller, Ito methods primarily, and Rennels method if no friction factor given), [m]

roughness [float, optional] Roughness of bend wall (used in Miller, Ito methods primarily, and Rennels method if no friction factor given), [m]

L_unimpeded [float, optional] The length of unimpeded pipe without any fittings, instrumentation, or flow disturbances downstream (assumed 20 diameters if not specified); used only in Miller method, [m]


Returns

K [float] Loss coefficient [-]

Notes

When inputting bend diameters, note that manufacturers often specify this as a multiplier of nominal diameter, which is different than actual diameter. Those require that rc be specified.

In the ‘Rennels’ method, rc is limited to 0.5 or above; which represents a sharp, square, inner edge - and an outer bend radius of 1.0. Losses are at a minimum when this value is large. Its first term represents surface friction loss; the second, secondary flows; and the third, flow separation. It encompasses the entire range of elbow and pipe bend configurations. It was developed for bend angles between 0 and 180 degrees; and r/D ratios above 0.5. Only smooth pipe data was used in its development. Note the loss coefficient includes the surface friction of the pipe as if it was straight.

References

[1], [2], [3], [4], [5], [6]

Examples

```python
>>> bend_rounded(Di=4.020, rc=4.0*5, angle=30, Re=1E5)
0.11519070808085191
```

_fluids.fittings_.**bend_rounded_Miller** *(Di, angle, Re, rc=None, bend_diameters=None, roughness=0.0, L_unimpeded=None)*

Calculates the loss coefficient for a rounded pipe bend according to Miller [1]. This is a sophisticated model which uses corrections for pipe roughness, the length of the pipe downstream before another interruption, and a correction for Reynolds number. It interpolates several times using several corrections graphs in [1].

Parameters

Di [float] Inside diameter of pipe, [m]

angle [float] Angle of bend, [degrees]

2.7. Fittings pressure drop (**fluids.fittings**)
Re [float] Reynolds number of the pipe (no specification if inlet or outlet properties should be used), [m]

rc [float, optional] Radius of curvature of the entrance, [m]

bend_diameters [float, optional] Number of diameters of pipe making up the bend radius (used if rc not provided; defaults to 5), [-]

roughness [float, optional] Roughness of bend wall, [m]

L_unimpeded [float, optional] The length of unimpeded pipe without any fittings, instrumentation, or flow disturbances downstream (assumed 20 diameters if not specified), [m]

Returns

K [float] Loss coefficient [-]

Notes

When inputting bend diameters, note that manufacturers often specify this as a multiplier of nominal diameter, which is different than actual diameter. Those require that rc be specified.

rc is limited to 0.5 or above; which represents a sharp, square, inner edge - and an outer bend radius of 1.0. Losses are at a minimum when this value is large.

This was developed for bend angles between 10 and 180 degrees; and r/D ratios between 0.5 and 10. Both smooth and rough data was used in its development from several sources.

Note the loss coefficient includes the surface friction of the pipe as if it was straight.

References

[1]

Examples

```python
>>> bend_rounded_Miller(Di=.6, bend_diameters=2, angle=90, Re=2e6,
... roughness=2E-5, L_unimpeded=30*.6)
0.152618207051459
```

`fluids.fittings.bend_rounded_Crane(Di, angle, rc=None, bend_diameters=None)`

Calculates the loss coefficient for any rounded bend in a pipe according to the Crane TP 410M [1] method. This method effectively uses an interpolation from tabulated values in [1] for friction factor multipliers vs. curvature radius.

Parameters
Di [float] Inside diameter of pipe, [m]

angle [float] Angle of bend, [degrees]

rc [float, optional] Radius of curvature of the entrance, optional [m]

bend_diameters [float, optional (used if rc not provided)] Number of diameters of pipe making up the bend radius [-]

Returns

K [float] Loss coefficient [-]

Notes

The Crane method does match the trend of increased pressure drop as roughness increases.

The points in [1] are extrapolated to other angles via a well-fitting Chebyshev approximation, whose accuracy can be seen in the below plot.

References

[1]
Examples

```python
>>> bend_rounded_Crane(Di=.4020, rc=.4*5, angle=30)
0.09321910015613409
```

`fluids.fittings.bend_miter(angle, Di=None, Re=None, roughness=0.0, L_unimpeded=None, method='Rennels')`

Returns loss coefficient for any single-joint miter bend in a pipe of angle `angle`, diameter `Di`, Reynolds number `Re`, roughness `roughness` unimpeded downstream length `L_unimpeded`, and using the specified method. This calculation has four methods available. The ‘Rennels’ method is based on a formula and extends to angles up to 150 degrees. The ‘Crane’ method extends only to 90 degrees; the ‘Miller’ and ‘Blevins’ methods extend to 120 degrees.

The Rennels [1] formula is:

\[ K = 0.42 \sin(\alpha/2) + 2.56 \sin^3(\alpha/2) \]

The ‘Crane’, ‘Miller’, and ‘Blevins’ methods are all in part graph or tabular based and do not have straightforward formulas.

Parameters

- `angle` [float] Angle of bend, [degrees]
- `Di` [float, optional] Inside diameter of pipe, [m]
- `Re` [float, optional] Reynolds number of the pipe (no specification if inlet or outlet properties should be used), [m]
- `roughness` [float, optional] Roughness of bend wall, [m]
- `L_unimpeded` [float, optional] The length of unimpeded pipe without any fittings, instrumentation, or flow disturbances downstream (assumed 20 diameters if not specified), [m]

Returns

- `K` [float] Loss coefficient with respect to either upstream or downstream diameter, [-]

Notes

This method is designed only for single-jointed miter bends. It is common for miter bends to have two or three sections, to further reduce the loss coefficient. Some methods exist in [2] for taking this into account. Because the additional configurations reduce the pressure loss, it is “common practice” to simply ignore their effect and accept the slight overdesign.

The following figure illustrates the different methods.
Comparison of available methods for mitre bend losses
Angle (x) vs. Loss coefficient (y)
References

[1], [2], [3], [4]

Examples

```python
>>> bend_miter(150)
2.7128147734758103
>>> bend_miter(Di=.6, angle=45, Re=1e6, roughness=1e-5, L_unimpeded=20,... method='Miller')
0.2944060416245167
```

`fluids.fittings.bend_miter_Miller(Di, angle, Re, roughness=0.0, L_unimpeded=None)`

Calculates the loss coefficient for a single miter bend according to Miller [1]. This is a sophisticated model which uses corrections for pipe roughness, the length of the pipe downstream before another interruption, and a correction for Reynolds number. It interpolates several times using several corrections graphs in [1].

**Parameters**

Di [float] Inside diameter of pipe, [m]

angle [float] Angle of miter bend, [degrees]

Re [float] Reynolds number of the pipe (no specification if inlet or outlet properties should be used), [m]

roughness [float, optional] Roughness of bend wall, [m]

L_unimpeded [float, optional] The length of unimpeded pipe without any fittings, instrumentation, or flow disturbances downstream (assumed 20 diameters if not specified), [m]

**Returns**

K [float] Loss coefficient [-]

**Notes**

Note the loss coefficient includes the surface friction of the pipe as if it was straight.

References

[1]

**Examples**

```python
>>> bend_miter_Miller(Di=.6, angle=90, Re=2e6, roughness=2e-5,... L_unimpeded=30*.6)
1.1921574594947668
```

`fluids.fittings.helix(Di, rs, pitch, N, fd)`

Returns loss coefficient for any size constant-pitch helix as shown in [1]. Has applications in immersed coils in tanks.

\[
K = N \left[ f \frac{\sqrt{(2\pi r)^2 + p^2}}{d} + 0.20 + 4.8f \right]
\]
Parameters

**Di** [float] Inside diameter of pipe, [m]

**rs** [float] Radius of spiral, [m]

**pitch** [float] Distance between two subsequent coil centers, [m]

**N** [float] Number of coils in the helix [-]

**fd** [float] Darcy friction factor [-]

Returns

**K** [float] Loss coefficient [-]

Notes

Formulation based on peak secondary flow as in two 180 degree bends per coil. Flow separation ignored. No f, Re, geometry limitations. Source not compared against others.

References

[1]

Examples

```python
>>> helix(Di=0.01, rs=0.1, pitch=.03, N=10, fd=.0185)
14.525134924495514
```

**fluids.fittings.spiral** *(Di, rmax, rmin, pitch, fd)*

Returns loss coefficient for any size constant-pitch spiral as shown in [1]. Has applications in immersed coils in tanks.

\[
K = \frac{r_{\text{max}} - r_{\text{min}}}{p} \left[ f \pi \left( \frac{r_{\text{max}} + r_{\text{min}}}{d} \right) + 0.20 + 4.8f \right] + \frac{13.2f}{(r_{\text{min}}/d)^2}
\]

Parameters

**Di** [float] Inside diameter of pipe, [m]

**rmax** [float] Radius of spiral at extremity, [m]

**rmin** [float] Radius of spiral at end near center, [m]

**pitch** [float] Distance between two subsequent coil centers, [m]

**fd** [float] Darcy friction factor [-]

Returns

**K** [float] Loss coefficient [-]

Notes

Source not compared against others.
References

[1]

Examples

```python
>>> spiral(Di=0.01, rmax=.1, rmin=.02, pitch=.01, fd=0.0185)
7.950918552775473
```

```python
def Darby3K(NPS=None, Re=None, name=None, K1=None, Ki=None, Kd=None)
    Returns loss coefficient for any various fittings, depending on the name input. Alternatively, the Darby constants K1, Ki and Kd may be provided and used instead. Source of data is [1]. Reviews of this model are favorable.

    \[ K_f = K_1 \frac{Re}{K_i} + K_i \left(1 + \frac{K_d}{D_{NPS}^{0.3}}\right) \]

    Note this model uses nominal pipe diameter in inches.

Parameters

- **NPS** [float] Nominal diameter of the pipe, [in]
- **Re** [float] Reynolds number, [-]
- **name** [str] String from Darby dict representing a fitting
- **K1** [float] K1 parameter of Darby model, optional [-]
- **Ki** [float] Ki parameter of Darby model, optional [-]
- **Kd** [float] Kd parameter of Darby model, optional [in]

Returns

- **K** [float] Loss coefficient [-]

Notes

Also described in Albright’s Handbook and Ludwig's Applied Process Design. Relatively uncommon to see it used.

The possibility of combining these methods with those above are attractive.

References

[1], [2]

Examples

```python
>>> Darby3K(NPS=2., Re=10000., name='Valve, Angle valve, 45°, full line size, β → 1')
1.1572523963562353
```

```python
>>> Darby3K(NPS=12., Re=10000., K1=950, Ki=0.25, Kd=4)
0.819510280626355
```
fluids.fittings.Hooper2K(Di, Re, name=None, K1=None, Kinfty=None)

Returns loss coefficient for any various fittings, depending on the name input. Alternatively, the Hooper constants K1, Kinfty may be provided and used instead. Source of data is [1]. Reviews of this model are favorable less favorable than the Darby method but superior to the constant-K method.

\[ K = \frac{K_1}{Re} + K_\infty \left( 1 + \frac{1 \text{ inch}}{D_{in}} \right) \]

Note this model uses actual inside pipe diameter in inches.

**Parameters**

- **Di** [float] Actual inside diameter of the pipe, [in]
- **Re** [float] Reynolds number, [-]
- **name** [str, optional] String from Hooper dict representing a fitting
- **K1** [float, optional] K1 parameter of Hooper model, optional [-]
- **Kinfty** [float, optional] Kinfty parameter of Hooper model, optional [-]

**Returns**

- **K** [float] Loss coefficient [-]

**Notes**

Also described in Ludwig’s Applied Process Design. Relatively uncommon to see it used. No actual example found.

**References**

[1], [2], [3]

**Examples**

```python
>>> Hooper2K(Di=2., Re=10000., name='Valve, Globe, Standard')
6.15
>>> Hooper2K(Di=2., Re=10000., K1=900, Kinfty=4)
6.09
```

fluids.fittings.Kv_to_Cv(Kv)

Convert valve flow coefficient from imperial to common metric units.

\[ C_v = 1.156 K_v \]

**Parameters**

- **Kv** [float] Metric Kv valve flow coefficient (flow rate of water at a pressure drop of 1 bar) [m³/hr]

**Returns**

- **Cv** [float] Imperial Cv valve flow coefficient (flow rate of water at a pressure drop of 1 psi) [gallons/minute]
Notes

Kv = 0.865 Cv is in the IEC standard 60534-2-1. It has also been said that Cv = 1.17Kv; this is wrong by current standards.

The conversion factor does not depend on the density of the fluid or the diameter of the valve. It is calculated with the definition of a US gallon as 231 cubic inches, and a psi as a pound-force per square inch.

The exact conversion coefficient between Kv to Cv is 1.1560992283536566; it is rounded in the formula above.

References

[1]

Examples

```python
>>> Kv_to_Cv(2)
2.3121984567073133
```

```
fluids.fittings.CV_TO_KV(Cv)

Convert valve flow coefficient from imperial to common metric units.

\[ K_v = C_v / 1.156 \]

Parameters

Cv [float] Imperial Cv valve flow coefficient (flow rate of water at a pressure drop of 1 psi) [gallons/minute]

Returns

Kv [float] Metric Kv valve flow coefficient (flow rate of water at a pressure drop of 1 bar) [m^3/hr]

Notes

Kv = 0.865 Cv is in the IEC standard 60534-2-1. It has also been said that Cv = 1.17Kv; this is wrong by current standards.

The conversion factor does not depend on the density of the fluid or the diameter of the valve. It is calculated with the definition of a US gallon as 231 cubic inches, and a psi as a pound-force per square inch.

The exact conversion coefficient between Kv to Cv is 1.1560992283536566; it is rounded in the formula above.

References

[1]

Examples

```python
>>> Cv_to_Kv(2.312)
1.9998283393826013
```
```python
fluids.fittings.Kv_to_K(Kv, D)

Convert valve flow coefficient from common metric units to regular loss coefficients.

\[ K = 1.6 \times 10^9 \frac{D^4}{K_v^2} \]

Parameters

- **Kv** [float] Metric Kv valve flow coefficient (flow rate of water at a pressure drop of 1 bar) [m³/hr]
- **D** [float] Inside diameter of the valve [m]

Returns

- **K** [float] Loss coefficient, [-]

Notes

Crane TP 410 M (2009) gives the coefficient of 0.04 (with diameter in mm).

It also suggests the density of water should be found between 5-40°C. Older versions specify the density should be found at 60 °F, which is used here, and the pressure for the appropriate density is back calculated.

\[ \Delta P = 1 \text{ bar} = \frac{1}{2} \rho V^2 \cdot K \]
\[ V = \frac{K_{\text{hr}}}{3600 \text{ sec}} \]
\[ \rho = 999.29744568 \text{ kg/m}^3 \text{ at } T = 60 \text{ °F}, P = 703572 \text{ Pa} \]

The value of density is calculated with IAPWS-95; it is chosen as it makes the coefficient a very convenient round number. Others constants that have been used are 1.604E9, and 1.60045E9.

References

[1]

Examples

```python
>>> Kv_to_K(2.312, .015)
15.153374600399898
```

```python
fluids.fittings.K_to_Kv(K, D)

Convert regular loss coefficient to valve flow coefficient.

\[ K_v = 4 \times 10^4 \sqrt{\frac{D^4}{K}} \]

Parameters

- **K** [float] Loss coefficient, [-]
- **D** [float] Inside diameter of the valve [m]

Returns

- **Kv** [float] Metric Kv valve flow coefficient (flow rate of water at a pressure drop of 1 bar) [m³/hr]

2.7. Fittings pressure drop (fluids.fittings)
```
Notes

Crane TP 410 M (2009) gives the coefficient of 0.04 (with diameter in mm).

It also suggests the density of water should be found between 5-40°C. Older versions specify the density should be found at 60 °F, which is used here, and the pessure for the appropriate density is back calculated.

\[
\Delta P = 1 \text{ bar} = \frac{1}{2} \rho V^2 \cdot K
\]

\[
V = \frac{Kv \cdot \text{hour}}{\frac{\pi}{4} D^2}
\]

\[
\rho = 999.29744568 \text{ kg/m}^3 \text{ at } T = 60 \text{ F}, P = 703572 Pa
\]

The value of density is calculated with IAPWS-95; it is chosen as it makes the coefficient a very convenient round number. Others constants that have been used are 1.604E9, and 1.60045E9.

References

[1]

Examples

```python
>>> K_to_Kv(15.15337460039990, .015)
2.312
```

fluida.fittings.Cv_to_K(Cv, D)

Convert imperial valve flow coefficient from imperial units to regular loss coefficients.

\[
K = 1.6 \times 10^9 \frac{D^4}{(Cv)^2}
\]

Parameters

- **Cv** [float] Imperial Cv valve flow coefficient (flow rate of water at a pressure drop of 1 psi) [gallons/minute]
- **D** [float] Inside diameter of the valve [m]

Returns

- **K** [float] Loss coefficient, [-]

Notes

The exact conversion coefficient between Kv to Cv is 1.1560992283536566; it is rounded in the formula above.

References

[1]

Examples
```python
>>> Cv_to_K(2.712, .015)
14.719595348352552
```

`fluids.fittings.K_to_Cv(K, D)`

Convert regular loss coefficient to imperial valve flow coefficient.

\[
K_v = 1.156 \cdot 4 \times 10^4 \sqrt[4]{\frac{D^4}{K}}
\]

**Parameters**

- **K** [float] Loss coefficient, [-]
- **D** [float] Inside diameter of the valve [m]

**Returns**

- **Cv** [float] Imperial Cv valve flow coefficient (flow rate of water at a pressure drop of 1 psi) [gallons/minute]

**Notes**

The conversion factor does not depend on the density of the fluid or the diameter of the valve. It is calculated with the definition of a US gallon as 231 cubic inches, and a psi as a pound-force per square inch.

The exact conversion coefficient between Kv to Cv is 1.1560992283536566; it is rounded in the formula above.

**References**

[1]

**Examples**

```python
>>> K_to_Cv(16, .015)
2.601223263795727
```

`fluids.fittings.change_K_basis(K1, D1, D2)`

Converts a loss coefficient \(K1\) from the basis of one diameter \(D1\) to another diameter, \(D2\). This is necessary when dealing with pipelines of changing diameter.

\[
K_2 = K_1 \frac{D_2^4}{D_1^4} = K_1 \frac{A_2^2}{A_1^2}
\]

**Parameters**

- **K1** [float] Loss coefficient with respect to diameter \(D\), [-]
- **D1** [float] Diameter of pipe for which \(K1\) has been calculated, [m]
- **D2** [float] Diameter of pipe for which \(K2\) will be calculated, [m]

**Returns**

- **K2** [float] Loss coefficient with respect to the second diameter, [-]
Notes

This expression is shown in [1] and can easily be derived:

\[ \frac{\rho V_1^2}{2} \cdot K_1 = \frac{\rho V_2^2}{2} \cdot K_2 \]

Substitute velocities for flow rate divided by area:

\[ \frac{8K_1Q^2\rho}{\pi^2D_1^4} = \frac{8K_2Q^2\rho}{\pi^2D_2^4} \]

From here, simplification and rearrangement is all that is required.

References

[1]

Examples

```python
>>> change_K_basis(K1=32.68875692997804, D1=.01, D2=.02)
523.0201108796487
```

`fluids.fittings.K_gate_valve_Crane(D1, D2, angle, fd=None)`

Returns loss coefficient for a gate valve of types wedge disc, double disc, or plug type, as shown in [1].

If \( \beta = 1 \) and \( \theta = 0 \):

\[ K = K_1 = K_2 = 8f_d \]

If \( \beta < 1 \) and \( \theta \leq 45^\circ \):

\[ K_2 = \frac{K + \sin \frac{\theta}{2} \left[ 0.8(1 - \beta^2) + 2.6(1 - \beta^2)^2 \right]}{\beta^4} \]

If \( \beta < 1 \) and \( \theta > 45^\circ \):

\[ K_2 = \frac{K + 0.5 \sqrt{\sin \frac{\theta}{2}(1 - \beta^2) + (1 - \beta^2)^2}}{\beta^4} \]

Parameters

- **D1** [float] Diameter of the valve seat bore (must be smaller or equal to D2), [m]
- **D2** [float] Diameter of the pipe attached to the valve, [m]
- **angle** [float] Angle formed by the reducer in the valve, [degrees]
- **fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor! [-]

Returns

- **K** [float] Loss coefficient with respect to the pipe inside diameter [-]
Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions [2].

References

[1], [2]

Examples

Example 7-4 in [1]; a 150 by 100 mm class 600 steel gate valve, conically tapered ports, length 550 mm, back of sear ring ~150 mm. The valve is connected to 146 mm schedule 80 pipe. The angle can be calculated to be 13 degrees. The valve is specified to be operating in turbulent conditions.

```python
>>> K_gate_valve_Crane(D1=.1, D2=.146, angle=13.115)
1.1466029421844073
```

The calculated result is lower than their value of 1.22; the difference is due to Crane’s generous intermediate rounding. A later, Imperial edition of Crane rounds differently - and comes up with K=1.06.

`fluids.fittings.K_angle_valve_Crane(D1, D2, fd=None, style=0)`

Returns the loss coefficient for all types of angle valve, (reduced seat or throttled) as shown in [1].

If $\beta = 1$:

$$K = K_1 = K_2 = N \cdot f_d$$

Otherwise:

$$K_2 = \frac{K + \left[0.5(1 - \beta^2) + (1 - \beta^2)^2\right]}{\beta^4}$$

For style 0 and 2, N = 55; for style 1, N=150.

Parameters

- **D1** [float] Diameter of the valve seat bore (must be smaller or equal to D2), [m]
- **D2** [float] Diameter of the pipe attached to the valve, [m]
- **fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]
- **style** [int, optional] One of 0, 1, or 2; refers to three different types of angle valves as shown in [1] [-]

Returns

- **K** [float] Loss coefficient with respect to the pipe inside diameter [-]

Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.
fluids.fittings.K_globe_valve_Crane(D1, D2, fd=None)
Returns the loss coefficient for all types of globe valve, (reduced seat or throttled) as shown in [1].

If \( \beta = 1 \):

\[
K = K_1 = K_2 = 340f_d
\]

Otherwise:

\[
K_2 = \frac{K + [0.5(1 - \beta^2) + (1 - \beta^2)^2]}{\beta^4}
\]

Parameters

- **D1** [float] Diameter of the valve seat bore (must be smaller or equal to D2), [m]
- **D2** [float] Diameter of the pipe attached to the valve, [m]
- **fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]

Returns

- **K** [float] Loss coefficient with respect to the pipe inside diameter [-]

Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

References

[1]

Examples

```
>>> K_globe_valve_Crane(.01, .02)
26.597361811128465
```

fluids.fittings.K_swing_check_valve_Crane(D=None, fd=None, angled=True)
Returns the loss coefficient for a swing check valve as shown in [1].

\[
K_2 = N \cdot f_d
\]

For angled swing check valves \( N = 100 \); for straight valves, \( N = 50 \).
Parameters

- **D** [float, optional] Diameter of the pipe attached to the valve, [m]
- **fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]
- **angled** [bool, optional] If True, returns a value 2x the unangled value; the style of the valve [-]

Returns

- **K** [float] Loss coefficient with respect to the pipe inside diameter [-]

Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

References

[1]

Examples

```python
>>> K_swing_check_valve_Crane(D=.02)
2.3974274785373257
```

`fluids.fittings.K_lift_check_valve_Crane(D1, D2, fd=None, angled=True)`

Returns the loss coefficient for a lift check valve as shown in [1].

If $\beta = 1$:

$$K = K_1 = K_2 = N \cdot f_d$$

Otherwise:

$$K_2 = \frac{K + [0.5(1 - \beta^2) + (1 - \beta^2)^2]}{\beta^4}$$

For angled lift check valves $N = 55$; for straight valves, $N = 600$.

Parameters

- **D1** [float] Diameter of the valve seat bore (must be smaller or equal to $D2$), [m]
- **D2** [float] Diameter of the pipe attached to the valve, [m]
- **fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]
- **angled** [bool, optional] If True, returns a value 2x the unangled value; the style of the valve [-]

Returns

- **K** [float] Loss coefficient with respect to the pipe inside diameter [-]
Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

References

[1]

Examples

```python
>>> K_lift_check_valve_Crane(.01, .02)
28.597361811128465
```

fluids.fittings.K_tilting_disk_check_valve_Crane(D, angle, fd=None)

Returns the loss coefficient for a tilting disk check valve as shown in [1]. Results are specified in [1] to be for the disk’s resting position to be at 5 or 25 degrees to the flow direction. The model is implemented here so as to switch to the higher loss 15 degree coefficients at 10 degrees, and use the lesser coefficients for any angle under 10 degrees.

\[ K = N \cdot f_d \]

N is obtained from the following table:

<table>
<thead>
<tr>
<th>angle = 5°</th>
<th>angle = 15°</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-8&quot;</td>
<td>40</td>
</tr>
<tr>
<td>10-14&quot;</td>
<td>30</td>
</tr>
<tr>
<td>16-48&quot;</td>
<td>20</td>
</tr>
</tbody>
</table>

The actual change of coefficients happen at <= 9" and <= 15".

Parameters

- D [float] Diameter of the pipe section the valve in mounted in; the same as the line size [m]
- angle [float] Angle of the tilting disk to the flow direction; nominally 5 or 15 degrees [degrees]
- fd [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!

Returns

- K [float] Loss coefficient with respect to the pipe inside diameter [-]

Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

References

[1]
Examples

>>> K_tilting_disk_check_valve_Crane(.01, 5)
1.1626516551826345

fluids.fittings.K_tilting_disk_check_valve_Crane(D1, D2, fd=None, style=0)
Returns the loss coefficient for a globe stop check valve as shown in [1].
If \( \beta = 1 \):

\[
K = K_1 = K_2 = N \cdot f_d
\]

Otherwise:

\[
K_2 = \frac{K + [0.5(1 - \beta^2) + (1 - \beta^2)^2]}{\beta^4}
\]

Style 0 is the standard form; style 1 is angled, with a restriction to force the flow up through the valve; style 2 is also angled but with a smaller restriction forcing the flow up. N is 400, 300, and 55 for those cases respectively.

Parameters

- **D1** [float] Diameter of the valve seat bore (must be smaller or equal to D2), [m]
- **D2** [float] Diameter of the pipe attached to the valve, [m]
- **fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]
- **style** [int, optional] One of 0, 1, or 2; refers to three different types of angle valves as shown in [1] [-]

Returns

- **K** [float] Loss coefficient with respect to the pipe inside diameter [-]

Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

References

[1]

Examples

>>> K_globe_stop_check_valve_Crane(.1, .02, style=1)
4.5235076518969795

fluids.fittings.K_angle_stop_check_valve_Crane(D1, D2, fd=None, style=0)
Returns the loss coefficient for a angle stop check valve as shown in [1].
If \( \beta = 1 \):

\[
K = K_1 = K_2 = N \cdot f_d
\]
Otherwise:
\[
K_2 = K + \frac{0.5(1 - \beta^2) + (1 - \beta^2)^2}{\beta^4}
\]

Style 0 is the standard form; style 1 has a restriction to force the flow up through the valve; style 2 is has the clearest flow area with no guides for the angle valve. N is 200, 350, and 55 for those cases respectively.

Parameters

- **D1** [float] Diameter of the valve seat bore (must be smaller or equal to \(D2\)), [m]
- **D2** [float] Diameter of the pipe attached to the valve, [m]
- **fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]
- **style** [int, optional] One of 0, 1, or 2; refers to three different types of angle valves as shown in [1] [-]

Returns

- **K** [float] Loss coefficient with respect to the pipe inside diameter [-]

Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

References

[1]

Examples

```python
>>> K_angle_stop_check_valve_Crane(.1, .02, style=1)
4.525425593879809
```

`fluids.fittings.K_ball_valve_Crane(D1, D2, angle, fd=None)`

Returns the loss coefficient for a ball valve as shown in [1].

If \(\beta = 1\):

\[K = K_1 = K_2 = 3fd\]

If \(\beta < 1\) and \(\theta \leq 45^\circ\):

\[K_2 = K + \frac{\sin \frac{\theta}{2} [0.8(1 - \beta^2) + 2.6(1 - \beta^2)^2]}{\beta^4}\]

If \(\beta < 1\) and \(\theta > 45^\circ\):

\[K_2 = K + 0.5 \sqrt{\sin \frac{\theta}{2} (1 - \beta^2) + (1 - \beta^2)^2} \frac{1}{\beta^4}\]

Parameters

- **D1** [float] Diameter of the valve seat bore (must be equal to or smaller than \(D2\)), [m]
**D2** [float] Diameter of the pipe attached to the valve, [m]

**angle** [float] Angle formed by the reducer in the valve, [degrees]

**fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]

**Returns**

**K** [float] Loss coefficient with respect to the pipe inside diameter [-]

**Notes**

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

**References**

[1]

**Examples**

```python
>>> K_ball_valve_Crane(.01, .02, 50)
14.051310974926592
```

**fluids.fittings.K_diaphragm_valve_Crane** *(D=None, fd=None, style=0)*

Returns the loss coefficient for a diaphragm valve of either weir (*style* = 0) or straight-through (*style* = 1) as shown in [1].

\[
K = K_1 = K_2 = N \cdot f_d
\]

For style 0 (weir), \(N = 149\); for style 1 (straight through), \(N = 39\).

**Parameters**

**D** [float, optional] Diameter of the pipe section the valve in mounted in; the same as the line size [m]

**fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]

**style** [int, optional] Either 0 (weir type valve) or 1 (straight through weir valve) [-]

**Returns**

**K** [float] Loss coefficient with respect to the pipe inside diameter [-]

**Notes**

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

**References**

[1]
Examples

```python
>>> K_diaphragm_valve_Crane(D=.1, style=0)
2.4269804835982565
```

`fluids.fittings.K_foot_valve_Crane(D=None, fd=None, style=0)`

Returns the loss coefficient for a foot valve of either poppet disc (`style = 0`) or hinged-disk (`style = 1`) as shown in [1]. Both valves are specified include the loss of the attached strainer.

\[ K = K_1 = K_2 = N \cdot f_d \]

For style 0 (poppet disk), \( N = 420 \); for style 1 (hinged disk), \( N = 75 \).

**Parameters**

- **D** [float, optional] Diameter of the pipe section the valve in mounted in; the same as the line size [m]
- **fd** [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]
- **style** [int, optional] Either 0 (poppet disk foot valve) or 1 (hinged disk foot valve) [-]

**Returns**

- **K** [float] Loss coefficient with respect to the pipe inside diameter [-]

**Notes**

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

**References**

[1]

**Examples**

```python
>>> K_foot_valve_Crane(D=0.2, style=0)
5.912221498436275
```

`fluids.fittings.K_butterfly_valve_Crane(D, fd=None, style=0)`

Returns the loss coefficient for a butterfly valve as shown in [1]. Three different types are supported; Centric (`style = 0`), double offset (`style = 1`), and triple offset (`style = 2`).

\[ K = N \cdot f_d \]

\( N \) is obtained from the following table:

<table>
<thead>
<tr>
<th>Size range</th>
<th>Centric</th>
<th>Double offset</th>
<th>Triple offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>2” - 8”</td>
<td>45</td>
<td>74</td>
<td>218</td>
</tr>
<tr>
<td>10” - 14”</td>
<td>35</td>
<td>52</td>
<td>96</td>
</tr>
<tr>
<td>16” - 24”</td>
<td>25</td>
<td>43</td>
<td>55</td>
</tr>
</tbody>
</table>

The actual change of coefficients happen at <= 9” and <= 15”.
Parameters

\( \textbf{D} \) [float] Diameter of the pipe section the valve in mounted in; the same as the line size [m]

\( \textbf{fd} \) [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]

\( \textbf{style} \) [int, optional] Either 0 (centric), 1 (double offset), or 2 (triple offset) [-]

Returns

\( \textbf{K} \) [float] Loss coefficient with respect to the pipe inside diameter [-]

Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

References

[1]

Examples

```python
>>> K_butterfly_valve_Crane(D=.1, style=2)
3.5508841974793284
```

\texttt{fluids.fittings.K\_plug\_valve\_Crane(D1, D2, angle, fd=None, style=0)}

Returns the loss coefficient for a plug valve or cock valve as shown in [1].

If \( \beta = 1 \):

\[
K = K_1 = K_2 = Nf_d
\]

Otherwise:

\[
K_2 = \frac{K + 0.5\sqrt{\sin^2\left(1 - \beta^2\right) + (1 - \beta^2)^2}}{\beta^4}
\]

Three types of plug valves are supported. For straight-through plug valves (\textit{style} = 0), \( N = 18 \). For 3-way, flow straight through (\textit{style} = 1) plug valves, \( N = 30 \). For 3-way, flow 90\(^\circ\) valves (\textit{style} = 2) \( N = 90 \).

Parameters

\( \textbf{D1} \) [float] Diameter of the valve plug bore (must be equal to or smaller than \textit{D2}), [m]

\( \textbf{D2} \) [float] Diameter of the pipe attached to the valve, [m]

\( \textbf{angle} \) [float] Angle formed by the reducer in the valve, [degrees]

\( \textbf{fd} \) [float, optional] Darcy friction factor calculated for the actual pipe flow in clean steel (roughness = 0.0018 inch) in the fully developed turbulent region; do not specify this to use the original Crane friction factor!, [-]

\( \textbf{style} \) [int, optional] Either 0 (straight-through), 1 (3-way, flow straight-through), or 2 (3-way, flow 90\(^\circ\)) [-]

Returns

\( \textbf{K} \) [float] Loss coefficient with respect to the pipe inside diameter [-]
Notes

This method is not valid in the laminar regime and the pressure drop will be underestimated in those conditions.

References

[1]

Examples

```python
>>> K_plug_valve_Crane(D1=.01, D2=.02, angle=50)
19.80513692341617
```

```python
fluids.fittings.K_branch_converging_Crane(D_run, D_branch, Q_run, Q_branch, angle=90)
```

Returns the loss coefficient for the branch of a converging tee or wye according to the Crane method [1].

\[
K_{\text{branch}} = C \left[ 1 + D\left(\frac{Q_{\text{branch}}}{Q_{\text{comb}}} \cdot \beta_{\text{branch}}^2\right)^2 - E \left(1 - \frac{Q_{\text{branch}}}{Q_{\text{comb}}}\right)^2 - \frac{F}{\beta_{\text{branch}}^2} \left(\frac{Q_{\text{branch}}}{Q_{\text{comb}}}\right)^2 \right]
\]

\[
\beta_{\text{branch}} = \frac{D_{\text{branch}}}{D_{\text{comb}}}
\]

In the above equation, D = 1, E = 2. See the notes for definitions of F and C.

Parameters

- **D_run** [float] Diameter of the straight-through inlet portion of the tee or wye [m]
- **D_branch** [float] Diameter of the pipe attached at an angle to the straight-through, [m]
- **Q_run** [float] Volumetric flow rate in the straight-through inlet of the tee or wye, [m³/s]
- **Q_branch** [float] Volumetric flow rate in the pipe attached at an angle to the straight-through, [m³/s]
- **angle** [float, optional] Angle the branch makes with the straight-through (tee=90, wye<90) [degrees]

Returns

- **K** [float] Loss coefficient of branch with respect to the velocity and inside diameter of the combined flow outlet [-]

Notes

F is linearly interpolated from the table of angles below. There is no cutoff to prevent angles from being larger or smaller than 30 or 90 degrees.

<table>
<thead>
<tr>
<th>Angle [°]</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>1.74</td>
</tr>
<tr>
<td>45</td>
<td>1.41</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>90</td>
<td>0</td>
</tr>
</tbody>
</table>

If \(\beta_{\text{branch}}^2 \leq 0.35\), C = 1
If \( \beta_{branch}^2 > 0.35 \) and \( Q_{branch}/Q_{comb} > 0.4 \), \( C = 0.55 \).

If neither of the above conditions are met:

\[
C = 0.9 \left( 1 - \frac{Q_{branch}}{Q_{comb}} \right)
\]

Note that there is an error in the text of [1]; the errata can be obtained here: http://www.flowoffluids.com/publications/tp-410-errata.aspx

References

[1]

Examples

Example 7-35 of [1]. A DN100 schedule 40 tee has 1135 liters/minute of water passing through the straight leg, and 380 liters/minute of water converging with it through a 90° branch. Calculate the loss coefficient in the branch. The calculated value there is -0.04026.

```python
>>> K_branch_converging_Crane(0.1023, 0.1023, 0.018917, 0.00633)
-0.04044108513625682
```

\texttt{fluids.fittings.K\_run\_converging\_Crane}(\texttt{D\_run}, \texttt{D\_branch}, \texttt{Q\_run}, \texttt{Q\_branch}, \texttt{angle}=90)

Returns the loss coefficient for the run of a converging tee or wye according to the Crane method [1].

\[
K_{branch} = C \left[ 1 + D \left( \frac{Q_{branch}}{Q_{comb}} \beta_{branch}^2 \right)^2 - E \left( 1 - \frac{Q_{branch}}{Q_{comb}} \right)^2 - \frac{F}{\beta_{branch}^2} \left( \frac{Q_{branch}}{Q_{comb}} \right)^2 \right]
\]

\[
\beta_{branch} = \frac{D_{branch}}{D_{comb}}
\]

In the above equation, \( C=1, D=0, E=1 \). See the notes for definitions of \( F \) and also the special case of 90°.

Parameters

\begin{itemize}
  \item \texttt{D\_run} [float] Diameter of the straight-through inlet portion of the tee or wye [m]
  \item \texttt{D\_branch} [float] Diameter of the pipe attached at an angle to the straight-through, [m]
  \item \texttt{Q\_run} [float] Volumetric flow rate in the straight-through inlet of the tee or wye, [m³/s]
  \item \texttt{Q\_branch} [float] Volumetric flow rate in the pipe attached at an angle to the straight-through, [m³/s]
  \item \texttt{angle} [float, optional] Angle the branch makes with the straight-through (tee=90, wye<90) [degrees]
\end{itemize}

Returns

\begin{itemize}
  \item \texttt{K} [float] Loss coefficient of run with respect to the velocity and inside diameter of the combined flow outlet [-]
\end{itemize}

Notes

\( F \) is linearly interpolated from the table of angles below. There is no cutoff to prevent angles from being larger or smaller than 30 or 60 degrees. The switch to the special 90° happens at 75°.

2.7. Fittings pressure drop (fluids.fittings)
For the special case of 90°, the formula used is as follows.

\[ K_{\text{run}} = 1.55 \left( \frac{Q_{\text{branch}}}{Q_{\text{comb}}} \right) - \left( \frac{Q_{\text{branch}}}{Q_{\text{comb}}} \right)^2 \]

References

[1]

Examples

Example 7-35 of [1]. A DN100 schedule 40 tee has 1135 liters/minute of water passing through the straight leg, and 380 liters/minute of water converging with it through a 90° branch. Calculate the loss coefficient in the run. The calculated value there is 0.03258.

```python
>>> K_run_converging_Crane(0.1023, 0.1023, 0.018917, 0.00633)
0.32575847854551254
```

`fluids.fittings.K_branch_diverging_Crane(D_run, D_branch, Q_run, Q_branch, angle=90)`

Returns the loss coefficient for the branch of a diverging tee or wye according to the Crane method [1].

\[ K_{\text{branch}} = G \left[ 1 + H \left( \frac{Q_{\text{branch}}}{Q_{\text{comb}} \beta_{\text{branch}}^2} \right)^2 - J \left( \frac{Q_{\text{branch}}}{Q_{\text{comb}} \beta_{\text{branch}}^2} \right) \cos \theta \right] \]

\[ \beta_{\text{branch}} = \frac{D_{\text{branch}}}{D_{\text{comb}}} \]

See the notes for definitions of H, J, and G.

Parameters

- `D_run` [float] Diameter of the straight-through inlet portion of the tee or wye [m]
- `D_branch` [float] Diameter of the pipe attached at an angle to the straight-through, [m]
- `Q_run` [float] Volumetric flow rate in the straight-through outlet of the tee or wye, [m^3/s]
- `Q_branch` [float] Volumetric flow rate in the pipe attached at an angle to the straight-through, [m^3/s]
- `angle` [float, optional] Angle the branch makes with the straight-through (tee=90, wye<90) [degrees]

Returns

- `K` [float] Loss coefficient of branch with respect to the velocity and inside diameter of the combined flow inlet [-]
Notes

If $\beta_{branch} = 1$, $\theta = 90^\circ$, $H = 0.3$ and $J = 0$. Otherwise $H = 1$ and $J = 2$.

$G$ is determined according to the following pseudocode:

```python
if angle < 75:
    if beta2 <= 0.35:
        if Q_ratio <= 0.4:
            G = 1.1 - 0.7*Q_ratio
        else:
            G = 0.85
    else:
        if Q_ratio <= 0.6:
            G = 1.0 - 0.6*Q_ratio
        else:
            G = 0.6
else:
    if beta2 <= 2/3.:
        G = 1
    else:
        G = 1 + 0.3*Q_ratio*Q_ratio
```

Note that there are several errors in the text of [1]; the errata can be obtained here: http://www.flowoffluids.com/publications/tp-410-errata.aspx

References

[1]

Examples

Example 7-36 of [1]. A DN150 schedule 80 wye has 1515 liters/minute of water exiting the straight leg, and 950 liters/minute of water exiting it through a 45° branch. Calculate the loss coefficient in the branch. The calculated value there is 0.4640.

```
>>> K_branch_diverging_Crane(0.146, 0.146, 0.02525, 0.01583, angle=45)
0.4639895627496694
```

`fluids.fittings.K_run_diverging_Crane(D_run, D_branch, Q_run, Q_branch, angle=90)`

Returns the loss coefficient for the run of a converging tee or wye according to the Crane method [1].

$$K_{run} = M \left( \frac{Q_{branch}}{Q_{comb}} \right)^2$$

$$\beta_{branch} = \frac{D_{branch}}{D_{comb}}$$

See the notes for the definition of $M$.

Parameters

- **D_run** [float] Diameter of the straight-through inlet portion of the tee or wye [m]
- **D_branch** [float] Diameter of the pipe attached at an angle to the straight-through, [m]
- **Q_run** [float] Volumetric flow rate in the straight-through outlet of the tee or wye, [m^3/s]
**Q_branch** [float] Volumetric flow rate in the pipe attached at an angle to the straight-through, [m³/s]

**angle** [float, optional] Angle the branch makes with the straight-through (tee=90, wye<90) [degrees]

**Returns**

**K** [float] Loss coefficient of run with respect to the velocity and inside diameter of the combined flow inlet [-]

**Notes**

M is calculated according to the following pseudocode:

```python
if beta*beta <= 0.4:
    M = 0.4
elif Q_branch/Q_comb <= 0.5:
    M = 2*(2*Q_branch/Q_comb - 1)
else:
    M = 0.3*(2*Q_branch/Q_comb - 1)
```

**References**

[1]

**Examples**

Example 7-36 of [1]. A DN150 schedule 80 wye has 1515 liters/minute of water exiting the straight leg, and 950 liters/minute of water exiting it through a 45° branch. Calculate the loss coefficient in the branch. The calculated value there is -0.06809.

```python
>>> K_run_diverging_Crane(0.146, 0.146, 0.02525, 0.01583, angle=45)
-0.06810067607153049
```

**fluids.fittings.v_lift_valve_Crane** *(rho, D1=None, D2=None, style='swing check angled')*

Calculates the approximate minimum velocity required to lift the disk or other controlling element of a check valve to a fully open, stable, position according to the Crane method [1].

\[
v_{\text{min}} = N \cdot \frac{\sqrt{\text{kg/m}^3}}{\rho} \cdot \frac{\text{kg/m}^3}{\rho}
\]

See the notes for the definition of values of N and which check valves use which formulas.

**Parameters**

**rho** [float] Density of the fluid [kg/m³]

**D1** [float, optional] Diameter of the valve bore (must be equal to or smaller than D2), [m]

**D2** [float, optional] Diameter of the pipe attached to the valve, [m]
style [str] The type of valve; one of ['swing check angled', 'swing check straight', 'swing check UL', 'lift check straight', 'lift check angled', 'tilting check 5°', 'tilting check 15°', 'stop check globe 1', 'stop check angle 1', 'stop check globe 2', 'stop check angle 2', 'stop check globe 3', 'stop check angle 3', 'foot valve poppet disc', 'foot valve hinged disc'], [-]

Returns

v_min [float] Approximate minimum velocity required to keep the disc fully lifted, preventing chattering and wear [m/s]

Notes

This equation is not dimensionless.

<table>
<thead>
<tr>
<th>Name/string</th>
<th>N</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>'swing check angled'</td>
<td>45</td>
<td>No</td>
</tr>
<tr>
<td>'swing check straight'</td>
<td>75</td>
<td>No</td>
</tr>
<tr>
<td>'swing check UL'</td>
<td>120</td>
<td>No</td>
</tr>
<tr>
<td>'lift check straight'</td>
<td>50</td>
<td>Yes</td>
</tr>
<tr>
<td>'lift check angled'</td>
<td>170</td>
<td>Yes</td>
</tr>
<tr>
<td>'tilting check 5°'</td>
<td>100</td>
<td>No</td>
</tr>
<tr>
<td>'tilting check 15°'</td>
<td>40</td>
<td>No</td>
</tr>
<tr>
<td>'stop check globe 1'</td>
<td>70</td>
<td>Yes</td>
</tr>
<tr>
<td>'stop check angle 1'</td>
<td>95</td>
<td>Yes</td>
</tr>
<tr>
<td>'stop check globe 2'</td>
<td>75</td>
<td>Yes</td>
</tr>
<tr>
<td>'stop check angle 2'</td>
<td>75</td>
<td>Yes</td>
</tr>
<tr>
<td>'stop check globe 3'</td>
<td>170</td>
<td>Yes</td>
</tr>
<tr>
<td>'stop check angle 3'</td>
<td>170</td>
<td>Yes</td>
</tr>
<tr>
<td>'foot valve poppet disc'</td>
<td>20</td>
<td>No</td>
</tr>
<tr>
<td>'foot valve hinged disc'</td>
<td>45</td>
<td>No</td>
</tr>
</tbody>
</table>

References

[1]

Examples

```python
>>> v_lift_valve_Crane(rho=998.2, D1=0.0627, D2=0.0779, style='lift check straight')
1.0252301935349286
```

2.8 Orifice plates, flow nozzles, Venturi tubes, cone and wedge meters (fluids.flow_meter)

fluids.flow_meter.C_Reader_Harris_Gallagher(D, Do, rho, mu, m, taps='corner')

Calculates the coefficient of discharge of the orifice based on the geometry of the plate, measured pressures of
the orifice, mass flow rate through the orifice, and the density and viscosity of the fluid.

\[
C = 0.5961 + 0.0261\beta^2 - 0.216\beta^8 + 0.000521 \left( \frac{10^6 \beta}{Re_D} \right)^{0.7} + (0.0188 + 0.0063A)\beta^{3.5} \left( \frac{10^6}{Re_D} \right)^{0.3} + (0.043 + 0.080 \exp(-10L_1) - 0.123 \exp(-7L_1))(1 - 0.11A)\frac{\beta^4}{1 - \beta^4} - 0.031(M'_2 - 0.8M^{1.1}_2)\beta^{1.3}
\]

\[
M'_2 = \frac{2L'_2}{1 - \beta}
\]

\[
A = \left( \frac{190000\beta}{Re_D} \right)^{0.8}
\]

\[
Re_D = \frac{\rho v D}{\mu}
\]

If \( D < 71.12 \text{ mm (2.8 in.)} \) (Note this is a continuous addition; there is no discontinuity):

\[
C^+ = 0.11(0.75 - \beta) \left( 2.8 - \frac{D}{0.0254} \right)
\]

If the orifice has corner taps:

\[ L_1 = L'_2 = 0 \]

If the orifice has D and D/2 taps:

\[ L_1 = 1 \]

\[ L'_2 = 0.47 \]

If the orifice has Flange taps:

\[ L_1 = L'_2 = \frac{0.0254}{D} \]

Parameters

- **D** [float] Upstream internal pipe diameter, [m]
- **Do** [float] Diameter of orifice at flow conditions, [m]
- **rho** [float] Density of fluid at \( P_1 \), [kg/m^3]
- **mu** [float] Viscosity of fluid at \( P_1 \), [Pa*s]
- **m** [float] Mass flow rate of fluid through the orifice, [kg/s]
- **taps** [str] The orientation of the taps; one of ‘corner’, ‘flange’, ‘D’, or ‘D/2’, [-]

Returns

- **C** [float] Coefficient of discharge of the orifice, [-]
Notes

The following limits apply to the orifice plate standard [1]:

The measured pressure difference for the orifice plate should be under 250 kPa.

There are roughness limits as well; the roughness should be under 6 micrometers, although there are many more conditions to that given in [1].

For orifice plates with D and D/2 or corner pressure taps:
- Orifice bore diameter must be larger than 12.5 mm (0.5 inches)
- Pipe diameter between 50 mm and 1 m (2 to 40 inches)
- Beta between 0.1 and 0.75 inclusive
- Reynolds number larger than 5000 (for $0.10 \leq \beta \leq 0.56$) or for $\beta \geq 0.56$, $Re_D \geq 16000/\beta^2$

For orifice plates with flange pressure taps:
- Orifice bore diameter must be larger than 12.5 mm (0.5 inches)
- Pipe diameter between 50 mm and 1 m (2 to 40 inches)
- Beta between 0.1 and 0.75 inclusive
- Reynolds number larger than 5000 and also larger than $170000/\beta^2 D$.

This is also presented in Crane’s TP410 (2009) publication, whereas the 1999 and 1982 editions showed only a graph for discharge coefficients.

References

[1], [2], [3], [4]

Examples

```python
>>> C_Reader_Harris_Gallagher(D=0.07391, Do=0.0222, rho=1.165, mu=1.85E-5, ...
... m=0.12, taps='flange')
0.5990326277163659
```

`fluids.flow_meter.differential_pressure_meter_solver(D, rho, mu, k, D2=None, P1=None, P2=None, m=None, meter_type='ISO 5167 orifice', taps=None)`

Calculates either the mass flow rate, the upstream pressure, the second pressure value, or the orifice diameter for a differential pressure flow meter based on the geometry of the meter, measured pressures of the meter, and the density, viscosity, and isentropic exponent of the fluid. This solves an equation iteratively to obtain the correct flow rate.

Parameters

- `D` [float] Upstream internal pipe diameter, [m]
- `rho` [float] Density of fluid at $P1$, [kg/m$^3$]
- `mu` [float] Viscosity of fluid at $P1$, [Pa*s]
- `k` [float] Isentropic exponent of fluid, [-]
**D2** [float, optional] Diameter of orifice, or venturi meter orifice, or flow tube orifice, or cone meter end diameter, or wedge meter fluid flow height, [m]

**P1** [float, optional] Static pressure of fluid upstream of differential pressure meter at the cross-section of the pressure tap, [Pa]

**P2** [float, optional] Static pressure of fluid downstream of differential pressure meter or at the prescribed location (varies by type of meter) [Pa]

**m** [float, optional] Mass flow rate of fluid through the flow meter, [kg/s]

**meter_type** [str, optional] One of ('ISO 5167 orifice', 'long radius nozzle', 'ISA 1932 nozzle', 'venuri nozzle', 'as cast convergent venturi tube', 'machined convergent venturi tube', 'rough welded convergent venturi tube', 'cone meter', 'wedge meter'), [-]

**taps** [str, optional] The orientation of the taps; one of ‘corner’, ‘flange’, ‘D’, or ‘D/2’; applies for orifice meters only, [-]

**Returns**

**ans** [float] One of \( m \), the mass flow rate of the fluid; \( P1 \), the pressure upstream of the flow meter; \( P2 \), the second pressure tap’s value; and \( D2 \), the diameter of the measuring device; units of respectively, kg/s, Pa, Pa, or m

**Notes**

See the appropriate functions for the documentation for the formulas and references used in each method.

The solvers make some assumptions about the range of values answers may be in.

Note that the solver for the upstream pressure uses the provided values of density, viscosity and isentropic exponent; whereas these values all depend on pressure (albeit to a small extent). An outer loop should be added with pressure-dependent values calculated in it for maximum accuracy.

It would be possible to solve for the upstream pipe diameter, but there is no use for that functionality.

**Examples**

```python
differential_pressure_meter_solver(D=0.07366, D2=0.05, P1=200000.0, ...
P2=183000.0, rho=999.1, mu=0.0011, k=1.33, ...
meter_type='ISO 5167 orifice', taps='D')
7.702338035732167
```

```python
differential_pressure_meter_solver(D=0.07366, m=7.702338, P1=200000.0, ...
P2=183000.0, rho=999.1, mu=0.0011, k=1.33, ...
meter_type='ISO 5167 orifice', taps='D')
0.04999999990831885
```

`fluids.flow_meter.differential_pressure_meter_dP(D, D2, P1, P2, C=None, meter_type='ISO 5167 orifice')`

Calculates either the non-recoverable pressure drop of a differential pressure flow meter based on the geometry of the meter, measured pressures of the meter, and for most models the meter discharge coefficient.

**Parameters**

**D** [float] Upstream internal pipe diameter, [m]

**D2** [float] Diameter of orifice, or venturi meter orifice, or flow tube orifice, or cone meter end diameter, or wedge meter fluid flow height, [m]
**P1** [float] Static pressure of fluid upstream of differential pressure meter at the cross-section of the pressure tap, [Pa]

**P2** [float] Static pressure of fluid downstream of differential pressure meter or at the prescribed location (varies by type of meter) [Pa]

**C** [float, optional] Coefficient of discharge (used only in orifice plates, and venturi nozzles), [-]


**Returns**

**dP** [float] Non-recoverable pressure drop of the differential pressure flow meter, [Pa]

**Notes**

See the appropriate functions for the documentation for the formulas and references used in each method. Wedge meters, and venturi nozzles do not have standard formulas available for pressure drop computation.

**Examples**

```python
>>> differential_pressure_meter_dP(D=0.07366, D2=0.05, P1=200000.0, ...
P2=183000.0, meter_type='as cast convergent venturi tube')
1788.5717754177406
```

**fluids.flow_meter.flow_meter_discharge** *(D, Do, P1, P2, rho, C, expansibility=1.0)*

Calculates the flow rate of an orifice plate based on the geometry of the plate, measured pressures of the orifice, and the density of the fluid.

\[
m = \left( \frac{\pi D_o^2}{4} \right) C \sqrt{\frac{2 \Delta P \rho_1}{\sqrt{1 - \beta^4}}} \epsilon
\]

**Parameters**

**D** [float] Upstream internal pipe diameter, [m]

**Do** [float] Diameter of orifice at flow conditions, [m]

**P1** [float] Static pressure of fluid upstream of orifice at the cross-section of the pressure tap, [Pa]

**P2** [float] Static pressure of fluid downstream of orifice at the cross-section of the pressure tap, [Pa]

**rho** [float] Density of fluid at P1, [kg/m^3]

**C** [float] Coefficient of discharge of the orifice, [-]

**expansibility** [float, optional] Expansibility factor (1 for incompressible fluids, less than 1 for real fluids), [-]

**Returns**

**m** [float] Mass flow rate of fluid, [kg/s]
Notes

This is formula 1-12 in [1] and also [2].

References

[1], [2]

Examples

```python
>>> flow_meter_discharge(D=0.0739, Do=0.0222, P1=1E5, P2=9.9E4, rho=1.1646, 
... C=0.5988, expansibility=0.9975)
0.01120390943807026
```

`fluids.flow_meter.orifice_expansibility(D, Do, P1, P2, k)`

Calculates the expansibility factor for orifice plate calculations based on the geometry of the plate, measured pressures of the orifice, and the isentropic exponent of the fluid.

\[
\epsilon = 1 - (0.351 + 0.256\beta^4 + 0.93\beta^8) \left[ 1 - \left( \frac{P_2}{P_1} \right)^{1/k} \right]
\]

Parameters

- **D** [float] Upstream internal pipe diameter, [m]
- **Do** [float] Diameter of orifice at flow conditions, [m]
- **P1** [float] Static pressure of fluid upstream of orifice at the cross-section of the pressure tap, [Pa]
- **P2** [float] Static pressure of fluid downstream of orifice at the cross-section of the pressure tap, [Pa]
- **k** [float] Isentropic exponent of fluid, [-]

Returns

- **expansibility** [float, optional] Expansibility factor (1 for incompressible fluids, less than 1 for real fluids), [-]

Notes

This formula was determined for the range of P2/P1 >= 0.80, and for fluids of air, steam, and natural gas. However, there is no objection to using it for other fluids.

References

[1], [2]

Examples

```python
>>> orifice_expansibility(D=0.0739, Do=0.0222, P1=1E5, P2=9.9E4, k=1.4)
0.9974739057343425
```
**discharge_coefficient_to_K**(*D, Do, C*)

Converts a discharge coefficient to a standard loss coefficient, for use in computation of the actual pressure drop of an orifice or other device.

\[ K = \left( \frac{\sqrt{1 - \beta^4(1 - C^4)}}{C\beta^2} - 1 \right)^2 \]

**Parameters**

- **D** [float] Upstream internal pipe diameter, [m]
- **Do** [float] Diameter of orifice at flow conditions, [m]
- **C** [float] Coefficient of discharge of the orifice, [-]

**Returns**

- **K** [float] Loss coefficient with respect to the velocity and density of the fluid just upstream of the orifice, [-]

**Notes**

If expansibility is used in the orifice calculation, the result will not match with the specified pressure drop formula in [1]; it can almost be matched by dividing the calculated mass flow by the expansibility factor and using that mass flow with the loss coefficient.

**References**

[1], [2]

**Examples**

```python
>>> discharge_coefficient_to_K(D=0.07366, Do=0.05, C=0.61512)
5.2314291729754
```

**K_to_discharge_coefficient**(*D, Do, K*)

Converts a standard loss coefficient to a discharge coefficient.

\[ C = \sqrt{\frac{1}{2\sqrt{K}\beta^4 + K^4} - \frac{\beta^4}{2\sqrt{K}^4 + K^4}} \]

**Parameters**

- **D** [float] Upstream internal pipe diameter, [m]
- **Do** [float] Diameter of orifice at flow conditions, [m]
- **K** [float] Loss coefficient with respect to the velocity and density of the fluid just upstream of the orifice, [-]

**Returns**

- **C** [float] Coefficient of discharge of the orifice, [-]
Notes

If expansibility is used in the orifice calculation, the result will not match with the specified pressure drop formula in [1]; it can almost be matched by dividing the calculated mass flow by the expansibility factor and using that mass flow with the loss coefficient.

This expression was derived with SymPy, and checked numerically. There were three other, incorrect roots.

References

[1], [2]

Examples

```python
>>> K_to_discharge_coefficient(D=0.07366, Do=0.05, K=5.2314291729754)
0.6151200000000001
```

`fluids.flow_meter.dP_orifice(D, Do, P1, P2, C)`

Calculates the non-recoverable pressure drop of an orifice plate based on the pressure drop and the geometry of the plate and the discharge coefficient.

\[
\Delta \bar{\omega} = \frac{\sqrt{1 - \beta^4(1 - C^2)} - C\beta^2}{\sqrt{1 - \beta^4(1 - C^2)} + C\beta^2}(P_1 - P_2)
\]

Parameters

- **D** [float] Upstream internal pipe diameter, [m]
- **Do** [float] Diameter of orifice at flow conditions, [m]
- **P1** [float] Static pressure of fluid upstream of orifice at the cross-section of the pressure tap, [Pa]
- **P2** [float] Static pressure of fluid downstream of orifice at the cross-section of the pressure tap, [Pa]
- **C** [float] Coefficient of discharge of the orifice, [-]

Returns

- **dP** [float] Non-recoverable pressure drop of the orifice plate, [Pa]

Notes

This formula can be well approximated by:

\[
\Delta \bar{\omega} = (1 - \beta^{1.9})(P_1 - P_2)
\]

The recoverable pressure drop should be recovered by 6 pipe diameters downstream of the orifice plate.

References

[1], [2]
Examples

```python
>>> dP_orifice(D=0.07366, Do=0.05, P1=200000.0, P2=183000.0, C=0.61512)
9069.474705745388
```

```text
fluids.flow_meter.velocity_of_approach_factor(D, Do)
Calculates a factor for orifice plate design called the velocity of approach.

\[
\text{Velocity of approach} = \frac{1}{\sqrt{1 - \beta^4}}
\]

Parameters

D  [float] Upstream internal pipe diameter, [m]
Do  [float] Diameter of orifice at flow conditions, [m]

Returns

velocity_of_approach  [float] Coefficient of discharge of the orifice, [-]
```

References

[1]

Examples

```python
>>> velocity_of_approach_factor(D=0.0739, Do=0.0222)
1.0040970074165514
```

```text
fluids.flow_meter.flow_coefficient(D, Do, C)
Calculates a factor for differential pressure flow meter design called the flow coefficient. This should not be confused with the flow coefficient often used when discussing valves.

\[
\text{Flow coefficient} = \frac{C}{\sqrt{1 - \beta^4}}
\]

Parameters

D  [float] Upstream internal pipe diameter, [m]
Do  [float] Diameter of flow meter characteristic dimension at flow conditions, [m]
C  [float] Coefficient of discharge of the flow meter, [-]

Returns

flow_coefficient  [float] Differential pressure flow meter flow coefficient, [-]
```

Notes

This measure is used not just for orifices but for other differential pressure flow meters [2].

It is sometimes given the symbol K. It is also equal to the product of the discharge coefficient and the velocity of approach factor [2].
References

[1], [2]

Examples

```python
>>> flow_coefficient(D=0.0739, Do=0.0222, C=0.6)
0.6024582044499308
```

```
fluctuations.flow_meter.nozzle_expansibility(D, Do, P1, P2, k, beta=None)
Calculates the expansibility factor for a nozzle or venturi nozzle, based on the geometry of the plate, measured pressures of the orifice, and the isentropic exponent of the fluid.

\[
\epsilon = \left\{ \left( \frac{\kappa \tau^2}{\kappa - 1} \right) \left( \frac{1 - \beta^4}{1 - \beta^4 \tau^2/\kappa} \right) \left[ \frac{1 - \tau(\kappa-1)/\kappa}{1 - \tau} \right] \right\}^{0.5}
\]

Parameters

- **D** [float] Upstream internal pipe diameter, [m]
- **Do** [float] Diameter of orifice of the venturi or nozzle, [m]
- **P1** [float] Static pressure of fluid upstream of orifice at the cross-section of the pressure tap, [Pa]
- **P2** [float] Static pressure of fluid downstream of orifice at the cross-section of the pressure tap, [Pa]
- **k** [float] Isentropic exponent of fluid, [-]
- **beta** [float, optional] Optional beta ratio, which is useful to specify for wedge meters or flow meters which have a different beta ratio calculation, [-]

Returns

- **expansibility** [float] Expansibility factor (1 for incompressible fluids, less than 1 for real fluids), [-]

Notes

This formula was determined for the range of P2/P1 \( \geq \) 0.75.

