ESSM Documentation

Release 1.0.2.dev1

Stan Schymanski

Dec 18, 2020

Contents

This package contains helpers to deal with physical variables and units.

The documentation is available on https://essm.rtfd.io/.

The quick installation for impatient users can be done by running:

```
pip install essm
```

CHAPTER 1

User's Guide

This part of the documentation will show you how to get started in using ESSM.

1.1 Installation

ESSM is on PyPI so all you need is:

```
$ pip install essm
```

1.1.1 Development

If you are planning to modify source code or generate new variables or equations, please install the package in development mode:

```
$ git clone https://github.com/environmentalscience/essm
$ cd environmental-science-for-sagemath
$ pip install -e .[all]
```

1.2 Usage

Variables module to deal with physical variables and units.

It allows attaching docstrings to variable names, defining their domains (e.g. integer, real or complex), their units and LaTeX representations. You can also provide a default value, which is particularly useful for physical constants.

1.2.1 Creating variables

To create custom variables, first import *Variable*:

```
>>> from essm.variables import Variable
```

To define units, you must first import these units from the library:

```
>>> from essm.variables.units import joule, kelvin, meter
```

Then you can define a custom variable with its name, description, domain, latex_name, unit, and an optional default value, e.g.:

```
>>> class demo_chamber_volume(Variable):
...     '''Volume of chamber.'''
...     latex_name = 'V_c'
...     domain = 'real'
...     name = 'V_c'
...     unit = meter ** 3
...     default = 1
```

```
>>> class demo_chamber_length(Variable):
...     '''Length of chamber.'''
...     latex_name = 'L_c'
...     domain = 'real'
...     name = 'L_c'
...     unit = meter
...     default = 1
```

Now, *demo_chamber_volume* is displayed as *V_c* and *demo_chamber_length* is displayed as *L_c*. You can type *help(demo_chamber_volume)* to inspects its metadata.

Variable.__defaults__ returns a dictionary with all variables and their default values, *Variable.__units__* returns their units, and *demo_chamber_volume.short_unit()* can be used to obtain the units in short notation.

1.2.2 Creating equations

To create custom equations, proceed similarly to above, i.e. first import *Equation*:

```
>>> from essm.equations import Equation
```

Then define an equation, e.g.:

```
>>> class demo_eq_volume(Equation):
...     '''Calculate chamber volume.
...
...     Uses :math:`V_{l}` and assumes cubic shape.
...     '''
...     expr = demo_chamber_volume == demo_chamber_length ** 3
```

1.2.3 Importing variables and equations

You can import pre-defined variables and equations as e.g.:

```
>>> from essm.variables.physics.thermodynamics import *
>>> from essm.equations.physics.thermodynamics import *
```

CHAPTER 2

Importable variables and equations

Here you find examples for importing pre-defined variables and equations fromm ESSM.

2.1 Importable variables and equations

This jupyter notebook can be found at: https://github.com/environmentalscience/essm/blob/master/docs/examples/importable_variables_equations.ipynb

Below, we will import some generic python packages that are used in this notebook:

[1]:
```
# Checking for essm version installed
import pkg_resources
pkg_resources.get_distribution("essm").version
```

[1]:
```
'0.4.2.dev5'
```

[14]:
```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:150% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

[4]:
```
from IPython.display import display
from sympy import init_printing, latex
init_printing()
from sympy.printing import StrPrinter
StrPrinter._print_Quantity = lambda self, expr: str(expr.abbrev)    # displays short␣
↪units (m instead of meter)
```

[5]:
```
import scipy as sc
# Import various functions from sympy
from sympy import Derivative, Eq, exp, log, solve, Symbol
```

[6]:
```
from essm.variables.utils import generate_metadata_table, ListTable
```

2.1.1 General physics variables and equations

Variables

Pre-defined thermodynamics variables can be imported from `essm.variables.physics.thermodynamics`:

[7]:
```
import essm.variables.physics.thermodynamics as physics_vars
vars = ['physics_vars.' + name for name in physics_vars.__all__]
generate_metadata_table([eval(name) for name in vars])
```

[7]:
```
[('Symbol', 'Name', 'Description', 'Definition', 'Default value', 'Units'),
 ('$\\alpha_a$',
  'alpha_a',
  'Thermal diffusivity of dry air.',
  '',
  '-',
  'm$^{2}$ s$^{-1}$'),
 ('$\\lambda_E$',
  'lambda_E',
  'Latent heat of evaporation.',
  '',
  '2450000.0',
  'J kg$^{-1}$'),
 ('$\\nu_a$',
  'nu_a',
  'Kinematic viscosity of dry air.',
  '',
  '-',
  'm$^{2}$ s$^{-1}$'),
 ('$\\rho_a$', 'rho_a', 'Density of dry air.', '', '-', 'kg m$^{-3}$'),
 ('$\\sigma$',
  'sigm',
  'Stefan-Boltzmann constant.',
  '',
  '5.67e-08',
  'J s$^{-1}$ m$^{-2}$ K$^{-4}$'),
 ('$c_{pa,mol}$',
  'c_pamol',
  'Molar specific heat of dry air.\n\n    https://en.wikipedia.org/wiki/Heat_capacity
↪#Specific_heat_capacity\n    ',
  '',
  '29.19',
  'J K$^{-1}$ mol$^{-1}$'),
 ('$c_{pa}$',
  'c_pa',
  'Specific heat of dry air.',
  '',
  '1010.0',
  'J K$^{-1}$ kg$^{-1}$'),
 ('$c_{pv}$',
  'c_pv',
  'Specific heat of water vapour at 300 K.\n\n    http://www.engineeringtoolbox.com/
↪water-vapor-d_979.html\n    ',
  '',
```

(continues on next page)

(continued from previous page)

```
 '1864',
 'J K$^{-1}$ kg$^{-1}$'),
('$C_{wa}$',
 'C_wa',
 'Concentration of water in air.',
 '',
 '-',
 'mol m$^{-3}$'),
('$D_{va}$',
 'D_va',
 'Binary diffusion coefficient of water vapour in air.',
 '',
 '-',
 'm$^{2}$ s$^{-1}$'),
('$g$', 'g', 'Gravitational acceleration.', '', '9.81', 'm s$^{-2}$'),
('$h_c$',
 'h_c',
 'Average 1-sided convective heat transfer coefficient.',
 '',
 '-',
 'J K$^{-1}$ s$^{-1}$ m$^{-2}$'),
('$k_a$',
 'k_a',
 'Thermal conductivity of dry air.',
 '',
 '-',
 'J K$^{-1}$ m$^{-1}$ s$^{-1}$'),
('$M_w$', 'M_w', 'Molar mass of water.', '', '0.018', 'kg mol$^{-1}$'),
('$M_{air}$',
 'M_air',
 'Molar mass of air.\n\n    http://www.engineeringtoolbox.com/molecular-mass-air-d_
↪679.html\n    ',
 '',
 '0.02897',
 'kg mol$^{-1}$'),
('$M_{N_2}$',
 'M_N2',
 'Molar mass of nitrogen.',
 '',
 '0.028',
 'kg mol$^{-1}$'),
('$M_{O_2}$', 'M_O2', 'Molar mass of oxygen.', '', '0.032', 'kg mol$^{-1}$'),
('$N_{Gr_L}$', 'Gr', 'Grashof number.', '', '-', '1'),
('$N_{Le}$', 'Le', 'Lewis number.', '', '-', '1'),
('$N_{Nu_L}$',
 'Nu',
 'Average Nusselt number over given length.',
 '',
 '-',
 '1'),
('$N_{Pr}$', 'Pr', 'Prandtl number (0.71 for air).', '', '-', '1'),
('$N_{Re_c}$',
 'Re_c',
 'Critical Reynolds number for the onset of turbulence.',
 '',
 '-',
 '1'),
```

(continues on next page)

(continued from previous page)

```
('$N_{Re_L}$',
 'Re',
 'Average Reynolds number over given length.',
 '',
 '-',
 '1'),
('$P_a$', 'P_a', 'Air pressure.', '', '-', 'Pa'),
('$P_{N2}$', 'P_N2', 'Partial pressure of nitrogen.', '', '-', 'Pa'),
('$P_{O2}$', 'P_O2', 'Partial pressure of oxygen.', '', '-', 'Pa'),
('$P_{was}$',
 'P_was',
 'Saturation water vapour pressure at air temperature.',
 '',
 '-',
 'Pa'),
('$P_{wa}$',
 'P_wa',
 'Water vapour pressure in the atmosphere.',
 '',
 '-',
 'Pa'),
('$R_d$', 'R_d', 'Downwelling global radiation.', '', '-', 'W m$^{-2}$'),
('$R_s$',
 'R_s',
 'Solar shortwave flux per area.',
 '',
 '-',
 'J s$^{-1}$ m$^{-2}$'),
('$R_u$', 'R_u', 'Upwelling global radiation.', '', '-', 'W m$^{-2}$'),
('$R_{mol}$',
 'R_mol',
 'Molar gas constant.',
 '',
 '8.314472',
 'J K$^{-1}$ mol$^{-1}$'),
('$T_0$', 'T0', 'Freezing point in Kelvin.', '', '273.15', 'K'),
('$T_a$', 'T_a', 'Air temperature.', '', '-', 'K'),
('$v_w$', 'v_w', 'Wind velocity.', '', '-', 'm s$^{-1}$'),
('$x_{N2}$',
 'x_N2',
 'Mole fraction of nitrogen in dry air.',
 '',
 '0.79',
 '1'),
('$x_{O2}$', 'x_O2', 'Mole fraction of oxygen in dry air.', '', '0.21', '1')]
```

Each of the above can also be imported one-by-one, using its Name, e.g.:

```
from essm.variables.physics.thermodynamics import R_mol
```

Equations

General equations based on the above variables can be imported from `essm.equations.physics.thermodynamics`:

[8]:
```
import essm.equations.physics.thermodynamics as physics_eqs
modstr = 'physics_eqs.'
eqs = [name for name in physics_eqs.__all__]
table = ListTable()
#table.append(('Name', 'Description', 'Equation'))
for name in eqs:
    table.append((name, eval(modstr+name).__doc__, latex('$'+latex(eval(modstr+name))+
↪'$')))
table
```

[8]:
```
[('eq_Le',
  'Le as function of alpha_a and D_va.\n\n    (Eq. B3 in :cite:`schymanski_leaf-scale_
↪2017`)\n    ',
  '$N_{Le} = \\frac{\\alpha_a}{D_{va}}$'),
 ('eq_Cwa',
  'C_wa as a function of P_wa and T_a.\n\n    (Eq. B9 in :cite:`schymanski_leaf-scale_
↪2017`)\n    ',
  '$C_{wa} = \\frac{P_{wa}}{R_{mol} T_a}$'),
 ('eq_Nu_forced_all',
  'Nu as function of Re and Re_c under forced conditions.\n\n    (Eqs. B13--B15 in :
↪cite:`schymanski_leaf-scale_2017`)\n    ',
  '$N_{Nu_L} = - \\frac{\\sqrt[3]{N_{Pr}} \\left(- 37 N_{Re_L}^{\\frac{4}{5}} + 37␣
↪\\left(N_{Re_L} + N_{Re_c} - \\frac{\\left|{N_{Re_L} - N_{Re_c}}\\right|}{2}
↪\\right)^{\\frac{4}{5}} - 664 \\sqrt{N_{Re_L} + N_{Re_c} - \\frac{\\left|{N_{Re_L} -
↪ N_{Re_c}}\\right|}{2}}\\right)}{1000}$'),
 ('eq_Dva',
  'D_va as a function of air temperature.\n\n    (Table A.3 in :cite:`monteith_
↪principles_2007`)\n    ',
  '$D_{va} = T_a p_1 - p_2$'),
 ('eq_alphaa',
  'alpha_a as a function of air temperature.\n\n    (Table A.3 in :cite:`monteith_
↪principles_2007`)\n    ',
  '$\\alpha_a = T_a p_1 - p_2$'),
 ('eq_ka',
  'k_a as a function of air temperature.\n\n    (Table A.3 in :cite:`monteith_
↪principles_2007`)\n    ',
  '$k_a = T_a p_1 + p_2$'),
 ('eq_nua',
  'nu_a as a function of air temperature.\n\n    (Table A.3 in :cite:`monteith_
↪principles_2007`)\n    ',
  '$\\nu_a = T_a p_1 - p_2$'),
 ('eq_rhoa_Pwa_Ta',
  'rho_a as a function of P_wa and T_a.\n\n    (Eq. B20 in :cite:`schymanski_leaf-
↪scale_2017`)\n    ',
  '$\\rho_a = \\frac{M_{N_2} P_{N2} + M_{O_2} P_{O2} + M_w P_{wa}}{R_{mol} T_a}$'),
 ('eq_Pa',
  "Calculate air pressure.\n\n    From partial pressures of N2, O2 and H2O, following␣
↪Dalton's law of\n    partial pressures.\n    ",
  '$P_a = P_{N2} + P_{O2} + P_{wa}$'),
 ('eq_PN2_PO2',
  "Calculate P_N2 as a function of P_O2.\n\n    It follows Dalton's law of partial␣
↪pressures.\n    ",
  '$P_{N2} = \\frac{P_{O2} x_{N2}}{x_{O2}}$'),
 ('eq_PO2',
  'Calculate P_O2 as a function of P_a, P_N2 and P_wa.',
  '$P_{O2} = \\frac{P_a x_{O2} - P_{wa} x_{O2}}{x_{N2} + x_{O2}}$'),
 ('eq_PN2',
  'Calculate P_N2 as a function of P_a, P_O2 and P_wa.',
```

(continues on next page)

(continued from previous page)

```
  '$P_{N2} = \\frac{P_a x_{N2} - P_{wa} x_{N2}}{x_{N2} + x_{O2}}$'),
 ('eq_rhoa',
  'Calculate rho_a from T_a, P_a and P_wa.',
  '$\\rho_a = \\frac{x_{N2} \\left(M_{N_2} P_a - P_{wa} \\left(M_{N_2} - M_
↪w\\right)\\right) + x_{O2} \\left(M_{O_2} P_a - P_{wa} \\left(M_{O_2} - M_
↪w\\right)\\right)}{R_{mol} T_a x_{N2} + R_{mol} T_a x_{O2}}$')]
```

2.1.2 Variables and equations related to plant leaves

These refer to the model by Schymanski & Or (2017).

Variables for leaf energy and water balance

Variables related to the model by can be imported from `essm.variables.leaf.energy_water`:

[9]:
```
import essm.variables.leaf.energy_water as leaf_energy
vars = ['leaf_energy.' + name for name in leaf_energy.__all__]
generate_metadata_table([eval(name) for name in vars])
```

```
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:Gr" will be overridden by "essm.variables.leaf.energy_water:
↪<class 'essm.variables.leaf.energy_water.Gr'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:h_c" will be overridden by "essm.variables.leaf.energy_water:
↪<class 'essm.variables.leaf.energy_water.h_c'>"
  instance[expr] = instance
```

[9]:
```
[('Symbol', 'Name', 'Description', 'Definition', 'Default value', 'Units'),
 ('$\\alpha_l$',
  'alpha_l',
  'Leaf albedo, fraction of shortwave radiation reflected by the leaf.',
  '',
  '-',
  '1'),
 ('$\\epsilon_l$',
  'epsilon_l',
  'Longwave emmissivity of the leaf surface.',
  '',
  '1.0',
  '1'),
 ('$\\rho_{al}$',
  'rho_al',
  'Density of air at the leaf surface.',
  '',
  '-',
  'kg m$^{-3}$'),
 ('$a_s$',
  'a_s',
  'Fraction of one-sided leaf area covered by stomata.\n\n    (1 if stomata are on␣
↪one side only, 2 if they are on both sides).\n    ',
  '',
  '-',
  '1'),
 ('$a_{sh}$',
```

(continues on next page)

(continued from previous page)

```
 'a_sh',
 'Fraction of projected area exchanging sensible heat with the air.',
 '',
 '2.0',
 '1'),
('$C_{wl}$',
 'C_wl',
 'Concentration of water in the leaf air space.',
 '',
 '-',
 'mol m$^{-3}$'),
('$E_l$',
 'E_l',
 'Latent heat flux from leaf.',
 '',
 '-',
 'J s$^{-1}$ m$^{-2}$'),
('$E_{l,mol}$',
 'E_lmol',
 'Transpiration rate in molar units.',
 '',
 '-',
 'mol s$^{-1}$ m$^{-2}$'),
('$g_{bw,mol}$',
 'g_bwmol',
 'Boundary layer conductance to water vapour.',
 '',
 '-',
 'mol s$^{-1}$ m$^{-2}$'),
('$g_{bw}$',
 'g_bw',
 'Boundary layer conductance to water vapour.',
 '',
 '-',
 'm s$^{-1}$'),
('$g_{sw,mol}$',
 'g_swmol',
 'Stomatal conductance to water vapour.',
 '',
 '-',
 'mol s$^{-1}$ m$^{-2}$'),
('$g_{sw}$',
 'g_sw',
 'Stomatal conductance to water vapour.',
 '',
 '-',
 'm s$^{-1}$'),
('$g_{tw,mol}$',
 'g_twmol',
 'Total leaf layer conductance to water vapour.',
 '',
 '-',
 'mol s$^{-1}$ m$^{-2}$'),
('$g_{tw}$',
 'g_tw',
 'Total leaf conductance to water vapour.',
 '',
```

(continues on next page)

(continued from previous page)

