CoolProp Documentation

Release 4.0.0

Ian Bell

December 28, 2013

Contents

CoolProp is an open-source database of fluid and humid air properties, formulated based on the most accurate formulations in open literature. It has been validated against the most accurate data available from the relevant references.

CHAPTER 1

Downloading CoolProp

1.1 How to get it?

1.1.1 Option 1 (easiest)

Head to https://sourceforge.net/projects/coolprop/files/CoolProp and download the most recent version. Each language has instructions on what you should do. All the files are already compiled and should work out of the box.

1.1.2 Option 1a (for Python users)

Nightly build installers are also available at https://sourceforge.net/projects/coolprop/files/CoolProp/Nightly for a limited subset of python configurations and are updated every night to be current with the main developer's personal codebase.

Warning: Nightly build may break your code, give wacky results, or otherwise. Use at your own risk.

1.1.3 Option 2 (for Python users)

CoolProp is now on PyPI. If you already have cython (version > 0.17) installed and your default compiler is already configured (see below), you can just do:

```
easy_install CoolProp
```

Or if you already have CoolProp installed, you can upgrade it with:

```
easy_install -U CoolProp
```

Or using pip:

```
pip install CoolProp
```

1.1.4 Option 3 (developers and the courageous)

CoolProp is an open-source project, and is actively looking for developers. The project is hosted in a git repository on github at:

```
https://github.com/ibell/coolprop
```

and you can check out the sources by doing:

```
svn checkout https://github.com/ibell/coolprop coolprop-code
```

or if you want to just browse the repository, you can go to https://github.com/ibell/coolprop.

1.2 Compiler Configuration

If you are on OSX or linux/unix, you probably don't have to do anything at all since python will just use the most recent version of gcc.

If you are a windows user and you have installed Visual Studio 2008 (even the free express version works) python will default to this compiler and everything should go just fine. Make sure you do not install the 2010 version since python 2.x versions are compiled with Visual Studio 2008 compiler. Yes I know that is annoying.

The only thing that is a bit tricky if if you have not installed Visual Studio 2008 and instead want to use the MINGW compiler (a windows version of the gcc compiler). It can be installed using the python(x,y) distribution for instance. In that case, if you are running a command line build of CoolProp, you need to do something like:

```
python setup.py build --compiler=mingw32 install
```

Or if you want to use `easy_install`, you need to create a distutils configuration file called `distutils.cfg` located in `c:\\Python27\\lib\\distutils\\distutils.cfg` with the contents:

```
[build]
compiler = mingw32
```

1.3 Uninstall

If you don't want CoolProp anymore, just delete the CoolProp folder in the Lib/site-packages folder for your distribution, as well as the CoolProp .egg file in Lib/site-packages

CHAPTER 2

Examples

The following examples are written in Python to demonstrate some of the functionalities of CoolProp. Similar calling conventions are used in the wrappers for other programming languages, as can be seen in the "other languages" section below:

2.1 Other Languages

2.1.1 Example Code for C++

Code

```
#include "CoolProp.h"
#include "HumidAirProp.h"
#include "CPState.h"
#include <iostream>
#include <stdlib.h>

#pragma GCC diagnostic ignored "-Wwrite-strings" //Ignore char* warnings

int main() {
    double T, h, p, D;

    printf("CoolProp version:\t%s\n",     get_global_param_string("version").c_str());
    printf("CoolProp gitrevision:\t%s\n", get_global_param_string("gitrevision").c_str());
    printf("CoolProp fluids:\t%s\n",      get_global_param_string("FluidsList").c_str());

    printf("\n************ USING EOS ************\n");

    printf("FLUID STATE INDEPENDENT INPUTS\n");
    printf("Critical Density Propane: %f kg/m^3\n", Props1("Propane", "rhocrit"));

    printf("\nTWO PHASE INPUTS (Pressure)\n");
    printf("Density of saturated liquid Propane at 101.325 kPa: %f kg/m^3\n",   Props("D", 'P', 101.3
    printf("Density of saturated vapor R290 at 101.325 kPa:     %f kg/m^3\n",   Props("D", 'P', 101.3

    printf("\nTWO PHASE INPUTS (Temperature)\n");
    printf("Density of saturated liquid Propane at 300 K: %f kg/m^3\n", Props("D", 'T', 300, 'Q', 0,
```

```
printf("Density of saturated vapor R290 at 300 K:     %f kg/m^3\n", Props("D", 'T', 300, 'Q', 1,

printf("\nSINGLE PHASE CYCLE (Propane)\n");
p = Props("P", 'T', 300, 'D', 1, "Propane");
h = Props("H", 'T', 300, 'D', 1, "Propane");
printf("T,D -> P,H : 300,1 -> %f,%f\n", p, h);

T = Props("T", 'P', p, 'H', h, "Propane");
D = Props("D", 'P', p, 'H', h, "Propane");
printf("P,H -> T,D : %f, %f -> %f, %f\n", p, h, T, D);

printf("\n************ USING TTSE **************\n");
enable_TTSE_LUT("Propane");

printf("TWO PHASE INPUTS (Pressure)\n");
printf("Density of saturated liquid Propane at 101.325 kPa: %f kg/m^3\n", Props("D", 'P', 101.325
printf("Density of saturated vapor R290 at 101.325 kPa:     %f kg/m^3\n", Props("D", 'P', 101.325

printf("\nTWO PHASE INPUTS (Temperature)");
printf("Density of saturated liquid Propane at 300 K: %f kg/m^3\n", Props("D", 'T', 300, 'Q', 0,
printf("Density of saturated vapor R290 at 300 K:     %f kg/m^3\n", Props("D", 'T', 300, 'Q', 1,

printf("\nSINGLE PHASE CYCLE (propane)\n");
p = Props("P", 'T', 300, 'D', 1, "Propane");
h = Props("H", 'T', 300, 'D', 1, "Propane");
printf("T,D -> P,H : 300,1 -> %f,%f", p, h);

T = Props("T", 'P', p, 'H', h, "Propane");
D = Props("D", 'P', p, 'H', h, "Propane");
printf("P,H -> T,D : %f, %f -> %f, %f\n", p, h, T, D);

disable_TTSE_LUT("Propane");

try
{
    printf("\n************ USING REFPROP **************\n");
    RPName = get_REFPROPname("Propane");
    printf("TWO PHASE INPUTS (Pressure)");
    printf("Density of saturated liquid Propane at 101.325 kPa: %f kg/m^3\n", Props("D", 'P', 101
    printf("Density of saturated vapor R290 at 101.325 kPa:     %f kg/m^3\n", Props("D", 'P', 101

    printf("\nTWO PHASE INPUTS (Temperature)");
    printf("Density of saturated liquid Propane at 300 K: %f kg/m^3\n", Props("D", 'T', 300, 'Q',
    printf("Density of saturated vapor R290 at 300 K:     %f kg/m^3\n", Props("D", 'T', 300, 'Q',

    printf("\nSINGLE PHASE CYCLE (propane)\n");
    p = Props("P", 'T', 300, 'D', 1, RPName);
    h = Props("H", 'T', 300, 'D', 1, RPName);
    printf("T,D -> P,H : 300,1 -> %f,%f\n", p, h);

    T = Props("T", 'P', p, 'H', h, RPName);
    D = Props("D", 'P', p, 'H', h, RPName);
    printf("P,H -> T,D : %f, %f -> %f, %f\n", p, h, T, D);
}
catch (std::exception &e)
{
    printf("\n************ CANT USE REFPROP ************\n");
}
```

```
    printf("\n************ CHANGE UNIT SYSTEM (default is kSI) ************\n");
    set_standard_unit_system(UNIT_SYSTEM_SI);
    printf("Vapor pressure of water at 373.15 K in SI units (Pa):   %f\n", Props("P", 'T', 373.15, 'Q

    set_standard_unit_system(UNIT_SYSTEM_KSI);
    printf("Vapor pressure of water at 373.15 K in kSI units (kPa): %f\n", Props("P", 'T', 373.15, 'Q

    printf("\n************ BRINES AND SECONDARY WORKING FLUIDS ************\n");
    printf("Density of 50%% (mass) ethylene glycol/water at 300 K, 101.325 kPa: %f kg/m^3\n", Props("
    printf("Viscosity of Therminol D12 at 350 K, 101.325 kPa: %f Pa-s\n",  Props("V", 'T', 350, 'P',

    printf("\n************ HUMID AIR PROPERTIES ************\n");
    printf("Humidity ratio of 50%% rel. hum. air at 300 K, 101.325 kPa: %f kg_w/kg_da\n", HAProps("W"
    printf("Relative humidity from last calculation: %f (fractional)\n",   HAProps("R", "T", 300, "P"
    return 0;
}
```

Output

```
CoolProp version:        4.0.0
CoolProp gitrevision:        b'aa5be70f193f12056808e4abb620f261133fca6d'
CoolProp fluids:        Water,R134a,Helium,Oxygen,Hydrogen,ParaHydrogen,OrthoHydrogen,Argon,CarbonDio

************ USING EOS *************
FLUID STATE INDEPENDENT INPUTS
Critical Density Propane: 220.478100 kg/m^3

TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.882952 kg/m^3
Density of saturated vapor R290 at 101.325 kPa:     2.416136 kg/m^3

TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.447375 kg/m^3
Density of saturated vapor R290 at 300 K:     21.629532 kg/m^3

SINGLE PHASE CYCLE (Propane)
T,D -> P,H : 300,1 -> 56.072763,634.733626
P,H -> T,D : 56.072763, 634.733626 -> 300.000000, 1.000000

************ USING TTSE ***************
TWO PHASE INPUTS (Pressure)
0.103 to build both two phase tables
2.07 to build single phase table with p,h
7.286 to build single phase table for T,rho
write time: 0.039
Density of saturated liquid Propane at 101.325 kPa: 580.882952 kg/m^3
Density of saturated vapor R290 at 101.325 kPa:     2.416136 kg/m^3

TWO PHASE INPUTS (Temperature)Density of saturated liquid Propane at 300 K: 489.447377 kg/m^3
Density of saturated vapor R290 at 300 K:     21.629532 kg/m^3

SINGLE PHASE CYCLE (propane)
T,D -> P,H : 300,1 -> 56.072764,634.733626P,H -> T,D : 56.072764, 634.733626 -> 300.000000, 1.000000

************ USING REFPROP ***************
TWO PHASE INPUTS (Pressure)Density of saturated liquid Propane at 101.325 kPa: 580.882952 kg/m^3
```

```
Density of saturated vapor R290 at 101.325 kPa:     2.416136 kg/m^3

TWO PHASE INPUTS (Temperature)Density of saturated liquid Propane at 300 K: 489.447375 kg/m^3
Density of saturated vapor R290 at 300 K:     21.629532 kg/m^3

SINGLE PHASE CYCLE (propane)
T,D -> P,H : 300,1 -> 56.072763,634.733626
P,H -> T,D : 56.072763, 634.733626 -> 300.000000, 1.000000

************ CHANGE UNIT SYSTEM (default is kSI) *************
Vapor pressure of water at 373.15 K in SI units (Pa):   101417.996660
Vapor pressure of water at 373.15 K in kSI units (kPa): 101.417997

************ BRINES AND SECONDARY WORKING FLUIDS *************
Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: 1061.179308 kg/m^3
Viscosity of Therminol D12 at 350 K, 101.325 kPa: 0.000523 Pa-s

************ HUMID AIR PROPERTIES *************
Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: 0.011096 kg_w/kg_da
Relative humidity from last calculation: 0.500000 (fractional)
```