References

[1], [2]

Examples

```python
>>> nozzle_expansibility(D=0.0739, Do=0.0222, P1=1E5, P2=9.9E4, k=1.4)
0.9945702344566746
```

```
fluids.flow_meter.C_long_radius_nozzle(D, Do, rho, mu, m)
Calculates the coefficient of discharge of a long radius nozzle used for measuring flow rate of fluid, based on the geometry of the nozzle, mass flow rate through the nozzle, and the density and viscosity of the fluid.

\[
C = 0.9965 - 0.00653\beta^{0.5} \left( \frac{10^6}{Re_D} \right)^{0.5}
\]

188 Chapter 2. API Reference
Parameters

\( D \) [float] Upstream internal pipe diameter, [m]
\( D_o \) [float] Diameter of long radius nozzle orifice at flow conditions, [m]
\( \rho \) [float] Density of fluid at \( P_1 \), [kg/m\(^3\)]
\( \mu \) [float] Viscosity of fluid at \( P_1 \), [Pa\( \cdot \)s]
\( m \) [float] Mass flow rate of fluid through the nozzle, [kg/s]

Returns

\( C \) [float] Coefficient of discharge of the long radius nozzle orifice, [-]

References

[1], [2]

Examples

```python
>>> C_long_radius_nozzle(D=0.07391, Do=0.0422, rho=1.2, mu=1.8E-5, m=0.1)
0.9805503704679863
```

\texttt{fluids.flow\_meter.C\_ISA\_1932\_nozzle}(D, Do, rho, mu, m)

Calculates the coefficient of discharge of an ISA 1932 style nozzle used for measuring flow rate of fluid, based on the geometry of the nozzle, mass flow rate through the nozzle, and the density and viscosity of the fluid.

\[
C = 0.9900 - 0.2262\beta^{4.1} - (0.00175\beta^2 - 0.0033\beta^{4.15}) \left( \frac{10^6}{Re_D} \right)^{1.15}
\]

Parameters

\( D \) [float] Upstream internal pipe diameter, [m]
\( D_o \) [float] Diameter of nozzle orifice at flow conditions, [m]
\( \rho \) [float] Density of fluid at \( P_1 \), [kg/m\(^3\)]
\( \mu \) [float] Viscosity of fluid at \( P_1 \), [Pa\( \cdot \)s]
\( m \) [float] Mass flow rate of fluid through the nozzle, [kg/s]

Returns

\( C \) [float] Coefficient of discharge of the nozzle orifice, [-]

References

[1], [2]

Examples

```python
>>> C\_ISA\_1932\_nozzle(D=0.07391, Do=0.0422, rho=1.2, mu=1.8E-5, m=0.1)
0.9635849973250495
```
fluids.flow_meter.C_venturi_nozzle(D, Do)
Calculates the coefficient of discharge of an Venturi style nozzle used for measuring flow rate of fluid, based on
the geometry of the nozzle.

\[ C = 0.9858 - 0.196\beta^{4.5} \]

**Parameters**

- D [float] Upstream internal pipe diameter, [m]
- Do [float] Diameter of nozzle orifice at flow conditions, [m]

**Returns**

- C [float] Coefficient of discharge of the nozzle orifice, [-]

**References**

[1], [2]

**Examples**

```python
>>> C_venturi_nozzle(D=0.07391, Do=0.0422)
0.9698996454169576
```

fluids.flow_meter.orifice_expansibility_1989(D, Do, P1, P2, k)
Calculates the expansibility factor for orifice plate calculations based on the geometry of the plate, measured
pressures of the orifice, and the isentropic exponent of the fluid.

\[ \epsilon = 1 - (0.41 + 0.35\beta^{4})\Delta P/kP_1 \]

**Parameters**

- D [float] Upstream internal pipe diameter, [m]
- Do [float] Diameter of orifice at flow conditions, [m]
- P1 [float] Static pressure of fluid upstream of orifice at the cross-section of the pressure tap, [Pa]
- P2 [float] Static pressure of fluid downstream of orifice at the cross-section of the pressure tap, [Pa]
- k [float] Isentropic exponent of fluid, [-]

**Returns**

- expansibility [float] Expansibility factor (1 for incompressible fluids, less than 1 for real fluids), [-]

**Notes**

This formula was determined for the range of P2/P1 >= 0.75, and for fluids of air, steam, and natural gas. However, there is no objection to using it for other fluids.

This is an older formula used to calculate expansibility factors for orifice plates.
In this standard, an expansibility factor formula transformation in terms of the pressure after the orifice is presented as well. This is the more standard formulation in terms of the upstream conditions. The other formula is below for reference only:

\[ \epsilon_2 = \sqrt{1 + \frac{\Delta P}{P_2}} - (0.41 + 0.35\beta^4) \left( \frac{\Delta P}{\kappa P_2 \sqrt{1 + \frac{\Delta P}{P_2}}} \right) \]

[2] recommends this formulation for wedge meters as well.

References

[1], [2]

Examples

```python
>>> orifice_expansibility_1989(D=0.0739, Do=0.0222, P1=1E5, P2=9.9E4, k=1.4)
0.9970510687411718
```

`fluids.flow_meter.dp_venturi_tube(D, Do, P1, P2)`

Calculates the non-recoverable pressure drop of a venturi tube differential pressure meter based on the pressure drop and the geometry of the venturi meter.

\[ \epsilon = \frac{\Delta \bar{w}}{\Delta P} \]

The \( \epsilon \) value is looked up in a table of values as a function of beta ratio and upstream pipe diameter (roughness impact).

Parameters

- **D** [float] Upstream internal pipe diameter, [m]
- **Do** [float] Diameter of venturi tube at flow conditions, [m]
- **P1** [float] Static pressure of fluid upstream of venturi tube at the cross-section of the pressure tap, [Pa]
- **P2** [float] Static pressure of fluid downstream of venturi tube at the cross-section of the pressure tap, [Pa]

Returns

- **dP** [float] Non-recoverable pressure drop of the venturi tube, [Pa]

Notes

The recoverable pressure drop should be recovered by 6 pipe diameters downstream of the venturi tube.

Note there is some information on the effect of Reynolds number as well in [1] and [2], with a curve showing an increased pressure drop from 1E5-6E5 to with a decreasing multiplier from 1.75 to 1; the multiplier is 1 for higher Reynolds numbers. This is not currently included in this implementation.

References

[1], [2]
Examples

```python
>>> dP_venturi_tube(D=0.07366, Do=0.05, P1=200000.0, P2=183000.0)
1788.5717754177406
```

```python
fluids.flow_meter.diameter_ratio_cone_meter(D, Dc)
Calculates the diameter ratio \( \beta \) used to characterize a cone flow meter.

\[
\beta = \sqrt{1 - \frac{d^2}{D^2}}
\]

Parameters

- **D** [float] Upstream internal pipe diameter, [m]
- **Dc** [float] Diameter of the largest end of the cone meter, [m]

Returns

- **beta** [float] Cone meter diameter ratio, [-]

References

[1]

Examples

```python
>>> diameter_ratio_cone_meter(D=0.2575, Dc=0.184)
0.6995709873957624
```

```python
fluids.flow_meter.diameter_ratio_wedge_meter(D, H)
Calculates the diameter ratio \( \beta \) used to characterize a wedge flow meter as given in [1] and [2].

\[
\beta = \left( \frac{1}{\pi} \left\{ \arccos \left( 1 - \frac{2H}{D} \right) - 2 \left[ 1 - \frac{2H}{D} \right] \left( \frac{H}{D} - \left( \frac{H}{D} \right)^2 \right)^{0.5} \right\} \right)^{0.5}
\]

Parameters

- **D** [float] Upstream internal pipe diameter, [m]
- **H** [float] Portion of the diameter of the clear segment of the pipe up to the wedge blocking flow; the height of the pipe up to the wedge, [m]

Returns

- **beta** [float] Wedge meter diameter ratio, [-]

References

[1], [2]

Examples
Calculates the expansibility factor for a cone flow meter, based on the geometry of the cone meter, measured pressures of the orifice, and the isentropic exponent of the fluid. Developed in [1], also shown in [2].

\[ \epsilon = 1 - (0.649 + 0.696\beta^4) \frac{\Delta P}{\kappa P_1} \]

Parameters

- **D** [float] Upstream internal pipe diameter, [m]
- **Dc** [float] Diameter of the largest end of the cone meter, [m]
- **P1** [float] Static pressure of fluid upstream of cone meter at the cross-section of the pressure tap, [Pa]
- **P2** [float] Static pressure of fluid at the end of the center of the cone pressure tap, [Pa]
- **k** [float] Isentropic exponent of fluid, [-]

Returns

- **expansibility** [float] Expansibility factor (1 for incompressible fluids, less than 1 for real fluids), [-]

Notes

This formula was determined for the range of \( P_2/P_1 \geq 0.75 \); the only gas used to determine the formula is air.

References

[1], [2]

Examples

```python
>>> cone_meter_expansibility_Stewart(D=1, Dc=0.9, P1=1E6, P2=8.5E5, k=1.2)
0.9157343
```

Calculates the non-recoverable pressure drop of a cone meter based on the measured pressures before and at the cone end, and the geometry of the cone meter according to [1].

\[ \Delta \omega = (1.09 - 0.813\beta)\Delta P \]

Parameters

- **D** [float] Upstream internal pipe diameter, [m]
- **Dc** [float] Diameter of the largest end of the cone meter, [m]
- **P1** [float] Static pressure of fluid upstream of cone meter at the cross-section of the pressure tap, [Pa]
- **P2** [float] Static pressure of fluid at the end of the center of the cone pressure tap, [Pa]
Returns

dP  [float] Non-recoverable pressure drop of the orifice plate, [Pa]

Notes

The recoverable pressure drop should be recovered by 6 pipe diameters downstream of the cone meter.

References

[1]

Examples

```python
>>> dP_cone_meter(1, .7, 1E6, 9.5E5)
25470.093437973323
```

`fluids.flow_meter.C_wedge_meter_Miller(D, H)`
Calculates the coefficient of discharge of an wedge flow meter used for measuring flow rate of fluid, based on the geometry of the differential pressure flow meter.

For half-inch lines:

\[ C = 0.7883 + 0.107(1 - \beta^2) \]

For 1 to 1.5 inch lines:

\[ C = 0.6143 + 0.718(1 - \beta^2) \]

For 1.5 to 24 inch lines:

\[ C = 0.5433 + 0.2453(1 - \beta^2) \]

Parameters

- D  [float] Upstream internal pipe diameter, [m]
- H  [float] Portion of the diameter of the clear segment of the pipe up to the wedge blocking flow; the height of the pipe up to the wedge, [m]

Returns

- C  [float] Coefficient of discharge of the wedge flow meter, [-]

Notes

There is an ISO standard being developed to cover wedge meters as of 2018.

Wedge meters can have varying angles; 60 and 90 degree wedge meters have been reported. Tap locations 1 or 2 diameters (upstream and downstream), and 2D upstream/1D downstream have been used. Some wedges are sharp; some are smooth. [2] gives some experimental values.

References

[1], [2]
**Examples**

```python
>>> C_wedge_meter_Miller(D=0.1524, H=0.3*0.1524)
0.7267069372687651
```

```python
fluids.flow_meter.C_wedge_meter_ISO_5167_6_2017(D, H)
Calculates the coefficient of discharge of an wedge flow meter used for measuring flow rate of fluid, based on the geometry of the differential pressure flow meter according to the ISO 5167-6 standard (draft 2017).

\[ C = 0.77 - 0.09\beta \]

**Parameters**

- **D** [float] Upstream internal pipe diameter, [m]
- **H** [float] Portion of the diameter of the clear segment of the pipe up to the wedge blocking flow; the height of the pipe up to the wedge, [m]

**Returns**

- **C** [float] Coefficient of discharge of the wedge flow meter, [-]

**Notes**

This standard applies for wedge meters in line sizes between 50 and 600 mm; and height ratios between 0.2 and 0.6. The range of allowable Reynolds numbers is large; between 1E4 and 9E6. The uncertainty of the flow coefficient is approximately 4%. Usually a 10:1 span of flow can be measured accurately. The discharge and entry length of the meters must be at least half a pipe diameter. The wedge angle must be 90 degrees, plus or minus two degrees.

The orientation of the wedge meter does not change the accuracy of this model.

There should be a straight run of 10 pipe diameters before the wedge meter inlet, and two of the same pipe diameters after it.

**References**

[1]

**Examples**

```python
>>> C_wedge_meter_ISO_5167_6_2017(D=0.1524, H=0.3*0.1524)
0.724792059539853
```

```python
dP_wedge_meter(D, H, P1, P2)
Calculates the non-recoverable pressure drop of a wedge meter based on the measured pressures before and at the wedge meter, and the geometry of the wedge meter according to [1].

\[ \Delta\bar{\omega} = (1.09 - 0.79\beta)\Delta P \]

**Parameters**

- **D** [float] Upstream internal pipe diameter, [m]
- **H** [float] Portion of the diameter of the clear segment of the pipe up to the wedge blocking flow; the height of the pipe up to the wedge, [m]

2.8. Orifice plates, flow nozzles, Venturi tubes, cone and wedge meters (fluids.flow_meter)
Fluids Documentation, Release 0.1

**P1** [float] Static pressure of fluid upstream of wedge meter at the cross-section of the pressure tap, [Pa]

**P2** [float] Static pressure of fluid at the end of the wedge meter pressure tap, [Pa]

**Returns**

**dP** [float] Non-recoverable pressure drop of the wedge meter, [Pa]

**Notes**

The recoverable pressure drop should be recovered by 5 pipe diameters downstream of the wedge meter.

**References**

[1]

**Examples**

```python
>>> dP_wedge_meter(1, .7, 1E6, 9.5E5)
20344.849697483587
```

**fluids.flow_meter.C_Reader_Harris_Gallagher_wet_venturi_tube**(mg, ml, rhog, rhol, D, Do, H=1)

Calculates the coefficient of discharge of the wet gas venturi tube based on the geometry of the tube, mass flow rates of liquid and vapor through the tube, the density of the liquid and gas phases, and an adjustable coefficient \( H \).

\[
C = 1 - 0.0463 \exp(-0.05Fr_{gas,th}) \cdot \min \left( 1, \sqrt{\frac{X}{0.016}} \right)
\]

\[
Fr_{gas,th} = \frac{Fr_{gas, densiometric}}{\beta^{2.5}}
\]

\[
\phi = \sqrt{1 + C_{Ch}X + X^2}
\]

\[
C_{Ch} = \left( \frac{\rho_l}{\rho_{l,g}} \right)^n + \left( \frac{\rho_{l,g}}{\rho_l} \right)^n
\]

\[
n = \max \left[ 0.583 - 0.18\beta^2 - 0.578 \exp \left( -0.8Fr_{gas, densiometric} \right), 0.392 - 0.18\beta^2 \right]
\]

\[
X = \left( \frac{m_l}{m_g} \right) \sqrt{\frac{\rho_{l,g}}{\rho_l}}
\]

\[
Fr_{gas, densiometric} = \frac{v_{gas}}{\sqrt{gD}} \sqrt{\frac{\rho_{l,g}}{\rho_l - \rho_{l,g}}} = \frac{4n_g}{\rho_{l,g} \pi D^2 \sqrt{\rho_l - \rho_{l,g}}}
\]

**Parameters**

- **mg** [float] Mass flow rate of gas through the venturi tube, [kg/s]
- **ml** [float] Mass flow rate of liquid through the venturi tube, [kg/s]
- **rhog** [float] Density of gas at \( P1 \), [kg/m\(^3\)]
- **rhol** [float] Density of liquid at \( P1 \), [kg/m\(^3\)]
- **D** [float] Upstream internal pipe diameter, [m]
Do  [float] Diameter of venturi tube at flow conditions, [m]

H  [float, optional] A surface-tension effect coefficient used to adjust for different fluids, (1 for a hydrocarbon liquid, 1.35 for water, 0.79 for water in steam) [-]

Returns

C  [float] Coefficient of discharge of the wet gas venturi tube flow meter (includes flow rate of gas ONLY), [-]

Notes

This model has more error than single phase differential pressure meters. The model was first published in [1], and became ISO 11583 later.

The limits of this correlation according to [2] are as follows:

\[ 0.4 \leq \beta \leq 0.75 \]
\[ 0 < X \leq 0.3 \]
\[ Fr_{gas, th} > 3 \]
\[ \frac{\rho_g}{\rho_l} > 0.02 \]
\[ D \geq 50 \text{ mm} \]

References

[1], [2]

Examples

```python
>>> C_Reade_Harris_Gallagher_wet_venturi_tube(mg=5.31926, ml=5.31926/2, ...
... rhog=50.0, rhol=800., D=.1, Do=.06, H=1)
0.9754210845876333
```

**fluids.flow_meter.dP_Reader_Harris_Gallagher_wet_venturi_tube** *(D, Do, P1, P2, ml, mg, rhol, rhog, H=1)*

Calculates the non-recoverable pressure drop of a wet gas venturi nozzle based on the pressure drop and the geometry of the venturi nozzle, the mass flow rates of liquid and gas through it, the densities of the vapor and liquid phase, and an adjustable coefficient *H*.

\[
Y = \frac{\Delta \omega}{\Delta P} - 0.0896 - 0.48\beta^9
\]
\[
Y_{max} = 0.61 \exp \left[ -11 \frac{\rho_{1.g}}{\rho_l} - 0.045 \frac{Fr_{gas}}{H} \right]
\]
\[
\frac{Y}{Y_{max}} = 1 - \exp \left[ -35X^{0.75} \exp \left( -0.28 \frac{Fr_{gas}}{H} \right) \right]
\]
\[
X = \left( \frac{m_l}{m_g} \right) \sqrt{\frac{\rho_{1.g}}{\rho_l}}
\]
\[
Fr_{gas, densiometric} = \frac{v_{gas}}{\sqrt{gD}} \sqrt{\frac{\rho_{1.g}}{\rho_l - \rho_{1.g}}} = \frac{4m_g}{\rho_{1.g}\pi D^2 \sqrt{gD}} \sqrt{\frac{\rho_{1.g}}{\rho_l - \rho_{1.g}}}
\]
Parameters

D  [float] Upstream internal pipe diameter, [m]
Do [float] Diameter of venturi tube at flow conditions, [m]
P1 [float] Static pressure of fluid upstream of venturi tube at the cross-section of the pressure tap, [Pa]
P2 [float] Static pressure of fluid downstream of venturi tube at the cross-section of the pressure tap, [Pa]
ml [float] Mass flow rate of liquid through the venturi tube, [kg/s]
mg [float] Mass flow rate of gas through the venturi tube, [kg/s]
rhol [float] Density of liquid at P1, [kg/m^3]
rhog [float] Density of gas at P1, [kg/m^3]
H  [float, optional] A surface-tension effect coefficient used to adjust for different fluids, (1 for a hydrocarbon liquid, 1.35 for water, 0.79 for water in steam) [-]

Returns

C  [float] Coefficient of discharge of the wet gas venturi tube flow meter (includes flow rate of gas ONLY), [-]

Notes

The model was first published in [1], and became ISO 11583 later.

References

[1],[2]

Examples

```python
>>> dP_Reader_Harris_Gallagher_wet_venturi_tube(D=.1, Do=.06, H=1,
...   P1=6E6, P2=6E6-5E4, ml=5.31926/2, mg=5.31926, rhog=50.0, rhol=800.,)
16957.43843129572
```

fluids.flow_meter.differential_pressure_meter_C_epsilon(D, D2, m, P1, P2, rho, mu,
                                    k, meter_type, taps=None)

Calculates the discharge coefficient and expansibility of a flow meter given the mass flow rate, the upstream pressure, the second pressure value, and the orifice diameter for a differential pressure flow meter based on the geometry of the meter, measured pressures of the meter, and the density, viscosity, and isentropic exponent of the fluid.

Parameters

D  [float] Upstream internal pipe diameter, [m]
D2 [float] Diameter of orifice, or venturi meter orifice, or flow tube orifice, or cone meter end diameter, or wedge meter fluid flow height, [m]
m  [float] Mass flow rate of fluid through the flow meter, [kg/s]
P1 [float] Static pressure of fluid upstream of differential pressure meter at the cross-section of the pressure tap, [Pa]
P2  [float] Static pressure of fluid downstream of differential pressure meter or at the prescribed location (varies by type of meter) [Pa]

rho  [float] Density of fluid at P1, [kg/m³]

mu  [float] Viscosity of fluid at P1, [Pa*s]

k  [float] Isentropic exponent of fluid, [-]

meter_type  [str] One of ('ISO 5167 orifice', 'long radius nozzle', 'ISA 1932 nozzle', 'venuri nozzle', 'as cast convergent venturi tube', 'machined convergent venturi tube', 'rough welded convergent venturi tube', 'cone meter', 'wedge meter'), [-]

taps  [str, optional] The orientation of the taps; one of ‘corner’, ‘flange’, ‘D’, or ‘D/2’; applies for orifice meters only, [-]

Returns

C  [float] Coefficient of discharge of the specified flow meter type at the specified conditions, [-]

expansibility  [float] Expansibility factor (1 for incompressible fluids, less than 1 for real fluids), [-]

Notes

This function should be called by an outer loop when solving for a variable.

Examples

```python
>>> differential_pressure_meter_C_epsilon(D=0.07366, D2=0.05, P1=200000.0, P2=183000.0, rho=999.1, mu=0.0011, k=1.33, m=7.702338035732168, meter_type='ISO 5167 orifice', taps='D')
(0.6151252900244296, 0.9711026966676307)
```

2.8. Orifice plates, flow nozzles, Venturi tubes, cone and wedge meters (fluids.flow_meter) 199
2.9 Friction factor and pipe roughness (fluids.friction)

`fluids.friction.friction_factor(Re, eD=0, Method='Clamond', Darcy=True, AvailableMethods=False)`

Calculates friction factor. Uses a specified method, or automatically picks one from the dictionary of available methods. 29 approximations are available as well as the direct solution, described in the table below. The default is to use the exact solution. Can also be accessed under the name `fd`.

For Re < 2040, [1] the laminar solution is always returned, regardless of selected method.

Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float, optional] Relative roughness of the wall, [-]

Returns

- **f** [float] Friction factor, [-]
- **methods** [list, only returned if AvailableMethods == True] List of methods which claim to be valid for the range of Re and eD given

Other Parameters

- **Method** [string, optional] A string of the function name to use
- **Darcy** [bool, optional] If False, will return fanning friction factor, 1/4 of the Darcy value
- **AvailableMethods** [bool, optional] If True, function will consider which methods claim to be valid for the range of Re and eD given

See also:

- *Colebrook*
- *Clamond*
## Notes

<table>
<thead>
<tr>
<th>Name</th>
<th>Re min</th>
<th>Re max</th>
<th>Re Default</th>
<th>( \epsilon/D ) Min</th>
<th>( \epsilon/D ) Max</th>
<th>( \epsilon/D ) Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clamond</td>
<td>0</td>
<td>None</td>
<td>None</td>
<td>0</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Rao Kumar 2007</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Eck 1973</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Jain 1976</td>
<td>5000</td>
<td>1.0E+7</td>
<td>None</td>
<td>4.0E-5</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Avci Karagoz 2009</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Swamee Jain 1976</td>
<td>5000</td>
<td>1.0E+8</td>
<td>None</td>
<td>1.0E-6</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Churchill 1977</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Brkic 2011</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Chen 1979</td>
<td>4000</td>
<td>4.0E+8</td>
<td>None</td>
<td>1.0E-7</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Round 1980</td>
<td>4000</td>
<td>4.0E+8</td>
<td>None</td>
<td>0</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Papaevangelo 2010</td>
<td>10000</td>
<td>1.0E+7</td>
<td>None</td>
<td>1.0E-5</td>
<td>0.001</td>
<td>None</td>
</tr>
<tr>
<td>Fang 2011</td>
<td>3000</td>
<td>1.0E+8</td>
<td>None</td>
<td>0</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Shacham 1980</td>
<td>4000</td>
<td>4.0E+8</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Barr 1981</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Churchill 1973</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Moody</td>
<td>4000</td>
<td>1.0E+8</td>
<td>None</td>
<td>0</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>Zigrang Sylvester1</td>
<td>4000</td>
<td>1.0E+8</td>
<td>None</td>
<td>4.0E-5</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Zigrang Sylvester2</td>
<td>4000</td>
<td>1.0E+8</td>
<td>None</td>
<td>4.0E-5</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Buzzelli 2008</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Haaland</td>
<td>4000</td>
<td>1.0E+8</td>
<td>None</td>
<td>1.0E-6</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Serghides 1</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Serghides 2</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Tsal 1989</td>
<td>4000</td>
<td>1.0E+8</td>
<td>None</td>
<td>0</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Alishul 1952</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Wood 1966</td>
<td>4000</td>
<td>5.0E+7</td>
<td>None</td>
<td>1.0E-5</td>
<td>0.04</td>
<td>None</td>
</tr>
<tr>
<td>Manadilli 1997</td>
<td>5245</td>
<td>1.0E+8</td>
<td>None</td>
<td>0</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Brkic 2011</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Romeo 2002</td>
<td>3000</td>
<td>1.5E+8</td>
<td>None</td>
<td>0</td>
<td>0.05</td>
<td>None</td>
</tr>
<tr>
<td>Sonnad Goudar 2006</td>
<td>4000</td>
<td>1.0E+8</td>
<td>None</td>
<td>1.0E-6</td>
<td>0.05</td>
<td>None</td>
</tr>
</tbody>
</table>

## References

[1]

## Examples

```python
>>> friction_factor(Re=1E5, eD=1E-4)
0.01851386607747165
```

```python
fluids.friction.friction_factor_curved(Re, Di, Dc, roughness=0.0, Method=None,
Rec_method='Schmidt', laminar_method='Schmidt laminar', turbulent_method='Schmidt turbulent',
Darcy=True, AvailableMethods=False)
```

Calculates friction factor fluid flowing in a curved pipe or helical coil, supporting both laminar and turbulent regimes. Selects the appropriate regime by default, and has default correlation choices. Optionally, a specific correlation can be specified with the `Method` keyword.

The default correlations are those recommended in [1], and are believed to be the best publicly available.
Parameters

- **Re** [float] Reynolds number with \( D=Di \), [-]
- **Di** [float] Inner diameter of the tube making up the coil, [m]
- **Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]
- **roughness** [float, optional] Roughness of pipe wall [m]

Returns

- **f** [float] Friction factor, [-]
- **methods** [list, only returned if AvailableMethods == True] List of methods in the regime the specified \( Re \) is in at the given \( Di \) and \( Dc \).

Other Parameters

- **Method** [string, optional] A string of the function name to use, overriding the default turbulent/laminar selection.
- **laminar_method** [str, optional] Friction factor correlation for the laminar regime; one of \[‘White’, ‘Mori Nakayama laminar’, ‘Schmidt laminar’\]; the default is ‘Schmidt laminar’.
- **Darcy** [bool, optional] If False, will return fanning friction factor, \( 1/4 \) of the Darcy value
- **AvailableMethods** [bool, optional] If True, function will consider which methods claim to be valid for the range of \( Re \) and \( eD \) given

See also:

- `fluids.geometry.HelicalCoil`
- `helical_turbulent_fd_Schmidt`
- `helical_turbulent_fd_Srinivasan`
- `helical_turbulent_fd_Mandal_Nigam`
- `helical_turbulent_fd_Ju`
- `helical_turbulent_fd_Guo`
- `helical_turbulent_fd_Czop`
- `helical_turbulent_fd_Prasad`
- `helical_turbulent_fd_Mori_Nakayama`
- `helical_laminar_fd_Schmidt`
- `helical_laminar_fd_Mori_Nakayama`
- `helical_laminar_fd_White`
- `helical_transition_Re_Schmidt`
- `helical_transition_Re_Srinivasan`
helical_transition_Re_Kutateladze_Borishanskii
helical_transition_Re_Kubair_Kuloor
helical_transition_Re_Ito
helical_transition_Re_Seth_Stahel

Notes
The range of accuracy of these correlations is much than that in a straight pipe.

References
[1]

Examples

```python
>>> friction_factor_curved(Re=1E5, Di=0.02, Dc=0.5)
0.022961996738387523
```

**fluids.friction.Colebrook** *(Re, eD, tol=None)*
Calculates Darcy friction factor using the Colebrook equation originally published in [1]. Normally, this function uses an exact solution to the Colebrook equation, derived with a CAS. A numerical can also be used.

\[
\frac{1}{\sqrt{f}} = -2 \log_{10} \left( \frac{e/D}{3.7} + 2.51 \cdot \frac{3.7}{Re \sqrt{f}} \right)
\]

Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float] Relative roughness, [-]
- **tol** [float, optional] None for analytical solution (default); user specified value to use the numerical solution; 0 to use *mpmath* and provide a bit-correct exact solution to the maximum fidelity of the system’s *float*; -1 to apply the Clamond solution where appropriate for greater speed (Re > 10), [-]

Returns

- **fd** [float] Darcy friction factor [-]

Notes
The solution is as follows:

\[
f_d = \frac{\ln(10)^2 \cdot 3.7^2 \cdot 2.51^2}{\left( \log(10)e/D \cdot Re - 2 \cdot 2.51 \cdot 3.7 \cdot \text{lambertW} \left[ \log(\sqrt{10}) \sqrt{10^{\left( \frac{e}{D} \cdot Re \right)}/2.51} \right] \right)}
\]

Some effort to optimize this function has been made. The *lambertw* function from scipy is used, and is defined to solve the specific function:

\[
y = x \exp(x)
\]
\[
\text{lambertW}(y) = x
\]
This is relatively slow despite its explicit form as it uses the mathematical function `lambertw` which is expensive to compute.

For high relative roughness and Reynolds numbers, an OverflowError can be encountered in the solution of this equation. The numerical solution is then used.

The numerical solution provides values which are generally within an rtol of 1E-12 to the analytical solution; however, due to the different rounding order, it is possible for them to be as different as rtol 1E-5 or higher. The 1E-5 accuracy regime has been tested and confirmed numerically for hundreds of thousand of points within the region 1E-12 < Re < 1E12 and 0 < eD < 0.1.

The numerical solution attempts the secant method using scipy’s `newton` solver, and in the event of nonconvergence, attempts the `fsolve` solver as well. An initial guess is provided via the Clamond function.

The numerical and analytical solution take similar amounts of time; the `mpmath` solution used when `tol=0` is approximately 45 times slower. This function takes approximately 8 us normally.

References

[1]

Examples

```python
>>> Colebrook(1E5, 1E-4)
0.018513866077471648
```

`fluids.friction.Clamond(Re, eD, fast=False)`

Calculates Darcy friction factor using a solution accurate to almost machine precision. Recommended very strongly. For details of the algorithm, see [1].

Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float] Relative roughness, [-]
- **fast** [bool, optional] If true, performs only one iteration, which gives roughly half the number of decimals of accuracy, [-]

Returns

- **fd** [float] Darcy friction factor [-]

Notes

This is a highly optimized function, 4 times faster than the solution using the LambertW function, and faster than many other approximations which are much less accurate.

The code used here is only slightly modified than that in [1], for further performance improvements.

For 10 < Re < 1E12, and 0 < eD < 0.01, this equation has been confirmed numerically to provide a solution to the Colebrook equation accurate to an rtol of 1E-9 or better - the same level of accuracy as the analytical solution to the Colebrook equation due to floating point precision.

Comparing this to the numerical solution of the Colebrook equation, identical values are given accurate to an rtol of 1E-9 for 10 < Re < 1E100, and 0 < eD < 1 and beyond.

However, for values of Re under 10, different answers from the Colebrook equation appear and then quickly a ValueError is raised.
References

[1]

Examples

```python
>>> Clamond(1E5, 1E-4)
0.01851386607747165
```

`fluids.friction.friction_laminar(Re)`
Calculates Darcy friction factor for laminar flow, as shown in [1] or anywhere else.

\[ f_d = \frac{64}{Re} \]

Parameters

- **Re** [float] Reynolds number, [-]

Returns

- **fd** [float] Darcy friction factor [-]

Notes

For round pipes, this valid for \( Re \approx < 2040 \).

Results in [2] show that this theoretical solution calculates too low of friction factors from \( Re = 10 \) and up, with an average deviation of 4%.

References

[1], [2]

Examples

```python
>>> friction_laminar(128)
0.5
```

`fluids.friction.one_phase_dP(m, rho, mu, D, roughness=0, L=1, Method=None)`
Calculates single-phase pressure drop. This is a wrapper around other methods.

Parameters

- **m** [float] Mass flow rate of fluid, [kg/s]
- **rho** [float] Density of fluid, [kg/m^3]
- **mu** [float] Viscosity of fluid, [Pa*s]
- **D** [float] Diameter of pipe, [m]
- **roughness** [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- **L** [float, optional] Length of pipe, [m]
- **Method** [string, optional] A string of the function name to use

2.9. Friction factor and pipe roughness (fluids.friction)
Returns

dP  [float] Pressure drop of the single-phase flow, [Pa]

References

[1]

Examples

```python
>>> one_phase_dP(10.0, 1000, 1E-5, .1, L=1)
63.43447321097365
```

`fluids.friction.one_phase_dP_gravitational(angle, rho, L=1.0, g=9.80665)`

This function handles calculation of one-phase liquid-gas pressure drop due to gravitation for flow inside channels. This is either a differential calculation for a segment with an infinitesimal difference in elevation (if `L'=1 or a discrete calculation.

\[- \left( \frac{dP}{dz} \right)_\text{grav} = \rho g \sin \theta \]

\[- (\Delta P)_\text{grav} = L \rho g \sin \theta \]

Parameters

- **angle**  [float] The angle of the pipe with respect to the horizontal, [degrees]
- **rho**  [float] Fluid density, [kg/m^3]
- **L**  [float, optional] Length of pipe, [m]
- **g**  [float, optional] Acceleration due to gravity, [m/s^2]

Returns

- **dP**  [float] Gravitational component of pressure drop for one-phase flow, [Pa/m] or [Pa]

Examples

```python
>>> one_phase_dP_gravitational(angle=90, rho=2.6)
25.49729
>>> one_phase_dP_gravitational(angle=90, rho=2.6, L=4)
101.98916
```

`fluids.friction.one_phase_dP_dz_acceleration(m, D, rho, dv_dP, dP_dL, dA_dL)`

This function handles calculation of one-phase fluid pressure drop due to acceleration for flow inside channels. This is a continuous calculation, providing the differential in pressure per unit length and should be called as part of an integration routine ([1], [2]).

\[- \left( \frac{\partial P}{\partial L} \right)_A = G^2 \frac{\partial P}{\partial L} \left[ \frac{\partial (1/\rho)}{\partial P} \right] - \frac{G^2}{\rho} \frac{1}{A} \frac{\partial A}{\partial L} \]

Parameters

- **m**  [float] Mass flow rate of fluid, [kg/s]
- **D**  [float] Diameter of pipe, [m]
\textbf{rho} [float] Fluid density, [kg/m^3]

\textbf{dv\_dP} [float] Derivative of mass specific volume of the fluid with respect to pressure, [m^3/(kg*Pa)]

\textbf{dP\_dL} [float] Pressure drop per unit length of pipe, [Pa/m]

\textbf{dA\_dL} [float] Change in area of pipe per unit length of pipe, [m^2/m]

\textbf{Returns}

\textbf{dP\_dz} [float] Acceleration component of pressure drop for one-phase flow, [Pa/m]

\textbf{Notes}

The value returned here is positive for pressure loss and negative for pressure increase.

As \textbf{dP\_dL} is not known, this equation is normally used in a more complicated way than this function provides; this method can be used to check the consistency of that routine.

\textbf{References}

[1]

\textbf{Examples}

\begin{verbatim}
>>> one_phase_dP_dz_acceleration(m=1, D=0.1, rho=827.1, dv_dP=-1.1E-5, dP_dL=5E5, dA_dL=0.0001)
89162.89116373913
\end{verbatim}

\texttt{fluids.friction.one_phase_dP_acceleration}\( (m, D, \text{rho\_o}, \text{rho\_i}) \)

This function handles calculation of one-phase fluid pressure drop due to acceleration for flow inside channels. This is a discrete calculation, providing the total differential in pressure for a given length and should be called as part of a segment solver routine.

\[- \frac{dP}{dz}_{acc} = G^2 \frac{d}{dz} \left( \frac{1}{\rho_o} - \frac{1}{\rho_i} \right)\]

\textbf{Parameters}

\begin{itemize}
  \item \textbf{m} [float] Mass flow rate of fluid, [kg/s]
  \item \textbf{D} [float] Diameter of pipe, [m]
  \item \textbf{rho\_o} [float] Fluid density out, [kg/m^3]
  \item \textbf{rho\_i} [float] Fluid density int, [kg/m^3]
\end{itemize}

\textbf{Returns}

\textbf{dP} [float] Acceleration component of pressure drop for one-phase flow, [Pa]

\textbf{Examples}

\begin{verbatim}
>>> one_phase_dP_acceleration(m=1, D=0.1, rho_o=827.1, rho_i=830)
0.06848289670840459
\end{verbatim}

2.9. Friction factor and pipe roughness (fluids.friction)
**Fluids Documentation, Release 0.1**

**fluids.friction.transmission_factor**(fd=None, F=None)

Calculates either transmission factor from Darcy friction factor, or Darcy friction factor from the transmission factor. Raises an exception if neither input is given.

Transmission factor is a term used in compressible gas flow in pipelines.

\[ F = \frac{2}{\sqrt{f_d}} \]
\[ f_d = \frac{4}{F^2} \]

**Parameters**

- **fd** [float, optional] Darcy friction factor, [-]
- **F** [float, optional] Transmission factor, [-]

**Returns**

- **fd** or **F** [float] Darcy friction factor or transmission factor [-]

**References**

[1]

**Examples**

```python
>>> transmission_factor(fd=0.0185)
14.704292441876154

>>> transmission_factor(F=20)
0.01
```

**fluids.friction.material_roughness**(ID, D=None, optimism=None)

Searches through either a dict of clean pipe materials or used pipe materials and conditions and returns the ID of the nearest material. Search is performed with either the standard library’s difflib or with the fuzzywuzzy module if available.

**Parameters**

- **ID** [str] Search terms for matching pipe materials, [-]
- **D** [float, optional] Diameter of desired pipe; used only if ID is in [2], [m]
- **optimism** [bool, optional] For values in [1], a minimum, maximum, and average value is normally given; if True, returns the minimum roughness; if False, the maximum roughness; and if None, returns the average roughness. Most entries do not have all three values, so fallback logic to return the closest entry is used, [-]

**Returns**

- **roughness** [float] Retrieved or calculated roughness, [m]

**References**

[1], [2]
Examples

```python
>>> material_roughness('condensate pipes')
0.0005
```

**fluids.friction.nearest_material_roughness**(name, clean=None)

Searches through either a dict of clean pipe materials or used pipe materials and conditions and returns the ID of the nearest material. Search is performed with either the standard library’s difflib or with the fuzzywuzzy module if available.

**Parameters**

- **name** [str] Search term for matching pipe materials
- **clean** [bool, optional] If True, search only clean pipe database; if False, search only the dirty database; if None, search both

**Returns**

- **ID** [str] String for lookup of roughness of a pipe, in either `roughness_clean_dict` or `HHR_roughness` depending on if clean is True.

**References**

[1]

**Examples**

```python
>>> nearest_material_roughness('condensate pipes', clean=False)
'Seamless steel tubes, Condensate pipes in open systems or periodically operated...
steam pipelines'
```

**fluids.friction.roughness_Farshad**(ID=None, D=None, coeffs=None)

Calculates or retrieves the roughness of a pipe based on the work of [1]. This function will return an average value for pipes of a given material, or if diameter is provided, will calculate one specifically for the pipe inner diameter according to the following expression with constants $A$ and $B$:

\[
\epsilon = A \cdot D^{B+1}
\]

Please note that $A$ has units of inches, and $B$ requires $D$ to be in inches as well.

The list of supported materials is as follows:

- ‘Plastic coated’
- ‘Carbon steel, honed bare’
- ‘Cr13, electropolished bare’
- ‘Cement lining’
- ‘Carbon steel, bare’
- ‘Fiberglass lining’
- ‘Cr13, bare’

If `coeffs` and `D` are given, the custom coefficients for the equation as given by the user will be used and `ID` is not required.

---

2.9. Friction factor and pipe roughness (fluids.friction)
Parameters

ID  [str, optional] Name of pipe material from above list
D  [float, optional] Actual inner diameter of pipe, [m]
coeffs  [tuple, optional] (A, B) Coefficients to use directly, instead of looking them up; they are actually dimensional, in the forms (inch^-B, -) but only coefficients with those dimensions are available [-]

Returns

epsilon  [float] Roughness of pipe [m]

Notes

The diameter-dependent form provides lower roughness values for larger diameters.
The measurements were based on DIN 4768/1 (1987), using both a “Dektak ST Surface Profiler” and a “Hommel Tester T1000”. Both instruments were found to be in agreement. A series of flow tests, in which pressure drop directly measured, were performed as well, with nitrogen gas as an operating fluid. The accuracy of the data from these tests is claimed to be within 1%.

Using those results, the authors back-calculated what relative roughness values would be necessary to produce the observed pressure drops. The average difference between this back-calculated roughness and the measured roughness was 6.75%.

For microchannels, this model will predict roughness much larger than the actual channel diameter.

References

[1]

Examples

```python
>>> roughness_Farshad('Cr13, bare', 0.05)
5.3141677781137006e-05
```

fluids.friction.Moody (Re, eD)
Calculates Darcy friction factor using the method in Moody (1947) as shown in [1] and originally in [2].

\[
    f_f = 1.375 \times 10^{-3} \left[ 1 + \left( 2 \times 10^4 \frac{\varepsilon}{D} + \frac{10^6}{Re} \right)^{1/3} \right]
\]

Parameters

Re  [float] Reynolds number, [-]
eD  [float] Relative roughness, [-]

Returns

fd  [float] Darcy friction factor [-]

Notes

Range is Re >= 4E3 and Re <= 1E8; eD >= 0 < 0.01.
References

[1], [2]

Examples

```python
>>> Moody(1E5, 1E-4)
0.01809185666808665
```

```python
fluids.friction.Alshul_1952(Re, eD)
Calculates Darcy friction factor using the method in Alshul (1952) as shown in [1].

\[ f_d = 0.11 \left( \frac{68}{Re} + \frac{\epsilon}{D} \right)^{0.25} \]

Parameters

- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

Returns

- fd [float] Darcy friction factor [-]

Notes

No range of validity specified for this equation.

References

[1]

Examples

```python
>>> Alshul_1952(1E5, 1E-4)
0.018382997825686878
```

```python
fluids.friction.Wood_1966(Re, eD)
Calculates Darcy friction factor using the method in Wood (1966) [2] as shown in [1].

\[ f_d = 0.094 \left( \frac{\epsilon}{D} \right)^{0.225} + 0.53 \left( \frac{\epsilon}{D} \right) + 88 \left( \frac{\epsilon}{D} \right)^{0.4} Re^{-A_1} \]

\[ A_1 = 1.62 \left( \frac{\epsilon}{D} \right)^{0.134} \]

Parameters

- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

Returns

- fd [float] Darcy friction factor [-]
Notes

Range is 4E3 <= Re <= 5E7; 1E-5 <= eD <= 4E-2.

References

[1], [2]

Examples

```python
>>> Wood_1966(1E5, 1E-4)
0.021587570560090762
```

```
fluids.friction.Churchill_1973(Re, eD)
Calculates Darcy friction factor using the method in Churchill (1973) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_d}} = -2 \log \left( \frac{\epsilon}{3.7D} + \left( \frac{7}{Re} \right)^{0.9} \right)
\]

Parameters

- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

Returns

- fd [float] Darcy friction factor [-]

Notes

No range of validity specified for this equation.

References

[1], [2]

Examples

```python
>>> Churchill_1973(1E5, 1E-4)
0.01846708694482294
```

```
fluids.friction.Eck_1973(Re, eD)
Calculates Darcy friction factor using the method in Eck (1973) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_d}} = -2 \log \left( \frac{\epsilon}{3.715D} + \frac{15}{Re} \right)
\]

Parameters

- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

Returns

- fd [float] Darcy friction factor [-]
Notes

No range of validity specified for this equation.

References

[1],[2]

Examples

```python
>>> Eck_1973(1E5, 1E-4)
0.01775666973488564
```

`fluids.friction.Jain_1976(Re, eD)`
Calculates Darcy friction factor using the method in Jain (1976) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_f}} = 2.28 - 4 \log \left[ \frac{\epsilon}{D} + \left( \frac{29.843}{Re} \right)^{0.9} \right]
\]

Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float] Relative roughness, [-]

Returns

- **fd** [float] Darcy friction factor [-]

Notes

Range is 5E3 <= Re <= 1E7; 4E-5 <= eD <= 5E-2.

References

[1],[2]

Examples

```python
>>> Jain_1976(1E5, 1E-4)
0.018436560312693327
```

`fluids.friction.Swamee_Jain_1976(Re, eD)`
Calculates Darcy friction factor using the method in Swamee and Jain (1976) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_f}} = -4 \log \left[ \left( \frac{6.97}{Re} \right)^{0.9} + \left( \frac{\epsilon}{3.7D} \right) \right]
\]

Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float] Relative roughness, [-]
Returns

\( \text{fd} \) [float] Darcy friction factor [-]

Notes

Range is 5E3 <= Re <= 1E8; 1E-6 <= eD <= 5E-2.

References

[1], [2]

Examples

```python
>>> Swamee_Jain_1976(1E5, 1E-4)
0.018452424431901808
```

\texttt{fluids.friction.Churchill\_1977(Re, eD)}

Calculates Darcy friction factor using the method in Churchill and (1977) [2] as shown in [1].

\[
f_f = 2 \left[ \frac{8}{Re} \right]^{12} + (A_2 + A_3)^{-1.5} \right]^{1/12}
\]

\[
A_2 = \left\{ 2.457 \ln \left[ \frac{7}{Re} \right]^{0.9} + 0.27 \frac{\epsilon}{D} \right\}^{16}
\]

\[
A_3 = \left( \frac{37530}{Re} \right)^{16}
\]

Parameters

\texttt{Re} [float] Reynolds number, [-]

\texttt{eD} [float] Relative roughness, [-]

Returns

\( \text{fd} \) [float] Darcy friction factor [-]

Notes

No range of validity specified for this equation.

References

[1], [2]
Examples

```python
>>> Churchill_1977(1E5, 1E-4)
0.018462624566280075
```

`fluids.friction.Chen_1979(Re, eD)`
Calculates Darcy friction factor using the method in Chen (1979) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_f}} = -4\log \left[ \frac{\epsilon}{3.7065D} - \frac{5.0452}{Re} \log A_4 \right]
\]

\[
A_4 = \left( \frac{\epsilon}{D} \right)^{1.098} \left( \frac{7.149}{Re} \right)^{0.8981} + \frac{2.8257}{2.8257}
\]

Parameters
- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

Returns
- fd [float] Darcy friction factor [-]

Notes
Range is 4E3 <= Re <= 4E8; 1E-7 <= eD <= 5E-2.

References
[1], [2]

Examples

```python
>>> Chen_1979(1E5, 1E-4)
0.018552817507472126
```

`fluids.friction.Round_1980(Re, eD)`
Calculates Darcy friction factor using the method in Round (1980) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_f}} = -3.6\log \left[ \frac{Re}{0.135Re + 6.5} \right]
\]

Parameters
- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

Returns
- fd [float] Darcy friction factor [-]

Notes
Range is 4E3 <= Re <= 4E8; 0 <= eD <= 5E-2.
References

[1], [2]

Examples

```python
>>> Round_1980(1E5, 1E-4)
0.01831475391244354
```

```
fluids.friction.Shacham_1980(Re, eD)
Calculates Darcy friction factor using the method in Shacham (1980) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_d}} = -4 \log \left[ \frac{\epsilon}{3.7D} - \frac{5.02}{Re} \log \left( \frac{\epsilon}{3.7D} + 14.5 \right) \right]
\]

Parameters
- `Re` [float] Reynolds number, [-]
- `eD` [float] Relative roughness, [-]

Returns
- `fd` [float] Darcy friction factor [-]

Notes

Range is 4E3 <= Re <= 4E8

References

[1], [2]

Examples

```python
>>> Shacham_1980(1E5, 1E-4)
0.01860641215097828
```

```
fluids.friction.Barr_1981(Re, eD)
Calculates Darcy friction factor using the method in Barr (1981) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_d}} = -2 \log \left\{ \frac{\epsilon}{3.7D} + \frac{4.518 \log(Re)}{Re \left[ 1 + \frac{Re^{0.52}}{29} \left( \frac{\epsilon}{D} \right)^{0.7} \right]} \right\}
\]

Parameters
- `Re` [float] Reynolds number, [-]
- `eD` [float] Relative roughness, [-]

Returns
- `fd` [float] Darcy friction factor [-]
Notes

No range of validity specified for this equation.

References

[1], [2]

Examples

```python
>>> Barr_1981(1E5, 1E-4)
0.01849836032779929
```

`fluids.friction.Zigrang_Sylvester_1(Re, eD)`

Calculates Darcy friction factor using the method in Zigrang and Sylvester (1982) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_f}} = -4 \log \left[ \frac{\epsilon}{3.7D} - \frac{5.02}{Re} \log A_5 \right]
\]

\[
A_5 = \frac{\epsilon}{3.7D} + 13 \frac{Re}{D}
\]

Parameters

- `Re [float]` Reynolds number, [-]
- `eD [float]` Relative roughness, [-]

Returns

- `fd [float]` Darcy friction factor [-]

Notes

Range is 4E3 \( \leq \text{Re} \leq 1\text{E8}; 4\text{E}-5 \leq \text{eD} \leq 5\text{E}-2.\)

References

[1], [2]

Examples

```python
>>> Zigrang_Sylvester_1(1E5, 1E-4)
0.018646892425980794
```

`fluids.friction.Zigrang_Sylvester_2(Re, eD)`

\[
\frac{1}{\sqrt{f}} = -4 \log \left[ \frac{\epsilon}{3.7D} - \frac{5.02}{Re} \log A_6 \right] \\
A_6 = \frac{\epsilon}{3.7D} - \frac{5.02}{Re} \log A_5 \\
A_5 = \frac{\epsilon}{3.7D} + \frac{13}{Re}
\]

**Parameters**
- \(Re\) [float] Reynolds number, [-]
- \(eD\) [float] Relative roughness, [-]

**Returns**
- \(fd\) [float] Darcy friction factor [-]

**Notes**
- Range is \(4E3 \leq Re \leq 1E8; 4E-5 \leq eD \leq 5E-2\)

**References**
- [1], [2]

**Examples**

```python
>>> Zigrang_Sylvester_2(1E5, 1E-4)
0.01850021312358548
```

\(\text{fluids.friction.Haaland}(Re, eD)\)

Calculates Darcy friction factor using the method in Haaland (1983) [2] as shown in [1].

\[
f_f = \left(-1.8 \log_{10} \left[ \left( \frac{\epsilon/D}{3.7} \right)^{1.11} + \frac{6.9}{Re} \right] \right)^{-2}
\]

**Parameters**
- \(Re\) [float] Reynolds number, [-]
- \(eD\) [float] Relative roughness, [-]

**Returns**
- \(fd\) [float] Darcy friction factor [-]

**Notes**
- Range is \(4E3 \leq Re \leq 1E8; 1E-6 \leq eD \leq 5E-2\)

**References**
- [1], [2]
Examples

```python
>>> Haaland(1E5, 1E-4)
0.018265053014793857
```

**fluids.friction.** Serghides\_1 (Re, eD)

\[
f = \left[A - \frac{(B - A)^2}{C - 2B + A}\right]^{-2}
\]

\[
A = -2 \log_{10} \left[\frac{\epsilon/D}{3.7} + \frac{12}{Re}\right]
\]

\[
B = -2 \log_{10} \left[\frac{\epsilon/D}{3.7} + \frac{2.51A}{Re}\right]
\]

\[
C = -2 \log_{10} \left[\frac{\epsilon/D}{3.7} + \frac{2.51B}{Re}\right]
\]

**Parameters**
- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

**Returns**
- fd [float] Darcy friction factor [-]

**Notes**
No range of validity specified for this equation.

**References**
[1], [2]

**Examples**

```python
>>> Serghides_1(1E5, 1E-4)
0.01851358983180063
```

**fluids.friction.** Serghides\_2 (Re, eD)

\[
f_d = \left[4.781 - \frac{(A - 4.781)^2}{B - 2A + 4.781}\right]^{-2}
\]

\[
A = -2 \log_{10} \left[\frac{\epsilon/D}{3.7} + \frac{12}{Re}\right]
\]

\[
B = -2 \log_{10} \left[\frac{\epsilon/D}{3.7} + \frac{2.51A}{Re}\right]
\]

**Parameters**
fluids Documentation, Release 0.1

Re [float] Reynolds number, [-]
eD [float] Relative roughness, [-]

Returns
fd [float] Darcy friction factor [-]

Notes
No range of validity specified for this equation.

References
[1], [2]

Examples
>>> Serghides_2(1E5, 1E-4)
0.018486377560664482

fluids.friction.Tsal_1989(Re, eD)
Calculates Darcy friction factor using the method in Tsal (1989) [2] as shown in [1].

\[ A = 0.11 \left( \frac{68}{Re} + \frac{\epsilon}{D} \right)^{0.25} \]

if \( A \geq 0.018 \) then \( fd = A \);
if \( A < 0.018 \) then \( fd = 0.0028 + 0.85A \).

Parameters
Re [float] Reynolds number, [-]
eD [float] Relative roughness, [-]

Returns
fd [float] Darcy friction factor [-]

Notes
Range is 4E3 <= Re <= 1E8; 0 <= eD <= 5E-2

References
[1], [2]

Examples
>>> Tsal_1989(1E5, 1E-4)
0.018382997825686878
Calculates Darcy friction factor using the method in Manadilli (1997) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_d}} = -2 \log \left[ \frac{\epsilon}{3.7D} + \frac{95}{Re^{0.983}} - \frac{96.82}{Re} \right]
\]

Parameters

- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

Returns

- fd [float] Darcy friction factor [-]

Notes

Range is 5.245E3 <= Re <= 1E8; 0 <= eD <= 5E-2

References

[1], [2]

Examples

```python
>>> Manadilli_1997(1E5, 1E-4)
0.01856964649724108
```

Calculates Darcy friction factor using the method in Romeo (2002) [2] as shown in [1].

\[
\frac{1}{\sqrt{f_d}} = -2 \log \left[ \frac{\epsilon}{3.7065D} \times \frac{5.0272}{Re} \times \log \left( \frac{\epsilon}{3.827D} - \frac{4.567}{Re} \times \log \left( \frac{\epsilon}{7.7918D} \times \frac{0.9924}{208.815 + Re} + \left( \frac{5.3326}{0.9345} \right) \right) \right) \right]
\]

Parameters

- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

Returns

- fd [float] Darcy friction factor [-]

Notes

Range is 3E3 <= Re <= 1.5E8; 0 <= eD <= 5E-2

References

[1], [2]
Examples

```python
>>> Romeo_2002(1E5, 1E-4)
0.018530291219676177
```

```
fluids.friction.Sonnad_Goudar_2006 (Re, eD)

\[
\frac{1}{\sqrt{F_d}} = 0.8686 \ln \left( \frac{0.4587 Re}{S/(S+1)} \right)
\]

\[
S = 0.1240 \times \frac{\epsilon}{D} \times Re + \ln(0.4587 Re)
\]

Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float] Relative roughness, [-]

Returns

- **fd** [float] Darcy friction factor [-]

Notes

Range is 4E3 <= Re <= 1E8; 1E-6 <= eD <= 5E-2

References

[1], [2]

Examples

```python
>>> Sonnad_Goudar_2006(1E5, 1E-4)
0.0185971269898162
```

```
fluids.friction.Rao_Kumar_2007 (Re, eD)

\[
\frac{1}{\sqrt{F_d}} = 2 \log \left( \frac{(2 \frac{\epsilon}{D})^{-1}}{\left( \frac{0.444+0.135 Re}{Re} \right) \beta} \right)
\]

\[
\beta = 1 - 0.55 \exp(-0.33 \ln \left[ \frac{Re}{6.5} \right]^{2})
\]

Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float] Relative roughness, [-]

Returns

- **fd** [float] Darcy friction factor [-]
Notes

No range of validity specified for this equation. This equation is fit to original experimental friction factor data. Accordingly, this equation should not be used unless appropriate consideration is given.

References

[1], [2]

Examples

```python
>>> Rao_Kumar_2007(1E5, 1E-4)
0.01197759334600925
```

```
fluids.friction.Buzzelli_2008(Re, eD)

\[
\frac{1}{\sqrt{f_D}} = B_1 - \left[ \frac{B_1 + 2 \log\left(\frac{B_2}{Re}\right)}{1 + \frac{2.12}{B_2}} \right]
\]

\[
B_1 = \frac{0.774 \ln(Re) - 1.41}{1 + 1.32 \sqrt{\frac{\epsilon}{D}}}
\]

\[
B_2 = \frac{\epsilon}{3.7 D} Re + 2.51 \times B_1
\]

Parameters

- Re [float] Reynolds number, [-]
- eD [float] Relative roughness, [-]

Returns

- fd [float] Darcy friction factor [-]

Notes

No range of validity specified for this equation.

References

[1], [2]

Examples

```python
>>> Buzzelli_2008(1E5, 1E-4)
0.01851394840136527
```

```
fluids.friction.Avci_Karagoz_2009(Re, eD)
Calculates Darcy friction factor using the method in Avci and Karagoz (2009) [2] as shown in [1].

\[
f_D = \left\{ \ln(Re) - \ln \left[ 1 + 0.01 Re \frac{\frac{\epsilon}{D}}{1 + 0.01 \left(\frac{\epsilon}{D}\right)^{0.5}} \right] \right\}^{2.4}
\]

2.9. Friction factor and pipe roughness (fluids.friction)
Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float] Relative roughness, [-]

Returns

- **fd** [float] Darcy friction factor [-]

Notes

No range of validity specified for this equation.

References

[1], [2]

Examples

```python
>>> Avci_Karagoz_2009(1E5, 1E-4)
0.01857058061066499
```

`fluids.friction.Papaevangelo_2010(Re, eD)`


\[
    f_D = \frac{0.2479 - 0.0000947(7 - \log Re)^4}{\left[\log\left(\frac{\epsilon}{3.615D} + \frac{7.366}{Re^{0.914}}\right)\right]^2}
\]

Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float] Relative roughness, [-]

Returns

- **fd** [float] Darcy friction factor [-]

Notes

Range is 1E4 <= Re <= 1E7; 1E-5 <= eD <= 1E-3

References

[1], [2]

Examples

```python
>>> Papaevangelo_2010(1E5, 1E-4)
0.015685600818488177
```
**fluids.friction.Brkic_2011_1** (*Re, eD*)

Calculates Darcy friction factor using the method in Brkic (2011) [2] as shown in [1].

\[
f_d = \left[ -2 \log\left(10^{-0.4343\beta + \frac{\epsilon}{3.71D}}\right) \right]^{-2}
\]

\[
\beta = \ln \left( \frac{Re}{1.816 \ln \left( \frac{1.1Re}{\ln(1+1.1Re)} \right)} \right)
\]

**Parameters**
- *Re* [float] Reynolds number, [-]
- *eD* [float] Relative roughness, [-]

**Returns**
- *fd* [float] Darcy friction factor [-]

**Notes**

No range of validity specified for this equation.

**References**

[1], [2]

**Examples**

```python
>>> Brkic_2011_1(1E5, 1E-4)
0.01812455874141297
```

**fluids.friction.Brkic_2011_2** (*Re, eD*)

Calculates Darcy friction factor using the method in Brkic (2011) [2] as shown in [1].

\[
f_d = \left[ -2 \log\left(2.18\beta \frac{Re}{3.71D} + \frac{\epsilon}{3.71D} \right) \right]^{-2}
\]

\[
\beta = \ln \left( \frac{Re}{1.816 \ln \left( \frac{1.1Re}{\ln(1+1.1Re)} \right)} \right)
\]

**Parameters**
- *Re* [float] Reynolds number, [-]
- *eD* [float] Relative roughness, [-]

**Returns**
- *fd* [float] Darcy friction factor [-]

**Notes**

No range of validity specified for this equation.
References

[1], [2]

Examples

```python
>>> Brkic_2011_2(1E5, 1E-4)
0.018619745410688716
```

**fluids.friction.Fang_2011**(Re, eD)
Calculates Darcy friction factor using the method in Fang (2011) [2] as shown in [1].

\[
f_D = 1.613 \left\{ \ln \left[ 0.234 \frac{\epsilon}{D}^{1.1007} - \frac{60.525}{R_e^{1.1105}} + \frac{56.291}{R_e^{1.0712}} \right] \right\}^{-2}
\]

Parameters

- **Re** [float] Reynolds number, [-]
- **eD** [float] Relative roughness, [-]

Returns

- **fd** [float] Darcy friction factor [-]

Notes

Range is 3E3 <= Re <= 1E8; 0 <= eD <= 5E-2

References

[1], [2]

Examples

```python
>>> Fang_2011(1E5, 1E-4)
0.018481390682985432
```

**fluids.friction.Blasius**(Re)
Calculates Darcy friction factor according to the Blasius formulation, originally presented in [1] and described more recently in [2].

\[
f_d = \frac{0.3164}{R_e^{0.25}}
\]

Parameters

- **Re** [float] Reynolds number, [-]

Returns

- **fd** [float] Darcy friction factor [-]

Notes

Developed for 3000 < Re < 200000.
References

[1], [2]

Examples

```python
>>> Blasius(10000)
0.03164
```

```
fluids.friction.von_Karman(eD)
Calculates Darcy friction factor for rough pipes at infinite Reynolds number from the von Karman equation (as given in [1] and [2]):

\[
\frac{1}{\sqrt{f_d}} = -2 \log_{10} \left( \frac{e/D}{3.7} \right)
\]

Parameters

- eD [float] Relative roughness, [-]

Returns

- fd [float] Darcy friction factor [-]

Notes

This case does not actually occur; Reynolds number is always finite. It is normally applied as a “limiting” value when a pipe’s roughness is so high it has a friction factor curve effectively independent of Reynolds number.

References

[1], [2]

Examples

```python
>>> von_Karman(1E-4)
0.01197365149564789
```

```
fluids.friction.Prandtl_von_Karman_Nikuradse(Re)
Calculates Darcy friction factor for smooth pipes as a function of Reynolds number from the Prandtl-von Karman Nikuradse equation as given in [1] and [2]:

\[
\frac{1}{\sqrt{f}} = -2 \log_{10} \left( \frac{2.51}{Re \sqrt{f}} \right)
\]

Parameters

- Re [float] Reynolds number, [-]

Returns

- fd [float] Darcy friction factor [-]
Notes

This equation is often stated as follows; the correct constant is not 0.8, but 2\log_{10}(2.51) or approximately 0.7993474:

\[ \frac{1}{\sqrt{f}} \approx 2 \log_{10}(\text{Re} \sqrt{f}) - 0.8 \]

This function is calculable for all Reynolds numbers between 1E151 and 1E-151. It is solved with the LambertW function from SciPy. The solution is:

\[ f_d = \frac{\frac{1}{4} \log_{10}^2}{\left( \text{lambertW} \left( \frac{\log(10)}{2(2.51)} \right) \right)^2} \]

References

[1], [2]

Examples

```python
>>> Prandtl_von_Karman_Nikuradse(1E7)
0.008102669430874914
```

**fluido\textunderscore friction\textunderscore ft\_Crane(D)**

Calculates the Crane fully turbulent Darcy friction factor for flow in commercial pipe, as used in the Crane formulas for loss coefficients in various fittings. Note that this is not generally applicable to loss due to friction in pipes, as it does not take into account the roughness of various pipe materials. But for fittings in any type of pipe, this is the friction factor to use in the Crane [1] method to get their loss coefficients.

**Parameters**

- D [float] Pipe inner diameter, [m]

**Returns**

- fd [float] Darcy Crane friction factor for fully turbulent flow, [-]

**Notes**

There is confusion and uncertainty regarding the friction factor table given in Crane TP 410M [1]. This function does not help: it implements a new way to obtain Crane friction factors, so that it can better be based in theory and give more precision (not accuracy) and trend better with diameters not tabulated in [1].

The data in [1] was digitized, and nominal pipe diameters were converted to actual pipe diameters. An objective function was sought which would produce the exact same values as in [1] when rounded to the same decimal place. One was found fairly easily by using the standard Colebrook friction factor formula, and using the diameter-dependent roughness values calculated with the roughness\_Farshad method for bare Carbon steel. A diameter-dependent Reynolds number was required to match the values; the \( \rho V/\mu \) term is set to 7.5E6.

The formula given in [1] is:

\[ f_T = \frac{0.25}{\left( \log_{10} \left( \frac{\varepsilon/D}{4}\right) \right)^2} \]
However, this function does not match the rounded values in [1] well and it is not very clear which roughness to use. Using both the value for new commercial steel (.05 mm) or a diameter-dependent value (roughness_Farshad), values were found to be too high and too low respectively. That function is based in theory - the limit of the Colebrook equation when $Re$ goes to infinity - but in the end real pipe flow is not infinity, and so a large error occurs from that use.

The following plot shows all these options, and that the method implemented here matches perfectly the rounded values in [1].

![Comparison of implementation options](chart.png)

**References**

[1]

**Examples**

```python
>> f_Crane(.1)
0.01628845962146481
```

Explicitly spelling out the function (note the exact same answer is not returned; it is accurate to 5-8 decimals however, for increased speed):
Di = 0.1
Colebrook(7.5E6*Di, eD=roughness_Farshad(ID='Carbon steel, bare', D=Di)/Di)
0.01628842543122547

**fluids.friction.helical_laminar_fd_White***(Re, Di, Dc)***

Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under laminar conditions, using the method of White [1] as shown in [2].

\[
f_{\text{curved}} = f_{\text{straight,laminar}} \left[ 1 - \left( 1 - \left( \frac{11.6}{De} \right)^{0.45} \right)^{\frac{1}{10.5}} \right]^{-1}
\]

**Parameters**

- **Re** [float] Reynolds number with \(D=Di\), [-]
- **Di** [float] Inner diameter of the coil, [m]
- **Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

**Returns**

- **fd** [float] Darcy friction factor for a curved pipe [-]

**Notes**

The range of validity of this equation is \(11.6 < De < 2000\), \(3.878 \times 10^{-4} < D_i/D_c < 0.066\).

The form of the equation means it yields nonsense results for \(De < 11.6\); at \(De < 11.6\), the equation is modified to return the straight pipe value.

This model is recommended in [3], with a slight modification for Dean numbers larger than 2000.

**References**

[1], [2], [3]

**Examples**

```python
>>> helical_laminar_fd_White(250, .02, .1)
0.4063281817830202
```

**fluids.friction.helical_laminar_fd_Mori_Nakayama***(Re, Di, Dc)***

Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under laminar conditions, using the method of Mori and Nakayama [1] as shown in [2] and [3].

\[
f_{\text{curved}} = f_{\text{straight,laminar}} \left( \frac{0.108 \sqrt{De}}{1 - 3.253 De^{-0.5}} \right)
\]

**Parameters**

- **Re** [float] Reynolds number with \(D=Di\), [-]
- **Di** [float] Inner diameter of the coil, [m]
**Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

**Returns**

**fd** [float] Darcy friction factor for a curved pipe [-]

**Notes**

The range of validity of this equation is 100 < De < 2000.

The form of the equation means it yields nonsense results for De < 42.328; under that, the equation is modified to return the value at De=42.328, which is a multiplier of 1.405296 on the straight pipe friction factor.

**References**

[1], [2], [3]

**Examples**

```python
>>> helical_laminar_fd_Mori_Nakayama(250, .02, .1)
0.4224582857779544
```

**fluilds.friction.helical_laminar_fd_Schmidt** *(Re, Di, Dc)*

Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under laminar conditions, using the method of Schmidt [1] as shown in [2] and [3].

\[ f_{\text{curved}} = f_{\text{straight,laminar}} \left[ 1 + 0.14 \left( \frac{D_i}{D_c} \right)^{0.97} Re^{1 - 0.644 \left( \frac{D_i}{D_c} \right)^{0.312}} \right] \]

**Parameters**

- **Re** [float] Reynolds number with \( D=D_i \), [-]
- **Di** [float] Inner diameter of the coil, [m]
- **Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

**Returns**

**fd** [float] Darcy friction factor for a curved pipe [-]

**Notes**

The range of validity of this equation is specified only for \( Re \), 100 < Re < Re_{critical}.

The form of the equation is such that as the curvature becomes negligible, straight tube result is obtained.

**References**

[1], [2], [3]
### Examples

```python
>>> helical_laminar_fd_Schmidt(250, .02, .1)
0.47460725672835236
```

fluids.friction.helical_turbulent_fd_Schmidt \((Re, Di, Dc, roughness=0)\)

Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under turbulent conditions, using the method of Schmidt [1], also shown in [2].

For \(Re_{crit} < Re < 2.2 \times 10^4\):

\[
f_{curv} = f_{str,turb} \left[ 1 + \frac{2.88 \times 10^4}{Re} \left( \frac{D_i}{D_c} \right)^{0.62} \right]
\]

For \(2.2 \times 10^4 < Re < 1.5 \times 10^5\):

\[
f_{curv} = f_{str,turb} \left[ 1 + 0.0823 \left( 1 + \frac{D_i}{D_c} \right) \left( \frac{D_i}{D_c} \right)^{0.53} Re^{0.25} \right]
\]

**Parameters**

- \(Re\) [float] Reynolds number with \(D=Di\), [-]
- \(Di\) [float] Inner diameter of the coil, [m]
- \(Dc\) [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]
- \(roughness\) [float, optional] Roughness of pipe wall [m]

**Returns**

- \(fd\) [float] Darcy friction factor for a curved pipe [-]

**Notes**

Valid from the transition to turbulent flow up to \(Re = 1.5 \times 10^5\). At very low curvatures, converges on the straight pipe result.

**References**

[1], [2]

**Examples**

```python
>>> helical_turbulent_fd_Schmidt(1E4, 0.01, .02)
0.08875550767040916
```

fluids.friction.helical_turbulent_fd_Mori_Nakayama \((Re, Di, Dc)\)

Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under turbulent conditions, using the method of Mori and Nakayama [1], also shown in [2] and [3].

\[
f_{curv} = 0.3 \left( \frac{D_i}{D_c} \right)^{0.5} \left[ Re \left( \frac{D_i}{D_c} \right)^2 \right]^{-0.2} \left[ 1 + 0.112 \left( Re \left( \frac{D_i}{D_c} \right)^2 \right)^{-0.2} \right]
\]
Parameters

- **Re** [float] Reynolds number with $D=Di$, [-]
- **Di** [float] Inner diameter of the coil, [m]
- **Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

Returns

- **fd** [float] Darcy friction factor for a curved pipe [-]

Notes

Valid from the transition to turbulent flow up to $Re = 6.5 \times 10^5 \sqrt{Di/Dc}$. Does not use a straight pipe correlation, and so will not converge on the straight pipe result at very low curvature.

References

[1], [2], [3]

Examples

```python
>>> helical_turbulent_fd_Mori_Nakayama(1E4, 0.01, .2)
0.037311802071379796
```

Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under turbulent conditions, using the method of Prasad [1], also shown in [2].

$$f_{curv} = f_{str,turb} \left[1 + 0.18 \left(Re \left(\frac{Di}{Dc}\right)^2\right)^{0.25}\right]$$

Parameters

- **Re** [float] Reynolds number with $D=Di$, [-]
- **Di** [float] Inner diameter of the coil, [m]
- **Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]
- **roughness** [float, optional] Roughness of pipe wall [m]

Returns

- **fd** [float] Darcy friction factor for a curved pipe [-]

Notes

No range of validity was specified, but the experiments used were with coil/tube diameter ratios of 17.24 and 34.9, hot water in the tube, and $1780 < Re < 59500$. At very low curvatures, converges on the straight pipe result.
References

[1], [2]

Examples

```python
>>> helical_turbulent_fd_Prasad(1E4, 0.01, .2)
0.043313098093994626
```

```python
fluids.friction.helical_turbulent_fd_Czop(Re, Di, Dc)
Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under turbulent conditions, using the method of Czop [1], also shown in [2].

\[ f_{\text{curve}} = 0.096 De^{-0.1517} \]

Parameters

- `Re` [float] Reynolds number with \( D=Di \), [-]
- `Di` [float] Inner diameter of the coil, [m]
- `Dc` [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

Returns

- `fd` [float] Darcy friction factor for a curved pipe [-]

Notes

Valid for \( 2 \times 10^4 < Re < 1.5 \times 10^5 \). Does not use a straight pipe correlation, and so will not converge on the straight pipe result at very low curvature.