```
 '-',
 'm s$^{-1}$'),
('$h_c$',
 'h_c',
 'Average 1-sided convective heat transfer coefficient.',
 '',
 '-',
 'J K$^{-1}$ s$^{-1}$ m$^{-2}$'),
('$H_l$',
 'H_l',
 'Sensible heat flux from leaf.',
 '',
 '-',
 'J s$^{-1}$ m$^{-2}$'),
('$L_A$', 'L_A', 'Leaf area.', '', '-', 'm$^{2}$'),
('$L_l$',
 'L_l',
 'Leaf width as characteristic length scale for convection.',
 '',
 '-',
 'm'),
('$N_{Gr_L}$', 'Gr', 'Grashof number.', '', '-', '1'),
('$P_{wl}$', 'P_wl', 'Water vapour pressure inside the leaf.', '', '-', 'Pa'),
('$r_{bw}$',
 'r_bw',
 'Boundary layer resistance to water vapour, inverse of $g_{bw}$.',
 '',
 '-',
 's m$^{-1}$'),
('$R_{la}$',
 'R_la',
 'Longwave radiation absorbed by leaf.',
 '',
 '-',
 'W m$^{-2}$'),
('$R_{ld}$',
 'R_ld',
 'Downwards emitted/reflected global radiation from leaf.',
 '',
 '-',
 'W m$^{-2}$'),
('$R_{ll}$',
 'R_ll',
 'Longwave radiation away from leaf.',
 '',
 '-',
 'W m$^{-2}$'),
('$R_{lu}$',
 'R_lu',
 'Upwards emitted/reflected global radiation from leaf.',
 '',
 '-',
 'W m$^{-2}$'),
('$r_{sw}$',
 'r_sw',
 'Stomatal resistance to water vapour, inverse of $g_{sw}$.',
 '',
```

(continues on next page)

(continued from previous page)

```
  '-',
  's m$^{-1}$'),
 ('$r_{tw}$',
  'r_tw',
  'Total leaf resistance to water vapour, $r_{bv} + r_{sv}$.',
  '',
  '-',
  's m$^{-1}$'),
 ('$T_l$', 'T_l', 'Leaf temperature.', '', '-', 'K'),
 ('$T_w$',
  'T_w',
  'Radiative temperature of objects surrounding the leaf.',
  '',
  '-',
  'K')]
```

Equations for leaf energy and water balance

General equations based on the above variables can be imported from `essm.equations.physics.thermodynamics`:

[10]:
```
import essm.equations.leaf.energy_water as leaf_energy
modstr = 'leaf_energy.'
eqs = [name for name in leaf_energy.__all__]
table = ListTable()
#table.append(('Name', 'Description', 'Equation'))
for name in eqs:
    table.append((name, eval(modstr+name).__doc__, latex('$'+latex(eval(modstr+name))+
↪'$')))
table
```

[10]:
```
[('eq_Rs_enbal',
  'Calculate R_s from energy balance.\n\n    (Eq. 1 in :cite:`schymanski_leaf-scale_
↪2017`)\n    ',
  '$R_s = E_l + H_l + R_{ll}$'),
 ('eq_Rll',
  'R_ll as function of T_l and T_w.\n\n    (Eq. 2 in :cite:`schymanski_leaf-scale_
↪2017`)\n    ',
  '$R_{ll} = a_{sh} \\epsilon_l \\sigma \\left(T_l^{4} - T_w^{4}\\right)$'),
 ('eq_Hl',
  'H_l as function of T_l.\n\n    (Eq. 3 in :cite:`schymanski_leaf-scale_2017`)\n
↪',
  '$H_l = a_{sh} h_c \\left(- T_a + T_l\\right)$'),
 ('eq_El',
  'E_l as function of E_lmol.\n\n    (Eq. 4 in :cite:`schymanski_leaf-scale_2017`)\n ␣
↪  ',
  '$E_l = E_{l,mol} M_w \\lambda_E$'),
 ('eq_Elmol',
  'E_lmol as functino of g_tw and C_wl.\n\n    (Eq. 5 in :cite:`schymanski_leaf-scale_
↪2017`)\n    ',
  '$E_{l,mol} = g_{tw} \\left(- C_{wa} + C_{wl}\\right)$'),
 ('eq_gtw',
  'g_tw from g_sw and g_bw.\n\n    (Eq. 6 in :cite:`schymanski_leaf-scale_2017`)\n
↪',
  '$g_{tw} = \\frac{1}{\\frac{1}{g_{sw}} + \\frac{1}{g_{bw}}}$'),
```

(continues on next page)

(continued from previous page)

```
 ('eq_gbw_hc',
  'g_bw as function of h_c.\n\n    (Eq. B2 in :cite:`schymanski_leaf-scale_2017`)\n ␣
↪ ',
  '$g_{bw} = \\frac{a_s h_c}{N_{Le}^{\\frac{2}{3}} c_{pa} \\rho_a}$'),
 ('eq_Cwl',
  'C_wl as function of P_wl and T_l.\n\n    (Eq. B4 in :cite:`schymanski_leaf-scale_
↪2017`)\n    ',
  '$C_{wl} = \\frac{P_{wl}}{R_{mol} T_l}$'),
 ('eq_Pwl',
  'Clausius-Clapeyron P_wl as function of T_l.\n\n    (Eq. B3 in :cite:`hartmann_
↪global_1994`)\n    ',
  '$P_{wl} = p_1 e^{- \\frac{M_w \\lambda_E \\left(- \\frac{1}{p_2} + \\frac{1}{T_l}
↪\\right)}{R_{mol}}}$'),
 ('eq_Elmol_conv',
  'E_lmol as function of g_twmol and P_wl.\n\n    (Eq. B6 in :cite:`schymanski_leaf-
↪scale_2017`)\n    ',
  '$E_{l,mol} = \\frac{g_{tw,mol} \\left(- P_{wa} + P_{wl}\\right)}{P_a}$'),
 ('eq_gtwmol_gtw',
  'g_twmol as a function of g_tw.\n\n    It uses eq_Elmol, eq_Cwl and eq_Elmol_conv.
↪\n    ',
  '$g_{tw,mol} = \\frac{g_{tw} \\left(P_a P_{wa} T_l - P_a P_{wl} T_a\\right)}{R_{mol}
↪ T_a T_l \\left(P_{wa} - P_{wl}\\right)}$'),
 ('eq_gtwmol_gtw_iso',
  'g_twmol as a function of g_tw at isothermal conditions.',
  '$g_{tw,mol} = \\frac{P_a g_{tw}}{R_{mol} T_a}$'),
 ('eq_hc',
  'h_c as a function of Nu and L_l.\n\n    (Eq. B10 in :cite:`schymanski_leaf-scale_
↪2017`)\n    ',
  '$h_c = \\frac{N_{Nu_L} k_a}{L_l}$'),
 ('eq_Re',
  'Re as a function of v_w and L_l.\n\n    (Eq. B11 in :cite:`schymanski_leaf-scale_
↪2017`)\n    ',
  '$N_{Re_L} = \\frac{L_l v_w}{\\nu_a}$'),
 ('eq_Gr',
  'Gr as function of air density within and outside of leaf.\n\n    (Eq. B12 in :cite:
↪`schymanski_leaf-scale_2017`)\n    ',
  '$N_{Gr_L} = \\frac{L_l^{3} g \\left(\\rho_a - \\rho_{al}\\right)}{\\nu_a^{2} \\rho_
↪{al}}$')]
```

Variables for leaf radiative balance

Variables related to the model by can be imported from `essm.variables.leaf.radiation`:

[11]:
```
import essm.variables.leaf.radiation as leaf_radiation
vars = ['leaf_radiation.' + name for name in leaf_radiation.__all__]
generate_metadata_table([eval(name) for name in vars])
```

```
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪leaf.energy_water:alpha_l" will be overridden by "essm.variables.leaf.radiation:
↪<class 'essm.variables.leaf.radiation.alpha_l'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:R_d" will be overridden by "essm.variables.leaf.radiation:
↪<class 'essm.variables.leaf.radiation.R_d'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪leaf.energy_water:R_la" will be overridden by "essm.variables.leaf.radiation:
↪'essm.variables.leaf.radiation.R_la'>"
```

(continues on next page)

(continued from previous page)

```
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪leaf.energy_water:R_ld" will be overridden by "essm.variables.leaf.radiation:<class
↪'essm.variables.leaf.radiation.R_ld'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪leaf.energy_water:R_lu" will be overridden by "essm.variables.leaf.radiation:<class
↪'essm.variables.leaf.radiation.R_lu'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:R_u" will be overridden by "essm.variables.leaf.radiation:
↪<class 'essm.variables.leaf.radiation.R_u'>"
  instance[expr] = instance
```

[11]:
```
[('Symbol', 'Name', 'Description', 'Definition', 'Default value', 'Units'),
 ('$alpha_l$',
  'alpha_l',
  'Leaf albedo, fraction of shortwave radiation reflected by the leaf.',
  '',
  '-',
  '1'),
 ('$R_d$',
  'R_d',
  'Downwelling global radiation.',
  '',
  '-',
  'J s$^{-1}$ m$^{-2}$'),
 ('$R_u$',
  'R_u',
  'Upwelling global radiation.',
  '',
  '-',
  'J s$^{-1}$ m$^{-2}$'),
 ('$R_{la}$',
  'R_la',
  'Longwave radiation absorbed by leaf.',
  '',
  '-',
  'J s$^{-1}$ m$^{-2}$'),
 ('$R_{ld}$',
  'R_ld',
  'Downwards emitted/reflected global radiation from leaf.',
  '',
  '-',
  'J s$^{-1}$ m$^{-2}$'),
 ('$R_{lu}$',
  'R_lu',
  'Upwards emitted/reflected global radiation from leaf.',
  '',
  '-',
  'J s$^{-1}$ m$^{-2}$'),
 ('$S_a$',
  'S_a',
  'Radiation sensor above leaf reading.',
  '',
  '-',
  'J s$^{-1}$ m$^{-2}$'),
```

(continues on next page)

(continued from previous page)

```
 ('$S_b$',
  'S_b',
  'Radiation sensor below leaf reading.',
  '',
  '-',
  'J s$^{-1}$ m$^{-2}$'),
 ('$S_s$',
  'S_s',
  'Radiation sensor beside leaf reading.',
  '',
  '-',
  'J s$^{-1}$ m$^{-2}$')]
```

2.1.3 Variables for leaf chamber model

These refer to the model by Schymanski & Or (2017) and ongoing work.

Leaf chamber insulation

Variables related to the model by can be imported from `essm.variables.chamber.insulation`:

[12]:
```
import essm.variables.chamber.insulation as chamber_ins
vars = ['chamber_ins.' + name for name in chamber_ins.__all__]
generate_metadata_table([eval(name) for name in vars])
```

[12]:
```
[('Symbol', 'Name', 'Description', 'Definition', 'Default value', 'Units'),
 ('$A_i$',
  'A_i',
  'Conducting area of insulation material.',
  '',
  '-',
  'm$^{2}$'),
 ('$c_{pi}$',
  'c_pi',
  'Heat capacity of insulation material.',
  '',
  '-',
  'J K$^{-1}$ kg$^{-1}$'),
 ('$dT_i$',
  'dT_i',
  'Temperature increment of insulation material.',
  '',
  '-',
  'K'),
 ('$L_i$', 'L_i', 'Thickness of insulation material.', '', '-', 'm'),
 ('$lambda_i$',
  'lambda_i',
  'Heat conductivity of insulation material.',
  '',
  '-',
  'J K$^{-1}$ m$^{-1}$ s$^{-1}$'),
 ('$Q_i$',
  'Q_i',
  'Heat conduction through insulation material.',
```

(continues on next page)

(continued from previous page)

```
  '',
  '-',
  'J s$^{-1}$'),
 ('$rho_i$',
  'rho_i',
  'Density of insulation material.',
  '',
  '-',
  'kg m$^{-3}$')]
```

Leaf chamber mass balance

Variables related to the model by can be imported from `essm.variables.chamber.mass`:

[13]:
```
import essm.variables.chamber.mass as chamber_mass
vars = ['chamber_mass.' + name for name in chamber_mass.__all__]
generate_metadata_table([eval(name) for name in vars])
```

```
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪leaf.energy_water:L_A" will be overridden by "essm.variables.chamber.mass:<class
↪'essm.variables.chamber.mass.L_A'>"
  instance[expr] = instance
```

[13]:
```
[('Symbol', 'Name', 'Description', 'Definition', 'Default value', 'Units'),
 ('$F_{in,mol,a}$',
  'F_in_mola',
  'Molar flow rate of dry air into chamber.',
  '',
  '-',
  'mol s$^{-1}$'),
 ('$F_{in,mol,w}$',
  'F_in_molw',
  'Molar flow rate of water vapour into chamber.',
  '',
  '-',
  'mol s$^{-1}$'),
 ('$F_{in,v}$',
  'F_in_v',
  'Volumetric flow rate into chamber.',
  '',
  '-',
  'm$^{3}$ s$^{-1}$'),
 ('$F_{out,mol,a}$',
  'F_out_mola',
  'Molar flow rate of dry air out of chamber.',
  '',
  '-',
  'mol s$^{-1}$'),
 ('$F_{out,mol,w}$',
  'F_out_molw',
  'Molar flow rate of water vapour out of chamber.',
  '',
  '-',
  'mol s$^{-1}$'),
 ('$F_{out,v}$',
  'F_out_v',
```

(continues on next page)

(continued from previous page)

```
 'Volumetric flow rate out of chamber.',
 '',
 '-',
 'm$^{3}$ s$^{-1}$'),
('$H_c$', 'H_c', 'Chamber height.', '', '-', 'm'),
('$L_A$', 'L_A', 'Leaf area.', '', '-', 'm$^{2}$'),
('$L_c$', 'L_c', 'Chamber length.', '', '-', 'm'),
('$n_c$', 'n_c', 'molar mass of gas in chamber.', '', '-', 'mol'),
('$P_{w,in}$', 'P_w_in', 'Vapour pressure of incoming air.', '', '-', 'Pa'),
('$P_{w,out}$', 'P_w_out', 'Vapour pressure of outgoing air.', '', '-', 'Pa'),
('$R_{H,in}$', 'R_H_in', 'Relative humidity of incoming air.', '', '-', '1'),
('$T_d$', 'T_d', 'Dew point temperature of incoming air.', '', '-', 'K'),
('$T_{in}$', 'T_in', 'Temperature of incoming air.', '', '-', 'K'),
('$T_{out}$',
 'T_out',
 'Temperature of outgoing air (= chamber T_a).',
 '',
 '-',
 'K'),
('$T_{room}$', 'T_room', 'Lab air temperature.', '', '-', 'K'),
('$V_c$', 'V_c', 'Chamber volume.', '', '-', 'm$^{3}$'),
('$W_c$', 'W_c', 'Chamber width.', '', '-', 'm')]
```

2.1.4 Bibliography

See https://essm.readthedocs.io/en/latest/api.html#bibliography

```
[ ]:
```

CHAPTER 3

Use examples

Here you find use examples for different API features of ESSM.

3.1 Use examples

This jupyter notebook can be found at: https://github.com/environmentalscience/essm/blob/master/docs/examples/api_features.ipynb

Below, we will import some generic python packages that are used in this notebook:

[1]:
```
# Checking for essm version installed
import pkg_resources
pkg_resources.get_distribution("essm").version
```

[1]:
```
'0.4.3.dev21+dirty'
```

[2]:
```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:120% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

[3]:
```
from IPython.display import display
from sympy import init_printing, latex
init_printing()
from sympy.printing import StrPrinter
StrPrinter._print_Quantity = lambda self, expr: str(expr.abbrev)    # displays short␣
↪units (m instead of meter)
```

[4]:
```
import scipy as sc
# Import various functions from sympy
from sympy import Derivative, Eq, exp, log, solve, Symbol
from essm.equations import Equation
from essm.variables import Variable
from essm.variables.utils import ListTable, generate_metadata_table
```

3.1.1 Importing variables and equations from essm

[5]:
```
from essm.variables.chamber import *
from essm.variables.leaf import *
from essm.variables.physics.thermodynamics import *
from essm.equations.leaf import *
from essm.equations.physics.thermodynamics import *
```

3.1.2 Plotting

Below, we will define a function to make plotting of equations very easy, and then show an example:

[6]:
```
import matplotlib.pyplot as plt
from sympy import latex
from sympy import N
from numpy import arange
from essm.variables.units import derive_unit, SI, Quantity
from essm.variables.utils import markdown

def plot_expr2(xvar_min_max, yldata, yllabel=None, yrdata=None,
               yrlabel='', clf=True, npoints=100, ylmin=None, ylmax=None,
               yrmin=None, yrmax=None, xlabel=None,
               colors=None,
               loc_legend_left='best', loc_legend_right='right',
               linestylesl=['-', '--', '-.', ':'],
               linestylesr=['-', '--', '-.', ':'],
               fontsize=None, fontsize_ticks=None, fontsize_labels=None,
               fontsize_legend=None,
               fig1=None, **args):
    '''
    Plot expressions as function of xvar from xmin to xmax.

    **Examples:**

    from essm.variables import Variable
    from essm.variables.physics.thermodynamics import T_a
    from essm.equations.physics.thermodynamics import eq_nua, eq_ka
    vdict = Variable.__defaults__.copy()
    expr = eq_nua.subs(vdict)
    exprr = eq_ka.subs(vdict)
    xvar = T_a
    yldata = [(expr.rhs, 'full'), (expr.rhs/2, 'half')]
    yrdata = exprr
    plot_expr2((T_a, 273, 373), yldata, yllabel = (nu_a), yrdata=yrdata)
    plot_expr2((T_a, 273, 373), yldata, yllabel = (nu_a),
               yrdata=[(1/exprr.lhs, 1/exprr.rhs)],
               loc_legend_right='lower right')
    plot_expr2((T_a, 273, 373), expr)
    plot_expr2((T_a, 273, 373), yldata, yllabel = (nu_a))
    '''
    (xvar, xmin, xmax) = xvar_min_max
    if not colors:
        if yrdata is not None:
            colors = ['black', 'blue', 'red', 'green']
        else:
            colors = ['blue', 'black', 'red', 'green']
```

(continues on next page)