2.1.2 Example Code for MATLAB

Code

```
% Example of CoolProp for MATLAB
% Ian Bell, 2013

disp(['CoolProp version: ', Props('version')])
disp(['CoolProp gitrevision: ', Props('gitrevision')])
disp(['CoolProp fluids: ', Props('FluidsList')])

disp(' ')
disp('************ USING EOS *************')
disp(' ')
disp('FLUID STATE INDEPENDENT INPUTS')
disp(['Critical Density Propane: ', num2str(Props('Propane','rhocrit')), ' kg/m^3'])
disp(['TWO PHASE INPUTS (Pressure)'])
disp(['Density of saturated liquid Propane at 101.325 kPa: ', num2str(Props('D','P',101.325,'Q',0,'P
disp(['Density of saturated vapor R290 at 101.325 kPa: ', num2str(Props('D','P',101.325,'Q',1,'R290')
disp(['TWO PHASE INPUTS (Temperature)'])
disp(['Density of saturated liquid Propane at 300 K: ', num2str(Props('D','T',300,'Q',0,'Propane')),
disp(['Density of saturated vapor R290 at 300 K: ', num2str(Props('D','T',300,'Q',1,'R290')), ' kg/m
disp(['SINGLE PHASE CYCLE (propane)'])
p = Props('P','T',300,'D',1,'Propane');
h = Props('H','T',300,'D',1,'Propane');
disp(['T,D -> P,H ', num2str(300),',',num2str(1), ' --> ',num2str(p),',',num2str(h)])
T = Props('T','P',p,'H',h,'Propane');
D = Props('D','P',p,'H',h,'Propane');
disp(['P,H -> T,D', num2str(p),',',num2str(h),'-->',num2str(T),',',num2str(D)])

disp([' '])
disp(['************ USING TTSE **************'])
disp([' '])
Props('Propane','enable_TTSE')
disp(['TWO PHASE INPUTS (Pressure)'])
```

```
disp(['Density of saturated liquid Propane at 101.325 kPa: ', num2str(Props('D','P',101.325,'Q',0,'Pr
disp(['Density of saturated vapor R290 at 101.325 kPa: ', num2str(Props('D','P',101.325,'Q',1,'R290')
disp(['TWO PHASE INPUTS (Temperature)'])
disp(['Density of saturated liquid Propane at 300 K: ', num2str(Props('D','T',300,'Q',0,'Propane')),
disp(['Density of saturated vapor R290 at 300 K: ', num2str(Props('D','T',300,'Q',1,'R290')), ' kg/m^
disp(['SINGLE PHASE CYCLE (propane)'])
p = Props('P','T',300,'D',1,'Propane');
h = Props('H','T',300,'D',1,'Propane');
disp(['T,D -> P,H ', num2str(300),',',num2str(1), ' --> ',num2str(p),',',num2str(h)])
T = Props('T','P',p,'H',h,'Propane');
D = Props('D','P',p,'H',h,'Propane');
disp(['P,H -> T,D ', num2str(p),',',num2str(h),' --> ',num2str(T),',',num2str(D)])
Props('Propane','disable_TTSE')

try
    disp(' ')
    disp('************ USING REFPROP **************')
    disp(' ')
    disp('FLUID STATE INDEPENDENT INPUTS')
    disp(['Critical Density Propane:', num2str(Props('REFPROP-Propane','rhocrit')), 'kg/m^3'])
    disp(['TWO PHASE INPUTS (Pressure)'])
    disp(['Density of saturated liquid Propane at 101.325 kPa: ', num2str(Props('D','P',101.325,'Q',0
    disp(['Density of saturated vapor R290 at 101.325 kPa: ', num2str(Props('D','P',101.325,'Q',1,'RE
    disp(['TWO PHASE INPUTS (Temperature)'])
    disp(['Density of saturated liquid Propane at 300 K: ', num2str(Props('D','T',300,'Q',0,'REFPROP-
    disp(['Density of saturated vapor R290 at 300 K: ', num2str(Props('D','T',300,'Q',1,'REFPROP-Prop
    disp(['SINGLE PHASE CYCLE (propane)'])
    p = Props('P','T',300,'D',1,'REFPROP-Propane');
    h = Props('H','T',300,'D',1,'REFPROP-Propane');
    disp(['T,D -> P,H ', num2str(300),',',num2str(1), ' --> ',num2str(p),',',num2str(h)])
    T = Props('T','P',p,'H',h,'REFPROP-Propane');
    D = Props('D','P',p,'H',h,'REFPROP-Propane');
    disp(['P,H -> T,D ', num2str(p),',',num2str(h),' --> ',num2str(T),',',num2str(D)])
catch
    disp(' ')
    disp('************ CANT USE REFPROP ************')
    disp(' ')
end

disp(' ')
disp('************ CHANGE UNIT SYSTEM (default is kSI) *************')
disp(' ')
Props('set_UNIT_SYSTEM_SI')
disp(['Vapor pressure of water at 373.15 K in SI units (Pa): ', num2str(Props('P','T',373.15,'Q',0,'W
Props('set_UNIT_SYSTEM_KSI')
disp(['Vapor pressure of water at 373.15 K in kSI units (kPa): ', num2str(Props('P','T',373.15,'Q',0,

disp(' ')
disp('************ BRINES AND SECONDARY WORKING FLUIDS *************')
disp(' ')
disp(['Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: ', num2str(Props('D','T',30
disp(['Viscosity of Therminol D12 at 350 K, 101.325 kPa: ', num2str(Props('V', 'T', 350, 'P', 101.325

disp(' ')
disp('************ HUMID AIR PROPERTIES *************')
disp(' ')
disp(['Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: ', num2str(HAProps('W','T',300,'P',
disp(['Relative humidity from last calculation: ', num2str(HAProps('R','T',300,'P',101.325,'W',HAProp
```

Output

```
  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

CoolProp version: 4.0.0
CoolProp gitrevision: b'4c5efbe7c47364771541575b5de18d28bcb4efa6'
CoolProp fluids: Water,R134a,Helium,Oxygen,Hydrogen,ParaHydrogen,OrthoHydrogen,Argon,CarbonDioxide,Ni

************ USING EOS *************

FLUID STATE INDEPENDENT INPUTS
Critical Density Propane: 220.4781 kg/m^3
TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.883 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.4161 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.4474 kg/m^3
Density of saturated vapor R290 at 300 K: 21.6295 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.0728,634.7336
P,H -> T,D56.0728,634.7336-->300,1

************ USING TTSE ***************

TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.883 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.4161 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.4474 kg/m^3
Density of saturated vapor R290 at 300 K: 21.6295 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.0728,634.7336
P,H -> T,D 56.0728,634.7336 --> 300,1

************ USING REFPROP ***************

FLUID STATE INDEPENDENT INPUTS
Critical Density Propane:220.4781kg/m^3
TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.883 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.4161 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.4474 kg/m^3
Density of saturated vapor R290 at 300 K: 21.6295 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.0728,634.7336
P,H -> T,D 56.0728,634.7336 --> 300,1

************ CHANGE UNIT SYSTEM (default is kSI) *************

Unit system set to SI
Vapor pressure of water at 373.15 K in SI units (Pa): 101417.9967
Unit system set to KSI
Vapor pressure of water at 373.15 K in kSI units (kPa): 101.418

************ BRINES AND SECONDARY WORKING FLUIDS *************
```

```
Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: 1061.1793kg/m^3
Viscosity of Therminol D12 at 350 K, 101.325 kPa: 0.00052288Pa-s

************ HUMID AIR PROPERTIES *************

Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: 0.011096 kg_w/kg_da
Relative humidity from last calculation: 0.5 (fractional)
```

2.1.3 Example Code for Octave

Code

```
% Example of CoolProp for Octave
% Ian Bell, 2013

CoolProp
disp(['CoolProp version: ', CoolProp.get_global_param_string('version')])
disp(['CoolProp gitrevision: ', CoolProp.get_global_param_string('gitrevision')])
disp(['CoolProp fluids: ', CoolProp.get_global_param_string('FluidsList')])

disp(' ')
disp('************ USING EOS *************')
disp(' ')
disp('FLUID STATE INDEPENDENT INPUTS')
disp(['Critical Density Propane: ', num2str(CoolProp.Props1('Propane','rhocrit')), ' kg/m^3'])
disp(['TWO PHASE INPUTS (Pressure)'])
disp(['Density of saturated liquid Propane at 101.325 kPa: ', num2str(CoolProp.Props('D','P',101.325,
disp(['Density of saturated vapor R290 at 101.325 kPa: ', num2str(CoolProp.Props('D','P',101.325,'Q',
disp(['TWO PHASE INPUTS (Temperature)'])
disp(['Density of saturated liquid Propane at 300 K: ', num2str(CoolProp.Props('D','T',300,'Q',0,'Pro
disp(['Density of saturated vapor R290 at 300 K: ', num2str(CoolProp.Props('D','T',300,'Q',1,'R290'))
disp(['SINGLE PHASE CYCLE (propane)'])
p = CoolProp.Props('P','T',300,'D',1,'Propane');
h = CoolProp.Props('H','T',300,'D',1,'Propane');
disp(['T,D -> P,H ', num2str(300),',',num2str(1), ' --> ',num2str(p),',',num2str(h)])
T = CoolProp.Props('T','P',p,'H',h,'Propane');
D = CoolProp.Props('D','P',p,'H',h,'Propane');
disp(['P,H -> T,D', num2str(p),',',num2str(h),'-->',num2str(T),',',num2str(D)])

disp([' '])
disp(['************ USING TTSE **************'])
disp([' '])
CoolProp.enable_TTSE_LUT('Propane');
disp(['TWO PHASE INPUTS (Pressure)'])
disp(['Density of saturated liquid Propane at 101.325 kPa: ', num2str(CoolProp.Props('D','P',101.325,
disp(['Density of saturated vapor R290 at 101.325 kPa: ', num2str(CoolProp.Props('D','P',101.325,'Q',
disp(['TWO PHASE INPUTS (Temperature)'])
disp(['Density of saturated liquid Propane at 300 K: ', num2str(CoolProp.Props('D','T',300,'Q',0,'Pro
disp(['Density of saturated vapor R290 at 300 K: ', num2str(CoolProp.Props('D','T',300,'Q',1,'R290'))
disp(['SINGLE PHASE CYCLE (propane)'])
p = CoolProp.Props('P','T',300,'D',1,'Propane');
h = CoolProp.Props('H','T',300,'D',1,'Propane');
disp(['T,D -> P,H ', num2str(300),',',num2str(1), ' --> ',num2str(p),',',num2str(h)])
T = CoolProp.Props('T','P',p,'H',h,'Propane');
D = CoolProp.Props('D','P',p,'H',h,'Propane');
disp(['P,H -> T,D ', num2str(p),',',num2str(h),' --> ',num2str(T),',',num2str(D)])
```

```
CoolProp.disable_TTSE_LUT('Propane');

try
    disp(' ')
    disp('************ USING REFPROP **************')
    disp(' ')
    disp('FLUID STATE INDEPENDENT INPUTS')
    disp(['Critical Density Propane:', num2str(CoolProp.Props('REFPROP-Propane','rhocrit')), 'kg/m^3'
    disp(['TWO PHASE INPUTS (Pressure)'])
    disp(['Density of saturated liquid Propane at 101.325 kPa: ', num2str(CoolProp.Props('D','P',101
    disp(['Density of saturated vapor R290 at 101.325 kPa: ', num2str(CoolProp.Props('D','P',101.325,
    disp(['TWO PHASE INPUTS (Temperature)'])
    disp(['Density of saturated liquid Propane at 300 K: ', num2str(CoolProp.Props('D','T',300,'Q',0,
    disp(['Density of saturated vapor R290 at 300 K: ', num2str(CoolProp.Props('D','T',300,'Q',1,'REF
    disp(['SINGLE PHASE CYCLE (propane)'])
    p = CoolProp.Props('P','T',300,'D',1,'REFPROP-Propane');
    h = CoolProp.Props('H','T',300,'D',1,'REFPROP-Propane');
    disp(['T,D -> P,H ', num2str(300),',',num2str(1), ' --> ',num2str(p),',',num2str(h)])
    T = CoolProp.Props('T','P',p,'H',h,'REFPROP-Propane');
    D = CoolProp.Props('D','P',p,'H',h,'REFPROP-Propane');
    disp(['P,H -> T,D ', num2str(p),',',num2str(h),' --> ',num2str(T),',',num2str(D)])
catch
    disp(' ')
    disp('************ CANT USE REFPROP ************')
    disp(' ')
end

 disp([' '])
 disp('************ CHANGE UNIT SYSTEM (default is kSI) ************')
 disp(' ')
 CoolProp.set_standard_unit_system(CoolProp.UNIT_SYSTEM_SI)
 disp(['Vapor pressure of water at 373.15 K in SI units (Pa):',num2str(CoolProp.Props('P','T',373.15,
 CoolProp.set_standard_unit_system(CoolProp.UNIT_SYSTEM_KSI)
 disp(['Vapor pressure of water at 373.15 K in kSI units (kPa):',num2str(CoolProp.Props('P','T',373.1

disp(' ')
disp('************ BRINES AND SECONDARY WORKING FLUIDS ************')
disp(' ')
disp(['Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: ', num2str(CoolProp.Props('
disp(['Viscosity of Therminol D12 at 350 K, 101.325 kPa: ', num2str(CoolProp.Props('V', 'T', 350, 'P'

disp(' ')
disp('************ HUMID AIR PROPERTIES ************')
disp(' ')
disp(['Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: ', num2str(CoolProp.HAProps('W','T'
disp(['Relative humidity from last calculation: ', num2str(CoolProp.HAProps('R','T',300,'P',101.325,'
```

Output

```
GNU Octave, version 3.6.1
Copyright (C) 2012 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "i686-pc-mingw32".
```

```
Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.

For information about changes from previous versions, type 'news'.

 - Use 'pkg list' to see a list of installed packages.
 - MSYS shell available (C:\Program Files (x86)\Octave-3.6.1\msys).
 - Graphics backend: qt.

CoolProp version: 4.0.0
CoolProp gitrevision: b'aa5be70f193f12056808e4abb620f261133fca6d'
CoolProp fluids: Water,R134a,Helium,Oxygen,Hydrogen,ParaHydrogen,OrthoHydrogen,Argon,CarbonDioxide,Ni

************ USING EOS *************

FLUID STATE INDEPENDENT INPUTS
Critical Density Propane: 220.48 kg/m^3
TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.88 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.4161 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.45 kg/m^3
Density of saturated vapor R290 at 300 K: 21.63 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.073,634.73
P,H -> T,D56.073,634.73-->300,1

************ USING TTSE ***************

TWO PHASE INPUTS (Pressure)
0.081 to build both two phase tables
Density of saturated liquid Propane at 101.325 kPa: 580.88 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.4161 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.45 kg/m^3
Density of saturated vapor R290 at 300 K: 21.63 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.073,634.73
P,H -> T,D 56.073,634.73 --> 300,1

************ USING REFPROP ***************

FLUID STATE INDEPENDENT INPUTS

************ CANT USE REFPROP ************


************ CHANGE UNIT SYSTEM (default is kSI) *************

Vapor pressure of water at 373.15 K in SI units (Pa):1.0142e+005
Vapor pressure of water at 373.15 K in kSI units (kPa):101.42

************ BRINES AND SECONDARY WORKING FLUIDS *************
```

```
Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: 1061.2kg/m^3
Viscosity of Therminol D12 at 350 K, 101.325 kPa: 0.00052288Pa-s

************ HUMID AIR PROPERTIES *************

Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: 0.011096 kg_w/kg_da
Relative humidity from last calculation: 0.5(fractional)
```