References

[1], [2]

Examples

```python
>>> helical_turbulent_fd_Czop(1E4, 0.01, .2)
0.02979575250574106
```

```python
fluids.friction.helical_turbulent_fd_Guo(Re, Di, Dc)
Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under turbulent conditions, using the method of Guo [1], also shown in [2].

\[ f_{\text{curve}} = 0.638Re^{-0.15} \left( \frac{D_i}{D_c} \right)^{0.51} \]

Parameters

- `Re` [float] Reynolds number with \( D=Di \), [-]
- `Di` [float] Inner diameter of the coil, [m]
**Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

**Returns**

**fd** [float] Darcy friction factor for a curved pipe [-]

**Notes**

Valid for $2 \times 10^4 < Re < 1.5 \times 10^5$. Does not use a straight pipe correlation, and so will not converge on the straight pipe result at very low curvature.

**References**

[1], [2]

**Examples**

```python
guofd(2E5, 0.01, .2)
0.022189161013253147
```

_fluids.friction_.**helical_turbulent_fd_Ju**(Re, Di, Dc, roughness=0)

Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under turbulent conditions, using the method of Ju et al. [1], also shown in [2].

$$f_{curv} = f_{str,turb} \left[ 1 + 0.11Re^{0.23} \left( \frac{D_i}{D_c} \right)^{0.14} \right]$$

**Parameters**

_**Re** [float] Reynolds number with $D=Di$, [-]_

_**Di** [float] Inner diameter of the coil, [m]_

_**Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]_

_roughness [float, optional] Roughness of pipe wall [m]_

**Returns**

**fd** [float] Darcy friction factor for a curved pipe [-]

**Notes**

Claimed to be valid for all turbulent conditions with $Dc > 11.6$. At very low curvatures, converges on the straight pipe result.

**References**

[1], [2]
Examples

```python
>>> helical_turbulent_fd_Ju(1E4, 0.01, .2)
0.04945959480770937
```

`fluids.friction.helical_turbulent_fd_Srinivasan(Re, Di, Dc)`

Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under turbulent conditions, using the method of Srinivasan [1], as shown in [2] and [3].

\[
    f_d = \frac{0.336}{Re \left( \frac{Di}{Dc} \right)^{0.2}}
\]

Parameters

- **Re** [float] Reynolds number with \(D=Di\), [-]
- **Di** [float] Inner diameter of the coil, [m]
- **Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

Returns

- **fd** [float] Darcy friction factor for a curved pipe [-]

Notes

Valid for 0.01 < \(Di/Dc\) < 0.15, with no Reynolds number criteria given in [2] or [3].

[2] recommends this method, using the transition criteria of Srinivasan as well. [3] recommends using either this method or the Ito method. This method did not make it into the popular review articles on curved flow.

References

[1], [2], [3]

Examples

```python
>>> helical_turbulent_fd_Srinivasan(1E4, 0.01, .02)
0.0570745212117107
```

`fluids.friction.helical_turbulent_fd_Mandal_Nigam(Re, Di, Dc, roughness=0)`

Calculates Darcy friction factor for a fluid flowing inside a curved pipe such as a helical coil under turbulent conditions, using the method of Mandal and Nigam [1], also shown in [2].

\[
    f_{curv} = f_{str,turb} \left[ 1 + 0.03De^{0.27} \right]
\]

Parameters

- **Re** [float] Reynolds number with \(D=Di\), [-]
- **Di** [float] Inner diameter of the coil, [m]
- **Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]
- **roughness** [float, optional] Roughness of pipe wall [m]
Returns

\[ fd \quad [\text{float}] \quad \text{Darcy friction factor for a curved pipe [-]} \]

Notes

Claimed to be valid for all turbulent conditions with \( 2500 < D e < 15000 \). At very low curvatures, converges on the straight pipe result.

References

[1], [2]

Examples

```python
>>> helical_turbulent_fd_Mandal_Nigam(1E4, 0.01, .2)
0.03831658117115902
```

\texttt{fluids.friction.helical_transition_Re_Seth_Stahel} \((\text{Di}, \text{Dc})\)

Calculates the transition Reynolds number for flow inside a curved or helical coil between laminar and turbulent flow, using the method of [1].

\[
Re_{\text{crit}} = 1900 \left[ 1 + 8 \sqrt{\frac{D_i}{D_c}} \right]
\]

Parameters

- \texttt{Di} [\text{float}] Inner diameter of the coil, [m]
- \texttt{Dc} [\text{float}] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

Returns

- \texttt{Re_crit} [\text{float}] Transition Reynolds number between laminar and turbulent [-]

Notes

At very low curvatures, converges to \( Re = 1900 \).

References

[1]

Examples

```python
>>> helical_transition_Re_Seth_Stahel(1, 7.)
7645.0599897402535
```
fluids.friction.helical_transition_Re_Ito(Di, Dc)
Calculates the transition Reynolds number for flow inside a curved or helical coil between laminar and turbulent flow, using the method of [1], as shown in [2] and in [3].

\[ Re_{crit} = 20000 \left( \frac{D_i}{D_c} \right)^{0.32} \]

Parameters

- **Di** [float] Inner diameter of the coil, [m]
- **Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

Returns

- **Re_crit** [float] Transition Reynolds number between laminar and turbulent [-]

Notes

At very low curvatures, converges to Re = 0. Recommended for \(0.00116 < \frac{d_i}{D_c} < 0.067\).

References

[1], [2], [3]

Examples

```python
>>> helical_transition_Re_Ito(1, 7.)
10729.972844697186
```

fluids.friction.helical_transition_Re_Kubair_Kuloor(Di, Dc)
Calculates the transition Reynolds number for flow inside a curved or helical coil between laminar and turbulent flow, using the method of [1], as shown in [2].

\[ Re_{crit} = 12730 \left( \frac{D_i}{D_c} \right)^{0.2} \]

Parameters

- **Di** [float] Inner diameter of the coil, [m]
- **Dc** [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

Returns

- **Re_crit** [float] Transition Reynolds number between laminar and turbulent [-]

Notes

At very low curvatures, converges to Re = 0. Recommended for \(0.0005 < \frac{d_i}{D_c} < 0.103\).

References

[1], [2]
Examples

```python
>>> helical_transition_Re_Kubair_Kuloor(1., 7.)
8625.986927588123
```

`fluids.friction.helical_transition_Re_Kutateladze_Borishanskii(Di, Dc)`
Calculates the transition Reynolds number for flow inside a curved or helical coil between laminar and turbulent flow, using the method of [1], also shown in [2].

\[
Re_{crit} = 2300 + 1.05 \times 10^4 \left( \frac{D_i}{D_c} \right)^{0.3}
\]

Parameters

- `Di` [float] Inner diameter of the coil, [m]
- `Dc` [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

Returns

- `Re_crit` [float] Transition Reynolds number between laminar and turbulent [-]

Notes

At very low curvatures, converges to Re = 2300. Recommended for \(0.0417 < \frac{d_i}{D_c} < 0.1667\).

References

[1], [2]

Examples

```python
>>> helical_transition_Re_Kutateladze_Borishanskii(1, 7.)
7121.143774574058
```

`fluids.friction.helical_transition_Re_Schmidt(Di, Dc)`
Calculates the transition Reynolds number for flow inside a curved or helical coil between laminar and turbulent flow, using the method of [1], also shown in [2] and [3]. Correlation recommended in [3].

\[
Re_{crit} = 2300 \left[ 1 + 8.6 \left( \frac{D_i}{D_c} \right)^{0.45} \right]
\]

Parameters

- `Di` [float] Inner diameter of the coil, [m]
- `Dc` [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

Returns

- `Re_crit` [float] Transition Reynolds number between laminar and turbulent [-]
Notes

At very low curvatures, converges to \( \text{Re} = 2300 \). Recommended for \( \frac{d_i}{D_c} < 0.14 \).

References

[1], [2], [3]

Examples

```python
>>> helical_transition_Re_Schmidt(1, 7.)
10540.094061770815
```

```
fluids.friction.helical_transition_Re_Srinivasan(Di, Dc)
```

Calculates the transition Reynolds number for flow inside a curved or helical coil between laminar and turbulent flow, using the method of [1], also shown in [2] and [3]. Correlation recommended in [3].

\[
\text{Re}_{\text{crit}} = 2100 \left[ 1 + 12 \left( \frac{D_i}{D_c} \right)^{0.5} \right]
\]

Parameters

- \( \text{Di} \) [float] Inner diameter of the coil, [m]
- \( \text{Dc} \) [float] Diameter of the helix/coil measured from the center of the tube on one side to the center of the tube on the other side, [m]

Returns

- \( \text{Re}_{\text{crit}} \) [float] Transition Reynolds number between laminar and turbulent [-]

Notes

At very low curvatures, converges to \( \text{Re} = 2100 \). Recommended for \( 0.004 < \frac{d_i}{D_c} < 0.1 \).

References

[1], [2], [3]

Examples

```python
>>> helical_transition_Re_Srinivasan(1, 7.)
11624.704719832524
```

```
fluids.friction.LAMINAR_TRANSITIONPIPE = 2040.0
```

**Fluids Documentation, Release 0.1**

**fluids.friction.friction_plate_Martin_1999** *(Re, plate_enlargement_factor)*

Calculates Darcy friction factor for single-phase flow in a Chevron-style plate heat exchanger according to [1].

\[
\frac{1}{\sqrt{f_d}} = \frac{\cos \phi}{\sqrt{0.045 \tan \phi + 0.09 \sin \phi + f_0 / \cos(\phi)}} + \frac{1 - \cos \phi}{\sqrt{3.8f_1}}
\]

\[
f_0 = 16 / Re \text{ for } Re < 2000
\]

\[
f_0 = (1.56 \ln Re - 3)^{-2} \text{ for } Re \geq 2000
\]

\[
f_1 = \frac{149}{Re} + 0.9625 \text{ for } Re < 2000
\]

\[
f_1 = \frac{9.75}{Re^{0.289}} \text{ for } Re \geq 2000
\]

**Parameters**

- **Re** [float] Reynolds number with respect to the hydraulic diameter of the channels, [-]
- **plate_enlargement_factor** [float] The extra surface area multiplier as compared to a flat plate caused the corrugations, [-]

**Returns**

- **fd** [float] Darcy friction factor [-]

**Notes**

Based on experimental data from Re from 200 - 10000 and enhancement factors calculated with chevron angles of 0 to 80 degrees. See PlateExchanger for further clarification on the definitions.

The length the friction factor gets multiplied by is not the flow path length, but rather the straight path length from port to port as if there were no chevrons.

Note there is a discontinuity at Re = 2000 for the transition from laminar to turbulent flow, although the literature suggests the transition is actually smooth.

This was first developed in [2] and only minor modifications by the original author were made before its republication in [1]. This formula is also suggested in [3]

**References**

[1], [2], [3]

**Examples**

```python
>>> friction_plate_Martin_1999(Re=20000, plate_enlargement_factor=1.15)
2.284018089834134
```

**fluids.friction.friction_plate_Martin_VDI** *(Re, plate_enlargement_factor)*

Calculates Darcy friction factor for single-phase flow in a Chevron-style plate heat exchanger according to [1].

\[
\frac{1}{\sqrt{f_d}} = \frac{\cos \phi}{\sqrt{0.28 \tan \phi + 0.36 \sin \phi + f_0 / \cos(\phi)}} + \frac{1 - \cos \phi}{\sqrt{3.8f_1}}
\]

\[
f_0 = 64 / Re \text{ for } Re < 2000
\]

\[
f_0 = (1.56 \ln Re - 3)^{-2} \text{ for } Re \geq 2000
\]
\[ f_1 = \frac{597}{Re} + 3.85 \text{ for } Re < 2000 \]
\[ f_1 = \frac{39}{Re^{0.289}} \text{ for } Re \geq 2000 \]

**Parameters**

- **Re** [float] Reynolds number with respect to the hydraulic diameter of the channels, [-]
- **plate_enlargement_factor** [float] The extra surface area multiplier as compared to a flat plate caused the corrugations, [-]

**Returns**

- **fd** [float] Darcy friction factor [-]

**See also:**

`friction_plate_Martin_1999`

**Notes**

Based on experimental data from Re from 200 - 10000 and enhancement factors calculated with chevron angles of 0 to 80 degrees. See `PlateExchanger` for further clarification on the definitions.

The length the friction factor gets multiplied by is not the flow path length, but rather the straight path length from port to port as if there were no chevrons.

Note there is a discontinuity at Re = 2000 for the transition from laminar to turbulent flow, although the literature suggests the transition is actually smooth.

This is a revision of the Martin’s earlier model, adjusted to predict higher friction factors.

There are three parameters in this model, a, b and c; it is possible to adjust them to better fit a known exchanger’s pressure drop.

**References**

[1]

**Examples**

```python
>>> friction_plate_Martin_VDI(Re=20000, plate_enlargement_factor=1.15)
2.702534119024076
```

`fluids.friction.friction_plate_Kumar(Re, chevron_angle)`

Calculates Darcy friction factor for single-phase flow in a well-designed Chevron-style plate heat exchanger according to [1]. The data is believed to have been developed by APV International Limited, since acquired by SPX Corporation. This uses a curve fit of that data published in [2].

\[ f_f = \frac{C_2}{Re^p} \]

C2 and p are coefficients looked up in a table, with varying ranges of Re validity and chevron angle validity. See the source for their exact values.

**Parameters**

- **Re** [float] Reynolds number with respect to the hydraulic diameter of the channels, [-]
\texttt{chevron\_angle} [float] Angle of the plate corrugations with respect to the vertical axis (the direction of flow if the plates were straight), between 0 and 90. Many plate exchangers use two alternating patterns; use their average angle for that situation [degrees]

Returns

\texttt{fd} [float] Darcy friction factor [-]

Notes

Data on graph from Re=0.1 to Re=10000, with chevron angles 30 to 65 degrees. See \textit{PlateExchanger} for further clarification on the definitions.

It is believed the constants used in this correlation were curve-fit to the actual graph in [1] by the author of [2] as there is no

The length the friction factor gets multiplied by is not the flow path length, but rather the straight path length from port to port as if there were no chevrons.

As the coefficients change, there are numerous small discontinuities, although the data on the graphs is continuous with sharp transitions of the slope.

The author of [1] states clearly this correlation is “applicable only to well designed Chevron PHEs”.

References

[1],[2]

Examples

\begin{verbatim}
>>> friction_plate_Kumar(Re=2000, chevron_angle=30)
2.9760669055634517
\end{verbatim}

\texttt{fluids.friction\_friction\_plate\_Muley\_Manglik(Re, chevron\_angle, plate\_enlargement\_factor)}

Calculates Darcy friction factor for single-phase flow in a Chevron-style plate heat exchanger according to [1], also shown and recommended in [2].

\[ f_d = [2.917 - 0.1277\beta + 2.016 \times 10^{-3}\beta^2] \times [20.78 - 19.02\phi + 18.93\phi^2 - 5.341\phi^3] \times Re^{-[0.2+0.0577 \sin(\pi\beta/45)+2.1]} \]

Parameters

\begin{itemize}
  \item \texttt{Re} [float] Reynolds number with respect to the hydraulic diameter of the channels, [-]
  \item \texttt{chevron\_angle} [float] Angle of the plate corrugations with respect to the vertical axis (the direction of flow if the plates were straight), between 0 and 90. Many plate exchangers use two alternating patterns; use their average angle for that situation [degrees]
  \item \texttt{plate\_enlargement\_factor} [float] The extra surface area multiplier as compared to a flat plate caused the corrugations, [-]
\end{itemize}

Returns

\begin{itemize}
  \item \texttt{fd} [float] Darcy friction factor [-]
\end{itemize}
Notes

Based on experimental data of plate enacement factors up to 1.5, and valid for Re > 1000 and chevron angles from 30 to 60 degrees with sinusoidal shape. See PlateExchanger for further clarification on the definitions.

The length the friction factor gets multiplied by is not the flow path length, but rather the straight path length from port to port as if there were no chevrons.

This is a continuous model with no discontinuities.

References

[1], [2]

Examples

```python
>>> friction_plate_Muley_Manglik(Re=2000, chevron_angle=45, plate_enlargement_factor=1.2)
1.0880870804075413
```

2.10 Tank and helical coil sizing (fluids.geometry)

class fluids.geometry.TANK (D=None, L=None, horizontal=True, sideA=None, sideB=None, sideA_a=None, sideB_a=None, sideA_f=None, sideA_k=None, sideB_f=None, sideB_k=None, sideA_a_ratio=None, sideB_a_ratio=None, L_over_D=None, V=None)

Bases: object

Class representing tank volumes and levels. All parameters are also attributes.

Parameters

- **D** [float] Diameter of the cylindrical section of the tank, [m]
- **L** [float] Length of the main cylindrical section of the tank, [m]
- **horizontal** [bool, optional] Whether or not the tank is a horizontal or vertical tank
- **sideA** [string, optional] The left (or bottom for vertical) head of the tank’s type; one of [None, ‘conical’, ‘ellipsoidal’, ‘torispherical’, ‘guppy’, ‘spherical’].
- **sideB** [string, optional] The right (or top for vertical) head of the tank’s type; one of [None, ‘conical’, ‘ellipsoidal’, ‘torispherical’, ‘guppy’, ‘spherical’].
- **sideA_a** [float, optional] The distance the head as specified by sideA extends down or to the left from the main cylindrical section, [m]
- **sideB_a** [float, optional] The distance the head as specified by sideB extends up or to the right from the main cylindrical section, [m]
- **sideA_f** [float, optional] Dimensionless dish-radius parameter for side A; also commonly given as the product of $f$ and $D$ ($fD$), which is called dish radius and has units of length, [-]
- **sideA_k** [float, optional] Dimensionless knuckle-radius parameter for side A; also commonly given as the product of $k$ and $D$ ($kD$), which is called the knuckle radius and has units of length, [-]
sideB_f [float, optional] Dimensionless dish-radius parameter for side B; also commonly given as the product of f and D (fD), which is called dish radius and has units of length, [-]

sideB_k [float, optional] Dimensionless knuckle-radius parameter for side B; also commonly given as the product of k and D (kD), which is called the knuckle radius and has units of length, [-]

L_over_D [float, optional] Ratio of length over diameter, used only when D and L are both unspecified but V is, [-]

V [float, optional] Volume of the tank; solved for if specified, using sideA_a_ratio/sideB_a_ratio, sideA, sideB, horizontal, and one of L_over_D, L, or D, [m³]

Notes

For torpshlerical tank heads, the following f and k parameters are used in standards. The default is ASME F&D.

<table>
<thead>
<tr>
<th></th>
<th>f</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:1 semi-elliptical</td>
<td>0.9</td>
<td>0.17</td>
</tr>
<tr>
<td>ASME F&amp;D</td>
<td>1</td>
<td>0.06</td>
</tr>
<tr>
<td>ASME 80/6</td>
<td>0.8</td>
<td>0.06</td>
</tr>
<tr>
<td>ASME 80/10 F&amp;D</td>
<td>0.8</td>
<td>0.1</td>
</tr>
<tr>
<td>DIN 28011</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>DIN 28013</td>
<td>0.8</td>
<td>0.154</td>
</tr>
</tbody>
</table>

Examples

Total volume of a tank:

```python
>>> TANK(D=1.2, L=4, horizontal=False).V_total
4.523893421169302
```

Volume of a tank at a given height:

```python
>>> TANK(D=1.2, L=4, horizontal=False).V_from_h(.5)
0.5654866776461628
```

Height of liquid for a given volume:

```python
>>> TANK(D=1.2, L=4, horizontal=False).h_from_V(.5)
0.44209706414415384
```

Surface area of a tank with a conical head:

```python
>>> T1 = TANK(V=10, L_over_D=0.7, sideB='conical', sideB_a=0.5)
(24.94775907657148, 5.118555935958284, 5.497246519930003, 14.331956620683192)
```

Solving for tank volumes, first horizontal, then vertical:

```python
>>> TANK(D=10., horizontal=True, sideA='conical', sideB='conical', V=500).L
4.699531057009147
```
Attributes

- **table** [bool] Whether or not a table of heights-volumes has been generated
- **h_max** [float] Height of the tank, [m]
- **V_total** [float] Total volume of the tank as calculated [m^3]
- **heights** [ndarray] Array of heights between 0 and h_max, [m]
- **volumes** [ndarray] Array of volumes calculated from the heights, [m^3]
- **A** [float] Total surface area of the tank, [m^2]
- **A_sideA** [float] Surface area of sideA, [m^2]
- **A_sideB** [float] Surface area of sideB, [m^2]
- **A_lateral** [float] Surface area of the lateral side, [m^2]
- **c_forward** [ndarray] Coefficients for the Chebyshev approximations in calculating V from h, [-]
- **c_backward** [ndarray] Coefficients for the Chebyshev approximations in calculating h from V, [-]

Methods

- **V_from_h**(self, h[, method]) Method to calculate the volume of liquid in a fully defined tank given a specified height h.
- **add_thickness**(self, thickness[,...]) Method to create a new tank instance with the same parameters as itself, except with an added thickness to it.
- **h_from_V**(self, V[, method]) Method to calculate the height of liquid in a fully defined tank given a specified volume of liquid in it V.
- **set_chebyshev_approximators**(self[, ...]) Method to derive and set coefficients for chebyshev polynomial function approximation of the height-volume and volume-height relationship.

Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_misc(self)</code></td>
<td>Set more parameters, after the tank is better defined than in the <code>__init__</code> function.</td>
</tr>
<tr>
<td><code>set_table(self[, n, dx])</code></td>
<td>Method to set an interpolation table of liquids levels versus volumes in the tank, for a fully defined tank.</td>
</tr>
<tr>
<td><code>solve_tank_for_V(self)</code></td>
<td>Method which is called to solve for tank geometry when a certain volume is specified.</td>
</tr>
</tbody>
</table>

V\_from\_h (self, h, method='full')
Method to calculate the volume of liquid in a fully defined tank given a specified height h. h must be under the maximum height. If the method is ‘chebyshev’, and the coefficients have not yet been calculated, they are created by calling `set_chebyshev_approximators`.

Parameters
- h [float] Height specified, [m]
- method [str] One of ‘full’ (calculated rigorously) or ‘chebyshev’

Returns
- V [float] Volume of liquid in the tank up to the specified height, [m^3]

add\_thickness (self, thickness, sideA\_thickness=None, sideB\_thickness=None)
Method to create a new tank instance with the same parameters as itself, except with an added thickness to it. This is useful to obtain ex. the inside of a tank and the outside; their different in volumes is the volume of the shell, and could be used to determine weight.

Parameters
- thickness [float] Thickness to add to the tank diameter, [m]
- sideA\_thickness [float, optional] The thickness to add to the sideA head; if not specified, it will be `thickness`, [m]
- sideB\_thickness [float, optional] The thickness to add to the sideB head; if not specified, it will be `thickness`, [m]

Returns
- TANK [TANK] Tank object, [-]

Notes
Be careful not to specify a negative thickness larger than the heads’ lengths, or the head will become concave! The same applies to adding a thickness to convex heads - they can become convex.

chebyshev = False

h\_from\_V (self, V, method='spline')
Method to calculate the height of liquid in a fully defined tank given a specified volume of liquid in it V. V must be under the maximum volume. If the method is ‘spline’, and the interpolation table is not yet defined, creates it by calling the method `set_table`. If the method is ‘chebyshev’, and the coefficients have not yet been calculated, they are created by calling `set_chebyshev_approximators`.

Parameters
- V [float] Volume of liquid in the tank up to the desired height, [m^3]
- method [str] One of ‘spline’, ‘chebyshev’, or ‘brenth’

Returns
**set_chebyshev_approximators** *(self, deg_forward=50, deg_backwards=200)*

Method to derive and set coefficients for chebyshev polynomial function approximation of the height-volume and volume-height relationship.

A single set of chebyshev coefficients is used for the entire height-volume and volume-height relationships respectively.

The forward relationship, \( V_{\text{from} \ h} \), requires far fewer coefficients in its fit than the reverse to obtain the same relative accuracy.

Optionally, \( \text{deg\_forward} \) or \( \text{deg\_backwards} \) can be set to None to try to automatically fit the series to machine precision.

**Parameters**

- **deg\_forward** [int, optional] The degree of the chebyshev polynomial to be created for the \( V_{\text{from} \ h} \) curve, [-]
- **deg\_backwards** [int, optional] The degree of the chebyshev polynomial to be created for the \( h_{\text{from} \ V} \) curve, [-]

**set_misc** *(self)*

Set more parameters, after the tank is better defined than in the __init__ function.

**Notes**

Two of D, L, and L\_over\_D must be known when this function runs. The other one is set from the other two first thing in this function. \( a\_\text{ratio} \) parameters are used to calculate a values for the heads here, if applicable. Radius is calculated here. Maximum tank height is calculated here. \( V\_\text{total} \) is calculated here.

**set_table** *(self, n=100, dx=None)*

Method to set an interpolation table of liquids levels versus volumes in the tank, for a fully defined tank. Normally run by the \( h_{\text{from} \ V} \) method, this may be run prior to its use with a custom specification. Either the number of points on the table, or the vertical distance between steps may be specified.

**Parameters**

- **n** [float, optional] Number of points in the interpolation table, [-]
- **dx** [float, optional] Vertical distance between steps in the interpolation table, [m]

**solve_tank_for_V** *(self)*

Method which is called to solve for tank geometry when a certain volume is specified. Will be called by the __init__ method if \( V \) is set.

**Notes**

Raises an error if L and either of \( \text{sideA\_a} \) or \( \text{sideB\_a} \) are specified; these can only be set once D is known.

 Raises an error if more than one of D, L, or L\_over\_D are specified. Raises an error if the head ratios are not provided.

Calculates initial guesses assuming no heads are present, and then uses fsolve to determine the correct dimensions for the tank.

Tested, but bugs and limitations are expected here.

**table = False**
class Fluids Documentation, Release 0.1

Fluids Documentation, Release 0.1

class fluids.geometry.HelicalCoil(Dt, Do=None, pitch=None, H=None, N=None, H_total=None, Do_total=None, Di=None)

Bases: object

Class representing a helical coiled tube, as are found in many heated tanks and some small nuclear reactors. All parameters are also attributes.

One set of the following parameters is required; inner tube diameter is optional.

- Tube outer diameter, coil outer diameter, pitch, number of coil turns
- Tube outer diameter, coil outer diameter, pitch, height
- Tube outer diameter, coil outer diameter, number of coil turns, height

Parameters

- **Dt** [float] Outer diameter of the tube wound to make up the helical spiral, [m]
- **Do** [float, optional] Diameter of the spiral as measured from the center of the coil on one side to the center of the coil on the other side, [m]
- **Do_total** [float, optional] Diameter of the spiral as measured from one edge of the tube to the other edge; equal to Do + Dt; either Do or Do_total may be specified and the other will be calculated [m]
- **pitch** [float, optional] Height change from one coil to the next as measured from the middles of the tube, [m]
- **H** [float, optional] Height of the spiral, as measured from the middle of the bottom of the tube to the middle of the top of the tube, [m]
- **H_total** [float, optional] Height of the spiral as measured from one edge of the tube to the other edge; equal to H_total + Dt; either may be specified and the other will be calculated [m]
- **N** [float, optional] Number of coil turns; may be specified along with pitch instead of specifying H or H_total, [-]
- **Di** [float, optional] Inner diameter of the tube; if specified, inside and annulus properties will be calculated, [m]

Notes

Do must be larger than Dt.

References

[1]

Examples

```python
>>> C1 = HelicalCoil(Do=30, H=20, pitch=5, Dt=2)
>>> C1.N, C1.tube_length, C1.surface_area
(4.0, 377.5212621504738, 2372.0360474917497)
```

Same coil, with the inputs one would physically measure from the coil, and a specified inlet diameter:
>>> C1 = HelicalCoil(Do_total=32, H_total=22, pitch=5, Dt=2, Di=1.8)

>>> C1.N, C1.tube_length, C1.surface_area
(4.0, 377.5212621504738, 2372.0360474917497)

>>> C1.inner_surface_area, C1.inlet_area, C1.inner_volume, C1.total_volume, C1.annulus_volume
(2134.832442742575, 2.5446900494077327, 960.6745992341587, 1186.0180237458749, 225.3434245117162)

Attributes

tube_circumference [float] Circumference of the tube as measured though its center, not inner or outer edges; \( C = \pi D_o \), [m]

tube_length [float] Length of tube used to make the helical coil; \( L = \sqrt{(\pi D_o \cdot N)^2 + H^2} \), [m]
surface_area [float] Surface area of the outer surface of the helical coil; \( A_t = \pi D_t L \), [m²]
inner_surface_area [float] Surface area of the inner surface of the helical coil; calculated if \( D_i \) is supplied; \( A_{inside} = \pi D_i L \), [m²]
inlet_area [float] Area of the inlet to the helical coil; calculated if \( D_i \) is supplied; \( A_{inlet} = \frac{\pi}{4} D_i^2 \), [m²]
inner_volume [float] Volume of the tube as would be filled by a fluid, useful for weight calculations; calculated if \( D_i \) is supplied; \( V_{inside} = A_i L \), [m³]
annulus_area [float] Area of the annulus (wall of the pipe); calculated if \( D_i \) is supplied; \( A_a = \frac{\pi}{4}(D_t^2 - D_i^2) \), [m²]
annulus_volume [float] Volume of the annulus (wall of the pipe); calculated if \( D_i \) is supplied, useful for weight calculations; \( V_a = A_a L \), [m³]
total_volume [float] Total volume occupied by the pipe and the fluid inside it; \( V = D_t L \), [m³]
helix_angle [float] Angle between the pitch and coil diameter; used in some calculations; \( \alpha = \arctan \left( \frac{D_t}{\pi D_o} \right) \), [radians]
curvature [float] Coil curvature, useful in some calculations; \( \delta = \frac{D_t}{D_o \left[1+4\pi^2 \tan^2(\alpha)\right]} \), [-]

```python
class fluids.geometry.PlateExchanger(amplitude, wavelength, chevron_angle=45, width=None, length=None, thickness=None, d_port=None, plates=None)
```

Bases: object

Class representing a plate heat exchanger with sinusoidal ridges. All parameters are also attributes.

Parameters

amplitude [float] Half the height of the wave of the ridges, [m]

wavelength [float] Distance between the bottoms of two of the ridges (sometimes called pitch), [m]

chevron_angle [float or tuple(2), optional] Angle of the plate corrugations with respect to the vertical axis (the direction of flow if the plates were straight), between 0 and 90. Many plate exchangers use two alternating patterns; use a tuple of the two angles for that situation [degrees]

width [float, optional] Width of the plates in the heat exchanger, between the gaskets, [m]
length  [float, optional] Length of the heat exchanger as measured from one port to the other, excluding the diameter of the ports themselves (little useful heat transfer happens there), [m]

thickness  [float, optional] Thickness of the metal making up the plates, [m]

d_port  [float, optional] The diameter of the ports in the plates, [m]

plates  [int, optional] The number of plates in the heat exchanger, including the two not used for heat transfer at the beginning and end [-]

Notes

Only wavelength and amplitude are required as inputs to this function.

References

[Ra5f43f3c7b6-1]

Examples

```python
>>> PlateExchanger(amplitude=5E-4, wavelength=3.7E-3, length=1.2, width=.3, ...
d_port=.05, plates=51)
<Plate heat exchanger, amplitude=0.0005 m, wavelength=0.0037 m, chevron_angles=45/45 degrees, area enhancement factor=1.16119, width=0.3 m, length=1.2 m, port直径=0.05 m, heat transfer area=20.4833 m^2, 51 plates>
```

Attributes

- **chevron_angles**  [tuple(2)] The two specified angles (repeated value if only one specified), [degrees]
- **chevron_angle**  [float] The averaged angle of the chevrons, [degrees]
- **inclination_angle**  [float] 90 - chevron_angle, used in many publications instead of chevron_angle, [degrees]
- **plate_corrugation_aspect_ratio**  [float] The aspect ratio of the corrugations $\gamma = \frac{4a}{\lambda}$. [-]
- **plate_enlargement_factor**  [float] The extra surface area multiplier as compared to a flat plate caused the corrugations, [-]
- **D_eq**  [float] Equivalent diameter of the channels, $D_{eq} = 4a$ [m]
- **D_hydraulic**  [float] Hydraulic diameter of the channels, $D_{hyd} = \frac{4a}{\phi}$ [m]
- **length_port**  [float] Port center to port center along the direction of flow, [m]
- **A_plate_surface**  [float] The surface area of one plate in the heat exchanger, including the extra due to corrugations (excluding the bit between the ports), $A_p = L \cdot W \cdot \phi$ [m^2]
- **A_heat_transfer**  [float] The total surface area available for heat transfer in the exchanger, the multiple of A_plate_surface by the number of plates after removing the two on the edges, [m^2]
- **A_channel_flow**  [float] The area for the fluid to flow in one channel, $W \cdot b$ [m^2]
- **channels**  [int] The number of plates minus one, [-]
channels_per_fluid  [int] Half the number of total channels, [-]


Methods

plate_enlargement_factor_analytical() Calculates the enhancement factor of the sinusoidal waves of the plate heat exchanger. This is the multiplier for the flat plate area to obtain the actual area available for heat transfer. Obtained from the following integral:

$$
\phi = \frac{\text{Effective area}}{\text{Projected area}} = \int_0^\lambda \sqrt{1 + \left(\frac{\gamma \pi}{2}\right)^2 \cos^2 \left(\frac{2\pi}{\lambda} x\right)} dx
$$

$$
\gamma = \frac{4a}{\lambda}
$$

The solution to the integral is:

..math:: \phi = \frac{2E\left(\frac{-4a^2\pi^2}{\lambda^2}\right)}{\pi}

where $E$ is the complete elliptic integral of the second kind, calculated with SciPy.

Parameters

amplitude  [float]

Half the height of the wave of the ridges, [m]

wavelength  [float] Distance between the bottoms of two of the ridges (sometimes called pitch), [m]

Returns

plate_enlargement_factor  [float] The extra surface area multiplier as compared to a flat plate caused the corrugations, [-]

Notes

This is the exact analytical integral, obtained via Mathematica, Maple, and quite a bit of trial and error. It is confirmed via numerical integration. The expression normally given is an approximation as follows:

$$
\phi = \frac{1}{6} \left(1 + \sqrt{1 + A^2} + 4 \sqrt{1 + A^2/2}\right)
$$

$$
A = \frac{2\pi a}{\lambda}
$$

Most plate heat exchangers approximate a sinusoidal geometry only.
Examples

```python
>>> PlateExchanger.plate_enlargement_factor_analytical(amplitude=5E-4, wavelength=3.7E-3)
1.1611862034509677
```

**plate_exchanger_identifier**

Method to create an identifying string in format ‘L’ + wavelength + ‘A’ + amplitude + ‘B’ + chevron angle-chevron angle. Wavelength and amplitude are specified in units of mm and rounded to two decimal places.

```python
class fluids.geometry.RectangularFinExchanger (fin_height, fin_thickness, fin_spacing, length=None, width=None, layers=None, plate_thickness=None, flow='crossflow')
```

Bases: object

Class representing a plate-fin heat exchanger with straight rectangular fins. All parameters are also attributes.

**Parameters**

- **fin_height** [float] The total distance between the two metal plates sandwiching the fins and holding them together (abbreviated $h$), [m]
- **fin_thickness** [float] The thickness of the material the fins were formed from (abbreviated $t$), [m]
- **fin_spacing** [float] The unit cell spacing from one fin to the next; the space between the sides of two fins plus one thickness (abbreviated $s$), [m]
- **length** [float, optional] The total length of the flow passage of the plate-fin exchanger (abbreviated $L$), [m]
- **width** [float, optional] The total width of the space the fins are in; this is also $N_{\text{fins}} \times s$ (abbreviated $W$), [m]
- **layers** [int, optional] The number of layers in the plate-fin exchanger; note these HX almost always single-pass only, [-]
- **plate_thickness** [float, optional] The thickness of the metal separator between layers, [m]
- **flow** [str, optional] One of ‘counterflow’, ‘crossflow’, or ‘parallelflow’

**Notes**

The only required parameters are the fin geometry itself; *fin_height, fin_thickness*, and *fin_spacing*.

**References**

[R6a2fe8cbf2f6-1], [R6a2fe8cbf2f6-2]

**Examples**

```python
>>> PFE = RectangularFinExchanger(0.03, 0.001, 0.012)
>>> PFE.Dh
0.01595
```

**Attributes**
channel_height [float] The height of the channel the fluid flows in channel height = fin height − fin thickness, [m]

channel_width [float] The width of the channel the fluid flows in channel width = fin spacing − fin thickness, [m]

fin_count [int] The number of fins per unit length of the layer, fin count = \frac{1}{\text{fin spacing}}, [1/m]

blockage_ratio [float] The fraction of the layer which is blocked to flow by the fins, blockage ratio = \frac{s·h−st−t(h−t)}{s·h}, [m]

A_channel [float] Flow area of a single channel in a single layer, channel area = (s − t)(h − t), [m]

P_channel [float] Wetted perimeter of a single channel in a single layer, channel perimeter = 2(s − t) + 2(h − t), [m]

Dh [float] Hydraulic diameter of a single channel in a single layer, D_{\text{hydraulic}} = \frac{4A_{\text{channel}}}{P_{\text{channel}}}, [m]

layer_thickness [float] The thickness of a single layer - the sum of a fin height and a plate thickness, [m]

layer_fin_count [int] The number of fins in a layer; rounded to the nearest whole fin, [-]

A_HX_layer [float] The surface area including fins for heat transfer in one layer of the HX, [m²]

A_HX [float] The total surface area of the heat exchanger with all layers combined, [m²]

height [float] The height of all the layers of the heat exchanger combined, plus one extra plate thickness, [m]

volume [float] The product of the height, width, and length of the HX, [m³]

A_specific_HX [float] The specific surface area of the heat exchanger - square meters per meter cubed, [m³]

Methods

```python
set_overall_geometry(self)
```

class fluids.geometry.RectangularOffsetStripFinExchanger:

Bases: fluids.geometry.RectangularFinExchanger

Methods

```python
set_overall_geometry()
```
class fluids.geometry.HyperbolicCoolingTower(H_inlet, D_outlet, H_outlet, D_inlet=None, 
D_base=None, D_throat=None, 
H_throat=None, H_support=None, 
D_support=None, n_support=None, 
inlet_rounding=None)

Bases: object

Class representing the geometry of a hyperbolic cooling tower, as used in many industries especially the power industry. All parameters are also attributes.

H_inlet, D_outlet, and H_outlet are always required. Additionally, one set of the following parameters is required: H_support, D_support, n_support, and inlet_rounding are all optional as well.

- Inlet diameter
- Inlet diameter and throat diameter
- Inlet diameter and throat height
- Inlet diameter, throat diameter, and throat height
- Base diameter, throat diameter, and throat height

If the inlet diameter is provided but the throat diameter and/or the throat height are missing, two heuristics are used to estimate them (to avoid these heuristics simply specify the values):

- Assume the throat elevation is 2/3 the elevation of the tower.
- Assume the throat diameter is 63% the diameter of the inlet.

Parameters

- H_inlet [float] Height of the inlet zone of the cooling tower (also called rain zone), [m]
- D_outlet [float] The inside diameter of the cooling tower outlet (top of the tower; the elevation the concrete section ends), [m]
- H_outlet [float] The height of the cooling tower outlet (top of the tower; the elevation the concrete section ends), [m]
- D_inlet [float, optional] The inside diameter of the cooling tower inlet at the elevation the concrete section begins, [m]
- D_base [float, optional] The diameter of the cooling tower at the very base of the tower (the bottom of the inlet zone, at the elevation of the ground), [m]
- D_throat [float, optional] The diameter of the cooling tower at its minimum section, called its throat; where the two hyperbolas meet, [m]
- h_throat [float, optional] The elevation of the cooling tower’s throat (its minimum section; where the two hyperbolas meet), [m]
- inlet_rounding [float, optional] Radius of an optional rounded protrusion from the lip of the cooling tower shell base, which curves upwards from the lip (used to reduce the dead zone area rather than having a flat lip), [m]
- H_support [float, optional] The height of each support column, [m]
- D_support [float, optional] The diameter of each support column, [m]
- n_support [int, optional] The number of support columns of the cooling tower, [m]
Notes

Note there are two hyperbolas in a hyperbolic cooling tower - one under the throat and one above it; they are not necessarily the same.

A hyperbolic cooling tower is not the absolute optimal design, but is is close. The optimality is determined by the amount of material required to build it while maintaining its rigidity. For thermal design purposes, a hyperbolic model covers any minor variation quite well.

References

[R9da9598dd59d-1], [R9da9598dd59d-2]

Examples

```python
>>> ct = HyperbolicCoolingTower(D_outlet=89.0, H_outlet=200, D_inlet=136.18, H_inlet=14.5)
>>> ct
<Hyperbolic cooling tower, inlet diameter=136.18 m, outlet diameter=89 m, inlet height=14.5 m, outlet height=200 m, throat diameter=85.7934 m, throat height=133.333 m, base diameter=146.427 m>
>>> ct.diameter(5)
142.84514486126062
```

Attributes

- **b_lower** [float] The \( b \) parameter in the hyperbolic equation for the lower section of the cooling tower, [m]
- **b_upper** [float] The \( b \) parameter in the hyperbolic equation for the upper section of the cooling tower, [m]

Methods

- **diameter** *(self, H)* Calculates cooling tower diameter at a specified height, using the formulas for either hyperbola, depending on the height specified.

```
diameter = D_{throat} \sqrt{H^2 + b^2} / b
```

The value of \( H \) and \( b \) used in the above equation is as follows:

- \( H_{throat} \) - \( H \) and \( b_{lower} \) if under the throat
- \( H - H_{throat} \) and \( b_{upper} \), if above the throat
Parameters

H [float] Height at which to calculate the cooling tower diameter, [m]

Returns

D [float] Diameter of the cooling tower at the specified height, [m]

plot (self, pts=100)

class fluids.geometry.AirCooledExchanger(tube_rows, tube_passes, tubes_per_row, tube_length, tube_diameter, fin_thickness, angle=None, pitch=None, pitch_parallel=None, pitch_normal=None, pitch_ratio=None, fin_diameter=None, fin_height=None, fin_density=None, fin_interval=None, parallel_bays=1, bundles_per_bay=1, fans_per_bay=1, corbels=False, tube_thickness=None, fan_diameter=None)

Bases: object

Class representing the geometry of an air cooled heat exchanger with one or more tube bays, fans, or bundles. All parameters are also attributes.

The minimum information required to describe an air cooler is as follows:

- tube_rows
- tube_passes
- tubes_per_row
- tube_length
- tube_diameter
- fin_thickness

Two of angle, pitch, pitch_parallel, and pitch_normal (pitch_ratio may take the place of pitch)

Either fin_diameter or fin_height. Either fin_density or fin_interval.

Parameters

- tube_rows [int] Number of tube rows per bundle, [-]
- tube_passes [int] Number of tube passes (times the fluid travels across one tube length), [-]
- tubes_per_row [float] Number of tubes per row per bundle, [-]
- tube_length [float] Total length of the tube bundle tubes, [m]
- tube_diameter [float] Diameter of the bare tube, [m]
- fin_thickness [float] Diameter of the bare tube, [m]
- angle [float, optional] Angle of the tube layout, [degrees]
- pitch [float, optional] Shortest distance between tube centers; defined in relation to the flow direction only, [m]
- pitch_parallel [float, optional] Distance between tube center along a line parallel to the flow; has been called longitudinal pitch, pp, s2, SL, and p2, [m]
- pitch_normal [float, optional] Distance between tube centers in a line 90° to the line of flow; has been called the transverse pitch, pn, s1, ST, and p1, [m]
**pitch_ratio** [float, optional] Ratio of the pitch to bare tube diameter, [-]

**fin_diameter** [float, optional] Outer diameter of each tube after including the fin on both sides, [m]

**fin_height** [float, optional] Height above bare tube of the tube fins, [m]

**fin_density** [float, optional] Number of fins per meter of tube, [1/m]

**fin_interval** [float, optional] Space between each fin, including the thickness of one fin at its base, [m]

**parallel_bays** [int, optional] Number of bays in the unit, [-]

**bundles_per_bay** [int, optional] Number of tube bundles per bay, [-]

**fans_per_bay** [int, optional] Number of fans per bay, [-]

**corbels** [bool, optional] Whether or not the air cooler has corbels, which increase the air velocity by adding half a tube to the sides for the case of non-rectangular tube layouts, [-]

**tube_thickness** [float, optional] Thickness of the bare metal tubes, [m]

**fan_diameter** [float, optional] Diameter of air cooler fan, [m]

**References**

[1]

**Examples**

```python
>>> AC = AirCooledExchanger(tube_rows=4, tube_passes=4, tubes_per_row=56, tube_length=10.9728,
    ... tube_diameter=1*inch, fin_thickness=0.013*inch, fin_density=10/inch,
    ... angle=30, pitch=2.5*inch, fin_height=0.625*inch, tube_thickness=0.00338,
    ... bundles_per_bay=2, parallel_bays=3, corbels=True)
```

**Attributes**

- **bare_length** [float] Length of bare tube between two fins bare length = fin interval − tfin, [m]
- **tubes_per_bundle** [float] Total number of tubes per bundle, \( N_{tubes/bundle} = N_{tubes/row} \cdot N_{rows}, [-] \)
- **tubes_per_bay** [float] Total number of tubes per bay, \( N_{tubes/bay} = N_{tubes/bundle} \cdot N_{bundles/bay}, [-] \)
- **tubes** [float] Total number of tubes in all bundles in all bays combined, \( N_{tubes} = N_{tubes/bay} \cdot N_{bays}, [-] \)
- **pitch_diagonal** [float] Distance between tube centers in a diagonal line between one normal tube and one parallel tube; \( s_D = \left( s_L^2 + \left( \frac{\pi}{2} \right)^2 \right)^{0.5}, [m] \)
- **A_bare_tube_per_tube** [float] Area of the bare tube including the portion hidden by the fin per tube, \( A_{bare, total/tube} = \pi D_{tube} L_{tube}, [m^2] \)
- **A_bare_tube_per_row** [float] Area of the bare tube including the portion hidden by the fin per tube row, \( A_{bare, total/row} = \pi D_{tube} L_{tube} N_{tubes/row}, [m^2] \)
- **A_bare_tube_per_bundle** [float] Area of the bare tube including the portion hidden by the fin per bundle, \( A_{bare, total/bundle} = \pi D_{tube} L_{tube} N_{tubes/bundle}, [m^2] \)
**A_bare_tube_per_bay** [float] Area of the bare tube including the portion hidden by the fin per bay

\[ A_{bare,\, total/bay} = \pi D_{tube} L_{tube} N_{tubes/bay}, \quad [m^2] \]

**A_bare_tube** [float] Area of the bare tube including the portion hidden by the fin per in all bundles and bays combined

\[ A_{bare,\, total} = \pi D_{tube} L_{tube} N_{tubes}, \quad [m^2] \]

**A_tube_showing_per_tube** [float] Area of the bare tube which is exposed per tube

\[ A_{bare,\, showing/tube} = \pi D_{tube} L_{tube} \left( 1 - \frac{l_{fin}}{\text{fin interval}} \right), \quad [m^2] \]

**A_tube_showing_per_row** [float] Area of the bare tube which is exposed per tube row, [m^2]

**A_tube_showing_per_bundle** [float] Area of the bare tube which is exposed per bundle, [m^2]

**A_tube_showing_per_bay** [float] Area of the bare tube which is exposed per bay, [m^2]

**A_tube_showing** [float] Area of the bare tube which is exposed in all bundles and bays combined, [m^2]

**A_per_fin** [float] Surface area per fin

\[ A_{fin} = 2\pi \left( \frac{D_{fin}^2}{4} - D_{tube}^2 \right) + \pi D_{fin} t_{fin}, \quad [m^2] \]

**A_fin_per_tube** [float] Surface area of all fins per tube

\[ A_{fin/tube} = N_{fins/m} \cdot L_{tube} \cdot A_{fin}, \quad [m^2] \]

**A_fin_per_row** [float] Surface area of all fins per row, [m^2]

**A_fin_per_bundle** [float] Surface area of all fins per bundle, [m^2]

**A_fin_per_bay** [float] Surface area of all fins per bay, [m^2]

**A_per_tube** [float] Surface area of combined finned and non-finned area exposed for heat transfer per tube

\[ A_{tube} = A_{bare,\, showing/tube} + A_{fin/tube}, \quad [m^2] \]

**A_per_row** [float] Surface area of combined finned and non-finned area exposed for heat transfer per tube row, [m^2]

**A_per_bundle** [float] Surface area of combined finned and non-finned area exposed for heat transfer per tube bundle, [m^2]

**A_per_bay** [float] Surface area of combined finned and non-finned area exposed for heat transfer per bay, [m^2]

**A** [float] Surface area of combined finned and non-finned area exposed for heat transfer in all bundles and bays combined, [m^2]

**A_increase** [float] Ratio of actual surface area to bare tube surface area

\[ A_{increase} = \frac{A_{tube}}{A_{bare,\, total/tube}}, \quad [-] \]

**A_tube_flow** [float] The area for the fluid to flow in one tube, \( \pi \cdot D_i^2, \quad [m^2] \)

**channels** [int] The number of tubes the fluid flows through at the inlet header, [-]

**tube_volume_per_tube** [float] Fluid volume per tube inside

\[ V_{tube,\, flow} = \frac{\pi}{4} D_i^2 L_{tube}, \quad [m^3] \]

**tube_volume_per_row** [float] Fluid volume of tubes per row, [m^3]

**tube_volume_per_bundle** [float] Fluid volume of tubes per bundle, [m^3]

**tube_volume_per_bay** [float] Fluid volume of tubes per bay, [m^3]

**tube_volume** [float] Fluid volume of tubes in all bundles and bays combined, [m^3]

**A_diagonal_per_bundle** [float] Air flow area along the diagonal plane per bundle

\[ A_d = 2N_{tubes/row} L_{tube}(P_d - D_{tube} - 2N_{fins/m} h_{fin} t_{fin}) + A_{extra,\, side}, \quad [m^2] \]
**A_normal_per_bundle** [float] Air flow area along the normal (transverse) plane; this is normally the minimum flow area, except for some staggered configurations $A_t = N_{\text{tubes/row}}L_{\text{tube}}(P_t - D_{\text{tube}} - 2N_{\text{fins/m}}h_{\text{fin}}) + A_{\text{extra side}}, \text{[m}^2\text{]}$

**A_min_per_bundle** [float] Minimum air flow area per bundle; this is the characteristic area for velocity calculation in most finned tube convection correlations $A_{\text{min}} = \min(A_d, A_t), \text{[m}^2\text{]}$

**A_min_per_bay** [float] Minimum air flow area per bay, [m²]

**A_min** [float] Minimum air flow area, [m²]

**A_face_per_bundle** [float] Face area per bundle $A_{\text{face}} = P_T(1 + N_{\text{tubes/row}})L_{\text{tube}}$; if corbels are used, add 0.5 to tubes/row instead of 1, [m²]

**A_face_per_bay** [float] Face area per bay, [m²]

**A_face** [float] Total face area, [m²]

**flow_area_contraction_ratio** [float] Ratio of $A_{\text{min}}$ to $A_{\text{face}}$, [-]

```
fluids.geometry.SA_partial_sphere(D, h)
```

Calculates surface area of a partial sphere according to [1]. If $h$ is half of $D$, the shape is half a sphere. No bottom is considered in this function. Valid inputs are positive values of $D$ and $h$, with $h$ always smaller or equal to $D$.

\[
a = \sqrt{h(2r - h)}
\]

\[
A = \pi(a^2 + h^2)
\]

**Parameters**

- **D** [float] Diameter of the sphere, [m]
- **h** [float] Height, as measured from the cap to where the sphere is cut off [m]

**Returns**

- **SA** [float] Surface area [m²]

**References**

[1]

**Examples**

```python
>>> SA_partial_sphere(1., 0.7)
2.199114857512855
```

```
fluids.geometry.V_partial_sphere(D, h)
```

Calculates volume of a partial sphere according to [1]. If $h$ is half of $D$, the shape is half a sphere. No bottom is considered in this function. Valid inputs are positive values of $D$ and $h$, with $h$ always smaller or equal to $D$.

\[
a = \sqrt{h(2r - h)}
\]

\[
V = \frac{1}{6}\pi h(3a^2 + h^2)
\]

**Parameters**

- **D** [float] Diameter of the sphere, [m]
**h** [float] Height, as measured up to where the sphere is cut off, [m]

**Returns**

**V** [float] Volume [m^3]

**References**

[1]

**Examples**

```python
>>> v_partial_sphere(1., 0.7)
0.4105014400690663
```

```python
fluids.geometry.V_horiz_conical(D, L, a, h, headonly=False)
```

Calculates volume of a tank with conical ends, according to [1].

\[
V_f = A_f L + \frac{2aR^2}{3} K, \quad 0 \leq h < R
\]

\[
V_f = A_f L + \frac{2aR^2}{3} \pi/2, \quad h = R
\]

\[
V_f = A_f L + \frac{2aR^2}{3} (\pi - K), \quad R < h \leq 2R
\]

\[
K = \cos^{-1} M + M^3 \cosh^{-1} \frac{1}{M} - 2M \sqrt{1 - M^2}
\]

\[
M = \left| \frac{R - h}{R} \right|
\]

\[
Af = R^2 \cos^{-1} \left( \frac{R - h}{R} \right) - (R-h) \sqrt{2Rh - h^2}
\]

**Parameters**

- **D** [float] Diameter of the main cylindrical section, [m]
- **L** [float] Length of the main cylindrical section, [m]
- **a** [float] Distance the cone head extends on one side, [m]
- **h** [float] Height, as measured up to where the fluid ends, [m]

**headonly** [bool, optional] Function returns only the volume of a single head side if True

**Returns**

**V** [float] Volume [m^3]

**References**

[1]

**Examples**

Matching example from [1], with inputs in inches and volume in gallons.
fluids.geometry.V_horiz_ellipsoidal(D, L, a, h, headonly=False)
Calculates volume of a tank with ellipsoidal ends, according to [1].

\[ V_f = A_f L + \pi ah^2 \left( 1 - \frac{h}{3R} \right) \]
\[ A_f = R^2 \cos^{-1} \left( \frac{R-h}{R} \right) - (R-h) \sqrt{2Rh-h^2} \]

Parameters
D [float] Diameter of the main cylindrical section, [m]
L [float] Length of the main cylindrical section, [m]
a [float] Distance the ellipsoidal head extends on one side, [m]
h [float] Height, as measured up to where the fluid ends, [m]
headonly [bool, optional] Function returns only the volume of a single head side if True

Returns
V [float] Volume [m^3]

References
[1]

Examples
Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_horiz_ellipsoidal(D=108, L=156, a=42, h=36)/231.2380.9565415578145
```

fluids.geometry.V_horiz_guppy(D, L, a, h, headonly=False)
Calculates volume of a tank with guppy heads, according to [1].

\[ V_f = A_f L + \frac{2aR^2}{3} \cos^{-1} \left( 1 - \frac{h}{R} \right) + \frac{2a}{9R} \sqrt{2Rh-h^2}(2h-3R)(h+R) \]
\[ A_f = R^2 \cos^{-1} \left( \frac{R-h}{R} \right) - (R-h) \sqrt{2Rh-h^2} \]

Parameters
D [float] Diameter of the main cylindrical section, [m]
L [float] Length of the main cylindrical section, [m]
a [float] Distance the guppy head extends on one side, [m]
h [float] Height, as measured up to where the fluid ends, [m]
headonly [bool, optional] Function returns only the volume of a single head side if True

Returns
V [float] Volume [m^3]
References

[1]

Examples

Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_horiz_guppy(D=108., L=156., a=42., h=36)/231.
1931.7208029476762
```

```
fluids.geometry.V_horiz_spherical(D, L, a, h, headonly=False)
Calculates volume of a tank with spherical heads, according to [1].

\[
V_f = \frac{\pi}{6} (3R^2 + a^2), \quad h = R, |a| \leq R
\]

\[
V_f = A_f L + \frac{\pi}{3} (3R^2 + a^2), \quad h = D, |a| \leq R
\]

\[
V_f = A_f L + \pi ah^2 \left( 1 - \frac{h}{3R} \right), \quad h = 0, \text{ or } |a| = 0, R, -R
\]

\[
V_f = A_f L + \frac{a}{|a|} \left\{ \frac{2r^3}{3} \left[ \cos^{-1} \frac{R^2 - rw}{R(w-r)} + \cos^{-1} \frac{R^2 + rw}{R(w+r)} - \frac{z}{r} \left( 2 + \left( \frac{R}{r} \right)^2 \cos^{-1} \frac{w}{R} \right) - 2 \left( \frac{w^2 - w^3}{3} \right) \tan^{-1} \frac{y}{z} + \frac{4w}{3} \right] \right\}, \quad h \neq R, D; a \neq 0, R, -R, |a| < 0.01D
\]

\[
A_f = R^2 \cos^{-1} \frac{R - h}{R} - (R - h)\sqrt{2Rh - h^2}
\]

\[
r = \frac{a^2 + R^2}{2|a|}
\]

\[
w = R - h
\]

\[
y = \sqrt{2Rh - h^2}
\]

\[
z = \sqrt{r^2 - R^2}
\]

Parameters

- **D** [float] Diameter of the main cylindrical section, [m]
- **L** [float] Length of the main cylindrical section, [m]
- **a** [float] Distance the spherical head extends on one side, [m]
- **h** [float] Height, as measured up to where the fluid ends, [m]
- **headonly** [bool, optional] Function returns only the volume of a single head side if True

Returns

- **V** [float] Volume [m^3]

References

[1]
Examples

Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_horiz_spherical(D=108., L=156., a=42., h=36)/231.
2303.9615116986183
```

`fluids.geometry.V_horiz_torispherical(D, L, f, k, headonly=False)`

Calculates volume of a tank with torispherical heads, according to [1].

\[
V_f = A_f L + 2V_1, \quad 0 \leq h < h_1
\]

\[
V_f = A_f L + 2(V_{1,max} + V_2 + V_3), \quad h_1 < h < h_2
\]

\[
V_f = A_f L + 2[V_{1,max} - V_1(h = D - h) + V_{2,max} + V_{3,max}], \quad h_2 \leq h \leq D
\]

\[
V_1 = \int_0^{\sqrt{2kDh-k^2}} \left[ n^2 \sin^{-1} \left( \frac{\sqrt{n^2-w^2}}{n} \right) - w \sqrt{n^2-w^2} \right] dx
\]

\[
V_2 = \int_0^{kD\cos\alpha} \left[ n^2 \left( \cos^{-1} \frac{w}{n} - \cos^{-1} \frac{g}{n} \right) - w \sqrt{n^2-w^2} + g\sqrt{n^2-g^2} \right] dx
\]

\[
V_3 = \int_w^g \left( r^2 - x^2 \right) \tan^{-1} \left( \frac{\sqrt{g^2-x^2}}{z} \right) dx - \frac{z}{2} \left( g^2 \cos^{-1} \frac{w}{g} - w \sqrt{2g(h - h_1) - (h - h_1)^2} \right)
\]

\[
V_{1,max} = v_1(h = h_1)
\]

\[
v_{2,max} = v_2(h = h_2)
\]

\[
v_{3,max} = \frac{\pi a_1}{6} (3g^2 + a_1^2)
\]

\[
a_1 = fD(1 - \cos \alpha)
\]

\[
\alpha = \sin^{-1} \frac{1 - 2k}{2(f - k)}
\]

\[
n = R - kD + \sqrt{k^2D^2 - x^2}
\]

\[
g = r \sin \alpha
\]

\[
r = fD
\]

\[
h_2 = D - h_1
\]

\[
w = R - h
\]

\[
z = \sqrt{r^2 - g^2}
\]

Parameters

D [float] Diameter of the main cylindrical section, [m]

L [float] Length of the main cylindrical section, [m]

f [float] Dimensionless dish-radius parameter; also commonly given as the product of f and D (fD), which is called dish radius and has units of length, [-]

k [float] Dimensionless knuckle-radius parameter; also commonly given as the product of k and D (kD), which is called the knuckle radius and has units of length, [-]

h [float] Height, as measured up to where the fluid ends, [m]

headonly [bool, optional] Function returns only the volume of a single head side if True

Returns

V [float] Volume [m^3]
References

[1]

Examples

Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_horiz_torispherical(D=108., L=156., f=1., k=0.06, h=36)/231.
2028.626670842139
```

`fluids.geometry.V_vertical_conical(D, a, h)`

Calculates volume of a vertical tank with a convex conical bottom, according to [1]. No provision for the top of the tank is made here.

\[
V_f = \frac{\pi}{4} \left( \frac{Dh}{a} \right) ^ 2 \left( \frac{h}{3} \right), \quad h < a
\]

\[
V_f = \frac{\pi D^2}{4} \left( h - \frac{2a}{3} \right), \quad h \geq a
\]

Parameters

- \( D \) [float] Diameter of the main cylindrical section, [m]
- \( a \) [float] Distance the cone head extends under the main cylinder, [m]
- \( h \) [float] Height, as measured up to where the fluid ends, [m]

Returns

- \( V \) [float] Volume [m^3]

References

[1]

Examples

Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_vertical_conical(132., 33., 24)/231.
250.67461381371024
```

`fluids.geometry.V_vertical_ellipsoidal(D, a, h)`

Calculates volume of a vertical tank with a convex ellipsoidal bottom, according to [1]. No provision for the top of the tank is made here.

\[
V_f = \frac{\pi}{4} \left( \frac{Dh}{a} \right) ^ 2 \left( a - \frac{h}{3} \right), \quad h < a
\]

\[
V_f = \frac{\pi D^2}{4} \left( h - \frac{a}{3} \right), \quad h \geq a
\]

Parameters

- \( D \) [float] Diameter of the main cylindrical section, [m]
Fluids Documentation, Release 0.1

a [float] Distance the ellipsoid head extends under the main cylinder, [m]

h [float] Height, as measured up to where the fluid ends, [m]

Returns

V [float] Volume [m^3]

References

[1]

Examples

Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_vertical_ellipsoidal(132., 33., 24)/231.
783.3581681678445
```

fluids.geometry.V_vertical_spherical(D, a, h)
Calculates volume of a vertical tank with a convex spherical bottom, according to [1]. No provision for the top of the tank is made here.

\[
V_f = \frac{\pi h^2}{4} \left( 2a + \frac{D^2}{2a} - \frac{4h}{3} \right), \quad h < a
\]

\[
V_f = \frac{\pi}{4} \left( \frac{2a^3}{3} - \frac{aD^2}{2} + hD^2 \right), \quad h \geq a
\]

Parameters

D [float] Diameter of the main cylindrical section, [m]
a [float] Distance the spherical head extends under the main cylinder, [m]
h [float] Height, as measured up to where the fluid ends, [m]

Returns

V [float] Volume [m^3]

References

[1]

Examples

Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_vertical_spherical(132., 33., 24)/231.
583.6018352850442
```

fluids.geometry.V_vertical_torispherical(D, f, k, h)
Calculates volume of a vertical tank with a convex torispherical bottom, according to [1]. No provision for the top of the tank is made here.

\[
V_f = \frac{\pi h^2}{4} \left( 2a_1 + \frac{D_1^2}{2a_1} - \frac{4h}{3} \right), \quad 0 \leq h \leq a_1
\]
\[ V_f = \frac{\pi}{4} \left( \frac{2a_1^3}{3} + \frac{a_1 D^2}{2} \right) + \pi u \left[ \frac{D}{2} - kD \right]^2 + \frac{\pi u^3}{2} + \pi D(1 - 2k) \left[ \frac{2u - t}{4} \sqrt{s + tu - u^2} + \frac{t\sqrt{s}}{4} + \frac{k^2 D^2}{2} \right] \]

\[ V_f = \frac{\pi}{4} \left( \frac{2a_1^3}{3} + \frac{a_1 D^2}{2} \right) + \frac{\pi t}{2} \left[ \frac{D}{2} - kD \right]^2 + \frac{\pi D^3}{12} + \pi D(1 - 2k) \left[ \frac{t\sqrt{s}}{4} + \frac{k^2 D^2}{2} \sin^{-1}(\cos \alpha) \right] + \frac{\pi D^2}{4} \left[ h - (a_1 + a_2) \right] \]

\[ \alpha = \sin^{-1} \left( \frac{1 - 2k}{2(f - k)} \right) \]

\[ a_1 = fD(1 - \cos \alpha) \]

\[ a_2 = kD \cos \alpha \]

\[ D_1 = 2fD \sin \alpha \]

\[ s = (kD \sin \alpha)^2 \]

\[ t = 2a_2 \]

\[ u = h - fD(1 - \cos \alpha) \]

**Parameters**

- **D** [float] Diameter of the main cylindrical section, [m]
- **f** [float] Dimensionless dish-radius parameter; also commonly given as the product of \(f\) and \(D\) (\(fD\)), which is called dish radius and has units of length, [-]
- **k** [float] Dimensionless knuckle-radius parameter; also commonly given as the product of \(k\) and \(D\) (\(kD\)), which is called the knuckle radius and has units of length, [-]
- **h** [float] Height, as measured up to where the fluid ends, [m]

**Returns**

- **V** [float] Volume [m³]

**References**

[1]

**Examples**

Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_vertical_torispherical(D=132., f=1.0, k=0.06, h=24)/231.
904.0688283793511
```

**Calculates volume of a vertical tank with a concave conical bottom, according to [1]. No provision for the top of the tank is made here.**

\[ V = \frac{\pi D^2}{12} \left( 3h + a - \frac{(a + h)^3}{a^2} \right), \quad 0 \leq h < |a| \]

\[ V = \frac{\pi D^2}{12}(3h + a), \quad h \geq |a| \]

**Parameters**
D [float] Diameter of the main cylindrical section, [m]
a [float] Negative distance the cone head extends inside the main cylinder, [m]
h [float] Height, as measured up to where the fluid ends, [m]

Returns
V [float] Volume [m^3]

References
[1]

Examples
Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_vertical_conical_concave(D=113., a=-33, h=15)/231
251.15825565795188
```

```
fluids.geometry.V_vertical_ellipsoidal_concave(D, a, h)
Calculates volume of a vertical tank with a concave ellipsoidal bottom, according to [1]. No provision for the top of the tank is made here.