(continued from previous page)

```
    if fontsize:
        fontsize_labels = fontsize
        fontsize_legend = fontsize
        fontsize_ticks = fontsize
    if not fig1:
        plt.close
        plt.clf
        fig = plt.figure(**args)
    else:
        fig = fig1
    if hasattr(xvar, 'definition'):
        unit1 = derive_unit(xvar)
        if unit1 != 1:
            strunit = ' (' + markdown(unit1) + ')'
        else:
            strunit = ''
        if not xlabel:
            xlabel = '$'+latex(xvar)+'$'+ strunit
    else:
        if not xlabel:
            xlabel = xvar
    if hasattr(yldata, 'lhs'):
        yldata = (yldata.rhs, yldata.lhs)
    if not yllabel:
        if type(yldata) is tuple:
            yllabel = yldata[1]
        else:
            try:
                yllabel = yldata[0][1]
            except Exception as e1:
                print(e1)
                print('yldata must be equation or list of (expr, name) tuples')

    if type(yllabel) is not str:
        unit1 = derive_unit(yllabel)
        if unit1 != 1:
            strunit = ' (' + markdown(unit1) + ')'
        else:
            strunit = ''

        yllabel = '$'+latex(yllabel)+'$'+ strunit
    if type (yldata) is not list and type(yldata) is not tuple:
        # If only an expression given
        yldata = [(yldata, '')]
    if type(yldata[0]) is not tuple:
        yldata = [yldata]
    if yrdata is not None:
        if yrlabel == '':
            if hasattr(yrdata, 'lhs'):
                yrlabel = yrdata.lhs
        if type (yrdata) is not list and type(yrdata) is not tuple:
            # If only an expression given
            yrdata = [yrdata]
    if type(yrlabel) is not str:
        yrlabel = '$'+latex(yrlabel)+'$'+ ' (' + markdown(derive_unit(yrlabel)) + ')'

    xstep = (xmax - xmin)/npoints
```

(continues on next page)

(continued from previous page)

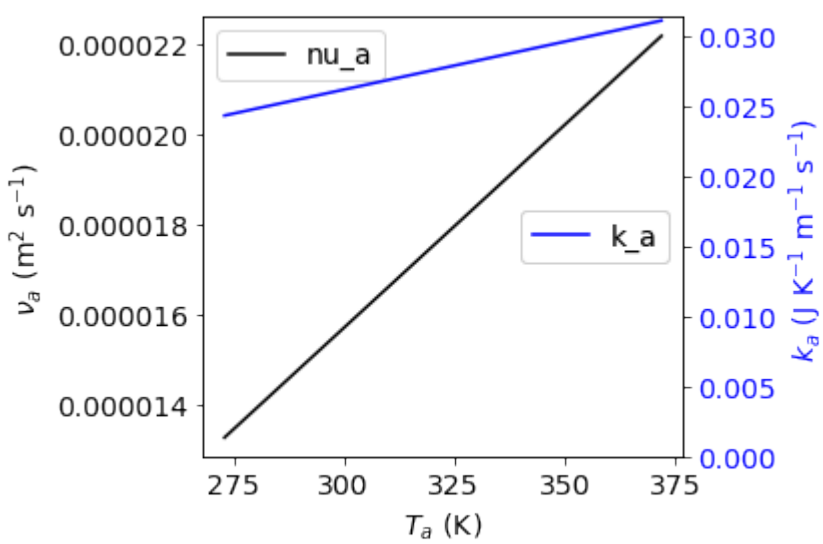
```
    xvals = arange(xmin, xmax, xstep)

    ax1 =  fig.add_subplot(1, 1, 1)
    if yrdata is not None:
        color = colors[0]
    else:
        color = 'black'
    if ylmin:    ax1.set_ylim(ymin=float(ylmin))
    if ylmax:    ax1.set_ylim(ymax=float(ylmax))
    ax1.set_xlabel(xlabel)
    ax1.set_ylabel(yllabel, color=color)
    ax1.tick_params(axis='y', labelcolor=color)
    i = 0
    for (expr1, y1var) in yldata:
        linestyle = linestylesl[i]
        if yrdata is None:
            color = colors[i]
        i= i + 1
        try:
            y1vals = [expr1.subs(xvar, dummy).n() for dummy in xvals]
            ax1.plot(xvals, y1vals, color=color, linestyle=linestyle, label=y1var)
        except Exception as e1:
            print([expr1.subs(xvar, dummy) for dummy in xvals])
            print(e1)
    if i > 1 or yrdata is not None:
        plt.legend(loc=loc_legend_left, fontsize=fontsize_legend)

    if yrdata is not None:
        ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis
        color = colors[1]
        ax2.set_ylabel(yrlabel, color=color)
        i = 0

        for item in yrdata:
            if type(item) is tuple:   # if item is tuple
                (expr2, y2var) = item
            else:
                try:
                    (y2var, expr2) = (item.lhs, item.rhs)
                except Exception as e1:
                    print(e1)
                    print('yrdata must be a list of equations or tuples (var, expr)')
                    return
            linestyle = linestylesr[i]
            i = i + 1
            try:
                y2vals = [expr2.subs(xvar, dummy).n() for dummy in xvals]
                ax2.plot(xvals, y2vals, color=color, linestyle=linestyle, label=y2var)
            except Exception as e1:
                print(expr2)
                print([expr2.subs(xvar, dummy).n() for dummy in xvals])
                print(e1)

            if not yrlabel:
                if hasattr(yrdata[0], 'lhs'):
                    yrlabel = yrdata[0].lhs
```

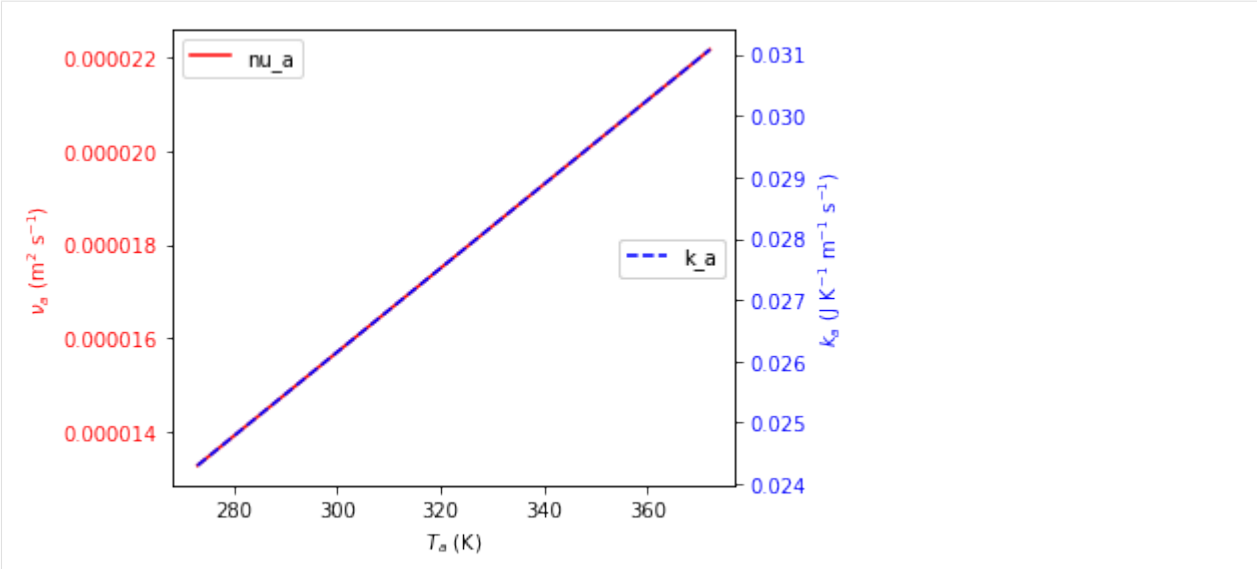
(continues on next page)

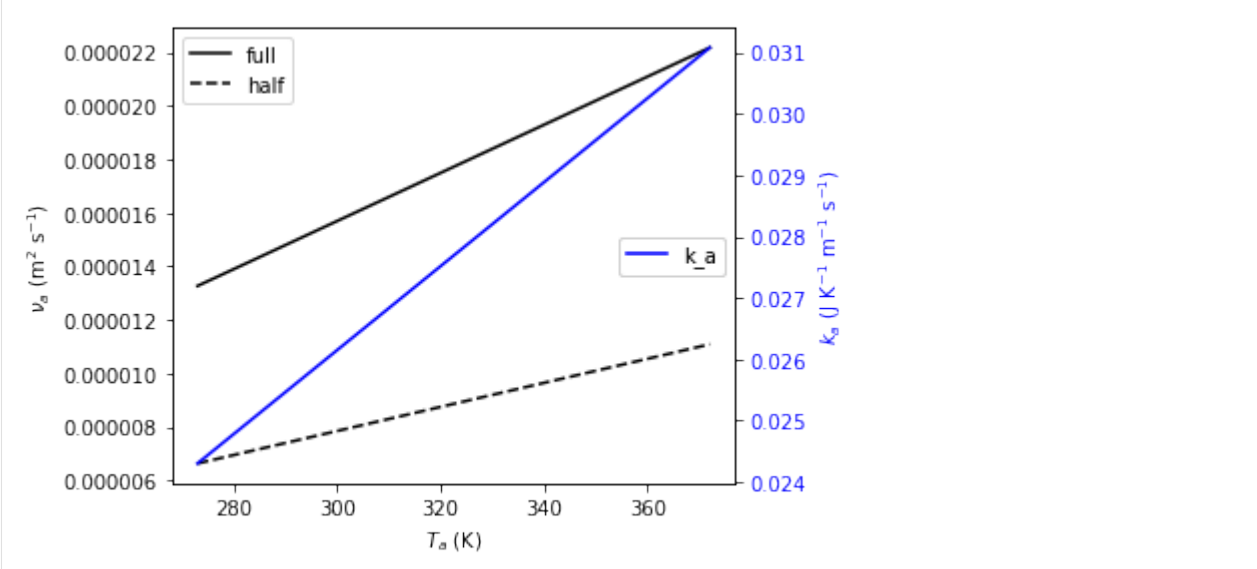
(continued from previous page)

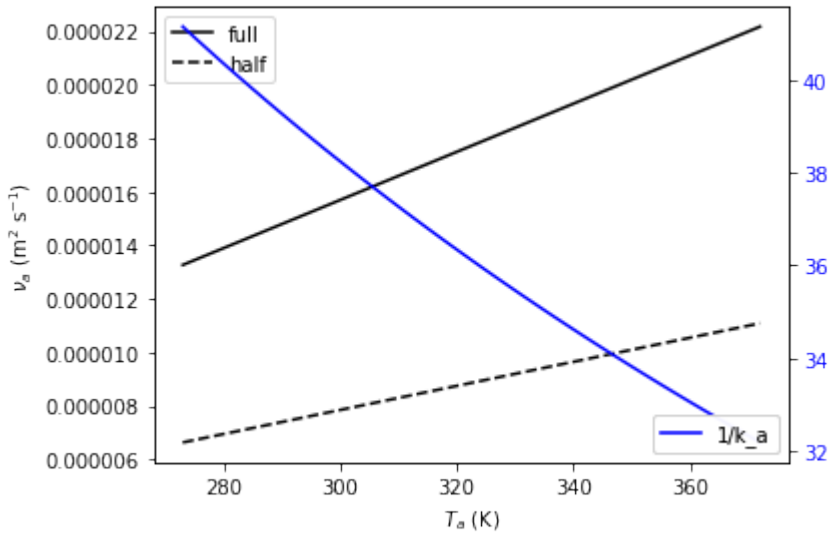
```
        if type(yrlabel) is not str:
            yrlabel = '$'+latex(yrlabel)+'$'+ ' (' + markdown(derive_unit(yrlabel)) +
↪')'
        ax2.tick_params(axis='y', labelcolor=color)
        if yrmin:    ax2.set_ylim(ymin=float(yrmin))
        if yrmax:    ax2.set_ylim(ymax=float(yrmax))
        leg=ax2.legend(loc=loc_legend_right, fontsize=fontsize_legend)
        ax2.add_artist(leg);
        for item in ([ax2.xaxis.label, ax2.yaxis.label]):
            item.set_fontsize(fontsize_labels)
        ax2.tick_params(axis='both', which='major', labelsize=fontsize_ticks)

    for item in ([ax1.xaxis.label, ax1.yaxis.label]):
        item.set_fontsize(fontsize_labels)
    ax1.tick_params(axis='both', which='major', labelsize=fontsize_ticks)
    fig.tight_layout()  # otherwise the right y-label is slightly clipped
    return fig
```

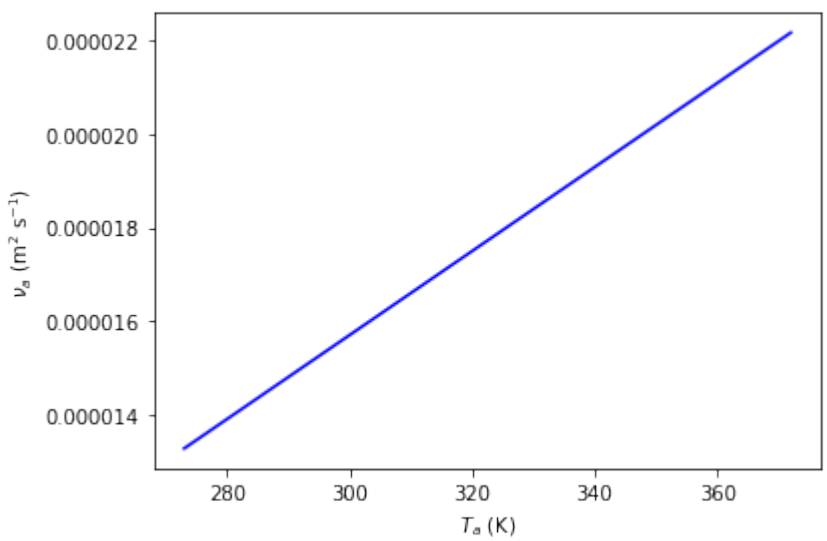
[7]:
```
vdict = Variable.__defaults__.copy()
expr = eq_nua.subs(vdict)
exprr = eq_ka.subs(vdict)
xvar = T_a
yldata = [(expr.rhs, 'full'), (expr.rhs/2, 'half')]
yrdata = exprr
fig=plot_expr2((T_a, 273, 373), yldata=expr, yrdata=exprr, yrmin=-0.0001,␣
↪fontsize=14) # note that yrmin=0 would have no effect
fig=plot_expr2((T_a, 273, 373), yldata=expr, yrdata=exprr, colors=['red', 'blue'],␣
↪linestylesr=['--'])
fig=plot_expr2((T_a, 273, 373), yldata, yllabel = (nu_a), yrdata=yrdata)
fig=plot_expr2((T_a, 273, 373), yldata, yllabel = (nu_a), yrdata=[(1/exprr.rhs, 1/
↪exprr.lhs)],
           loc_legend_right='lower right')
fig=plot_expr2((T_a, 273, 373), expr)
fig=plot_expr2((T_a, 273, 373), yldata, yllabel = (nu_a))
```

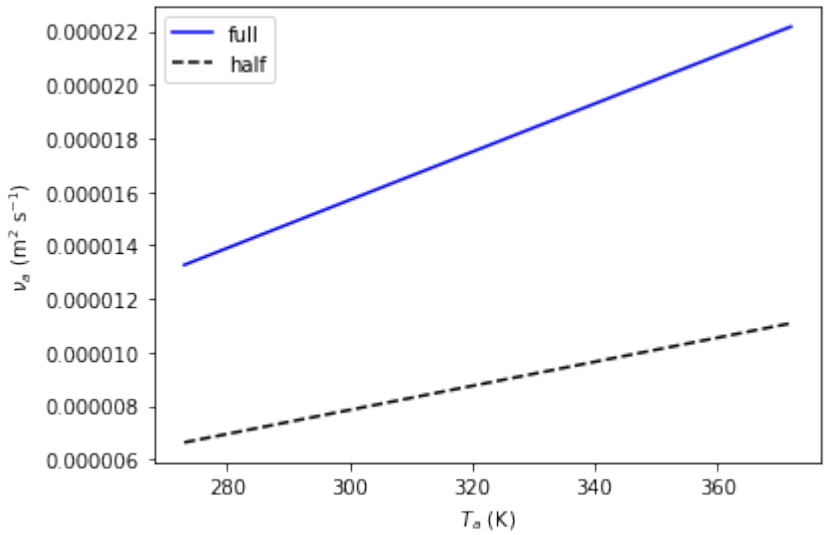



nu_a
k_a
0.000022
0.000020
0.000018
0.000016
0.000014
ν_a (m^2 s^{-1})
0.031
0.030
0.029
0.028
0.027
0.026
0.025
0.024
k_a (J K^{-1} m^{-1} s^{-1})
280
300
320
340
360
T_a (K)


full
half
k_a
0.000022
0.000020
0.000018
0.000016
0.000014
0.000012
0.000010
0.000008
0.000006
ν_a (m^2 s^{-1})
0.031
0.030
0.029
0.028
0.027
0.026
0.025
0.024
k_a (J K^{-1} m^{-1} s^{-1})
280
300
320
340
360
T_a (K)


full
half
1/k_a
0.000022
0.000020
0.000018
0.000016
0.000014
0.000012
0.000010
0.000008
0.000006
ν_a (m^2 s^{-1})
40
38
36
34
32
280
300
320
340
360
T_a (K)

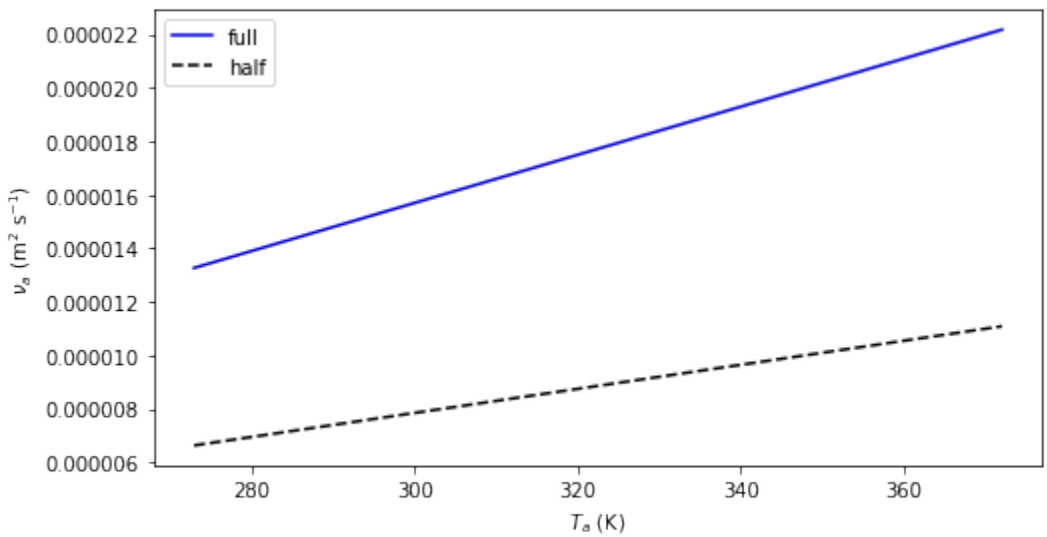

0.000022
0.000020
0.000018
0.000016
0.000014
ν_a (m^2 s^{-1})
280
300
320
340
360
T_a (K)



We can also manipulate the figure later, e.g. change the width:

[8]:
```
%matplotlib inline
fig.set_figwidth(8)
fig
```

[8]:



3.1.3 Creating new variables

To create custom variables, first import `Variable`:

[9]:
```
from essm.variables import Variable
```

To define units, you can either import these units from the library, e.g.