2.1.4 Example Code for C#

Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            double T, h, p, D;
            Console.Write("CoolProp version: " + CoolProp.get_global_param_string("version") + "\n");
            Console.Write("CoolProp gitrevision: " + CoolProp.get_global_param_string("gitrevision")
            Console.Write("CoolProp fluids: " + CoolProp.get_global_param_string("FluidsList") + "\n

            Console.Write(" " + "\n");
            Console.Write("************ USING EOS *************" + "\n");
            Console.Write(" " + "\n");
            Console.Write("FLUID STATE INDEPENDENT INPUTS" + "\n");
            Console.Write("Critical Density Propane: " + CoolProp.Props1("Propane", "rhocrit") + "kg/
            Console.Write("TWO PHASE INPUTS (Pressure)" + "\n");
            Console.Write("Density of saturated liquid Propane at 101.325 kPa: " + CoolProp.Props("D"
            Console.Write("Density of saturated vapor R290 at 101.325 kPa: " + CoolProp.Props("D", 'P
            Console.Write("TWO PHASE INPUTS (Temperature)" + "\n");
            Console.Write("Density of saturated liquid Propane at 300 K: " + CoolProp.Props("D", 'T',
            Console.Write("Density of saturated vapor R290 at 300 K: " + CoolProp.Props("D", 'T', 300
            Console.Write("SINGLE PHASE CYCLE (propane)" + "\n");
            p = CoolProp.Props("P", 'T', 300, 'D', 1, "Propane");
            h = CoolProp.Props("H", 'T', 300, 'D', 1, "Propane");
            Console.Write("T,D -> P,H " + 300 + "," + 1 + " --> " + p + ',' + h + "\n");
            T = CoolProp.Props("T", 'P', p, 'H', h, "Propane");
            D = CoolProp.Props("D", 'P', p, 'H', h, "Propane");
            Console.Write("P,H -> T,D " + p + ',' + h + " --> " + T + ',' + D + "\n");

            Console.Write(" " + "\n");
            Console.Write("************ USING TTSE ***************" + "\n");
            Console.Write(" " + "\n");
            CoolProp.enable_TTSE_LUT("Propane");
            Console.Write("TWO PHASE INPUTS (Pressure)" + "\n");
            Console.Write("Density of saturated liquid Propane at 101.325 kPa: " + CoolProp.Props("D"
            Console.Write("Density of saturated vapor R290 at 101.325 kPa: " + CoolProp.Props("D", 'P
            Console.Write("TWO PHASE INPUTS (Temperature)" + "\n");
            Console.Write("Density of saturated liquid Propane at 300 K: " + CoolProp.Props("D", 'T',
```

```
Console.Write("Density of saturated vapor R290 at 300 K: " + CoolProp.Props("D", 'T', 300
Console.Write("SINGLE PHASE CYCLE (propane)" + "\n");
p = CoolProp.Props("P", 'T', 300, 'D', 1, "Propane");
h = CoolProp.Props("H", 'T', 300, 'D', 1, "Propane");
Console.Write("T,D -> P,H " + 300 + ","+ 1+ " --> " + p + ',' + h + "\n");
T = CoolProp.Props("T", 'P', p, 'H', h, "Propane");
D = CoolProp.Props("D", 'P', p, 'H', h, "Propane");
Console.Write("P,H -> T,D " + p + ',' + h + " --> " + T + ',' + D + "\n");
CoolProp.disable_TTSE_LUT("Propane");

try
{
    Console.Write(" " + "\n");
    Console.Write("************ USING REFPROP ***************" + "\n");
    Console.Write(" " + "\n");
    Console.Write("TWO PHASE INPUTS (Pressure)" + "\n");
    Console.Write("Density of saturated liquid Propane at 101.325 kPa: " + CoolProp.Props
    Console.Write("Density of saturated vapor R290 at 101.325 kPa: " + CoolProp.Props("D"
    Console.Write("TWO PHASE INPUTS (Temperature)" + "\n");
    Console.Write("Density of saturated liquid Propane at 300 K: " + CoolProp.Props("D",
    Console.Write("Density of saturated vapor R290 at 300 K: " + CoolProp.Props("D", 'T',
    Console.Write("SINGLE PHASE CYCLE (propane)" + "\n");
    p = CoolProp.Props("P",'T',300,'D',1,"Propane");
    h = CoolProp.Props("H",'T',300,'D',1,"Propane");
    Console.Write("T,D -> P,H " + 300 + "," + 1 + " --> " + p + ',' + h + "\n");
    T = CoolProp.Props("T",'P',p,'H',h,"Propane");
    D = CoolProp.Props("D",'P',p,'H',h,"Propane");
    Console.Write("P,H -> T,D " + p + ',' + h + " --> " + T + ',' + D + "\n");
}
catch
{
    Console.Write(" " + "\n");
    Console.Write("************ CANT USE REFPROP ************" + "\n");
    Console.Write(" " + "\n");
}

Console.Write(" " + "\n");
Console.Write("************ CHANGE UNIT SYSTEM (default is kSI) *************" + "\n");
Console.Write(" " + "\n");
int SI = (int)unit_systems.UNIT_SYSTEM_SI;
int kSI = (int)unit_systems.UNIT_SYSTEM_KSI;
CoolProp.set_standard_unit_system(SI);
Console.Write("Vapor pressure of water at 373.15 K in SI units (Pa): " + CoolProp.Props("
CoolProp.set_standard_unit_system(kSI);
Console.Write("Vapor pressure of water at 373.15 K in kSI units (kPa): " + CoolProp.Props

Console.Write(" " + "\n");
Console.Write("************ BRINES AND SECONDARY WORKING FLUIDS *************" + "\n");
Console.Write(" " + "\n");
Console.Write("Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: " + Coo
Console.Write("Viscosity of Therminol D12 at 350 K, 101.325 kPa: " + CoolProp.Props("V",

Console.Write(" " + "\n");
Console.Write("************ HUMID AIR PROPERTIES *************" + "\n");
Console.Write(" " + "\n");
Console.Write("Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: " + CoolProp.HA
Console.Write("Relative humidity from last calculation: " + CoolProp.HAProps("R", "T", 30
```

```
            //Console.Write("Enter to quit");
            //Console.ReadLine();
        }
    }
}
```

Output

```
CoolProp version: 4.0.0
CoolProp gitrevision: b'aa5be70f193f12056808e4abb620f261133fca6d'
CoolProp fluids: Water,R134a,Helium,Oxygen,Hydrogen,ParaHydrogen,OrthoHydrogen,Argon,CarbonDioxide,Ni

************ USING EOS *************

FLUID STATE INDEPENDENT INPUTS
Critical Density Propane: 220.4781kg/m^3
TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.882951954822 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.41613600878819 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.447375251959 kg/m^3
Density of saturated vapor R290 at 300 K: 21.6295320184622 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.0727627482929,634.733625928477
P,H -> T,D 56.0727627482929,634.733625928477 --> 300,0.999999999999999

************ USING TTSE ***************

TWO PHASE INPUTS (Pressure)
0.12 to build both two phase tables
2.291 to build single phase table with p,h
11.046 to build single phase table for T,rho
write time: 0.035
Density of saturated liquid Propane at 101.325 kPa: 580.882952265691 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.41613600655146 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.447377441148 kg/m^3
Density of saturated vapor R290 at 300 K: 21.6295318773021 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.0727644058219,634.733625854333
P,H -> T,D 56.0727644058219,634.733625854333 --> 299.999999906753,0.999999979733272

************ USING REFPROP ***************

TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.882951954822 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.41613600878819 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.447375251959 kg/m^3
Density of saturated vapor R290 at 300 K: 21.6295320184622 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.0727627482929,634.733625928477
P,H -> T,D 56.0727627482929,634.733625928477 --> 300,0.999999999999999

************ CHANGE UNIT SYSTEM (default is kSI) *************

Vapor pressure of water at 373.15 K in SI units (Pa): 101417.996659952
```

```
Vapor pressure of water at 373.15 K in kSI units (kPa): 101.417996659952

************ BRINES AND SECONDARY WORKING FLUIDS *************

Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: 1061.17930772046kg/m^3
Viscosity of Therminol D12 at 350 K, 101.325 kPa: 0.000522884941050795Pa-s

************ HUMID AIR PROPERTIES *************

Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: 0.011096223750095 kg_w/kg_da
Relative humidity from last calculation: 0.5(fractional)
```

2.1.5 Example Code for Java

Code

```
// Example for Java
// Ian Bell, 2013

public class Example {
    static {
        System.loadLibrary("CoolProp");
    }

    public static void main(String argv[]){

        double T, h, p, D;
        int SI,KSI;
        System.out.println("CoolProp version: " + CoolProp.get_global_param_string("version"));
        System.out.println("CoolProp gitrevision: " + CoolProp.get_global_param_string("gitrevision")
        System.out.println("CoolProp fluids: " + CoolProp.get_global_param_string("FluidsList"));

        System.out.println(" ");
        System.out.println("************ USING EOS *************");
        System.out.println(" ");
        System.out.println("FLUID STATE INDEPENDENT INPUTS");
        System.out.println("Critical Density Propane: " + CoolProp.Props1("Propane", "rhocrit") + "kg
        System.out.println("TWO PHASE INPUTS (Pressure)");
        System.out.println("Density of saturated liquid Propane at 101.325 kPa: " + CoolProp.Props("D
        System.out.println("Density of saturated vapor R290 at 101.325 kPa: " + CoolProp.Props("D", '
        System.out.println("TWO PHASE INPUTS (Temperature)");
        System.out.println("Density of saturated liquid Propane at 300 K: " + CoolProp.Props("D", 'T'
        System.out.println("Density of saturated vapor R290 at 300 K: " + CoolProp.Props("D", 'T', 30
        System.out.println("SINGLE PHASE CYCLE (propane)");
        p = CoolProp.Props("P", 'T', 300, 'D', 1, "Propane");
        h = CoolProp.Props("H", 'T', 300, 'D', 1, "Propane");
        System.out.println("T,D -> P,H " + 300 + "," + 1 + " --> " + p + ',' + h);
        T = CoolProp.Props("T", 'P', p, 'H', h, "Propane");
        D = CoolProp.Props("D", 'P', p, 'H', h, "Propane");
        System.out.println("P,H -> T,D " + p + ',' + h + " --> " + T + ',' + D);

        System.out.println(" ");
        System.out.println("************ USING TTSE **************");
        System.out.println(" ");
        CoolProp.enable_TTSE_LUT("Propane");
        System.out.println("TWO PHASE INPUTS (Pressure)");
```

```
System.out.println("Density of saturated liquid Propane at 101.325 kPa: " + CoolProp.Props("D
System.out.println("Density of saturated vapor R290 at 101.325 kPa: " + CoolProp.Props("D", '
System.out.println("TWO PHASE INPUTS (Temperature)");
System.out.println("Density of saturated liquid Propane at 300 K: " + CoolProp.Props("D", 'T'
System.out.println("Density of saturated vapor R290 at 300 K: " + CoolProp.Props("D", 'T', 30
System.out.println("SINGLE PHASE CYCLE (propane)");
p = CoolProp.Props("P", 'T', 300, 'D', 1, "Propane");
h = CoolProp.Props("H", 'T', 300, 'D', 1, "Propane");
System.out.println("T,D -> P,H " + 300 + ","+ 1+ " --> " + p + ',' + h);
T = CoolProp.Props("T", 'P', p, 'H', h, "Propane");
D = CoolProp.Props("D", 'P', p, 'H', h, "Propane");
System.out.println("P,H -> T,D " + p + ',' + h + " --> " + T + ',' + D);
CoolProp.disable_TTSE_LUT("Propane");

try
{
    System.out.println(" ");
    System.out.println("************ USING REFPROP ***************");
    System.out.println(" ");
    System.out.println("TWO PHASE INPUTS (Pressure)");
    System.out.println("Density of saturated liquid Propane at 101.325 kPa: " + CoolProp.Prop
    System.out.println("Density of saturated vapor R290 at 101.325 kPa: " + CoolProp.Props("D
    System.out.println("TWO PHASE INPUTS (Temperature)");
    System.out.println("Density of saturated liquid Propane at 300 K: " + CoolProp.Props("D",
    System.out.println("Density of saturated vapor R290 at 300 K: " + CoolProp.Props("D", 'T'
    System.out.println("SINGLE PHASE CYCLE (propane)");
    p = CoolProp.Props("P",'T',300,'D',1,"Propane");
    h = CoolProp.Props("H",'T',300,'D',1,"Propane");
    System.out.println("T,D -> P,H " + 300 + "," + 1 + " --> " + p + ',' + h);
    T = CoolProp.Props("T",'P',p,'H',h,"Propane");
    D = CoolProp.Props("D",'P',p,'H',h,"Propane");
    System.out.println("P,H -> T,D " + p + ',' + h + " --> " + T + ',' + D);
}
catch(Exception e)
{
    System.out.println(" ");
    System.out.println("************ CANT USE REFPROP ************");
    System.out.println(" ");
}