V = \pi \frac{D^2}{12} \left( 3h + 2a - \frac{(a + h)(2a - h)}{a^2} \right), \quad 0 \leq h < |a|

V = \pi \frac{D^2}{12} (3h + 2a), \quad h \geq |a|

Parameters
D [float] Diameter of the main cylindrical section, [m]
a [float] Negative distance the ellipsoidal head extends inside the main cylinder, [m]
h [float] Height, as measured up to where the fluid ends, [m]

Returns
V [float] Volume [m^3]

References
[1]

Examples
Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_vertical_ellipsoidal_concave(D=113., a=-33, h=15)/231
44.84968851034856
```
fluids.geometry.V_vertical_spherical_concave \((D, a, h)\)
Calculates volume of a vertical tank with a concave spherical bottom, according to [1]. No provision for the top of the tank is made here.

\[
V = \frac{\pi}{12} \left[ 3D^2h + \frac{a}{2}(3D^2 + 4a^2) + (a + h)^3 \left( 4 - \frac{3D^2 + 12a^2}{2a(a + h)} \right) \right], \quad 0 \leq h < |a|
\]

\[
V = \frac{\pi}{12} \left[ 3D^2h + \frac{a}{2}(3D^2 + 4a^2) \right], \quad h \geq |a|
\]

**Parameters**

- \(D\) [float] Diameter of the main cylindrical section, [m]
- \(a\) [float] Negative distance the spherical head extends inside the main cylinder, [m]
- \(h\) [float] Height, as measured up to where the fluid ends, [m]

**Returns**

- \(V\) [float] Volume [m\(^3\)]

**References**

[1]

**Examples**

Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_vertical_spherical_concave(D=113., a=-33, h=15)/231
112.81405437348528
```

fluids.geometry.V_vertical_torispherical_concave \((D, f, k, h)\)
Calculates volume of a vertical tank with a concave torispherical bottom, according to [1]. No provision for the top of the tank is made here.

\[
V = \pi D^2h/4 - v_1(h = a_1 + a_2) + v_1(h = a_1 + a_2 - h), \quad 0 \leq h < a_2
\]

\[
V = \pi D^2h/4 - v_1(h = a_1 + a_2) + v_2(h = a_1 + a_2 - h), \quad a_2 \leq h < a_1 + a_2
\]

\[
V = \pi D^2h/4 - v_1(h = a_1 + a_2) + 0, \quad h \geq a_1 + a_2
\]

\[
v_1 = \frac{\pi}{4} \left( \frac{2a_1^3}{3} + \frac{a_1D_1^2}{2} \right) + \pi u \left( \frac{D}{2} - kd \right)^2 + s + \frac{\pi tu^2}{2} - \frac{\pi u^3}{3} + \pi D(1 - 2k) \left[ \frac{2u - t}{4} \sqrt{s + tu - u^2} + \frac{t\sqrt{s}}{4} + \frac{kD^2}{2} \right]
\]

\[
v_2 = \frac{\pi}{4} \left( 2a_1 + \frac{D_1^2}{2a_1} - \frac{4h}{3} \right)
\]

\[
\alpha = \sin^{-1} \left( \frac{1 - 2k}{2(f - k)} \right)
\]

\[
a_1 = fD(1 - \cos \alpha)
\]

\[
a_2 = kD \cos \alpha
\]

\[
D_1 = 2fD \sin \alpha
\]
\[
\begin{align*}
    s &= (kD \sin \alpha)^2 \\
    t &= 2a_2 \\
    u &= h - fD(1 - \cos \alpha)
\end{align*}
\]

**Parameters**

- **D** [float] Diameter of the main cylindrical section, [m]
- **f** [float] Dimensionless dish-radius parameter; also commonly given as the product of \( f \) and \( D \) (\( fD \)), which is called dish radius and has units of length, [-]
- **k** [float] Dimensionless knuckle-radius parameter; also commonly given as the product of \( k \) and \( D \) (\( kD \)), which is called the knuckle radius and has units of length, [-]
- **h** [float] Height, as measured up to where the fluid ends, [m]

**Returns**

- **V** [float] Volume \([\text{m}^3]\)

**References**

[1]

**Examples**

Matching example from [1], with inputs in inches and volume in gallons.

```python
>>> V_vertical_torispherical_concave(D=113., f=0.71, k=0.081, h=15)/231
103.88569287163769
```

`fluids.geometry.a_torispherical(D,f,k)`

Calculates depth of a torispherical head according to [1].

\[
\begin{align*}
    a &= a_1 + a_2 \\
    \alpha &= \sin^{-1} \frac{1 - 2k}{2(f - k)} \\
    a_1 &= fD(1 - \cos \alpha) \\
    a_2 &= kD \cos \alpha
\end{align*}
\]

**Parameters**

- **D** [float] Diameter of the main cylindrical section, [m]
- **f** [float] Dimensionless dish-radius parameter; also commonly given as the product of \( f \) and \( D \) (\( fD \)), which is called dish radius and has units of length, [-]
- **k** [float] Dimensionless knuckle-radius parameter; also commonly given as the product of \( k \) and \( D \) (\( kD \)), which is called the knuckle radius and has units of length, [-]

**Returns**

- **a** [float] Depth of head [m]
References

[1]

Examples

Example from [1].

```python
>>> a_torispherical(D=96., f=0.9, k=0.2)
25.684268924767125
```

**fluids.geometry.SA_ellipsoidal_head**(D, a)

Calculates the surface area of an ellipsoidal head according to [1]. Formula below is for the full shape, the result of which is halved. The formula also does not support D being larger than a; this is ensured by simply swapping the variables if necessary, as geometrically the result is the same. In the equations, a is the same and c is D.

\[
SA = 2\pi a^2 + \frac{\pi c^2}{e_1} \ln \left( \frac{1 + e_1}{1 - e_1} \right)
\]

\[
e_1 = \sqrt{1 - \frac{c^2}{a^2}}
\]

**Parameters**

- D [float] Diameter of the main cylindrical section, [m]
- a [float] Distance the ellipsoidal head extends, [m]

**Returns**

- SA [float] Surface area [m^2]

References

[1]

Examples

**Spherical case**

```python
>>> SA_ellipsoidal_head(2, 1)
6.283185307179586
```

**fluids.geometry.SA_conical_head**(D, a)

Calculates the surface area of a conical head according to [1].

\[
SA = \frac{\pi D}{2} \sqrt{a^2 + \left( \frac{D}{2} \right)^2}
\]

**Parameters**

- D [float] Diameter of the main cylindrical section, [m]
- a [float] Distance the conical head extends, [m]

**Returns**

- SA [float] Surface area [m^2]
References

[1]

Examples

```python
>>> SA_conical_head(2, 1)
4.442882938158366
```
fluids.geometry.SA_guppy_head(D, a)
Calculates the surface area of a guppy head according to [1]. Some work was involved in combining formulas for the ellipse of the head, and the conic section on the sides.

\[
SA = \frac{\pi D}{4} \sqrt{D^2 + a^2} + \frac{\pi D}{2} a
\]

Parameters

- D [float] Diameter of the main cylindrical section, [m]
- a [float] Distance the conical head extends, [m]

Returns

SA [float] Surface area [m^2]

References

[1]

Examples

```python
>>> SA_guppy_head(2, 1)
6.654000019110157
```
fluids.geometry.SA_torispheroidal(D, f, k)
Calculates surface area of a torispherical head according to [1]. Somewhat involved. Equations are adapted to be used for a full head.

\[
SA = S_1 + S_2
\]

\[
S_1 = 2\pi D^2 f_d \alpha
\]

\[
S_2 = 2\pi D^2 f_k \left( \alpha - \alpha_1 + (0.5 - f_k) \left( \sin^{-1} \left( \frac{\alpha - \alpha_2}{f_k} \right) - \sin^{-1} \left( \frac{\alpha_1 - \alpha_2}{f_k} \right) \right) \right)
\]

\[
\alpha_1 = f_d \left( 1 - \sqrt{1 - \left( \frac{0.5 - f_k}{f_d - f_k} \right)^2} \right)
\]

\[
\alpha_2 = f_d - \sqrt{f_d^2 - 2f_d f_k + f_k - 0.25}
\]

\[
\alpha = \frac{a}{D_i}
\]

Parameters
Fluids Documentation, Release 0.1

D [float] Diameter of the main cylindrical section, [m]

f [float] Dimensionless dish-radius parameter; also commonly given as the product of $f$ and $D$ ($fD$), which is called dish radius and has units of length, [-]

k [float] Dimensionless knuckle-radius parameter; also commonly given as the product of $k$ and $D$ ($kD$), which is called the knuckle radius and has units of length, [-]

Returns

SA [float] Surface area [m^2]

References

[1]

Examples

Example from [1].

```python
>>> SA_torispheroidal(D=2.54, f=1.039370079, k=0.062362205)
6.00394283477063
```

Calculates partially full volume of a vertical or horizontal tank with different head types according to [1].

Parameters

h [float] Height of the liquid in the tank, [m]

D [float] Diameter of the cylindrical section of the tank, [m]

L [float] Length of the main cylindrical section of the tank, [m]

horizontal [bool, optional] Whether or not the tank is a horizontal or vertical tank

sideA [string, optional] The left (or bottom for vertical) head of the tank’s type; one of [None, ‘conical’, ‘ellipsoidal’, ‘torispherical’, ‘guppy’, ‘spherical’].

sideB [string, optional] The right (or top for vertical) head of the tank’s type; one of [None, ‘conical’, ‘ellipsoidal’, ‘torispherical’, ‘guppy’, ‘spherical’].

sideA_a [float, optional] The distance the head as specified by sideA extends down or to the left from the main cylindrical section, [m]

sideB_a [float, optional] The distance the head as specified by sideB extends up or to the right from the main cylindrical section, [m]

sideA_f [float, optional] Dimensionless dish-radius parameter for side A; also commonly given as the product of $f$ and $D$ ($fD$), which is called dish radius and has units of length, [-]

sideA_k [float, optional] Dimensionless knuckle-radius parameter for side A; also commonly given as the product of $k$ and $D$ ($kD$), which is called the knuckle radius and has units of length, [-]

sideB_f [float, optional] Dimensionless dish-radius parameter for side B; also commonly given as the product of $f$ and $D$ ($fD$), which is called dish radius and has units of length, [-]
sideB_k [float, optional] Dimensionless knuckle-radius parameter for side B; also commonly given as the product of $k$ and $D$ ($kD$), which is called the knuckle radius and has units of length, [-]

Returns

V [float] Volume up to h [m^3]

References

[1]

Examples

```python
>>> V_from_h(h=7, D=1.5, L=5., horizontal=False, sideA='conical',
... sideB='conical', sideA_a=2., sideB_a=1.)
10.013826583317465
```

Calculates the surface area of a cylindrical tank with optional heads. In the degenerate case of being provided with only $D$ and $L$, provides the surface area of a cylinder.

Parameters

- **D** [float] Diameter of the cylindrical section of the tank, [m]
- **L** [float] Length of the main cylindrical section of the tank, [m]
- **sideA** [string, optional] The left (or bottom for vertical) head of the tank’s type; one of [None, ‘conical’, ‘ellipsoidal’, ‘torispherical’, ‘guppy’, ‘spherical’].
- **sideB** [string, optional] The right (or top for vertical) head of the tank’s type; one of [None, ‘conical’, ‘ellipsoidal’, ‘torispherical’, ‘guppy’, ‘spherical’].
- **sideA_a** [float, optional] The distance the head as specified by sideA extends down or to the left from the main cylindrical section, [m]
- **sideB_a** [float, optional] The distance the head as specified by sideB extends up or to the right from the main cylindrical section, [m]
- **sideA_f** [float, optional] Dish-radius parameter for side A; fD = dish radius [1/m]
- **sideA_k** [float, optional] knuckle-radius parameter for side A; kD = knuckle radius [1/m]
- **sideB_f** [float, optional] Dish-radius parameter for side B; fD = dish radius [1/m]
- **sideB_k** [float, optional] knuckle-radius parameter for side B; kD = knuckle radius [1/m]

Returns

- **SA** [float] Surface area of the tank [m^2]
- **areas** [tuple, only returned if full_output == True] (sideA_SA, sideB_SA, lateral_SA)

Other Parameters

- **full_output** [bool, optional] Returns a tuple of (sideA_SA, sideB_SA, lateral_SA) if True
Examples

Cylinder, Spheroid, Long Cones, and spheres. All checked.

```python
>>> SA_tank(D=2, L=2)
18.84955592153876
>>> SA_tank(D=1., L=0, sideA='ellipsoidal', sideA_a=2, sideB='ellipsoidal',
... sideB_a=2)
28.480278854014387
>>> SA_tank(D=1., L=5, sideA='conical', sideA_a=2, sideB='conical',
... sideB_a=2)
22.18452243965656
>>> SA_tank(D=1., L=5, sideA='spherical', sideA_a=0.5, sideB='spherical',
... sideB_a=0.5)
18.84955592153876
```

**flows.geometry.sphericity** *(A, V)*

Returns the sphericity of a particle of surface area *A* and volume *V*. Sphericity is the ratio of the surface area of a sphere with the same volume as the particle (equivalent diameter) to the actual surface area of the particle.

\[
\Psi = \frac{A \text{ of sphere with } V_p}{A_p} = \frac{\pi^{\frac{1}{3}} (6V_p)^{\frac{2}{3}}}{A_p}
\]

**Parameters**

- **A** [float] Surface area of particle, [m^2]
- **V** [float] Volume of particle, [m^3]

**Returns**

- **Psi** [float] Sphericity [-]

**Notes**

All non-spherical particles have sphericities less than 1 but greater than 0. Many common geometrical shapes have their results calculated exactly in [2].

**References**

[1], [2]

**Examples**

```python
>>> sphericity(10., 2.)
0.767663317071005
```

For a cube of side length *a*=3, the surface area is 6*a^2=54 and volume *a^3=27*. Its sphericity is then:

```python
>>> sphericity(A=54, V=27)
0.8059959770082346
```

**flows.geometry.aspect_ratio** *(Dmin, Dmax)*

Returns the aspect ratio of a shape with minimum and maximum dimension, *Dmin* and *Dmax*.

\[
A_R = \frac{D_{\text{min}}}{D_{\text{max}}}
\]
Parameters

- **Dmin**  [float] Minimum dimension, [m]
- **Dmax**  [float] Maximum dimension, [m]

Returns

- **a_r**  [float] Aspect ratio [-]

Examples

```python
>>> aspect_ratio(.2, 2)
0.1
```

**fluids.geometry.circularity**(*A*, *P*)

Returns the circularity of a shape with area *A* and perimeter *P*.

\[
f_{circ} = \frac{4\pi A}{P^2}
\]

Defined to be 1 for a circle. Used to characterize particles. Any non-circular shape must have a circularity less than one.

Parameters

- **A**  [float] Area of the shape, [m^2]
- **P**  [float] Perimeter of the shape, [m]

Returns

- **f_circ**  [float] Circularity of the shape [-]

Examples

Square, side length = 2 (all squares are the same):

```python
>>> circularity(A=(2*2), P=4*2)
0.7853981633974483
```

Rectangle, one side length = 1, second side length = 100

```python
>>> D1 = 1
>>> D2 = 100
>>> A = D1*D2
>>> P = 2*D1 + 2*D2
>>> circularity(A, P)
0.030796908671598795
```

**fluids.geometry.A_cylinder**(*D*, *L*)

Returns the surface area of a cylinder.

\[
A = \pi DL + 2 \cdot \frac{\pi D^2}{4}
\]

Parameters

- **D**  [float] Diameter of the cylinder, [m]
- **L**  [float] Length of the cylinder, [m]
Returns

\[ A \] [float] Surface area \([\text{m}^2]\]

Examples

```python
>>> A_cylinder(0.01, .1)
0.0032986722862692833
```

```python
fluids.geometry.V_cylinder(D, L)
Returns the volume of a cylinder.

\[ V = \frac{\pi D^2}{4} L \]

Parameters

- \( D \) [float] Diameter of the cylinder, \([\text{m}]\)
- \( L \) [float] Length of the cylinder, \([\text{m}]\)

Returns

\[ V \] [float] Volume \([\text{m}^3]\]

Examples

```python
>>> V_cylinder(0.01, .1)
7.853981633974484e-06
```

```python
fluids.geometry.A_hollow_cylinder(Di, Do, L)
Returns the surface area of a hollow cylinder.

\[ A = \pi D_o L + \pi D_i L + 2 \left( \frac{\pi D_o^2}{4} - 2 \cdot \frac{\pi D_i^2}{4} \right) \]

Parameters

- \( D_i \) [float] Diameter of the hollow in the cylinder, \([\text{m}]\)
- \( D_o \) [float] Diameter of the exterior of the cylinder, \([\text{m}]\)
- \( L \) [float] Length of the cylinder, \([\text{m}]\)

Returns

\[ A \] [float] Surface area \([\text{m}^2]\)

Examples

```python
>>> A_hollow_cylinder(0.005, 0.01, 0.1)
0.004830198704894308
```

```python
fluids.geometry.V_hollow_cylinder(Di, Do, L)
Returns the volume of a hollow cylinder.

\[ V = \frac{\pi D_o^2}{4} L - L \frac{\pi D_i^2}{4} \]
```

2.10. Tank and helical coil sizing (fluids.geometry)
Parameters

- **Di** [float] Diameter of the hollow in the cylinder, [m]
- **Do** [float] Diameter of the exterior of the cylinder, [m]
- **L** [float] Length of the cylinder, [m]

Returns

- **V** [float] Volume [m³]

Examples

```python
>>> V_hollow_cylinder(0.005, 0.01, 0.1)
5.890486225480862e-06
```

```
fluids.geometry.A_multiple_hole_cylinder(Do, L, holes)
```

Returns the surface area of a cylinder with multiple holes. Calculation will naively return a negative value or other impossible result if the number of cylinders added is physically impossible. Holes may be of different shapes, but must be perpendicular to the axis of the cylinder.

\[
A = \pi D_o L + 2 \cdot \frac{\pi D^2_o}{4} + \sum_i \left( \pi D_i L - 2 \cdot \frac{\pi D^2_i}{4} \right)
\]

Parameters

- **Do** [float] Diameter of the exterior of the cylinder, [m]
- **L** [float] Length of the cylinder, [m]
- **holes** [list] List of tuples containing (diameter, count) pairs of descriptions for each of the holes sizes.

Returns

- **A** [float] Surface area [m²]

Examples

```python
>>> A_multiple_hole_cylinder(0.01, 0.1, [(0.005, 1)])
0.004830198704894308
```

```
fluids.geometry.V_multiple_hole_cylinder(Do, L, holes)
```

Returns the solid volume of a cylinder with multiple cylindrical holes. Calculation will naively return a negative value or other impossible result if the number of cylinders added is physically impossible.

\[
V = \frac{\pi D^2_o}{4} L - L \frac{\pi D^2_i}{4}
\]

Parameters

- **Do** [float] Diameter of the exterior of the cylinder, [m]
- **L** [float] Length of the cylinder, [m]
- **holes** [list] List of tuples containing (diameter, count) pairs of descriptions for each of the holes sizes.

Returns

- **V** [float] Volume [m³]
Examples

```python
>>> V_multiple_hole_cylinder(0.01, 0.1, [(0.005, 1)])
5.890486225480862e-06
```

```
fluids.geometry.pitch_angle_solver(angle=None, pitch=None, pitch_parallel=None, pitch_normal=None)
```

Utility to take any two of `angle`, `pitch`, `pitch_parallel`, and `pitch_normal` and calculate the other two. This is useful for applications with tube banks, as in shell and tube heat exchangers or air coolers and allows for a wider range of user input.

\[ \text{pitch normal} = \text{pitch} \cdot \sin(\text{angle}) \]

\[ \text{pitch parallel} = \text{pitch} \cdot \cos(\text{angle}) \]

**Parameters**

- `angle` [float, optional] The angle of the tube layout, [degrees]
- `pitch` [float, optional] The shortest distance between tube centers; defined in relation to the flow direction only, [m]
- `pitch_parallel` [float, optional] The distance between tube center along a line parallel to the flow; has been called `longitudinal` pitch, `pp`, `s2`, `SL`, and `p2`, [m]
- `pitch_normal` [float, optional] The distance between tube centers in a line 90° to the line of flow; has been called the `transverse` pitch, `pn`, `s1`, `ST`, and `p1`, [m]

**Returns**

- `angle` [float] The angle of the tube layout, [degrees]
- `pitch` [float] The shortest distance between tube centers; defined in relation to the flow direction only, [m]
- `pitch_parallel` [float] The distance between tube center along a line parallel to the flow; has been called `longitudinal` pitch, `pp`, `s2`, `SL`, and `p2`, [m]
- `pitch_normal` [float] The distance between tube centers in a line 90° to the line of flow; has been called the `transverse` pitch, `pn`, `s1`, `ST`, and `p1`, [m]

**Notes**

For the 90 and 0 degree case, the normal or parallel pitches can be zero; given the angle and the zero value, obviously is it not possible to calculate the pitch and a math error will be raised.

No exception will be raised if three or four inputs are provided; the other two will simply be calculated according to the list of if statements used.

An exception will be raised if only one input is provided.

**References**

[1]
Examples

```python
>>> pitch_angle_solver(pitch=1, angle=30)
(30, 1, 0.8660254037844387, 0.49999999999999994)
```

## 2.11 Jet Pump (ejector/eductor) Sizing and Rating (fluids.jet_pump)

The `fluids.jet_pump.liquid_jet_pump` function calculates the remaining two variables in a liquid jet pump, using a model presented in [1] as well as [2], [3], and [4].

\[
N = \frac{2R + \frac{2CM^2R^2}{1-R^2} - R^2(1 + CM)(1 + M)(1 + K_m + K_d + \alpha^2) - \frac{CM^2R^2}{(1-R)^2}(1 + K_s)}{(1 + K_p) - 2R - \frac{2CM^2R^2}{1-R^2} + R^2(1 + CM)(1 + M)(1 + K_m + K_d + \alpha^2) + (1 - J) \left( \frac{CM^2R^2}{(1-R)^2} \right) (1 + K_s)}
\]

\[
P_1 - P_2 = \frac{1}{2} \rho_p \left( \frac{Q_p}{A_n} \right)^2 \left[ (1 + K_p) - C(1 + K_s) \left( \frac{MR}{1-R} \right)^2 \right]
\]

- Pressure ratio: \( N = \frac{P_b - P_2}{P_1 - P_b} \)
- Volume flow ratio: \( M = \frac{Q_s}{Q_p} \)
- Jet pump efficiency: \( \eta = M \cdot N = \frac{Q_s(P_b - P_2)}{Q_p(P_1 - P_b)} \)

\[
R = \frac{A_n}{A_m}
\]

\[
C = \frac{\rho_s}{\rho_p}
\]

There is no guarantee a solution will be found for the provided variable values, but every combination of two missing variables are supported.

**Parameters**

- `rhop` [float] The density of the primary (motive) fluid, [kg/m^3]
- `rhos` [float] The density of the secondary fluid (drawn from the vacuum chamber), [kg/m^3]
- `Kp` [float, optional] The primary nozzle loss coefficient, [-]
- `Ks` [float, optional] The secondary inlet loss coefficient, [-]
- `Km` [float, optional] The mixing chamber loss coefficient, [-]
- `Kd` [float, optional] The diffuser loss coefficient, [-]
- `d_nozzle` [float, optional] The inside diameter of the primary fluid’s nozzle, [m]
- `d_mixing` [float, optional] The diameter of the mixing chamber, [m]
- `d_diffuser` [float, optional] The diameter of the diffuser at its exit, [m]
- `Qp` [float, optional] The volumetric flow rate of the primary fluid, [m^3/s]
Fluids Documentation, Release 0.1

Qs [float, optional] The volumetric flow rate of the secondary fluid, [m^3/s]
P1 [float, optional] The pressure of the primary fluid entering its nozzle, [Pa]
P2 [float, optional] The pressure of the secondary fluid at the entry of the ejector, [Pa]
P5 [float, optional] The pressure at the exit of the diffuser, [Pa]
nozzle_retracted [bool, optional] Whether or not the primary nozzle’s exit is before the mixing chamber, or somewhat inside it, [-]
max_variations [int, optional] When the initial guesses do not lead to a converged solution, try this many more guesses at converging the problem, [-]

Returns

solution [dict] Dictionary of calculated parameters, [-]

Notes

The assumptions of the model are:

• The flows are one dimensional except in the mixing chamber.
• The mixing chamber has constant cross-sectional area.
• The mixing happens entirely in the mixing chamber, prior to entry into the diffuser.
• The primary nozzle is in a straight line with the middle of the mixing chamber.
• Both fluids are incompressible, and have no excess volume on mixing.
• Primary and secondary flows both enter the mixing throat with their own uniform velocity distribution; the mixed stream leaves with a uniform velocity profile.
• If the secondary fluid is a gas, it undergoes isothermal compression in the throat and diffuser.
• If the secondary fluid is a gas or contains a bubbly gas, it is homogeneously distributed in a continuous liquid phase.
• Heat transfer between the fluids is negligible - there is no change in density due to temperature changes
• The change in the solubility of a dissolved gas, if there is one, is negligibly changed by temperature or pressure changes.

The model can be derived from the equations in liquid_jet_pump_ancillary and the following:

Conservation of energy at the primary nozzle, secondary inlet, and diffuser exit: .. math:

\[ P_1 = P_3 + \frac{\rho_p}{2} V_n^2 + K_p \left( \frac{1}{2} \rho_p V_n^2 \right) \]

\[ P_2 = P_3 + \frac{\rho_s}{2} V_s^2 + K_s \left( \frac{1}{2} \rho_s V_s^2 \right) \]

\[ P_5 = P_4 + \frac{\rho_d}{2} V_d^2 - K_d \left( \frac{1}{2} \rho_d V_d^2 \right) \]

Continuity of the ejector:

\[ \rho_p Q_p + \rho_s Q_s = \rho_d Q_d \]

References

[1], [2], [3], [4]
Examples

```python
>>> ans = liquid_jet_pump(rhop=998., rhos=1098., Km=.186, Kd=0.12, Ks=0.11,
... Kp=0.04, d_mixing=0.045, Qs=0.01, Qp=.01, P2=133600,
... P5=200E3, nozzle_retracted=False, max_variations=10000)

```  

```python
>>> s = []
>>> for key, value in ans.items():
...     s.append(' %s: %g' % (key, value))

```  

```python
sorted(s)

['M: 1', 'N: 0.293473', 'P1: 426256', 'P2: 133600', 'P5: 200000', 'Qp: 0.01', 'Qs: 0.01', 'R: 0.247404', 'alpha: 1e-06', 'd_diffuser: 45', 'd_mixing: 0.045', 'd_nozzle: 0.0223829', 'efficiency: 0.293473']

```  

```python
fluids.jet_pump.liquid_jet_pump_ancillary(rhop, rhos, Kp, Ks, d_nozzle=None,
... d_mixing=None, Qp=None, Qs=None,
P1=None, P2=None)
```

Calculates the remaining variable in a liquid jet pump when solving for one if the inlet variables only and the rest of them are known. The equation comes from conservation of energy and momentum in the mixing chamber.

The variable to be solved for must be one of `d_nozzle`, `d_mixing`, `Qp`, `Qs`, `P1`, or `P2`.

\[ P_1 - P_2 = \frac{1}{2} \rho_p V_n^2 (1 + K_p) - \frac{1}{2} \rho_s V_n^3 (1 + K_s) \]

Rearrange to express \( V_3 \) in terms of \( V_n \), and using the density ratio \( C \), the expression becomes:

\[ P_1 - P_2 = \frac{1}{2} \rho_p V_n^2 \left( (1 + K_p) - C(1 + K_s) \left( \frac{MR}{1 - R} \right)^2 \right) \]

Using the primary nozzle area and flow rate:

\[ P_1 - P_2 = \frac{1}{2} \rho_p \left( \frac{Q_p}{A_n} \right)^2 \left( (1 + K_p) - C(1 + K_s) \left( \frac{MR}{1 - R} \right)^2 \right) \]

For \( P, P_2, Q_s, \) and \( Q_p \), the equation can be rearranged explicitly for them. For \( d_mixing \) and \( d_nozzle \), a bounded solver is used searching between 1E-9 m and 20 times the other diameter which was specified.

Parameters

- `rhop` [float] The density of the primary (motive) fluid, [kg/m^3]
- `rhos` [float] The density of the secondary fluid (drawn from the vacuum chamber), [kg/m^3]
- `Kp` [float] The primary nozzle loss coefficient, [-]
- `Ks` [float] The secondary inlet loss coefficient, [-]
- `d_nozzle` [float, optional] The inside diameter of the primary fluid’s nozzle, [m]
- `d_mixing` [float, optional] The diameter of the mixing chamber, [m]
- `Qp` [float, optional] The volumetric flow rate of the primary fluid, [m^3/s]
- `Qs` [float, optional] The volumetric flow rate of the secondary fluid, [m^3/s]
- `P1` [float, optional] The pressure of the primary fluid entering its nozzle, [Pa]
- `P2` [float, optional] The pressure of the secondary fluid at the entry of the ejector, [Pa]

Returns

- `solution` [float] The parameter not specified (one of `d_nozzle`, `d_mixing`, `Qp`, `Qs`, `P1`, or `P2`), (units of m, m, m^3/s, m^3/s, Pa, or Pa respectively)
Notes

The following SymPy code was used to obtain the analytical formulas (they are not shown here due to their length):

```python
>>> from sympy import *
>>> A_nozzle, A_mixing, Qs, Qp, P1, P2, rhos, rhop, Ks, Kp = symbols('A_nozzle, A_mixing, Qs, Qp, P1, P2, rhos, rhop, Ks, Kp')
>>> R = A_nozzle/A_mixing
>>> M = Qs/Qp
>>> C = rhos/rhop
>>> rhs = rhop/2*(Qp/A_nozzle)**2*((1+Kp) - C*(1 + Ks)*((M*R)/(1-R))**2)
>>> new = Eq(P1 - P2, rhs)
>>> #solve(new, Qp)
>>> #solve(new, Qs)
>>> #solve(new, P1)
>>> #solve(new, P2)
```

References

[1]

Examples

Calculating primary fluid nozzle inlet pressure P1:

```python
>>> liquid_jet_pump_ancillary(rhop=998., rhos=1098., Ks=0.11, Kp=.04,... P2=133600, Qp=0.01, Qs=0.01, d_mixing=0.045, d_nozzle=0.02238)
426434.60314398084
```

2.12 Mixing (fluids.mixing)

The `fluids.mixing.agitator_time_homogeneous(N, P, T, H, mu, rho, D=None, homogeneity=0.95)` function calculates time for a fluid mixing in a tank with an impeller to reach a specified level of homogeneity, according to [1].

\[
N_p = \frac{P_g}{\rho N^3 D^5}
\]

\[
Re_{imp} = \frac{\rho D^2 N}{\mu}
\]

\[
\text{constant} = N_p^{1/3} Re_{imp}
\]

\[
Fo = 5.2/\text{constant}\text{for turbulent regime}
\]

\[
Fo = (183/\text{constant})^2\text{for transition regime}
\]

Parameters

- N [float]: Speed of impeller, [revolutions/s]
- P [float]: Actual power required to mix, ignoring mechanical inefficiencies [W]
- T [float]: Tank diameter, [m]
Fluids Documentation, Release 0.1

**H [float] Tank height, [m]**

**mu [float] Mixture viscosity, [Pa*s]**

**rho [float] Mixture density, [kg/m^3]**

**D [float, optional] Impeller diameter [m]**

**homogeneity [float, optional] Fraction completion of mixing, []**

**Returns**

**t [float] Time for specified degree of homogeneity [s]**

**Notes**

If impeller diameter is not specified, assumed to be 0.5 tank diameters.

The first example is solved forward rather than backwards here. A rather different result is obtained, but is accurate.

No check to see if the mixture if laminar is currently implemented. This would under predict the required time.

**References**

[1]

**Examples**

```python
>>> agitator_time_homogeneous(D=36*.0254, N=56/60., P=957., T=1.83, H=1.83, mu=0.018, rho=1020, homogeneity=.995)
15.143198226374668
```

```python
>>> agitator_time_homogeneous(D=1, N=125/60., P=298., T=3, H=2.5, mu=.5, rho=980, homogeneity=.95)
67.7575069865228
```

**fluids.mixing.Kp_helical_ribbon_Rieger (D, h, nb, pitch, width, T)**

Calculates product of power number and Reynolds number for a specified geometry for a helical ribbon mixer in the laminar regime. One of several correlations listed in [1], it used more data than other listed correlations and was recommended.

\[
K_p = 82.8 \frac{h}{D} \left( \frac{c}{D} \right)^{-0.38} \left( \frac{p}{D} \right)^{-0.35} \left( \frac{w}{D} \right)^{0.20} n_b^{0.78}
\]

**Parameters**

**D [float] Impeller diameter [m]**

**h [float] Ribbon mixer height, [m]**

**nb [float] Number of blades, [-]**

**pitch [float] Height of one turn around a helix [m]**

**width [float] Width of one blade [m]**

**T [float] Tank diameter, [m]**

**Returns**

**Kp [float] Product of Power number and Reynolds number for laminar regime []**
Notes

Example is from example 9-6 in [1]. Confirmed.

References

[1], [2]

Examples

```python
>>> Kp_helical_ribbon_Rieger(D=1.9, h=1.9, nb=2, pitch=1.9, width=.19, T=2)
357.39749163259256
```

```python
fluids.mixing.time_helical_ribbon_Grenville(Kp, N)
```

Calculates product of time required for mixing in a helical ribbon coil in the laminar regime according to the Grenville [2] method recommended in [1].

\[
t = 896 \times 10^3 K_p^{-1.69} / N
\]

Parameters

- **Kp** [float] Product of power number and Reynolds number for laminar regime
- **N** [float] Speed of impeller, [revolutions/s]

Returns

- **t** [float] Time for homogeneity [s]

Notes

Degree of homogeneity is not specified. Example is from example 9-6 in [1]. Confirmed.

References

[1], [2]

Examples

```python
>>> time_helical_ribbon_Grenville(357.4, 4/60.)
650.980654028894
```

```python
fluids.mixing.size_tee(Q1, Q2, D, D2, n=1, pipe_diameters=5)
```

Calculates CoV of an optimal or specified tee for mixing at a tee according to [1]. Assumes turbulent flow. The smaller stream is injected into the main pipe, which continues straight. COV calculation is according to [2].

\[ \text{TODO} \]

Parameters

- **Q1** [float] Volumetric flow rate of larger stream [m^3/s]
- **Q2** [float] Volumetric flow rate of smaller stream [m^3/s]
D  [float] Diameter of pipe after tee [m]
D2  [float] Diameter of mixing inlet, optional (optimally calculated if not specified) [m]
n  [float] Number of jets, 1 to 4 []
pipe_diameters  [float] Number of diameters along tail pipe for CoV calculation, 0 to 5 []

Returns

CoV  [float] Standard deviation of dimensionless concentration [-]

Notes

Not specified if this works for liquid also, though probably not. Example is from example Example 9-6 in [1].
Low precision used in example.

References

[1], [2]

Examples

```python
>>> size_tee(Q1=11.7, Q2=2.74, D=0.762, D2=None, n=1, pipe_diameters=5)
0.2940930233038544
```

fluids.mixing.COV_motionless_mixer(Ki, Q1, Q2, pipe_diameters)
Calculates CoV of a motionless mixer with a regression parameter in [1] and originally in [2].

\[
\frac{CoV}{CoV_0} = K_1 L / D
\]

Parameters

Ki  [float] Correlation parameter specific to a mixer's design, [-]
Q1  [float] Volumetric flow rate of larger stream [m^3/s]
Q2  [float] Volumetric flow rate of smaller stream [m^3/s]
pipe_diameters  [float] Number of diameters along tail pipe for CoV calculation, 0 to 5 []

Returns

CoV  [float] Standard deviation of dimensionless concentration [-]

Notes

Example 7-8.3.2 in [1], solved backwards.

References

[1], [2]
2.13 Hydrology, weirs and open flow (fluids.open_flow)

fluids.open_flow.Q_weir_V_Shen(h1, angle=90)
Calculates the flow rate across a V-notch (triangular) weir from the height of the liquid above the tip of the notch, and with the angle of the notch. Most of these type of weir are 90 degrees. Model from [1] as reproduced in [2].

Flow rate is given by:

\[ Q = C \tan \left( \frac{\theta}{2} \right) \sqrt{g(h_1 + k)^{2.5}} \]

Parameters

- **h1** [float] Height of the fluid above the notch [m]
**angle** [float, optional] Angle of the notch [degrees]

**Returns**

Q [float] Volumetric flow rate across the weir [m^3/s]

**Notes**

angles = [20, 40, 60, 80, 100] Cs = [0.59, 0.58, 0.575, 0.575, 0.58] k = [0.0028, 0.0017, 0.0012, 0.001, 0.001]

The following limits apply to the use of this equation:

$h_1 \geq 0.05 \text{ m } h_2 > 0.45 \text{ m } h_1/h_2 \leq 0.4 \text{ m } b > 0.9 \text{ m}$

\[
\frac{h_1}{b} \tan \left( \frac{\theta}{2} \right) < 2
\]


**References**

[1], [2]

**Examples**

```python
>>> Q_weir_V_Shen(0.6, angle=45)
0.21071725775478228
```

`fluids.open_flow.Q_weir_rectangular_Kindsvater_Carter(h_1, h_2, b)`

Calculates the flow rate across rectangular weir from the height of the liquid above the crest of the notch, the liquid depth beneath it, and the width of the notch. Model from [1] as reproduced in [2].

Flow rate is given by:

\[
Q = 0.554 \left( 1 - 0.0035 \frac{h_1}{h_2} \right) (b + 0.0025) \sqrt{g(h_1 + 0.0001)}^{1.5}
\]

**Parameters**

- **h1** [float] Height of the fluid above the crest of the weir [m]
- **h2** [float] Height of the fluid below the crest of the weir [m]
- **b** [float] Width of the rectangular flow section of the weir [m]

**Returns**

Q [float] Volumetric flow rate across the weir [m^3/s]

**Notes**

The following limits apply to the use of this equation:

\[
b/b_1 \leq 0.2 \text{ } h_1/h_2 < 2 \text{ } b > 0.15 \text{ m } h_1 > 0.03 \text{ m } h_2 > 0.1 \text{ m}
\]

**References**

[1], [2]
Examples

```python
e>>> Q_weir_rectangular_Kindsvater_Carter(0.2, 0.5, 1)
0.15545928949179422
```

```python
fluids.open_flow.Q_weir_rectangular_SIA(h1, h2, b, b1)
```
Calculates the flow rate across rectangular weir from the height of the liquid above the crest of the notch, the liquid depth beneath it, and the width of the notch. Model from [1] as reproduced in [2].

Flow rate is given by:

\[
Q = 0.544 \left[ 1 + 0.064 \left( \frac{b}{b_1} \right)^2 + \frac{0.00026 - 0.00519(b/b_1)^2}{h_1 + 0.0016} \right] \left[ 1 + 0.5 \left( \frac{b}{b_1} \right)^4 \left( \frac{h_1}{h_1 + h_2} \right)^2 \right] b \sqrt{gh_1}^{1.5}
\]

Parameters

- `h1` [float] Height of the fluid above the crest of the weir [m]
- `h2` [float] Height of the fluid below the crest of the weir [m]
- `b` [float] Width of the rectangular flow section of the weir [m]
- `b1` [float] Width of the full section of the channel [m]

Returns

- `Q` [float] Volumetric flow rate across the weir [m^3/s]

Notes

The following limits apply to the use of this equation:

- `b/b1` 0.2
- `h1/h2` < 2
- `b` > 0.15 m
- `h1` > 0.03 m
- `h2` > 0.1 m

References

[1], [2]

Examples

```python
e>>> Q_weir_rectangular_SIA(0.2, 0.5, 1, 2)
1.0408858453811165
```

```python
fluids.open_flow.Q_weir_rectangular_full_Ackers(h1, h2, b)
```
Calculates the flow rate across a full-channel rectangular weir from the height of the liquid above the crest of the weir, the liquid depth beneath it, and the width of the channel. Model from [1] as reproduced in [2], confirmed with [3].

Flow rate is given by:

\[
Q = 0.564 \left( 1 + 0.150 \frac{h_1}{h_2} \right) b \sqrt{gh_1}^{1.5}
\]

Parameters

- `h1` [float] Height of the fluid above the crest of the weir [m]
- `h2` [float] Height of the fluid below the crest of the weir [m]
**b** [float] Width of the channel section [m]

**Returns**

**Q** [float] Volumetric flow rate across the weir [m^3/s]

**Notes**

The following limits apply to the use of this equation:

\[ h_1 > 0.02 \text{ m} \quad h_2 > 0.15 \text{ m} \quad h_1/h_2 < 2.2 \]

**References**

[1], [2], [3]

**Examples**

Example as in [3], matches. However, example is unlikely in practice.

```python
>>> Q_weir_rectangular_full_Ackers(h1=0.9, h2=0.6, b=5)
9.251938159899948
```

Calculates the flow rate across a full-channel rectangular weir from the height of the liquid above the crest of the weir, the liquid depth beneath it, and the width of the channel. Model from [1] as reproduced in [2].

Flow rate is given by:

\[
Q = \frac{2}{3} \sqrt{2} \left( 0.615 + \frac{0.000615}{h_1 + 0.0016} \right) \frac{b \sqrt{gh_1} + 0.5 \left( \frac{h_1}{h_1 + h_2} \right)^2 b \sqrt{gh_1^{1.5}}}{b \sqrt{gh_1^{1.5}}}
\]

**Parameters**

- **h1** [float] Height of the fluid above the crest of the weir [m]
- **h2** [float] Height of the fluid below the crest of the weir [m]
- **b** [float] Width of the channel section [m]

**Returns**

- **Q** [float] Volumetric flow rate across the weir [m^3/s]

**Notes**

The following limits apply to the use of this equation:

\[ 0.025 < h < 0.8 \text{ m} \quad b > 0.3 \text{ m} \quad h_2 > 0.3 \text{ m} \quad h_1/h_2 < 1 \]

**References**

[1], [2]
Example compares terribly with the Ackers expression - probable error in [2]. DO NOT USE.

```python
>>> Q_weir_rectangular_full_SIA(h1=0.3, h2=0.4, b=2)
1.1875825055400384
```

**fluids.open_flow.Q_weir_rectangular_full_Rehbock** (*h1*, *h2*, *b*)
Calculates the flow rate across a full-channel rectangular weir from the height of the liquid above the crest of the weir, the liquid depth beneath it, and the width of the channel. Model from [1] as reproduced in [2].

Flow rate is given by:

\[
Q = \frac{2}{3} \sqrt{2} \left( 0.602 + 0.0832 \frac{h_1}{h_2} \right) b \sqrt{g} (h_1 + 0.00125)^{1.5}
\]

**Parameters**
- *h1* [float] Height of the fluid above the crest of the weir [m]
- *h2* [float] Height of the fluid below the crest of the weir [m]
- *b* [float] Width of the channel section [m]

**Returns**
- *Q* [float] Volumetric flow rate across the weir [m^3/s]

**Notes**
The following limits apply to the use of this equation:
0.03 m < *h1* < 0.75 m, *b* > 0.3 m, *h2* > 0.3 m, *h1*/*h2* < 1

**References**
[1], [2]

**Examples**

```python
>>> Q_weir_rectangular_full_Rehbock(h1=0.3, h2=0.4, b=2)
0.6486856330601333
```

**fluids.open_flow.Q_weir_rectangular_full_Kindsvater_Carter** (*h1*, *h2*, *b*)
Calculates the flow rate across a full-channel rectangular weir from the height of the liquid above the crest of the weir, the liquid depth beneath it, and the width of the channel. Model from [1] as reproduced in [2].

Flow rate is given by:

\[
Q = \frac{2}{3} \sqrt{2} \left( 0.602 + 0.0832 \frac{h_1}{h_2} \right) b \sqrt{g} (h_1 + 0.00125)^{1.5}
\]

**Parameters**
- *h1* [float] Height of the fluid above the crest of the weir [m]
- *h2* [float] Height of the fluid below the crest of the weir [m]
- *b* [float] Width of the channel section [m]
Returns

\( Q \) [float] Volumetric flow rate across the weir [m^3/s]

Notes

The following limits apply to the use of this equation:
\( h_1 > 0.03 \, \text{m} \quad b > 0.15 \, \text{m} \quad h_2 > 0.1 \, \text{m} \quad \frac{h_1}{h_2} < 2 \)

References

[1], [2]

Examples

```python
>>> Q_weir_rectangular_full_Kinshvater_Carter(h1=0.3, h2=0.4, b=2)
0.641560300081563
```

\texttt{fluids.open\_flow.V\_Manning(Rh, S, n)}

Predicts the average velocity of a flow across an open channel of hydraulic radius \( Rh \) and slope \( S \), given the Manning roughness coefficient \( n \).

Flow rate is given by:

\[ V = \frac{1}{n} R_h^{2/3} S^{0.5} \]

Parameters

- \( Rh \) [float] Hydraulic radius of the channel, Flow Area/Wetted perimeter [m]
- \( S \) [float] Slope of the channel, m/m [-]
- \( n \) [float] Manning roughness coefficient [s/m^(1/3)]

Returns

\( V \) [float] Average velocity of the channel [m/s]

Notes

This is equation is often given in imperial units multiplied by 1.49.

References

[1], [2]

Examples

Example is from [2], matches.

```python
>>> V_Manning(0.2859, 0.005236, 0.03)
1.0467781958118971
```
fluids.open_flow.n_Manning_to_C_Chezy(n, Rh)
Converts a Manning roughness coefficient to a Chezy coefficient, given the hydraulic radius of the channel.

\[ C = \frac{1}{n} R_h^{1/6} \]

Parameters
- \( n \) [float] Manning roughness coefficient \([s/m^{1/3}]\)
- \( Rh \) [float] Hydraulic radius of the channel, Flow Area/Wetted perimeter \([m]\)

Returns
- \( C \) [float] Chezy coefficient \([m^{0.5}/s]\)

References
[1]

Examples
Custom example, checked.

```python
>>> n_Manning_to_C_Chezy(0.05, Rh=5)
26.15320972023661
```

fluids.open_flow.C_Chezy_to_n_Manning(C, Rh)
Converts a Chezy coefficient to a Manning roughness coefficient, given the hydraulic radius of the channel.

\[ n = \frac{1}{C} R_h^{1/6} \]

Parameters
- \( C \) [float] Chezy coefficient \([m^{0.5}/s]\)
- \( Rh \) [float] Hydraulic radius of the channel, Flow Area/Wetted perimeter \([m]\)

Returns
- \( n \) [float] Manning roughness coefficient \([s/m^{1/3}]\)

References
[1]

Examples
Custom example, checked.

```python
>>> C_Chezy_to_n_Manning(26.15, Rh=5)
0.05000613713238358
```
fluids.open_flow.V_Chezy(Rh, S, C)

Predicts the average velocity of a flow across an open channel of hydraulic radius Rh and slope S, given the Chezy coefficient C.

Flow rate is given by:

\[ V = C \sqrt{SR_h} \]

Parameters

- **Rh**  [float] Hydraulic radius of the channel, Flow Area/Wetted perimeter [m]
- **S**  [float] Slope of the channel, m/m [-]
- **C**  [float] Chezy coefficient [m^{0.5}/s]

Returns

- **V**  [float] Average velocity of the channel [m/s]

References

[1], [2], [3]

Examples

Custom example, checked.

```python
>>> V_Chezy(Rh=5, S=0.001, C=26.153)
1.8492963648371776
```

2.14 Packed bed pressure drop (fluids.packed_bed)

fluids.packed_bed.dP_packed_bed(dp, voidage, vs, rho, mu, L=1, Dt=None, sphericity=None, Method=None, AvailableMethods=False)

This function handles choosing which pressure drop in a packed bed correlation is used. Automatically select which correlation to use if none is provided. Returns None if insufficient information is provided.

Preferred correlations are ‘Erdim, Akgiray & Demir’ when tube diameter is not provided, and ‘Harrison, Brunner & Hecker’ when tube diameter is provided. If you are using a particles in a narrow tube between 2 and 3 particle diameters, expect higher than normal voidages (0.4-0.5) and used the method ‘Guo, Sun, Zhang, Ding & Liu’.

Parameters

- **dp**  [float] Particle diameter of spheres [m]
- **voidage**  [float] Void fraction of bed packing [-]
- **vs**  [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area) [m/s]
- **rho**  [float] Density of the fluid [kg/m^3]
- **mu**  [float] Viscosity of the fluid, [Pa*s]
- **L**  [float, optional] Length the fluid flows in the packed bed [m]
- **Dt**  [float, optional] Diameter of the tube, [m]
sphericity  [float, optional] Sphericity of the particles [-]

Returns

dP  [float] Pressure drop across the bed [Pa]

methods  [list, only returned if AvailableMethods == True] List of methods which can be used to calculate dP with the given inputs

Other Parameters

Method  [string, optional] A string of the function name to use, as in the dictionary packed_beds_correlations

AvailableMethods  [bool, optional] If True, function will consider which methods which can be used to calculate dP with the given inputs and return them as a list

Examples

```python
>>> dP_packed_bed(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
1438.2826958844414
>>> dP_packed_bed(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3, Dt=0.01)
1255.1625662548427
>>> dP_packed_bed(dp=0.05, voidage=0.492, vs=0.1, rho=1E3, mu=1E-3, Dt=0.015, Method='Guo, Sun, Zhang, Ding & Liu')
18782.499710673364
```

**Fluids Documentation, Release 0.1**

**2.14. Packed bed pressure drop (fluids.packed_bed)**

Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], as shown in [2] and [3]. Eighteenth most accurate correlation overall in the review of [2].

Most often presented in the following form:

\[
\Delta P = \frac{150\mu(1 - \epsilon)^2v_sL}{e^3d_p^2} + \frac{1.75(1 - \epsilon)\rho v_s^2L}{e^3d_p^2}
\]

It is also often presented with a term for sphericity, which is multiplied by particle diameter everywhere in the equation. However, this is highly empirical and better correlations for beds of differently-shaped particles exist. To use sphericity in this model, multiple the input particle diameter by the spericity separately.

In the review of [2], it is expressed in terms of a parameter \( f_p \), shown below. This is a convenient means of expressing all forms of pressure drop in packed beds correlations in a way that allows for easy comparison.

\[
f_p = \left( 150 + 1.75 \frac{Re}{1-\epsilon} \right) \frac{(1 - \epsilon)^2}{e^3Re}
\]

\[
Re = \frac{\rho v_s d_p}{\mu}
\]

\[
f_p = \frac{\Delta P d_p}{\rho v_s^2 L}
\]

**Parameters**

dp  [float] Particle diameter of spheres [m]

voidage  [float] Void fraction of bed packing [-]

vs  [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]

rho  [float] Density of the fluid [kg/m^3]
fluids.packed_bed. Kuo_Nydegger (dp, voidage, vs, rho, mu, L=1)
Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], as shown in [2] and [3]. Thirty-eighth most accurate correlation overall in the review of [2].
\[
 f_p = \left( 276.23 + 5.05 \left( \frac{Re}{1-\epsilon} \right)^{0.87} \right) \frac{(1-\epsilon)^2}{\epsilon^3 Re} \\
 f_p = \frac{\Delta P d_p}{\rho v_s^2 L} \\
 Re = \frac{\rho v_s d_p}{\mu}
\]

Parameters
- dp [float] Particle diameter of spheres [m]
- voidage [float] Void fraction of bed packing [-]
- vs [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- rho [float] Density of the fluid [kg/m^3]
- mu [float] Viscosity of the fluid, [Pa*s]
- L [float, optional] Length the fluid flows in the packed bed [m]

Returns
- dP [float] Pressure drop across the bed [Pa]

Notes
Validity range shown in [2] as for a range of \( 460 < Re < 14600 \), \( 0.3760 < \epsilon < 0.3901 \). Developed with data from rough granular ball propellants beds, with air.
Fluids Documentation, Release 0.1

References

[1], [2], [3]

Examples

```python
>>> Kuo_Nydegger(dp=8E-1, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
0.02565146097368624
```

`fluids.packed_bed.Jones_Krier(dp, voidage, vs, rho, mu, L=1)`

Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], also shown in [2]. Tenth most accurate correlation overall in the review of [2].

\[
f_p = \left( 150 + 3.89 \left( \frac{Re}{1 - \epsilon} \right)^{0.87} (1 - \epsilon)^2 \right) \left( \frac{1 - \epsilon}{e^3 Re} \right)
\]

\[
f_p = \frac{\Delta P d_p}{\rho v_s^2 L}
\]

\[
Re = \frac{\rho v_s d_p}{\mu}
\]

Parameters

- `dp` [float] Particle diameter of spheres [m]
- `voidage` [float] Void fraction of bed packing [-]
- `vs` [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- `rho` [float] Density of the fluid [kg/m^3]
- `mu` [float] Viscosity of the fluid, [Pa*s]
- `L` [float, optional] Length the fluid flows in the packed bed [m]

Returns

- `dP` [float] Pressure drop across the bed [Pa]

Notes

Validity range shown in [1] as for a range of \( 733 < Re < 126,670 \) and \( 0.3804 < \epsilon < 0.4304 \). Developed from data of spherical glass beads.

References

[1], [2]

Examples

```python
>>> Jones_Krier(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
1362.2719449873746
```
fluids.packed_bed.Carman(dp, voidage, vs, rho, mu, L=1)
Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], as shown in [2]. Fifth most accurate correlation overall in the review of [2]. Also shown in [3].

\[
f_p = \left( 180 + 2.871 \left( \frac{Re}{1-\epsilon} \right)^{0.9} \right) \frac{(1-\epsilon)^2}{\epsilon^3 Re}
\]

\[
f_p = \frac{\Delta P d_p}{\rho v_s^2 L}
\]

\[
Re = \frac{\rho v_s d_p}{\mu}
\]

Parameters
- **dp** [float] Particle diameter of spheres [m]
- **voidage** [float] Void fraction of bed packing [-]
- **vs** [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- **rho** [float] Density of the fluid [kg/m^3]
- **mu** [float] Viscosity of the fluid, [Pa*s]
- **L** [float, optional] Length the fluid flows in the packed bed [m]

Returns
- **dP** [float] Pressure drop across the bed [Pa]

Notes
Valid in [1], [2], and [3] for a range of 300 < Re_{ Erg } < 60,000.

References
[1], [2], [3]

Examples

```python
>>> Carman(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
1614.721678121775
```

fluids.packed_bed.Hicks(dp, voidage, vs, rho, mu, L=1)
Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], as shown in [2]. Twenty-third most accurate correlation overall in the review of [2]. Also shown in [3].

\[
f_p = 6.8 \left( 1-\epsilon \right)^{1.2} Re^{0.2}\epsilon^3
\]

\[
f_p = \frac{\Delta P d_p}{\rho v_s^2 L}
\]

\[
Re = \frac{\rho v_s d_p}{\mu}
\]

Parameters
Fluids Documentation, Release 0.1

**dp** [float] Particle diameter of spheres [m]

**voidage** [float] Void fraction of bed packing [-]

**vs** [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]

**rho** [float] Density of the fluid [kg/m^3]

**mu** [float] Viscosity of the fluid, [Pa*s]

**L** [float, optional] Length the fluid flows in the packed bed [m]

**Returns**

**dP** [float] Pressure drop across the bed [Pa]

**Notes**

Valid in [1], [2], and [3] for a range of \(300 < Re_{Erg} < 60,000\).

**References**

[1], [2], [3]

**Examples**

```python
>>> Hicks(dp=0.01, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
3.631703956680737
```

**fluids.packed_bed.Brauer** (\(dp, \text{voidage}, \text{vs}, \text{rho}, \text{mu}, L=1\))

Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], as shown in [2]. Seventh most accurate correlation overall in the review of [2]. Also shown in [3].

\[
f_p = \left(160 + 3\left(\frac{Re}{1-\epsilon}\right)^{0.9}\frac{(1-\epsilon)^2}{\epsilon^3 Re}\right)^{1/2} \epsilon
\]

\[
f_p = \frac{\Delta P d_p}{\rho v_s L}
\]

\[
Re = \frac{\rho v_s d_p}{\mu}
\]

**Parameters**

**dp** [float] Particle diameter of spheres [m]

**voidage** [float] Void fraction of bed packing [-]

**vs** [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]

**rho** [float] Density of the fluid [kg/m^3]

**mu** [float] Viscosity of the fluid, [Pa*s]

**L** [float, optional] Length the fluid flows in the packed bed [m]

**Returns**

**dP** [float] Pressure drop across the bed [Pa]
Notes

Original has not been reviewed. In [2], is stated as for a range of $2 < Re_{Erg} < 20,000$. In [3], is stated as for a range of $0.01 < Re_{Erg} < 40,000$.

References

[1], [2], [2]

Examples

```python
>>> Brauer(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
1441.5479196020563
```

Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], as shown in [2].

Third most accurate correlation overall in the review of [2].

$$f_p = \left(160 + 3 \left( \frac{Re}{1-\epsilon} \right)^0.9 \right) \left(1 - \epsilon \right)^2$$

$$R_e = \frac{\rho v_s d_p}{\mu}$$

Parameters

- `dp` [float] Particle diameter of spheres [m]
- `voidage` [float] Void fraction of bed packing [-]
- `vs` [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- `rho` [float] Density of the fluid [kg/m^3]
- `mu` [float] Viscosity of the fluid, [Pa*s]
- `L` [float, optional] Length the fluid flows in the packed bed [m]

Returns

- `dP` [float] Pressure drop across the bed [Pa]

Notes

Developed for gas flow through pebbles in nuclear reactors.

In [2], stated as for a range of $1 < RE_{Erg} < 100,000$. In [1], a limit on porosity is stated as $0.36 < \epsilon < 0.42$.

References

[1], [2]
Examples

```python
>>> KTA(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
1440.409277034248
```

`fluids.packed_bed.Erdim_Akgiray_Demir(dp, voidage, vs, rho, mu, L=1)`

Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], claiming to be the best model to date.

\[
f_v = 160 + 2.81Re_{Erg}^{0.904}
\]

\[
f_v = \frac{\Delta P d_p^2}{\mu v_s L} \left(1 - \epsilon\right)^2
\]

\[
Re_{Erg} = \frac{\rho v_s d_p}{\mu(1 - \epsilon)}
\]

**Parameters**

- **dp** [float] Particle diameter of spheres [m]
- **voidage** [float] Void fraction of bed packing [-]
- **vs** [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- **rho** [float] Density of the fluid [kg/m^3]
- **mu** [float] Viscosity of the fluid, [Pa*s]
- **L** [float, optional] Length the fluid flows in the packed bed [m]

**Returns**

- **dP** [float] Pressure drop across the bed [Pa]

**Notes**

Developed with data in the range of:

\[
2 < Re_{Erg} < 3582
\]

\[
4 < d_t/d_p < 34.1
\]

\[
0.377 < \epsilon < 0.470
\]

**References**

[1]

**Examples**

```python
>>> Erdim_Akgiray_Demir(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
1438.2826958844414
```

`fluids.packed_bed.Fahien_Schriver(dp, voidage, vs, rho, mu, L=1)`

Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], as shown in [2]. Second most accurate correlation overall in the review of [2].

\[
f_p = \left(q \frac{f_{1L}}{Re_{Erg}} + (1 - q) \left( f_2 + \frac{f_{1T}}{Re_{Erg}} \right) \right) \frac{1 - \epsilon}{\epsilon^3}
\]
\[ q = \exp \left( -\frac{\epsilon^2 (1 - \epsilon)}{12.6} Re_{Erg} \right) \]

\[ f_{1L} = \frac{136}{(1 - \epsilon)^{0.38}} \]

\[ f_{1T} = \frac{29}{(1 - \epsilon)^{1.45} \epsilon^2} \]

\[ f_2 = \frac{1.87 \epsilon^{0.75}}{(1 - \epsilon)^{0.26}} \]

\[ f_p = \frac{\Delta P d_p}{\rho \epsilon^2 L} \]

\[ Re_{Erg} = \frac{\rho \epsilon v d_p}{\mu (1 - \epsilon)} \]

**Parameters**

- **dp** [float] Particle diameter of spheres [m]
- **voidage** [float] Void fraction of bed packing [-]
- **vs** [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- **rho** [float] Density of the fluid [kg/m^3]
- **mu** [float] Viscosity of the fluid, [Pa*s]
- **L** [float, optional] Length the fluid flows in the packed bed [m]

**Returns**

- **dP** [float] Pressure drop across the bed [Pa]

**Notes**

No range of validity available.

**References**

[1], [2]

**Examples**

```python
>>> Fahien_Schriver(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
1470.6175541844711
```

**fluids.packed_bed.Tallmadge** *(dp, voidage, vs, rho, mu, L=1)*

Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], as shown in [2] and [3].

\[ f_p = \left( 150 + 4.2 \left( \frac{Re}{1 - \epsilon} \right)^{5/6} \right) \frac{(1 - \epsilon)^2}{c^3 Re} \]

\[ f_p = \frac{\Delta P d_p}{\rho \epsilon^2 L} \]

\[ Re = \frac{\rho \epsilon v d_p}{\mu} \]
Parameters

- **dp**  [float] Particle diameter of spheres [m]
- **voidage**  [float] Void fraction of bed packing [-]
- **vs**  [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- **rho**  [float] Density of the fluid [kg/m^3]
- **mu**  [float] Viscosity of the fluid, [Pa*s]
- **L**  [float, optional] Length the fluid flows in the packed bed [m]

Returns

- **dP**  [float] Pressure drop across the bed [Pa]

Notes

The validity range shown in [2] is a range of 0.1 < Re < 100000.