`from essm.variables.units import joule, kelvin, meter`

or import the appropriate units from sympy, e.g.

```
from sympy.physics.units import joule, kelvin, meter
```

[10]:
```
from sympy.physics.units import joule, kelvin, meter, mole, pascal, second
```

Then you can define a custom variable with its name, description, domain, latex_name, unit, and an optional default value, e.g.:

[11]:
```
class R_mol(Variable):
    """Molar gas constant."""
    unit = joule/(kelvin*mole)
    latex_name = 'R_{mol}'
    default = 8.314472
```

```
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:R_mol" will be overridden by "__main__:<class '__main__.R_mol
↪'>"
  instance[expr] = instance
```

The variables defined above hold information about their docstring, units, latex representations and default values if any. Each can be accessed by e.g.:

[12]:
```
print(R_mol.__doc__)
print(R_mol.definition.unit)
print(R_mol.definition.latex_name)
print(R_mol.definition.default)
```

```
Molar gas constant.
J/(K*mol)
R_{mol}
8.314472
```

We will now define a few additional variables.

[13]:
```
class P_g(Variable):
    """Pressure of gas."""
    unit = pascal

class V_g(Variable):
    """Volume of gas."""
    unit = meter**3

class n_g(Variable):
    """Amount of gas."""
    unit = mole

class n_w(Variable):
    """Amount of water."""
    unit = mole

class T_g(Variable):
    """Temperature of gas."""
    unit = kelvin

class P_wa(Variable):
    """Partial pressure of water vapour in air."""
    unit = pascal
    latex_name = 'P_{wa}'
```

```
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:P_wa" will be overridden by "__main__:<class '__main__.P_wa'>
↪"
  instance[expr] = instance
```

Variables with expressions as definitions

[14]:
```
class Delta_Pwa(Variable):
    """Slope of saturated vapour pressure, $\partial P_{wa} / \partial T_g$"""
    expr = Derivative(P_wa,T_g)
    latex_name = r'\Delta'
```

[15]:
```
Delta_Pwa.definition.unit
```

[15]:

$$\frac{Pa}{K}$$

[16]:
```
Delta_Pwa.definition.expr
```

[16]:

$$\frac{d}{dT_g} P_{wa}$$

[17]:
```
generate_metadata_table([Delta_Pwa])
```

[17]:
```
[('Symbol', 'Name', 'Description', 'Definition', 'Default value', 'Units'),
 ('$\\Delta$',
  'Delta_Pwa',
  'Slope of saturated vapour pressure, $\\partial P_{wa} / \\partial T_g$',
  '$\\frac{d}{d T_g} P_{wa}$',
  '-',
  'Pa K$^{-1}$')]
```

Linking assumptions to variables

We can specify if a given variable is a complex, real, integer etc. by using the `assumptions` property during variable definition:

[18]:
```
class x(Variable):
     """Positive real variable."""
     assumptions = {'positive': True, 'real': True}

print(solve(x**2 - 1))
```

```
[1]
```

3.1.4 Creating new equations

Equations have a left hand side and a right hand side and if they contain variables with units, the units of each addend must be the same.

Custom equation

To create custom equations, first import `Equation`:

[19]:
```
from essm.equations import Equation
```

We will now define an equation representing the ideal gas law, based on the variables defined above:

[20]:
```
class eq_ideal_gas_law(Equation):
    """Ideal gas law."""

    expr = Eq(P_g*V_g, n_g*R_mol*T_g)
```

Note that whenever an equation is defined, its units are checked for consistency in the background and if they are not consistent, an error message will be printed. To illustrate this, we will try to define the above equation again, but omit temperature on the right hand side:

[21]:
```
try:
    class eq_ideal_gas_law(Equation):
        """Ideal gas law."""

        expr = Eq(P_g*V_g, n_g*R_mol)
except Exception as exc1:
    print(exc1)
```

```
Dimension of "R_mol*n_g" is Dimension(length**2*mass/(temperature*time**2)), but it␣
↪should be the same as P_g*V_g, i.e. Dimension(length**2*mass/time**2)
```

The equation can be displayed in typesetted form, and the documentation string can be accessed in a similar way as for Variable:

[22]:
```
display(eq_ideal_gas_law)
print(eq_ideal_gas_law.__doc__)
```

$$P_g V_g = R_{mol} T_g n_g$$

```
Ideal gas law.
```

New equation based on manipulation of previous equations

We can use the above equation just as any Sympy expression, and e.g. solve it for pressure:

[23]:
```
soln = solve(eq_ideal_gas_law, P_g, dict=True); print(soln)
```

```
[{P_g: R_mol*T_g*n_g/V_g}]
```

If we want to define a new equation based on a manipulation of eq_ideal_gas_law we can specify that the parent of the new equation is `eq_ideal_gas_law.definition`:

[24]:
```
class eq_Pg(eq_ideal_gas_law.definition):
    """Calculate pressure of ideal gas."""

    expr = Eq(P_g, soln[0][P_g])
eq_Pg
```

[24]:

$$P_g = \frac{R_{mol} T_g n_g}{V_g}$$

We can also have nested inheritance, if we now define another equation based on eq_Pg:

[25]:
```
class eq_Pwa_nw(eq_Pg.definition):
    """Calculate vapour pressure from amount of water in gas."""

    expr = Eq(P_wa, eq_Pg.rhs.subs(n_g, n_w))
eq_Pwa_nw
```

[25]:

$$P_{wa} = \frac{R_{mol} T_g n_w}{V_g}$$

Show inheritance of equations

To see the inheritance (what other equations it depends on) of the newly created equation:

[26]:
```
eq_Pwa_nw.definition.__bases__
```

[26]:
```
(__main__.eq_Pg,)
```

[27]:
```
[parent.name for parent in eq_Pwa_nw.definition.__bases__]
```

[27]:
```
['eq_Pg']
```

[28]:
```
[parent.expr for parent in eq_Pwa_nw.definition.__bases__]
```

[28]:

$$\left[P_g = \frac{R_{mol} T_g n_g}{V_g}\right]$$

We can also use a pre-defined function to extract the set of parents:

[29]:
```
from essm.variables.utils import get_parents
get_parents(eq_Pwa_nw)
```

[29]:
```
{'eq_Pg'}
```

We can use another function to get all parents recursively:

[30]:
```
from essm.variables.utils import get_allparents
get_allparents(eq_Pwa_nw)
```

[30]:
```
{'eq_Pg', 'eq_ideal_gas_law'}
```

Computational burden of deriving equations within class definition

If we solve for a variable to derive a new equation, is the solve() command performed every time this equation is used?

[31]:
```
class eq_Pg1(eq_ideal_gas_law.definition):
    """Calculate pressure of ideal gas."""
    from sympy import solve
    soln = solve(eq_ideal_gas_law, P_g, dict=True); print(soln)
    expr = Eq(P_g, soln[0][P_g])
eq_Pg1
```

```
[{P_g: R_mol*T_g*n_g/V_g}]
```

```
/home/stan/Programs/essm/essm/equations/_core.py:107: UserWarning: "__main__:eq_Pg"␣
↪will be overridden by "__main__:<class '__main__.eq_Pg1'>"
  instance[expr] = instance
```

[31]:

$$P_g = \frac{R_{mol} T_g n_g}{V_g}$$

[32]:
```
%time
eq_Pg.subs({R_mol: 8.314, T_g: 300, n_g: 0.1, V_g: 1})
```

```
CPU times: user 3 µs, sys: 2 µs, total: 5 µs
Wall time: 8.34 µs
```

[32]:

$$P_g = 249.42$$

[33]:
```
%time
eq_Pg1.subs({R_mol: 8.314, T_g: 300, n_g: 0.1, V_g: 1})
```

```
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 7.87 µs
```

[33]:

$$P_g = 249.42$$

There is actually no difference!

Empirical equations with internal variables

Empirical equations not only contain variables but also numbers. As an example, we will try to define the Clausius-Clapeyron equation for saturation vapour pressure in the following example, after defining a few additional variables used in this equation.

$$P_{wa} = 611 e^{\frac{-M_w \lambda_E (1/T_g - 1/273)}{R_{mol}}}$$

[34]:
```
from sympy.physics.units import joule, kilogram
class lambda_E(Variable):
    """Latent heat of evaporation."""
    unit = joule/kilogram
    latex_name = '\\lambda_E'
    default = 2.45e6

class M_w(Variable):
    """Molar mass of water."""
    unit = kilogram/mole
    default = 0.018
```

```
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:lambda_E" will be overridden by "__main__:<class '__main__.
↪lambda_E'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:M_w" will be overridden by "__main__:<class '__main__.M_w'>"
  instance[expr] = instance
```

[35]:
```
from sympy import exp
try:
    class eq_Pwa_CC(Equation):
        """Clausius-Clapeyron P_wa as function of T_g.

        \cite[Eq. B3]{hartmann_global_1994}
        """

        expr = Eq(P_wa, 611.*exp(-M_w*lambda_E*(1/T_g - 1/273.)/R_mol))
except Exception as exc1:
    print(exc1)
```

```
Dimension of "1/T_g" is Dimension(1/temperature), but it should be the same as -0.
↪00366300366300366, i.e. Dimension(1)
```

The unit mismatch reported in the error message stems from the fact that the numbers in the empirical equation actually need units. Since the term in the exponent has to be non-dimensional, the units of `611` must be the same as those of `P_wa`, i.e. pascal. The units of the subtraction term in the exponent must match, meaning that `273` needs units of kelvin. To avoid the error message, we can define the empirical numbers as internal variables to the equation we want to define:

[36]:
```
class eq_Pwa_CC(Equation):
    """Clausius-Clapeyron P_wa as function of T_g.

    Eq. B3 in :cite{hartmann_global_1994}
    """

    class p_CC1(Variable):
        """Internal parameter of eq_Pwl."""
        unit = pascal
        latex_name = '611'
        default = 611.



    class p_CC2(Variable):
        """Internal parameter of eq_Pwl."""
        unit = kelvin
        latex_name = '273'
        default = 273.

    expr = Eq(P_wa, p_CC1*exp(-M_w*lambda_E*(1/T_g - 1/p_CC2)/R_mol))
```

In the above, we defined the latex representation of the empirical constants as their actual values, so the equation displays in the familiar way:

[37]:
```
eq_Pwa_CC
```

[37]:

$$P_{wa} = 611e^{-\frac{M_w \lambda_E \left(-\frac{1}{273} + \frac{1}{T_g}\right)}{R_{mol}}}$$

All default values of variables defined along with the variable definitions are stored in a dictionary that can be accessed as `Variable.__defaults__`. We can substitute the values from this dictionary into our empirical equation to plot saturation vapour pressure as a function of temperature:

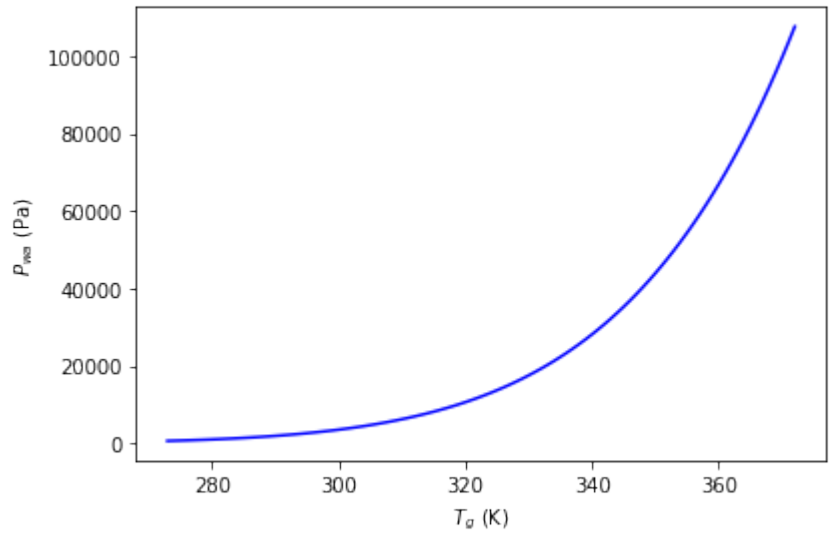
[38]:
```
expr = eq_Pwa_CC.subs(Variable.__defaults__)
print(expr)
```

(continues on next page)

(continued from previous page)

```
xvar = T_g
p = plot_expr2((xvar, 273, 373), expr)
```

```
Eq(P_wa, 167405731976.232*exp(-5304.00487246815/T_g))
```



Piecewise defined equations

[39]:
```
from sympy import Piecewise
expr = Eq(P_wa, Piecewise((0, T_a < 0), (eq_Pwa_CC.rhs, T_a >= 0)))
expr
```

[39]:

$$P_{wa} = \begin{cases} 0 & \text{for } T_a < 0 \\ 611e^{-\frac{M_w \lambda_E \left(-\frac{1}{273} + \frac{1}{T_g}\right)}{R_{mol}}} & \text{otherwise} \end{cases}$$

[40]:
```
try:
    class eq1(Equation):
        """Test"""
        expr = Eq(P_wa, Piecewise((0, T_a < 0), (eq_Pwa_CC.rhs, T_a >= 0)))
    display(eq1)
except Exception as e1:
    print(e1)
```

$$P_{wa} = \begin{cases} 0 & \text{for } T_a < 0 \\ 611e^{-\frac{M_w \lambda_E \left(-\frac{1}{273} + \frac{1}{T_g}\right)}{R_{mol}}} & \text{otherwise} \end{cases}$$

If the above returns a dimension error, then unit checking for ‘‘Piecewise‘‘ has not been implemented yet.

3.1.5 Substituting into integrals and derivatives and evaluating

Above, we defined `Delta_Pwa` as a variable that represents the partial derivative of `P_wa` with respect to `T_g`:

```
class Delta_Pwa(Variable):
    """Slope of saturated vapour pressure, $\partial P_{ws} / \partial T_g"""
    expr = P_wa(T_g).diff(T_g)
    #unit = pascal/kelvin
    latex_name = r'\Delta'
```

This definition can be accessed by typing `Delta_Pwa.definition.expr`. Example:

[41]:
```
print(Delta_Pwa.definition.expr)
display(Eq(Delta_Pwa, Delta_Pwa.definition.expr))
```

```
Derivative(P_wa, T_g)
```

$$\Delta = \frac{d}{dT_g} P_{wa}$$

We also defined the Clausius-Clapeyron approximation to $P_{wa}(T_g)$ as `eq_Pwa_CC`.