System.out.println(" ");
System.out.println("************ CHANGE UNIT SYSTEM (default is kSI) *************");
System.out.println(" ");
CoolProp.set_standard_unit_system(unit_systems.UNIT_SYSTEM_SI.swigValue());
System.out.println("Vapor pressure of water at 373.15 K in SI units (Pa): " + CoolProp.Props
CoolProp.set_standard_unit_system(unit_systems.UNIT_SYSTEM_KSI.swigValue());
System.out.println("Vapor pressure of water at 373.15 K in kSI units (kPa): " + CoolProp.Prop

System.out.println(" ");
System.out.println("************ BRINES AND SECONDARY WORKING FLUIDS *************");
System.out.println(" ");
System.out.println("Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: " + Co
System.out.println("Viscosity of Therminol D12 at 350 K, 101.325 kPa: " + CoolProp.Props("V",

System.out.println(" ");
System.out.println("************ HUMID AIR PROPERTIES *************");
System.out.println(" ");
System.out.println("Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: " + CoolProp.P
```

```
        System.out.println("Relative humidity from last calculation: " + CoolProp.HAProps("R", "T", 
    }
}
```

Output

```
CoolProp version: 4.0.0
CoolProp gitrevision: b'aa5be70f193f12056808e4abb620f261133fca6d'
CoolProp fluids: Water,R134a,Helium,Oxygen,Hydrogen,ParaHydrogen,OrthoHydrogen,Argon,CarbonDioxide,Ni

************ USING EOS *************

FLUID STATE INDEPENDENT INPUTS
Critical Density Propane: 220.4781kg/m^3
TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.8829519548221 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.416136008788186 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.44737525195865 kg/m^3
Density of saturated vapor R290 at 300 K: 21.62953201846219 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.07276274829289,634.7336259284773
P,H -> T,D 56.07276274829289,634.7336259284773 --> 300.0000000000003,0.999999999999999

************ USING TTSE ***************

TWO PHASE INPUTS (Pressure)
0.087 to build both two phase tables
Density of saturated liquid Propane at 101.325 kPa: 580.8829522656912 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.416136006551463 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.4473774411483 kg/m^3
Density of saturated vapor R290 at 300 K: 21.629531877302085 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.07276440582186,634.7336258543331
P,H -> T,D 56.07276440582186,634.7336258543331 --> 299.9999999067525,0.9999999797332724

************ USING REFPROP ***************

TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.8829519548221 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.416136008788186 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.44737525195865 kg/m^3
Density of saturated vapor R290 at 300 K: 21.62953201846219 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300,1 --> 56.07276274829289,634.7336259284773
P,H -> T,D 56.07276274829289,634.7336259284773 --> 300.0000000000003,0.999999999999999

************ CHANGE UNIT SYSTEM (default is kSI) *************

Vapor pressure of water at 373.15 K in SI units (Pa): 101417.99665995208
Vapor pressure of water at 373.15 K in kSI units (kPa): 101.41799665995208

************ BRINES AND SECONDARY WORKING FLUIDS *************

Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: 1061.1793077204613kg/m^3
```

```
Viscosity of Therminol D12 at 350 K, 101.325 kPa: 5.228849410507948E-4Pa-s

************ HUMID AIR PROPERTIES *************

Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: 0.011096223750095016 kg_w/kg_da
Relative humidity from last calculation: 0.49999999999999994(fractional)
```

2.1.6 Example Code for Python

Code

```
# Example of CoolProp for Python
# Ian Bell, 2013

from __future__ import print_function

import CoolProp
import CoolProp.CoolProp as CP

print('CoolProp version: ', CoolProp.__version__)
print('CoolProp gitrevision: ', CoolProp.__gitrevision__)
print('CoolProp fluids: ', CoolProp.__fluids__)


print(' ')
print('************ USING EOS *************')
print(' ')
print('FLUID STATE INDEPENDENT INPUTS')
print('Critical Density Propane:', CP.Props('Propane', 'rhocrit'), 'kg/m^3')
print('TWO PHASE INPUTS (Pressure)')
print('Density of saturated liquid Propane at 101.325 kPa:',
       CP.Props('D', 'P', 101.325, 'Q', 0, 'Propane'), 'kg/m^3')
print('Density of saturated vapor R290 at 101.325 kPa:',
       CP.Props('D', 'P', 101.325, 'Q', 1, 'R290'), 'kg/m^3')
print('TWO PHASE INPUTS (Temperature)')
print('Density of saturated liquid Propane at 300 K:',
       CP.Props('D', 'T', 300, 'Q', 0, 'Propane'), 'kg/m^3')
print('Density of saturated vapor R290 at 300 K:',
       CP.Props('D', 'T', 300, 'Q', 1, 'R290'), 'kg/m^3')

p = CP.Props('P', 'T', 300, 'D', 1, 'Propane')
h = CP.Props('H', 'T', 300, 'D', 1, 'Propane')
T = CP.Props('T', 'P', p, 'H', h, 'Propane')
D = CP.Props('D', 'P', p, 'H', h, 'Propane')
print('SINGLE PHASE CYCLE (propane)')
print('T,D -> P,H', 300, ',', 1, '-->', p, ',', h)
print('P,H -> T,D', p, ',', h, '-->', T, ',', D)


CP.enable_TTSE_LUT('Propane')
print(' ')
print('************ USING TTSE ***************')
print(' ')
print('TWO PHASE INPUTS (Pressure)')
print('Density of saturated liquid Propane at 101.325 kPa:',
       CP.Props('D', 'P', 101.325, 'Q', 0, 'Propane'), 'kg/m^3')
```

```
print('Density of saturated vapor R290 at 101.325 kPa:',
       CP.Props('D', 'P', 101.325, 'Q', 1, 'R290'), 'kg/m^3')
print('TWO PHASE INPUTS (Temperature)')
print('Density of saturated liquid Propane at 300 K:',
       CP.Props('D', 'T', 300, 'Q', 0, 'Propane'), 'kg/m^3')
print('Density of saturated vapor R290 at 300 K:',
       CP.Props('D', 'T', 300, 'Q', 1, 'R290'), 'kg/m^3')

p = CP.Props('P', 'T', 300, 'D', 1, 'Propane')
h = CP.Props('H', 'T', 300, 'D', 1, 'Propane')
T = CP.Props('T', 'P', p, 'H', h, 'Propane')
D = CP.Props('D', 'P', p, 'H', h, 'Propane')
print('SINGLE PHASE CYCLE (propane)')
print('T,D -> P,H', 300, ',', 1, '-->', p, ',', h)
print('P,H -> T,D', p, ',', h, '-->', T, ',', D)
CP.disable_TTSE_LUT('Propane')

try:
    print(' ')
    print('************ USING REFPROP ***************')
    print(' ')
    print('TWO PHASE INPUTS (Pressure)')
    print('Density of saturated liquid Propane at 101.325 kPa:',
           CP.Props('D', 'P', 101.325, 'Q', 0, 'REFPROP-Propane'), 'kg/m^3')
    print('Density of saturated vapor Propane at 101.325 kPa:',
           CP.Props('D', 'P', 101.325, 'Q', 1, 'REFPROP-propane'), 'kg/m^3')
    print('TWO PHASE INPUTS (Temperature)')
    print('Density of saturated liquid Propane at 300 K:',
           CP.Props('D', 'T', 300, 'Q', 0, 'REFPROP-propane'), 'kg/m^3')
    print('Density of saturated vapor Propane at 300 K:',
           CP.Props('D', 'T', 300, 'Q', 1, 'REFPROP-propane'), 'kg/m^3')

    p = CP.Props('P', 'T', 300, 'D', 1, 'Propane')
    h = CP.Props('H', 'T', 300, 'D', 1, 'Propane')
    T = CP.Props('T', 'P', p, 'H', h, 'Propane')
    D = CP.Props('D', 'P', p, 'H', h, 'Propane')
    print('SINGLE PHASE CYCLE (propane)')
    print('T,D -> P,H', 300, ',', 1, '-->', p, ',', h)
    print('P,H -> T,D', p, ',', h, '-->', T, ',', D)
except:
    print(' ')
    print('************ CANT USE REFPROP ************')
    print(' ')

print(' ')
print('************ CHANGE UNIT SYSTEM (default is kSI) *************')
print(' ')
CP.set_standard_unit_system(CoolProp.UNIT_SYSTEM_SI)
print('Vapor pressure of water at 373.15 K in SI units (Pa):',
       CP.Props('P', 'T', 373.15, 'Q', 0, 'Water'))
CP.set_standard_unit_system(CoolProp.UNIT_SYSTEM_KSI)
print('Vapor pressure of water at 373.15 K in kSI units (kPa):',
       CP.Props('P', 'T', 373.15, 'Q', 0, 'Water'))

print(' ')
print('************ BRINES AND SECONDARY WORKING FLUIDS *************')
print(' ')
print('Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa:',
```

```
       CP.Props('D', 'T', 300, 'P', 101.325, 'MEG-50%'), 'kg/m^3')
print('Viscosity of Therminol D12 at 350 K, 101.325 kPa:',
       CP.Props('V', 'T', 350, 'P', 101.325, 'TD12'), 'Pa-s')

print(' ')
print('************ HUMID AIR PROPERTIES *************')
print(' ')
print('Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa:',
       CP.HAProps('W', 'T', 300, 'P', 101.325, 'R', 0.5), 'kg_w/kg_da')
print('Relative humidity from last calculation:',
       CP.HAProps('R', 'T', 300, 'P', 101.325, 'W',
                  CP.HAProps('W', 'T', 300, 'P', 101.325, 'R', 0.5)),
       '(fractional)')
```

Output

```
CoolProp version:  4.0.0
CoolProp gitrevision:  aa5be70f193f12056808e4abb620f261133fca6d
CoolProp fluids:  ['Water', 'R134a', 'Helium', 'Oxygen', 'Hydrogen', 'ParaHydrogen', 'OrthoHydrogen',

************ USING EOS *************

FLUID STATE INDEPENDENT INPUTS
Critical Density Propane: 220.4781 kg/m^3
TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.882951955 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.41613600879 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.447375252 kg/m^3
Density of saturated vapor R290 at 300 K: 21.6295320185 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300 , 1 --> 56.0727627483 , 634.733625928
P,H -> T,D 56.0727627483 , 634.733625928 --> 300.0 , 1.0

************ USING TTSE ***************

TWO PHASE INPUTS (Pressure)
0.085 to build both two phase tables
Density of saturated liquid Propane at 101.325 kPa: 580.882952266 kg/m^3
Density of saturated vapor R290 at 101.325 kPa: 2.41613600655 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.447377441 kg/m^3
Density of saturated vapor R290 at 300 K: 21.6295318773 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300 , 1 --> 56.0727644058 , 634.733625854
P,H -> T,D 56.0727644058 , 634.733625854 --> 299.999999907 , 0.999999979733

************ USING REFPROP ***************

TWO PHASE INPUTS (Pressure)
Density of saturated liquid Propane at 101.325 kPa: 580.882951955 kg/m^3
Density of saturated vapor Propane at 101.325 kPa: 2.41613600879 kg/m^3
TWO PHASE INPUTS (Temperature)
Density of saturated liquid Propane at 300 K: 489.447375252 kg/m^3
Density of saturated vapor Propane at 300 K: 21.6295320185 kg/m^3
SINGLE PHASE CYCLE (propane)
T,D -> P,H 300 , 1 --> 56.0727627483 , 634.733625928
```

```
P,H -> T,D 56.0727627483 , 634.733625928 --> 300.0 , 1.0

************ CHANGE UNIT SYSTEM (default is kSI) *************

Vapor pressure of water at 373.15 K in SI units (Pa): 101417.99666
Vapor pressure of water at 373.15 K in kSI units (kPa): 101.41799666

************ BRINES AND SECONDARY WORKING FLUIDS *************

Density of 50% (mass) ethylene glycol/water at 300 K, 101.325 kPa: 1061.17930772 kg/m^3
Viscosity of Therminol D12 at 350 K, 101.325 kPa: 0.000522884941051 Pa-s

************ HUMID AIR PROPERTIES *************

Humidity ratio of 50% rel. hum. air at 300 K, 101.325 kPa: 0.0110962237501 kg_w/kg_da
Relative humidity from last calculation: 0.5 (fractional)
```

2.2 Sample Props Code

To use the Props function, import it and do some calls, do something like this

Or go to the *Fluid Properties* documentation.

All the possible input and output parameters are listed in the `CoolProp.CoolProp.Props()` documentation

2.3 Sample HAProps Code

To use the HAProps function, import it and do some calls, do something like this

Or go to the *Humid Air Properties* documentation.

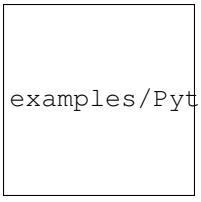
2.4 Plotting

2.4.1 Python Plotting

Note: The python plotting API has been changed as of v4.0. The examples shown on this page use the new python plotting API. Examples using the old python plotting API can be found here *python-plotting-old*.

The following example can be used to create a Temperature-Entropy plot for propane (R290):

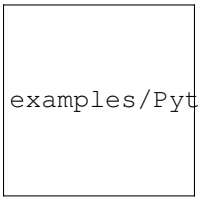
```
from CoolProp.Plots import PropsPlot

ts_plot = PropsPlot('R290', 'Ts')
ts_plot.show()
```



examples/Python/../../_build/plot_directive/examples/Python/plotting-1.pdf

The following example can be used to create a Pressure-Enthalpy plot for R410A:

```
from CoolProp.Plots import PropsPlot

ph_plot = PropsPlot('R410A', 'Ph')
ph_plot.show()
```

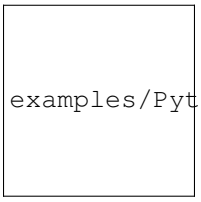


The available plots are:

PT	Pressure-Temperature
PD	Pressure-Density
PH	Pressure-Enthalpy
PS	Pressure-Entropy
TD	Temperature-Density
TS	Temperatre-Entropy
HS	Enthalpy-Entropy

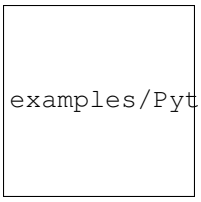
The following, more advanced example, can be used to draw lines of constant properties for n-Pentane. Note the different ways to invoke the `CoolProp.Plots.Plots.PropsPlot.draw_isolines()` function draw:

```
from CoolProp.Plots import PropsPlot

ref_fluid = 'n-Pentane'
ts_plot = PropsPlot(ref_fluid, 'Ts')
ts_plot.draw_isolines('Q', [0.3, 0.5, 0.7, 0.8])
ts_plot.draw_isolines('P', [100, 2000], num=5)
ts_plot.draw_isolines('D', [2, 600], num=7)
ts_plot.set_axis_limits([-2, 1.5, 200, 500])
ts_plot.show()
```



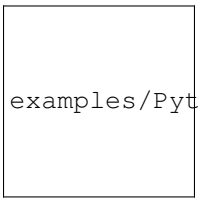
Some of the commonly used Matplotlib functions, such as `title()`, `xlabel()` and `ylabel()` have been wrapped in the `CoolProp.Plots.Plots.PropsPlot` class to make the plotting of graphs a little simpler, for example:

```
from CoolProp.Plots import PropsPlot

ts_plot = PropsPlot('Water', 'Ts')
ts_plot.title('Ts Graph for Water')
ts_plot.xlabel(r's $[{kJ}/{kg K}]$')
ts_plot.ylabel(r'T $[K]$')
ts_plot.grid()
ts_plot.show()
```

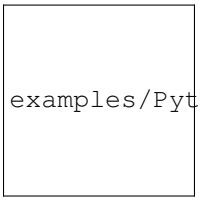


The following two examples show how the `matplotlib.pyplot` functions and `matplotlib.pyplot.axes` functions can also be used along side the `CoolProp.Plots.Plots.PropsPlot` class

```
from CoolProp.Plots import PropsPlot

ph_plot = PropsPlot('Water', 'Ph')
ax = ph_plot.axis
ax.set_yscale('log')
ax.text(400, 5500, 'Saturated Liquid', fontsize=15, rotation=40)
ax.text(2700, 3500, 'Saturated Vapour', fontsize=15, rotation=-100)
ph_plot.show()
```




```
from matplotlib import pyplot
from CoolProp.Plots import PropsPlot

ref_fluid = 'R600a'
fig = pyplot.figure(1, figsize=(10, 10), dpi=100)
for i, gtype in enumerate(['PT', 'PD', 'PS', 'PH', 'TD', 'TS', 'HS']):
    ax = pyplot.subplot(4, 2, i+1)
    if gtype.startswith('P'):
        ax.set_yscale('log')
    props_plot = PropsPlot(ref_fluid, gtype, axis=ax)
    props_plot.title(gtype)
    props_plot._draw_graph()
pyplot.tight_layout()
pyplot.show()
```



CHAPTER 3

Fluid Properties

3.1 Sample Code

The documentation of the `CoolProp.CoolProp` module, or the `CoolProp.State` module are also available.

3.2 Introduction

Nearly all the fluids modeling in CoolProp are based on Helmholtz energy formulations. This is a convenient construction of the equation of state because all the thermodynamic properties of interest can be obtained directly from partial derivatives of the Helmholtz energy.

It should be noted that the EOS are typically valid over the entire range of the fluid, from subcooled liquid to superheated vapor, to supercritical fluid.

Annoyingly, different authors have selected different sets of nomenclature for the Helmholtz energy. For consistency, the nomenclature of Lemmon will be used here. Also, some authors present results on a mole-basis or mass-basis, further complicating comparisons.

3.3 Thermodynamic properties of Fluid

In general, the EOS are based on non-dimensional terms δ and τ, where these terms are defined by

$$\delta = \rho/\rho_c$$
$$\tau = T_c/T$$

where ρ_c and T_c are the critical density of the fluid if it is a pure fluid. For pseudo-pure mixtures, the critical point is typically not used as the reducing state point, and often the maximum condensing temperature on the saturation curve is used instead.

The non-dimensional Helmholtz energy of the fluid is given by

$$\alpha = \alpha^0 + \alpha^r$$

where α^0 is the ideal-gas contribution to the Helmholtz energy, and α^r is the residual Helmholtz energy contribution which accounts for non-ideal behavior. For a given set of δ and τ, each of the terms α^0 and α^r are known. The exact

form of the Helmholtz energy terms is fluid dependent, but a relatively simple example is that of Nitrogen, which has the ideal-gas Helmholtz energy of

$$\alpha^0 = \ln\delta + a_1 \ln\tau + a_2 + a_3\tau + a_4\tau^{-1} + a_5\tau^{-2} + a_6\tau^{-3} + a_7 \ln[1 - \exp(-a_8\tau)]$$

and the non-dimensional residual Helmholtz energy of

$$\alpha^r = \sum_{k=1}^{6} N_k \delta^{i_k} \tau^{j_k} + \sum_{k=7}^{32} N_k \delta^{i_k} \tau^{j_k} \exp(-\delta^{l_k}) + \sum_{k=33}^{36} N_k \delta^{i_k} \tau^{j_k} \exp(-\phi_k(\delta-1)^2 - \beta_k(\tau-\gamma_k)^2)$$

and all the terms other than δ and τ are fluid-dependent correlation parameters.

The other thermodynamic parameters can then be obtained through analytic derivatives of the Helmholtz energy terms. For instance, the pressure is given by

$$p = \rho RT \left[1 + \delta \left(\frac{\partial \alpha^r}{\partial \delta}\right)_\tau\right]$$

and the specific internal energy by

$$\frac{u}{RT} = \tau \left[\left(\frac{\partial \alpha^0}{\partial \tau}\right)_\delta + \left(\frac{\partial \alpha^r}{\partial \tau}\right)_\delta\right]$$

and the specific enthalpy by

$$\frac{h}{RT} = \tau \left[\left(\frac{\partial \alpha^0}{\partial \tau}\right)_\delta + \left(\frac{\partial \alpha^r}{\partial \tau}\right)_\delta\right] + \delta \left(\frac{\partial \alpha^r}{\partial \delta}\right)_\tau + 1$$

which can also be written as

$$\frac{h}{RT} = \frac{u}{RT} + \frac{p}{\rho RT}$$

The specific entropy is given by

$$\frac{s}{R} = \tau \left[\left(\frac{\partial \alpha^0}{\partial \tau}\right)_\delta + \left(\frac{\partial \alpha^r}{\partial \tau}\right)_\delta\right] - \alpha^0 - \alpha^r$$

and the specific heats at constant volume and constant pressure respectively are given by

$$\frac{c_v}{R} = -\tau^2 \left[\left(\frac{\partial^2 \alpha^0}{\partial \tau^2}\right)_\delta + \left(\frac{\partial^2 \alpha^r}{\partial \tau^2}\right)_\delta\right]$$

$$\frac{c_p}{R} = \frac{c_v}{R} + \frac{\left[1 + \delta\left(\frac{\partial \alpha^r}{\partial \delta}\right)_\tau - \delta\tau\left(\frac{\partial^2 \alpha^r}{\partial \delta \partial \tau}\right)\right]^2}{\left[1 + 2\delta\left(\frac{\partial \alpha^r}{\partial \delta}\right)_\tau + \delta^2\left(\frac{\partial^2 \alpha^r}{\partial \delta^2}\right)_\tau\right]}$$

The EOS is set up with temperature and density as the two independent properties, but often other inputs are known, most often temperature and pressure because they can be directly measured. As a result, if the density is desired for a known temperature and pressure, it can be obtained iteratively. The following algorithm is used to obtain a reasonable guess for the initial value for the iterative solver:

1. If the fluid is superheated, use a guess of ideal gas ($\rho = p/(RT)$)
2. If the fluid is subcooled, use a guess of saturated liquid density
3. If the fluid is supercritical, use a guess of ideal gas ($\rho = p/(RT)$)
4. No solution for density as a function of temperature and pressure if the fluid is two-phase

3.4 Saturation State

If the fluid is somewhere in the two-phase region, or saturation state properties are desired, saturated liquid and vapor properties can be obtained. At equilibrium, the Gibbs function of the liquid and vapor are equal, as are the pressures of the saturated liquid and vapor. For nearly all pure fluids, ancillary equations for the density of saturated liquid and saturated vapor as a function of temperature are provided, given by ρ' and ρ'' respectively. Thus for pure fluids, for a given temperature, initial guesses for the densities of saturated liquid and vapor are given by ρ' and ρ''. Using one of the densities, a guess for the saturation pressure can be obtained. Then, the saturation pressure is iteratively altered using a numerical method. For each saturation pressure, the saturated liquid and vapor densities are updated using the full EOS to match the imposed temperature and guessed pressure. Because the density is known explicitly from the EOS, Newton's method can be used to update the densities. For Newton's method, the derivative $\partial\rho/\partial p$ is needed, which can be given explicitly as

$$\frac{\partial p}{\partial \rho} = RT\left[1 + 2\delta\left(\frac{\partial \alpha^r}{\partial \delta}\right)_\tau + \delta^2\left(\frac{\partial^2 \alpha^r}{\partial \delta^2}\right)_\tau\right]$$

and the value for ρ is updated by employing

$$\rho_{new} = \rho_{old} - \frac{p(T,\rho_{old}) - p_{guess}}{\frac{\partial p}{\partial \rho}(T,\rho_{old})}$$

until $|p(T,\rho_{old}) - p_{guess}|$ is sufficiently small. Then the numerical method calculates the Gibbs function for saturated liquid and saturated vapor, and uses the difference in Gibbs functions to update the guess for the saturation pressure. Then the densities are calculated again. At convergence, the set of ρ', ρ'', and p_{sat} are known for a given saturation temperature. If the fluid is not a pure fluid, the best that you can do is to use the ancillary equations to calculate the saturation densities and saturation pressure.

As you might imagine, doing all this work to calculate the saturation state for pure fluids is computationally *very* expensive, so a lookup table method has been implemented for the saturation densities and saturation pressure. From Python, you can turn on the saturation lookup table with:

```
UseSaturationLUT(True)
```

or use the full EOS by calling:

```
UseSaturationLUT(False)
```

3.5 Properties as a function of h,p

As a reminder, the EOS is typically set up as a function of $\tau = T_c/T$ and $\delta = \rho/\rho_c$. Thus, if you know pressure and enthalpy, you can set up a system of residuals in terms of δ and τ in order to yield back the given pressure and enthalpy. Of course you still need a good guess value to start from. See below for that. The system of equations can be given by:

$$f_1 = \frac{\delta}{\tau}\left(1 + \delta\frac{\partial \alpha^r}{\partial \delta}\right) - \frac{p_0}{\rho_c R T_c}$$

$$f_2 = \left(1 + \delta\frac{\partial \alpha^r}{\partial \delta}\right) + \tau\left(\frac{\partial \alpha^0}{\partial \tau} + \frac{\partial \alpha^r}{\partial \tau}\right) - \tau\frac{h_0}{RT_c}$$

where the partials can be given by

$$\frac{\partial f_1}{\partial \tau} = \left(1 + \delta\frac{\partial \alpha^r}{\partial \delta}\right)\left(\frac{-\delta}{\tau^2}\right) + \frac{\delta}{\tau}\left(\delta\frac{\partial^2 \alpha^r}{\partial \delta \partial \tau}\right)$$

$$\frac{\partial f_1}{\partial \delta} = \left(1 + \delta \frac{\partial \alpha^r}{\partial \delta}\right)(\frac{1}{\tau}) + \frac{\delta}{\tau}\left(\frac{\partial \alpha^r}{\partial \delta} + \delta \frac{\partial^2 \alpha^r}{\partial \delta^2}\right) = \left(\frac{1}{\tau}\right)\left(1 + 2\delta \frac{\partial \alpha^r}{\partial \delta} + \delta^2 \frac{\partial^2 \alpha^r}{\partial \delta^2}\right)$$

$$\frac{\partial f_2}{\partial \tau} = \left(\delta \frac{\partial^2 \alpha^r}{\partial \delta \partial \tau}\right) + \left(\frac{\partial \alpha^0}{\partial \tau} + \frac{\partial \alpha^r}{\partial \tau}\right) + \tau\left(\frac{\partial^2 \alpha^0}{\partial \tau^2} + \frac{\partial^2 \alpha^r}{\partial \tau^2}\right) - \frac{h_0}{RT_c}$$

$$\frac{\partial f_2}{\partial \delta} = \left(\frac{\partial \alpha^r}{\partial \delta} + \delta \frac{\partial^2 \alpha^r}{\partial \delta^2}\right) + \tau\left(\frac{\partial^2 \alpha^0}{\partial \tau \partial \delta} + \frac{\partial^2 \alpha^r}{\partial \tau \partial \delta}\right)$$

and the jacobian is then

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial \tau} & \frac{\partial f_1}{\partial \delta} \\ \frac{\partial f_2}{\partial \tau} & \frac{\partial f_2}{\partial \delta} \end{bmatrix}$$

3.6 Use of Extended Corresponding States for Transport Properties

For a limited selection of fluids, correlations are provided for the viscosity and the thermal conductivity. But for many fluids, no correlations are available, and therefore other methods must be employed. The extended corresponding states is a method of estimating the transport properties of a fluid by analogy with the transport properties of a fluid that are well defined.

Implementing the ECS method is quite a challenge, but CoolProp is one of the only fluid property databases that properly implements it. And the onlyopen-source package that does. A multi-step method is required, which is hopefully clearly laid out here.

To begin with, the reference fluid must be selected that the fluid of interest will be compared with. Ideally the shape of the molecules should be similar, but in practice, most fluids use R134a as the reference fluid since its thermodynamic and transport properties are well quantified with reference-quality correlations.

Once the reference fluid has been selected, the conformal state of the reference fluid must be determined. The conformal state is the state at which the transport properties of the reference fluid and the fluid of interest are (in theory) the same. In practice, at low densities the shape factors are assumed to be unity, and the conformal temperature and molar density are obtained from

$$T_0 = T\frac{T_0^c}{T_j^c}$$

$$\overline{\rho_0} = \overline{\rho}\frac{\overline{\rho_0}^c}{\overline{\rho_j}^c}$$

Exact solution for the conformal temperature

If you have Helmholtz EOS for both the fluid and the reference fluid, you need to find a conformal temperature for the reference fluid that will yield the same compressibility factor and the residual Helmholtz energy

$$Z_j(T_j, \rho_j) = Z_0(T_0, \rho_0)$$

$$\alpha_j^r(T_j, \rho_j) = \alpha_0^r(T_0, \rho_0)$$

where "j" is for the fluid of interest, and the subscript "0" is for the reference fluid. The left side of each equation is already known from the equation of state. Literature suggests that solving for T_0 and ρ_0 directly is quite challenging. See McLinden 2000 or Klein 1997.

Alternatively, if the shape factors θ and ϕ are known, either from correlation or otherwise, the conformal temperature and density can be calculated directly.

$$T_0 = \frac{T}{f} = T\frac{T_0^c}{T_j^c \theta(T_j, \rho_j)}$$

$$\rho_0 = \rho h = \rho \frac{\rho_0^c}{\rho_j^c}\phi(T_j, \rho_j)$$

3.7 Conversion from ideal gas term to Helmholtz energy term

Much of the time the coefficients for the ideal-gas part of the Helmholtz energy are given directly, but sometimes only the gas-specific heat is provided. Therefore you need to be able to go from specific heat to ideal-gas Helmholtz Energy. The ideal-gas Helmholtz energy is given by Equation 23 from Lemmon, 2004, Equations of State for Mixtures of R-32, R-125, R-134a, R-143a, and R-152a, J. Phys. Chem. Ref. Data, Vol. 33, No. 2, 2004 or

$$a_0 = -RT + RT\ln\frac{\rho T}{\rho_0 T_0} + h_0^0 - Ts_0^0 + \int_{T_0}^{T} c_p^0(T)dT - T\int_{T_0}^{T}\frac{c_p^0(T)}{T}dT$$

non-dimensionalizing

$$\alpha_0 = \frac{a_0}{RT} = -1 + \ln\frac{\rho T}{\rho_0 T_0} + \frac{h_0^0}{RT} - \frac{s_0^0}{R} + \frac{1}{RT}\int_{T_0}^{T} c_p^0(T)dT - \frac{1}{R}\int_{T_0}^{T}\frac{c_p^0(T)}{T}dT$$

Now we might want to do a change of variable in the integrals. If so, do a u-substitution in the integrals.

$$T = \frac{T_c}{\tau}$$

where

$$dT = -\frac{T_c}{\tau^2}d\tau$$

$$\alpha_0 = -1 + \ln\frac{\rho T}{\rho_0 T_0} + \frac{h_0^0}{RT} - \frac{s_0^0}{R} + \frac{1}{RT}\int_{\tau_0}^{\tau} c_p^0(T)(-\frac{T_c}{\tau^2}d\tau) - \frac{1}{R}\int_{\tau_0}^{\tau}\frac{c_p^0(\tau)}{T}(-\frac{T_c}{\tau^2}d\tau)$$

Simplifying and factoring the τ term yields

$$\alpha_0 = -1 + \ln\frac{\rho T}{\rho_0 T_0} + \frac{h_0^0}{RT} - \frac{s_0^0}{R} - \frac{\tau}{R}\int_{\tau_0}^{\tau}\frac{c_p^0(\tau)}{\tau^2}d\tau + \frac{1}{R}\int_{\tau_0}^{\tau}\frac{c_p^0(\tau)}{\tau}d\tau$$

which finally yields the solution as of Equation 3 from Lemmon, 2003 (and others)

The specific-heat contribution can then be taken as a sum of the contributions

for a term of the form

$$\frac{c_p^0}{R} = \frac{(B/T)^2\exp(B/T)}{(\exp(B/T)-1)^2}$$

the contribution is found from

$$\frac{1}{T}\int_{T_0}^{T}\frac{(B/T)^2\exp(B/T)}{(\exp(B/T)-1)^2}dT - \int_{T_0}^{T}\frac{(B/T)^2\exp(B/T)}{(\exp(B/T)-1)^2}\frac{1}{T}dT$$

$$\frac{1}{T}\left[\frac{B}{\exp(B/T)-1}\right|_{T_0}^{T} - \left[\frac{B}{T}\left(\frac{1}{\exp(B/T)-1}+1\right) - \log[\exp(B/T)-1]\right|_{T_0}^{T} dT$$

Factor out a B, First two terms cancel, leaving

$$-\left[\frac{B}{T} - \log[\exp(B/T)-1]\right|_{T_0}^{T} dT$$

$$\left[\log[\exp(B/T)-1] - \frac{B}{T}\right|_{T_0}^{T} dT$$

$$\log[\exp(B/T)-1] - \frac{B}{T} - (\log[\exp(B/T_0)-1] - \frac{B}{T_0})$$

or in terms of τ

$$\log[\exp(B\tau/Tc)-1] - \frac{B\tau}{Tc} - (\log[\exp(B\tau_0/T_c)-1] - \frac{B\tau_0}{T_c})$$

for a term of the form

$$\frac{c_p^0}{R} = c$$

the contribution is found from

$$\frac{1}{T}\int_{T_0}^{T} c dT - \int_{T_0}^{T} \frac{c}{T} dT$$

$$\frac{c}{T}(T-T_0) - c\log(T/T_0)$$

or in terms of τ

$$c - \frac{cT_0\tau}{T_c} + c\log(\tau/\tau_0)$$

for a term of the form

$$\frac{c_p^0}{R} = cT^t, t \neq 0$$

the contribution is found from

$$\frac{1}{T}\int_{T_0}^{T} cT^t dT - \int_{T_0}^{T} \frac{cT^t}{T} dT$$

$$\frac{c}{T}\left(\frac{T^{t+1}}{t+1} - \frac{T_0^{t+1}}{t+1}\right) - c\left(\frac{T^t}{t} - \frac{T_0^t}{t}\right)$$

$$cT^t\left(\frac{1}{t+1} - \frac{1}{t}\right) - c\frac{T_0^{t+1}}{T(t+1)} + c\frac{T_0^t}{t}$$

or in terms of τ

$$cT_c^t\tau^{-t}\left(\frac{1}{t+1} - \frac{1}{t}\right) - c\frac{T_0^{t+1}\tau}{T_c(t+1)} + c\frac{T_0^t}{t}$$

These terms can be summarized by the following table:

$\frac{c_p^0}{R}$Term	α^0Term	ClassName
$a_k \frac{(b_k/T)^2 \exp(b_k/T)}{(\exp(b_k/T)-1)^2}$	$a_k \ln\left[1-\exp\left(\frac{-b_k\tau}{T_c}\right)\right]$	phi0_Planck_Einstein$(a, b/Tc, [iStart, iEnd])$
$ac\frac{(b/T)^2\exp(-b/T)}{(c\exp(-b/T)+1)^2}$	$a\ln\left[c+\exp\left(\frac{b\tau}{T_c}\right)\right]$	phi0_Planck_Einstein2$(a, b/Tc, c)$
$yuck$	$a_k\tau^{b_k}$	phi0_power $(a, b, [iStart, iEnd])$
a	$a - a\frac{\tau}{\tau_0} + a\ln\left(\frac{\tau}{\tau_0}\right)$	phi0_cp0_constant$(a, Tc, T0)$
$a_1 + a_2\left(\frac{a_3/T}{\sinh(a_3/T)}\right)^2 + a_4\left(\frac{a_5/T}{\cosh(a_5/T)}\right)^2$	$yuck$	phi0_cp0_AlyLee$(a, Tc, T0, R)$
n/a	$\log(\delta) + a_1 + a_2\tau$	phi0_lead$(a1, a2)$
n/a	$a\log\tau$	phi0_logtau(a)

If the reference enthalpy is known, you can determine the constants from

$$\frac{h_0}{RT} = \tau\left[\left(\frac{\partial\alpha^0}{\partial\tau}\right)_\delta + \left(\frac{\partial\alpha^r}{\partial\tau}\right)_\delta\right] + \delta\left(\frac{\partial\alpha^r}{\partial\delta}\right)_\tau + 1$$

$$\left(\frac{\partial\alpha^0}{\partial\tau}\right)_\delta = \frac{1}{\tau}\left(\frac{h_0}{RT} - \delta\left(\frac{\partial\alpha^r}{\partial\delta}\right)_\tau - 1\right) - \left(\frac{\partial\alpha^r}{\partial\tau}\right)_\delta$$

For the specific heat The two integral terms are

$$-\frac{\tau}{R}\int_{\tau_0}^{\tau}\frac{c_p^0}{\tau^2}d\tau + \frac{1}{R}\int_{\tau_0}^{\tau}\frac{c_p^0}{\tau}d\tau$$

First derivative

$$\frac{d}{d\tau}\left[-\frac{\tau}{R}\int_{\tau_0}^{\tau}\frac{c_p^0}{\tau^2}d\tau + \frac{1}{R}\int_{\tau_0}^{\tau}\frac{c_p^0}{\tau}d\tau\right] = -\frac{c_p^0}{\tau R} - \frac{1}{R}\int_{\tau_0}^{\tau}\frac{c_p^0}{\tau^2}d\tau + \frac{c_p^0}{\tau R} = -\frac{1}{R}\int_{\tau_0}^{\tau}\frac{c_p^0}{\tau^2}d\tau$$

Second Derivative

$$\frac{d^2}{d\tau^2}\left[-\frac{\tau}{R}\int_{\tau_0}^{\tau}\frac{c_p^0}{\tau^2}d\tau + \frac{1}{R}\int_{\tau_0}^{\tau}\frac{c_p^0}{\tau}d\tau\right] = \frac{d}{d\tau}\left[-\frac{1}{R}\int_{\tau_0}^{\tau}\frac{c_p^0}{\tau^2}d\tau\right] = -\frac{c_p^0}{\tau^2 R}$$

3.8 Converting Bender and mBWR EOS

If the EOS is of the form

$$\frac{p}{\rho RT} = Z(T,\rho) = 1 + \sum_i n_i T^{s_i}\rho^{r_i} + \sum_i n_i T^{s_i}\rho^{r_i}\exp\left(-\gamma_i\left(\frac{\rho}{\rho_c}\right)^2\right)$$

To convert to standard power form in CoolProp, use