References

[1], [2], [3]

Examples

```python
>>> Tallmadge(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
1365.2739144209424
```

`fluids.packed_bed.Idelchik(dp, voidage, vs, rho, mu, L=1)`

Calculates pressure drop across a packed bed of spheres as in [2], originally in [1].

\[
\frac{\Delta P}{L \rho v_s^2} d_p = 0.765 \frac{30}{Re_l} + \frac{3}{Re_l^{0.7}} + 0.3 \\
Re_l = (0.45/\epsilon^{0.5}) Re_{Erg} \\
Re_{Erg} = \frac{\rho v_s D_p}{\mu (1-\epsilon)}
\]

Parameters

- **dp**  [float] Particle diameter of spheres [m]
- **voidage**  [float] Void fraction of bed packing [-]
- **vs**  [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- **rho**  [float] Density of the fluid [kg/m^3]
- **mu**  [float] Viscosity of the fluid, [Pa*s]
- **L**  [float, optional] Length the fluid flows in the packed bed [m]

Returns

- **dP**  [float] Pressure drop across the bed [Pa]
Notes

\[ 0.001 < Re_{Erg} < 1000 \] This equation is valid for void fractions between 0.3 and 0.8. Cited as by Bernshtein.

References

[1], [2]

Examples

```
>>> Idelchik(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3)
1571.909125999067
```

`fluids.packed_bed.Harrison_Brunner_Hecker(dp, voidage, vs, rho, mu, L=L, Dt=None)`
Calculates pressure drop across a packed bed of spheres using a correlation developed in [1], also shown in [2]. Fourth most accurate correlation overall in the review of [2]. Applies a wall correction if diameter of tube is provided.

\[
f_p = \left( 119.8A + 4.63B \left( \frac{Re}{1-\epsilon} \right)^{5/6} \right) \frac{(1-\epsilon)^2}{\epsilon^5 Re}
\]

\[
A = \left( 1 + \frac{d_p}{6(1-\epsilon)D_t} \right)^2
\]

\[
B = 1 - \frac{\pi^2 d_p}{24D_t} \left( 1 - \frac{0.5d_p}{D_t} \right)
\]

\[
f_p = \frac{\Delta P d_p}{\rho v_s^2 L}
\]

\[
Re = \frac{\rho v_s d_p}{\mu}
\]

Parameters

- dp [float] Particle diameter of spheres [m]
- voidage [float] Void fraction of bed packing [-]
- vs [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- rho [float] Density of the fluid [kg/m^3]
- mu [float] Viscosity of the fluid, [Pa*s]
- L [float, optional] Length the fluid flows in the packed bed [m]
- Dt [float, optional] Diameter of the tube, [m]

Returns

- dP [float] Pressure drop across the bed [Pa]

Notes

Uses data from other sources only. Correlation will underestimate pressure drop if tube diameter is not provided. Limits are specified in [1] as:

\[
304 \text{ Chapter 2. API Reference}
\]
\[ 0.72 < \text{Re} < 7700 \]
\[ 8.3 < \frac{d_t}{d_p} < 50 \]
\[ 0.33 < \epsilon < 0.88 \]

References

[1], [2]

Examples

```python
>>> Harrison_Brunner_Hecker(dp=8E-4, voidage=0.4, vs=1E-3, rho=1E3, mu=1E-3, L=1, Dt=1E-2)
1255.1625662548427
```

fluids.packed_bed.\texttt{Montillet\_Akkari\_Comiti}(dp, voidage, vs, rho, mu, L=1, Dt=None)
Calculates pressure drop across a packed bed of spheres as in [2], originally in [1]. Wall effect adjustment is used if \( Dt \) is provided.

\[
\frac{\Delta P}{L \rho V_s^2} \frac{\epsilon^3}{(1 - \epsilon)} = a \left( \frac{D_c}{D_p} \right)^{0.20} \left( \frac{1000}{Re_p} + \frac{60}{Re_p^{0.5}} + 12 \right)
\]

\[
Re_p = \frac{\rho V_s D_p}{\mu}
\]

Parameters

- \texttt{dp} [float] Particle diameter of spheres [m]
- \texttt{voidage} [float] Void fraction of bed packing [-]
- \texttt{vs} [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- \texttt{rho} [float] Density of the fluid [kg/m³]
- \texttt{mu} [float] Viscosity of the fluid, [Pa*s]
- \texttt{L} [float, optional] Length the fluid flows in the packed bed [m]
- \texttt{Dt} [float, optional] Diameter of the tube, [m]

Returns

- \texttt{dP} [float] Pressure drop across bed [Pa]

Notes

10 < \( Re_p < 2500 \) if \( D_c/D > 50 \), set to 2.2. \( a = 0.061 \) for \( \epsilon < 0.4 \), \( 0.050 \) for \( \epsilon > 0.4 \).

References

[1], [2]
Examples

Custom example:

```python
>>> Montillet_Akkari_Comiti(dp=0.0008, voidage=0.4, L=0.5, vs=0.00132629120,
rho=1000., mu=1.00E-003)
1148.1905244077548
```

`fluids.packed_bed` *Guo_Sun*  
(dp, voidage, vs, rho, mu, Dt, L=1)

Calculates pressure drop across a packed bed of spheres using a correlation developed in [1]. This is valid for highly-packed particles at particle/tube diameter ratios between 2 and 3, where a ring packing structure occurs. If a packing ratio is so low, it is important to use this model because in some cases its predictions are as low as half those of other models!

\[
f_v = 180 + \left( 9.5374 \frac{d_p}{D_t} - 2.8054 \right) Re^{0.97}_{Erg}
\]

\[
f_v = \frac{\Delta P d_p^2}{\mu v_s L (1-\epsilon)^2}
\]

\[
Re_{Erg} = \frac{\rho v_s d_p}{\mu (1-\epsilon)}
\]

Parameters

- `dp` [float] Particle diameter of spheres [m]
- `voidage` [float] Void fraction of bed packing [-]
- `vs` [float] Superficial velocity of the fluid (volumetric flow rate/cross-sectional area)[m/s]
- `rho` [float] Density of the fluid [kg/m^3]
- `mu` [float] Viscosity of the fluid, [Pa*s]
- `Dt` [float] Diameter of the tube, [m]
- `L` [float, optional] Length the fluid flows in the packed bed [m]

Returns

- `dP` [float] Pressure drop across the bed [Pa]

Notes

Developed with data in the range of:

\[
100 < Re_{m} < 33000 \\
2 < d_t/d_p < 31 \\
0.476 < \epsilon < 0.492
\]

References

[1]
Guo_Sun(dp=14.2E-3, voidage=0.492, vs=0.6, rho=1E3, mu=1E-3, Dt=40.9E-3)
42019.529911473706

fluids.packed_bed.voidage_Benyahia_Oneil(Dpe, Dt, sphericity)
Calculates voidage of a bed of arbitrarily shaped uniform particles packed into a bed or tube of diameter Dt, with equivalent sphere diameter Dp. Shown in [1], and cited by various authors. Correlations exist also for spheres, solid cylinders, hollow cylinders, and 4-hole cylinders. Based on a series of physical measurements.

\[ \epsilon = 0.1504 + \frac{0.2024}{\phi} + \frac{1.0814}{(\frac{d_t}{d_{pe}} + 0.1226)^2} \]

Parameters
- **Dpe** [float] Equivalent spherical particle diameter (diameter of a sphere with the same volume), [m]
- **Dt** [float] Diameter of the tube, [m]
- **sphericity** [float] Sphericity of particles in bed []

Returns
- **voidage** [float] Void fraction of bed packing []

Notes
Average error of 5.2%; valid 1.5 < dtube/dp < 50 and 0.42 < sphericity < 1

References
[1]

Examples

```python
>>> voidage_Benyahia_Oneil(Dpe=1E-3, Dt=1E-2, sphericity=.8)
0.41395363849210065
```

fluids.packed_bed.voidage_Benyahia_Oneil_spherical(Dp, Dt)
Calculates voidage of a bed of spheres packed into a bed or tube of diameter Dt, with sphere diameters Dp. Shown in [1], and cited by various authors. Correlations exist also for solid cylinders, hollow cylinders, and 4-hole cylinders. Based on a series of physical measurements.

\[ \epsilon = 0.390 + 1.740 \left( \frac{d_{cyl}}{d_p} + 1.140 \right)^2 \]

Parameters
- **Dp** [float] Spherical particle diameter, [m]
- **Dt** [float] Diameter of the tube, [m]

Returns
- **voidage** [float] Void fraction of bed packing []
Notes
Average error 1.5%, 1.5 < ratio < 50.

References
[1]

Examples
```python
>>> voidage_Benyahia_Oneil_spherical(Dp=.001, Dt=.05)
0.3906653157443224
```

`fluids.packed_bed.voidage_Benyahia_Oneil_cylindrical(Dpe, Dt, sphericity)`
Calculates voidage of a bed of cylindrical uniform particles packed into a bed or tube of diameter `Dt`, with equivalent sphere diameter `Dpe`. Shown in [1], and cited by various authors. Correlations exist also for spheres, solid cylinders, hollow cylinders, and 4-hole cylinders. Based on a series of physical measurements.

\[ \epsilon = 0.373 + \frac{1.703}{(d_{\text{cy}l}/d_{\text{pe}} + 0.611)^2} \]

Parameters
- `Dpe` [float] Equivalent spherical particle diameter (diameter of a sphere with the same volume), [m]
- `Dt` [float] Diameter of the tube, [m]
- `sphericity` [float] Sphericity of particles in bed []

Returns
- `voidage` [float] Void fraction of bed packing []

Notes
Average error 1.6%; 1.7 < ratio < 26.3.

References
[1]

Examples
```python
>>> voidage_Benyahia_Oneil_cylindrical(Dpe=.01, Dt=.1, sphericity=.6)
0.38812523109607894
```
2.15 Packing & demister pressure drop (fluids.packed_tower)

This module contains correlations and functions for calculating pressure drop from packings and demisters; separation efficiency of demisters; demister pressure drop; and demister geometry.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker or contact the author at Caleb.Andrew.Bell@gmail.com.

- Packing Pressure Drop
- Packing Flooding
- Demister Pressure Drop
- Demister Separation Efficiency
- Demister Geometry

2.15.1 Packing Pressure Drop

The Robbins equation in the `fluids.packed_tower` module calculates pressure drop across a packed column.

**Equation:**

\[ \Delta P = C_3 G_f^2 10^{C_4 L_f} + 0.4 \frac{L_f}{\mu_0} 0.1 \frac{G_f}{L_f} \]

- \( C_3 \) and \( C_4 \) are tabulated coefficients for different packing geometries.
- \( G_f \) is the specific gas mass flow rate.
- \( L_f \) is the height of the packing.
- \( \mu_0 \) is the viscosity of the liquid.
- \( L \) is the specific liquid mass flow rate.
- \( G \) is the specific gas mass flow rate.
- \( \rho_0 \) is the density of the liquid.
- \( \rho_g \) is the density of the gas.
- \( \mu \) is the viscosity of the liquid.
- \( H \) is the height of the packing.
- \( F_{pd} \) is the Robbins packing factor.

**Parameters**

- \( L \) [float] Specific liquid mass flow rate [kg/s/m^2]
- \( G \) [float] Specific gas mass flow rate [kg/s/m^2]
- \( \rho_0 \) [float] Density of liquid [kg/m^3]
- \( \rho_g \) [float] Density of gas [kg/m^3]
- \( \mu \) [float] Viscosity of liquid [Pa*s]
- \( H \) [float] Height of packing [m]
- \( F_{pd} \) [float] Robbins packing factor (tabulated for packings) [1/ft]

**Returns**

- \( \Delta P \) [float] Pressure drop across packing [Pa]

**Notes**

Perry’s displayed equation has a typo in a superscript. This model is based on the example in Perry’s.
References

[1]

Examples

```python
>>> Robbins(L=12.2, G=2.03, rhol=1000., rhog=1.1853, mul=0.001, H=2.0, Fpd=24.0)
619.6624593438102
```

`fluids.packed_tower.Stichlmair_dry(Vg, rhog, mug, voidage, specific_area, C1, C2, C3, H=1.0)`

Calculates dry pressure drop across a packed column, using the Stichlmair [1] correlation. Uses three regressed constants for each type of packing, and voidage and specific area.

Pressure drop is given by:

\[
\Delta P_{dry} = \frac{3}{4} f_0 \frac{1 - \epsilon}{\epsilon^{1.55}} \rho G H d_p V_g^2
\]

\[
f_0 = C_1 \frac{R_e_g}{R_e_g} + C_2 \frac{R_e_g^{0.5}}{R_e_g^{0.5}} + C_3
\]

\[
d_p = \frac{6(1 - \epsilon)}{a}
\]

Parameters

- `Vg` [float] Superficial velocity of gas, Q/A [m/s]
- `rhog` [float] Density of gas [kg/m^3]
- `mug` [float] Viscosity of gas [Pa*s]
- `voidage` [float] Voidage of bed of packing material []
- `specific_area` [float] Specific area of the packing material [m^2/m^3]
- `C1` [float] Packing-specific constant []
- `C2` [float] Packing-specific constant []
- `C3` [float] Packing-specific constant []
- `H` [float, optional] Height of packing [m]

Returns

- `dP_dry` [float] Pressure drop across dry packing [Pa]

Notes

This model is used by most process simulation tools. If H is not provided, it defaults to 1. If Z is not provided, it defaults to 1.

References

[1]
Examples

```python
>>> Stichlmair_dry(Vg=0.4, rhog=5., mug=5E-5, voidage=0.68,
... specific_area=260., C1=32., C2=7, C3=1)
236.80904286559885
```

`fluids.packed_tower.Stichlmair_wet(Vg, Vl, rhog, rhol, mug, voidage, specific_area, C1, C2, C3, H=1)`

Calculates dry pressure drop across a packed column, using the Stichlmair [1] correlation. Uses three regressed constants for each type of packing, and voidage and specific area. This model is for irrigated columns only.

Pressure drop is given by:

\[
\frac{\Delta P_{irr}}{H} = \frac{\Delta P_{dry}}{H} \left(1 - \epsilon + h_T\right) \left(\frac{1}{1 - \epsilon}\right)^{(2+\epsilon)/3} \left(\frac{\epsilon}{\epsilon - h_T}\right)^{4.65}
\]

\[
h_T = h_0 \left[1 + 20 \left(\frac{\Delta P_{irr}}{H \rho_L g}\right)^2\right]
\]

\[
F_{rL} = \frac{V_{L}^2 g}{\rho_L a^{4.65}}
\]

\[
h_0 = 0.555 F_{rL}^{1/3}
\]

\[
c = \frac{-C_1/Re_g - C_2/(2Re_g^{0.5})}{f_0}
\]

\[
\Delta P_{dry} = \frac{3}{4} f_0 \frac{1 - \epsilon}{\epsilon^{4.65} \rho_g} H \frac{d_p V_g^2}{d_p}
\]

\[
f_0 = \frac{C_1}{Re_g} + \frac{C_2}{Re_g^{0.5}} + C_3
\]

\[
d_p = \frac{6(1 - \epsilon)}{a}
\]

Parameters

- **Vg** [float] Superficial velocity of gas, Q/A [m/s]
- **Vl** [float] Superficial velocity of liquid, Q/A [m/s]
- **rhog** [float] Density of gas [kg/m^3]
- **rhol** [float] Density of liquid [kg/m^3]
- **mug** [float] Viscosity of gas [Pa*s]
- **voidage** [float] Voidage of bed of packing material []
- **specific_area** [float] Specific area of the packing material [m^2/m^3]
- **C1** [float] Packing-specific constant []
- **C2** [float] Packing-specific constant []
- **C3** [float] Packing-specific constant []
- **H** [float, optional] Height of packing [m]

Returns

- **dP** [float] Pressure drop across irrigated packing [Pa]
Notes

This model is used by most process simulation tools. If H is not provided, it defaults to 1. If Z is not provided, it defaults to 1. A numerical solver is used and needed by this model. Its initial guess is the dry pressure drop. Convergence problems may occur. The model as described in [1] appears to have a typo, and could not match the example. As described in [2], however, the model works.

References

[1], [2]

Examples

Example is from [1], matches.

```python
>>> Stichlmair_wet(Vg=0.4, Vl = 5E-3, rhog=5., rhol=1200., mug=5E-5, ...
... voidage=0.68, specific_area=260., C1=32., C2=7., C3=1.)
539.8768237253518
```

2.15.2 Packing Flooding

`fluids.packed_tower.Stichlmair_flood(Vl, rhog, rhol, mug, voidage, specific_area, C1, C2, C3, H=1.0)`

Calculates gas rate for flooding of a packed column, using the Stichlmair [1] correlation. Uses three regressed constants for each type of packing, and voidage and specific area.

Pressure drop is given by:

\[
\frac{\Delta P_{irr}}{H} = \frac{\Delta P_{dry}}{H} \left( \frac{1 - \epsilon + h_T}{1 - \epsilon} \right)^{(2+c)/3} \left( \frac{\epsilon}{\epsilon - h_T} \right)^{4.65}
\]

\[
h_T = h_0 \left[ 1 + 20 \left( \frac{\Delta P_{irr}}{H \rho_L g} \right)^2 \right]
\]

\[
F_{rL} = \frac{V_L^2 a}{g \epsilon^{4.65}}
\]

\[
h_0 = 0.555 F_{rL}^{1/3}
\]

\[
c = \frac{-C_1/Re_g - C_2/(2Re_g^{0.5})}{f_0}
\]

\[
\Delta P_{dry} = 3 \frac{f_0}{4} \frac{1 - \epsilon}{\epsilon^{4.65} \rho_L g} \frac{H}{d_p} V_g^2
\]

\[
f_0 = \frac{C_1}{Re_g} + \frac{C_2}{Re_g^{0.5}} + C_3
\]

\[
d_p = \frac{6(1 - \epsilon)}{a}
\]

Parameters

- **Vl** [float] Superficial velocity of liquid, Q/A [m/s]
- **rhog** [float] Density of gas [kg/m^3]
rhol [float]  Density of liquid [kg/m^3]
mug [float]  Viscosity of gas [Pa*s]
voidage [float] Voidage of bed of packing material [

specific_area [float]  Specific area of the packing material [m^2/m^3]
C1 [float]  Packing-specific constant []
C2 [float]  Packing-specific constant []
C3 [float]  Packing-specific constant []
H [float, optional]  Height of packing [m]

Returns
Vg [float]  Superficial velocity of gas, Q/A [m/s]

Notes
A numerical solver is used to solve this model.

References
[1]

Examples
Example is from [1], matches.

```python
>>> Stichlmair_flood(Vl = 5E-3, rhog=5., rhol=1200., mug=5E-5, ...
... voidage=0.68, specific_area=260., C1=32., C2=7., C3=1.)
0.6394323542746928
```

2.15.3 Demister Pressure Drop

fluids.packed_tower.dp_demister_dry_Setekleiv_Svendsen(S, voidage, vs, rho, mu, L=L)
Calculates dry pressure drop across a demister, using the correlation in [1]. This model is for dry demisters with no holdup only.

\[
\frac{\Delta P c^2}{\rho f v^2} = 10.29 - \frac{565}{69.6SL - (SL)^2 - 779} - \frac{74.9}{160.9 - 4.85SL} + 45.33 \left( \frac{\mu f c S^2 L}{\rho f v} \right)^{0.75}
\]

Parameters
S [float]  Specific area of the demister, normally ~250-1000 [m^2/m^3]
voidage [float] Voidage of bed of the demister material, normally ~0.98 []
vs [float]  Superficial velocity of fluid, Q/A [m/s]
rho [float]  Density of fluid [kg/m^3]
mu [float]  Viscosity of fluid [Pa*s]
L [float, optional]  Length of the demister [m]
Returns

\[ dP \] [float] Pressure drop across a dry demister [Pa]

Notes

Useful at startup and in modeling. Dry pressure drop is normally negligible compared to wet pressure drop. Coefficients obtained by evolutionary programming and may not fit data outside of the limits of the variables.

References

[1]

Examples

```python
>>> dP_demister_dry_Setekleiv_Svendsen(S=250, voidage=.983, vs=1.2, rho=10, mu=3E-5, L=1)
320.3280788941329
```

Calculates dry pressure drop across a demister, using the correlation in [1]. This model is for dry demisters with no holdup only. Developed with literature data included as well as their own experimental data.

\[
\frac{\Delta P c^2}{\rho_f v^2} = 7.3 - \frac{320}{69.6SL - (SL)^2 - 779} - \frac{52.4}{161 - 4.85SL} + 27.2 \left( \frac{\mu_f c S^2 L}{\rho_f v} \right)^{0.75}
\]

Parameters

S [float] Specific area of the demister, normally ~250-1000 [m^2/m^3]
voidage [float] Voidage of bed of the demister material, normally ~0.98 []
vs [float] Superficial velocity of fluid, Q/A [m/s]
rho [float] Density of fluid [kg/m^3]
mu [float] Viscosity of fluid [Pa*s]
L [float, optional] Length of the demister [m]

Returns

\[ dP \] [float] Pressure drop across a dry demister [Pa]

Notes

Useful at startup and in modeling. Dry pressure drop is normally negligible compared to wet pressure drop. Coefficients obtained by evolutionary programming and may not fit data outside of the limits of the variables.

References

[1]
Calculated wet pressure drop across a demister, using the correlation in [1]. Uses only their own experimental data.

\[
\Delta P = 0.002357(1 - \epsilon)^{0.375798}(V)^{0.81317}(d_w)^{-1.56114147}
\]

**Parameters**

- **vs** [float] Superficial velocity of fluid, Q/A [m/s]
- **voidage** [float] Voidage of bed of the demister material, normally ~0.98
- **d_wire** [float] Diameter of mesh wire, [m]
- **L** [float, optional] Length of the demister [m]

**Returns**

- **dP** [float] Pressure drop across a dry demister [Pa]

**Notes**

No dependency on the liquid properties is included here. Because of the exponential nature of the correlation, the limiting pressure drop as V is lowered is 0 Pa. A dry pressure drop correlation should be compared with results from this at low velocities, and the larger of the two pressure drops used.

The correlation in [1] was presented as follows, with wire diameter in units of mm, density in kg/m^3, V in m/s, and dP in Pa/m.

\[
\Delta P = 3.88178(\rho_{mesh})^{0.375798}(V)^{0.81317}(d_w)^{-1.56114147}
\]

Here, the correlation is converted to base SI units and to use voidage; not all demisters are stainless steel as in [1]. A density of 7999 kg/m^3 was used in the conversion.

In [1], V ranged from 0.98-7.5 m/s, rho from 80.317-208.16 kg/m^3, depth from 100 to 200 mm, wire diameter of 0.2mm to 0.32 mm, and particle diameter from 1 to 5 mm.

**References**

[1]
2.15.4 Demister Separation Efficiency

`fluids.packed_tower.separation_demister_ElDessouky(vs, voidage, d_wire, d_drop)`

Calculates droplet removal by a demister as a fraction from 0 to 1, using the correlation in [1]. Uses only their own experimental data.

\[ \eta = 0.85835(d_w)^{-0.28264}(1 - \epsilon)^{0.099625}(V)^{0.106878}(d_p)^{0.383197} \]

**Parameters**

- **vs** [float] Superficial velocity of fluid, Q/A [m/s]
- **voidage** [float] Voidage of bed of the demister material, normally ~0.98 []
- **d_wire** [float] Diameter of mesh wire, [m]
- **d_drop** [float] Drop diameter, [m]

**Returns**

- **eta** [float] Fraction droplets removed by mass [-]

**Notes**

No dependency on the liquid properties is included here. Because of the exponential nature of the correlation, for smaller diameters separation quickly lowers. This correlation can predict a separation larger than 1 for higher velocities, lower voidages, lower wire diameters, and large droplet sizes. This function truncates these larger values to 1.

The correlation in [1] was presented as follows, with wire diameter in units of mm, density in kg/m^3, V in m/s, separation in %, and particle diameter in mm.

\[ \eta = 17.5047(d_w)^{-0.28264}(\rho_{mesh})^{0.099625}(V)^{0.106878}(d_p)^{0.383197} \]

Here, the correlation is converted to base SI units and to use voidage; not all demisters are stainless steel as in [1]. A density of 7999 kg/m^3 was used in the conversion.

In [1], V ranged from 0.98-7.5 m/s, rho from 80.317-208.16 kg/m^3, depth from 100 to 200 mm, wire diameter of 0.2 mm to 0.32 mm, and particle diameter from 1 to 5 mm.

**References**

[1]

**Examples**

```python
>>> separation_demister_ElDessouky(1.35, 0.974, 0.0002, 0.005)
0.8982892997640582
```

2.15.5 Demister Geometry

`fluids.packed_tower.voidage_experimental(m, rho, D, H)`

Calculates voidage of a bed or mesh given an experimental weight and fixed density, diameter, and height, as shown in [1]. The formula is also self-evident.

\[ \epsilon = 1 - \frac{\rho_{mesh} m_{mesh}}{\rho_{material} \pi D^2 H} \]
Parameters

- \( m \) [float] Mass of mesh or bed particles weighted, [kg]
- \( \rho \) [float] Density of solid particles or mesh [kg/m^3]
- \( D \) [float] Diameter of the cylindrical bed [m]
- \( H \) [float] Height of the demister or bed [m]

Returns

- \( \text{voidage} \) [float] Voidage of bed of the material []

Notes

Should be trusted over manufacturer data.

References

[1]

Examples

```python
>>> voidage_experimental(m=126, rho=8000, D=1, H=1)
0.979464771704212
```

\[ S = \frac{4(1 - \epsilon)}{d_{wire}} \]

Parameters

- \( \text{voidage} \) [float] Voidage of the mesh []
- \( d \) [float] Diameter of the wires making the mesh, [m]

Returns

- \( S \) [float] Specific area of the mesh [m^2/m^3]

Notes

Should be preferred over manufacturer data. Can also be used to show that manufacturer’s data is inconsistent with their claimed voidage and wire diameter.

References

[1]
Examples

```python
>>> specific_area_mesh(voidage=.934, d=3e-4)
879.9999999999994
```

2.16 Particle Size Distributions (fluids.particle_size_distribution)

This module contains particle distribution characterization, fitting, interpolating, and manipulation functions. It may be used with discrete particle size distributions, or with statistical ones with parameters specified.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker or contact the author at Caleb.Andrew.Bell@gmail.com.

- Particle Size Distribution Base Class
- Discrete Particle Size Distributions
- Statistical Particle Size Distributions
- Helper functions: Lognormal Distribution
- Helper functions: Gates Gaudin Schuhman Distribution
- Helper functions: Rosin Rammler Distribution
- Sieves
- Point Spacing

2.16.1 Particle Size Distribution Base Class

class fluids.particle_size_distribution.ParticleSizeDistributionContinuous
Bases: object

Base class representing a continuous particle size distribution specified by a mathematical/statistical function. This class holds the common methods only.

Notes

Although the stated units of input are in meters, this class is actually independent of the units provided; all results will be consistent with the provided unit.

References

[R6899669cf6c8-1]

Examples

Example problem from [R6899669cf6c8-1].
```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6)
```

Attributes

**vssa** The volume-specific surface area of a particle size distribution.

Methods

- `cdf(self, d[, n])` Computes the cumulative distribution density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.
- `cdf_discrete(self, ds[, n])` Computes the cumulative distribution functions for a list of specified particle diameters.
- `delta_cdf(self, d_min, d_max[, n])` Computes the difference in cumulative distribution function between two particle size diameters.
- `dn(self, fraction[, n])` Computes the diameter at which a specified fraction of the distribution falls under.
- `ds_discrete(self[, d_min, d_max, pts, ...])` Create a particle spacing mesh to perform calculations with, according to one of several ways.
- `fractions_discrete(self, ds[, n])` Computes the fractions of the cumulative distribution functions which lie between the specified specified particle diameters.
- `mean_size(self, p, q)` Calculates the mean particle size according to moment-ratio notation.
- `mean_size_ISO(self, k, r)` Calculates the mean particle size according to moment notation (ISO).
- `pdf(self, d[, n])` Computes the probability density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.
- `plot_cdf(self[, n, d_min, d_max, pts, method])` Plot the cumulative distribution function of the particle size distribution.
- `plot_pdf(self[, n, d_min, d_max, pts, ...])` Plot the probability density function of the particle size distribution.

**cdf**(self, d, n=None)
Computes the cumulative distribution density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis. The evaluation function varies with the distribution chosen.

\[
Q_n(d) = \int_0^d q_n(d)
\]

Parameters

- `d` [float] Particle size diameter, [m]
- `n` [int, optional] None (for the order specified when the distribution was created), 0 (number), 1 (length), 2 (area), 3 (volume/mass), or any integer, [-]

Returns

- `cdf` [float] The cumulative distribution function at the specified diameter and order, [-]

2.16. Particle Size Distributions (fluids.particle_size_distribution)
Notes

Analytical integrals can be found for most distributions even when order conversions are necessary.

Examples

```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6, order=3)
>>> for n in (0, 1, 2, 3):
...    print(psd.cdf(5e-6, n))
0.933192798731
0.841344746069
0.691462461274
0.5
```

cdf_discrete (self, ds, n=None)
Computes the cumulative distribution functions for a list of specified particle diameters.

Parameters

ds [list[float]] Particle size diameters, [m]
n [int, optional] None (for the order specified when the distribution was created), 0 (number), 1 (length), 2 (area), 3 (volume/mass), or any integer, [-]

Returns

`cdfs [float] The cumulative distribution functions at the specified diameters and order, [-]`

Examples

```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6, order=3)
>>> psd.cdf_discrete([1e-6, 1e-5, 1e-4, 1e-3])
[0.0006434710129138987, 0.9171714809983015, 0.9999999989602018, 1.0]
```

delta_cdf (self, d_min, d_max, n=None)
Computes the difference in cumulative distribution function between two particle size diameters.

\[ \Delta Q_n = Q_n(d_{\text{max}}) - Q_n(d_{\text{min}}) \]

Parameters

d_min [float] Lower particle size diameter, [m]
d_max [float] Upper particle size diameter, [m]
n [int, optional] None (for the order specified when the distribution was created), 0 (number), 1 (length), 2 (area), 3 (volume/mass), or any integer, [-]

Returns

delta_cdf [float] The difference in the cumulative distribution function for the two diameters specified, [-]

Examples

```python
```
```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6, order=3)
>>> psd.delta_cdf(1e-6, 1e-5)
0.9165280099853876
```

**dn** *(self, fraction, n=None)*

Computes the diameter at which a specified *fraction* of the distribution falls under. Utilizes a bounded solver to search for the desired diameter.

**Parameters**

- **fraction** [float] Fraction of the distribution which should be under the calculated diameter,

- **n** [int, optional] None (for the *order* specified when the distribution was created), 0 (number), 1 (length), 2 (area), 3 (volume/mass), or any integer, [-]

**Returns**

- **d** [float] Particle size diameter, [m]

**Examples**

```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6, order=3)
>>> psd.dn(.5)
5e-06
>>> psd.dn(1)
0.0002947436533523378
>>> psd.dn(0)
0.0
```

**ds_discrete** *(self, d_min=None, d_max=None, pts=20, limit=1e-09, method='logarithmic')*

Create a particle spacing mesh to perform calculations with, according to one of several ways. The allowable meshes are ‘linear’, ‘logarithmic’, a geometric series specified by a Renard number such as ‘R10’, or the meshes available in one of several sieve standards.

**Parameters**

- **d_min** [float, optional] The minimum diameter at which the mesh starts, [m]
- **d_max** [float, optional] The maximum diameter at which the mesh ends, [m]
- **pts** [int, optional] The number of points to return for the mesh (note this is not respected by sieve meshes), [-]
- **limit** [float] If *d_min* or *d_max* is not specified, it will be calculated as the *dn* at which this limit or 1-limit exists (this is ignored for Renard numbers), [-]

**Returns**

- **ds** [list[float]] The generated mesh diameters, [m]

**Notes**

Note that when specifying a Renard series, only one of *d_min* or *d_max* can be respected! Provide only one of those numbers.
Note that when specifying a sieve standard the number of points is not respected!

References

[1], [2]

fractions_discrete (self, ds, n=None)

Computes the fractions of the cumulative distribution functions which lie between the specified specified particle diameters. The first diameter contains the cdf from 0 to it.

Parameters

ds [list[float]] Particle size diameters, [m]
n [int, optional] None (for the order specified when the distribution was created), 0 (number), 1 (length), 2 (area), 3 (volume/mass), or any integer, [-]

Returns

fractions [float] The differences in the cumulative distribution functions at the specified diameters and order, [-]

Examples

```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6, order=3)
>>> psd.fractions_discrete([1e-6, 1e-5, 1e-4, 1e-3])
[0.0006434710129138987, 0.9165280099853876, 0.08282851796190027, 1.0]
```

mean_size (self, p, q)

Calculates the mean particle size according to moment-ratio notation. This is the more common and often convenient definition.

\[
\bar{D}_{p,q}^{(p-q)} = \frac{\sum_i n_i D_i^p}{\sum_i n_i D_i^q}
\]

\[
\bar{D}_{p,p} = \exp \left[ \frac{\sum_i n_i D_i^p \ln D_i}{\sum_i n_i D_i^p} \right], \text{ if } p = q
\]

Note that \( n_i \) in the above equation is replaceable with the fraction of particles in that bin.

Parameters

p [int] Power and/or subscript of D moment in the above equations, [-]

q [int] Power and/or subscript of D moment in the above equations, [-]

Returns

d_pq [float] Mean particle size according to the specified p and q, [m]

Notes

The following is a list of common names for specific mean diameters.

- \( D[-3, 0] \): arithmetic harmonic mean volume diameter
- \( D[-2, 1] \): size-weighted harmonic mean volume diameter
- \( D[-1, 2] \): area-weighted harmonic mean volume diameter
• $D[-2, 0]$: arithmetic harmonic mean area diameter
• $D[-1, 1]$: size-weighted harmonic mean area diameter
• $D[-1, 0]$: arithmetic harmonic mean diameter
• $D[0, 0]$: arithmetic geometric mean diameter
• $D[1, 1]$: size-weighted geometric mean diameter
• $D[2, 2]$: area-weighted geometric mean diameter
• $D[3, 3]$: volume-weighted geometric mean diameter
• $D[1, 0]$: arithmetic mean diameter
• $D[2, 1]$: size-weighted mean diameter
• $D[3, 2]$: area-weighted mean diameter, Sauter mean diameter
• $D[4, 3]$: volume-weighted mean diameter, De Brouckere diameter
• $D[2, 0]$: arithmetic mean area diameter
• $D[3, 1]$: size-weighted mean area diameter
• $D[4, 2]$: area-weighted mean area diameter
• $D[5, 3]$: volume-weighted mean area diameter
• $D[3, 0]$: arithmetic mean volume diameter
• $D[4, 1]$: size-weighted mean volume diameter
• $D[5, 2]$: area-weighted mean volume diameter
• $D[6, 3]$: volume-weighted mean volume diameter

This notation was first introduced in [1].

The sum of p and q is called the order of the mean size [3].

$\bar{D}_{p,q} \equiv \bar{D}_{q,p}$

References

[1], [2], [3]

Examples

```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6)
>>> psd.mean_size(3, 2)
4.412494512922977e-06
```

Note that for the case where $p = q$, a different set of formulas are required - which do not have analytical results for many distributions. Therefore, a close numerical approximation is used instead, to perturb the values of $p$ and $q$ so they are 1E-9 away from each other. This leads only to slight errors, as in the example below where the correct answer is 5E-6.

```python
>>> psd.mean_size(3, 3)
4.9999999304923345e-06
```
mean_size_ISO(self, k, r)
Calculates the mean particle size according to moment notation (ISO). This system is related to the
moment-ratio notation as follows; see the mean_size method for the full formulas.
\[
\bar{x}_{p-q,q} \equiv \bar{x}_{k+r,r} \equiv D_{p,q}
\]

Parameters

- k [int] Power and/or subscript of D moment in the above equations, [-]
- r [int] Power and/or subscript of D moment in the above equations, [-]

Returns

- x_kr [float] Mean particle size according to the specified k and r in the ISO series, [m]

Notes

The following is a list of common names for specific mean diameters in the ISO naming convention.

- x[-3, 0]: arithmetic harmonic mean volume diameter
- x[-3, 1]: size-weighted harmonic mean volume diameter
- x[-3, 2]: area-weighted harmonic mean volume diameter
- x[-2, 0]: arithmetic harmonic mean area diameter
- x[-2, 1]: size-weighted harmonic mean area diameter
- x[-1, 0]: arithmetic harmonic mean diameter
- x[0, 0]: arithmetic geometric mean diameter
- x[0, 1]: size-weighted geometric mean diameter
- x[0, 2]: area-weighted geometric mean diameter
- x[0, 3]: volume-weighted geometric mean diameter
- x[1, 0]: arithmetic mean diameter
- x[1, 1]: size-weighted mean diameter
- x[1, 2]: area-weighted mean diameter, Sauter mean diameter
- x[1, 3]: volume-weighted mean diameter, De Brouckere diameter
- x[2, 0]: arithmetic mean area diameter
- x[1, 1]: size-weighted mean area diameter
- x[2, 1]: area-weighted mean area diameter
- x[2, 2]: volume-weighted mean area diameter
- x[3, 0]: arithmetic mean volume diameter
- x[3, 1]: size-weighted mean volume diameter
- x[3, 2]: area-weighted mean volume diameter
- x[3, 3]: volume-weighted mean volume diameter

When working with continuous distributions, the ISO series must be used to perform the actual calculations.
References

[1]

Examples

```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6)
>>> psd.mean_size_ISO(1, 2)
4.412484512922977e-06
```

```python
pdf(self, d, n=None)
```

Computes the probability density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis. The evaluation function varies with the distribution chosen. The interconversion between distribution orders is performed using the following formula [1]:

\[
q_s(d) = \frac{x^{(s-r)}q_r(d)dd}{\int_0^\infty d^{(s-r)}q_r(d)dd}
\]

**Parameters**

- **d** [float] Particle size diameter, [m]
- **n** [int, optional] None (for the order specified when the distribution was created), 0 (number), 1 (length), 2 (area), 3 (volume/mass), or any integer, [-]

**Returns**

- **pdf** [float] The probability density function at the specified diameter and order, [-]

**Notes**

The pdf order conversions are typically available analytically after some work. They have been verified numerically. See the various functions with names ending with ‘basis_integral’ for the formulations. The distributions normally do not have analytical limits for diameters of 0 or infinity, but large values suffice to capture the area of the integral.

References

[1]

Examples

```python
>>> psd = PSDLognormal(s=0.5, d_characteristic=5E-6, order=3)
>>> psd.pdf(1e-5)
30522.765209509154
>>> psd.pdf(1e-5, n=3)
30522.765209509154
>>> psd.pdf(1e-5, n=0)
1238.661379483343
```

```python
plot_cdf(self, n=[0,1,2,3], d_min=None, d_max=None, pts=500, method='logarithmic')
```

Plot the cumulative distribution function of the particle size distribution. The plotted range can be specified using `d_min` and `d_max`, or estimated automatically. One or more order can be plotted, by providing an iterable of ints as the value of `n` or just one int.
Parameters

n [tuple(int) or int, optional] None (for the order specified when the distribution was created), 0 (number), 1 (length), 2 (area), 3 (volume/mass), or any integer; as many as desired may be specified, [-]

d_min [float, optional] Lower particle size diameter, [m]

d_max [float, optional] Upper particle size diameter, [m]

pts [int, optional] The number of points for values to be calculated, [-]

method [str, optional] Either 'linear', 'logarithmic', a Renard number like 'R10' or 'R5' or'R2.5', or one of the sieve standards 'ISO 3310-1 R40/3', 'ISO 3310-1 R20', 'ISO 3310-1 R20/3', 'ISO 3310-1', 'ISO 3310-1 R10', 'ASTM E11'. [-]

plot_pdf (self, n=(0, 1, 2, 3), d_min=None, d_max=None, pts=500, normalized=False, method='linear')

Plot the probability density function of the particle size distribution. The plotted range can be specified using d_min and d_max, or estimated automatically. One or more order can be plotted, by providing an iterable of ints as the value of n or just one int.

Parameters

n [tuple(int) or int, optional] None (for the order specified when the distribution was created), 0 (number), 1 (length), 2 (area), 3 (volume/mass), or any integer; as many as desired may be specified, [-]

d_min [float, optional] Lower particle size diameter, [m]

d_max [float, optional] Upper particle size diameter, [m]

pts [int, optional] The number of points for values to be calculated, [-]

normalized [bool, optional] Whether to display the actual probability density function, which may have a huge magnitude - or to divide each point by the sum of all the points. Doing this is a common practice, but the values at each point are dependent on the number of points being plotted, and the distribution of the points; [-]

method [str, optional] Either 'linear', 'logarithmic', a Renard number like 'R10' or 'R5' or'R2.5', or one of the sieve standards 'ISO 3310-1 R40/3', 'ISO 3310-1 R20', 'ISO 3310-1 R20/3', 'ISO 3310-1', 'ISO 3310-1 R10', 'ASTM E11'. [-]

vssa

The volume-specific surface area of a particle size distribution.

\[ VSSA = \frac{6}{\bar{x}_{1,2}} \]

Returns

VSSA [float] The volume-specific surface area of the distribution, [m^2/m^3]

References

[1]

Examples
2.16.2 Discrete Particle Size Distributions

class fluids.particle_size_distribution.ParticleSizeDistribution(ds, fractions, cdf=False, order=3, monotonic=True)

Class representing a discrete particle size distribution specified by a series of diameter bins, and the quantity of particles in each bin. The quantities may be specified as either the fraction of particles in each bin, or as cumulative distributions. The input fractions can be specified to be in a mass basis (order=3), number basis (order=0), or the orders in between for length basis or area basis. If the fractions do not sum to 1, and cdf is False, then the fractions are normalized. This allows flow rates or counts of size bins to be given as well.

Parameters

ds [list[float]] Diameter bins; length of the specified quantities, optionally +1 that length to specify a cutoff diameter for the smallest diameter bin, [m]

fractions [list[float], optional] The mass/mole/volume/length/area/count fractions or cumulative distributions or counts of each particle size in each diameter bin (the type is specified by order), [-]

order [int, optional] 0 for a number distribution as input; 1 for length distribution; 2 for area distribution; 3 for mass, mole, or volume distribution, [-]

cdf [bool, optional] If the distribution is given as increasing fractions with 1 as the last result, cdf must be set to True, [-]

monotonic [bool, optional] If True, for interpolated quantities, monotonic splines will be used instead of the standard splines, [-]

Notes

Although the stated units of input are in meters, this class is actually independent of the units provided; all results will be consistent with the provided unit.

References

[R8b42d45c9b22-1], [R8b42d45c9b22-2]

Examples

Example problem from [R8b42d45c9b22-1], calculating several diameters and the cumulative distribution.

```python
>>> ds = 1E-6*np.array([240, 360, 450, 562.5, 703, 878, 1097, 1371, 1713, 2141, 2676, 3345, 4181, 5226, 6532])
>>> numbers = [65, 119, 232, 410, 629, 849, 990, 981, 825, 579, 297, 111, 21, 1]
>>> psd = ParticleSizeDistribution(ds=ds, fractions=numbers, order=0)
```
Attributes

fractions [list[float]] The mass/mole/volume basis fractions of particles in each bin, [-]

area_fractions [list[float]] The area fractions of particles in each bin, [-]

length_fractions [list[float]] The length fractions of particles in each bin, [-]

number_fractions [list[float]] The number fractions of particles in each bin, [-]

fraction_cdf [list[float]] The cumulative mass/mole/volume basis fractions of particles in each bin, [-]

area_cdf [list[float]] The cumulative area fractions of particles in each bin, [-]

length_cdf [list[float]] The cumulative length fractions of particles in each bin, [-]

number_cdf [list[float]] The cumulative number fractions of particles in each bin, [-]

size_classes [bool] Whether or not the diameter bins were set as size classes (as length of fractions + 1), [-]

N [int] The number of provided points, [-]

Methods

cdf(self, d[, n]) Computes the cumulative distribution density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.

cdf_discrete(self, ds[, n]) Computes the cumulative distribution functions for a list of specified particle diameters.

delta_cdf(self, d_min, d_max[, n]) Computes the difference in cumulative distribution function between two particle size diameters.

di_power(self, i[, power]) Method to calculate a power of a particle class/bin in a generic way so as to support when there are as many ds as fractions, or one more diameter spec than fractions.

dn(self, fraction[, n]) Computes the diameter at which a specified fraction of the distribution falls under.

ds_discrete(self[, d_min, d_max, pts, ...]) Create a particle spacing mesh to perform calculations with, according to one of several ways.

fit(self[, x0, distribution, n]) Incomplete method to fit experimental values to a curve.

fractions_discrete(self, ds[, n]) Computes the fractions of the cumulative distribution functions which lie between the specified specified particle diameters.

mean_size(self, p, q) Calculates the mean particle size according to moment-ratio notation.

mean_size_ISO(self, k, r) Calculates the mean particle size according to moment notation (ISO).
pdf(self, d[, n]) Computes the probability density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.

plot_cdf(self[, n, d_min, d_max, pts, method]) Plot the cumulative distribution function of the particle size distribution.

plot_pdf(self[, n, d_min, d_max, pts, ...]) Plot the probability density function of the particle size distribution.

Dis
Representative diameters of each bin.

di_power (self, i, power=1)
Method to calculate a power of a particle class/bin in a generic way so as to support when there are as many ds as fractions, or one more diameter spec than fractions.

When each bin has a lower and upper bound, the formula is as follows [1].

\[ D_i^r = \frac{D_i^{(r+1)}_{ub} - D_i^{(r+1)}_{lb}}{(D_i_{ub} - D_i_{lb})(r + 1)} \]

Where \(ub\) represents the upper bound, and \(lb\) represents the lower bound. Otherwise, the standard definition is used:

\[ D_i^r = D_i^r \]

Parameters

- \(i\) [int] The index of the diameter for the calculation, [-]
- \(power\) [int] The exponent, [-]

Returns

- di_power [float] The representative bin diameter raised to power, [m^power]

References

[1] fit (self, x0=None, distribution='lognormal', n=None, **kwargs)
Incomplete method to fit experimental values to a curve. It is very hard to get good initial guesses, which are really required for this. Differential evolution is promising. This API is likely to change in the future.

interpolated

mean_size (self, p, q)
Calculates the mean particle size according to moment-ratio notation. This is the more common and often convenient definition.

\[ [\bar{D}_{p,q}]^{(p-q)} = \frac{\sum n_i D_i^p}{\sum n_i D_i^q} \]
\[ [\bar{D}_{p,p}] = \exp \left[ \frac{\sum n_i D_i^p \ln D_i}{\sum n_i D_i^p} \right], \text{ if } p = q \]

Note that \(n_i\) in the above equation is replaceable with the fraction of particles in that bin.

Parameters
p  [int] Power and/or subscript of D moment in the above equations, [-]
q  [int] Power and/or subscript of D moment in the above equations, [-]

Returns

d_pq  [float] Mean particle size according to the specified p and q, [m]

Notes

The following is a list of common names for specific mean diameters.

- \( \bar{D}_{-3, 0} \): arithmetic harmonic mean volume diameter
- \( \bar{D}_{-2, 1} \): size-weighted harmonic mean volume diameter
- \( \bar{D}_{-1, 2} \): area-weighted harmonic mean volume diameter
- \( \bar{D}_{-2, 0} \): arithmetic harmonic mean area diameter
- \( \bar{D}_{-1, 1} \): size-weighted harmonic mean area diameter
- \( \bar{D}_{-1, 0} \): arithmetic harmonic mean diameter
- \( \bar{D}_{0, 0} \): arithmetic geometric mean diameter
- \( \bar{D}_{1, 1} \): size-weighted geometric mean diameter
- \( \bar{D}_{2, 2} \): area-weighted geometric mean diameter
- \( \bar{D}_{3, 3} \): volume-weighted geometric mean diameter
- \( \bar{D}_{1, 0} \): arithmetic mean diameter
- \( \bar{D}_{2, 1} \): size-weighted mean diameter
- \( \bar{D}_{2, 0} \): arithmetic mean area diameter
- \( \bar{D}_{3, 1} \): size-weighted mean area diameter
- \( \bar{D}_{4, 2} \): area-weighted mean area diameter
- \( \bar{D}_{5, 3} \): volume-weighted mean area diameter
- \( \bar{D}_{3, 0} \): arithmetic mean volume diameter
- \( \bar{D}_{4, 1} \): size-weighted mean volume diameter
- \( \bar{D}_{5, 2} \): area-weighted mean volume diameter
- \( \bar{D}_{6, 3} \): volume-weighted mean volume diameter

This notation was first introduced in [1].

The sum of p and q is called the order of the mean size [3].

\[
\bar{D}_{p,q} = \bar{D}_{q,p}
\]

References

[1], [2], [3]
Examples

```python
>>> ds = 1E-6*np.array([240, 360, 450, 562.5, 703, 878, 1097, 1371, 1713,
                      2141, 2676, 3345, 4181, 5226, 6532])
>>> numbers = [65, 119, 232, 410, 629, 849, 990, 981, 825, 579, 297, 111, 21, 1]
>>> psd = ParticleSizeDistribution(ds=ds, fractions=numbers, order=0)
>>> psd.mean_size(3, 2)
0.002269321031745045
```

(mean_size_ISO(self, k, r))
Calculates the mean particle size according to moment notation (ISO). This system is related to the moment-ratio notation as follows; see the mean_size method for the full formulas.

\[ x_{p,q} \equiv x_{k+r,r} \equiv D_{p,q} \]

Parameters

- **k** [int] Power and/or subscript of D moment in the above equations, [-]
- **r** [int] Power and/or subscript of D moment in the above equations, [-]

Returns

- **x_kr** [float] Mean particle size according to the specified k and r in the ISO series, [m]

Notes

The following is a list of common names for specific mean diameters in the ISO naming convention.

- • x[-3, 0]: arithmetic harmonic mean volume diameter
- • x[-3, 1]: size-weighted harmonic mean volume diameter
- • x[-3, 2]: area-weighted harmonic mean volume diameter
- • x[-2, 0]: arithmetic harmonic mean area diameter
- • x[-2, 1]: size-weighted harmonic mean area diameter
- • x[-1, 0]: arithmetic harmonic mean diameter
- • x[0, 0]: arithmetic geometric mean diameter
- • x[0, 1]: size-weighted geometric mean diameter
- • x[0, 2]: area-weighted geometric mean diameter
- • x[0, 3]: volume-weighted geometric mean diameter
- • x[1, 0]: arithmetic mean diameter
- • x[1, 1]: size-weighted mean diameter, **Sauter mean diameter**
- • x[1, 2]: area-weighted mean diameter, **De Brouckere diameter**
- • x[1, 3]: volume-weighted mean diameter
- • x[2, 0]: arithmetic mean area diameter
- • x[1, 1]: size-weighted mean area diameter
- • x[2, 2]: area-weighted mean area diameter
- • x[2, 3]: volume-weighted mean area diameter
• $x[3, 0]$: arithmetic mean volume diameter
• $x[3, 1]$: size-weighted mean volume diameter
• $x[3, 2]$: area-weighted mean volume diameter
• $x[3, 3]$: volume-weighted mean volume diameter

When working with continuous distributions, the ISO series must be used to perform the actual calculations.

References

[1]

Examples

```python
>>> ds = 1E-6*np.array([240, 360, 450, 562.5, 703, 1097, 1371, 1713, 2141, 2676, 3345, 4181, 5226, 6532])
>>> numbers = [65, 119, 232, 410, 629, 849, 981, 981, 825, 579, 297, 111, 21, 1]
>>> psd = ParticleSizeDistribution(ds=ds, fractions=numbers, order=0)
>>> psd.mean_size_ISO(1, 2)
0.002269321031745045
```

```python
class PSDInterpolated(ds, fractions, order=3, monotonic=True):
```

```
Attributes

• vssa The volume-specific surface area of a particle size distribution.

Methods
```
**2.16.3 Statistical Particle Size Distributions**

```python
class fluids.particle_size_distribution.PSDLognormal(d_characteristic, s, order=3, d_min=None, d_max=None)
```

**Attributes**

- `vssa` The volume-specific surface area of a particle size distribution.

**Methods**

- `cdf(self, d[, n])` Computes the cumulative distribution density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.
- `cdf_discrete(self, ds[, n])` Computes the cumulative distribution functions for a list of specified particle diameters.
- `delta_cdf(self, d_min, d_max[, n])` Computes the difference in cumulative distribution function between two particle size diameters.
- `dn(self, fraction[, n])` Computes the diameter at which a specified fraction of the distribution falls under.
- `ds_discrete(self[, d_min, d_max, pts, ...])` Create a particle spacing mesh to perform calculations with, according to one of several ways.
- `fractions_discrete(self, ds[, n])` Computes the fractions of the cumulative distribution functions which lie between the specified particle diameters.
- `mean_size(self, p, q)` Calculates the mean particle size according to moment-ratio notation.
- `mean_size_ISO(self, k, r)` Calculates the mean particle size according to moment notation (ISO).
- `pdf(self, d[, n])` Computes the probability density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.
- `plot_cdf(self[, n, d_min, d_max, pts, method])` Plot the cumulative distribution function of the particle size distribution.
- `plot_pdf(self[, n, d_min, d_max, pts, ...])` Plot the probability density function of the particle size distribution.
Table 8 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delta_cdf(self, d_min, d_max[, n])</td>
<td>Computes the difference in cumulative distribution function between two particle size diameters.</td>
</tr>
<tr>
<td>dn(self, fraction[, n])</td>
<td>Computes the diameter at which a specified fraction of the distribution falls under.</td>
</tr>
<tr>
<td>ds_discrete(self[, d_min, d_max, pts, ...])</td>
<td>Create a particle spacing mesh to perform calculations with, according to one of several ways.</td>
</tr>
<tr>
<td>fractions_discrete(self, ds[, n])</td>
<td>Computes the fractions of the cumulative distribution functions which lie between the specified specified particle diameters.</td>
</tr>
<tr>
<td>mean_size(self, p, q)</td>
<td>Calculates the mean particle size according to moment-ratio notation.</td>
</tr>
<tr>
<td>mean_size_ISO(self, k, r)</td>
<td>Calculates the mean particle size according to moment notation (ISO).</td>
</tr>
<tr>
<td>pdf(self, d[, n])</td>
<td>Computes the probability density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.</td>
</tr>
<tr>
<td>plot_cdf(self[, n, d_min, d_max, pts, method])</td>
<td>Plot the cumulative distribution function of the particle size distribution.</td>
</tr>
<tr>
<td>plot_pdf(self[, n, d_min, d_max, pts, ...])</td>
<td>Plot the probability density function of the particle size distribution.</td>
</tr>
</tbody>
</table>

name = 'Lognormal'
points = False
truncated = False
class fluids.particle_size_distribution.PSDGatesGaudinSchuhman(d_characteristic, m, order=3, d_min=None, d_max=None)

Bases: fluids.particle_size_distribution.ParticleSizeDistributionContinuous

Attributes

vssa The volume-specific surface area of a particle size distribution.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdf(self, d[, n])</td>
<td>Computes the cumulative distribution density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.</td>
</tr>
<tr>
<td>cdf_discrete(self, ds[, n])</td>
<td>Computes the cumulative distribution functions for a list of specified particle diameters.</td>
</tr>
<tr>
<td>delta_cdf(self, d_min, d_max[, n])</td>
<td>Computes the difference in cumulative distribution function between two particle size diameters.</td>
</tr>
<tr>
<td>dn(self, fraction[, n])</td>
<td>Computes the diameter at which a specified fraction of the distribution falls under.</td>
</tr>
<tr>
<td>ds_discrete(self[, d_min, d_max, pts, ...])</td>
<td>Create a particle spacing mesh to perform calculations with, according to one of several ways.</td>
</tr>
</tbody>
</table>
Table 9 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fractions_discrete(self, ds[, n])</td>
<td>Computes the fractions of the cumulative distribution functions which lie between the specified specified particle diameters.</td>
</tr>
<tr>
<td>mean_size(self, p, q)</td>
<td>Calculates the mean particle size according to moment-ratio notation.</td>
</tr>
<tr>
<td>mean_size_ISO(self, k, r)</td>
<td>Calculates the mean particle size according to moment notation (ISO).</td>
</tr>
<tr>
<td>pdf(self, d[, n])</td>
<td>Computes the probability density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.</td>
</tr>
<tr>
<td>plot_cdf(self[, n, d_min, d_max, pts, method])</td>
<td>Plot the cumulative distribution function of the particle size distribution.</td>
</tr>
<tr>
<td>plot_pdf(self[, n, d_min, d_max, pts,...])</td>
<td>Plot the probability density function of the particle size distribution.</td>
</tr>
</tbody>
</table>

```
name = 'Gates Gaudin Schuhman'
points = False
truncated = False
class fluids.particle_size_distribution.PSDRosinRammler(k, m, order=3, d_min=...)
Bases: fluids.particle_size_distribution.ParticleSizeDistributionContinuous

Attributes
vssa The volume-specific surface area of a particle size distribution.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdf(self, d[, n])</td>
<td>Computes the cumulative distribution density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.</td>
</tr>
<tr>
<td>cdf_discrete(self, ds[, n])</td>
<td>Computes the cumulative distribution functions for a list of specified particle diameters.</td>
</tr>
<tr>
<td>delta_cdf(self, d_min, d_max[, n])</td>
<td>Computes the difference in cumulative distribution function between two particle size diameters.</td>
</tr>
<tr>
<td>dn(self, fraction[, n])</td>
<td>Computes the diameter at which a specified fraction of the distribution falls under.</td>
</tr>
<tr>
<td>ds_discrete(self[, d_min, d_max, pts,...])</td>
<td>Create a particle spacing mesh to perform calculations with, according to one of several ways.</td>
</tr>
<tr>
<td>fractions_discrete(self, ds[, n])</td>
<td>Computes the fractions of the cumulative distribution functions which lie between the specified specified particle diameters.</td>
</tr>
<tr>
<td>mean_size(self, p, q)</td>
<td>Calculates the mean particle size according to moment-ratio notation.</td>
</tr>
<tr>
<td>mean_size_ISO(self, k, r)</td>
<td>Calculates the mean particle size according to moment notation (ISO).</td>
</tr>
</tbody>
</table>
```

Continued on next page
Table 10 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdf(self, d[, n])</td>
<td>Computes the probability density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.</td>
</tr>
<tr>
<td>plot_cdf(self[, n, d_min, d_max, pts, method])</td>
<td>Plot the cumulative distribution function of the particle size distribution.</td>
</tr>
<tr>
<td>plot_pdf(self[, n, d_min, d_max, pts, ...])</td>
<td>Plot the probability density function of the particle size distribution.</td>
</tr>
</tbody>
</table>

```python
name = 'Rosin Rammler'
points = False
truncated = False
class fluids.particle_size_distribution.PSDCustom(distribution, order=3.0, d_excessive=1.0, name=None, d_min=None, d_max=None)
Bases: fluids.particle_size_distribution.ParticleSizeDistributionContinuous
```

Attributes

- **vssa** The volume-specific surface area of a particle size distribution.

Methods

- **cdf(self, d[, n])** Computes the cumulative distribution density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.
- **cdf_discrete(self, ds[, n])** Computes the cumulative distribution functions for a list of specified particle diameters.
- **delta_cdf(self, d_min, d_max[, n])** Computes the difference in cumulative distribution function between two particle size diameters.
- **dn(self, fraction[, n])** Computes the diameter at which a specified fraction of the distribution falls under.
- **ds_discrete(self[, d_min, d_max, pts, ...])** Create a particle spacing mesh to perform calculations with, according to one of several ways.
- **fractions_discrete(self, ds[, n])** Computes the fractions of the cumulative distribution functions which lie between the specified specified particle diameters.
- **mean_size(self, p, q)** Calculates the mean particle size according to moment-ratio notation.
- **mean_size_ISO(self, k, r)** Calculates the mean particle size according to moment notation (ISO).
- **pdf(self, d[, n])** Computes the probability density function of a continuous particle size distribution at a specified particle diameter, an optionally in a specified basis.
- **plot_cdf(self[, n, d_min, d_max, pts, method])** Plot the cumulative distribution function of the particle size distribution.
- **plot_pdf(self[, n, d_min, d_max, pts, ...])** Plot the probability density function of the particle size distribution.
2.16.4 Helper functions: Lognormal Distribution

```python
fluids.particle_size_distribution.pdf_lognormal(d, d_characteristic, s)
```
Calculates the probability density function of a lognormal particle distribution given a particle diameter \(d\), characteristic particle diameter \(d_{\text{characteristic}}\), and distribution standard deviation \(s\).

\[
q(d) = \frac{1}{ds\sqrt{2\pi}} \exp \left[ -0.5 \left( \frac{\ln(d/d_{\text{characteristic}})}{s} \right)^2 \right]
\]

**Parameters**
- \(d\) [float] Specified particle diameter, [m]
- \(d_{\text{characteristic}}\) [float] Characteristic particle diameter; often \(D[3, 3]\) is used for this purpose but not always, [m]
- \(s\) [float] Distribution standard deviation, [-]

**Returns**
- \(pdf\) [float] Lognormal probability density function, [-]

**Notes**
The characteristic diameter can be in terms of number density (denoted \(q_0(d)\)), length density \((q_1(d))\), surface area density \((q_2(d))\), or volume density \((q_3(d))\). Volume density is most often used. Interconversions among the distributions is possible but tricky.

The standard distribution (i.e. the one used in Scipy) can perform the same computation with \(d_{\text{characteristic}}\) as the value of \(\text{scale}\).

```python
>>> import scipy.stats
>>> scipy.stats.lognorm.pdf(x=1E-4, s=1.1, scale=1E-5)
405.5420921156425
```
Scipy’s calculation is over 300 times slower however, and this expression is numerically integrated so speed is required.

**References**

[1]

**Examples**

```python
>>> pdf_lognormal(d=1E-4, d_characteristic=1E-5, s=1.1)
405.5420921156425
```

```python
fluids.particle_size_distribution.cdf_lognormal(d, d_characteristic, s)
```
Calculates the cumulative distribution function of a lognormal particle distribution given a particle diameter \(d\), characteristic particle diameter \(d_{\text{characteristic}}\), and distribution standard deviation \(s\).

\[
Q(d) = 0.5 \left( 1 + \text{erf} \left( \frac{\ln(d/d_{\text{characteristic}})}{s\sqrt{2}} \right) \right)
\]
Parameters

d [float] Specified particle diameter, [m]

d_characteristic [float] Characteristic particle diameter; often D[3, 3] is used for this purpose but not always, [m]

Returns

cdf [float] Lognormal cumulative density function, [-]

Notes

The characteristic diameter can be in terms of number density (denoted \(q_0(d)\)), length density (\(q_1(d)\)), surface area density (\(q_2(d)\)), or volume density (\(q_3(d)\)). Volume density is most often used. Interconversions among the distributions is possible but tricky.

The standard distribution (i.e. the one used in Scipy) can perform the same computation with \(d_{\text{characteristic}}\) as the value of \(scale\).

```python
>>> import scipy.stats
>>> scipy.stats.lognorm.cdf(x=1E-4, s=1.1, scale=1E-5)
0.9818369875798177
```

Scipy’s calculation is over 100 times slower however.

References

[1]

Examples

```python
>>> cdf_lognormal(d=1E-4, d_characteristic=1E-5, s=1.1)
0.9818369875798176
```

fluids.particle_size_distribution.pdf_lognormal_basis_integral(d, d_characteristic, s, n)

Calculates the integral of the multiplication of \(d^n\) by the lognormal pdf, given a particle diameter \(d\), characteristic particle diameter \(d_{\text{characteristic}}\), distribution standard deviation \(s\), and exponent \(n\).

\[
\int d^n \cdot q(d) \, dd = \frac{1}{2} \exp \left( \frac{s^2 n^2}{2} \right) d^n \left( \frac{d}{d_{\text{characteristic}}} \right)^{-n} \text{erf} \left[ \frac{s^2 n - \log(d/d_{\text{characteristic}})}{\sqrt{2} s} \right]
\]

This is the crucial integral required for interconversion between different bases such as number density (denoted \(q_0(d)\)), length density (\(q_1(d)\)), surface area density (\(q_2(d)\)), or volume density (\(q_3(d)\)).

Parameters

d [float] Specified particle diameter, [m]

d_characteristic [float] Characteristic particle diameter; often D[3, 3] is used for this purpose but not always, [m]

s [float] Distribution standard deviation, [-]
n  [int] Exponent of the multiplied n

Returns

pdf_basis_integral  [float] Integral of lognormal pdf multiplied by d^n, [-]

Notes

This integral has been verified numerically. This integral is itself integrated, so it is crucial to obtain an analytical form for at least this integral.

Note overflow or zero division issues may occur for very large values of s, larger than 10. No mathematical limit was able to be obtained with a CAS.

Examples

```python
>>> pdf_lognormal_basis_integral(d=1E-4, d_characteristic=1E-5, s=1.1, n=-2)
56228306549.26362
```

2.16.5 Helper functions: Gates Gaudin Schuhman Distribution

fluids.particle_size_distribution.pdf_Gates_Gaudin_Schuhman(d, d_characteristic, m)

Calculates the probability density of a particle distribution following the Gates, Gaudin and Schuhman (GGS) model given a particle diameter d, characteristic (maximum) particle diameter d_characteristic, and exponent m.

\[ q(d) = \frac{n}{d} \left( \frac{d}{d_{\text{characteristic}}} \right)^m \text{ if } d < d_{\text{characteristic}} \text{ else 0} \]

Parameters

- d  [float] Specified particle diameter, [m]
- d_characteristic  [float] Characteristic particle diameter; in this model, it is the largest particle size diameter in the distribution, [m]
- m  [float] Particle size distribution exponent, [-]

Returns

pdf  [float] GGS probability density function, [-]

Notes

The characteristic diameter can be in terms of number density (denoted \( q_0(d) \)), length density (\( q_1(d) \)), surface area density (\( q_2(d) \)), or volume density (\( q_3(d) \)). Volume density is most often used. Interconversions among the distributions is possible but tricky.