[42]:
```
display(eq_Pwa_CC)
print(eq_Pwa_CC.__doc__)
```

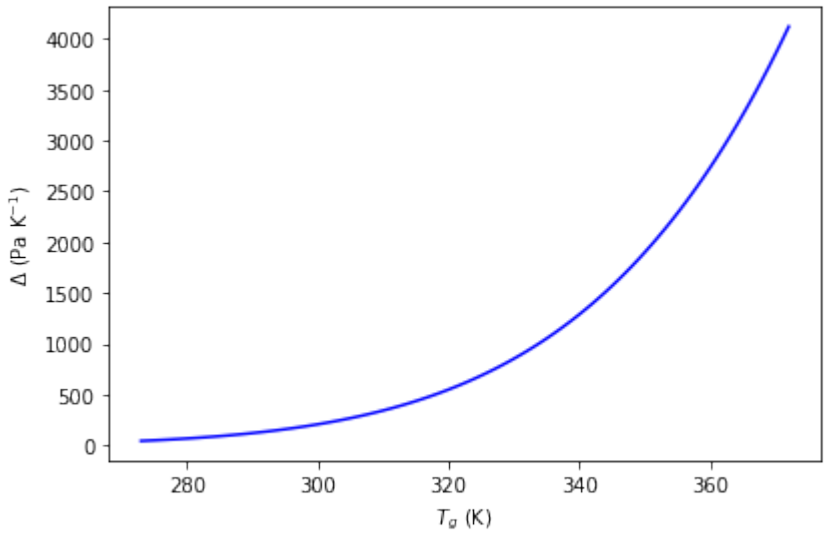
$$P_{wa} = 611 e^{-\frac{M_w \lambda_E \left(-\frac{1}{273} + \frac{1}{T_g}\right)}{R_{mol}}}$$

```
Clausius-Clapeyron P_wa as function of T_g.

    Eq. B3 in :cite{hartmann_global_1994}
```

If we want to substitute this approximation into `Delta_Pwa.definition.expr`, we need to use `replace` instead of `subs` and evaluate the derivative using `doit()`:

[43]:
```
expr = Eq(Delta_Pwa, Delta_Pwa.definition.expr.replace(P_wa, eq_Pwa_CC.rhs).doit())
display(expr)
p = plot_expr2((T_g, 273, 373), expr.subs(Variable.__defaults__))
```

$$\Delta = \frac{M_w \lambda_E 611 e^{-\frac{M_w \lambda_E \left(-\frac{1}{273} + \frac{1}{T_g}\right)}{R_{mol}}}}{R_{mol} T_g^2}$$



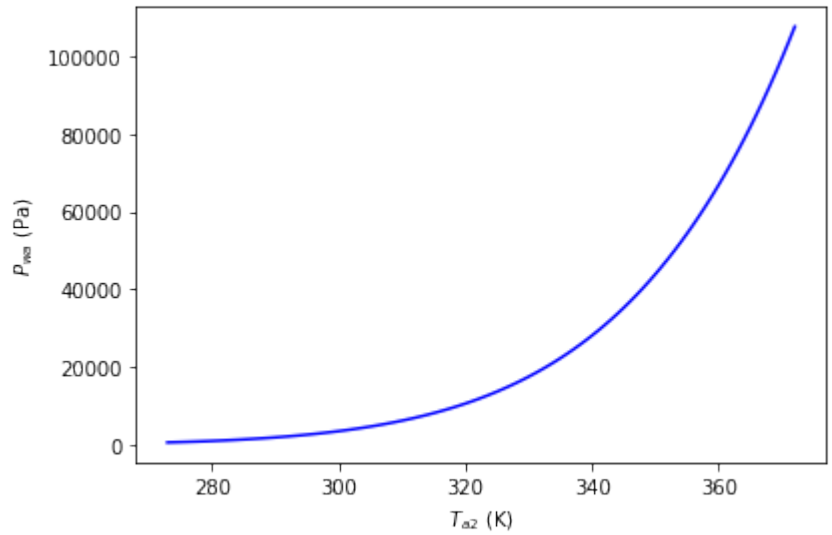
If we only had the slope of the curve, we could take the integral to get the absolute value:

[44]:
```
from sympy import Integral
class T_a1(Variable):
    """Air temperature"""
    unit = kelvin
    latex_name = r'T_{a1}'

class T_a2(Variable):
    """Air temperature"""
    unit = kelvin
    latex_name = r'T_{a2}'

class P_wa1(Variable):
    """P_wa at T1"""
    unit = pascal
    latex_name = r'P_{wa1}'

class eq_Pwa_Delta(Equation):
    """P_wa deduced from the integral of Delta"""
    expr = Eq(P_wa, P_wa1 + Integral(Delta_Pwa, (T_g, T_a1, T_a2)))
display(eq_Pwa_Delta)
```

$$P_{wa} = P_{wa1} + \int_{T_{a1}}^{T_{a2}} \Delta \, dT_g$$

[45]:
```
expr_Delta = eq_Pwa_CC.rhs.diff(T_g)
expr = Eq(P_wa, eq_Pwa_Delta.rhs.replace(Delta_Pwa, expr_Delta).doit())
vdict = Variable.__defaults__.copy()
vdict[T_a1] = 273.
vdict[P_wa1] = eq_Pwa_CC.rhs.subs(T_g, T_a1).subs(vdict)
display(expr.subs(vdict))
p = plot_expr2((T_a2, 273, 373), expr.subs(vdict))
```

$$P_{wa} = 167405731976.232e^{-\frac{5304.00487246815}{T_{a2}}}$$



3.1.6 Unit conversions

Values for variables are often given in obscure units, but to convert to our standard units, we can use the dictionary `SI_EXTENDED_DIMENSIONS`:

[46]:
```
from sympy.physics.units import convert_to, kilo, mega, joule, kilogram, meter,␣
↪second, inch, hour
from essm.variables.units import SI_EXTENDED_DIMENSIONS, SI_EXTENDED_UNITS
value1 = 0.3
unit1 = inch/hour
print(value1*unit1)
unit2 = Variable.get_dimensional_expr(unit1).subs(SI_EXTENDED_DIMENSIONS)
print(convert_to(value1*unit1, unit2))
```

```
0.3*inch/hour
2.11666666666667e-6*m/s
```

3.1.7 Exporting definitions

The below example creates a file called `test_variable-definitions.py` with all the variable definitions and relevant imports used in this notebook. Then, we re-import all the variables from the newly created file and create another file called `test_equation-definitions.py` with all the equation definitions and relevant imports. To re-use the equations in another notebook, just execute `from test_equation-definitions.py import *`

[47]:
```
from essm._generator import EquationWriter, VariableWriter
```

[48]:
```
StrPrinter._print_Quantity = lambda self, expr: str(expr.name)    # displays long␣
↪units (meter instead of m)
writer = VariableWriter(docstring='Variables defined in api_features.ipynb and␣
↪dependencies.')
for variable in Variable.__registry__.keys():
    writer.var(variable)
writer.write('test_variable_definitions.py')
StrPrinter._print_Quantity = lambda self, expr: str(expr.abbrev)    # displays short␣
↪units (m instead of meter)
```

(continues on next page)

(continued from previous page)

[49]:
```
from test_variable_definitions import *
```

```
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:alpha_a" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.alpha_a'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:c_pa" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.c_pa'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:c_pamol" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.c_pamol'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:c_pv" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.c_pv'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:C_wa" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.C_wa'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:D_va" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.D_va'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:g" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.g'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:Gr" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.Gr'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:h_c" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.h_c'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:k_a" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.k_a'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:lambda_E"␣
↪will be overridden by "test_variable_definitions:<class 'test_variable_definitions.
↪lambda_E'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:Le" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.Le'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:M_air" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.M_air'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:M_N2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.M_N2'>"
```

(continues on next page)

(continued from previous page)

```
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:M_O2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.M_O2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:M_w" will␣
↪be overridden by "test_variable_definitions:<class 'test_variable_definitions.M_w'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:nu_a" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.nu_a'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:Nu" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.Nu'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:P_a" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.P_a'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:Pr" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.Pr'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:P_N2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.P_N2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:P_O2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.P_O2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:P_wa"␣
↪will be overridden by "test_variable_definitions:<class 'test_variable_definitions.
↪P_wa'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:P_was" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.P_was'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:R_d" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.R_d'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:Re_c" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.Re_c'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:Re" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.Re'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:rho_a" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.rho_a'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:R_u" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.R_u'>"
```

(continues on next page)

(continued from previous page)

```
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:R_mol"␣
↪will be overridden by "test_variable_definitions:<class 'test_variable_definitions.
↪R_mol'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:R_s" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.R_s'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:sigm" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.sigm'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:T0" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.T0'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:T_a" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.T_a'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:v_w" will be overridden by "test_variable_definitions:<class
↪'test_variable_definitions.v_w'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:x_N2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.x_N2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.variables.
↪physics.thermodynamics:x_O2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.x_O2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.equations.
↪physics.thermodynamics:p_Dva1" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.p_Dva1'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.equations.
↪physics.thermodynamics:p_Dva2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.p_Dva2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.equations.
↪physics.thermodynamics:p_alpha1" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.p_alpha1'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.equations.
↪physics.thermodynamics:p_alpha2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.p_alpha2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.equations.
↪physics.thermodynamics:p_ka1" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.p_ka1'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.equations.
↪physics.thermodynamics:p_ka2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.p_ka2'>"
  instance[expr] = instance
```

(continues on next page)

(continued from previous page)

```
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.equations.
↪physics.thermodynamics:p_nua1" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.p_nua1'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "essm.equations.
↪physics.thermodynamics:p_nua2" will be overridden by "test_variable_definitions:
↪<class 'test_variable_definitions.p_nua2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:P_g" will␣
↪be overridden by "test_variable_definitions:<class 'test_variable_definitions.P_g'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:V_g" will␣
↪be overridden by "test_variable_definitions:<class 'test_variable_definitions.V_g'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:n_g" will␣
↪be overridden by "test_variable_definitions:<class 'test_variable_definitions.n_g'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:n_w" will␣
↪be overridden by "test_variable_definitions:<class 'test_variable_definitions.n_w'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:T_g" will␣
↪be overridden by "test_variable_definitions:<class 'test_variable_definitions.T_g'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:Delta_Pwa
↪" will be overridden by "test_variable_definitions:<class 'test_variable_
↪definitions.Delta_Pwa'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:x" will␣
↪be overridden by "test_variable_definitions:<class 'test_variable_definitions.x'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:p_CC1"␣
↪will be overridden by "test_variable_definitions:<class 'test_variable_definitions.
↪p_CC1'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:p_CC2"␣
↪will be overridden by "test_variable_definitions:<class 'test_variable_definitions.
↪p_CC2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:T_a1"␣
↪will be overridden by "test_variable_definitions:<class 'test_variable_definitions.
↪T_a1'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:T_a2"␣
↪will be overridden by "test_variable_definitions:<class 'test_variable_definitions.
↪T_a2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "__main__:P_wa1"␣
↪will be overridden by "test_variable_definitions:<class 'test_variable_definitions.
↪P_wa1'>"
  instance[expr] = instance
```

Since we re-imported names that already had definitions associated with them, we got a warning for each of them before it was overwritten. This tells us that the same variable used before may now have a different meaning, which could introduce inconsistency in the notebook. Here, however, we know that they are all the same. Now that we have re-imported all variables, we can use the `EquationWriter` to generate another python file with all the equations and variables they depend on, as shown below. The re-import was necessary so that the import statements do not point to `__main__` for locally defined variables.

[50]:
```
StrPrinter._print_Quantity = lambda self, expr: str(expr.name)    # displays long␣
↪units (meter instead of m)
writer = EquationWriter(docstring='Equations defined in api_features.ipynb and␣
↪dependencies.')
eqs_with_deps = []
for eq in Equation.__registry__.keys():
    parents = tuple(get_parents(eq))
    if parents == ():
        writer.eq(eq)
    else:
        eqs_with_deps.append(eq) # Equations with dependencies must be at the end
for eq in eqs_with_deps:
    writer.eq(eq)
writer.write('test_equation_definitions.py')

StrPrinter._print_Quantity = lambda self, expr: str(expr.abbrev)    # displays short␣
↪units (m instead of meter)
```

These definitions are re-imported in `examples_numerics.ipynb`, where you can also find examples for numerical computations using essm equations and variables.

3.2 Use examples for numerical calculations

This jupyter notebook can be found at: https://github.com/environmentalscience/essm/blob/master/docs/examples/examples_numerics.ipynb

Below, we will import variable and equation defintions that were previously exported from api_features.ipynb by running the file `test_equation_definitions.py`:

[1]:
```
from IPython.display import display
from sympy import init_printing, latex
init_printing()
from sympy.printing import StrPrinter
StrPrinter._print_Quantity = lambda self, expr: str(expr.abbrev)    # displays short␣
↪units (m instead of meter)
```

[2]:
```
%run -i 'test_equation_definitions.py'
```

```
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_Dva1" will be overridden by "__main__:<class '__main__.eq_Dva.p_Dva1'>
↪"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_Dva2" will be overridden by "__main__:<class '__main__.eq_Dva.p_Dva2'>
↪"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_alpha2" will be overridden by "__main__:<class '__main__.eq_alphaa.p_
↪alpha2'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_alpha1" will be overridden by "__main__:<class '__main__.eq_alphaa.p_
↪alpha1'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_ka2" will be overridden by "__main__:<class '__main__.eq_ka.p_ka2'>"
```

(continues on next page)

(continued from previous page)

```
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_ka1" will be overridden by "__main__:<class '__main__.eq_ka.p_ka1'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_nua1" will be overridden by "__main__:<class '__main__.eq_nua.p_nua1'>
↪"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_nua2" will be overridden by "__main__:<class '__main__.eq_nua.p_nua2'>
↪"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_CC2" will be overridden by "__main__:<class '__main__.eq_Pwa_CC.p_CC2
↪'>"
  instance[expr] = instance
/home/stan/Programs/essm/essm/variables/_core.py:89: UserWarning: "test_variable_
↪definitions:p_CC1" will be overridden by "__main__:<class '__main__.eq_Pwa_CC.p_CC1
↪'>"
  instance[expr] = instance
```

3.2.1 Numerical evaluations

See here for detailed instructions on how to turn sympy expressions into code: https://docs.sympy.org/latest/modules/codegen.html

We will first list all equations defined in this worksheet:

[3]:
```
for eq in Equation.__registry__.keys():
        print(eq.definition.name + ': ' + str(eq))
```

```
eq_Le: Eq(Le, alpha_a/D_va)
eq_Cwa: Eq(C_wa, P_wa/(R_mol*T_a))
eq_Nu_forced_all: Eq(Nu, -Pr**(1/3)*(-37*Re**(4/5) + 37*(Re + Re_c - Abs(Re - Re_c)/
↪2)**(4/5) - 664*sqrt(Re + Re_c - Abs(Re - Re_c)/2))/1000)
eq_Dva: Eq(D_va, T_a*p_Dva1 - p_Dva2)
eq_alphaa: Eq(alpha_a, T_a*p_alpha1 - p_alpha2)
eq_ka: Eq(k_a, T_a*p_ka1 + p_ka2)
eq_nua: Eq(nu_a, T_a*p_nua1 - p_nua2)
eq_rhoa_Pwa_Ta: Eq(rho_a, (M_N2*P_N2 + M_O2*P_O2 + M_w*P_wa)/(R_mol*T_a))
eq_Pa: Eq(P_a, P_N2 + P_O2 + P_wa)
eq_PN2_PO2: Eq(P_N2, P_O2*x_N2/x_O2)
eq_ideal_gas_law: Eq(P_g*V_g, R_mol*T_g*n_g)
eq_Pwa_CC: Eq(P_wa, p_CC1*exp(-M_w*lambda_E*(-1/p_CC2 + 1/T_g)/R_mol))
eq1: Eq(P_wa, Piecewise((0, T_a < 0), (p_CC1*exp(-M_w*lambda_E*(-1/p_CC2 + 1/T_g)/R_
↪mol), True)))
eq_Pwa_Delta: Eq(P_wa, P_wa1 + Integral(Delta_Pwa, (T_g, T_a1, T_a2)))
eq_PO2: Eq(P_O2, (P_a*x_O2 - P_wa*x_O2)/(x_N2 + x_O2))
eq_PN2: Eq(P_N2, (P_a*x_N2 - P_wa*x_N2)/(x_N2 + x_O2))
eq_rhoa: Eq(rho_a, (x_N2*(M_N2*P_a - P_wa*(M_N2 - M_w)) + x_O2*(M_O2*P_a - P_wa*(M_O2␣
↪- M_w)))/(R_mol*T_a*x_N2 + R_mol*T_a*x_O2))
eq_Pg: Eq(P_g, R_mol*T_g*n_g/V_g)
eq_Pwa_nw: Eq(P_wa, R_mol*T_g*n_w/V_g)
```

Substitution of equations and values into equations

The easiest way is to define a dictionary with all variables we want to substitute as keys. We start with the default variables and then add more. First, however, we will define a function to display the contents of a dictionary:

[4]:
```
def print_dict(vdict, list_vars=None):
    """Print values and units of variables in vdict."""
    if not list_vars:
        list_vars = vdict.keys()
    for var1 in list_vars:
        unit1 = var1.definition.unit
        if unit1 == 1:
            unit1 = ''
        if vdict[var1] is not None:
            print('{0}: {1} {2}'.format(var1.name, str(vdict[var1]), str(unit1)))
```

[5]:
```
vdict = Variable.__defaults__.copy()
print_dict(vdict)
```

```
c_pa: 1010.0 J/(K*kg)
c_pamol: 29.19 J/(K*mol)
c_pv: 1864 J/(K*kg)
g: 9.81 m/s**2
lambda_E: 2450000.0 J/kg
M_air: 0.02897 kg/mol
M_N2: 0.028 kg/mol
M_O2: 0.032 kg/mol
M_w: 0.018 kg/mol
R_mol: 8.314472 J/(K*mol)
sigm: 5.67e-08 J/(K**4*m**2*s)
T0: 273.15 K
x_N2: 0.79
x_O2: 0.21
p_Dva1: 1.49e-07 m**2/(K*s)
p_Dva2: 1.96e-05 m**2/s
p_alpha1: 1.32e-07 m**2/(K*s)
p_alpha2: 1.73e-05 m**2/s
p_ka1: 6.84e-05 J/(K**2*m*s)
p_ka2: 0.00563 J/(K*m*s)
p_nua1: 9e-08 m**2/(K*s)
p_nua2: 1.13e-05 m**2/s
p_CC1: 611.0 Pa
p_CC2: 273.0 K
```

We can substitute a range of equations into each other by using the custom function `subs_eq`:

[6]:
```
from essm.variables.utils import subs_eq
subs_eq(eq_Le, [eq_alphaa, eq_Dva])
```

[6]:

$$N_{Le} = \frac{T_a p_1 - p_2}{T_a p_1 - p_2}$$

We can also use subs_eq to substitute equations into each other and a dictionary with values. We will first add an entry for T_a into the dictionary and then substitute:

[7]:
```
vdict[T_a] = 300.
subs_eq(eq_Le, [eq_alphaa, eq_Dva], vdict)
```

[7]:

$$N_{Le} = 0.888446215139442$$

Evaluation of equations for long lists of variable sets

Substitution of variables into equations takes a lot of time if they need to be evaluated for a large number of variables. We can use theano to speed this up:

[8]:
```
#import theano
from sympy.printing.theanocode import theano_function
import numpy as np
```

```
WARNING (theano.tensor.blas): Using NumPy C-API based implementation for BLAS␣
↪functions.
```

We will now create two long lists of values representing T_g and n_g respectively and show how long it takes to compute ideal gas law values.