$$\delta\sum_i d_i a_i \tau^{t_i}\delta^{d_i-1} = \sum_i n_i T^{s_i}\rho^{r_i} = \sum_i n_i\left(\frac{T_c}{\tau}\right)^{s_i}(\rho_c\delta)^{r_i} = \sum_i n_i T_c^{s_i}\rho_c^{r_i}\tau^{-s_i}\delta^{r_i}$$

The left-hand-side is the derivative of the residual Helmholtz energy with respect to delta times the reduced density since

$$\frac{p}{\rho RT} = 1 + \delta\left(\frac{\partial\alpha^r}{\partial\delta}\right)_\tau$$

where

$$\delta : d_i - 1 + 1 = r_i \Rightarrow d_i = r_i$$

$$\tau : t_i = -s_i$$

$$c : d_i a_i = n_i T_c^{s_i} \rho_c^{r_i}$$

$$p = \rho RT + \sum_i n_i T^{s_i} \rho^{r_i} + \sum_i n_i T^{s_i} \rho^{r_i} \exp\left(-\gamma_i \left(\frac{\rho}{\rho_c}\right)^2\right) \text{(Eq3.28)}$$

$$\frac{p}{\rho RT} = 1 + \sum_i \frac{n_i}{R} T^{s_i - 1} \rho^{r_i - 1} + \sum_i \frac{n_i}{R} T^{s_i - 1} \rho^{r_i - 1} \exp\left(-\gamma_i \left(\frac{\rho}{\rho_c}\right)^2\right)$$

$$\delta \sum_i d_i a_i \tau^{t_i} \delta^{d_i - 1} = \sum_i \frac{n_i}{R} \left(\frac{T_c}{\tau}\right)^{s_i - 1} (\rho_c \delta)^{r_i - 1} = \sum_i \frac{n_i}{R} T_c^{s_i - 1} \rho_c^{r_i - 1} \tau^{-(s_i - 1)} \delta^{r_i - 1}$$

$$\delta : 1 + d_i - 1 = r_i - 1$$

$$\tau : t_i = -\left(s_i - 1\right)$$

$$c : d_i a_i = \frac{n_i}{R} T_c^{s_i - 1} \rho_c^{r_i - 1}$$

In the Bender EOS, for the exponential part you have terms that can be converted to reduced form

$$a_i \delta^{d_i} \tau^{t_i} \exp(-\gamma \delta^2)$$

which yields the terms in the following table (from Span, 2000)

From Bender				Power term			
i	d_i	t_i	γ_i	n_i	d_i	t_i	γ_i
14	2	3	γ	$n_{14}/(2\gamma) + n_{17}/(2\gamma^2)$	0	3	0
15	2	4	γ	$n_{15}/(2\gamma) + n_{17}/(2\gamma^2)$	0	4	0
16	2	5	γ	$n_{16}/(2\gamma) + n_{17}/(2\gamma^2)$	0	5	0
17	4	3	γ	$-n_{14}/(2\gamma) - n_{17}/(2\gamma^2)$	0	3	γ
18	4	4	γ	$-n_{15}/(2\gamma) - n_{18}/(2\gamma^2)$	0	4	γ
19	4	5	γ	$-n_{16}/(2\gamma) - n_{19}/(2\gamma^2)$	0	5	γ
20				$-n_{17}/(2\gamma)$	2	3	γ
21				$-n_{18}/(2\gamma)$	2	4	γ
22				$-n_{19}/(2\gamma)$	2	5	γ

Warning: If the terms in the EOS are in terms of T and ρ rather than τ and δ, make sure to multiply appropriately by the critical densities in the exponential term. For instance in Polt paper, the first constant should be $n_{14}\rho_c^2/(2\gamma) + n_{17}\rho_c^4/(2\gamma^2)/T_c^3$ Be careful!

CHAPTER 4

Tabular Taylor Series Extrapolation

4.1 Introduction

Lookup tables are an effective way of greatly speeding up the computational time when the state variable inputs are variables other than temperature and density (upon which the equations of state are based).

The tables are constructed with enthalpy and pressure as the independent variables. Other inputs are also possible, but the efficiency of the lookup table method is maximized through the use of these independent variables

TTSE was added to CoolProp as of version 3.0. As of version 4.0, bicubic interpolation (see http://en.wikipedia.org/wiki/Bicubic_interpolation) has been added, which greatly improves the accuracy at a small speed penalty (~10%) over TTSE.

4.2 Usage

The tables can be enabled by calling the function `enable_TTSE_LUT()` of the CoolPropStateClass (only C++), or alternatively, the `enable_TTSE_LUT(FluidName)` (Python + anything that calls the DLL) function. This will enable ALL thermodynamic calls for this fluid to use the TTSE method. The function `disable_TTSE_LUT` is used to disable the TTSE method

Two types of tables are built:

1. Tables are constructed along each saturation curve, all the way from the triple point (or minimum) pressure to the critical pressure
2. A single-phase table is constructed for the areas outside of the two-phase region

When the single phase tables are constructed, a default range is employed, which is taken to be from the triple point pressure to the twice the critical pressure, and from the saturated liquid enthalpy at the triple point pressure to an enthalpy that is three times the latent heat at the triple point pressure plus the saturated liquid enthalpy at the triple point pressure.

If you do not like that default range, BEFORE you call enable_TTSE_LUT(), call `set_TTSESinglePhase_LUT_range(FluidName,hmin,hmax,pmin,pmax)`

The single-phase (but not the two-phase) tables are stored in binary form in files for faster loading in the HOME/CoolProp-TTSEData/*FluidName* folder where HOME is your home folder. If you do not want to save the tables to file, you can call the function disable_TTSE_LUT_writing() to disable the writing of the single phase tables to file BEFORE the tables are built

4.3 Example in Python

Note: most other programming languages that are wrappers around the CoolProp.h header should behave in nearly exactly the same fashion, excepting perhaps the `get_TTSESinglePhase_LUT_range` function as it uses passing by reference.

4.4 How it Works (TTSE)

Tables are built of $T(p,h)$, $s(p,h)$, and $\rho(p,h)$ as well as the derivatives of each term with respect to p and h. The property of interest is then expanded around the nearest grid point to yield representations like

$$T = T_{i,j} + \Delta h \left(\frac{\partial T}{\partial h}\right)_p + \Delta p \left(\frac{\partial T}{\partial p}\right)_h + \frac{1}{2}\Delta h^2 \left(\frac{\partial^2 T}{\partial h^2}\right)_p + \frac{1}{2}\Delta p^2 \left(\frac{\partial^2 T}{\partial p^2}\right)_h + \Delta h \Delta p \left(\frac{\partial^2 T}{\partial p \partial h}\right)$$

$$s = s_{i,j} + \Delta h \left(\frac{\partial s}{\partial h}\right)_p + \Delta p \left(\frac{\partial s}{\partial p}\right)_h + \frac{1}{2}\Delta h^2 \left(\frac{\partial^2 s}{\partial h^2}\right)_p + \frac{1}{2}\Delta p^2 \left(\frac{\partial^2 s}{\partial p^2}\right)_h + \Delta h \Delta p \left(\frac{\partial^2 s}{\partial p \partial h}\right)$$

$$\rho = s_{i,j} + \Delta h \left(\frac{\partial \rho}{\partial h}\right)_p + \Delta p \left(\frac{\partial \rho}{\partial p}\right)_h + \frac{1}{2}\Delta h^2 \left(\frac{\partial^2 \rho}{\partial h^2}\right)_p + \frac{1}{2}\Delta p^2 \left(\frac{\partial^2 \rho}{\partial p^2}\right)_h + \Delta h \Delta p \left(\frac{\partial^2 \rho}{\partial p \partial h}\right)$$

$$\Delta h = h - h_i$$
$$\Delta p = p - p_i$$

See the IAPWS TTSE report for a description of the method. Analytic derivatives are used to build the tables

4.5 How it Works (Bicubic)

In the TTSE method, the derivatives are calculated at every grid point. In the bicubic method (see http://en.wikipedia.org/wiki/Bicubic_interpolation) we use the known derivatives at each grid point in order to develop C_1 continuous bicubic functions in each cell. The independent variables (T, ρ) or (p, h) are normalized into unit variables that vary between 0 and 1. Then the bicubic coefficients are found for the cell, and the bicubic form is evaluated.

CHAPTER 5

Humid Air Properties

If you are feeling impatient, jump to *Sample HAProps Code*, or to go to the code documentation `CoolProp.HumidAirProp`, otherwise, hang in there.

Humid air can be modeled as a mixture of air and water vapor. In the simplest analysis, water and air are treated as ideal gases but in principle there is interaction between the air and water molecules that must be included through the use of interaction parameters.

Because humid air is a mixture of dry air (treated as a pseudo-pure gas) and water vapor (treated as a real gas), three variables are required to fix the state by the state postulate.

In the analysis that follows, the three parameters that are ultimately needed to calculate everything else are the dry bulb temperature T, the total pressure p, and the molar fraction of water ψ_w. The molar fraction of air is simply $\psi_a = 1 - \psi_w$.

Of course, it is not so straightforward to measure the mole fraction of water vapor molecules, so other measures are used. There are three different variables that can be used to obtain the mole fraction of water vapor without resorting to iterative methods.

1. Humidity ratio

The humidity ratio W is the ratio of the mass of water vapor to the mass of air in the mixture. Thus the mole fraction of water can be obtained from

$$\psi_w = \frac{n_w}{n} = \frac{n_w}{n_a + n_w} = \frac{m_w/M_w}{m_a/M_a + m_w/M_w} = \frac{m_w}{(M_w/M_a)m_a + m_w} = \frac{1}{(M_w/M_a)/W + 1} = \frac{W}{(M_w/M_a) + W}$$

or

$$\psi_w = \frac{W}{\varepsilon + W}$$

where the ratio of mole masses ε is given by $\varepsilon = M_w/M_a$

2. Relative Humidity

The relative humidity φ is defined as the ratio of the mole fraction of water in the humid air to the saturation mole fraction of water. Because of the presence of air with the water, the pure water saturated vapor pressure $p_{w,s}$ must be multiplied by an enhancement factor f that is very close to one near atmospheric conditions.

Mathematically, the result is

$$\varphi = \frac{\psi_w}{\psi_{w,s}}$$

where

$$\psi_{w,s} = \frac{f p_{w,s}}{p}$$

The product p_s is defined by $p_s = f p_{w,s}$, and $p_{w,s}$ is the saturation pressure of pure water (or ice) at temperature T. This yields the result for ψ_w of

$$\varphi = \frac{\psi_w}{p_s/p}$$

$$\psi_w = \frac{\varphi p_s}{p}$$

3. Dewpoint temperature

The dewpoint temperature is defined as the temperature at which the actual vapor pressure of water is equal to the saturation vapor pressure. At the given dewpoint, the vapor pressure of water is given by

$$p_w = f(p, T_{dp}) p_{w,s}(T_{dp})$$

and the mole fraction of water vapor is obtained from

$$\psi_w = \frac{p_w}{p}$$

Once the state has been fixed by a set of T, p, ψ_w, any parameter of interest can be calculated

5.1 Molar Volume

$$p = \frac{\bar{R}T}{\bar{v}}\left(1 + \frac{B_m}{\bar{v}} + \frac{C_m}{\bar{v}^2}\right) \tag{5.1}$$

The bracketed term on the right hand side is the compressibility Z factor, equal to 1 for ideal gas, and is a measure of non-ideality of the air. The virial terms are given by

$$B_m = (1-\psi_w)^2 B_{aa} + 2(1-\psi_w)\psi_w B_{aw} + \psi_w^2 B_{ww}$$
$$C_m = (1-\psi_w)^3 C_{aaa} + 3(1-\psi_w)^2 \psi_w C_{aaw} + 3(1-\psi_w)\psi_w^2 C_{aww} + \psi_w^3 C_{www}$$

where the virial coefficients are described in ASRAE RP-1485 and their values are provided in *Humid Air Validation*. All virial terms are functions only of temperature.

Usually the temperature is known, the water mole fraction is calculated, and $\bar{v}$ is found using iterative methods, in HAProps, using a secant solver and the first guess that the compressibility factor is 1.0.

5.2 Molar Enthalpy

The molar enthalpy of humid air is obtained from

$$\bar{h} = (1-\psi_w)\bar{h}_a^o + \psi_w \bar{h}_w^o + \bar{R}T\left[(B_m - T\frac{dB_m}{dT})\frac{1}{\bar{v}} + \left(C_m - \frac{T}{2}\frac{dC_m}{dT}\right)\frac{1}{\bar{v}^2}\right]$$