References

[1], [2]
Examples

```python
>>> pdf_Gates_Gaudin_Schuhman(d=2E-4, d_characteristic=1E-3, m=2.3)
283.835768512045
```

Calculates the cumulative distribution function of a particle distribution following the Gates, Gaudin and Schuhman (GGS) model given a particle diameter \(d\), characteristic (maximum) particle diameter \(d_{\text{characteristic}}\), and exponent \(m\).

\[
Q(d) = \left(\frac{d}{d_{\text{characteristic}}}\right)^m \text{ if } d < d_{\text{characteristic}} \text{ else } 1
\]

Parameters

- **d** [float] Specified particle diameter, [m]
- **d_characteristic** [float] Characteristic particle diameter; in this model, it is the largest particle size diameter in the distribution, [m]
- **m** [float] Particle size distribution exponent, [-]

Returns

- **cdf** [float] GGS cumulative density function, [-]

Notes

The characteristic diameter can be in terms of number density (denoted \(q_0(d)\)), length density (\(q_1(d)\)), surface area density (\(q_2(d)\)), or volume density (\(q_3(d)\)). Volume density is most often used. Interconversions among the distributions is possible but tricky.

References

[1], [2]

Examples

```python
>>> cdf_Gates_Gaudin_Schuhman(d=2E-4, d_characteristic=1E-3, m=2.3)
0.024681354508800397
```

Calculates the integral of the multiplication of \(d^n\) by the Gates, Gaudin and Schuhman (GGS) model given a particle diameter \(d\), characteristic (maximum) particle diameter \(d_{\text{characteristic}}\), and exponent \(m\).

\[
\int d^n \cdot q(d) \, dd = \frac{m}{m+n} d^n \left(\frac{d}{d_{\text{characteristic}}}\right)^m
\]

Parameters

- **d** [float] Specified particle diameter, [m]
**d_characteristic** [float] Characteristic particle diameter; in this model, it is the largest particle size diameter in the distribution, [m]

**m** [float] Particle size distribution exponent, [-]

**n** [int] Exponent of the multiplied n, [-]

**Returns**

**pdf_basis_integral** [float] Integral of Rosin Rammler pdf multiplied by d^n, [-]

**Notes**

This integral does not have any numerical issues as $d$ approaches 0.

**Examples**

```python
>>> pdf_Gates_Gaudin_Schuhman_basis_integral(d=2E-4, d_characteristic=1E-3, m=2.3, n=-3)
-10136984887.543015
```

### 2.16.6 Helper functions: Rosin Rammler Distribution

**fluids.particle_size_distribution.pdf_Rosin_Rammler**($d, k, m$)
Calculates the probability density of a particle distribution following the Rosin-Rammler (RR) model given a particle diameter $d$, and the two parameters $k$ and $m$.

$$q(d) = kmd^{(m-1)}\exp(-kd^m)$$

**Parameters**

| d | [float] Specified particle diameter, [m] |
| k | [float] Parameter in the model, [(1/m)^m] |
| m | [float] Parameter in the model, [-] |

**Returns**

**pdf** [float] RR probability density function, [-]

**References**

[1], [2]

**Examples**

```python
>>> pdf_Rosin_Rammler(1E-3, 200, 2)
0.3999200079994667
```

**fluids.particle_size_distribution.cdf_Rosin_Rammler**($d, k, m$)
Calculates the cumulative distribution function of a particle distribution following the Rosin-Rammler (RR) model given a particle diameter $d$, and the two parameters $k$ and $m$.

$$Q(d) = 1 - \exp(-kd^m)$$
Parameters

- **d** [float] Specified particle diameter, [m]
- **k** [float] Parameter in the model, [(1/m)^m]
- **m** [float] Parameter in the model, [-]

Returns

- **cdf** [float] RR cumulative density function, [-]

Notes

The characteristic diameter can be in terms of number density (denoted $q_0(d)$), length density ($q_1(d)$), surface area density ($q_2(d)$), or volume density ($q_3(d)$). Volume density is most often used. Interconversions among the distributions is possible but tricky.

References

[1], [2]

Examples

```python
>>> cdf_Rosin_Rammler(5E-2, 200, 2)
0.3934693402873667
```

```
fluids.particle_size_distribution.pdf_Rosin_Rammler_basis_integral(d, k, m, n)
```

Calculates the integral of the multiplication of $d^n$ by the Rosin Rammler (RR) pdf, given a particle diameter $d$, and the two parameters $k$ and $m$.

$$\int d^n \cdot q(d) \, dd = -d^{m+n} k(d^m k)^{-\frac{m+n}{m}} \Gamma\left(\frac{m+n}{m}\right) \text{gammaincc}\left(\frac{m+n}{m}, k d^m\right)$$

Parameters

- **d** [float] Specified particle diameter, [m]
- **k** [float] Parameter in the model, [(1/m)^m]
- **m** [float] Parameter in the model, [-]
- **n** [int] Exponent of the multiplied $n$, [-]

Returns

- **pdf_basis_integral** [float] Integral of Rosin Rammler pdf multiplied by $d^n$, [-]

Notes

This integral was derived using a CAS, and verified numerically. The `gammaincc` function is that from `scipy.special`, and `gamma` from the same.

For very high powers of $n$ or $m$ when the diameter is very low, exceptions may occur.
Examples

```python
>>> '{:g}'.format(pdf_Rosin_Rammler_basis_integral(5E-2, 200, 2, 3))
'-0.000452399'
```

### 2.16.7 Sieves

```python
class fluids.particle_size_distribution.Sieve(designation, old_designation=None, opening=None, opening_inch=None, Y_variation_avg=None, X_variation_max=None, max_opening=None, compliance_samples=None, compliance_sd=None, inspection_samples=None, inspection_sd=None, calibration_samples=None, calibration_sd=None, d_wire=None, d_wire_min=None, d_wire_max=None)
```

**Bases:** object

Class for storing data on sieves. If a property is not available, it is set to None.

**Attributes**

- **designation** [str] The standard name of the sieve - its opening’s length in units of millimeters
- **old_designation** [str] The older, imperial-esque name of the sieve; in Numbers, or inches for large sieves
- **opening** [float] The opening length of the sieve holes, [m]
- **opening_inch** [float] The opening length of the sieve holes in the rounded inches as stated in common tables (not exactly equal to the opening), [inch]
- **Y_variation_avg** [float] The allowable average variation in the Y direction of the sieve openings, [m]
- **X_variation_max** [float] The allowable maximum variation in the X direction of the sieve openings, [m]
- **max_opening** [float] The maximum allowable opening of the sieve, [m]
- **calibration_samples** [float] The number of opening sample inspections required for calibration-type sieve openings (per 100 ft² of sieve material), [1/(ft²)]
- **compliance_sd** [float] The maximum standard deviation of compliance-type sieve openings, [-]
- **inspection_samples** [float] The number of opening sample inspections required for inspection-type sieve openings (based on an 8-inch sieve), [-]
- **inspection_sd** [float] The maximum standard deviation of inspection-type sieve openings, [-]
- **calibration_samples** [float] The number of opening sample inspections required for calibration-type sieve openings (based on an 8-inch sieve), [-]
- **calibration_sd** [float] The maximum standard deviation of calibration-type sieve openings, [-]
- **d_wire** [float] Typical wire diameter of the specified sieve size, [m]
- **d_wire_min** [float] Permissible minimum wire diameter of specified sieve size, [m]
**d_wire_max** [float] Permissible maximum wire diameter of specified sieve size, [m]

**fluids.particle_size_distribution.ASTM_E11_sieves =**

Dictionary containing ASTM E-11 sieve series *Sieve* objects, indexed by their size in mm as a string.

**References**

[1]

**fluids.particle_size_distribution.ISO_3310_1_sieves =**

Dictionary containing all of the individual *Sieve* objects, on the ISO 3310-1:2016 series, indexed by their size in mm as a string.

**References**

[1]

**fluids.particle_size_distribution.ISO_3310_1_R20 =**

List containing all of the individual *Sieve* objects, on the ISO 3310-1:2016 R20 series only, ordered from largest openings to smallest.

**References**

[1]

**fluids.particle_size_distribution.ISO_3310_1_R20_3 =**

List containing all of the individual *Sieve* objects, on the ISO 3310-1:2016 R20/3 series only, ordered from largest openings to smallest.

**References**

[1]

**fluids.particle_size_distribution.ISO_3310_1_R40_3 =**

List containing all of the individual *Sieve* objects, on the ISO 3310-1:2016 R40/3 series only, ordered from largest openings to smallest.

**References**

[1]

**fluids.particle_size_distribution.ISO_3310_1_R10 =**

List containing all of the individual *Sieve* objects, on the ISO 3310-1:2016 R10 series only, ordered from largest openings to smallest.

**References**

[1]
2.16.8 Point Spacing

`fluids.particle_size_distribution.psd_spacing(d_min=None, d_max=None, pts=20, method='logarithmic')`

Create a particle spacing mesh in one of several ways for use in modeling discrete particle size distributions. The allowable meshes are ‘linear’, ‘logarithmic’, a geometric series specified by a Renard number such as ‘R10’, or the meshes available in one of several sieve standards.

**Parameters**
- **d_min** [float, optional] The minimum diameter at which the mesh starts, [m]
- **d_max** [float, optional] The maximum diameter at which the mesh ends, [m]
- **pts** [int, optional] The number of points to return for the mesh (note this is not respected by sieve meshes), [-]

**Returns**
- **ds** [list[float]] The generated mesh diameters, [m]

**Notes**
Note that when specifying a Renard series, only one of d_min or d_max can be respected! Provide only one of those numbers.
Note that when specifying a sieve standard the number of points is not respected!

**References**
[1], [2]

**Examples**

```python
gap = psd_spacing(d_min=5e-5, d_max=5e-4, method='ISO 3310-1 R20/3')
[6.3e-05, 9e-05, 0.000125, 0.00018, 0.00025, 0.000355, 0.0005]
```

2.17 Pipe schedules (fluids.piping)

`fluids.piping.nearest_pipe(Do=None, Di=None, NPS=None, schedule='40')`

Searches for and finds the nearest standard pipe size to a given specification. Acceptable inputs are:
- Nominal pipe size
- Nominal pipe size and schedule
- Outer diameter Do
- Outer diameter Do and schedule
- Inner diameter Di
- Inner diameter Di and schedule

Parameters

- **Do** [float] Pipe outer diameter, [m]
- **Di** [float] Pipe inner diameter, [m]
- **NPS** [float] Nominal pipe size, [-]
- **schedule** [str] String representing schedule size

Returns

- **NPS** [float] Nominal pipe size, [-]
- **Di** [float] Pipe inner diameter, [m]
- **Do** [float] Pipe outer diameter, [m]
- **t** [float] Pipe wall thickness, [m]

Notes

Internal units within this function are mm. The imperial schedules are not quite identical to these value, but all rounding differences happen in the sub-0.1 mm level.

References

[1], [2]

Examples

```python
>>> nearest_pipe(Di=0.021)
(1, 0.02664, 0.0334, 0.0033799999999999998)
>>> nearest_pipe(Do=.273, schedule='5S')
(10, 0.26630000000000004, 0.2731, 0.0034)
```

`fluids.piping.gauge_from_t(t, SI=True, schedule='BWG')`

Looks up the gauge of a given wire thickness of given schedule. Values are all non-linear, and tabulated internally.

Parameters

- **t** [float] Thickness, [m]
- **SI** [bool, optional] If False, requires that the thickness is given in inches not meters
- **schedule** [str] Gauge schedule, one of ‘BWG’, ‘AWG’, ‘SWG’, ‘MWG’, ‘BSWG’, or ‘SSWG’

Returns

- **gauge** [float-like] Wire Gauge, [-]
Notes

An internal variable, tol, is used in the selection of the wire gauge. If the next smaller wire gauge is within 10% of the difference between it and the previous wire gauge, the smaller wire gauge is selected. Accordingly, this function can return a gauge with a thickness smaller than desired in some circumstances.

- Birmingham Wire Gauge (BWG) ranges from 0.2 (0.5 inch) to 36 (0.004 inch).
- American Wire Gauge (AWG) ranges from 0.167 (0.58 inch) to 51 (0.00099 inch). These are used for electrical wires.
- Steel Wire Gauge (SWG) ranges from 0.143 (0.49 inch) to 51 (0.0044 inch). Also called Washburn & Moen wire gauge, American Steel gauge, Wire Co. gauge, and Roebling wire gauge.
- Music Wire Gauge (MWG) ranges from 0.167 (0.004 inch) to 46 (0.18 inch). Also called Piano Wire Gauge.
- British Standard Wire Gage (BSWG) ranges from 0.143 (0.5 inch) to 51 (0.001 inch). Also called Imperial Wire Gage (IWG).
- Stub’s Steel Wire Gage (SSWG) ranges from 1 (0.227 inch) to 80 (0.013 inch).

References

[1]

Examples

```python
>>> gauge_from_t(.5, SI=False, schedule='BWG')
0.2
```

fluids.piping.t_from_gauge(gauge, SI=True, schedule='BWG')

Looks up the thickness of a given wire gauge of given schedule. Values are all non-linear, and tabulated internally.

Parameters

- `gauge` [float-like] Wire Gauge, []
- `SI` [bool, optional] If False, will return a thickness in inches not meters

Returns

- `t` [float] Thickness, [m]

Notes

- Birmingham Wire Gauge (BWG) ranges from 0.2 (0.5 inch) to 36 (0.004 inch).
- American Wire Gauge (AWG) ranges from 0.167 (0.58 inch) to 51 (0.00099 inch). These are used for electrical wires.
- Steel Wire Gauge (SWG) ranges from 0.143 (0.49 inch) to 51 (0.0044 inch). Also called Washburn & Moen wire gauge, American Steel gauge, Wire Co. gauge, and Roebling wire gauge.
- Music Wire Gauge (MWG) ranges from 0.167 (0.004 inch) to 46 (0.18 inch). Also called Piano Wire Gauge.
• British Standard Wire Gage (BSWG) ranges from 0.143 (0.5 inch) to 51 (0.001 inch). Also called Imperial Wire Gage (IWG).
• Stub’s Steel Wire Gage (SSWG) ranges from 1 (0.227 inch) to 80 (0.013 inch)

References

[1]

Examples

```python
>>> t_from_gauge(.2, False, 'BWG')
0.5
```

2.18 Pump and motor sizing (fluids.pump)

`fluids.pump.VFD_efficiency(P, load=1)`

Returns the efficiency of a Variable Frequency Drive according to [1]. These values are generic, and not standardized as minimum values. Older VFDs often have much worse performance.

Parameters

- `P` [float] Power, [W]
- `load` [float, optional] Fraction of motor’s rated electrical capacity being used

Returns

- `efficiency` [float] VFD efficiency, [-]

Notes

The use of a VFD does change the characteristics of a pump curve’s efficiency, but this has yet to be quantified. The effect is small. This value should be multiplied by the product of the pump and motor efficiency to determine the overall efficiency.

Efficiency table is in units of hp, so a conversion is performed internally. If load not specified, assumed 1 - where maximum efficiency occurs. Table extends down to 3 hp and up to 400 hp; values outside these limits are rounded to the nearest known value. Values between standardized sizes are interpolated linearly. Load values extend down to 0.016.

The table used is for Pulse Width Modulation (PWM) VFDs.

References

[1]
Fluids Documentation, Release 0.1

```python
>>> VFD_efficiency(10*hp)
0.96
>>> VFD_efficiency(100*hp, load=0.2)
0.92
```

```python
fluids.pump.CSA_motor_efficiency(P, closed=False, poles=2, high_efficiency=False)
```

Returns the efficiency of a NEMA motor according to [1]. These values are standards, but are only for full-load operation.

**Parameters**

- `P` [float] Power, [W]
- `closed` [bool, optional] Whether or not the motor is enclosed
- `poles` [int, optional] The number of poles of the motor
- `high_efficiency` [bool, optional] Whether or not to look up the high-efficiency value

**Returns**

- `efficiency` [float] Guaranteed full-load motor efficiency, [-]

**Notes**

Criteria for being required to meet the high-efficiency standard is:

- Designed for continuous operation
- Operates by three-phase induction
- Is a squirrel-cage or cage design
- Is NEMA type A, B, or C with T or U frame; or IEC design N or H
- Is designed for single-speed operation
- Has a nominal voltage of less than 600 V AC
- Has a nominal frequency of 60 Hz or 50/60 Hz
- Has 2, 4, or 6 pole construction
- Is either open or closed

Pretty much every motor is required to meet the low-standard efficiency table, however.

Several low-efficiency standard high power values were added to allow for easy programming; values are the last listed efficiency in the table.

**References**

[1]

**Examples**

```python
>>> CSA_motor_efficiency(100*hp)
0.93
>>> CSA_motor_efficiency(100*hp, closed=True, poles=6, high_efficiency=True)
0.95
```
fluids.pump.motor_efficiency_underloaded\((P, load=0.5)\)

Returns the efficiency of a motor operating under its design power according to [1]. These values are generic; manufacturers usually list 4 points on their product information, but full-scale data is hard to find and not regulated.

**Parameters**

- \(P\) [float] Power, [W]
- \(load\) [float, optional] Fraction of motor’s rated electrical capacity being used

**Returns**

- \(efficiency\) [float] Motor efficiency, [-]

**Notes**

If the efficiency returned by this function is unattractive, use a VFD. The curves used here are polynomial fits to [1]’s graph, and curves were available for the following motor power ranges: 0-1 hp, 1.5-5 hp, 10 hp, 15-25 hp, 30-60 hp, 75-100 hp If above the upper limit of one range, the next value is returned.

**References**

[1]  

**Examples**

```python
>>> motor_efficiency_underloaded(1*hp)
0.8705179600980149
>>> motor_efficiency_underloaded(10.1*hp, .1)
0.6728425932357025
```

fluids.pump.corripio_pump_efficiency\((Q)\)

Estimates pump efficiency using the method in Corripio (1982) as shown in [1] and originally in [2]. Estimation only

\[
\eta_P = -0.316 + 0.24015 \ln(Q) - 0.01199 \ln(Q)^2
\]

**Parameters**

- \(Q\) [float] Volumetric flow rate, [m\(^3\)/s]

**Returns**

- \(efficiency\) [float] Pump efficiency, [-]

**Notes**

For Centrifugal pumps only. Range is 50 to 5000 GPM, but input variable is in metric. Values above this range and below this range will go negative, although small deviations are acceptable. Example 16.5 in [1].

**References**

[1], [2]
Examples

```python
>>> Corripio_pump_efficiency(461./15850.323)
0.705888867095162
```

`fluids.pump.Corripio_pump_efficiency(P)`

\[ \eta_M = 0.8 + 0.0319 \ln(P_B) - 0.00182 \ln(P_B)^2 \]

Parameters
- `P` [float] Power, [W]

Returns
- `efficiency` [float] Motor efficiency, [-]

Notes
Example 16.5 in [1].

References
[1], [2]

Examples

```python
>>> Corripio_motor_efficiency(137*745.7)
0.9128920875679222
```

`fluids.pump.Corripio_motor_efficiency(P)`

\[ \eta_M = 0.8 + 0.0319 \ln(P_B) - 0.00182 \ln(P_B)^2 \]

Parameters
- `P` [float] Power, [W]

Returns
- `efficiency` [float] Motor efficiency, [-]

Notes
Example 16.5 in [1].

References
[1], [2]

Examples

```python
>>> Corripio_motor_efficiency(137*745.7)
0.9128920875679222
```

`fluids.pump.specific_speed(Q, H, n=3600.0)`
Returns the specific speed of a pump operating at a specified Q, H, and n.

\[ n_S = \frac{n \sqrt{Q}}{H^{0.75}} \]

Parameters
- `Q` [float] Flow rate, [m^3/s]
- `H` [float] Head generated by the pump, [m]
- `n` [float, optional] Speed of pump [rpm]

Returns
- `nS` [float] Specific Speed, [rpm*m^0.75/s^0.5]

Notes
Defined at the BEP, with maximum fitting diameter impeller, at a given rotational speed.
References

[1]

Examples

Example from [1].

```python
>>> specific_speed(0.0402, 100, 3550)
22.50823182748925
```

```
fluids.pump.specif...em>specific_diameter</em>(<em>Q</em>, <em>H</em>, <em>D</em>)

Returns the specific diameter of a pump operating at a specified <em>Q</em>, <em>H</em>, and <em>D</em>.

\[
D_s = \frac{DH^{1/4}}{\sqrt{Q}}
\]

Parameters

- <em>Q</em> [float] Flow rate, [m$^3$/s]
- <em>H</em> [float] Head generated by the pump, [m]
- <em>D</em> [float] Pump impeller diameter [m]

Returns

- <em>Ds</em> [float] Specific diameter, [m$^{0.25}$/s$^{0.5}$]

Notes

Used in certain pump sizing calculations.

References

[1]

Examples

```python
>>> specific_diameter(Q=0.1, H=10., D=0.1)
0.5623413251903491
```

```
fluids.pump.<em>speed_synchronous</em>(<em>f</em>, <em>poles</em>=2, <em>phase</em>=3)

Returns the synchronous speed of a synchronous motor according to [1].

\[
N_s = \frac{120f \cdot \text{phase}}{\text{poles}}
\]

Parameters

- <em>f</em> [float] Line frequency, [Hz]
- <em>poles</em> [int, optional] The number of poles of the motor
- <em>phase</em> [int, optional] Line AC phase

Returns

- <em>Ns</em> [float] Speed of synchronous motor, [rpm]
Notes

Synchronous motors have no slip. Large synchronous motors are not self-starting.

References

[1]

Examples

```python
>>> speed_synchronous(50, poles=12)
1500.0
>>> speed_synchronous(60, phase=1)
3600.0
```

```python
fluids.pump.nema_sizes = [186.42496789556753, 248.5666238607567, 372.84993579113507, 559.2749036867026, 745.6998715822701, ...
list: all NEMA motor sizes in increasing order, in Watts.

fluids.pump.nema_sizes_hp = [0.25, 0.3333333333333333, 0.5, 0.75, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 5.5, 7.5, 10.0, 15.0, 20.0, 25.0, 30.0, 40.0, 50.0, 60.0, 75.0, 100.0, 125.0, 150.0, 175.0, 200.0, 250.0, 300.0, 350.0, 400.0, 450.0, 500.0]
list: all NEMA motor sizes in increasing order, in horsepower.

fluids.pump.motor_round_size(P)
Rounds up the power for a motor to the nearest NEMA standard power. The returned power is always larger or equal to the input power.

Parameters

- \( P \) [float] Power, [W]

Returns

- \( P_{\text{actual}} \) [float] Actual power, equal to or larger than input [W]

Notes

An exception is raised if the power required is larger than any of the NEMA sizes. Larger motors are available, but are unstandardized.

References

[1]

Examples

```python
>>> motor_round_size(1E5)
111854.98073734052
```

```python
fluids.pump.current_ideal(P, V, phase=3, PF=1)
Returns the current drawn by a motor of power \( P \) operating at voltage \( V \), with line AC of phase \( phase \) and power factor \( PF \) according to [1].

Single-phase power:

\[
I = \frac{P}{V \cdot PF}
\]

2.18. Pump and motor sizing (fluids.pump)
3-phase power:

\[ I = \frac{P}{V \cdot PF \sqrt{3}} \]

**Parameters**
- \( P \) [float] Power, [W]
- \( V \) [float] Voltage, [V]
- \( \text{phase} \) [int, optional] Line AC phase, either 1 or 3
- \( PF \) [float, optional] Power factor of motor

**Returns**
- \( I \) [float] Power drawn by motor, [A]

**Notes**
Does not include power used by the motor’s fan, or startor, or internal losses. These are all significant.

**References**
[1]

**Examples**

```python
>>> current_ideal(V=120, P=1E4, PF=1, phase=1)
83.33333333333333
```

### 2.19 Safety/relief valve sizing (fluids.safety_valve)

**fluids.safety_valve.API520_round_size(A)**

Rounds up the area from an API 520 calculation to an API526 standard valve area. The returned area is always larger or equal to the input area.

**Parameters**
- \( A \) [float] Minimum discharge area [m^2]

**Returns**
- \( area \) [float] Actual discharge area [m^2]

**Notes**
To obtain the letter designation of an input area, lookup the area with the following:

\[ \text{API526_letters[API526_A.index(area)]} \]

An exception is raised if the required relief area is larger than any of the API 526 sizes.
References

[1]

Examples

From [1], checked with many points on Table 8.

```python
>>> API520_round_size(1E-4)
0.00012645136
>>> API526_letters[API526_A.index(API520_round_size(1E-4))]
'E'
```

```python
fluids.safety_valve.API520_C(k)
Calculates coefficient C for use in API 520 critical flow relief valve sizing.

\[ C = 0.03948 \sqrt[\frac{k}{k + 1}] {\frac{2}{k + 1}} \]

Parameters

- k [float] Isentropic coefficient or ideal gas heat capacity ratio [-]

Returns

- C [float] Coefficient C [-]

Notes

If C cannot be established, assume a coefficient of 0.0239, the highest value possible for C.

Although not dimensional, C varies with the units used.

If k is exactly equal to 1, the expression is undefined, and the formula must be simplified as follows from an application of L’Hopital’s rule.

\[ C = 0.03948 \sqrt{\frac{1}{e}} \]

References

[1]

Examples

From [1], checked with many points on Table 8.

```python
>>> API520_C(1.35)
0.02669419967057233
```

```python
fluids.safety_valve.API520_F2(k, P1, P2)
Calculates coefficient F2 for subcritical flow for use in API 520 subcritical flow relief valve sizing.

\[ F_2 = \sqrt{\frac{k}{k - 1}} r^2 \left[ \frac{1 - r^{k-1}}{1 - r} \right] \]
```
\[ r = \frac{P_2}{P_1} \]

**Parameters**

- **k** [float] Isentropic coefficient or ideal gas heat capacity ratio [-]
- **P1** [float] Upstream relieving pressure; the set pressure plus the allowable overpressure, plus atmospheric pressure, [Pa]
- **P2** [float] Built-up backpressure; the increase in pressure during flow at the outlet of a pressure-relief device after it opens, [Pa]

**Returns**

- **F2** [float] Subcritical flow coefficient \(F2\) [-]

**Notes**

\(F2\) is completely dimensionless.

**References**

[1]

**Examples**

From [1] example 2, matches.

```python
>>> API520_F2(1.8, 1E6, 7E5)
0.8600724121105563
```

```python
fluids.safety_valve.API520_Kv(Re)
```

Calculates correction due to viscosity for liquid flow for use in API 520 relief valve sizing.

\[
K_v = \left( 0.9935 + \frac{2.878}{Re^{0.5}} + \frac{342.75}{Re^{1.5}} \right)^{-1}
\]

**Parameters**

- **Re** [float] Reynolds number for flow out the valve [-]

**Returns**

- **Kv** [float] Correction due to viscosity [-]

**Notes**

Reynolds number in the standard is defined as follows, with \(Q\) in \(L/min\), \(G_1\) as specific gravity, \(\mu\) in centipoise, and area in \(mm^2\):

\[
Re = \frac{Q(18800G_1)}{\mu \sqrt{A}}
\]

It is unclear how this expression was derived with a constant of 18800; the following code demonstrates what the constant should be:
>>> from scipy.constants import *
>>> liter/minute*1000./(0.001*(milli**2)**0.5)
16666.666666666668

References
[1]

Examples
From [1], checked with example 5.

>>> API520_Kv(100)
0.6157445891444229

fluids.safety_valve.API520_N(P1)
Calculates correction due to steam pressure for steam flow for use in API 520 relief valve sizing.

\[ K_N = 0.02764P_1 - 1000 \]
\[ -0.03324P_1 - 1061 \]

Parameters
P1 [float] Upstream relieving pressure; the set pressure plus the allowable overpressure, plus atmospheric pressure, [Pa]

Returns
KN [float] Correction due to steam temperature [-]

Notes
Although not dimensional, KN varies with the units used.

For temperatures above 922 K or 22057 kPa, KN is not defined.

Internally, units of kPa are used to match the equation in the standard.

References
[1]

Examples
Custom example:

>>> API520_N(1774700)
0.9490406958152466

fluids.safety_valve.API520_SH(T1, P1)
Calculates correction due to steam superheat for steam flow for use in API 520 relief valve sizing. 2D interpolation among a table with 28 pressures and 10 temperatures is performed.

Parameters
T1 [float] Temperature of the fluid entering the valve [K]

P1 [float] Upstream relieving pressure; the set pressure plus the allowable overpressure, plus atmospheric pressure, [Pa]

**Returns**

KSH [float] Correction due to steam superheat [-]

**Notes**

For P above 20679 kPag, use the critical flow model. Superheat cannot be above 649 degrees Celsius. If T1 is above 149 degrees Celsius, returns 1.

**References**

[1]

**Examples**

Custom example from table 9:

```python
>>> API520_SH(593+273.15, 1066.325E3)
0.7201800000000002
```

`fluids.safety_valve.API520_B(Pset, Pback, overpressure=0.1)`

Calculates capacity correction due to backpressure on balanced spring-loaded PRVs in vapor service. For pilot operated valves, this is always 1. Applicable up to 50% of the percent gauge backpressure, For use in API 520 relief valve sizing. 1D interpolation among a table with 53 backpressures is performed.

**Parameters**

Pset [float] Set pressure for relief [Pa]

Pback [float] Backpressure, [Pa]

overpressure [float, optional] The maximum fraction overpressure; one of 0.1, 0.16, or 0.21, []

**Returns**

Kb [float] Correction due to vapor backpressure [-]

**Notes**

If the calculated gauge backpressure is less than 30%, 38%, or 50% for overpressures of 0.1, 0.16, or 0.21, a value of 1 is returned.

Percent gauge backpressure must be under 50%.

**References**

[1]
Examples

Custom examples from figure 30:

```python
>>> API520_B(1E6, 5E5)
0.7929945420944432
```

`fluids.safety_valve.API520_W(Pset, Pback)`
Calculates capacity correction due to backpressure on balanced spring-loaded PRVs in liquid service. For pilot operated valves, this is always 1. Applicable up to 50% of the percent gauge backpressure. For use in API 520 relief valve sizing. 1D interpolation among a table with 53 backpressures is performed.

**Parameters**
- **Pset** [float] Set pressure for relief [Pa]
- **Pback** [float] Backpressure, [Pa]

**Returns**
- **KW** [float] Correction due to liquid backpressure [-]

**Notes**
If the calculated gauge backpressure is less than 15%, a value of 1 is returned.

**References**
[1]

Examples

Custom example from figure 31:

```python
>>> API520_W(1E6, 3E5) # 22% overpressure
0.9511471848008564
```

`fluids.safety_valve.API520_A_g(m, T, Z, MW, k, P1, P2=101325, Kd=0.975, Kb=1, Kc=1)`
Calculates required relief valve area for an API 520 valve passing a gas or a vapor, at either critical or sub-critical flow.

For critical flow:

\[
A = \frac{m}{CKdKbKc} \sqrt{\frac{TZ}{M}}
\]

For sub-critical flow:

\[
A = \frac{17.9m}{FP_2KdKc} \sqrt{\frac{TZ}{MP_1(P_1 - P_2)}}
\]

**Parameters**
- **m** [float] Mass flow rate of vapor through the valve, [kg/s]
- **T** [float] Temperature of vapor entering the valve, [K]
- **Z** [float] Compressibility factor of the vapor, [-]
MW [float] Molecular weight of the vapor, [g/mol]

k [float] Isentropic coefficient or ideal gas heat capacity ratio [-]

P1 [float] Upstream relieving pressure; the set pressure plus the allowable overpressure, plus atmospheric pressure, [Pa]

P2 [float, optional] Built-up backpressure; the increase in pressure during flow at the outlet of a pressure-relief device after it opens, [Pa]

Kd [float, optional] The effective coefficient of discharge, from the manufacturer or for preliminary sizing, using 0.975 normally or 0.62 when used with a rupture disc as described in [1], []

Kb [float, optional] Correction due to vapor backpressure [-]

Kc [float, optional] Combination correction factor for installation with a rupture disk upstream of the PRV, []

Returns

A [float] Minimum area for relief valve according to [1], [m^2]

Notes

Units are interally kg/hr, kPa, and mm^2 to match [1].

References

[1]

Examples

Example 1 from [1] for critical flow, matches:

```python
>>> API520_A_g(m=24270/3600., T=348., Z=0.90, MW=51., k=1.11, P1=670E3, Kb=1, Kc=1)
0.0036990460646834414
```

Example 2 from [1] for sub-critical flow, matches:

```python
>>> API520_A_g(m=24270/3600., T=348., Z=0.90, MW=51., k=1.11, P1=670E3, P2=532E3, Kd=0.975, Kb=1, Kc=1)
0.004248358775943481
```

The mass flux in (kg/(s*m^2)) can be found by dividing the specified mass flow by the calculated area:

```python
>>> (24270/3600.)/API520_A_g(m=24270/3600., T=348., Z=0.90, MW=51., k=1.11, P1=670E3, Kd=0.975, Kb=1, Kc=1)
1822.541960488834
```

```
fluids.safety_valve.API520_A_steam(m, T, P1, Kd=0.975, Kb=1, Kc=1)
```

Calculates required relief valve area for an API 520 valve passing a steam, at either saturation or superheat but not partially condensed.

\[
A = \frac{190.5m}{P_1 K_d K_b K_c K_N K_{SH}}
\]
Parameters

\[ m \] [float] Mass flow rate of steam through the valve, [kg/s]

\[ T \] [float] Temperature of steam entering the valve, [K]

\[ P1 \] [float] Upstream relieving pressure; the set pressure plus the allowable overpressure, plus atmospheric pressure, [Pa]

\[ Kd \] [float, optional] The effective coefficient of discharge, from the manufacturer or for preliminary sizing, using 0.975 normally or 0.62 when used with a rupture disc as described in [1], [ ]

\[ Kb \] [float, optional] Correction due to vapor backpressure [-]

\[ Kc \] [float, optional] Combination correction factor for installation with a rupture disk upstream of the PRV, []

Returns

\[ A \] [float] Minimum area for relief valve according to [1], [m^2]

Notes

Units are interlally kg/hr, kPa, and mm^2 to match [1]. With the provided temperature and pressure, the KN coefficient is calculated with the function API520_N; as is the superheat correction KSH, with the function API520_SH.

References

[1]

Examples

Example 4 from [1], matches:

```python
>>> API520_A_steam(m=69615/3600., T=592.5, P1=12236E3, Kd=0.975, Kb=1, Kc=1)
0.0011034712423692733
```

2.20 Liquid-Vapor Separators (fluids.separator)

`fluids.separator.v_Sounders_Brown(K, rhol, rhog)`

Calculates the maximum allowable vapor velocity in a two-phase separator to permit separation between entrained droplets and the gas using an empirical \( K \) factor, named after Sounders and Brown [1]. This is a simplifying expression for terminal velocity and drag on particles.

\[ v_{max} = K_{SB} \sqrt{\frac{\rho_l - \rho_g}{\rho_g}} \]

Parameters

\( K \) [float] Sounders Brown \( K \) factor for two-phase separator design, [m/s]

\( \text{rhol} \) [float] Density of liquid phase [kg/m^3]

\( \text{rhog} \) [float] Density of gas phase [kg/m^3]
Returns

\( v_{\text{max}} \) [float] Maximum allowable vapor velocity in a two-phase separator to permit separation between entrained droplets and the gas, [m/s]

Notes

The Sounders Brown K factor is related to the terminal velocity as shown in the following expression.

\[
v_{\text{term}} = v_{\text{max}} = \sqrt{\frac{4gd_p(\rho_p - \rho_f)}{3Cd\rho_f}}
\]

\[
v_{\text{term}} = \sqrt{\frac{(\rho_p - \rho_f)}{\rho_f}} \frac{4gd_p}{3Cd}
\]

\[
v_{\text{term}} = K_{SB} \sqrt{\frac{4gd_p}{3Cd}}
\]

Note this form corresponds to the Newton’s law range (Re > 500), but in reality droplets are normally in the intermediate or Stoke’s law region [2]. For this reason using the drag coefficient expression directly is cleaner, but identical results can be found with the Sounders Brown equation.

References

[1], [2]

Examples

```python
>>> v_Sounders_Brown(K=0.08, rhol=985.4, rhog=1.3)
2.2010906387516167
```

`fluids.separator.K_separator_Watkins(x, rhol, rhog, horizontal=False, method='spline')`

Calculates the Sounders-Brown K factor as used in determining maximum allowable gas velocity in a two-phase separator in either a horizontal or vertical orientation. This function approximates a graph published in [1] to determine \( K \) as used in the following equation:

\[
v_{\text{max}} = K_{SB} \sqrt{\frac{\rho_l - \rho_g}{\rho_g}}
\]

The graph has \( K_{SB} \) on its y-axis, and the following as its x-axis:

\[
\frac{m_l}{m_g} \sqrt{\frac{\rho_g}{\rho_l}} = \frac{(1 - x)}{x} \sqrt{\frac{\rho_g}{\rho_l}}
\]

Cubic spline interpolation is the default method of retrieving a value from the graph, which was digitized with Engauge-Digitizer.

Also supported are two published curve fits to the graph. The first is that of Blackwell (1984) [2], as follows:

\[
K_{SB} = \exp(-1.942936 - 0.814894X - 0.179390X^2 - 0.0123790X^3 + 0.000386235X^4 + 0.000259550X^5)
\]

\[
X = \log \left( \frac{1 - x}{x} \sqrt{\frac{\rho_g}{\rho_l}} \right)
\]
The second is that of Branan (1999), as follows:

\[ K_{SB} = \exp(-1.877478097 - 0.81145804597X - 0.1870744085X^2 - 0.0145228667X^3 - 0.00101148518X^4) \]

\[ X = \log \left( \frac{1 - x}{x} \right) \sqrt{\frac{\rho_g}{\rho_l}} \]

**Parameters**

- \( x \) [float] Quality of fluid entering separator, [-]
- \( \text{rhol} \) [float] Density of liquid phase [kg/m\(^3\)]
- \( \text{rhog} \) [float] Density of gas phase [kg/m\(^3\)]
- \( \text{horizontal} \) [bool, optional] Whether to use the vertical or horizontal value; horizontal is 1.25 higher
- \( \text{method} \) [str] One of ‘spline’, ‘blackwell’, or ‘branan’

**Returns**

- \( K \) [float] Sounders Brown horizontal or vertical \( K \) factor for two-phase separator design only, [m/s]

**Notes**

Both the ‘branan’ and ‘blackwell’ models are used frequently. However, the spline is much more accurate.

No limits checking is enforced. However, the x-axis spans only 0.006 to 5.4, and the function should not be used outside those limits.

**References**

[1], [2], [3]

**Examples**

```python
>>> K_separator_Watkins(0.88, 985.4, 1.3, horizontal=True)
0.07951613600476297
```

**fluids.separator.K_separator_demister_York** (*P*, *horizontal=*

Calculates the Sounders Brown \( K \) factor as used in determining maximum permissible gas velocity in a two-phase separator in either a horizontal or vertical orientation, with a demister. This function is a curve fit to [1] published in [2] and is widely used.

For \( 1 < P < 15 \) psia:

\[ K = 0.1821 + 0.0029P + 0.0460 \ln P \]

For \( 15 \leq P \leq 40 \) psia:

\[ K = 0.35 \]

For \( P < 5500 \) psia:

\[ K = 0.430 - 0.023 \ln P \]

In the above equations, \( P \) is in units of psia.
Parameters

- **P** [float] Pressure of separator, [Pa]
  - **horizontal** [bool, optional] Whether to use the vertical or horizontal value; horizontal is 1.25 times higher, [-]

Returns

- **K** [float] Sounders Brown Horizontal or vertical $K$ factor for two-phase separator design with a demister, [m/s]

Notes

If the input pressure is under 1 psia, 1 psia is used. If the input pressure is over 5500 psia, 5500 psia is used.

References

[2], [1]

Examples

```python
>>> K_separator_demister_York(975*psi)
0.08281536035331669
```

```python
fluids.separator.K_Sounders_Brown_theoretical(D, Cd, g=9.80665)
```

Converts a known drag coefficient into a Sounders-Brown $K$ factor for two-phase separator design. This factor is the traditional way for separator diameters to be obtained although it is unnecessary and the theoretical drag coefficient method can be used instead.

\[
K_{SB} = \sqrt{\frac{(\rho_p - \rho_f)}{\rho_f}} = \sqrt{\frac{4gd_p}{3C_D}}
\]

Parameters

- **D** [float] Design diameter of the droplets, [m]
- **Cd** [float] Drag coefficient [-]
- **g** [float, optional] Acceleration due to gravity, [m/s^2]

Returns

- **K** [float] Sounders Brown $K$ factor for two-phase separator design, [m/s]

Notes

Drag coefficient is a function of velocity; so iteration is needed to obtain the most correct answer. The following example shows the use of iteration to obtain the final velocity:

```python
>>> from fluids import *
>>> V = 2.0
>>> D = 150E-6
>>> rho = 1.3
>>> rhol = 700.
```
>>> mu = 1E-5
>>> for i in range(10):
...     Re = Reynolds(V=V, rho=rho, mu=mu, D=D)
...     Cd = drag_sphere(Re)
...     K = K_Sounders_Brown_theoretical(D=D, Cd=Cd)
...     V = v_Sounders_Brown(K, rhol=rhol, rhog=rho)
...     print(V)
0.760933074177
0.562429393401
0.507328950507
0.489571420955
0.483560219469
0.481490760336
0.480774149346
0.48052499591
0.480439162498
0.480409176902

The use of Sounders-Brown constants can be replaced as follows (the v_terminal method includes its own solver for terminal velocity):

>>> from fluids.drag import v_terminal
>>> v_terminal(D=D, rhop=rhol, rho=rho, mu=mu)
0.4803932186998833

References

[1]

Examples

>>> K_Sounders_Brown_theoretical(D=150E-6, Cd=0.5)
0.06263114241333939

2.21 Pneumatic conveying (fluids.saltation)

fluids.saltation.Rizk(mp, dp, rhog, D)
Calculates saltation velocity of the gas for pneumatic conveying, according to [1] as described in [2] and many others.

\[
\mu = \left( \frac{1}{10^{1440d_p+1196}} \right) (Fr_s)^{1100d_p+2.5}
\]

\[
Fr_s = \frac{V_{salt}}{\sqrt{gD}}
\]

\[
\mu = \frac{mp}{\frac{1}{4}D^2V\rho_f}
\]

Parameters

mp [float] Solid mass flow rate, [kg/s]
Fluids Documentation, Release 0.1

\[ \text{dp} \quad \text{[float] Particle diameter, [m]} \]
\[ \text{rhog} \quad \text{[float] Gas density, [kg/m}^3\text{]} \]
\[ D \quad \text{[float] Diameter of pipe, [m]} \]

Returns

\[ V \quad \text{[float] Saltation velocity of gas, [m/s]} \]

Notes

Model is rearranged to be explicit in terms of saltation velocity internally.

References

[1], [2], [3]

Examples

Example is from [3].

```python
>>> Rizk(mp=0.25, dp=100E-6, rhog=1.2, D=.078)
9.8833092829357
```

\text{fluids.saltation.\texttt{Matsumoto\_1974}}(mp, rhop, dp, rhog, D, V\text{\textendash}terminal=1)

Calculates saltation velocity of the gas for pneumatic conveying, according to [1]. Also described in [2].

\[
\mu = 0.448 \left( \frac{\rho_p}{\rho_f} \right)^{0.50} \left( \frac{Fr_p}{10} \right)^{1.75} \left( \frac{Fr_s}{10} \right)^3 \]

\[
Fr_s = \frac{V_{salt}}{\sqrt{gD}}
\]

\[
Fr_p = \frac{V_{terminal}}{\sqrt{gd_p}}
\]

\[
\mu = \frac{m_p}{\frac{2}{3} D^2 V \rho_f}
\]

Parameters

\texttt{mp} \quad \text{[float] Solid mass flow rate, [kg/s]} \]
\texttt{rhop} \quad \text{[float] Particle density, [kg/m}^3\text{]} \]
\texttt{dp} \quad \text{[float] Particle diameter, [m]} \]
\texttt{rhog} \quad \text{[float] Gas density, [kg/m}^3\text{]} \]
\texttt{D} \quad \text{[float] Diameter of pipe, [m]} \]
\texttt{V\textendash}terminal \quad \text{[float] Terminal velocity of particle settling in gas, [m/s]} \]

Returns

\[ V \quad \text{[float] Saltation velocity of gas, [m/s]} \]
Notes

Model is rearranged to be explicit in terms of saltation velocity internally. Result looks high, something may be wrong. For particles > 0.3 mm.

References

[1], [2]

Examples

```python
>>> Matsumoto_1974(mp=1., rhop=1000., dp=1E-3, rhog=1.2, D=0.1, Vterminal=5.24)
19.583617317317895
```

```
 fluids.saltation.Matsumoto_1975(mp, rhop, dp, rhog, D, Vterminal=1)
 Calculates saltation velocity of the gas for pneumatic conveying, according to [1]. Also described in [2].

\[
\mu = 1.11 \left( \frac{\rho_p}{\rho_f} \right)^{0.55} \left( \frac{Fr_p}{10} \right)^{-2.3} \left( \frac{Fr_s}{10} \right)^{3}
\]

\[
Fr_s = \frac{V_{salt}}{\sqrt{gD}}
\]

\[
Fr_p = \frac{V_{terminal}}{\sqrt{gD_p}}
\]

\[
\mu = \frac{m_p}{\frac{\pi}{4} D^2 V \rho_f}
\]

Parameters

- **mp** [float] Solid mass flow rate, [kg/s]
- **rhop** [float] Particle density, [kg/m^3]
- **dp** [float] Particle diameter, [m]
- **rhog** [float] Gas density, [kg/m^3]
- **D** [float] Diameter of pipe, [m]
- **Vterminal** [float] Terminal velocity of particle settling in gas, [m/s]

Returns

- **V** [float] Saltation velocity of gas, [m/s]

Notes

Model is rearranged to be explicit in terms of saltation velocity internally. Result looks high, something may be wrong. For particles > 0.3 mm.

References

[1], [2]
Examples

```python
>>> Matsumoto_1975(mp=1., rhop=1000., dp=1E-3, rhog=1.2, D=0.1, Vterminal=5.24)
18.04523091703009
```

fluids.saltation.Matsumoto_1977(mp, rhop, dp, rhog, D, Vterminal=1)
Calculates saltation velocity of the gas for pneumatic conveying, according to [1] and reproduced in [2], [3], and [4].

First equation is used if third equation yields d*_p higher than dp. Otherwise, use equation 2.

\[
\mu = 5560 \left( \frac{d_p}{D} \right)^{1.43} \left( \frac{Fr_s}{10} \right)^4
\]

\[
\mu = 0.373 \left( \frac{\rho_p}{\rho_f} \right)^{1.06} \left( \frac{Fr_p}{10} \right)^{-3.7} \left( \frac{Fr_s}{10} \right)^{3.61}
\]

\[
d_p^* = 1.39 \left( \frac{\rho_p}{\rho_f} \right)^{-0.74}
\]

\[
Fr_s = \frac{V_{salt}}{\sqrt{gD}}
\]

\[
Fr_p = \frac{V_{terminal}}{\sqrt{gd_p}}
\]

\[
\mu = \frac{m_p}{\frac{\pi}{4} D^2 \rho_f}
\]

Parameters

- `mp` [float] Solid mass flow rate, [kg/s]
- `rhop` [float] Particle density, [kg/m^3]
- `dp` [float] Particle diameter, [m]
- `rhog` [float] Gas density, [kg/m^3]
- `D` [float] Diameter of pipe, [m]
- `Vterminal` [float] Terminal velocity of particle settling in gas, [m/s]

Returns

- `V` [float] Saltation velocity of gas, [m/s]

Notes

Model is rearranged to be explicit in terms of saltation velocity internally.

References

[1], [2], [3], [4]
Examples

Example is only a self-test.

Course routine, terminal velocity input is from example in [2].

```python
>>> Matsumoto_1977(mp=1., rhop=1000., dp=1E-3, rhog=1.2, D=0.1, Vterminal=5.24)
16.64284834446686
```

`fluids.saltation.Schade(mp, rhop, dp, rhog, D)`
Calculates saltation velocity of the gas for pneumatic conveying, according to [1] as described in [2], [3], [4], and [5].

\[ \frac{F_{rs}}{\mu} = 0.11 \left( \frac{D}{d_p} \right)^{0.025} \left( \frac{\rho_p}{\rho_f} \right)^{0.34} \]

\[ F_{rs} = \frac{V_{salt}}{\sqrt{gD}} \]

\[ \mu = \frac{m_p}{\frac{\pi}{4} D^2 V_f} \]

**Parameters**

- `mp` [float] Solid mass flow rate, [kg/s]
- `rhop` [float] Particle density, [kg/m^3]
- `dp` [float] Particle diameter, [m]
- `rhog` [float] Gas density, [kg/m^3]
- `D` [float] Diameter of pipe, [m]

**Returns**

- `V` [float] Saltation velocity of gas, [m/s]

**Notes**

Model is rearranged to be explicit in terms of saltation velocity internally.

**References**

[1], [2], [3], [4], [5]

**Examples**

```python
>>> Schade(mp=1., rhop=1000., dp=1E-3, rhog=1.2, D=0.1)
13.697415809497912
```

`fluids.saltation.Weber_saltation(mp, rhop, dp, rhog, D, Vterminal=4)`
Calculates saltation velocity of the gas for pneumatic conveying, according to [1] as described in [2], [3], [4], and [5].

If \( V_{terminal} \) is under 3 m/s, use equation 1; otherwise, equation 2.

\[ F_{rs} = \left( 7 + \frac{8}{3} V_{terminal} \right) \mu^{0.25} \left( \frac{d_p}{D} \right)^{0.1} \]
Fluids Documentation, Release 0.1

\[ Fr_s = 15\mu^{0.25} \left( \frac{d_p}{D} \right)^{0.1} \]

\[ Fr_s = \frac{V_{salt}}{\sqrt{gD}} \]

\[ \mu = \frac{m_p}{\frac{\pi}{4} D^2 V \rho_f} \]

Parameters

- **mp** [float] Solid mass flow rate, [kg/s]
- **rhop** [float] Particle density, [kg/m^3]
- **dp** [float] Particle diameter, [m]
- **rhog** [float] Gas density, [kg/m^3]
- **D** [float] Diameter of pipe, [m]
- **Vterminal** [float] Terminal velocity of particle settling in gas, [m/s]

Returns

- **V** [float] Saltation velocity of gas, [m/s]

Notes

Model is rearranged to be explicit in terms of saltation velocity internally.

References

[1], [2], [3], [4], [5]

Examples

Examples are only a self-test.

```python
>>> Weber_saltation(mp=1, rhop=1000., dp=1E-3, rhog=1.2, D=0.1, Vterminal=4)
15.227445436331474
```

**fluids.saltation.Geldart_Ling** (mp, rhog, D, mug)

Calculates saltation velocity of the gas for pneumatic conveying, according to [1] as described in [2] and [3].

if \( G_s/D < 47000 \), use equation 1, otherwise use equation 2.

\[ V_{salt} = 1.5G_s^{0.456}D^{-0.01}\mu^{0.055}\rho_f^{-0.42} \]

\[ V_{salt} = 8.7G_s^{0.302}D^{0.153}\mu^{0.055}\rho_f^{-0.42} \]

\[ Fr_s = 15\mu^{0.25} \left( \frac{d_p}{D} \right)^{0.1} \]

\[ Fr_s = \frac{V_{salt}}{\sqrt{gD}} \]

\[ \mu = \frac{m_p}{\frac{\pi}{4} D^2 V \rho_f} \]

\[ G_s = \frac{m_p}{A} \]
Parameters

- **mp** [float] Solid mass flow rate, [kg/s]
- **rhog** [float] Gas density, [kg/m$^3$]
- **D** [float] Diameter of pipe, [m]
- **mug** [float] Gas viscosity, [Pa*s]

Returns

- **V** [float] Saltation velocity of gas, [m/s]

Notes

Model is rearranged to be explicit in terms of saltation velocity internally.

References

[1], [2], [3]

Examples

```python
>>> Geldart_Ling(1., 1.2, 0.1, 2E-5)
7.467495862402707
```

### 2.22 Two phase flow (fluids.two_phase)

**fluids.two_phase.two_phase_dp**

```python
fluids.two_phase.two_phase_dp(m, x, rhol, D, L=1, rhog=None, mul=None, mug=None, sigma=None, P=None, Pc=None, roughness=0, angle=0, Method=None, AvailableMethods=False)
```

This function handles calculation of two-phase liquid-gas pressure drop for flow inside channels. 23 calculation methods are available, with varying input requirements. A correlation will be automatically selected if none is specified. The full list of correlation can be obtained with the `AvailableMethods` flag.

If no correlation is selected, the following rules are used, with the earlier options attempted first:

- If rhog, mul, mug, and sigma are specified, use the Kim_Mudawar model
- If rhog, mul, and mug are specified, use the Chisholm model
- If mul, P, and Pc are specified, use the Zhang_Webb model
- If rhog and sigma are specified, use the Lombardi_Pedrocchi model

Parameters

- **m** [float] Mass flow rate of fluid, [kg/s]
- **x** [float] Quality of fluid, [-]
- **rhol** [float] Liquid density, [kg/m$^3$]
- **D** [float] Diameter of pipe, [m]
- **L** [float, optional] Length of pipe, [m]
**Fluids Documentation, Release 0.1**

```
**rhog** [float, optional] Gas density, [kg/m³]

**mul** [float, optional] Viscosity of liquid, [Pa*s]

**mug** [float, optional] Viscosity of gas, [Pa*s]

**sigma** [float, optional] Surface tension, [N/m]

**P** [float, optional] Pressure of fluid, [Pa]

**Pc** [float, optional] Critical pressure of fluid, [Pa]

**roughness** [float, optional] Roughness of pipe for use in calculating friction factor, [m]

**angle** [float, optional] The angle of the pipe with respect to the horizontal, [degrees]

**Returns**

**dP** [float] Pressure drop of the two-phase flow, [Pa]

**methods** [list, only returned if AvailableMethods == True] List of methods which can be used to calculate two-phase pressure drop with the given inputs.

**Other Parameters**

**Method** [string, optional] A string of the function name to use, as in the dictionary `two_phase_correlations`.

**AvailableMethods** [bool, optional] If True, function will consider which methods which can be used to calculate two-phase pressure drop with the given inputs and return them as a list instead of performing a calculation.

**Notes**

These functions may be integrated over, with properties recalculated as the fluid’s quality changes.

This model considers only the frictional pressure drop, not that due to gravity or acceleration.

**Examples**

```python
>>> two_phase_dP(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
... sigma=0.0487, D=0.05, L=1)
840.4137796786074
```

**fluids.two_phase.two_phase_dP_acceleration**(m, D, xi, xo, alpha_i, alpha_o, rho_li, rho_gi, rho_lo=None, rho_go=None)

This function handles calculation of two-phase liquid-gas pressure drop due to acceleration for flow inside channels. This is a discrete calculation for a segment with a known difference in quality (and ideally known inlet and outlet pressures so density dependence can be included).

\[
\Delta P_{acc} = G^2 \left\{ \left[ \frac{(1-x_o)^2}{\rho_{l,o}(1-\alpha_o)} + \frac{x_o^2}{\rho_{g,o}\alpha_o} \right] - \left[ \frac{(1-x_i)^2}{\rho_{l,i}(1-\alpha_i)} + \frac{x_i^2}{\rho_{g,i}\alpha_i} \right] \right\}
\]

**Parameters**

**m** [float] Mass flow rate of fluid, [kg/s]

**D** [float] Diameter of pipe, [m]

**xi** [float] Quality of fluid at inlet, [-]

**xo** [float] Quality of fluid at outlet, [-]
alpha_i [float] Void fraction at inlet (area of gas / total area of channel), [-]
alpha_o [float] Void fraction at outlet (area of gas / total area of channel), [-]
rho_li [float] Liquid phase density at inlet, [kg/m^3]
rho_gi [float] Gas phase density at inlet, [kg/m^3]
rho_lo [float, optional] Liquid phase density at outlet, [kg/m^3]
rho_go [float, optional] Gas phase density at outlet, [kg/m^3]

Returns

dP [float] Acceleration component of pressure drop for two-phase flow, [Pa]

Notes

The use of different gas and liquid phase densities at the inlet and outlet is optional; the outlet densities conditions will be assumed to be those of the inlet if they are not specified.

There is a continuous variant of this method which can be integrated over, at the expense of a speed. The differential form of this is as follows ([1], [3]):

\[-(\frac{dP}{dz})_{acc} = G^2 \frac{dz}{dz} \left[ \frac{(1-x)^2}{\rho_l(1-\alpha)} + \frac{x^2}{\rho_g\alpha} \right] \]

References

[1], [2], [3]

Examples

```python
>>> two_phase_dP_acceleration(m=1, D=0.1, xi=0.372, xo=0.557, rho_li=827.1,
... rho_gi=3.919, alpha_i=0.992, alpha_o=0.996)
706.8560377214725
```

```
fluids.two_phase.two_phase_dp_dz_acceleration(m, D, x, rho_l, rho_g, dv_dP_l, dv_dP_g, 
dx_dP, dp_dL, dA_dL)
```

This function handles calculation of two-phase liquid-gas pressure drop due to acceleration for flow inside channels. This is a continuous calculation, providing the differential in pressure per unit length and should be called as part of an integration routine ([1], [2], [3]).

\[-\left( \frac{\partial P}{\partial L} \right)_A = G^2 \left( \frac{1}{\rho_g} \frac{1}{\rho_l} \right) \partial P \partial x + \partial P \partial L \left[ x \frac{\partial (1/\rho_g)}{\partial P} + (1-x) \frac{\partial (1/\rho_l)}{\partial P} \right] \right) - \frac{G^2}{\rho_{hom}} \frac{1}{A} \frac{\partial A}{\partial L} \]

Parameters

**m**  [float] Mass flow rate of fluid, [kg/s]

**D**  [float] Diameter of pipe, [m]

**x**  [float] Quality of fluid [-]

**rho_l** [float] Liquid density, [kg/m^3]

**rho_g** [float] Gas density, [kg/m^3]

**dv_dP_l** [float] Derivative of mass specific volume of the liquid phase with respect to pressure, [m^3/(kg*Pa)]
\(dv_{\text{d}P_{\text{g}}}\) [float] Derivative of mass specific volume of the gas phase with respect to pressure, \([\text{m}^3/(\text{kg}\cdot\text{Pa})]\)

\(dx_{\text{d}P}\) [float] Derivative of mass quality of the two-phase fluid with respect to pressure (numerical derivatives may be convenient for this), \([1/\text{Pa}]\)

\(dP_{\text{d}L}\) [float] Pressure drop per unit length of pipe, \([\text{Pa}/\text{m}]\)

\(dA_{\text{d}L}\) [float] Change in area of pipe per unit length of pipe, \([\text{m}^2/\text{m}]\)

Returns

\(dP_{\text{d}z}\) [float] Acceleration component of pressure drop for two-phase flow, \([\text{Pa}/\text{m}]\)

Notes

This calculation has the homogeneous model built in to it as its derivation is shown in [1]. The discrete calculation is more flexible as different void fractions may be used.

References

[1], [2], [3]

Examples

```python
>>> two_phase_dP_dz_acceleration(m=1, D=0.1, x=0.372, rhol=827.1,
... rhog=3.919, dv_dP_l=-5e-12, dv_dP_g=-4e-7, dx_dP=-2e-7, dP_dL=120.0,
... dA_dL=0.0001)
20.137876617489034
```

\(\text{fluids}.\text{two}_\text{phase}\.\text{two}_\text{phase}_\text{dP}_\text{gravitational}\)(angle, z, alpha_i, rho_li, rho_gi, alpha_o=None, rho_lo=None, rho_go=None, g=9.80665)

This function handles calculation of two-phase liquid-gas pressure drop due to gravitation for flow inside channels. This is a discrete calculation for a segment with a known difference in elevation (and ideally known inlet and outlet pressures so density dependence can be included).

\[-ΔP_{\text{grav}} = g \sin θz \left\{ \left[ α_oρ_{g,o} + (1 - α_o)ρ_{l,o}\right] + \left[ α_iρ_{g,i} + (1 - α_i)ρ_{l,i}\right] \right\}/2 \]

Parameters

- **angle** [float] The angle of the pipe with respect to the horizontal, [degrees]
- **z** [float] The total length of the pipe, [m]
- **alpha_i** [float] Void fraction at inlet (area of gas / total area of channel), [-]
- **rho_li** [float] Liquid phase density at inlet, [kg/m^3]
- **rho_gi** [float] Gas phase density at inlet, [kg/m^3]
- **alpha_o** [float, optional] Void fraction at outlet (area of gas / total area of channel), [-]
- **rho_lo** [float, optional] Liquid phase density at outlet, [kg/m^3]
- **rho_go** [float, optional] Gas phase density at outlet, [kg/m^3]
- **g** [float, optional] Acceleration due to gravity, [m/s^2]
Returns

\( dP \) [float] Gravitational component of pressure drop for two-phase flow, [Pa]

Notes

The use of different gas and liquid phase densities and void fraction at the inlet and outlet is optional; the outlet densities and void fraction will be assumed to be those of the inlet if they are not specified. This does not add much accuracy.

There is a continuous variant of this method which can be integrated over, at the expense of a speed. The differential form of this is as follows ([1], [2]):

\[-\left( \frac{dP}{dz} \right)_{grav} = [\alpha \rho_g + (1 - \alpha) \rho_l] g \sin \theta\]

References

[1], [2], [3]

Examples

Example calculation, page 13-2 from [3]:

```python
>>> two_phase_dP_gravitational(angle=90, z=2, alpha_i=0.9685, rho_li=1518.,
... rho_gi=2.6)
987.237416829999
```

The same calculation, but using average inlet and outlet conditions:

```python
>>> two_phase_dP_gravitational(angle=90, z=2, alpha_i=0.9685, rho_li=1518.,
... rho_gi=2.6, alpha_o=0.968, rho_lo=1517.9, rho_go=2.59)
994.5416058829999
```

\( \text{fluids.two_phase.} \text{two_phase_dP_dz_gravitational} \) \( (\text{angle, alpha, rhol, rhog, g=9.80665}) \)

This function handles calculation of two-phase liquid-gas pressure drop due to gravitation for flow inside channels. This is a differential calculation for a segment with an infinitesimal difference in elevation for use in performing integration over a pipe as shown in [1] and [2].

\[-\left( \frac{dP}{dz} \right)_{grav} = [\alpha \rho_g + (1 - \alpha) \rho_l] g \sin \theta\]

Parameters

- **angle** [float] The angle of the pipe with respect to the horizontal, [degrees]
- **alpha** [float] Void fraction (area of gas / total area of channel), [-]
- **rhol** [float] Liquid phase density, [kg/m^3]
- **rhog** [float] Gas phase density, [kg/m^3]
- **g** [float, optional] Acceleration due to gravity, [m/s^2]

Returns

\( dP_dz \) [float] Gravitational component of pressure drop for two-phase flow, [Pa/m]
References

[1], [2]

Examples

```python
>>> two_phase_dP_dz_gravitational(angle=90, alpha=0.9685, rhol=1518, ...
... rhog=2.6)
493.6187084149995
```

**fluids.two_phase.Beggs_Brill**

(\(m, x, rhol, rhog, mul, mug, sigma, P, D, angle, roughness=0.0, L=1.0, g=9.80665, acceleration=True\))

Calculates the two-phase pressure drop according to the Beggs-Brill correlation ([1], [2], [3]).

**Parameters**

- **m** [float] Mass flow rate of fluid, [kg/s]
- **x** [float] Mass quality of fluid, [-]
- **rhol** [float] Liquid density, [kg/m^3]
- **rhog** [float] Gas density, [kg/m^3]
- **mul** [float] Viscosity of liquid, [Pa*s]
- **mug** [float] Viscosity of gas, [Pa*s]
- **sigma** [float] Surface tension, [N/m]
- **P** [float] Pressure of fluid (used only if \(acceleration=True\)), [Pa]
- **D** [float] Diameter of pipe, [m]
- **angle** [float] The angle of the pipe with respect to the horizontal, [degrees]
- **roughness** [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- **L** [float, optional] Length of pipe, [m]
- **g** [float, optional] Acceleration due to gravity, [m/s^2]
- **acceleration** [bool] Whether or not to include the original acceleration component, [-]

**Returns**

- **dP** [float] Pressure drop of the two-phase flow, [Pa]

**Notes**

The original acceleration formula is fairly primitive and normally neglected. The model was developed assuming smooth pipe, so leaving **roughness** to zero may be wise.

Note this is a “mechanistic” pressure drop model - the gravitational pressure drop cannot be separated from the frictional pressure drop.

**References**

[1], [2], [3]
Examples