[9]:
```
npoints = 10000
xmin = 290.
xmax = 310.
Tvals = np.arange(xmin, xmax, (xmax - xmin)/npoints)
xmin = 0.1
xmax = 0.5
nvals = np.arange(xmin, xmax, (xmax-xmin)/npoints)
```

[10]:
```
%%time
# looping
expr = eq_ideal_gas_law.rhs.subs(Variable.__defaults__)
resvals0 = []
for i in range(len(Tvals)):
    resvals0.append(expr.subs({T_g: Tvals[i], n_g: nvals[i]}))
```

```
CPU times: user 7.46 s, sys: 43.4 ms, total: 7.5 s
Wall time: 7.68 s
```

[11]:
```
%%time
# Using theano
f1 = theano_function([T_g, n_g], [eq_ideal_gas_law.rhs.subs(Variable.__defaults__)],␣
↪dims={T_g:1, n_g:1})
resvals1 = f1(Tvals,nvals)
```

```
CPU times: user 106 ms, sys: 11.7 ms, total: 118 ms
Wall time: 623 ms
```

[12]:
```
list(resvals0) == list(resvals1)
```

[12]:
```
True
```

Both approaches give identical results, but ‘‘theano_function‘‘ makes it a lot faster.

Numerical solution

Some equations cannot be solved analytically for a given variable, e.g. eq_Nu_forced_all cannot be solved analytically for Re if Nu is given, so we can use numerical solvers instead:

[13]:
```
from sympy import nsolve
```

[14]:
```
vdict = Variable.__defaults__.copy()
vdict[Pr] = 0.71
vdict[Re_c] = 3000.
vdict[Nu] = 1000.
expr = eq_Nu_forced_all.subs(vdict)
nsolve(expr, 1000.)
```

[14]:

$$690263.0346446$$

Now applying to a long list of Nu-values:

[15]:
```
npoints = 100
xmin = 1000.
xmax = 1200.
Nuvals = np.arange(xmin, xmax, (xmax - xmin)/npoints)
```

[16]:
```
%%time
# Solving for a range of Nu values
vdict = Variable.__defaults__.copy()
vdict[Pr] = 0.71
vdict[Re_c] = 3000.
resvals = []
for Nu1 in Nuvals:
    vdict[Nu] = Nu1
    resvals.append(nsolve(eq_Nu_forced_all.subs(vdict), 1000.))
```

```
CPU times: user 1.56 s, sys: 3.9 ms, total: 1.56 s
Wall time: 1.56 s
```

We will now again use a theano function to make it faster. First we import optimize from scipy and preapre the theano_function:

[17]:
```
import scipy.optimize as sciopt
vdict = Variable.__defaults__.copy()
vdict[Pr] = 0.71
vdict[Re_c] = 3000.
expr = eq_Nu_forced_all.subs(vdict)
expr1 = expr.rhs - expr.lhs
fun_tf = theano_function([Re, Nu], [expr1], dims={Nu:1, Re:1})
x0vals = np.full(Nuvals.shape, fill_value=2000.) # array of same shape as Nuvals,␣
↪with initial guess
```

[18]:
```
%%time
# Solving for a range of Nu values
resvals1 = sciopt.fsolve(fun_tf, args=Nuvals, x0=x0vals)
```

```
CPU times: user 7.35 ms, sys: 0 ns, total: 7.35 ms
Wall time: 7.24 ms
```

[19]:
```
np.mean(abs((resvals - resvals1)/resvals))
```

[19]:

$$5.35695853236626 \cdot 10^{-11}$$

Using theano and scipy makes it 2 orders of magnitude faster and the results are different only by 10:math:'^{-10}'%! Note, however, that scipy gets slowed down for large arrays, so it is more efficient to re-run it repreatedly with subsections of the arra:

[20]:
```
npoints = 1000
xmin = 1000.
xmax = 1200.
Nuvals = np.arange(xmin, xmax, (xmax - xmin)/npoints)
x0vals = np.full(Nuvals.shape, fill_value=2000.)
```

[21]:
```
%%time
# Solving for a range of Nu values
resvals1 = sciopt.fsolve(fun_tf, args=Nuvals, x0=x0vals)
```

```
CPU times: user 1.6 s, sys: 4.13 ms, total: 1.61 s
Wall time: 1.61 s
```

We will now test that we can process Nuvals bit by bit and re-create it consistently:

[22]:
```
# Solving for a range of Nu values
imax = len(Nuvals)
i0 = 0
idiff = 100
i1 = i0
resvals2 = []
while i1 < imax - 1:
    i0 = i1    # note that resvals[0:2] + resvals[2:4] = resvals[0:4]
    i1 = min(i0+idiff, imax)
    resvals0 = Nuvals[i0:i1]
    resvals2 = np.append(resvals2,resvals0)
print(list(resvals2) == list(Nuvals))
```

```
True
```

Now we will run fsolve for portions of Nuvals bit by bit:

[23]:
```
%%time
# Solving for a range of Nu values
imax = len(Nuvals)
i0 = 0
idiff = 100
i1 = i0
resvals2 = []
while i1 < imax - 1:
    i0 = i1    # note that resvals[0:2] + resvals[2:4] = resvals[0:4]
    i1 = min(i0+idiff, imax)
    resvals0 = sciopt.fsolve(fun_tf, args=Nuvals[i0:i1], x0=x0vals[i0:i1])
    resvals2 = np.append(resvals2,resvals0)
```

```
CPU times: user 56.5 ms, sys: 16 µs, total: 56.5 ms
Wall time: 56 ms
```

[24]:
```
np.mean(abs((resvals1 - resvals2)/resvals1))
```

[24]:

$$7.122603685219754e-10$$

It is strange that resvals1 and resvals2 are different at all, but anyway, it is clear that slicing the data in relatively small portions is important to keep ‘‘scipy.optimize.fsolve‘‘ time-efficient.

Generate code from sympy expressions and execute

Need to install gfortran system-wide first!

[25]:
```
from sympy.utilities.autowrap import autowrap
```

[26]:
```
from sympy import symbols
x, y, z = symbols('x y z')
expr = ((x - y + z)**(13)).expand()
autowrap_func = autowrap(expr)
```

[27]:
```
%%time
autowrap_func(1, 4, 2)
```

```
CPU times: user 8 µs, sys: 0 ns, total: 8 µs
Wall time: 13.4 µs
```

[27]:

$$-1.0$$

[28]:
```
%%time
expr.subs({x:1, y:4, z:2})
```

```
CPU times: user 70 ms, sys: 3.77 ms, total: 73.8 ms
Wall time: 72.4 ms
```

[28]:

$$-1$$

Use of `autowrap` made the calculation 3 orders of magnitude faster than substitution of values into the original expression!

Another way is to use `binary_function`:

[29]:
```
from sympy.utilities.autowrap import binary_function
f = binary_function('f', expr)
```

[30]:
```
%%time
f(x,y,z).evalf(2, subs={x:1, y:4, z:2})
```

```
CPU times: user 2.47 ms, sys: 0 ns, total: 2.47 ms
Wall time: 2.65 ms
```

[30]:

$$-1.0$$

However, for the above example, `binary_function` was 2 orders of magnitude slower than `autowrap`.

[ ]:
```

```

CHAPTER 4

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

4.1 API Docs

4.1.1 Jupyter notebooks with API examples

A jupyter notebook with tables of importable variables and equations can be found at https://github.com/environmentalscience/essm/blob/master/docs/examples/importable_variables_equations.ipynb

A jupyter notebook with use examples for the API can be found at https://github.com/environmentalscience/essm/blob/master/docs/examples/api_features.ipynb

4.1.2 Variables

Variables module to deal with physical variables and units.

It allows attaching docstrings to variable names, defining their domains (e.g. integer, real or complex), their units and LaTeX representations. You can also provide a default value, which is particularly useful for physical constants.

Creating variables

To create custom variables, first import *Variable*:

```
>>> from essm.variables import Variable
```

To define units, you must first import these units from the library:

```
>>> from essm.variables.units import joule, kelvin, meter
```

Then you can define a custom variable with its name, description, domain, latex_name, unit, and an optional default value.

Example: .. code-block:: python

from .variables.units import meter

class demo_chamber_volume1(Variable): ‘’‘Volume of Chamber 1.’‘’

name = ‘V_c1’ domain = ‘real’ latex_name = ‘V_{c1}’ unit = meter ** 3 default = 1

Now, demo_chamber_volume is displayed as V_c1 and it will be nicely rendered in LaTeX as V_{c1}.

You can type *help(demo_chamber_volume)* to inspects its metadata.

Variable.__defaults__ returns a dictionary with all variables and their default values, *Variable.__units__* returns their units, and *demo_chamber_volume.short_unit()* can be used to obtain the units in short notation.

This module also contains libraries of pre-defined variables, which can be imported into your session, e.g.:

```
>>> from essm.variables.physics.thermodynamics import *
>>> from essm.variables.leaf.energy_water import *
```

Chamber

Chamber related variables.

Mass

Chamber mass and energy balance.

class essm.variables.chamber.mass.**W_c**

Bases: *essm.variables._core.Variable*

Chamber width.

assumptions = {'real': True}

latex_name = 'W_c'

name = 'W_c'

unit = meter

class essm.variables.chamber.mass.**L_c**

Bases: *essm.variables._core.Variable*

Chamber length.

assumptions = {'real': True}

latex_name = 'L_c'

name = 'L_c'

unit = meter

class essm.variables.chamber.mass.**H_c**

Bases: *essm.variables._core.Variable*

Chamber height.

assumptions = {'real': True}

latex_name = 'H_c'

name = 'H_c'

unit = meter

class essm.variables.chamber.mass.**V_c**

Bases: *essm.variables._core.Variable*

Chamber volume.

assumptions = {'real': True}

latex_name = 'V_c'

name = 'V_c'

unit = meter3**

class essm.variables.chamber.mass.**n_c**

Bases: *essm.variables._core.Variable*

molar mass of gas in chamber.

assumptions = {'real': True}

latex_name = 'n_c'

name = 'n_c'

unit = mole

class essm.variables.chamber.mass.**F_in_v**

Bases: *essm.variables._core.Variable*

Volumetric flow rate into chamber.

assumptions = {'real': True}

latex_name = 'F_{in,v}'

name = 'F_in_v'

unit = meter3/second**

class essm.variables.chamber.mass.**F_in_mola**

Bases: *essm.variables._core.Variable*

Molar flow rate of dry air into chamber.

assumptions = {'real': True}

latex_name = 'F_{in,mol,a}'

name = 'F_in_mola'

unit = mole/second

class essm.variables.chamber.mass.**F_in_molw**

Bases: *essm.variables._core.Variable*

Molar flow rate of water vapour into chamber.

assumptions = {'real': True}

latex_name = 'F_{in,mol,w}'

name = 'F_in_molw'

unit = mole/second

class essm.variables.chamber.mass.**F_out_mola**
Bases: *essm.variables._core.Variable*

Molar flow rate of dry air out of chamber.

assumptions = {'real': True}

latex_name = 'F_{out,mol,a}'

name = 'F_out_mola'

unit = mole/second

class essm.variables.chamber.mass.**F_out_molw**
Bases: *essm.variables._core.Variable*

Molar flow rate of water vapour out of chamber.

assumptions = {'real': True}

latex_name = 'F_{out,mol,w}'

name = 'F_out_molw'

unit = mole/second

class essm.variables.chamber.mass.**F_out_v**
Bases: *essm.variables._core.Variable*

Volumetric flow rate out of chamber.

assumptions = {'real': True}

latex_name = 'F_{out,v}'

name = 'F_out_v'

unit = meter3/second**

class essm.variables.chamber.mass.**T_d**
Bases: *essm.variables._core.Variable*

Dew point temperature of incoming air.

assumptions = {'real': True}

latex_name = 'T_d'

name = 'T_d'

unit = kelvin

class essm.variables.chamber.mass.**T_in**
Bases: *essm.variables._core.Variable*

Temperature of incoming air.

assumptions = {'real': True}

latex_name = 'T_{in}'

name = 'T_in'

unit = kelvin

class essm.variables.chamber.mass.**T_out**
Bases: *essm.variables._core.Variable*

Temperature of outgoing air (= chamber T_a).

assumptions = {'real': True}

latex_name = 'T_{out}'

name = 'T_out'

unit = kelvin

class essm.variables.chamber.mass.**T_room**

Bases: *essm.variables._core.Variable*

Lab air temperature.

assumptions = {'real': True}

latex_name = 'T_{room}'

name = 'T_room'

unit = kelvin

class essm.variables.chamber.mass.**P_w_in**

Bases: *essm.variables._core.Variable*

Vapour pressure of incoming air.

assumptions = {'real': True}

latex_name = 'P_{w,in}'

name = 'P_w_in'

unit = pascal

class essm.variables.chamber.mass.**P_w_out**

Bases: *essm.variables._core.Variable*

Vapour pressure of outgoing air.

assumptions = {'real': True}

latex_name = 'P_{w,out}'

name = 'P_w_out'

unit = pascal

class essm.variables.chamber.mass.**R_H_in**

Bases: *essm.variables._core.Variable*

Relative humidity of incoming air.

assumptions = {'real': True}

latex_name = 'R_{H,in}'

name = 'R_H_in'

unit = 1

class essm.variables.chamber.mass.**L_A**

Bases: *essm.variables._core.Variable*

Leaf area.

assumptions = {'real': True}

latex_name = 'L_A'

name = 'L_A'

unit = meter2**

Insulation

Chamber insulation material.

class essm.variables.chamber.insulation.**c_pi**

Bases: *essm.variables._core.Variable*

Heat capacity of insulation material.

assumptions = {'real': True}

latex_name = 'c_{pi}'

name = 'c_pi'

unit = joule/(kelvin*kilogram)

class essm.variables.chamber.insulation.**lambda_i**

Bases: *essm.variables._core.Variable*

Heat conductivity of insulation material.

assumptions = {'real': True}

latex_name = 'lambda_i'

name = 'lambda_i'

unit = joule/(kelvin*meter*second)

class essm.variables.chamber.insulation.**rho_i**

Bases: *essm.variables._core.Variable*

Density of insulation material.

assumptions = {'real': True}

latex_name = 'rho_i'

name = 'rho_i'

unit = kilogram/meter3**

class essm.variables.chamber.insulation.**L_i**

Bases: *essm.variables._core.Variable*

Thickness of insulation material.

assumptions = {'real': True}

latex_name = 'L_i'

name = 'L_i'

unit = meter

class essm.variables.chamber.insulation.**A_i**

Bases: *essm.variables._core.Variable*

Conducting area of insulation material.

assumptions = {'real': True}

latex_name = 'A_i'

name = 'A_i'

unit = meter2**

class essm.variables.chamber.insulation.**Q_i**
Bases: *essm.variables._core.Variable*

Heat conduction through insulation material.

assumptions = {'real': True}

latex_name = 'Q_i'

name = 'Q_i'

unit = joule/second

class essm.variables.chamber.insulation.**dT_i**
Bases: *essm.variables._core.Variable*

Temperature increment of insulation material.

assumptions = {'real': True}

latex_name = 'dT_i'

name = 'dT_i'

unit = kelvin

Leaf

Variables related to leaf energy and water balance.

Radiation

Leaf radiation balance.

class essm.variables.leaf.radiation.**alpha_l**
Bases: *essm.variables._core.Variable*

Leaf albedo, fraction of shortwave radiation reflected by the leaf.

assumptions = {'real': True}

latex_name = 'alpha_l'

name = 'alpha_l'

unit = 1

class essm.variables.leaf.radiation.**R_d**
Bases: *essm.variables._core.Variable*

Downwelling global radiation.

assumptions = {'real': True}

latex_name = 'R_d'

name = 'R_d'

unit = joule/(meter2*second)**

class essm.variables.leaf.radiation.**R_la**
Bases: *essm.variables._core.Variable*

Longwave radiation absorbed by leaf.

assumptions = {'real': True}

latex_name = 'R_{la}'

name = 'R_la'

unit = joule/(meter2*second)**

class essm.variables.leaf.radiation.**R_ld**
Bases: *essm.variables._core.Variable*

Downwards emitted/reflected global radiation from leaf.

assumptions = {'real': True}

latex_name = 'R_{ld}'

name = 'R_ld'

unit = joule/(meter2*second)**

class essm.variables.leaf.radiation.**R_lu**
Bases: *essm.variables._core.Variable*

Upwards emitted/reflected global radiation from leaf.

assumptions = {'real': True}

latex_name = 'R_{lu}'

name = 'R_lu'

unit = joule/(meter2*second)**

class essm.variables.leaf.radiation.**R_u**
Bases: *essm.variables._core.Variable*

Upwelling global radiation.

assumptions = {'real': True}

latex_name = 'R_u'

name = 'R_u'

unit = joule/(meter2*second)**

class essm.variables.leaf.radiation.**S_a**
Bases: *essm.variables._core.Variable*

Radiation sensor above leaf reading.

assumptions = {'real': True}

latex_name = 'S_a'

name = 'S_a'

unit = joule/(meter2*second)**

class essm.variables.leaf.radiation.**S_b**
Bases: *essm.variables._core.Variable*

Radiation sensor below leaf reading.

assumptions = {'real': True}

latex_name = 'S_b'

name = 'S_b'

unit = joule/(meter2*second)**

class essm.variables.leaf.radiation.**S_s**

Bases: *essm.variables._core.Variable*

Radiation sensor beside leaf reading.

assumptions = {'real': True}

latex_name = 'S_s'

name = 'S_s'

unit = joule/(meter2*second)**

Energy and Water

Unsorted variables related to leaf model.

class essm.variables.leaf.energy_water.**alpha_l**

Bases: *essm.variables._core.Variable*

Leaf albedo, fraction of shortwave radiation reflected by the leaf.

assumptions = {'real': True}

domain = 'real'

latex_name = '\\alpha_l'

name = 'alpha_l'

unit = 1

class essm.variables.leaf.energy_water.**a_s**

Bases: *essm.variables._core.Variable*

Fraction of one-sided leaf area covered by stomata.