```python
>>> Beggs_Brill(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
... sigma=0.0487, P=1E7, D=0.05, angle=0, roughness=0, L=1)
686.9724506803472
```

`fluids.two_phase.Lockhart_Martinelli(m, x, rhol, rhog, mul, mug, D, L=None, Re_c=2000)`

Calculates two-phase pressure drop with the Lockhart and Martinelli (1949) correlation as presented in non-graphical form by Chisholm (1967).

\[
\Delta P = \Delta P_l \phi_l^2
\]

\[
\phi_l^2 = 1 + \frac{C}{X} + \frac{1}{X^2}
\]

\[
X^2 = \frac{\Delta P_l}{\Delta P_g}
\]

<table>
<thead>
<tr>
<th>Liquid</th>
<th>Gas</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turbulent</td>
<td>Turbulent</td>
<td>20</td>
</tr>
<tr>
<td>Laminar</td>
<td>Turbulent</td>
<td>12</td>
</tr>
<tr>
<td>Turbulent</td>
<td>Laminar</td>
<td>10</td>
</tr>
<tr>
<td>Laminar</td>
<td>Laminar</td>
<td>5</td>
</tr>
</tbody>
</table>

This model has its own friction factor calculations, to be consistent with its Reynolds number transition and the procedure specified in the original work. The equation \(64/\text{Re} \) is used up to \( \text{Re}_c \), and above it the Blasius equation is used as follows:

\[
f_d = \frac{0.184}{\text{Re}^{0.2}}
\]

Parameters

- `m` [float] Mass flow rate of fluid, [kg/s]
- `x` [float] Quality of fluid, [-]
- `rhol` [float] Liquid density, [kg/m^3]
- `rhog` [float] Gas density, [kg/m^3]
- `mul` [float] Viscosity of liquid, [Pa*s]
- `mug` [float] Viscosity of gas, [Pa*s]
- `D` [float] Diameter of pipe, [m]
- `L` [float, optional] Length of pipe, [m]
- `Re_c` [float, optional] Transition Reynolds number, used to decide which friction factor equation to use and which C value to use from the table above.

Returns

- `dP` [float] Pressure drop of the two-phase flow, [Pa]
**Notes**

Developed for horizontal flow. Very popular. Many implementations of this model assume turbulent-turbulent flow.

The original model proposed that the transition Reynolds number was 1000 for laminar flow, and 2000 for turbulent flow; it proposed no model for Re_l < 1000 and Re_g between 1000 and 2000 and also Re_g < 1000 and Re_l between 1000 and 2000.

No correction is available in this model for rough pipe.

[3] examined the original data in [1] again, and fit more curves to the data, separating them into different flow regimes. There were 229 datum in the turbulent-turbulent regime, 9 in the turbulent-laminar regime, 339 in the laminar-turbulent regime, and 42 in the laminar-laminar regime. Errors from [3]’s curves were 13.4%, 3.5%, 14.3%, and 12.0% for the above regimes, respectively. [2]’s fits provide further error.

**References**

[1], [2], [3], [4]

**Examples**

```python
>>> Lockhart_Martinelli(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, ...
... mug=14E-6, D=0.05, L=1)
716.4695654888484
```

`fluids.two_phase.Friedel(m, x, rhol, rhog, mul, mug, sigma, D, roughness=0, L=1)`

Calculates two-phase pressure drop with the Friedel correlation.

\[
\Delta P_{friction} = \Delta P_{io}\phi_{io}^2
\]

\[
\phi_{io}^2 = E + \frac{3.24FH}{Fr^{0.0454}We^{0.035}}
\]

\[
H = \left(\frac{\rho_l}{\rho_g}\right)^{0.91}\left(\frac{\mu_g}{\mu_l}\right)^{0.19}\left(1 - \frac{\mu_g}{\mu_l}\right)^{0.7}
\]

\[
F = x^{0.78}(1-x)^{0.224}
\]

\[
E = (1 - x)^2 + x^2 \left(\frac{\rho_l f_{d,go}}{\rho_g f_{d,lo}}\right)
\]

\[
F_r = \frac{G^2_{fp}}{gD\rho_H^2}
\]

\[
We = \frac{G^2_{fp}D}{\sigma\rho_H}
\]

\[
\rho_H = \left(\frac{x}{\rho_g} + \frac{1 - x}{\rho_l}\right)^{-1}
\]

**Parameters**

- **m** [float] Mass flow rate of fluid, [kg/s]
- **x** [float] Quality of fluid, [-]
- **rhol** [float] Liquid density, [kg/m^3]
**rhog** [float] Gas density, [kg/m$^3$]

**mul** [float] Viscosity of liquid, [Pa*s]

**mug** [float] Viscosity of gas, [Pa*s]

**sigma** [float] Surface tension, [N/m]

**D** [float] Diameter of pipe, [m]

**roughness** [float, optional] Roughness of pipe for use in calculating friction factor, [m]

**L** [float, optional] Length of pipe, [m]

**Returns**

**dP** [float] Pressure drop of the two-phase flow, [Pa]

**Notes**

Applicable to vertical upflow and horizontal flow. Known to work poorly when mul/mug > 1000. Gives mean errors on the order of 40%. Tested on data with diameters as small as 4 mm.

The power of 0.0454 is given as 0.045 in [2], [3], [4], and [5]; [6] and [2] give 0.0454 and [2] also gives a similar correlation said to be presented in [1], so it is believed this 0.0454 was the original power. [6] also gives an expression for friction factor claimed to be presented in [1]; it is not used here.

**References**

[1], [2], [3], [4], [5], [6]

**Examples**

Example 4 in [6]:

```python
>>> Friedel(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
     ... sigma=0.0487, D=0.05, roughness=0, L=1)
738.6500525002245
```

Calculates two-phase pressure drop with the Chisholm (1973) correlation from [1], also in [2] and [3].

\[
\frac{\Delta P_{tp}}{\Delta P_{lo}} = \phi_{ch}^2
\]

\[
\phi_{ch}^2 = 1 + (\Gamma^2 - 1) \left\{ B x^2 (2-n)/2 (1-x)(2-n)/2 + x^2-n \right\}
\]

\[
\Gamma^2 = \frac{(\Delta P/L)_{go}}{(\Delta P/L)_{lo}}
\]

For Gamma < 9.5:

\[
B = \frac{55}{G_{tp}^{0.3}} \text{ for } G_{tp} > 1900
\]

\[
B = \frac{2400}{G_{tp}} \text{ for } 500 < G_{tp} < 1900
\]
\[ B = 4.8 \text{ for } G_{tp} < 500 \]

For \(9.5 < \Gamma < 28\):

\[ B = \frac{520}{\Gamma^{0.5}} \text{ for } G_{tp} < 600 \]
\[ B = \frac{21}{\Gamma} \text{ for } G_{tp} > 600 \]

For \(\Gamma > 28\):

\[ B = \frac{15000}{\Gamma^{2} G_{tp}^{0.5}} \]

If `rough_correction` is True, the following correction to \(B\) is applied:

\[
\frac{B_{\text{rough}}}{B_{\text{smooth}}} = \left[ 0.5 \left( 1 + \left( \frac{\mu_g}{\mu_l} \right)^2 + 10^{-600e/D} \right) \right]^{0.25-n} \]

\[ n = \frac{\log \frac{f_{d,lo}}{f_{d,go}}}{\log \frac{Re_{go}}{Re_{lo}}} \]

**Parameters**

- \(m\) [float] Mass flow rate of fluid, [kg/s]
- \(x\) [float] Quality of fluid, [-]
- \(\text{rhol}\) [float] Liquid density, [kg/m^3]
- \(\text{rhog}\) [float] Gas density, [kg/m^3]
- \(\text{mul}\) [float] Viscosity of liquid, [Pa*s]
- \(\text{mug}\) [float] Viscosity of gas, [Pa*s]
- \(D\) [float] Diameter of pipe, [m]
- \(\text{roughness}\) [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- \(L\) [float, optional] Length of pipe, [m]
- `rough_correction` [bool, optional] Whether or not to use the roughness correction proposed in the 1968 version of the correlation

**Returns**

- \(dP\) [float] Pressure drop of the two-phase flow, [Pa]

**Notes**

Applicable for \(0 < x < 1\). \(n = 0.25\), the exponent in the Blassius equation. Originally developed for smooth pipes, a roughness correction is included as well from the Chisholm’s 1968 work [4]. Neither [2] nor [3] have any mention of the correction however.

**References**

[1], [2], [3], [4]
**Examples**

```python
>>> Chisholm(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6,
... mug=14E-6, D=0.05, roughness=0, L=1)
1084.1489922923738
```

`fluids.two_phase.Kim_Mudawar(m, x, rhol, rhog, mul, mug, sigma, D, L=1)`

Calculates two-phase pressure drop with the Kim and Mudawar (2012) correlation as in [1], also presented in [2].

\[
\Delta P = \Delta P_l \phi_i^2
\]

\[
\phi_i^2 = 1 + \frac{C}{X} + \frac{1}{X^2}
\]

\[
X^2 = \frac{\Delta P_l}{\Delta P_g}
\]

For turbulent liquid, turbulent gas:

\[
C = 0.39 \mathcal{R}_e^{0.03} \mathcal{S}_{ug}^{0.10} \left( \frac{\rho_l}{\rho_g} \right)^{0.35}
\]

For turbulent liquid, laminar gas:

\[
C = 8.7 \times 10^{-4} \mathcal{R}_e^{0.17} \mathcal{S}_{ug}^{0.50} \left( \frac{\rho_l}{\rho_g} \right)^{0.14}
\]

For laminar liquid, turbulent gas:

\[
C = 0.0015 \mathcal{R}_e^{0.59} \mathcal{S}_{ug}^{0.19} \left( \frac{\rho_l}{\rho_g} \right)^{0.36}
\]

For laminar liquid, laminar gas:

\[
C = 3.5 \times 10^{-5} \mathcal{R}_e^{0.44} \mathcal{S}_{ug}^{0.50} \left( \frac{\rho_l}{\rho_g} \right)^{0.48}
\]

This model has its own friction factor calculations, to be consistent with its Reynolds number transition. As their model was regressed with these equations, more error is obtained when using any other friction factor calculation. The laminar equation \(64/\mathcal{R}e\) is used up to \(\mathcal{R}e=2000\), then the Blasius equation with a coefficient of 0.316, and above \(\mathcal{R}e = 20000\),

\[
f_d = \frac{0.184}{\mathcal{R}e^{0.2}}
\]

**Parameters**

- `m` [float] Mass flow rate of fluid, [kg/s]
- `x` [float] Quality of fluid, [-]
- `rhol` [float] Liquid density, [kg/m³]
- `rhog` [float] Gas density, [kg/m³]
- `mul` [float] Viscosity of liquid, [Pa*s]
- `mug` [float] Viscosity of gas, [Pa*s]
- `sigma` [float] Surface tension, [N/m]
**D** [float] Diameter of pipe, [m]

**L** [float, optional] Length of pipe, [m]

**Returns**

**dP** [float] Pressure drop of the two-phase flow, [Pa]

**Notes**

The critical Reynolds number in this model is 2000, with a Reynolds number definition using actual liquid and gas flows. This model also requires liquid-only Reynolds number to be calculated.

No attempt to incorporate roughness into the model was made in [1].

The model was developed with hydraulic diameter from 0.0695 to 6.22 mm, mass velocities 4 to 8528 kg/m^2/s, flow qualities from 0 to 1, reduced pressures from 0.0052 to 0.91, superficial liquid Reynolds numbers up to 79202, superficial gas Reynolds numbers up to 253810, liquid-only Reynolds numbers up to 89798, 7115 data points from 36 sources and working fluids air, CO₂, N₂, water, ethanol, R12, R22, R134a, R236ea, R245fa, R404A, R407C, propane, methane, and ammonia.

**References**

[1], [2]

**Examples**

```python
>>> Kim_Mudawar(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
... sigma=0.0487, D=0.05, L=1)
840.4137796786074
```

`fluids.two_phase.Baroczy_Chisholm(m, x, rhol, rhog, mul, mug, D, roughness=0, L=1)`

Calculates two-phase pressure drop with the Baroczy (1966) model. It was presented in graphical form originally; Chisholm (1973) made the correlation non-graphical. The model is also shown in [3].

\[
\frac{\Delta P_{tp}}{\Delta P_{lo}} = \phi_{ch}^2
\]

\[
\phi_{ch}^2 = 1 + (\Gamma^2 - 1) \left\{ B x^{(2-n)/2} (1 - x)^{(2-n)/2} + x^{2-n} \right\}
\]

\[
\Gamma^2 = \left( \frac{\Delta P}{L} \right)_{go} \left( \frac{\Delta P}{L} \right)_{lo}
\]

For Gamma < 9.5:

\[
B = 55 \frac{G^{0.5}}{\Gamma_{tp}^{1.5}}
\]

For 9.5 < Gamma < 28:

\[
B = 520 \frac{G^{0.5}}{\Gamma_{tp}^{1.5}}
\]

For Gamma > 28:

\[
B = 15000 \frac{G^{0.5}}{\Gamma_{tp}^{1.5}}
\]
Parameters

- \(m\) [float] Mass flow rate of fluid, [kg/s]
- \(x\) [float] Quality of fluid, [-]
- \(rhol\) [float] Liquid density, [kg/m\(^3\)]
- \(rhog\) [float] Gas density, [kg/m\(^3\)]
- \(mul\) [float] Viscosity of liquid, [Pa*s]
- \(mug\) [float] Viscosity of gas, [Pa*s]
- \(D\) [float] Diameter of pipe, [m]
- \(\text{roughness}\) [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- \(L\) [float, optional] Length of pipe, [m]

Returns

- \(dP\) [float] Pressure drop of the two-phase flow, [Pa]

Notes

Applicable for \(0 < x < 1\). \(n = 0.25\), the exponent in the Blassius equation. The \texttt{Chisholm_1973} function should be used in preference to this.

References

[1], [2], [3]

Examples

```python
>>> Baroczy_Chisholm(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6,
... mug=14E-6, D=0.05, roughness=0, L=1)
1084.1489922923738
```

\texttt{fluids.two\_phase.Theissing}(m, x, rhol, rhog, mul, mug, D, \text{roughness}=0, \text{L}=1)

Calculates two-phase pressure drop with the Theissing (1980) correlation as shown in [2] and [3].

\[
\Delta P_{tp} = \left[\Delta P_{lo}^{1/n_1} (1 - x)^{1/\epsilon} + \Delta P_{go}^{1/(n_1 + n_2)} x^{1/\epsilon}\right]^{n_2}
\]

\[
\epsilon = 3 - 2 \left(\frac{2 \sqrt{\rho_l/\rho_g}}{1 + \rho_l/\rho_g}\right)^{0.7/n}
\]

\[
n_1 = \frac{n_1 + n_2 (\Delta P_g/\Delta P_l)^{0.1}}{1 + (\Delta P_g/\Delta P_l)^{0.1}}
\]

\[
n_2 = \frac{\ln (\Delta P_l/\Delta P_{lo})}{\ln (1 - x)}
\]

Parameters

- \(m\) [float] Mass flow rate of fluid, [kg/s]
x [float] Quality of fluid, [-]
rhol [float] Liquid density, [kg/m^3]
rhog [float] Gas density, [kg/m^3]
mul [float] Viscosity of liquid, [Pa*s]
mug [float] Viscosity of gas, [Pa*s]
D [float] Diameter of pipe, [m]
roughness [float, optional] Roughness of pipe for use in calculating friction factor, [m]
L [float, optional] Length of pipe, [m]

Returns
dP [float] Pressure drop of the two-phase flow, [Pa]

Notes
Applicable for 0 < x < 1. Notable, as it can be used for two-phase liquid-liquid flow as well as liquid-gas flow.

References
[1], [2], [3]

Examples
```python
>>> Theissing(m=0.6, x=.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
... D=0.05, roughness=0, L=1)
497.6156370699528
```

fluids.two_phase.Muller_Steinhagen_Heck(m, x, rhol, rhog, mul, mug, D, roughness=0, L=1)
Calculates two-phase pressure drop with the Muller-Steinhagen and Heck (1986) correlation from [1], also in [2] and [3].

\[ \Delta P_{tp} = G_{MSH}(1 - x)^{1/3} + \Delta P_{go}x^{3} \]
\[ G_{MSH} = \Delta P_{lo} + 2[\Delta P_{go} - \Delta P_{lo}]x \]

Parameters
m [float] Mass flow rate of fluid, [kg/s]
x [float] Quality of fluid, [-]
rhol [float] Liquid density, [kg/m^3]
rhog [float] Gas density, [kg/m^3]
mul [float] Viscosity of liquid, [Pa*s]
mug [float] Viscosity of gas, [Pa*s]
D [float] Diameter of pipe, [m]
roughness [float, optional] Roughness of pipe for use in calculating friction factor, [m]
L [float, optional] Length of pipe, [m]
Returns

\( dP \) [float] Pressure drop of the two-phase flow, [Pa]

Notes

Applicable for \( 0 < x < 1 \). Developed to be easily integrated. The contribution of each term to the overall pressure drop can be understood in this model.

References

[1], [2], [3]

Examples

```python
>>> Muller_Steinhagen_Heck(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6,
...    mug=14E-6, D=0.05, roughness=0, L=1)
793.4465457435081
```

\( \Delta P \) friction = \( \Delta P \) gd \( \phi \) 2

\( \phi \) gd = 1 + \left( \frac{dP}{dL} \right) Fr \left[ \frac{\rho_l}{\mu_l} \left( \frac{\mu_l}{\mu_g} \right)^{0.25} - 1 \right]

\left( \frac{dP}{dL} \right) Fr = f Fr \left[ x + 4 \left( x^{1.8} - x^{10} f Fr^{0.5} \right) \right]

\( f Fr = Fr^0.3 + 0.0055 \left( \ln \frac{1}{Fr} \right)^2 \)

\( Fr l = \frac{G_{lp}^2}{g D \rho_l^2} \)

Parameters

- \( m \) [float] Mass flow rate of fluid, [kg/s]
- \( x \) [float] Quality of fluid, [-]
- \( rhol \) [float] Liquid density, [kg/m^3]
- \( rhog \) [float] Gas density, [kg/m^3]
- \( mul \) [float] Viscosity of liquid, [Pa*s]
- \( mug \) [float] Viscosity of gas, [Pa*s]
- \( D \) [float] Diameter of pipe, [m]
- \( roughness \) [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- \( L \) [float, optional] Length of pipe, [m]

Returns

- \( dP \) [float] Pressure drop of the two-phase flow, [Pa]
Notes

Developed for evaporators. Applicable from $0 < x < 1$.

In the model, if $Fr_l$ is more than 1, $f_Fr$ is set to 1.

References

[1], [2], [3], [4]

Examples

```python
>>> Gronnerud(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6, ...
... D=0.05, roughness=0, L=1)
384.1254114447411
```

```
fluids.two_phase.Lombardi_Pedrocchi(m, x, rhol, rhog, sigma, D, L=1)
Calculates two-phase pressure drop with the Lombardi-Pedrocchi (1972) correlation from [1] as shown in [2] and [3].

\[
\Delta P_{tp} = 0.83C_{tp}^{1.4}\sigma^{0.4}L
\]

Parameters

- **m** [float] Mass flow rate of fluid, [kg/s]
- **x** [float] Quality of fluid, [-]
- **rhol** [float] Liquid density, [kg/m^3]
- **rhog** [float] Gas density, [kg/m^3]
- **sigma** [float] Surface tension, [N/m]
- **D** [float] Diameter of pipe, [m]
- **L** [float, optional] Length of pipe, [m]

Returns

- **dP** [float] Pressure drop of the two-phase flow, [Pa]

Notes

This is a purely empirical method. [3] presents a review of this and other correlations. It did not perform best, but there were also correlations worse than it.

References

[1], [2], [3]

Examples
fluids.two_phase.Jung_Radermacher \((m, x, \text{rhol}, \text{rhog}, \text{mul}, \text{mug}, D, \text{roughness}=0, L=1)\)

Calculates two-phase pressure drop with the Jung-Radermacher (1989) correlation, also shown in [2] and [3].

\[
\frac{\Delta P_{tp}}{\Delta P_{lo}} = \phi_{tp}^2
\]

\[
\phi_{tp}^2 = 12.82X_{tt}^{-1.47}(1 - x)^{1.8}
\]

**Parameters**

- \(m\) [float] Mass flow rate of fluid, [kg/s]
- \(x\) [float] Quality of fluid, [-]
- \(\text{rhol}\) [float] Liquid density, [kg/m\(^3\)]
- \(\text{rhog}\) [float] Gas density, [kg/m\(^3\)]
- \(\text{mul}\) [float] Viscosity of liquid, [Pa\(\cdot\)s]
- \(\text{mug}\) [float] Viscosity of gas, [Pa\(\cdot\)s]
- \(D\) [float] Diameter of pipe, [m]
- \(\text{roughness}\) [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- \(L\) [float, optional] Length of pipe, [m]

**Returns**

- \(\Delta P\) [float] Pressure drop of the two-phase flow, [Pa]

**Notes**

Applicable for \(0 < x < 1\). Developed for the annular flow regime in turbulent-turbulent flow.

**References**

[1], [2], [3]

**Examples**

```python
>>> Jung_Radermacher(m=0.6, x=0.1, rhol=915., rhog=2.67, sigma=0.045, ...
... D=0.05, L=1)
1567.328374498781
```

**fluids.two_phase.Tran** \((m, x, \text{rhol}, \text{rhog}, \text{mul}, \text{mug}, \text{sigma}, D, \text{roughness}=0, L=1)\)

Calculates two-phase pressure drop with the Tran (2000) correlation, also shown in [2] and [3].

\[
\Delta P = dP_{lo}\phi_{lo}^2
\]

\[
\phi_{lo}^2 = 1 + (4.3\Gamma^2 - 1)[\text{Co} \cdot x^{0.875}(1 - x)^{0.875} + x^{1.75}]
\]

\[
\Gamma^2 = \left(\frac{\Delta P}{\Delta P_{lo}}\right)_{lo}
\]
Parameters

- \( m \) [float] Mass flow rate of fluid, [kg/s]
- \( x \) [float] Quality of fluid, [-]
- \( rhol \) [float] Liquid density, [kg/m^3]
- \( rhog \) [float] Gas density, [kg/m^3]
- \( mul \) [float] Viscosity of liquid, [Pa*s]
- \( mug \) [float] Viscosity of gas, [Pa*s]
- \( sigma \) [float] Surface tension, [N/m]
- \( D \) [float] Diameter of pipe, [m]
- \( \text{roughness} \) [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- \( L \) [float, optional] Length of pipe, [m]

Returns

- \( dP \) [float] Pressure drop of the two-phase flow, [Pa]

Notes

Developed for boiling refrigerants in channels with hydraulic diameters of 2.4 mm to 2.92 mm.

References

[1], [2], [3]

Examples

```python
>>> Tran(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
... sigma=0.0487, D=0.05, roughness=0, L=1)
423.2563312951232
```

`fluids.two_phase.Chen_Friedel(m, x, rhol, rhog, mul, mug, sigma, D, roughness=0, L=1)`

Calculates two-phase pressure drop with the Chen modification of the Friedel correlation, as given in [1] and also shown in [2] and [3].

\[
\Delta P = \Delta P_{Friedel} \Omega
\]

For \( Bo < 2.5 \):

\[
\Omega = \frac{0.0333Re_{to}^{0.45}}{Re_{g}^{0.09}(1 + 0.4 \exp(-Bo))}
\]

For \( Bo \geq 2.5 \):

\[
\Omega = \frac{We^{0.2}}{2.5 + 0.06Bo}
\]

Parameters

- \( m \) [float] Mass flow rate of fluid, [kg/s]
x [float] Quality of fluid, [-]
rhol [float] Liquid density, [kg/m^3]
rhog [float] Gas density, [kg/m^3]
mul [float] Viscosity of liquid, [Pa*s]
mug [float] Viscosity of gas, [Pa*s]
sigma [float] Surface tension, [N/m]
D [float] Diameter of pipe, [m]

**Returns**

dP [float] Pressure drop of the two-phase flow, [Pa]

**Notes**

Applicable ONLY to mini/microchannels; yields drastically too low pressure drops for larger channels. For more details, see the Friedel correlation.

It is not explicitly stated in [1] how to calculate the liquid mixture density for use in calculation of Weber number; the homogeneous model is assumed as it is used in the Friedel model.

The bond number used here is 1/4 the normal value, i.e.:

\[
B_0 = \frac{g(\rho_l - \rho_g)D^2}{4\sigma}
\]

**References**

[1], [2], [3]

**Examples**

```python
>>> Chen_Friedel(m=.0005, x=0.9, rhol=950., rhog=1.4, mul=1E-3, mug=1E-5,
... sigma=0.02, D=0.003, roughness=0, L=1)
6249.247540588871
```

**fluids.two_phase.Zhang_Webb (m, x, rhol, mul, P, Pc, D, roughness=0, L=1)**

Calculates two-phase pressure drop with the Zhang-Webb (2001) correlation as shown in [1] and also given in [2].

\[
\phi_{2o}^2 = (1 - x)^2 + 2.87x^2 \left( \frac{P}{P_c} \right)^{-1} + 1.68x^{0.8}(1 - x)^{0.25} \left( \frac{P}{P_c} \right)^{-1.64}
\]

**Parameters**

- m [float] Mass flow rate of fluid, [kg/s]
- x [float] Quality of fluid, [-]
- rhol [float] Liquid density, [kg/m^3]
- mul [float] Viscosity of liquid, [Pa*s]
Fluids Documentation, Release 0.1

**P** [float] Pressure of fluid, [Pa]

**Pc** [float] Critical pressure of fluid, [Pa]

**D** [float] Diameter of pipe, [m]

**roughness** [float, optional] Roughness of pipe for use in calculating friction factor, [m]

**L** [float, optional] Length of pipe, [m]

**Returns**

**dP** [float] Pressure drop of the two-phase flow, [Pa]

**Notes**

Applicable for 0 < x < 1. Corresponding-states method developed with R-134A, R-22 and R-404A in tubes of hydraulic diameters of 2.13 mm, 6.25 mm, and 3.25 mm. For the author’s 119 data points, the mean deviation was 11.5%. Recommended for reduced pressures larger than 0.2 and tubes of diameter 1-7 mm.

Does not require known properties for the gas phase.

**References**

[1], [2]

**Examples**

```python
>>> Zhang_Webb(m=0.6, x=0.1, rhol=915., mul=180E-6, P=2E5, Pc=4055000, ...
... D=0.05, roughness=0, L=1)
712.099804205621
```

**fluids.two_phase.Xu_Fang** *(m, x, rhol, rho, mul, mug, sigma, D, roughness=0, L=1)*

Calculates two-phase pressure drop with the Xu and Fang (2013) correlation. Developed after a comprehensive review of available correlations, likely meaning it is quite accurate.

\[
\Delta P = \Delta P_{lo} \phi_{lo}^2
\]

\[
\phi_{lo}^2 = Y^2 x^3 + (1 - x^{2.59})^{0.632}[1 + 2x^{1.17}(Y^2 - 1) + 0.00775x^{-0.475} Fr_{tp}^{0.535} We_{tp}^{0.188}]
\]

\[
Y^2 = \frac{\Delta P_{go}}{\Delta P_{lo}}
\]

\[
Fr_{tp} = \frac{G_{tp}^2}{gD\rho_{tp}^2}
\]

\[
We_{tp} = \frac{G_{tp}^2 D}{\sigma\rho_{tp}}
\]

\[
\frac{1}{\rho_{tp}} = \frac{1 - x}{\rho_l} + \frac{x}{\rho_g}
\]

**Parameters**

**m** [float] Mass flow rate of fluid, [kg/s]

**x** [float] Quality of fluid, [-]

**rhol** [float] Liquid density, [kg/m^3]
**Parameters**

- `m` [float] Mass flow rate of fluid, [kg/s]
- `x` [float] Quality of fluid, [-]
- `rhol` [float] Liquid density, [kg/m^3]
- `rhog` [float] Gas density, [kg/m^3]
- `mul` [float] Viscosity of liquid, [Pa*s]
- `mug` [float] Viscosity of gas, [Pa*s]
- `D` [float] Diameter of pipe, [m]
- `roughness` [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- `L` [float, optional] Length of pipe, [m]

**Returns**

- `dP` [float] Pressure drop of the two-phase flow, [Pa]

**References**

[1]

**Examples**

```python
>>> Xu_Fang(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
   ... sigma=0.0487, D=0.05, roughness=0, L=1)
604.05632116267
```


\[
\Delta P = \Delta P_l \phi_l^2 \\
\phi_l^2 = X^{-1.9} \\
X = 18.65 \left( \frac{\rho_g}{\rho_l} \right)^{0.5} \left( \frac{1-x}{x} \right) \frac{Re_g}{Re_l^{0.5}}
\]

**Parameters**

- `m` [float] Mass flow rate of fluid, [kg/s]
- `x` [float] Quality of fluid, [-]
- `rhol` [float] Liquid density, [kg/m^3]
- `rhog` [float] Gas density, [kg/m^3]
- `mul` [float] Viscosity of liquid, [Pa*s]
- `mug` [float] Viscosity of gas, [Pa*s]
- `D` [float] Diameter of pipe, [m]
- `roughness` [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- `L` [float, optional] Length of pipe, [m]

**Returns**

- `dP` [float] Pressure drop of the two-phase flow, [Pa]
**References**

[1], [2], [3]

**Examples**

```python
>>> Yu_France(m=0.6, x=.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
... D=0.05, roughness=0, L=1)
1146.983322553957
```

`fluids.two_phase.Wang_Chiang_Lu(m, x, rhol, rhog, mul, mug, D, roughness=0, L=1)`

Calculates two-phase pressure drop with the Wang, Chiang, and Lu (1997) correlation given in [1] and reviewed in [2] and [3].

\[
\Delta P = \Delta P_g \phi_g^2
\]

\[
\phi_g^2 = 1 + 9.397X^{0.62} + 0.564X^{2.45} \text{ for } G \geq 200 \text{kg/m}^2/\text{s}
\]

\[
\phi_g^2 = 1 + CX + X^2 \text{ for lower mass fluxes}
\]

\[
C = 0.000004566X^{0.128}Re^{0.938} \left( \frac{\rho_l}{\rho_g} \right)^{-2.15} \left( \frac{\mu_l}{\mu_g} \right)^{5.1}
\]

\[
X^2 = \frac{\Delta P_l}{\Delta P_g}
\]

**Parameters**

- `m` [float] Mass flow rate of fluid, [kg/s]
- `x` [float] Quality of fluid, [-]
- `rhol` [float] Liquid density, [kg/m^3]
- `rhog` [float] Gas density, [kg/m^3]
- `mul` [float] Viscosity of liquid, [Pa*s]
- `mug` [float] Viscosity of gas, [Pa*s]
- `D` [float] Diameter of pipe, [m]
- `roughness` [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- `L` [float, optional] Length of pipe, [m]

**Returns**

- `dP` [float] Pressure drop of the two-phase flow, [Pa]

**References**

[1], [2], [3]
Examples

```python
>>> Wang_Chiang_Lu(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, 
... mug=14E-6, D=0.05, roughness=0, L=1)
448.29981978639154
```

```python
fluids.two_phase.Hwang_Kim(m, x, rhol, rhog, mul, mug, sigma, D, roughness=0, L=1)
```

Calculates two-phase pressure drop with the Hwang and Kim (2006) correlation as in [1], also presented in [2] and [3].

\[
\Delta P = \Delta P_l \phi_l^2
\]

\[
C = 0.227 Re_\theta^{0.452} X^{-0.32} C_0^{-0.82}
\]

\[
\phi_l^2 = 1 + \frac{C}{X} + \frac{1}{X^2}
\]

\[
X^2 = \frac{\Delta P_l}{\Delta P_g}
\]

Parameters

- **m** [float] Mass flow rate of fluid, [kg/s]
- **x** [float] Quality of fluid, [-]
- **rhol** [float] Liquid density, [kg/m^3]
- **rhog** [float] Gas density, [kg/m^3]
- **mul** [float] Viscosity of liquid, [Pa*s]
- **mug** [float] Viscosity of gas, [Pa*s]
- **sigma** [float] Surface tension, [N/m]
- **D** [float] Diameter of pipe, [m]
- **roughness** [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- **L** [float, optional] Length of pipe, [m]

Returns

- **dP** [float] Pressure drop of the two-phase flow, [Pa]

Notes

Developed with data for microtubes of diameter 0.244 mm and 0.792 mm only. Not likely to be suitable to larger diameters.

References

[1], [2], [3]
Examples

```python
>>> Hwang_Kim(m=0.0005, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
... sigma=0.0487, D=0.003, roughness=0, L=1)
798.302774184557
```

**fluids.two_phase.Zhang_Hibiki_Mishima**(*m*, *x*, *rhol*, *rhog*, *mul*, *mug*, *sigma*, *D*, *roughness=0, L=1, flowtype='adiabatic vapor')

Calculates two-phase pressure drop with the Zhang, Hibiki, Mishima and (2010) correlation as in [1], also presented in [2] and [3].

\[
\Delta P = \Delta P_l \phi^2
\]

\[
\phi^2 = 1 + \frac{C}{X} + \frac{1}{X^2}
\]

\[
X^2 = \frac{\Delta P_l}{\Delta P_g}
\]

For adiabatic liquid-vapor two-phase flow:

\[
C = 21[1 - \exp(-0.142/Co)]
\]

For adiabatic liquid-gas two-phase flow:

\[
C = 21[1 - \exp(-0.674/Co)]
\]

For flow boiling:

\[
C = 21[1 - \exp(-0.358/Co)]
\]

**Parameters**

- *m* [float] Mass flow rate of fluid, [kg/s]
- *x* [float] Quality of fluid, [-]
- *rhol* [float] Liquid density, [kg/m^3]
- *rhog* [float] Gas density, [kg/m^3]
- *mul* [float] Viscosity of liquid, [Pa*s]
- *mug* [float] Viscosity of gas, [Pa*s]
- *sigma* [float] Surface tension, [N/m]
- *D* [float] Diameter of pipe, [m]
- *roughness* [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- *L* [float, optional] Length of pipe, [m]
- *flowtype* [str] One of ‘adiabatic vapor’, ‘adiabatic gas’, or ‘flow boiling’

**Returns**

- *dP* [float] Pressure drop of the two-phase flow, [Pa]

**Notes**

Seems fairly reliable.
fluids.two_phase.Mishima_Hibiki \((m, x, rhol, rhog, mul, mug, sigma, D, roughness=0, L=1)\)
Calculates two-phase pressure drop with the Mishima and Hibiki (1996) correlation as in [1], also presented in [2] and [3].

\[
\Delta P = \Delta P_l \phi^2_l
\]
\[
C = 21[1 - \exp(-319D)]
\]
\[
\phi^2_l = 1 + \frac{C}{X} + \frac{1}{X^2}
\]
\[
X^2 = \frac{\Delta P_l}{\Delta P_g}
\]

**Parameters**
- \(m\) [float] Mass flow rate of fluid, [kg/s]
- \(x\) [float] Quality of fluid, [-]
- \(rhol\) [float] Liquid density, [kg/m^3]
- \(rhog\) [float] Gas density, [kg/m^3]
- \(mul\) [float] Viscosity of liquid, [Pa*s]
- \(mug\) [float] Viscosity of gas, [Pa*s]
- \(sigma\) [float] Surface tension, [N/m]
- \(D\) [float] Diameter of pipe, [m]
- \(roughness\) [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- \(L\) [float, optional] Length of pipe, [m]

**Returns**
- \(dP\) [float] Pressure drop of the two-phase flow, [Pa]

**References**
[1], [2], [3]
Fluids Documentation, Release 0.1

Examples

```python
>>> Mishima_Hibiki(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6,
...   mug=14E-6, sigma=0.0487, D=0.05, roughness=0, L=1)
732.4268200606265
```

**fluids.two_phase.Bankoff** *(m, x, rhol, rhog, mul, mug, D, roughness=0, L=1)*

Calculates two-phase pressure drop with the Bankoff (1960) correlation, as shown in [2], [3], and [4].

\[
\Delta P_{tp} = \phi_l^{7/4} \Delta P_l
\]

\[
\phi_l = \frac{1}{1 - x} \left[ 1 - \gamma \left( 1 - \frac{\rho_g}{\rho_l} \right)^{3/7} \right] \left[ 1 + x \left( \frac{\rho_l}{\rho_g} - 1 \right) \right]
\]

\[
\gamma = \frac{0.71 + 2.35 \left( \frac{\mu_g}{\mu_l} \right)}{1 + \frac{1 - x}{x} \cdot \frac{\rho_g}{\rho_l}}
\]

Parameters

- **m** [float] Mass flow rate of fluid, [kg/s]
- **x** [float] Quality of fluid, [-]
- **rhol** [float] Liquid density, [kg/m^3]
- **rhog** [float] Gas density, [kg/m^3]
- **mul** [float] Viscosity of liquid, [Pa*s]
- **mug** [float] Viscosity of gas, [Pa*s]
- **D** [float] Diameter of pipe, [m]
- **roughness** [float, optional] Roughness of pipe for use in calculating friction factor, [m]
- **L** [float, optional] Length of pipe, [m]

Returns

- **dP** [float] Pressure drop of the two-phase flow, [Pa]

Notes

This correlation is not actually shown in [1]. Its origin is unknown. The author recommends against using this.

References

[1], [2], [3], [4]

Examples

```python
>>> Bankoff(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6,
...   D=0.05, roughness=0, L=1)
4746.059442453399
```
# 2.23 Two-phase flow voidage (fluids.two_phase_voidage)

**fluids.two_phase_voidage.Thom**

Calculates void fraction in two-phase flow according to the model of [1] as given in [2].

\[
\alpha = \left[ 1 + \left( \frac{1 - x}{x} \right) \left( \frac{\rho_g}{\rho_l} \right)^{0.89} \left( \frac{\mu_l}{\mu_g} \right)^{0.18} \right]^{-1}
\]

**Parameters**

- *x* [float] Quality at the specific tube interval [\]
- *rhol* [float] Density of the liquid [kg/m^3]
- *rhog* [float] Density of the gas [kg/m^3]
- *mul* [float] Viscosity of liquid [Pa*s]
- *mug* [float] Viscosity of gas [Pa*s]

**Returns**

- *alpha* [float] Void fraction (area of gas / total area of channel), [-]

**Notes**

Based on experimental data for boiling of water. [3] presents a slightly different model. However, its results are almost identical. A comparison can be found in the unit tests. Neither expression was found in [1] in a brief review.

**References**

[1], [2], [3]

**Examples**

```python
>>> Thom(.4, 800, 2.5, 1E-3, 1E-5)
0.9801482164042417
```

**fluids.two_phase_voidage.Zivi**

Calculates void fraction in two-phase flow according to the model of [1] as given in [2] and [3].

\[
\alpha = \left[ 1 + \left( \frac{1 - x}{x} \right) \left( \frac{\rho_g}{\rho_l} \right)^{2/3} \right]^{-1}
\]

**Parameters**

- *x* [float] Quality at the specific tube interval [\]
- *rhol* [float] Density of the liquid [kg/m^3]
- *rhog* [float] Density of the gas [kg/m^3]

**Returns**

- *alpha* [float] Void fraction (area of gas / total area of channel), [-]
Notes

Based on experimental data for boiling of water. More complicated variants of this are also in [1].

References

[1], [2], [3]

Examples

```python
>>> Zivi(.4, 800, 2.5)
0.968939909056356
```

```
fluids.two_phase_voidage.Smith(x, rhol, rhog)
Calculates void fraction in two-phase flow according to the model of [1], also given in [2] and [3].

\[
\alpha = \left\{ \begin{array}{l}
1 + \left( \frac{1 - x}{x} \right) \left( \frac{\rho_g}{\rho_l} \right) \left[ K + (1 - K) \sqrt{ \frac{\rho_l}{\rho_g} + K \left( \frac{1 - x}{x} \right) } \right]^{-1}
\end{array} \right.
\]

Parameters

- \( x \) [float] Quality at the specific tube interval []
- \( \text{rhol} \) [float] Density of the liquid [kg/m^3]
- \( \text{rhog} \) [float] Density of the gas [kg/m^3]

Returns

- \( \alpha \) [float] Void fraction (area of gas / total area of channel), [-]

Notes

[1] is an easy to read paper and has been reviewed. The form of the expression here is rearranged somewhat differently than in [1] but has been verified to be numerically equivalent. The form of this in [3] is missing a square root on a bracketed term; this appears in multiple papers by the authors.

References

[1], [2], [3]

Examples

```python
>>> Smith(.4, 800, 2.5)
0.959981235534199
```

```
fluids.two_phase_voidage.Fauske(x, rhol, rhog)
Calculates void fraction in two-phase flow according to the model of [1], as given in [2] and [3].

\[
\alpha = \left[ 1 + \left( \frac{1 - x}{x} \right) \left( \frac{\rho_g}{\rho_l} \right)^{0.5} \right]^{-1}
\]
Parameters

- \( x \) [float] Quality at the specific tube interval [\]
- \( rhol \) [float] Density of the liquid [kg/m^3]
- \( rhog \) [float] Density of the gas [kg/m^3]

Returns

- \( alpha \) [float] Void fraction (area of gas / total area of channel), [-]

Notes

[1] has not been reviewed. However, both [2] and [3] present it the same way.

References

[1], [2], [3]

Examples

```python
>>> Fauske(.4, 800, 2.5)
0.9226347262627932
```

\( \text{fluids.two_phase_voidage. Chisholm_voidage}(x, rhol, rhog) \)
Calculates void fraction in two-phase flow according to the model of [1], as given in [2] and [3].  
\[
\alpha = \left[ \frac{1}{1 - x} \left( \frac{\rho_g}{\rho_l} \right) \sqrt{1 - x \left( \frac{\rho_g}{\rho_l} \right)} \right]^{-1}
\]

Parameters

- \( x \) [float] Quality at the specific tube interval [\]
- \( rhol \) [float] Density of the liquid [kg/m^3]
- \( rhog \) [float] Density of the gas [kg/m^3]

Returns

- \( alpha \) [float] Void fraction (area of gas / total area of channel), [-]

Notes

[1] has not been reviewed. However, both [2] and [3] present it the same way.

References

[1], [2], [3]

Examples
Fluids Documentation, Release 0.1

```python
>>> Chisholm_voidage(.4, 800, 2.5)
0.949525900374774
```

**fluids.two_phase_voidage.Turner_Wallis(x, rhol, rhog, mul, mug)**

Calculates void fraction in two-phase flow according to the model of [1], as given in [2] and [3].

\[
\alpha = \left[ 1 + \left( \frac{1 - x}{x} \right)^{0.72} \left( \frac{\rho_g}{\rho_l} \right)^{0.4} \left( \frac{\mu_l}{\mu_g} \right)^{0.08} \right]^{-1}
\]

**Parameters**

- `x` [float] Quality at the specific tube interval []
- `rhol` [float] Density of the liquid [kg/m^3]
- `rhog` [float] Density of the gas [kg/m^3]
- `mul` [float] Viscosity of liquid [Pa*s]
- `mug` [float] Viscosity of gas [Pa*s]

**Returns**

- `alpha` [float] Void fraction (area of gas / total area of channel), [-]

**Notes**

[1] has not been reviewed. However, both [2] and [3] present it the same way, if slightly differently rearranged.

**References**

[1], [2], [3]

**Examples**

```python
>>> Turner_Wallis(.4, 800, 2.5, 1E-3, 1E-5)
0.8384824581634625
```

**fluids.two_phase_voidage.homogeneous(x, rhol, rhog)**

Calculates void fraction in two-phase flow according to the homogeneous flow model, reviewed in [1], [2], and [3].

\[
\alpha = \frac{1}{1 + \left( \frac{1 - x}{x} \right) \frac{\rho_g}{\rho_l}}
\]

**Parameters**

- `x` [float] Quality at the specific tube interval []
- `rhol` [float] Density of the liquid [kg/m^3]
- `rhog` [float] Density of the gas [kg/m^3]

**Returns**

- `alpha` [float] Void fraction (area of gas / total area of channel), [-]
References

[1], [2], [3]

Examples

```python
>>> homogeneous(.4, 800, 2.5)
0.995334370139969
```

```
fluids.two_phase_voidage.Chisholm_Armand(x, rhol, rhog)
Calculates void fraction in two-phase flow according to the model presented in [1] based on that of [2] as shown in [3], [4], and [5].

\[ \alpha = \alpha_h \left( \frac{\alpha_h}{\alpha_h + (1 - \alpha_h)^{0.5}} \right) \]

Parameters

- `x` [float] Quality at the specific tube interval []
- `rhol` [float] Density of the liquid [kg/m^3]
- `rhog` [float] Density of the gas [kg/m^3]

Returns

- `alpha` [float] Void fraction (area of gas / total area of channel), [-]

References

[1], [2], [3], [4], [5]

Examples

```python
>>> Chisholm_Armand(.4, 800, 2.5)
0.9357814394262114
```

```
fluids.two_phase_voidage.Armand(x, rhol, rhog)
Calculates void fraction in two-phase flow according to the model presented in [1] as shown in [2], [3], and [4].

\[ \alpha = 0.833\alpha_h \]

Parameters

- `x` [float] Quality at the specific tube interval []
- `rhol` [float] Density of the liquid [kg/m^3]
- `rhog` [float] Density of the gas [kg/m^3]

Returns

- `alpha` [float] Void fraction (area of gas / total area of channel), [-]

References

[1], [2], [3], [4]
Examples

```python
>>> Armand(.4, 800, 2.5)
0.8291135303265941
```

`fluids.two_phase_voidage.Nishino_Yamazaki(x, rhol, rhog)`

Calculates void fraction in two-phase flow according to the model presented in [1] as shown in [2].

\[
\alpha = 1 - \left( \frac{1 - x \rho_g}{x \rho_l} \right)^{0.5} \alpha_h^{0.5}
\]

Parameters

- `x` [float] Quality at the specific tube interval []
- `rhol` [float] Density of the liquid [kg/m^3]
- `rhog` [float] Density of the gas [kg/m^3]

Returns

- `alpha` [float] Void fraction (area of gas / total area of channel), [-]

Notes

[3] either shows this model as iterative in terms of voidage, or forgot to add a H subscript to its second voidage term; the second is believed more likely.

References

[1], [2], [3]

Examples

```python
>>> Nishino_Yamazaki(.4, 800, 2.5)
0.931694583962682
```

`fluids.two_phase_voidage.Guzhov(x, rhol, rhog, m, D)`

Calculates void fraction in two-phase flow according to the model in [1] as shown in [2] and [3].

\[
\alpha = 0.81[1 - \exp(-2.2 \sqrt{Fr_{tp}})] \alpha_h
\]

\[
Fr_{tp} = \frac{G_{tp}^2}{gD \rho_{tp}^2}
\]

\[
\rho_{tp} = \left( \frac{1 - x \rho_g}{x \rho_l} + \frac{x \rho_g}{\rho_l} \right)^{-1}
\]

Parameters

- `x` [float] Quality at the specific tube interval []
- `rhol` [float] Density of the liquid [kg/m^3]
- `rhog` [float] Density of the gas [kg/m^3]
mass [float] Mass flow rate of both phases, [kg/s]

Diameter of the channel, [m]

Returns
alpha [float] Void fraction (area of gas / total area of channel), [-]

References
[1], [2], [3]

Examples

```python
>>> Guzhov(.4, 800, 2.5, 1, .3)
0.7626030108534588
```

fluids.two_phase_voidage.Kawahara (x, rhol, rhog, D)
Calculates void fraction in two-phase flow according to the model presented in [1], also reviewed in [2] and [3]. This expression is for microchannels.

\[
\alpha = \frac{C_1 \alpha_{0.5}}{1 - C_2 \alpha_{0.5}}
\]

Parameters
x [float] Quality at the specific tube interval []
rhol [float] Density of the liquid [kg/m^3]
rhog [float] Density of the gas [kg/m^3]
D [float] Diameter of the channel, [m]

Returns
alpha [float] Void fraction (area of gas / total area of channel), [-]

Notes
C1 and C2 were constants for different diameters. Only diameters of 100 and 50 micrometers were studied in [1]. Here, the coefficients are distributed for three ranges, > 250 micrometers, 250-75 micrometers, and < 75 micrometers.

The Armand model is used for the first, C1 and C2 are 0.03 and 0.97 for the second, and C1 and C2 are 0.02 and 0.98 for the third.

References
[1], [2], [3]
Examples

```python
>>> Kawahara(.4, 800, 2.5, 100E-6)
0.9276148194410238
```
mug [float] Viscosity of gas [Pa*s]
m [float] Mass flow rate of both phases, [kg/s]
D [float] Diameter of the channel, [m]

Returns
alpha [float] Void fraction (area of gas / total area of channel), [-]

Notes
[1] does not specify how it defines the liquid Reynolds number. [2] disagrees with [3] and [4]; the later variant was selected, with:

\[ Re_l = \frac{G_{tp} D}{\mu_l} \]

The lower limit on Reynolds number is not enforced.

References
[1], [2], [3], [4]

Examples
```python
>>> Tandon_Varma_Gupta(.4, 800, 2.5, 1E-3, 1E-5, m=1, D=0.3)
0.9228265670341428
```

**fluids.two_phase_voidage.** Harms (x, rhol, rhog, mul, mug, m, D)
Calculates void fraction in two-phase flow according to the model of [1] also given in [2] and [3].

\[
\alpha = \left[ 1 - 10.06 \left( Re_l^{-0.875} \right) \left( 1.74 + 0.104 Re_l^{0.5} \right)^2 \right] \left[ 1.376 + \frac{7.242}{X_{lt}^{1.855}} \right]^{-0.57} \\
Re_l = \frac{G_{tp} (1 - x) D}{\mu_l}
\]

Parameters

x [float] Quality at the specific tube interval []
rhol [float] Density of the liquid [kg/m\(^3\)]
rhog [float] Density of the gas [kg/m\(^3\)]
mul [float] Viscosity of liquid [Pa*s]
mug [float] Viscosity of gas [Pa*s]
m [float] Mass flow rate of both phases, [kg/s]
D [float] Diameter of the channel, [m]

Returns
alpha [float] Void fraction (area of gas / total area of channel), [-]
Notes

[1] has been reviewed.

References

[1], [2], [3]

Examples

```python
>>> Harms(.4, 800, 2.5, 1E-3, 1E-5, m=1, D=0.3)
0.9653289762907554
```

```
fluids.two_phase_voidage.Domanski_Didion(x, rhol, rhog, mul, mug)

Calculates void fraction in two-phase flow according to the model of [1] also given in [2] and [3].

if Xtt < 10:
    \[ \alpha = (1 + X_{tt}^{0.8})^{-0.378} \]

Otherwise:

    \[ \alpha = 0.823 - 0.157 \ln(X_{tt}) \]

Parameters

- **x** [float] Quality at the specific tube interval []
- **rhol** [float] Density of the liquid [kg/m^3]
- **rhog** [float] Density of the gas [kg/m^3]
- **mul** [float] Viscosity of liquid [Pa*s]
- **mug** [float] Viscosity of gas [Pa*s]

Returns

- **alpha** [float] Void fraction (area of gas / total area of channel), [-]

Notes

[1] has been reviewed. [2] gives an exponent of -0.38 instead of -0.378 as is in [1]. [3] describes only the novel half of the correlation. The portion for Xtt > 10 is novel; the other is said to be from their 31st reference, Wallis. There is a discontinuity at Xtt = 10.

References

[1], [2], [3]
Examples

```python
>>> Domanski_Didion(.4, 800, 2.5, 1E-3, 1E-5)
0.9355795597059169
```

```
fluids.two_phase_voidage.Graham(x, rhol, rhog, mul, mug, m, D, g=9.80665)
Calculates void fraction in two-phase flow according to the model of [1] also given in [2] and [3].
\[
\alpha = 1 - \exp\left\{-1 - 0.3 \ln(Ft) - 0.0328[\ln(Ft)]^2\right\}
\]
\[
Ft = \left[ \frac{G_{tp}x^3}{(1-x)\rho_g^2gD} \right]^{0.5}
\]
\[
\alpha = 0 \text{ for } Ft \leq 0.01032
\]

Parameters

- `x` [float] Quality at the specific tube interval []
- `rhol` [float] Density of the liquid [kg/m^3]
- `rhog` [float] Density of the gas [kg/m^3]
- `mul` [float] Viscosity of liquid [Pa*s]
- `mug` [float] Viscosity of gas [Pa*s]
- `m` [float] Mass flow rate of both phases, [kg/s]
- `D` [float] Diameter of the channel, [m]
- `g` [float, optional] Acceleration due to gravity, [m/s^2]

Returns

- `alpha` [float] Void fraction (area of gas / total area of channel), [-]

Notes

[1] has been reviewed. [2] does not list that the expression is not real below a certain value of Ft.

References

[1], [2], [3]

Examples

```python
>>> Graham(.4, 800, 2.5, 1E-3, 1E-5, m=1, D=0.3)
0.6403336287530644
```

```
fluids.two_phase_voidage.Yashar(x, rhol, rhog, mul, mug, m, D, g=9.80665)
Calculates void fraction in two-phase flow according to the model of [1] also given in [2] and [3].
\[
\alpha = \left[ 1 + \frac{1}{Ft} + X_{tt} \right]^{-0.321}
\]
\[
Ft = \left[ \frac{G_{tp}x^3}{(1-x)\rho_g^2gD} \right]^{0.5}
\]
Parameters

- \( x \) [float] Quality at the specific tube interval
- \( rhol \) [float] Density of the liquid [kg/m^3]
- \( rhog \) [float] Density of the gas [kg/m^3]
- \( mul \) [float] Viscosity of liquid [Pa*s]
- \( mug \) [float] Viscosity of gas [Pa*s]
- \( m \) [float] Mass flow rate of both phases, [kg/s]
- \( D \) [float] Diameter of the channel, [m]
- \( g \) [float, optional] Acceleration due to gravity, [m/s^2]

Returns

- \( \alpha \) [float] Void fraction (area of gas / total area of channel), [-]

Notes

[1] has been reviewed; both [2] and [3] give it correctly.

References

[1], [2], [3]

Examples

```python
>>> Yashar(.4, 800, 2.5, 1E-3, 1E-5, m=1, D=0.3)
0.7934893185789146
```

\[
\alpha = 1 - \frac{2(1 - x)^2}{1 - 2x + \left[1 + 4x(1 - x)\left(\frac{\mu}{\rho_g} - 1\right)\right]^{0.5}}
\]

fluids.two_phase_voidage.Huq_Loth\((x, rhol, rhog)\)

Calculates void fraction in two-phase flow according to the model of [1], also given in [2], [3], and [4].

Parameters

- \( x \) [float] Quality at the specific tube interval
- \( rhol \) [float] Density of the liquid [kg/m^3]
- \( rhog \) [float] Density of the gas [kg/m^3]

Returns

- \( \alpha \) [float] Void fraction (area of gas / total area of channel), [-]

Notes

[1] has been reviewed, and matches the expressions given in the reviews [2], [3], and [4]; the form of the expression is rearranged somewhat differently.
References
[1], [2], [3], [4]

Examples

```python
>>> Huq_Loth(0.4, 800, 2.5)
0.9593868838476147
```

`fluids.two_phase_voidage.Kopte_Newell_Chato(x, rhol, rhog, mul, mug, m, D, g=9.80665)`
Calculates void fraction in two-phase flow according to the model of [1] also given in [2].

\[
\alpha = 1.045 - \exp\left\{-1 - 0.342 \ln(F_t) - 0.0268[\ln(F_t)]^2 + 0.00597[\ln(F_t)]^3\right\}
\]

\[
F_t = \left[ \frac{G_t x^3}{(1 - x) \rho_g g D} \right]^{0.5}
\]

\[
\alpha = \alpha_h \text{ for } F_t \leq 0.044
\]

Parameters
- `x` [float] Quality at the specific tube interval []
- `rhol` [float] Density of the liquid [kg/m^3]
- `rhog` [float] Density of the gas [kg/m^3]
- `mul` [float] Viscosity of liquid [Pa*s]
- `mug` [float] Viscosity of gas [Pa*s]
- `m` [float] Mass flow rate of both phases, [kg/s]
- `D` [float] Diameter of the channel, [m]
- `g` [float, optional] Acceleration due to gravity, [m/s^2]

Returns
- `alpha` [float] Void fraction (area of gas / total area of channel), [-]

Notes
[1] has been reviewed. If is recommended this expression not be used above Ft values of 454.

References
[1], [2]

Examples

```python
>>> Kopte_Newell_Chato(.4, 800, 2.5, 1E-3, 1E-5, m=1, D=0.3)
0.6864466770087425
```
fluids.two_phase_voidage.Steiner \((x, \rho_l, \rho_g, \sigma, m, D, g=9.80665)\)
Calculates void fraction in two-phase flow according to the model of [1] also given in [2] and [3].

\[
\alpha = \frac{x}{\rho_g} \left[ C_0 \left( \frac{x}{\rho_g} + \frac{1-x}{\rho_l} \right) + \frac{v_{gm}}{G} \right]^{-1}
\]

\[
v_{gm} = \frac{1.18(1-x)}{\rho_l^{0.5}} \left[ g \sigma (\rho_l - \rho_g) \right]^{0.25}
\]

\[
C_0 = 1 + 0.12(1 - x)
\]

Parameters

- \(x\) [float] Quality at the specific tube interval []
- \(\rho_l\) [float] Density of the liquid [kg/m^3]
- \(\rho_g\) [float] Density of the gas [kg/m^3]
- \(\sigma\) [float] Surface tension of liquid [N/m]
- \(m\) [float] Mass flow rate of both phases, [kg/s]
- \(D\) [float] Diameter of the channel, [m]
- \(g\) [float, optional] Acceleration due to gravity, [m/s^2]

Returns

- \(\alpha\) [float] Void fraction (area of gas / total area of channel), [-]

Notes

[1] has been reviewed.

References

[1], [2], [3]

Examples

```python
>>> Steiner(0.4, 800., 2.5, sigma=0.02, m=1, D=0.3)
0.895950181381335
```

fluids.two_phase_voidage.Rouhani_1 \((x, \rho_l, \rho_g, \sigma, m, D, g=9.80665)\)
Calculates void fraction in two-phase flow according to the model of [1] as given in [2] and [3].

\[
\alpha = \frac{x}{\rho_l} \left[ C_0 \left( \frac{x}{\rho_l} + \frac{1-x}{\rho_g} \right) + \frac{v_{gm}}{G} \right]^{-1}
\]

\[
v_{gm} = \frac{1.18(1-x)}{\rho_l^{0.5}} \left[ g \sigma (\rho_l - \rho_g) \right]^{0.25}
\]

\[
C_0 = 1 + 0.2(1 - x)
\]

Parameters

- \(x\) [float] Quality at the specific tube interval []
**rhol** [float] Density of the liquid [kg/m^3]

**rhog** [float] Density of the gas [kg/m^3]

**sigma** [float] Surface tension of liquid [N/m]

**m** [float] Mass flow rate of both phases, [kg/s]

**D** [float] Diameter of the channel, [m]

**g** [float, optional] Acceleration due to gravity, [m/s^2]

**Returns**

**alpha** [float] Void fraction (area of gas / total area of channel), [-]

**Notes**

The expression as quoted in [2] and [3] could not be found in [1].

**References**

[1], [2], [3]

**Examples**

```python
>>> Rouhani_1(0.4, 800., 2.5, sigma=0.02, m=1, D=0.3)
0.8588420244136714
```

*fluids.two_phase_voidage.Rouhani_2* *(x, rhol, rhog, sigma, m, D, g=9.80665)*

Calculates void fraction in two-phase flow according to the model of [1] as given in [2] and [3].

\[
\alpha = \frac{x}{\rho_g} \left[ C_0 \left( \frac{x}{\rho_g} + \frac{1-x}{\rho_l} \right) + \frac{v_{gm}}{G} \right]^{-1}
\]

\[
v_{gm} = \frac{1.18(1-x)}{\rho_l^{0.5}} [g\sigma(\rho_l - \rho_g)]^{0.25}
\]

\[
C_0 = 1 + 0.2(1-x)(gD)^{0.25} \left( \frac{\rho_l}{G_{tp}} \right)^{0.5}
\]

**Parameters**

- **x** [float] Quality at the specific tube interval []
- **rhol** [float] Density of the liquid [kg/m^3]
- **rhog** [float] Density of the gas [kg/m^3]
- **sigma** [float] Surface tension of liquid [N/m]
- **m** [float] Mass flow rate of both phases, [kg/s]
- **D** [float] Diameter of the channel, [m]
- **g** [float, optional] Acceleration due to gravity, [m/s^2]

**Returns**

- **alpha** [float] Void fraction (area of gas / total area of channel), [-]
Notes

The expression as quoted in [2] and [3] could not be found in [1].

References

[1], [2], [3]

Examples

```python
>>> Rouhani_2(0.4, 800., 2.5, sigma=0.02, m=1, D=0.3)
0.44819733138968865
```

```
fluids.two_phase_voidage.Nicklin_Wilkes_Davidson(x, rhol, rhog, m, D, g=9.80665)
Calculates void fraction in two-phase flow according to the model of [1] as given in [2] and [3].

\[
\alpha = \frac{x}{\rho_g} \left[ C_0 \left( \frac{x}{\rho_g} + \frac{1-x}{\rho_l} \right) + \frac{v_{gm}}{G} \right]^{-1}
\]

\[
v_{gm} = 0.35 \sqrt{gD}
\]

\[
C_0 = 1.2
\]

Parameters

- **x** [float] Quality at the specific tube interval []
- **rhol** [float] Density of the liquid [kg/m^3]
- **rhog** [float] Density of the gas [kg/m^3]
- **m** [float] Mass flow rate of both phases, [kg/s]
- **D** [float] Diameter of the channel, [m]
- **g** [float, optional] Acceleration due to gravity, [m/s^2]

Returns

- **alpha** [float] Void fraction (area of gas / total area of channel), [-]

References

[1], [2], [3]

Examples

```python
>>> Nicklin_Wilkes_Davidson(0.4, 800., 2.5, m=1, D=0.3)
0.6798826626721431
```

```
fluids.two_phase_voidage.Gregory_Scott(x, rhol, rhog)
Calculates void fraction in two-phase flow according to the model of [1] as given in [2] and [3].

\[
\alpha = \frac{x}{\rho_g} \left[ C_0 \left( \frac{x}{\rho_g} + \frac{1-x}{\rho_l} \right) + \frac{v_{gm}}{G} \right]^{-1}
\]"
\[ v_{gm} = 0 \]
\[ C_0 = 1.19 \]

**Parameters**
- \( x \) [float] Quality at the specific tube interval [\]
- \( \text{rhol} \) [float] Density of the liquid [kg/m\(^3\)]
- \( \text{rhog} \) [float] Density of the gas [kg/m\(^3\)]

**Returns**
- \( \alpha \) [float] Void fraction (area of gas / total area of channel), [-]

**References**
[1], [2], [3]

**Examples**

```python
>>> Gregory_Scott(0.4, 800., 2.5)
0.8364154370924108
```

`fluids.two_phase_voidage.Dix(x, rhol, rhog, sigma, m, D, g=9.80665)`

Calculates void fraction in two-phase flow according to the model of [1] as given in [2] and [3].

\[
\alpha = \frac{x}{\rho_g} \left[ C_0 \left( \frac{x}{\rho_g} + \frac{1-x}{\rho_l} \right) + \frac{v_{gm}}{G} \right]^{-1}
\]

\[
v_{gm} = 2.9 \left( g \sigma \frac{\rho_l - \rho_g}{\rho_l^2} \right)^{0.25}
\]

\[
C_0 = \frac{v_{sg}}{v_m} \left[ 1 + \left( \frac{v_{sl}}{v_{sg}} \right) \left( \frac{\sigma}{\rho_l g} \right)^{0.1} \right]
\]

\[
v_{gs} = \frac{mx}{\rho_g \pi D^2}
\]

\[
v_{ls} = \frac{m(1-x)}{\rho_l \pi D^2}
\]

\[
v_m = v_{gs} + v_{ls}
\]

**Parameters**
- \( x \) [float] Quality at the specific tube interval [\]
- \( \text{rhol} \) [float] Density of the liquid [kg/m\(^3\)]
- \( \text{rhog} \) [float] Density of the gas [kg/m\(^3\)]
- \( \text{sigma} \) [float] Surface tension of liquid [N/m]
- \( m \) [float] Mass flow rate of both phases, [kg/s]
- \( D \) [float] Diameter of the channel, [m]
- \( g \) [float, optional] Acceleration due to gravity, [m/s\(^2\)]

**Returns**
- \( \alpha \) [float] Void fraction (area of gas / total area of channel), [-]
Notes

Has formed the basis for several other correlations.

References

[1], [2], [3]

Examples

```python
>>> Dix(0.4, 800., 2.5, sigma=0.02, m=1, D=0.3)
0.8268737961156514
```

`fluids.two_phase_voidage.Sun_Duffey_Peng(x, rhol, rhog, sigma, m, D, P, Pc, g=9.80665)`

Calculates void fraction in two-phase flow according to the model of [1] as given in [2] and [3].

\[
\alpha = \frac{x}{\rho_g} \left[ C_0 \left( \frac{x}{\rho_g} + \frac{1-x}{\rho_l} \right) + \frac{v_{gm}}{G} \right]^{-1}
\]

\[
v_{gm} = 1.41 \left[ \frac{g\sigma(\rho_l - \rho_g)}{\rho_l^2} \right]^{0.25}
\]

\[
C_0 = \left( 0.82 + 0.18 \frac{P}{P_c} \right)^{-1}
\]

Parameters

- `x` [float] Quality at the specific tube interval
- `rhol` [float] Density of the liquid [kg/m³]
- `rhog` [float] Density of the gas [kg/m³]
- `sigma` [float] Surface tension of liquid [N/m]
- `m` [float] Mass flow rate of both phases, [kg/s]
- `D` [float] Diameter of the channel, [m]
- `P` [float] Pressure of the fluid, [Pa]
- `Pc` [float] Critical pressure of the fluid, [Pa]
- `g` [float, optional] Acceleration due to gravity, [m/s²]

Returns

- `alpha` [float] Void fraction (area of gas / total area of channel), [-]

References

[1], [2], [3]
Examples

```python
>>> Sun_Duffey_Peng(0.4, 800., 2.5, sigma=0.02, m=1, D=0.3, P=1E5, Pc=7E6)
0.7696546506515833
```

**fluids.two_phase_voidage.**

**Xu_Fang_voidage** (*x*, *rhol*, *rhog*, *m*, *D*, *g*=9.80665)

Calculates void fraction in two-phase flow according to the model developed in the review of [1].

\[
\alpha = \left[ 1 + \left( 1 + 2Fr_{lo}^{-0.2} \alpha_{h}^{3.5} \right) \left( \frac{1 - x}{x} \right) \left( \frac{\rho_{g}}{\rho_{l}} \right) \right]^{-1}
\]

**Parameters**

- *x* [float] Quality at the specific tube interval 
- *rhol* [float] Density of the liquid [kg/m^3]
- *rhog* [float] Density of the gas [kg/m^3]
- *m* [float] Mass flow rate of both phases, [kg/s]
- *D* [float] Diameter of the channel, [m]
- *g* [float, optional] Acceleration due to gravity, [m/s^2]

**Returns**

- **alpha** [float] Void fraction (area of gas / total area of channel), [-]

**Notes**

Claims an AARD of 5.0%, and suitability for any flow regime, mini and micro channels, adiabatic, evaporating, or condensing flow, and for Frlo from 0.02 to 145, rhog/rhol from 0.004-0.153, and x from 0 to 1.

**References**

[1]

**Examples**