(1 if stomata are on one side only, 2 if they are on both sides).

assumptions = {'real': True}

domain = 'real'

latex_name = 'a_s'

name = 'a_s'

unit = 1

class essm.variables.leaf.energy_water.**a_sh**

Bases: *essm.variables._core.Variable*

Fraction of projected area exchanging sensible heat with the air.

assumptions = {'real': True}

default = 2.0

domain = 'real'

latex_name = 'a_{sh}'

name = 'a_sh'

unit = 1

class essm.variables.leaf.energy_water.**C_wl**

Bases: *essm.variables._core.Variable*

Concentration of water in the leaf air space.

assumptions = {'real': True}

domain = 'real'

latex_name = 'C_{wl}'

name = 'C_wl'

unit = mole/meter3**

class essm.variables.leaf.energy_water.**E_lmol**

Bases: *essm.variables._core.Variable*

Transpiration rate in molar units.

assumptions = {'real': True}

domain = 'real'

latex_name = 'E_{l,mol}'

name = 'E_lmol'

unit = mole/(meter2*second)**

class essm.variables.leaf.energy_water.**E_l**

Bases: *essm.variables._core.Variable*

Latent heat flux from leaf.

assumptions = {'real': True}

domain = 'real'

latex_name = 'E_l'

name = 'E_l'

unit = joule/(meter2*second)**

class essm.variables.leaf.energy_water.**epsilon_l**

Bases: *essm.variables._core.Variable*

Longwave emmissivity of the leaf surface.

assumptions = {'real': True}

default = 1.0

domain = 'real'

latex_name = '\\epsilon_l'

name = 'epsilon_l'

unit = 1

class essm.variables.leaf.energy_water.**g_bw**

Bases: *essm.variables._core.Variable*

Boundary layer conductance to water vapour.

assumptions = {'real': True}

domain = 'real'

latex_name = 'g_{bw}'

name = 'g_bw'

unit = meter/second

class essm.variables.leaf.energy_water.**g_bwmol**

Bases: *essm.variables._core.Variable*

Boundary layer conductance to water vapour.

assumptions = {'real': True}

domain = 'real'

latex_name = 'g_{bw,mol}'

name = 'g_bwmol'

unit = mole/(meter2*second)**

class essm.variables.leaf.energy_water.**Gr**

Bases: *essm.variables._core.Variable*

Grashof number.

assumptions = {'real': True}

domain = 'real'

latex_name = 'N_{Gr_L}'

name = 'Gr'

unit = 1

class essm.variables.leaf.energy_water.**g_sw**

Bases: *essm.variables._core.Variable*

Stomatal conductance to water vapour.

assumptions = {'real': True}

domain = 'real'

latex_name = 'g_{sw}'

name = 'g_sw'

unit = meter/second

class essm.variables.leaf.energy_water.**g_swmol**

Bases: *essm.variables._core.Variable*

Stomatal conductance to water vapour.

assumptions = {'real': True}

domain = 'real'

latex_name = 'g_{sw,mol}'

name = 'g_swmol'

unit = mole/(meter2*second)**

class essm.variables.leaf.energy_water.**g_tw**

Bases: *essm.variables._core.Variable*

Total leaf conductance to water vapour.

assumptions = {'real': True}

domain = 'real'

latex_name = 'g_{tw}'

name = 'g_tw'

unit = meter/second

class essm.variables.leaf.energy_water.**g_twmol**

Bases: *essm.variables._core.Variable*

Total leaf layer conductance to water vapour.

assumptions = {'real': True}

domain = 'real'

latex_name = 'g_{tw,mol}'

name = 'g_twmol'

unit = mole/(meter2*second)**

class essm.variables.leaf.energy_water.**h_c**

Bases: *essm.variables._core.Variable*

Average 1-sided convective heat transfer coefficient.

assumptions = {'real': True}

domain = 'real'

latex_name = 'h_c'

name = 'h_c'

unit = joule/(kelvin*meter2*second)**

class essm.variables.leaf.energy_water.**H_l**

Bases: *essm.variables._core.Variable*

Sensible heat flux from leaf.

assumptions = {'real': True}

domain = 'real'

latex_name = 'H_l'

name = 'H_l'

unit = joule/(meter2*second)**

class essm.variables.leaf.energy_water.**L_A**

Bases: *essm.variables._core.Variable*

Leaf area.

assumptions = {'real': True}

domain = 'real'

latex_name = 'L_A'

name = 'L_A'

unit = meter2**

class essm.variables.leaf.energy_water.**L_l**

Bases: *essm.variables._core.Variable*

Leaf width as characteristic length scale for convection.

assumptions = {'real': True}

domain = 'real'

latex_name = 'L_l'

name = 'L_l'

unit = meter

class essm.variables.leaf.energy_water.**P_wl**

Bases: *essm.variables._core.Variable*

Water vapour pressure inside the leaf.

assumptions = {'real': True}

domain = 'real'

latex_name = 'P_{wl}'

name = 'P_wl'

unit = pascal

class essm.variables.leaf.energy_water.**r_bw**

Bases: *essm.variables._core.Variable*

Boundary layer resistance to water vapour, inverse of g_{bw}.

assumptions = {'real': True}

domain = 'real'

latex_name = 'r_{bw}'

name = 'r_bw'

unit = second/meter

class essm.variables.leaf.energy_water.**r_sw**

Bases: *essm.variables._core.Variable*

Stomatal resistance to water vapour, inverse of g_{sw}.

assumptions = {'real': True}

domain = 'real'

latex_name = 'r_{sw}'

name = 'r_sw'

unit = second/meter

class essm.variables.leaf.energy_water.**r_tw**

Bases: *essm.variables._core.Variable*

Total leaf resistance to water vapour, $r_{bv} + r_{sv}$.

assumptions = {'real': True}

domain = 'real'

latex_name = 'r_{tw}'

name = 'r_tw'

unit = second/meter

class essm.variables.leaf.energy_water.**rho_al**

Bases: *essm.variables._core.Variable*

Density of air at the leaf surface.

assumptions = {'real': True}

domain = 'real'

latex_name = '\\rho_{al}'

name = 'rho_al'

unit = kilogram/meter3**

class essm.variables.leaf.energy_water.**R_la**

Bases: *essm.variables._core.Variable*

Longwave radiation absorbed by leaf.

assumptions = {'real': True}

domain = 'real'

latex_name = 'R_{la}'

name = 'R_la'

unit = watt/meter2**

class essm.variables.leaf.energy_water.**R_ll**

Bases: *essm.variables._core.Variable*

Longwave radiation away from leaf.

assumptions = {'real': True}

domain = 'real'

latex_name = 'R_{ll}'

name = 'R_ll'

unit = watt/meter2**

class essm.variables.leaf.energy_water.**R_ld**

Bases: *essm.variables._core.Variable*

Downwards emitted/reflected global radiation from leaf.

assumptions = {'real': True}

domain = 'real'

latex_name = 'R_{ld}'

name = 'R_ld'

unit = watt/meter2**

class essm.variables.leaf.energy_water.**R_lu**

Bases: *essm.variables._core.Variable*

Upwards emitted/reflected global radiation from leaf.

assumptions = {'real': True}

domain = 'real'

latex_name = 'R_{lu}'

name = 'R_lu'

unit = watt/meter2**

class essm.variables.leaf.energy_water.**T_l**

Bases: *essm.variables._core.Variable*

Leaf temperature.

assumptions = {'real': True}

domain = 'real'

latex_name = 'T_l'

name = 'T_l'

unit = kelvin

class essm.variables.leaf.energy_water.**T_w**

Bases: *essm.variables._core.Variable*

Radiative temperature of objects surrounding the leaf.

assumptions = {'real': True}

domain = 'real'

latex_name = 'T_w'

name = 'T_w'

unit = kelvin

Physics

General and atmospheric thermodynamics variables.

class essm.variables.physics.thermodynamics.**alpha_a**

Bases: *essm.variables._core.Variable*

Thermal diffusivity of dry air.

assumptions = {'real': True}

domain = 'real'

latex_name = '\\alpha_a'

name = 'alpha_a'

unit = meter2/second**

class essm.variables.physics.thermodynamics.**c_pa**

Bases: *essm.variables._core.Variable*

Specific heat of dry air.

assumptions = {'real': True}

default = 1010.0

domain = 'real'

latex_name = 'c_{pa}'

name = 'c_pa'

unit = joule/(kelvin*kilogram)

class essm.variables.physics.thermodynamics.**c_pamol**

Bases: *essm.variables._core.Variable*

Molar specific heat of dry air.

https://en.wikipedia.org/wiki/Heat_capacity#Specific_heat_capacity

assumptions = {'real': True}

default = 29.19

domain = 'real'

latex_name = 'c_{pa,mol}'

name = 'c_pamol'

unit = joule/(kelvin*mole)

class essm.variables.physics.thermodynamics.**c_pv**

Bases: *essm.variables._core.Variable*

Specific heat of water vapour at 300 K.

http://www.engineeringtoolbox.com/water-vapor-d_979.html

assumptions = {'real': True}

default = 1864

domain = 'real'

latex_name = 'c_{pv}'

name = 'c_pv'

unit = joule/(kelvin*kilogram)

class essm.variables.physics.thermodynamics.**C_wa**

Bases: *essm.variables._core.Variable*

Concentration of water in air.

assumptions = {'real': True}

domain = 'real'

latex_name = 'C_{wa}'

name = 'C_wa'

unit = mole/meter3**

class essm.variables.physics.thermodynamics.**D_va**

Bases: *essm.variables._core.Variable*

Binary diffusion coefficient of water vapour in air.

assumptions = {'real': True}

domain = 'real'

latex_name = 'D_{va}'

name = 'D_va'

unit = meter2/second**

class essm.variables.physics.thermodynamics.**g**

Bases: *essm.variables._core.Variable*

Gravitational acceleration.

assumptions = {'real': True}

default = 9.81

domain = 'real'

latex_name = 'g'

name = 'g'

unit = meter/second2**

class essm.variables.physics.thermodynamics.**Gr**

Bases: *essm.variables._core.Variable*

Grashof number.

assumptions = {'real': True}

domain = 'real'

latex_name = 'N_{Gr_L}'

name = 'Gr'

unit = 1

class essm.variables.physics.thermodynamics.**h_c**

Bases: *essm.variables._core.Variable*

Average 1-sided convective heat transfer coefficient.

assumptions = {'real': True}

domain = 'real'

latex_name = 'h_c'

name = 'h_c'

unit = joule/(kelvin*meter2*second)**

class essm.variables.physics.thermodynamics.**k_a**

Bases: *essm.variables._core.Variable*

Thermal conductivity of dry air.

assumptions = {'real': True}

domain = 'real'

latex_name = 'k_a'

name = 'k_a'

unit = joule/(kelvin*meter*second)

class essm.variables.physics.thermodynamics.**lambda_E**

Bases: *essm.variables._core.Variable*

Latent heat of evaporation.

assumptions = {'real': True}

default = 2450000.0

domain = 'real'

latex_name = '\\lambda_E'

name = 'lambda_E'

unit = joule/kilogram

class essm.variables.physics.thermodynamics.**Le**

Bases: *essm.variables._core.Variable*

Lewis number.

assumptions = {'real': True}

domain = 'real'

latex_name = 'N_{Le}'

name = 'Le'

unit = 1

class essm.variables.physics.thermodynamics.**M_air**

Bases: *essm.variables._core.Variable*

Molar mass of air.

http://www.engineeringtoolbox.com/molecular-mass-air-d_679.html

assumptions = {'real': True}

default = 0.02897

domain = 'real'

latex_name = 'M_{air}'

name = 'M_air'

unit = kilogram/mole

class essm.variables.physics.thermodynamics.**M_N2**

Bases: *essm.variables._core.Variable*

Molar mass of nitrogen.

assumptions = {'real': True}

default = 0.028

domain = 'real'

latex_name = 'M_{N_2}'

name = 'M_N2'

unit = kilogram/mole

class essm.variables.physics.thermodynamics.**M_O2**

Bases: *essm.variables._core.Variable*

Molar mass of oxygen.

assumptions = {'real': True}

default = 0.032

domain = 'real'

latex_name = 'M_{O_2}'

name = 'M_O2'

unit = kilogram/mole

class essm.variables.physics.thermodynamics.**M_w**

Bases: *essm.variables._core.Variable*

Molar mass of water.

assumptions = {'real': True}

default = 0.018

domain = 'real'

latex_name = 'M_w'

name = 'M_w'

unit = kilogram/mole

class essm.variables.physics.thermodynamics.**nu_a**

Bases: *essm.variables._core.Variable*

Kinematic viscosity of dry air.

assumptions = {'real': True}

domain = 'real'

latex_name = '\\nu_a'

name = 'nu_a'

unit = meter2/second**

class essm.variables.physics.thermodynamics.**Nu**

Bases: *essm.variables._core.Variable*

Average Nusselt number over given length.

assumptions = {'real': True}

domain = 'real'

latex_name = 'N_{Nu_L}'

name = 'Nu'

unit = 1

class essm.variables.physics.thermodynamics.**P_a**

Bases: *essm.variables._core.Variable*

Air pressure.

assumptions = {'real': True}

domain = 'real'

latex_name = 'P_a'

name = 'P_a'

unit = pascal

class essm.variables.physics.thermodynamics.**Pr**

Bases: *essm.variables._core.Variable*

Prandtl number (0.71 for air).

assumptions = {'real': True}

domain = 'real'

latex_name = 'N_{Pr}'

name = 'Pr'

unit = 1

class essm.variables.physics.thermodynamics.**P_N2**

Bases: *essm.variables._core.Variable*

Partial pressure of nitrogen.

assumptions = {'real': True}

domain = 'real'

latex_name = 'P_{N2}'

name = 'P_N2'

unit = pascal

class essm.variables.physics.thermodynamics.**P_O2**

Bases: *essm.variables._core.Variable*

Partial pressure of oxygen.

assumptions = {'real': True}

domain = 'real'

latex_name = 'P_{O2}'

name = 'P_O2'

unit = pascal

class essm.variables.physics.thermodynamics.**P_wa**

Bases: *essm.variables._core.Variable*

Water vapour pressure in the atmosphere.

assumptions = {'real': True}

domain = 'real'

latex_name = 'P_{wa}'

name = 'P_wa'

unit = pascal

class essm.variables.physics.thermodynamics.**P_was**

Bases: *essm.variables._core.Variable*

Saturation water vapour pressure at air temperature.

assumptions = {'real': True}

domain = 'real'

latex_name = 'P_{was}'

name = 'P_was'

unit = pascal

class essm.variables.physics.thermodynamics.**R_d**

Bases: *essm.variables._core.Variable*

Downwelling global radiation.

assumptions = {'real': True}

domain = 'real'

latex_name = 'R_d'

name = 'R_d'

unit = watt/meter2**

class essm.variables.physics.thermodynamics.**Re_c**

Bases: *essm.variables._core.Variable*

Critical Reynolds number for the onset of turbulence.

assumptions = {'real': True}

domain = 'real'

latex_name = 'N_{Re_c}'

name = 'Re_c'

unit = 1

class essm.variables.physics.thermodynamics.**Re**

Bases: *essm.variables._core.Variable*

Average Reynolds number over given length.

assumptions = {'real': True}

domain = 'real'

latex_name = 'N_{Re_L}'

name = 'Re'

unit = 1

class essm.variables.physics.thermodynamics.**rho_a**

Bases: *essm.variables._core.Variable*

Density of dry air.

assumptions = {'real': True}

domain = 'real'

latex_name = '\\rho_a'

name = 'rho_a'

unit = kilogram/meter3**

class essm.variables.physics.thermodynamics.**R_u**

Bases: *essm.variables._core.Variable*

Upwelling global radiation.

assumptions = {'real': True}

domain = 'real'

latex_name = 'R_u'

name = 'R_u'

unit = watt/meter2**

class essm.variables.physics.thermodynamics.**R_mol**

Bases: *essm.variables._core.Variable*

Molar gas constant.

assumptions = {'real': True}

default = 8.314472

domain = 'real'

latex_name = 'R_{mol}'

name = 'R_mol'

unit = joule/(kelvin*mole)

class essm.variables.physics.thermodynamics.**R_s**

Bases: *essm.variables._core.Variable*

Solar shortwave flux per area.

assumptions = {'real': True}

domain = 'real'

latex_name = 'R_s'

name = 'R_s'

unit = joule/(meter2*second)**

class essm.variables.physics.thermodynamics.**sigm**

Bases: *essm.variables._core.Variable*

Stefan-Boltzmann constant.

assumptions = {'real': True}

default = 5.67e-08

domain = 'real'

latex_name = '\\sigma'

name = 'sigm'

unit = joule/(kelvin4*meter**2*second)**

class essm.variables.physics.thermodynamics.**T0**

Bases: *essm.variables._core.Variable*

Freezing point in Kelvin.

assumptions = {'real': True}

default = 273.15

domain = 'real'

latex_name = 'T_0'

name = 'T0'

unit = kelvin

class essm.variables.physics.thermodynamics.**T_a**

Bases: *essm.variables._core.Variable*

Air temperature.

assumptions = {'real': True}

domain = 'real'

latex_name = 'T_a'

name = 'T_a'

unit = kelvin

class essm.variables.physics.thermodynamics.**v_w**

Bases: *essm.variables._core.Variable*

Wind velocity.

assumptions = {'real': True}

domain = 'real'

latex_name = 'v_w'

name = 'v_w'

unit = meter/second

class essm.variables.physics.thermodynamics.**x_N2**

Bases: *essm.variables._core.Variable*

Mole fraction of nitrogen in dry air.

assumptions = {'real': True}

default = 0.79

domain = 'real'

latex_name = 'x_{N2}'

name = 'x_N2'

unit = 1

class essm.variables.physics.thermodynamics.**x_O2**

Bases: *essm.variables._core.Variable*

Mole fraction of oxygen in dry air.

```
assumptions = {'real': True}
default = 0.21
domain = 'real'
latex_name = 'x_{O2}'
name = 'x_O2'
unit = 1
```

Units

Define unit symbols.