```python
>>> Xu_Fang_voidage(0.4, 800., 2.5, m=1, D=0.3)
0.9414660089942093
```

**fluids.two_phase_voidage.**

**Woldesemayat_Ghajar** (*x*, *rhol*, *rhog*, *sigma*, *m*, *D*, *P*, *angle*=0, *g*=9.80665)

Calculates void fraction in two-phase flow according to the model of [1].

\[
\alpha = \frac{v_{gs}}{v_{gs} \left( 1 + \left( \frac{v_{ls}}{v_{gs}} \right)^{0.1} \right) + 2.9 \left[ \frac{gD \sigma \left( 1 + \cos \theta \right) \left( \rho_{l} - \rho_{g} \right)}{\rho_{l}^{2}} \right]^{0.25} \left( 1.22 + 1.22 \sin \theta \right) \frac{P}{P_{atm}}} \]

\[
v_{gs} = \frac{m x}{\rho_{g} \frac{\pi}{4} D^{2}}
\]

\[
v_{ls} = \frac{m \left( 1 - x \right)}{\rho_{l} \frac{\pi}{4} D^{2}}
\]

**Parameters**

2.23. Two-phase flow voidage (fluids.two_phase_voidage)
Fluids Documentation, Release 0.1

x [float] Quality at the specific tube interval [-]

\textbf{rhol} [float] Density of the liquid [kg/m^3]

\textbf{rhog} [float] Density of the gas [kg/m^3]

\textbf{sigma} [float] Surface tension of liquid [N/m]

\textbf{m} [float] Mass flow rate of both phases, [kg/s]

\textbf{D} [float] Diameter of the channel, [m]

\textbf{P} [float] Pressure of the fluid, [Pa]

\textbf{angle} [float] Angle of the channel with respect to the horizontal (vertical = 90), [degrees]

\textbf{g} [float, optional] Acceleration due to gravity, [m/s^2]

Returns

\textbf{alpha} [float] Void fraction (area of gas / total area of channel), [-]

Notes

Strongly recommended.

References

[1]

Examples

```python
>>> Woldesemayat_Ghajar(0.4, 800., 2.5, sigma=0.2, m=1, D=0.3, P=1E6, angle=45)
0.7640815513429202
```

\texttt{fluids\_two\_phase\_voidage.Lockhart\_Martinelli\_Xtt} \((x, rhol, rhog, mul, mug, pow\_x=0.9,\)

\begin{itemize}
  \item \texttt{pow\_rho=0.5, pow\_mu=0.1, n=None)}
\end{itemize}

Calculates the Lockhart-Martinelli Xtt two-phase flow parameter in a general way according to [2]. [1] is said to describe this. However, very different definitions of this parameter have been used elsewhere. Accordingly, the powers of each of the terms can be set. Alternatively, if the parameter \(n\) is provided, the powers for viscosity and phase fraction will be calculated from it as shown below.

\[
X_{tt} = \left(1 - \frac{x}{x}\right)^{0.9} \left(\frac{\rho_g}{\rho_l}\right)^{0.5} \left(\frac{\mu_l}{\mu_g}\right)^{0.1}
\]

\[
X_{tt} = \left(1 - \frac{x}{x}\right)^{(2-n)/2} \left(\frac{\rho_g}{\rho_l}\right)^{0.5} \left(\frac{\mu_l}{\mu_g}\right)^{n/2}
\]

Parameters

\begin{itemize}
  \item \textbf{x} [float] Quality at the specific tube interval [-]
  \item \textbf{rhol} [float] Density of the liquid [kg/m^3]
  \item \textbf{rhog} [float] Density of the gas [kg/m^3]
  \item \textbf{mul} [float] Viscosity of liquid [Pa*s]
  \item \textbf{mug} [float] Viscosity of gas [Pa*s]
\end{itemize}
pow_x  [float, optional] Power for the phase ratio (1-x)/x, [-]
pow_rho [float, optional] Power for the density ratio rhog/rhol, [-]
pow_mu  [float, optional] Power for the viscosity ratio mul/mug, [-]
n      [float, optional] Number to be used for calculating pow_x and pow_mu if provided, [-]

Returns

    Xtt  [float] Xtt Lockhart-Martinelli two-phase flow parameter [-]

Notes

    Xtt is best regarded as an empirical parameter. If used, n is often 0.2 or 0.25.

References

[1], [2]

Examples

>>> Lockhart_Martinelli_Xtt(0.4, 800, 2.5, 1E-3, 1E-5)
0.12761659240532292

fluids.two_phase_voidage.two_phase_voidage_experimental (rho_lg, rhol, rhog)

    Calculates the void fraction for two-phase liquid-gas pipeflow. If the weight of fluid in a pipe pipe could be
    measured and the volume of the pipe were known, an effective density of the two-phase mixture could be
    calculated. This is directly relatable to the void fraction of the pipe, a parameter used to predict the pressure
drop. This function converts that measured effective two-phase density to void fraction for use in developing
    correlations.

        \[ \alpha = \frac{\rho_m - \rho_l}{\rho_g - \rho_l} \]

Parameters

    rho_lg  [float] Two-phase effective density [kg/m^3]
    rhol    [float] Density of the liquid [kg/m^3]
    rhog    [float] Density of the gas [kg/m^3]

Returns

    alpha  [float] Void fraction (area of gas / total area of channel), [-]

References

[1]

Examples

>>> two_phase_voidage_experimental(481.0, 800, 2.5)
0.4
fluids.two_phase_voidage.density_two_phase(alpha, rhol, rhog)
Calculates the “effective” density of fluid in a liquid-gas flow. If the weight of fluid in a pipe could be measured and the volume of the pipe were known, an effective density of the two-phase mixture could be calculated. This is directly relatable to the void fraction of the pipe, a parameter used to predict the pressure drop. This function converts void fraction to effective two-phase density.

\[ \rho_m = \alpha \rho_g + (1 - \alpha) \rho_l \]

**Parameters**
- **alpha** [float] Void fraction (area of gas / total area of channel), [-]
- **rhol** [float] Density of the liquid [kg/m^3]
- **rhog** [float] Density of the gas [kg/m^3]

**Returns**
- **rho_lg** [float] Two-phase effective density [kg/m^3]

**Notes**

THERE IS NO THERMODYNAMIC DEFINITION FOR THIS QUANTITY. DO NOT USE THIS VALUE IN SINGLE-PHASE CORRELATIONS.

**References**

[1]

**Examples**

```python
density_two_phase(.4, 800, 2.5)
```

481.0

fluids.two_phase_voidage.Beattie_Walley(x, mul, mug, rhol, rhog)
Calculates a suggested definition for liquid-gas two-phase flow viscosity in internal pipe flow according to the form in [1] and shown in [2] and [3].

\[ \mu_m = \mu_l (1 - \alpha_m) (1 + 2.5 \alpha_m) + \mu_g \alpha_m \]

\[ \alpha_m = \frac{1}{1 + \left( \frac{1-x}{x} \right) \frac{\mu_g}{\mu_l}} \text{(homogeneous model)} \]

**Parameters**
- **x** [float] Quality of the gas-liquid flow, [-]
- **mul** [float] Viscosity of liquid, [Pa*s]
- **mug** [float] Viscosity of gas, [Pa*s]
- **rhol** [float] Density of the liquid [kg/m^3]
- **rhog** [float] Density of the gas [kg/m^3]

**Returns**
- **mu_lg** [float] Liquid-gas viscosity (a suggested definition, potentially useful for empirical work only!) [Pa*s]
Notes

This model converges to the liquid or gas viscosity as the quality approaches either limits.

References

[1], [2], [3]

Examples

```python
>>> Beattie_Whalley(x=0.4, mul=1E-3, mug=1E-5, rhol=850, rhog=1.2)
1.7363806909512365e-05
```

Calculates a suggested definition for liquid-gas two-phase flow viscosity in internal pipe flow according to the form in [1] and shown in [2] and [3].

\[
\mu_m = \left( \frac{x}{\mu_g} + \frac{1-x}{\mu_l} \right)^{-1}
\]

Parameters

- x [float] Quality of the gas-liquid flow, [-]
- mul [float] Viscosity of liquid, [Pa*s]
- mug [float] Viscosity of gas, [Pa*s]

Returns

- mu_lg [float] Liquid-gas viscosity (a suggested definition, potentially useful for empirical work only!) [Pa*s]

Notes

This model converges to the liquid or gas viscosity as the quality approaches either limits.

[3] states this is the most common definition of two-phase liquid-gas viscosity.

References

[1], [2], [3]

Examples

```python
>>> McAdams(x=0.4, mul=1E-3, mug=1E-5)
2.4630541871921184e-05
```

Calculates a suggested definition for liquid-gas two-phase flow viscosity in internal pipe flow according to the form in [1] and shown in [2] and [3].

\[
\mu_m = x\mu_g + (1-x)\mu_l
\]
Parameters

- **x** [float] Quality of the gas-liquid flow, [-]
- **mul** [float] Viscosity of liquid, [Pa*s]
- **mug** [float] Viscosity of gas, [Pa*s]

Returns

- **mu_lg** [float] Liquid-gas viscosity (a suggested definition, potentially useful for empirical work only!) [Pa*s]

Notes

This model converges to the liquid or gas viscosity as the quality approaches either limits.

References

[1], [2], [3]

Examples

```python
>>> Cicchitti(x=0.4, mul=1E-3, mug=1E-5)
0.0006039999999999999
```

The model can be used to calculate the liquid-gas viscosity in internal pipe flow according to the form in [1] and shown in [2] and [3].

\[
\mu_m = \frac{\mu_l \mu_g}{\mu_g + x^{1.4}(\mu_l - \mu_g)}
\]

Parameters

- **x** [float] Quality of the gas-liquid flow, [-]
- **mul** [float] Viscosity of liquid, [Pa*s]
- **mug** [float] Viscosity of gas, [Pa*s]

Returns

- **mu_lg** [float] Liquid-gas viscosity (a suggested definition, potentially useful for empirical work only!) [Pa*s]

Notes

This model converges to the liquid or gas viscosity as the quality approaches either limits.

References

[1], [2]
Examples

```python
>>> Lin_Kwok(x=0.4, mul=1E-3, mug=1E-5)
3.515119398126066e-05
```

**Fluids Documentation**

`Fourar_Bories (x, mul, mug, rhol, rhog)`

Calculates a suggested definition for liquid-gas two-phase flow viscosity in internal pipe flow according to the form in [1] and shown in [2] and [3].

\[ \mu_m = \rho_m \left( \sqrt{x \nu_g} + \sqrt{(1 - x) \nu_l} \right)^2 \]

**Parameters**

- `x` [float] Quality of the gas-liquid flow, [-]
- `mul` [float] Viscosity of liquid, [Pa*s]
- `mug` [float] Viscosity of gas, [Pa*s]
- `rhol` [float] Density of the liquid, [kg/m^3]
- `rhog` [float] Density of the gas, [kg/m^3]

**Returns**

- `mu_lg` [float] Liquid-gas viscosity (a suggested definition, potentially useful for empirical work only!) [Pa*s]

**Notes**

This model converges to the liquid or gas viscosity as the quality approaches either limits.

This was first expressed in the equivalent form as follows:

\[ \mu_m = \rho_m \left( x \nu_g + (1 - x) \nu_l + 2 \sqrt{x(1 - x) \nu_g \nu_l} \right) \]

**References**

[1], [2], [3]

**Examples**

```python
>>> Fourar_Bories(x=0.4, mul=1E-3, mug=1E-5, rhol=850, rhog=1.2)
2.127617150298565e-05
```

`liquid_gas_voidage(x, rhol, rhog, D=None, m=None, mul=None, mug=None, sigma=None, P=None, Pc=None, angle=0, g=9.80665, Method=None, AvailableMethods=False)`

This function handles calculation of two-phase liquid-gas voidage for flow inside channels. 29 calculation methods are available, with varying input requirements. A correlation will be automatically selected if none is specified. The full list of correlation can be obtained with the `AvailableMethods` flag.

This function is used to calculate the (liquid) holdup as well, as:

\[ \text{holdup} = 1 - \text{voidage} \]
If no correlation is selected, the following rules are used, with the earlier options attempted first:

- TODO: defaults

**Parameters**

- `x` [float] Quality of fluid, [-]
- `rhol` [float] Liquid density, [kg/m^3]
- `rhog` [float] Gas density, [kg/m^3]
- `D` [float, optional] Diameter of pipe, [m]
- `m` [float, optional] Mass flow rate of fluid, [kg/s]
- `mul` [float, optional] Viscosity of liquid, [Pa*s]
- `mug` [float, optional] Viscosity of gas, [Pa*s]
- `sigma` [float, optional] Surface tension, [N/m]
- `P` [float, optional] Pressure of fluid, [Pa]
- `Pc` [float, optional] Critical pressure of fluid, [Pa]
- `angle` [float, optional] Angle of the channel with respect to the horizontal (vertical = 90), [degrees]
- `g` [float, optional] Acceleration due to gravity, [m/s^2]

**Returns**

- `alpha` [float] Void fraction (area of gas / total area of channel), [-]
- `methods` [list, only returned if AvailableMethods == True] List of methods which can be used to calculate two-phase liquid-gas voidage with the given inputs.

**Other Parameters**

- `Method` [string, optional] A string of the function name to use, as in the dictionary `two_phase_voidage_correlations`.
- `AvailableMethods` [bool, optional] If True, function will consider which methods which can be used to calculate two-phase liquid-gas voidage with the given inputs and return them as a list instead of performing a calculation.

**Examples**

```python
>>> liquid_gas_voidage(m=0.6, x=0.1, rhol=915., rhog=2.67, mul=180E-6, mug=14E-6, ...
... sigma=0.0487, D=0.05)
0.9744097632663492
```

**fluids.two_phase_voidage.gas_liquid_viscosity**

```python
fluids.two_phase_voidage.gas_liquid_viscosity(x, mul, mug, rhol=None, rhog=None,
... Method=None, AvailableMethods=False)
```

This function handles the calculation of two-phase liquid-gas viscosity. Six calculation methods are available; three of them require only `x`, `mul`, and `mug`; the other three require `rhol` and `rhog` as well.

The ‘McAdams’ method will be used if no method is specified. The full list of correlation can be obtained with the `AvailableMethods` flag.

**ALL OF THESE METHODS ARE ONLY SUGGESTED DEFINITIONS, POTENTIALLY USEFUL FOR EMPIRICAL WORK ONLY!**
Parameters

\[ x \ [\text{float}] \text{Quality of fluid, [-]} \]
\[ \text{mul} \ [\text{float}] \text{Viscosity of liquid, [Pa*s]} \]
\[ \text{mug} \ [\text{float}] \text{Viscosity of gas, [Pa*s]} \]
\[ \text{rhol} \ [\text{float, optional}] \text{Liquid density, [kg/m^3]} \]
\[ \text{rhog} \ [\text{float, optional}] \text{Gas density, [kg/m^3]} \]

Returns

\[ \text{mu}_l g \ [\text{float}] \text{Liquid-gas viscosity (a suggested definition, potentially useful for empirical work only!)} \ [\text{Pa*s}] \]
\[ \text{methods} \ [\text{list, only returned if AvailableMethods == True}] \text{List of methods which can be used to calculate two-phase liquid-gas viscosity with the given inputs.} \]

Other Parameters

\[ \text{Method} \ [\text{string, optional}] \text{A string of the function name to use, as in the dictionary liq-uid_gas_viscosity_correlations.} \]
\[ \text{AvailableMethods} \ [\text{bool, optional}] \text{If True, function will consider which methods which can be used to calculate two-phase liquid-gas viscosity with the given inputs and return them as a list instead of performing a calculation.} \]

Notes

All of these models converge to the liquid or gas viscosity as the quality approaches either limits. Other definitions have been proposed, such as using only liquid viscosity.

These values cannot just be plugged into single phase correlations!

Examples

```python
>>> gas_liquid_viscosity(x=0.4, mul=1E-3, mug=1E-5, rhol=850, rhog=1.2, Method='Duckler')
1.2092040385066917e-05
>>> gas_liquid_viscosity(x=0.4, mul=1E-3, mug=1E-5)
2.4630541871921184e-05
```

2.24 Support for pint Quantities (fluids.units)

Basic module which wraps all fluids functions and classes to be compatible with the pint unit handling library. All other object - dicts, lists, etc - are not wrapped. Supports star imports; so the same objects exported when importing from the main library will be imported from here.

```python
>>> from fluids.units import *
```

There is no global unit registry in pint, and each registry must be a singleton. However, there is a default registry which is suitable for use in multiple modules at once.

This default registry should be imported in one of the following ways (it does not need to be called \( u \); it can be imported from pint as \( ureg \) or any other name):
>>> from pint import _DEFAULT_REGISTRY as u  
Note that if the star import convention is used, it will be imported as u for you. Unlike the normal convention, this
registry is already initialized. To repeat it again, you CANNOT do the following in your project and work with
fluids.units.

>>> from pint import UnitRegistry  
>>> u = UnitRegistry()  # NO

All dimensional arguments to functions in fluids.units must be provided as Quantity objects.

>>> Reynolds(V=3.5*u.m/u.s, D=2*u.m, rho=997.1*u.kg/u.m**3, mu=1E-3*u.Pa*u.s)
<Quantity(6979700.0, 'dimensionless')>

The result is always one or more Quantity objects, depending on the signature of the function called.

For arguments whose documentation specify they are dimensionless, they can optionally be passed in without making
them dimensionless numbers with pint.

>>> speed_synchronous(50*u.Hz, poles=12)
<Quantity(1500.0, 'revolutions_per_minute')>

>>> speed_synchronous(50*u.Hz, poles=12*u.dimensionless)
<Quantity(1500.0, 'revolutions_per_minute')>

It is good practice to use dimensionless quantities as follows, but it is optional.

>>> K_separator_Watkins(0.88*u.dimensionless, 985.4*u.kg/u.m**3, 1.3*u.kg/u.m**3, ˓
→horizontal=True)
<Quantity(0.0794470406403, 'meter / second')>

Like all pint registries, the default unit system can be changed. However, all functions will still return the unit their
documentation says they do. To convert to the new base units, use the method .to_base_units().

>>> u.default_system = 'imperial'
>>> K_separator_Watkins(0.88*u.dimensionless, 985.4*u.kg/u.m**3, 1.3*u.kg/u.m**3, ˓
→horizontal=True).to_base_units()
<Quantity(0.0868843401578, 'yard / second')>

The order of the arguments to a function is the same as it is in the regular library; it won’t try to infer argument position
from their units, an exception will be raised.

>>> K_separator_Watkins(985.4*u.kg/u.m**3, 1.3*u.kg/u.m**3, 0.88*u.dimensionless, ˓
→horizontal=True)
Exception: Converting 0.88 dimensionless to units of kg/m^3 raised
DimensionalityError: Cannot convert from 'dimensionless' (dimensionless) to
'kilogram / meter ** 3' ([mass] / [length] ** 3)

Support for classes is provided by wrapping each class by a proxy class which reads the docstrings of each method
and the main class to determine the inputs and outputs. Properties, attributes, inputs, and units are all included.

>>> T1 = TANK(L=3*u.m, D=150*u.cm, horizontal=True)
>>> T1.V_total, T1.h_max
(<Quantity(5.30143760293, 'meter ** 3')>, <Quantity(1.5, 'meter')>)
>>> T1.V_from_h(0.1*u.m)
<Quantity(0.151783071377, 'meter ** 3')>

424 Chapter 2. API Reference
Note that static methods cannot be used with the base class, only an instantiated class. This is because the proxy class wraps the methods only on creation of the object.

```
>>> ATMOSPHERE_1976.thermal_conductivity(300*u.K)
AttributeError: type object 'ATMOSPHERE_1976' has no attribute 'thermal_conductivity'
```

```
>>> ATMOSPHERE_1976(0*u.m).thermal_conductivity(300*u.K)
<Quantity(0.0262520007809, 'watt / kelvin / meter')>
```

### 2.25 Support for numpy arrays (fluids.vectorized)

Basic module which wraps all fluids functions with numpy’s `vectorize`. All other object - dicts, classes, etc - are not wrapped. Supports star imports; so the same objects exported when importing from the main library will be imported from here.

```
>>> from fluids.vectorized import *
```

Inputs do not need to be numpy arrays; they can be any iterable:

```
>>> fluids.vectorized.friction_factor(Re=[100, 1000, 10000], eD=0)
array([0.64 , 0.064 , 0.03088295])
```

Note that because this needs to import fluids itself, `fluids.vectorized` needs to be imported separately; the following will cause an error:

```
>>> import fluids
>>> fluids.vectorized  # Won't work, has not been imported yet
```

The correct syntax is as follows:

```
>>> import fluids.vectorized  # Necessary
>>> from fluids.vectorized import *  # May be used without first importing fluids
```
3.1 7.1 Smooth Pipe (Plastic)

Water at 30 degrees Celsius flows through a 20 m length of 50 mm plastic (smooth wall) pipe, at a flow rate of 200 L/min.

Calculate the Reynolds number and friction factor.

```
[1]: from fluids.units import *
from thermo.units import Stream
from math import *

Q = 0.2*u.m**3/u.min
T = 30*u.degC
P = 2*u.bar  # assumed
water = Stream('water', T=T, P=P, Q=Q)

NPS, Di, Do, t = nearest_pipe(Di=0.05*u.m)
v = Q/(pi/4*Di**2)
Re = Reynolds(D=Di, rho=water.rho, mu=water.mu, V=v)
print('Reynolds number = %s' % Re)
fd = friction_factor(Re=Re, eD=_roughness['Glass']/Di)
print('Darcy friction factor = %s' % fd)

Reynolds number = 101005.004737 dimensionless
Darcy friction factor = 0.0181086578522 dimensionless
```

The values presented in their solution are Re=100600; and fd=0.0179. The difference in values is due to the precision they used.

3.2 7.2 L, L over D and K from Kv for Conventional Type Valves

A 150 mm class 125 Y-pattern globe valve with a Kv=500 flow coefficient is given.
Calculate the resistance coefficient $K$, L/D equivalent, and the length for complete turbulence in the flow.

Use schedule 40, 150 mm pipe as a reference.

```python
from fluids.units import *

D = .154*u.m
Kv = 500*u.m**3/u.hour

K = Kv_to_K(Kv, D)
L_D = L_equiv_from_K(K, fd=.015)
L = D*L_D

print('Loss coefficient = %s' % K)
print('Equivalent length = %s' % L_D)
print('Length for complete turbulence = %s' % L)
```

Loss coefficient = 3.5996713984 dimensionless
Equivalent length = 239.978093227 dimensionless
Length for complete turbulence = 36.9566263569 meter

The results calculated in Crane TP 410M are 3.6, 240, and 37.0 respectively.

### 3.3 7.3 L, L over D, K, and Kv for Conventional Type Valves

A 100 mm class 600 steel angle valve, has a full area seat.

Calculate its resistance coefficient $K$, flow coefficient $Kv$, the equivalent length of it L/D, and the length for complete turbulent L.

```python
from fluids.units import *

NPS, D1, Do, t = nearest_pipe(Do=0.103*u.m, schedule='80')
fd = 0.0165  # provided - note equivalent length is proportional to this value
d = 0.0972*u.m  # diameter of seat
K = K_angle_valve_Crane(D1=d, D2=Di, fd=fd, style=1)
Kv = K_to_Kv(K, d)
L_D = L_equiv_from_K(K, fd)
L = L_D*d

print('Loss coefficient = %s' % K)
print('Valve flow coefficient = %s' % Kv)
print('Equivalent length = %s' % L_D)
print('Length for complete turbulence = %s' % L)
```

Loss coefficient = 2.47296378211 dimensionless
Valve flow coefficient = 240.316529308 meter ** 3 / hour
Equivalent length = 149.876592855 dimensionless
Length for complete turbulence = 14.5680048255 meter

The values given in the solution are $K=2.475$, $Kv = 240.2$, L/D=150, and $L = 14.6$.

### 3.4 7.4 Venturi Type Valves

A 150 by 100 mm class 600 steel gate valve, conically tapered ports, length 550 mm, back of seat ring ~150 mm. The valve is connected to 146 mm schedule 80 pipe. The angle can be calculated to be 13 degrees. The valve is specified to be operating in turbulent conditions.
The values calculated in the problem use a friction factor of 0.015; this is the source of the discrepancies. Their procedure for loss in valves and fittings is based around the roughness of commercial steel pipe with a roughness of 0.0018 inches, but in their examples they simply look their friction factors up in a table which does not consider the diameter of the pipe. Their calculated values are $K_2 = 1.22$, $L/D=81.3$, and $L = 11.9$.

### 3.5 7.5 Lift Check Valves

A lift check valve of type globe (with a wing-guided disc) is added to a 80 mm Schedule 40 horizontal pipe carrying water at a flow rate of 300 L/min.

Calculate the check valve size, and pressure drop. The disc should be fully lifted at the specified flow.
print('Loss coefficient = %s' % \(K_2\))

\[v_{\text{pipe}} = \frac{Q}{(\pi/4 \times D_{80}^2)}\]

\[dP = 0.5 \times \rho \times v_{\text{pipe}}^2 \times K_2\]

\[\text{print ('Pressure drop = $s' \ dP.to(u.Pa))}\]

Minimum velocity = 1.582563779 meter / second

Velocity in 80 mm valve = 1.04907178712 meter / second

Minimum velocity = 1.02523019353 meter / second

Velocity in 60 mm valve = 1.6193609196 meter / second

Loss coefficient = 24.8796766231 dimensionless

Pressure drop = 13666.0409068 pascal

The values calculated in the example are \(K_2 = 26.3\) and pressure drop = 14450 Pa. Interestingly, the formula for minimum lift velocity used in their example does not use the ratio of diameters as the formula in their appendix shows. Otherwise the examples match.

3.6 7.6 Reduced Port Ball Valve

Water is discharged at 15 degrees Celsius from a tank with 7 m of head to atmosphere through:

- 60 meters of 80 mm schedule 40 pipe
- Six 80 mm standard 90 degree threaded elbows
- One 80 mm flanged ball valve, with a 60 mm diameter seat, 16 degree conical inlet and 30 degree conival outlet.
- The entrance is sharp-edged and flush with the tank

```
[1]: from thermo.units import Chemical
from fluids.units import *
from math import pi
water = Chemical('water', T=15*u.degC)
rho = water.rho
mu = water.mu

H = 7*u.m
L = 60*u.m
fd = 0.017 # assumed in their example
NPS, D_pipe, Do_pipe, t = nearest_pipe(Do=80*u.mm)

K = K_from_f(fd=fd, L=L, D=D_pipe)
K += entrance_sharp()
K += exit_normal()
K += 6*bend_rounded(D_pipe, angle=90*u.degrees, fd=fd, bend_diameters=0.65)
ball_valve_angle = 0.5*(15+30)*u.degrees # use the average angle
K += K_ball_valve_Crane(D1=D_pipe, D2=60*u.mm, angle=ball_valve_angle, fd=fd)

v = (2*u.gravity*H/K)**0.5
print('Velocity = $s' \ v.to_base_units())
Q = v*pi/4*D_pipe**2
print('Flow rate = $s' \ Q.to(u.L/u.min))
Re = Reynolds(D=D_pipe, rho=rho, mu=mu, V=v)
f_d = friction_factor(Re=Re, eD=0.0018*u.inch/D_pipe)

Velocity = 2.77004132376 meter / second
Flow rate = 792.547439913 liter / minute
```
The radius of curvature of the elbows was not specified; 0.65 bend diameters matches their results most closely. They modified the ball valve equation to support both an inlet and an outlet angle; the average value is used here.

Their calculated values are 2.74 m/s and flow rate of 781 L/min.

The calculation can be performed more accurately by iterating; a naive approach is shown below. A very different flow rate is obtained when the roughness of the pipe is considered in the friction factor calculation.

\[ \text{[2]: } fd = 0.017 \]

```python
for i in range(7):
    K += entrance_sharp()
    K += exit_normal()
    K += 6*bend_rounded(D_pipe, angle=90*u.degrees, fd=fd, bend_diameters=0.65)
    ball_valve_angle = 0.5*(15+30)*u.degrees  # use the average angle
    K += K_ball_valve_Crane(D1=D_pipe, D2=60*u.mm, angle=ball_valve_angle, fd=fd)
    v = (2*u.gravity*H/K)**0.5
    Q = v*pi/4*D_pipe**2
    print('Flow rate = %s liter / minute' % Q.to(u.L/u.min))
    Re = Reynolds(D=D_pipe, rho=rho, mu=mu, V=v)
    fd = friction_factor(Re=Re, eD=0.0018*u.inch/D_pipe)
```

3.7 7.7 Laminar flow in Valves, Fittings, and Pipe - System from Example 7.6

SAE oil is discharged at 15 degrees Celsius from a tank with 7 m of head to atmosphere through:

- 60 meters of 80 mm schedule 40 pipe
- Six 80 mm standard 90 degree threaded elbows
- One 80 mm flanged ball valve, with a 60 mm diameter seat, 16 degree conical inlet and 30 degree conival outlet.
- The entrance is sharp-edged and flush with the tank

This is the same problem as 7.6, except the properties of the fluid are sufficiently viscous to put it into the laminar regime although not by much.

\[ \text{[1]: } \text{from fluids.units import *} \]

```python
from math import pi
rho = 875.*u.kg/u.m**3
mu = 78*u.cP
H = 7*u.m
L = 60*u.m
NPS, D_pipe, Do_pipe, t = nearest_pipe(Do=80*u.mm)
fd = 0.017
```
Fluids Documentation, Release 0.1

Re = 1E5
for i in range(20):
    K = K_from_f(fd=fd, L=L, D=D_pipe)
    K += entrance_sharp()
    K += exit_normal()
    K += 6*bend_rounded(D_pipe, angle=90*u.degrees, fd=fd, bend_diameters=0.65)
    ball_valve_angle = 0.5*(15+30)*u.degrees # use the average angle
    K += K_ball_valve_Crane(D1=D_pipe, D2=60*u.mm, angle=ball_valve_angle, fd=fd)

v = (2*u.gravity*H/K)**0.5
Q = v*pi/4*D_pipe**2
print('Flow rate = %s, Reynolds number = %s' %(Q.to(u.L/u.min), Re))
Re = Reynolds(D=D_pipe, rho=rho, mu=mu, V=v)
fd = friction_factor(Re=Re, eD=0.0018*u.inch/D_pipe)

Flow rate = 792.547439913 liter / minute, Reynolds number = 100000.0
Flow rate = 428.461048981 liter / minute, Reynolds number = 2421.30022376
Flow rate = 485.75141276 liter / minute, Reynolds number = 1308.98515537
Flow rate = 515.733166793 liter / minute, Reynolds number = 1484.01211735
Flow rate = 530.620769781 liter / minute, Reynolds number = 1575.60894058
Flow rate = 537.828263434 liter / minute, Reynolds number = 1653.64327082
Flow rate = 540.060753053 liter / minute, Reynolds number = 1661.27083289
Flow rate = 544.060753053 liter / minute, Reynolds number = 1661.02708328
Flow rate = 544.23506934 liter / minute, Reynolds number = 1662.1523975 dimensionless
Flow rate = 544.317558945 liter / minute, Reynolds number = 1662.68469092
Flow rate = 544.35689163 liter / minute, Reynolds number = 1663.93670372
Flow rate = 544.375055237 liter / minute, Reynolds number = 1663.05594438
Flow rate = 544.383791685 liter / minute, Reynolds number = 1663.11235981
Flow rate = 544.387924908 liter / minute, Reynolds number = 1663.13905041
Flow rate = 544.389880327 liter / minute, Reynolds number = 1663.15167777
Flow rate = 544.390805428 liter / minute, Reynolds number = 1663.15765173
Flow rate = 544.39124309 liter / minute, Reynolds number = 1663.16047799
Flow rate = 544.391450145 liter / minute, Reynolds number = 1663.16181509

No solution is actually presented in the example; but the result of their initial guess of a velocity of 1.5 m/s gives 511.2 L/min.

Chapter 3. Solved fluids mechanics problems
3.8 7.8 Laminar flow in Valves, Fittings, and Pipe - SAE oil through a pipe and globe valve

SAE 30 Oil at 40 degrees Celsius and a flow rate of 600 barrels/hour flows in a 60 m long 200mm schedule 40 pipe and passes through a 200 mm globe valve, full area seat.

Calculate the pressure drop.

1: from fluids.units import *
from thermo import Stream
from math import *

Q = 600*u.barrel/u.hour
L = 60*u.m
NPS, D_pipe, Do_pipe, t = nearest_pipe(Di=200*u.mm)

rho = 869.2*u.kg/u.m**3
mu = 130*u.cP

V = Q/(pi/4*D_pipe**2)
Re = Reynolds(D=D_pipe, V=V, rho=rho, mu=mu)
fd = friction_factor(Re=Re)
ft = friction_factor(Re=1E6, eD=0.0018*u.inch/D_pipe)
K = K_globe_valve_Crane(D1=D_pipe, D2=D_pipe, fd=ft)
K += K_from_f(fd=fd, L=L, D=D_pipe)
dP_from_K(K=K, V=V, rho=rho)

5781.61276167 pascal

The example gives a pressure drop of 6382 Pa.

3.9 7.10 Piping Systems - Steam

40 bar steam, 450 degrees Celsius flows though a 120 m long horizontal 150mm schedule 80 pipe at a rate of 40000 kg/hr.

There are three 90 degree weld elbows with rc=1.5, 1 fully open class 600 150mm x 100mm venturi class gate valve, one class 600 150 mm class y pattern globe valve with a seat diameter of 90% the inside pipe diameter (disc fully lifted).

Calculate the pressure drop through the system.

1: from math import *
from fluids.units import *
from thermo.units import Stream

m = 40000*u.kg/u.hr
T = 450*u.degC
P = 40*u.bar

steam = Stream('water', T=T, P=P, m=m)

rho = steam.rho
mu = steam.mu
print('Density = %s' %rho)
print('Viscosity = %s' %mu)
Density = 12.5195049457 kilogram / meter ** 3
Viscosity = 2.65571609065e-05 pascal * second

\[ L = 120 \text{ u.m} \]

NPS, D\_pipe, Do\_pipe, t = nearest\_pipe(Do=150*u.mm, schedule='80')
Q = m/rho
v = Q/(\pi/4*D\_pipe**2)
Re = Reynolds(rho=rho, mu=mu, D=D\_pipe, V=v)
fd = friction\_factor(Re=Re, eD=0.0018*u.inch/D\_pipe)
K\_elbow = bend\_rounded(Di=D\_pipe, angle=90*u.degrees, fd=fd, bend\_diameters=1.5)
K\_friction = K\_from\_f(fd=fd, L=L, D=D\_pipe)
K\_globe\_valve = K\_globe\_valve\_Crane(D1=0.9*D\_pipe, D2=D\_pipe, fd=fd)
K\_gate\_valve = K\_gate\_valve\_Crane(D1=100*u.mm, D2=D\_pipe, angle=13.115*u.degrees, fd=fd)

# Angle and inside diameter are taken from example 7.4,
K\_tot = K\_gate\_valve + K\_globe\_valve + 3*K\_elbow + K\_friction

dP = dP\_from\_K(K=K\_tot, rho=rho, V=v)

print('Crane elbow term = 0.63; calculated = %s dimensionless' % (3*K\_elbow))
print('Crane globe valve term = 1.44; calculated = %s dimensionless' % (K\_globe\_valve))
print('Crane gate valve term = 1.22; calculated = %s dimensionless' % (K\_gate\_valve))
print('Crane friction term = 12.3; calculated = %s dimensionless' % (K\_friction))

Darcy friction factor = 0.04071651477
Pressure drop = 1008693.59173 pascal

This compares terribly to the example, which calculates a pressure drop of 274800 Pa!

Two errors are apparent - their example uses a different coefficient (55) in the globe valve pressure drop equation than that shown in their appendix (340); and they re-use their prior calculated gate valve, despite the friction factor being different in this example. This, plus their use of a constant 0.015 friction factor, explains the difference.

The example below uses their calculated globe valve pressure drop and their friction factor. The result (268500 Pa) compares well with their calculation; the additional decimals and better physical properties explain the rest.

L = 120*u.m
NPS, D\_pipe, Do\_pipe, t = nearest\_pipe(Do=150*u.mm, schedule='80')
Q = m/rho
v = Q/(\pi/4*D\_pipe**2)
Re = Reynolds(rho=rho, mu=mu, D=D\_pipe, V=v)
fd = 0.015
K\_elbow = bend\_rounded(Di=D\_pipe, angle=90*u.degrees, fd=fd, bend\_diameters=1.5)
K\_friction = K\_from\_f(fd=fd, L=L, D=D\_pipe)
K\_globe\_valve = 1.44
K\_gate\_valve = K\_gate\_valve\_Crane(D1=100*u.mm, D2=D\_pipe, angle=13.115*u.degrees, fd=fd)
K\_tot = K\_gate\_valve + K\_globe\_valve + 3*K\_elbow + K\_friction

dP = dP\_from\_K(K=K\_tot, rho=rho, V=v)
print('Crane elbow term = 0.63; calculated = %s' % (3*K_elbow))
print('Crane globe valve term = 1.44; calculated = %s' % (K_globe_valve))
print('Crane gate valve term = 1.22; calculated = %s' % (K_gate_valve))
print('Crane friction term = 12.3; calculated = %s' % (K_friction))
print('Darcy friction factor = %s' % fd)
print('Pressure drop = %s' % dP.to(u.Pa))

Crane elbow term = 0.63; calculated = 0.598864740713 dimensionless
Crane globe valve term = 1.44; calculated = 1.44
Crane gate valve term = 1.22; calculated = 1.16148152353 dimensionless
Crane friction term = 12.3; calculated = 12.2984421973 dimensionless
Darcy friction factor = 0.015
Pressure drop = 269976.656247 pascal

3.10 7.11 Flat heating Coils - Water

Water at 80 degrees Celsius flows through a flat heating coil at a rate of 60 L/min. There are 7 180 degree bends in it. The coil is 8 m long, with 0.5 m of straight length on the inlet and exit. The r/D of the bends is 4. The pipe is schedule 40, 25 mm pipe.

[1]: from math import *
from fluids.units import *
from thermo.units import Chemical

water = Chemical('water', P=2*u.bar, T=80*u.degC)   # P assumed
rho = water.rho
mu = water.mu
Q = 60*u.L/u.min
L = (1*8 + 0.5*2)*u.m

NPS, D_pipe, Do_pipe, t = nearest_pipe(Di=25*u.mm)
v = Q/(pi/4*D_pipe**2)
Re = Reynolds(rho=rho, mu=mu, D=D_pipe, V=v)
fd = friction_factor(Re=Re, eD=0.0018*u.inch/D_pipe)

K_elbow = bend_rounded(Di=D Pipe, angle=180*u.degrees, fd=fd, bend_diameters=5)
K_friction = K_from_f(fd=fd, L=L, D=D_pipe)
K_tot = 7*K_elbow + K_friction
dP = dP_from_K(K=K_tot, rho=rho, V=v)
print('Pressure drop = %s' % dP.to(u.Pa))

Pressure drop = 61330.665674 pascal

The value presented in the solution is 19609 Pa. They chose a constant friction factor of 0.024 in this calculation. If this were used, the result compares much better. Their friction factor can be obtained at a roughness of 0.05 mm.

[2]: fd = 0.024
K_elbow = bend_rounded(Di=D Pipe, angle=180*u.degrees, fd=fd, bend_diameters=5)
K_friction = K_from_f(fd=fd, L=L, D=D_pipe)
K_tot = 7*K_elbow + K_friction
dP = dP_from_K(K=K_tot, rho=rho, V=v)
print('Pressure drop = %s' % dP.to(u.Pa))

3.10. 7.11 Flat heating Coils - Water
3.11 7.13 Flow given in English Units - Oil

Fuel oil at a specific gravity of 0.815 (kinematic viscosity of 2.7 centistokes) flows at 2 inch, schedule 40 steel pipe 100 foot long at a rate of 2 US gallons/second.

Calculate the pressure drop in bars and psi.

The pressure drop calculated in the example is 66500 Pa (9.65 psi). The discrepancy is from their friction factor; they use 0.0230. The result is matched exactly if their friction factor is used.

3.12 7.14 Bernoulli’s Theorem-Water

Water at 15 degrees Celsius is flowing through the piping system shown in Crane TP 410M’s example at 1500 L/min. Calculate the velocity in both 4 and 5 inch sizes; and the pressure drop.
Note: This problem suggests to handle the changing size elbow by adding on the result of a smooth expansion, which is also used here.

```python
[1]:
from math import *
from fluids.units import *
from thermo.units import Chemical

water = Chemical('water', P=2*u.bar, T=15*u.degC)
rho = water.rho
mu = water.mu

Q = 1500*u.L/u.min
r_d = 1.5
_, D1, _, _ = nearest_pipe(Di=100*u.mm)
_, D2, _, _ = nearest_pipe(Di=125*u.mm)
L1 = 34*u.m
L2 = (22+45)*u.m
dH = 22*u.m
beta = D1/D2

V1 = Q/(pi/4*D1**2)
V2 = Q/(pi/4*D2**2)
Re1 = Reynolds(rho=rho, mu=mu, V=V1, D=D1)
Re2 = Reynolds(rho=rho, mu=mu, V=V2, D=D2)
fd1 = friction_factor(Re=Re1, eD=0.0018*u.inch/D1)
fd2 = friction_factor(Re=Re2, eD=0.0018*u.inch/D2)
fd = (fd1+fd2)/2
dP = rho*u.gravity*dH
K_D1 = bend_rounded(Di=D1, angle=90*u.degrees, fd=fd, bend_diameters=r_d)
K_D1 += diffuser_conical(D1, D2, angle=30*u.degrees, fd=fd)
K_D1 += K_from_f(fd=fd1, L=L1, D=D1)
K_D2 = bend_rounded(Di=D2, angle=90*u.degrees, fd=fd, bend_diameters=r_d)
K_D2 += K_from_f(fd=fd2, L=L2, D=D2)
dP += dP_from_K(K=K_D1, rho=rho, V=V1)
dP += dP_from_K(K=K_D2, rho=rho, V=V2)
dP.to(u.Pa)

[1]: 332625.262817 pascal

The result calculated in Crane’s TP 410m is 26450 Pa. Their friction factor is 0.018. Again, it that value is used, the result calculated matches theirs - except this is off by an order of magnitude.

In this edition, the gravitational term was forgotten. The prior 8th edition lists a value of 2.6 bar as the result for this problem. If their friction factor is used with this model, the following calculates a pressure drop of 2.62 bar.

```python
[2]:
fd = fd1 = fd2 = .018
dP = rho*u.gravity*dH
K_D1 = bend_rounded(Di=D1, angle=90*u.degrees, fd=fd, bend_diameters=r_d)
K_D1 += contraction_round(D1, D2, r_d=D1)
K_D1 += K_from_f(fd=fd1, L=L1, D=D1)
K_D2 = bend_rounded(Di=D2, angle=90*u.degrees, fd=fd, bend_diameters=r_d)
K_D2 += K_from_f(fd=fd2, L=L2, D=D2)
```

(continues on next page)
\[ dP += dP_{\text{from}_K(K=K_D1, \rho=\rho, V=V1)} \]
\[ dP += dP_{\text{from}_K(K=K_D2, \rho=\rho, V=V2)} \]
\[ dP \to (\text{u.Pa}) \]

\[ 262439.430838 \text{ pascal} \]

### 3.13 7.29 Orifice Flow Rate Calculation

```python
from fluids.units import *
from thermo.units import Chemical

P1 = 2*u.bar  # The full set of equations requires actual pressures not just the pressure difference, so an initial pressure of 2 bar is assumed.
P2 = P1 - 17*u.kPa
taps = 'D'
meter_type = 'ISO 5167 orifice'
Do = 50*u.mm

NPS, D, _, t_pipe = nearest_pipe(Do=80*u.mm, schedule=80)

D

0.07366 meter

water = Chemical('water', T=15*u.degC, P=P1)
rho = water.rho
mu = water.mu
k = water.isentropic_exponent
rho, mu, k

(<Quantity(999.149354395, 'kilogram / meter ** 3')>,
 <Quantity(0.00113751328021, 'pascal * second')>,
 <Quantity(1.32952393955, 'dimensionless')>)

m = differential_pressure_meter_solver(D=D, D2=Do, P1=P1, P2=P2, rho=rho, mu=mu, k=k, meter_type=meter_type, taps=taps)
print(m)
Q = m/rho
print(Q.to('L/min'))

7.70427741167 kilogram / second
462.650196056 liter / minute
```

The answer given in TP410M is 478 gallons/minute; however the errata specifies this value is in units of liters/minute instead.

This calculation matches their result well, given they did not include expansivity in their calculations and read a value of C from a graph.

A calculator at flow of fluids, which also does not include expansivity, gives an answer of 476.777 L/min, along with 7.93916 kg/s (http://www.flowoffluids.com/calculators/flow-through-orifices.aspx).
Get the latest version of fluids from https://pypi.python.org/pypi/fluids/

If you have an installation of Python with pip, simple install it with:

$$\text{pip install fluids}$$

Alternatively, if you are using \texttt{conda} as your package management, you can simply install fluids in your environment from \texttt{conda-forge} channel with:

$$\text{conda install -c conda-forge fluids}$$

To get the git version, run:

$$\text{git clone git://github.com/CalebBell/fluids.git}$$
The latest development version of fluids’s sources can be obtained at

https://github.com/CalebBell/fluids
Bug reports

To report bugs, please use the fluids’s Bug Tracker at:

https://github.com/CalebBell/fluids/issues

If you have further questions about the usage of the library, feel free to contact the author at Caleb.Andrew.Bell@gmail.com.
Fluids is MIT licensed. See LICENSE.txt for full information on the terms & conditions for usage of this software, and a DISCLAIMER OF ALL WARRANTIES.

Although not required by the fluids license, if it is convenient for you, please cite fluids if used in your work. Please also consider contributing any changes you make back, such that they may be incorporated into the main library and all of us will benefit from them.
To cite fluids in publications use:

Caleb Bell (2016-2018). fluids: Fluid dynamics component of Chemical Engineering Design Library (ChEDL)
https://github.com/CalebBell/fluids.
CHAPTER 9

Indices and tables

- genindex
- modindex
- search


Hedin, A. E., M. A. Biondi, R. G. Burnside, G. Hernandez, R. M. Johnson, T. L. Killeen, C. Maza-
udier, et al. “Revised Global Model of Thermosphere Winds Using Satellite and Ground-Based Ob-
doi:10.1029/91JA00251.

al. “Empirical Wind Model for the Upper, Middle and Lower Atmosphere.” Journal of Atmospheric and 

Drob, Douglas P., John T. Emmert, John W. Meriwether, Jonathan J. Makela, Eelco Doornbos, 
Mark Conde, Gonzalo Hernandez, et al. “An Update to the Horizontal Wind Model (HWM): The 
Quiet Time Thermosphere.” Earth and Space Science 2, no. 7 (July 1, 2015): 2014EA000089. 


2012.


Coelho, Paulo M., and Carlos Pinho. “Considerations about Equations for Steady State Flow in Natu-
ral Gas Pipelines.” Journal of the Brazilian Society of Mechanical Sciences and Engineering 29, no. 3 


2012.


Coelho, Paulo M., and Carlos Pinho. “Considerations about Equations for Steady State Flow in Natu-
ral Gas Pipelines.” Journal of the Brazilian Society of Mechanical Sciences and Engineering 29, no. 3 


[1] IEC 60534-2-1 / ISA-75.01.01-2007

[1] IEC 60534-2-1 / ISA-75.01.01-2007


[1] IEC 60534-2-1 / ISA-75.01.01-2007

[1] IEC 60534-2-1 / ISA-75.01.01-2007

[1] IEC 60534-2-1 / ISA-75.01.01-2007

[1] IEC 60534-2-1 / ISA-75.01.01-2007

[1] IEC 60534-2-1 / ISA-75.01.01-2007

[1] IEC 60534-2-1 / ISA-75.01.01-2007


H. Rouse, Fluid Mechanics for Hydraulic Engineers, Dover, New York, N.Y., 1938


F. Engelund, E. Hansen, Monograph on Sediment Transport in Alluvial Streams, Monograph Denmark Technical University, Hydraulic Lab, Denmark, 1967.


2. Hooper, William B. “Calculate Head Loss Caused by Change in Pipe Size.” Chemical Engineering 95, no. 16 (November 7, 1988): 89.


1. ISA-75.01.01-2007 (60534-2-1 Mod) Draft

1. ISA-75.01.01-2007 (60534-2-1 Mod) Draft

1. ISA-75.01.01-2007 (60534-2-1 Mod) Draft

1. ISA-75.01.01-2007 (60534-2-1 Mod) Draft

1. ISA-75.01.01-2007 (60534-2-1 Mod) Draft

1. ISA-75.01.01-2007 (60534-2-1 Mod) Draft


[1] HI 1.3 Rotodynamic Centrifugal Pumps for Design and Applications


Bibliography 491


Bibliography 493


fluids.atmosphere, 29
fluids.compressible, 42
fluids.control_valve, 63
fluids.core, 76
fluids.drag, 109
fluids.filters, 124
fluids.fittings, 128
fluids.flow_meter, 179
fluids.friction, 200
fluids.geometry, 244
fluids.jet_pump, 280
fluids.mixing, 283
fluids.open_flow, 287
fluids.packed_bed, 294
fluids.packed_tower, 309
fluids.particle_size_distribution, 318
fluids.piping, 345
fluids.pump, 348
fluids.safety_valve, 354
fluids.saltation, 365
fluids.separator, 361
fluids.two_phase, 371
fluids.two_phase_voidage, 397
A

A_cylinder() (in module fluids.geometry), 276
A_hollow_cylinder() (in module fluids.geometry), 277
A_multiple_hole_cylinder() (in module fluids.geometry), 278
a_torispherical() (in module fluids.geometry), 270
add_thickness() (fluids.geometry.TANK method), 247
agitator_time_homogeneous() (in module fluids.mixing), 283
AirCooledExchanger (class in fluids.geometry), 257
airmass() (in module fluids.atmosphere), 34
Almedeij() (in module fluids.drag), 122
Alshul_1952() (in module fluids.friction), 211
API520_A_g() (in module fluids.safety_valve), 359
API520_A_steam() (in module fluids.safety_valve), 359
API520_B() (in module fluids.safety_valve), 358
API520_C() (in module fluids.safety_valve), 355
API520_F2() (in module fluids.safety_valve), 356
API520_N() (in module fluids.safety_valve), 357
API520_round_size() (in module fluids.safety_valve), 354
API520_SH() (in module fluids.safety_valve), 357
API520_W() (in module fluids.safety_valve), 359
Archimedes() (in module fluids.core), 100
Armand() (in module fluids.two_phase_voidage), 401
aspect_ratio() (in module fluids.geometry), 275
ASTM_E11_sieves (in module fluids.particle_size_distribution), 344
ATMOSPHERE_1976 (class in fluids.atmosphere), 29
ATMOSPHERE_NRLMSISE00 (class in fluids.atmosphere), 32
attrrs (fluids.atmosphere.ATMOSPHERE_NRLMSISE00 attribute), 34
Avci_Karagoz_2009() (in module fluids.friction), 223

B

Bankoff() (in module fluids.two_phase), 396
Barati() (in module fluids.drag), 113
Barati_high() (in module fluids.drag), 114
Baroczy() (in module fluids.two_phase_voidage), 404
Baroczy_Chisholm() (in module fluids.two_phase), 382
Barr_1981() (in module fluids.friction), 216
Beattie_Whalley() (in module fluids.two_phase_voidage), 418
Beggs_Blill() (in module fluids.two_phase), 376
Bejan_L() (in module fluids.core), 97
Bejan_p() (in module fluids.core), 98
bend_miter() (in module fluids.fittings), 154
bend_miter_Miller() (in module fluids.fittings), 156
bend_rounded() (in module fluids.fittings), 150
bend_rounded_Crane() (in module fluids.fittings), 152
bend_rounded_Miller() (in module fluids.fittings), 151
Biot() (in module fluids.core), 91
Blasius() (in module fluids.friction), 226
Boiling() (in module fluids.core), 99
Bond() (in module fluids.core), 87
Brauer() (in module fluids.packed_bed), 299
Brkic_2011_1() (in module fluids.friction), 224
Brkic_2011_2() (in module fluids.friction), 225
Buzzelli_2008() (in module fluids.friction), 223

C

C_Chezy_to_n_Manning() (in module fluids.open_flow), 293
c_ideal_gas() (in module fluids.core), 103
C_ISA_1932_nozzle() (in module fluids.flow_meter), 189
C_long_radius_nozzle() (in module fluids.flow_meter), 188
Index

E

earthsun_distance() (in module fluids.atmosphere), 39
Eck_1973() (in module fluids.friction), 212
Eckert() (in module fluids.core), 94
Engelund_Hansen() (in module fluids.drag), 115
entrance_angled() (in module fluids.fittings), 142
entrance_beveled() (in module fluids.fittings), 146
entrance_beveled_orifice() (in module fluids.fittings), 148
entrance_distance() (in module fluids.fittings), 141
entrance_distance_45_Miller() (in module fluids.fittings), 148
entrance_rounded() (in module fluids.fittings), 144
entrance_sharp() (in module fluids.fittings), 141
Eotvos() (in module fluids.core), 109
Erdim_Akgiray_Demir() (in module fluids.packed_bed), 301
Ergun() (in module fluids.packed_bed), 295
Euler() (in module fluids.core), 92
exit_normal() (in module fluids.fittings), 149

F

Fahien_Schriver() (in module fluids.packed_bed), 301
Fang_2011() (in module fluids.friction), 226

fluids

fluids.body(39)
fluids.body(297)
fluids.body(76)
fluids.body(37)
fluids.body(329)

Fluids Documentation, Release 0.1
interpolated (fluids.particle_size_distribution.ParticleSizeDistribution, 329)
is_choked_turbulent_g() (in module fluids.control_valve, 68)
is_choked_turbulent_l() (in module fluids.control_valve, 67)
is_critical_flow() (in module fluids.compressible, 60)
isentropic_efficiency() (in module fluids.compressible, 57)
isentropic_T_rise_compression() (in module fluids.compressible, 58)
isentropic_work_compression() (in module fluids.compressible, 55)
ISO_3310_1_R10 (in module fluids.particle_size_distribution), 344
ISO_3310_1_R20 (in module fluids.particle_size_distribution), 344
ISO_3310_1_R20_3 (in module fluids.particle_size_distribution), 344
ISO_3310_1_R40_3 (in module fluids.particle_size_distribution), 344
ISO_3310_1_sieves (in module fluids.particle_size_distribution), 344
isothermal_gas() (in module fluids.compressible), 52
isothermal_work_compression() (in module fluids.compressible), 54

J
Jain_1976() (in module fluids.friction), 213
Jakob() (in module fluids.core), 94
Jones_Krier() (in module fluids.packed_bed), 297
Jung_Radermacher() (in module fluids.two_phase), 387

K
K_angle_stop_check_valve_Crane() (in module fluids.fittings), 169
K_angle_valve_Crane() (in module fluids.fittings), 165
K_ball_valve_Crane() (in module fluids.fittings), 170
K_branch_converging_Crane() (in module fluids.fittings), 174
K_branch_diverging_Crane() (in module fluids.fittings), 176
K_butterfly_valve_Crane() (in module fluids.fittings), 172
K_diaphragm_valve_Crane() (in module fluids.fittings), 171
K_foot_valve_Crane() (in module fluids.fittings), 172
K_from_f() (in module fluids.core), 105
K_gate_valve_Crane() (in module fluids.fittings), 164
K_globe_check_valve_Crane() (in module fluids.fittings), 169
K_globe_valve_Crane() (in module fluids.fittings), 166
K_lift_check_valve_Crane() (in module fluids.fittings), 167
K_motiveless_mixer() (in module fluids.mixing), 287
K_plug_valve_Crane() (in module fluids.fittings), 173
K_run_converging_Crane() (in module fluids.fittings), 175
K_run_diverging_Crane() (in module fluids.fittings), 177
K_separator_demister_York() (in module fluids.separator), 363
K_separator_Watkins() (in module fluids.separator), 362
K_Sounders_Brown_theoretical() (in module fluids.separator), 364
K_swing_check_valve_Crane() (in module fluids.fittings), 166
K_tilting_disk_check_valve_Crane() (in module fluids.fittings), 168
K_to_Cv() (in module fluids.fittings), 163
K_to_discharge_coefficient() (in module fluids.flow_meter), 185
K_to_Kv() (in module fluids.fittings), 161
Kawahara() (in module fluids.two_phase voidage), 403
Khan_Richardson() (in module fluids.drag), 118
Kim_Mudawar() (in module fluids.two_phase), 381
Knudsen() (in module fluids.core), 87
Kopte_Newell_Chato() (in module fluids.two_phase voidage), 409
Kp_helical_ribbon_Rieger() (in module fluids.mixing), 284
KTA() (in module fluids.packed_bed), 300
Kuo_Nydegger() (in module fluids.packed_bed), 296
Kv_to_Cv() (in module fluids.fittings), 159
Kv_to_K() (in module fluids.fittings), 160

L
L_equiv_from_K() (in module fluids.core), 106
L_from_K() (in module fluids.core), 107
LAMINAR_TRANSITION_PIPE (in module fluids.friction), 240
Lewis() (in module fluids.core), 84
Lin_Kwok() (in module fluids.two_phase voidage), 420
M

liquid_gas_voidage() (in module fluids.two_phase_voidage), 421
liquid_jet_pump() (in module fluids.jet_pump), 280
liquid_jet_pump_ancillary() (in module fluids.jet_pump), 282
Lockhart_Martinelli() (in module fluids.two_phase), 377
Lockhart_Martinelli_Xtt() (in module fluids.two_phase_voidage), 416
Lombardi_Pedrocchi() (in module fluids.two_phase), 386
loss_coefficient_piping() (in module fluids.control_valve), 69

Mach() (in module fluids.core), 86
Manadilli_1997() (in module fluids.friction), 220
material_roughness() (in module fluids.friction), 208
Matsumoto_1974() (in module fluids.salvation), 366
Matsumoto_1975() (in module fluids.salvation), 367
Matsumoto_1977() (in module fluids.salvation), 368
McAdams() (in module fluids.two_phase_voidage), 419
mean_size() (fluids.particle_size_distribution.ParticleSizeDistributionContinuous, method), 329
mean_size() (fluids.particle_size_distribution.ParticleSizeDistribution method), 322
mean_size_ISO() (fluids.particle_size_distribution.ParticleSizeDistributionContinuous, method), 331
mean_size_ISO() (fluids.particle_size_distribution.ParticleSizeDistribution method), 323
Mikhailov_Freire() (in module fluids.drag), 121
Mishima_Hibiki() (in module fluids.two_phase), 395
Montillet_Akkari_Comiti() (in module fluids.packed_bed), 305
Moody() (in module fluids.friction), 210
Morrison() (in module fluids.drag), 123
Morsi_Alexander() (in module fluids.drag), 116
Morton() (in module fluids.core), 88
motor_efficiency_underloaded() (in module fluids.pump), 349
motor_round_size() (in module fluids.pump), 353
Muller() (in module fluids.compressible), 50
Muller_Steinhagen_Heck() (in module fluids.two_phase), 384
MWs (fluids.atmosphere.ATMOSPHERE_NRLMSISE00 attribute), 34

N

N_Manning_to_C_Chezy() (in module fluids.open_flow), 292
name (fluids.particle_size_distribution.ParticleSizeDistribution attribute), 332
name (fluids.particle_size_distribution.PSDCustom attribute), 336
name (fluids.particle_size_distribution.PSDGaussGaudinSchuhman attribute), 335
name (fluids.particle_size_distribution.PSDInterpolated attribute), 333
name (fluids.particle_size_distribution.PSDLognormal attribute), 334
name (fluids.particle_size_distribution.PSDRosinRammler attribute), 336

nearest_material_roughness() (in module fluids.friction), 209

nearest_pipe() (in module fluids.piping), 345
nema_sizes (in module fluids.pump), 353
nema_sizes_hp (in module fluids.pump), 353
Nicklin_Wilkes_Davidson() (in module fluids.two_phase_voidage), 412
Nishino_Yamazaki() (in module fluids.two_phase_voidage), 402
nozzle_expansibility() (in module fluids.flow_meter), 188
nozzle_expansibility() (in module fluids.core), 104
Nusselt() (in module fluids.core), 78

Ohnesorge() (in module fluids.core), 101

Ohnesorge_() (in module fluids.core), 101
one_phase_dP() (in module fluids.compressible), 48
one_phase_dP() (in module fluids.friction), 205
one_phase_dP() (in module fluids.drag), 104
one_phase_dP() (in module fluids.core), 104

one_phase_dPAcceleration() (in module fluids.compressible), 205
one_phase_dPAcceleration() (in module fluids.core), 104
one_phase_dPGravitational() (in module fluids.friction), 206
one_phase_dPAcceleration() (in module fluids.core), 104
one_phase_dPGravitational() (in module fluids.core), 104

orifice_expansibility() (in module fluids.flow_meter), 184
orifice_expansibility_1989() (in module fluids.flow_meter), 190

P

P_critical_flow() (in module fluids.compressible), 59
P_from_head() (in module fluids.core), 108
P_isothermal_critical_flow() (in module fluids.compressible), 59
P_stagnation() (in module fluids.compressible), 61
Panhandle_A() (in module fluids.compressible), 42
Panhandle_B() (in module fluids.compressible), 43
Papaevangelou_2010() (in module fluids.friction), 224
Rouhani_2 (module fluids.two_phase_voidage), 411
Round_1980 (module fluids.friction), 215
round_edge_grill (module fluids.filters), 127
round_edge_open_mesh (module fluids.filters), 125
round_edge_screen (module fluids.filters), 124
Rouse (module fluids.drag), 114

S
SA_conical_head (module fluids.geometry), 271
SA_ellipsoidal_head (module fluids.geometry), 271
SA_guppy_head (module fluids.geometry), 272
SA_partial_sphere (module fluids.geometry), 260
SA_tank (module fluids.geometry), 274
SA_torispheroidal (module fluids.geometry), 272
Schade (module fluids.salitation), 369
Schmidt (module fluids.core), 80
separation_demister_ElDessouky (module fluids.packed_tower), 316
Sergides_1 (module fluids.friction), 219
Sergides_2 (module fluids.friction), 219
set_chebyshev_approximators (fluids.geometry.TANK method), 248
set.misc (fluids.geometry.TANK method), 248
set_overall_geometry (fluids.geometry.RectangularFinExchanger method), 254
set_table (fluids.geometry.TANK method), 248
Shacham_1980 (module fluids.friction), 216
Sherwood (module fluids.core), 79
Sieve (class in fluids.particle_size_distribution), 343
size_classes (fluids.particle_size_distribution.ParticleSizeDistribution.attribute), 332
size_control_valve_g (module fluids.control_valve), 64
size_control_valve_l (module fluids.control_valve), 63
size_tee (module fluids.mixing), 285
Smith (module fluids.two_phase_voidage), 398
solar_irradiation (module fluids.atmosphere), 36
solar_position (module fluids.atmosphere), 35
solve_tank_for_V (fluids.geometry.TANK method), 248
Song_Xu (module fluids.drag), 123
sonic_velocity (fluids.atmosphere.ATMOSPHERE_1976 static method), 31
Sonnad_Goudar_2006 (module fluids.friction), 222
specific_area_mesh (module fluids.packed_tower), 317
specific_diameter (module fluids.pump), 352
specific_speed (module fluids.pump), 351
speed_synchronous (module fluids.pump), 352
sphericity (module fluids.geometry), 275
spiral (module fluids.filters), 157
Spitzglass_high (module fluids.compressible), 45
Spitzglass_low (module fluids.compressible), 46
square_edge_grill (module fluids.filters), 126
square_edge_screen (module fluids.filters), 126
stagnation_energy (module fluids.compressible), 61
Stanton (module fluids.core), 92
Steiner (module fluids.two_phase_voidage), 409
Stichlmair_dry (module fluids.packed_tower), 310
Stichlmair_flood (module fluids.packed_tower), 312
Stichlmair_wet (module fluids.packed_tower), 311
Stokes (module fluids.drag), 113
Stokes_number (module fluids.core), 95
Strouhal (module fluids.core), 90
Sun_Duffey_Peng (module fluids.two_phase_voidage), 414
sunrise_sunset (module fluids.atmosphere), 38
Suratman (module fluids.core), 101
Swamee_Jain_1976 (module fluids.friction), 213
Swamee_Ojha (module fluids.drag), 118

T
T_critical_flow (module fluids.compressible), 58
t_from_gauge (module fluids.piping), 347
T_stagnation (module fluids.compressible), 62
T_stagnation_ideal (module fluids.compressible), 62
table (fluids.geometry.TANK attribute), 248
Tallmadge (module fluids.packed_bed), 302
Tandon_Varma_Gupta (module fluids.two_phase_voidage), 404
TANK (class in fluids.geometry), 244
Terfous (module fluids.drag), 120
Theissing (module fluids.two_phase), 383
Woldesemayat_Ghajar() (in module fluids.two_phase_voidage), 415
Wood_1966() (in module fluids.friction), 211

X
Xu_Fang() (in module fluids.two_phase), 390
Xu_Fang_voidage() (in module fluids.two_phase_voidage), 415

Y
Yashar() (in module fluids.two_phase_voidage), 407
Yen() (in module fluids.drag), 119
Yu_France() (in module fluids.two_phase), 391

Z
Zhang_Hibiki_Mishima() (in module fluids.two_phase), 394
Zhang_Webb() (in module fluids.two_phase), 389
Zigrang_Sylvester_1() (in module fluids.friction), 217
Zigrang_Sylvester_2() (in module fluids.friction), 217
Zivi() (in module fluids.two_phase_voidage), 397