`essm.variables.units.`**`derive_baseunit`**(*expr*, *name=None*)
Derive SI base unit from an expression, omitting scale factors.

`essm.variables.units.`**`derive_unit`**(*expr*, *name=None*)
Derive SI unit from an expression, omitting scale factors.

`essm.variables.units.`**`markdown`**(*unit*)
Return markdown representation of a unit.

4.1.3 Equations

Equations module to deal with physical equations.

It allows attaching docstrings to equation names (including references) and defining internal variables.

Creating equations

To create custom equations, first import *Equation* and variables needed:

```
>>> from essm.equations import Equation
>>> from essm.variables.physics.thermodynamics import *
```

Importing pre-defined equations

You can import pre-defined equations as e.g.:

```
>>> from essm.equations.physics.thermodynamics import *
```

Leaf

Equations related to leaf energy and water balance.

Energy Water

Leaf energy and water balance equations.

class essm.equations.leaf.energy_water.**eq_Rs_enbal**

Bases: *essm.equations._core.Equation*

Calculate R_s from energy balance.

(Eq. 1 in [SO17])

expr = Eq(R_s, E_l + H_l + R_ll)

name = 'eq_Rs_enbal'

class essm.equations.leaf.energy_water.**eq_Rll**

Bases: *essm.equations._core.Equation*

R_ll as function of T_l and T_w.

(Eq. 2 in [SO17])

expr = Eq(R_ll, a_sh*epsilon_l*sigm*(T_l4 - T_w**4))**

name = 'eq_Rll'

class essm.equations.leaf.energy_water.**eq_Hl**

Bases: *essm.equations._core.Equation*

H_l as function of T_l.

(Eq. 3 in [SO17])

expr = Eq(H_l, a_sh*h_c*(-T_a + T_l))

name = 'eq_Hl'

class essm.equations.leaf.energy_water.**eq_El**

Bases: *essm.equations._core.Equation*

E_l as function of E_lmol.

(Eq. 4 in [SO17])

expr = Eq(E_l, E_lmol*M_w*lambda_E)

name = 'eq_El'

class essm.equations.leaf.energy_water.**eq_Elmol**

Bases: *essm.equations._core.Equation*

E_lmol as functino of g_tw and C_wl.

(Eq. 5 in [SO17])

expr = Eq(E_lmol, g_tw*(-C_wa + C_wl))

name = 'eq_Elmol'

class essm.equations.leaf.energy_water.**eq_gtw**

Bases: *essm.equations._core.Equation*

g_tw from g_sw and g_bw.

(Eq. 6 in [SO17])

expr = Eq(g_tw, 1/(1/g_sw + 1/g_bw))

name = 'eq_gtw'

class essm.equations.leaf.energy_water.**eq_gbw_hc**

Bases: *essm.equations._core.Equation*

g_bw as function of h_c.

(Eq. B2 in [SO17])

expr = Eq(g_bw, a_s*h_c/(Le(2/3)*c_pa*rho_a))**

name = 'eq_gbw_hc'

class essm.equations.leaf.energy_water.**eq_Cwl**

Bases: *essm.equations._core.Equation*

C_wl as function of P_wl and T_l.

(Eq. B4 in [SO17])

expr = Eq(C_wl, P_wl/(R_mol*T_l))

name = 'eq_Cwl'

class essm.equations.leaf.energy_water.**eq_Pwl**

Bases: *essm.equations._core.Equation*

Clausius-Clapeyron P_wl as function of T_l.

(Eq. B3 in [Har94])

expr = Eq(P_wl, p_CC1*exp(-M_w*lambda_E*(-1/p_CC2 + 1/T_l)/R_mol))

name = 'eq_Pwl'

p_CC1 = p_CC1

p_CC2 = p_CC2

class essm.equations.leaf.energy_water.**eq_Elmol_conv**

Bases: *essm.equations._core.Equation*

E_lmol as function of g_twmol and P_wl.

(Eq. B6 in [SO17])

expr = Eq(E_lmol, g_twmol*(-P_wa + P_wl)/P_a)

name = 'eq_Elmol_conv'

class essm.equations.leaf.energy_water.**eq_gtwmol_gtw**

Bases: *essm.equations.leaf.energy_water.eq_Elmol*, *essm.equations.leaf.energy_water.eq_Cwl*, *essm.equations.leaf.energy_water.eq_Elmol_conv*

g_twmol as a function of g_tw.

It uses eq_Elmol, eq_Cwl and eq_Elmol_conv.

expr = Eq(g_twmol, g_tw*(P_a*P_wa*T_l - P_a*P_wl*T_a)/(R_mol*T_a*T_l*(P_wa - P_wl)))

name = 'eq_gtwmol_gtw'

class essm.equations.leaf.energy_water.**eq_gtwmol_gtw_iso**

Bases: *essm.equations.leaf.energy_water.eq_gtwmol_gtw*

g_twmol as a function of g_tw at isothermal conditions.

expr = Eq(g_twmol, P_a*g_tw/(R_mol*T_a))

name = 'eq_gtwmol_gtw_iso'

class essm.equations.leaf.energy_water.**eq_hc**

Bases: *essm.equations._core.Equation*

h_c as a function of Nu and L_l.

(Eq. B10 in [SO17])

expr = Eq(h_c, Nu*k_a/L_l)

name = 'eq_hc'

class essm.equations.leaf.energy_water.**eq_Re**

Bases: *essm.equations._core.Equation*

Re as a function of v_w and L_l.

(Eq. B11 in [SO17])

expr = Eq(Re, L_l*v_w/nu_a)

name = 'eq_Re'

class essm.equations.leaf.energy_water.**eq_Gr**

Bases: *essm.equations._core.Equation*

Gr as function of air density within and outside of leaf.

(Eq. B12 in [SO17])

expr = Eq(Gr, L_l3*g*(rho_a - rho_al)/(nu_a**2*rho_al))**

name = 'eq_Gr'

Physics

General and atmospheric thermodynamics equations.

class essm.equations.physics.thermodynamics.**eq_Le**

Bases: *essm.equations._core.Equation*

Le as function of alpha_a and D_va.

(Eq. B3 in [SO17])

expr = Eq(Le, alpha_a/D_va)

name = 'eq_Le'

class essm.equations.physics.thermodynamics.**eq_Cwa**

Bases: *essm.equations._core.Equation*

C_wa as a function of P_wa and T_a.

(Eq. B9 in [SO17])

expr = Eq(C_wa, P_wa/(R_mol*T_a))

name = 'eq_Cwa'

class essm.equations.physics.thermodynamics.**eq_Nu_forced_all**

Bases: *essm.equations._core.Equation*

Nu as function of Re and Re_c under forced conditions.

(Eqs. B13–B15 in [SO17])

expr = Eq(Nu, -Pr(1/3)*(-37*Re**(4/5) + 37*(Re + Re_c - Abs(Re - Re_c)/2)**(4/5) - 6**

name = 'eq_Nu_forced_all'

class essm.equations.physics.thermodynamics.**eq_Dva**

Bases: *essm.equations._core.Equation*

D_va as a function of air temperature.

(Table A.3 in [MU07])

expr = Eq(D_va, T_a*p_Dva1 - p_Dva2)

name = 'eq_Dva'

p_Dva1 = p_Dva1

p_Dva2 = p_Dva2

class essm.equations.physics.thermodynamics.**eq_alphaa**

Bases: *essm.equations._core.Equation*

alpha_a as a function of air temperature.

(Table A.3 in [MU07])

expr = Eq(alpha_a, T_a*p_alpha1 - p_alpha2)

name = 'eq_alphaa'

p_alpha1 = p_alpha1

p_alpha2 = p_alpha2

class essm.equations.physics.thermodynamics.**eq_ka**

Bases: *essm.equations._core.Equation*

k_a as a function of air temperature.

(Table A.3 in [MU07])

expr = Eq(k_a, T_a*p_ka1 + p_ka2)

name = 'eq_ka'

p_ka1 = p_ka1

p_ka2 = p_ka2

class essm.equations.physics.thermodynamics.**eq_nua**

Bases: *essm.equations._core.Equation*

nu_a as a function of air temperature.

(Table A.3 in [MU07])

expr = Eq(nu_a, T_a*p_nua1 - p_nua2)

name = 'eq_nua'

p_nua1 = p_nua1

p_nua2 = p_nua2

class essm.equations.physics.thermodynamics.**eq_rhoa_Pwa_Ta**

Bases: *essm.equations._core.Equation*

rho_a as a function of P_wa and T_a.

(Eq. B20 in [SO17])

expr = Eq(rho_a, (M_N2*P_N2 + M_O2*P_O2 + M_w*P_wa)/(R_mol*T_a))

name = 'eq_rhoa_Pwa_Ta'

class essm.equations.physics.thermodynamics.**eq_Pa**

Bases: *essm.equations._core.Equation*

Calculate air pressure.

From partial pressures of N2, O2 and H2O, following Dalton's law of partial pressures.

expr = Eq(P_a, P_N2 + P_O2 + P_wa)

name = 'eq_Pa'

class essm.equations.physics.thermodynamics.**eq_PN2_PO2**

Bases: *essm.equations._core.Equation*

Calculate P_N2 as a function of P_O2.

It follows Dalton's law of partial pressures.

expr = Eq(P_N2, P_O2*x_N2/x_O2)

name = 'eq_PN2_PO2'

class essm.equations.physics.thermodynamics.**eq_PO2**

Bases: *essm.equations.physics.thermodynamics.eq_Pa*, *essm.equations.physics.thermodynamics.eq_PN2_PO2*

Calculate P_O2 as a function of P_a, P_N2 and P_wa.

expr = Eq(P_O2, (P_a*x_O2 - P_wa*x_O2)/(x_N2 + x_O2))

name = 'eq_PO2'

class essm.equations.physics.thermodynamics.**eq_PN2**

Bases: *essm.equations.physics.thermodynamics.eq_Pa*, *essm.equations.physics.thermodynamics.eq_PN2_PO2*

Calculate P_N2 as a function of P_a, P_O2 and P_wa.

expr = Eq(P_N2, (P_a*x_N2 - P_wa*x_N2)/(x_N2 + x_O2))

name = 'eq_PN2'

class essm.equations.physics.thermodynamics.**eq_rhoa**

Bases: *essm.equations.physics.thermodynamics.eq_rhoa_Pwa_Ta*, *essm.equations.physics.thermodynamics.eq_PN2*, *essm.equations.physics.thermodynamics.eq_PO2*

Calculate rho_a from T_a, P_a and P_wa.

expr = Eq(rho_a, (x_N2*(M_N2*P_a - P_wa*(M_N2 - M_w)) + x_O2*(M_O2*P_a - P_wa*(M_O2 - M

name = 'eq_rhoa'

4.1.4 Internals

Core variable type.

class essm.variables._core.**Variable**

Bases: object

Base type for all physical variables.

static check_unit(*expr*)
Check if base dimensions of expression are consistent.

Checks for dimension mismatches of the addends, thus preventing expressions like *meter* + *second* to be created.

static collect_factor_and_basedimension(*expr*)
Return tuple with factor expression and dimension expression.

static get_dimensional_expr(*expr*)
Return dimensions of expression.

Core equation type. Contains class definitions related to equations.

class essm.equations._core.**Equation**
Bases: object

Base type for all equations.

classmethod args()
Return equation arguments from registry if exist.

class essm.equations._core.**EquationMeta**
Bases: essm.bases.RegistryType

Equation interface.

Defines an equation with a docstring and internal variables, if needed.

Example:

```
from ..variables.units import meter, second
class test(Equation):
    '''Test equation.'''

    class d(Variable):
        '''Internal variable.'''
        unit = meter

    class t(Variable):
        '''Internal variable.'''
        unit = second

    class v(Variable):
        '''Internal variable.'''
        unit = meter/second

    expr = v == d / t
```

Raises **ValueError** – if the units are inconsistent.

Example:

```
from ..variables.units import meter, second
class test(Equation):
    '''Test equation with inconsistent units.'''

    class d(Variable):
        '''Internal variable.'''
        unit = meter
```

(continues on next page)

(continued from previous page)

```
    class t(Variable):
        '''Internal variable.'''
        unit = second

    class v(Variable):
        '''Internal variable.'''
        unit = meter/second

    expr = v == d * t
```

Since the units of v and d*t are not the same, this returns:

```
ValueError: Invalid expression units: meter/second == meter*second
```

Build and register new variable.

CHAPTER 5

Additional Notes

Notes on how to contribute, legal information and changes are here for the interested.

5.1 Changes

5.1.1 `v1.1`

released 2020-12-20

Features

- **utils:** Enable supplementary_imports in VariableWriter and EquationWriter (PR #84)

5.1.2 `v1.0.1`

released 2020-11-05

Bug Fixes

- **global:** Allow dimensionless variables in functions (PR #94)

5.1.3 `v1.0.0`

released 2020-09-24

Bug Fixes

- **utils:** Update code to work with isort5 (PR #89)

Features

- **global:** Refactor code to work with sympy>=1.6 (PR #90)

5.1.4 v0.4.3

released 2020-06-18

Bug Fixes

- **utils:** Include expr in variable definitions when writing to file (PR #87)

Features

- **documentation:** Add use examples as Jupyter notebooks and integrate in documentation (PR #83)
- **utils:** Enable writers of .py files for re-import of variable and equation definitions (PR #84)

5.1.5 v0.4.2

released 2020-04-28

Bug Fixes

- **utils:** Improve markdown representation of units (PR #79)
- **variables:** Fix generate_metadata_table for selected variables (PR #80)

5.1.6 v0.4.1

released 2019-11-20

Bug Fixes

- **equations:** Improve dimensional testing of equations and substitution (PR #73)
- **equations:** Add support for Integral and Piecewise in Equation PR (PR #76)

Features

- **utils:** subs_eq() for simultaneous substitutions.(PR #75)

5.1.7 v0.3.0

released 2019-04-09

Bug Fixes

- **equations:** improve substitutions with equations (79ac37d)

Features

- **utils:** add definition to metadata table (3ceaa69)

5.1.8 v0.2.0

released 2019-04-04

- global: adapt to Python 3 and Sympy >=1.3
- global: removal of SageMath mentions
- docs: fix latex representation of x_O2 as x_{O2}
- equations: extend replace_variables
- equations: make .subs() on equation return an equality
- units: reverted missing dimension lookup
- variables: behave as Symbols
- variables: better markdown formatting of units
- variables: changes base class to Symbol
- variables: enableddictionaries with symbols in replace_variables
- variables: fix derive_unit for dimensionless expression
- variables: fix latex rendering
- variables: generate_metadata_table with HTML
- variables: include assumptions from cls attribute
- variables: modify derive_unit to work with summations
- variables: remove Dimension deprecation warnings
- variables: remove internal SI and refer to sympy.physics.units.systems.si
- variables: respect unit in variable with expr
- variables: set dimension and scale factor using method
- variables: support dimensionless variable expression
- variables: support replacing variables by their default values

5.1.9 v0.1.0

released 2017-06-29

- Initial public release.

5.2 License

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Lesser General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software.  If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary.  To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.
```

(continues on next page)

(continued from previous page)

```
  The precise terms and conditions for copying, distribution and
modification follow.

                    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License.  The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language.  (Hereinafter, translation is included without limitation in
the term "modification".)  Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

  1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary way, to print or display an
    announcement including an appropriate copyright notice and a
    notice that there is no warranty (or else, saying that you provide
    a warranty) and that users may redistribute the program under
    these conditions, and telling the user how to view a copy of this
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)
```

(continues on next page)

(continued from previous page)

```
These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable.  However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
```

(continues on next page)

(continued from previous page)

```
compelled to copy the source along with the object code.

  4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

  5. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Program or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

  6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

  7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
```

(continues on next page)

(continued from previous page)

```
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded.  In such case, this License incorporates
the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number.  If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation.  If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.

  10. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author
to ask for permission.  For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this.  Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and
of promoting the sharing and reuse of software generally.

                            NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW.  EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.  SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

                       END OF TERMS AND CONDITIONS

             How to Apply These Terms to Your New Programs
```

(continues on next page)

(continued from previous page)

```
  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

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

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.
```

5.3 Authors

- Jiri Kuncar <jiri.kuncar@gmail.com>
- Stan Schymanski <stan.schymanski@env.ethz.ch>

CHAPTER 6

References

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Bibliography

[Har94] Dennis L. Hartmann. *Global physical climatology*. Academic Press, 1994. ISBN 978-0-12-328530-0.

[MU07] John Monteith and Mike Unsworth. *Principles of Environmental Physics*. Academic Press, Oxford, UK, 3rd edition, November 2007. ISBN 0-12-505103-4.

[SO17] S. J. Schymanski and D. Or. Leaf-scale experiments reveal an important omission in the Penman–Monteith equation. *Hydrol. Earth Syst. Sci.*, 21(2):685–706, February 2017. URL: http://www.hydrol-earth-syst-sci.net/21/685/2017/, doi:10.5194/hess-21-685-2017.

Python Module Index

e

Index

A

C

D

E

F

G

H

K

L

M

N

P

Q

R

S

T

U

V

W

X