with $\bar{h}$ in kJ/kmol. For both air and water, the full EOS is used to evaluate the enthalpy

$$\bar{h}_a^o = \bar{h}_0 + \bar{R}T\left[1 + \tau\left(\frac{\partial \alpha^o}{\partial \tau}\right)_\delta\right]$$

which is in kJ/kmol, using the mixture $\bar{v}$ to define the parameter $\delta = 1/(\bar{v}\bar{\rho}_c)$ for each fluid, and using the critical molar density for the fluid obtained from $\bar{\rho}_c = 1000\rho_c/M$ to give units of mol/m^3. The offset enthalpies for air and water are given by

$$\bar{h}_{0,a} = -7,914.149298 \text{ kJ/kmol}$$
$$\bar{h}_{0,w} = -0.01102303806 \text{ kJ/kmol}$$

respectively. The enthalpy per kg of dry air is given by

$$h = \bar{h}\frac{1+W}{M_{ha}}$$

5.3 Enhancement factor

The enhancement factor is a parameter that includes the impact of the air on the saturation pressure of water vapor. It is only a function of temperature and pressure, but it must be iteratively obtained due to the nature of the expression for the enhancement factor.

$\psi_{w,s}$ is given by $\psi_{w,s} = fp_{w,s}/p$, where f can be obtained from

$$\ln(f) = \left[\begin{array}{l}
\left[\frac{(1+k_T p_{w,s})(p-p_{w,s}) - k_T\frac{(p^2-p_{w,s}^2)}{2}}{\bar{R}T}\right]\bar{v}_{w,s} + \ln[1-\beta_H(1-\psi_{w,s})p] \\
+\left[\frac{(1-\psi_{w,s})^2 p}{\bar{R}T}\right]B_{aa} - 2\left[\frac{(1-\psi_{w,s})^2 p}{\bar{R}T}\right]B_{aw} - \left[\frac{(p-p_{w,s}-(1-\psi_{w,s})^2 p)}{\bar{R}T}\right]B_{ww} \\
+\left[\frac{(1-\psi_{w,s})^3 p^2}{(\bar{R}T)^2}\right]C_{aaa} + \left[\frac{3(1-\psi_{w,s})^2[1-2(1-\psi_{w,s})]p^2}{2(\bar{R}T)^2}\right]C_{aaw} \\
-\left[\frac{3(1-\psi_{w,s})^2\psi_{w,s}p^2}{(\bar{R}T)^2}\right]C_{aww} - \left[\frac{(3-2\psi_{w,s})\psi_{w,s}^2 p^2 - p_{w,s}^2}{2(\bar{R}T)^2}\right]C_{www} \\
-\left[\frac{(1-\psi_{w,s})^2(-2+3\psi_{w,s})\psi_{w,s}p^2}{(\bar{R}T)^2}\right]B_{aa}B_{ww} \\
-\left[\frac{2(1-\psi_{w,s})^3(-1+3\psi_{w,s})p^2}{(\bar{R}T)^2}\right]B_{aa}B_{aw} \\
+\left[\frac{6(1-\psi_{w,s})^2\psi_{w,s}^2 p^2}{(\bar{R}T)^2}\right]B_{ww}B_{aw} - \left[\frac{3(1-\psi_{w,s})^4 p^2}{2(\bar{R}T)^2}\right]B_{aa}^2 \\
-\left[\frac{2(1-\psi_{w,s})^2\psi_{w,s}(-2+3\psi_{w,s})p^2}{(\bar{R}T)^2}\right]B_{aw}^2 - \left[\frac{p_{w,s}^2-(4-3\psi_{w,s})(\psi_{w,s})^3 p^2}{2(\bar{R}T)^2}\right]B_{ww}^2
\end{array}\right]$$

5.4 Isothermal Compressibility

For water, the isothermal compressibility [in 1/Pa] is evaluated from

$$k_T = \frac{1}{\rho\frac{\partial p}{\partial \rho}}\frac{1 \text{ kPa}}{1000 \text{ Pa}}$$

with

$$\frac{\partial p}{\partial \rho} = RT\left[1 + 2\delta\left(\frac{\partial \alpha^r}{\partial \delta}\right)_\tau + \delta^2\left(\frac{\partial^2 \alpha^r}{\partial \delta^2}\right)_\tau\right]$$

in kPa/(kg/m^3). And for ice,

$$k_T = \left(\frac{\partial^2 g}{\partial p^2}\right)\left(\frac{\partial g}{\partial p}\right)_T^{-1}\frac{1\text{ kPa}}{1000\text{ Pa}}$$

5.5 Sample HAProps Code

To use the HAProps function, import it and do some calls, do something like this

5.6 Humid Air Validation

Values here are obtained at documentation build-time using the Humid Air Properties module

$$p = \frac{\bar{R}T}{\bar{v}}\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)$$

$$0 = \frac{d}{dT}\left[\frac{\bar{R}T}{\bar{v}}\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)\right]$$

$$0 = \frac{\bar{R}T}{\bar{v}}\left(0+\frac{d}{dT}\left[\frac{B_m}{\bar{v}}\right]+\frac{d}{dT}\left[\frac{C_m}{\bar{v}^2}\right]\right)+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)\bar{R}\left(\frac{\bar{v}-T\frac{d\bar{v}}{dT}}{\bar{v}^2}\right)$$

$$0 = \frac{\bar{R}T}{\bar{v}}\left(0+\frac{\bar{v}\frac{dB_m}{dT}-B_m\frac{d\bar{v}}{dT}}{\bar{v}^2}+\frac{\bar{v}^2\frac{dC_m}{dT}-2C_m\bar{v}\frac{d\bar{v}}{dT}}{\bar{v}^4}\right)+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)\bar{R}\left(\frac{\bar{v}-T\frac{d\bar{v}}{dT}}{\bar{v}^2}\right)$$

$$0 = \frac{\bar{R}T}{\bar{v}}\left(\frac{1}{\bar{v}}\frac{dB_m}{dT}-\frac{B_m}{\bar{v}^2}\frac{d\bar{v}}{dT}+\frac{1}{\bar{v}^2}\frac{dC_m}{dT}-\frac{2C_m}{\bar{v}^3}\frac{d\bar{v}}{dT}\right)+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)\bar{R}\left(\frac{1}{\bar{v}}-\frac{T}{\bar{v}^2}\frac{d\bar{v}}{dT}\right)$$

$$0 = \frac{T}{\bar{v}}\left(\frac{1}{\bar{v}}\frac{dB_m}{dT}-\frac{B_m}{\bar{v}^2}\frac{d\bar{v}}{dT}+\frac{1}{\bar{v}^2}\frac{dC_m}{dT}-\frac{2C_m}{\bar{v}^3}\frac{d\bar{v}}{dT}\right)+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)\left(\frac{1}{\bar{v}}-\frac{T}{\bar{v}^2}\frac{d\bar{v}}{dT}\right)$$

$$\frac{d\bar{v}}{dT}\left(\frac{B_m}{\bar{v}^2}\frac{T}{\bar{v}}+\frac{2C_m}{\bar{v}^3}\frac{T}{\bar{v}}+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)\frac{T}{\bar{v}^2}\right)=\frac{T}{\bar{v}}\left(\frac{1}{\bar{v}}\frac{dB_m}{dT}+\frac{1}{\bar{v}^2}\frac{dC_m}{dT}\right)+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)\left(\frac{1}{\bar{v}}\right)$$

$$\frac{d\bar{v}}{dT}\left(\frac{B_m}{\bar{v}^2}T+\frac{2TC_m}{\bar{v}^3}+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)\frac{T}{\bar{v}}\right)=T\left(\frac{1}{\bar{v}}\frac{dB_m}{dT}+\frac{1}{\bar{v}^2}\frac{dC_m}{dT}\right)+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)$$

$$\frac{d\bar{v}}{dT} = \frac{T\left(\frac{1}{\bar{v}}\frac{dB_m}{dT}+\frac{1}{\bar{v}^2}\frac{dC_m}{dT}\right)+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)}{\left(\frac{B_m}{\bar{v}^2}T+\frac{2TC_m}{\bar{v}^3}+\left(1+\frac{B_m}{\bar{v}}+\frac{C_m}{\bar{v}^2}\right)\frac{T}{\bar{v}}\right)}$$

$$\frac{d\bar{v}}{dT} = \frac{T\left(\frac{1}{\bar{v}}\frac{dB_m}{dT}+\frac{1}{\bar{v}^2}\frac{dC_m}{dT}\right)+Z}{\left(\frac{B_m}{\bar{v}^2}T+\frac{2TC_m}{\bar{v}^3}+Z\frac{T}{\bar{v}}\right)}$$

$$\frac{d\bar{v}}{dT} = \frac{\left(\frac{dB_m}{dT} + \frac{1}{\bar{v}}\frac{dC_m}{dT}\right) + Z\frac{\bar{v}}{T}}{\left(\frac{B_m}{\bar{v}} + \frac{2C_m}{\bar{v}^2} + Z\right)}$$

where

$$Z = \left(1 + \frac{B_m}{\bar{v}} + \frac{C_m}{\bar{v}^2}\right)$$

$$\bar{h} = \bar{h}_0 + (1-\psi_w)\,\bar{h}_a^0 + \psi_w \bar{h}_w^0 + \bar{R}T\left[\left(B_m - T\frac{dB_m}{dT}\right)\frac{1}{\bar{v}} + \left(C_m - \frac{T}{2}\frac{dC_m}{dT}\right)\frac{1}{\bar{v}^2}\right]$$

$$\bar{c}_p = \frac{d\bar{h}}{dT} = \frac{\delta\bar{h}}{\delta T} + \frac{\delta\bar{h}}{\delta\bar{v}}\frac{\delta\bar{v}}{\delta T}$$

$$\frac{\delta\bar{h}}{\delta\bar{v}} = (1-\psi_w)\,\frac{d\bar{h}_a^0}{d\delta}\frac{d\delta}{d\bar{v}} + \psi_w\frac{d\bar{h}_w^0}{d\delta}\frac{d\delta}{d\bar{v}} + \bar{R}T\left[\left(B_m - T\frac{dB_m}{dT}\right)\frac{-1}{\bar{v}^2} + \left(C_m - \frac{T}{2}\frac{dC_m}{dT}\right)\frac{-2}{\bar{v}^3}\right]$$

$$\frac{\delta\bar{h}}{\delta T} = (1-\psi_w)\,\frac{d\bar{h}_a^0}{d\tau}\frac{d\tau}{dT} + \psi_w\frac{d\bar{h}_w^0}{d\tau}\frac{d\tau}{dT} + \bar{R}\left[\left(B_m - T\frac{dB_m}{dT}\right)\frac{1}{\bar{v}} + \left(C_m - \frac{T}{2}\frac{dC_m}{dT}\right)\frac{1}{\bar{v}^2}\right] + \bar{R}T\left[\left(\frac{dB_m}{dT} - \frac{dB_m}{dT} - T\frac{d^2B_m}{dT^2}\right.\right.$$

$$\frac{\delta\bar{h}}{\delta T} = (1-\psi_w)\,\frac{d\bar{h}_a^0}{d\tau}\frac{d\tau}{dT} + \psi_w\frac{d\bar{h}_w^0}{d\tau}\frac{d\tau}{dT} + \bar{R}\left[\left(B_m - T\frac{dB_m}{dT}\right)\frac{1}{\bar{v}} + \frac{C_m}{\bar{v}^2}\right] + \bar{R}T^2\left[\left(-\frac{d^2B_m}{dT^2}\right)\frac{1}{\bar{v}} + \left(-\frac{1}{2}\frac{d^2C_m}{dT^2}\right)\frac{1}{\bar{v}^2}\right]$$

Changelog for CoolProp

6.1 4.0.0

- API CHANGE: Some functions have been condensed, functions get_errstring, get_REFPROPname, etc. have been rolled into get_global_param_string.
- MAJOR: Code now is on github (https://github.com/ibell/coolprop)
- MAJOR: Internally all units are SI, functions should do the necessary conversions using conversion_factor() and get/set_standard_unit_system()
- MAJOR: Brines and incompressible liquids are added to CoolPropStateClass
- MAJOR: Preparing to phase out of DerivTerms function, Props now handles derivatives as well.
- Wrappers added for Java, Javascript, MathCAD, MathCAD Prime
- Improved wrapper for Labview (Thanks to the Sergei and guys at UGent)
- Improved plotting in Python (Thanks Logan)
- Improved Modelica wrapper and added incompressible fluids with p,T and p,h as state variables
- Added viscosity for n-Hexane
- Added R1233zd(E)
- Added more incompressible liquids: Therminol D12, Therminol VP-1, Therminol 72, Therminol 66, Dowtherm J, Dowtherm Q, Texatherm 22, Nitrate Salt Blend, Syltherm XLT, Dynalene HC-10, Dynalene HC-20, Dynalene HC-30, Dynalene HC-40, Dynalene HC-50
- Added slurry ice as incompressible solution of either water-ethanol, water-NaCl or water-propylene glycol with solid content as input
- Added corrosion inhibitor ZitrecAC, anti-freezing agent Pekasol2000
- Added Lithium-Bromide/water as incompressible solution (Thanks to Jaroslav PÃ¡tek)

6.2 3.3.0 (revision 660)

- MAJOR: Added bicubic interpolation to TTSE method. Enable by calling `set_TTSE_mode(Fluid,"TTSE")` or `set_TTSE_mode(Fluid,"BICUBIC")` (for bicubic in-

terpolation). Default is normal TTSE interpolation

- Added deuterium and its isomers from preprint
- Isobutane aliases added
- More work on incompressible liquids

6.3 3.2.0 (revision 619)

- Added the function PropsU to python wrapper which allows for use of SI of kSI set of units
- Both inputs to Props can be iterables
- Added the fluid R1234ze(Z)
- Renamed R1234ze to R1234ze(E)
- H-S works
- P-H, P-S fixed
- Fixed n-Undecane entropy
- Fixes to wetbulb temperature
- First code for Mixtures - not exposed through API
- CAS numbers added for all fluids - retrieve using the function `get_CAS_code`

6.4 3.1.2 (revision 577)

- Added the fluids Fluorine, Methanol, R114, R13, R14, R21, RC318, R12, R113
- Isolines are now available for plots (H/T Jorrit Wronski)
- Environmental information on fluids is included, can be obtained using keys GWP100, ODP
- Fixed a bug in HAProps between 273.15 K and 273.16 K
- Fixed some small bugs in ECS for transport properties
- Fixed some bugs in higher derivatives of Helmholtz energy terms

6.5 3.1.1 (revision 544)

- Added the fluid Propyne
- Fixed ECS core code
- Added ECS parameters and changed reference fluids for a lot of fluids
- Fixed Air and H2S transport equations
- Fixed compilation bug for sources

6.6 3.1 (revision 534)

- Added the fluids Propylene, Cyclopentane, R236FA, R236EA, R227EA, R123, R152A, R227EA, R365MFC, R161, HFE143M, Benzene, R11, Undecane, R125, Cyclopropane, Neon, R124
- Added the viscosity and conductivity correlations for a lot of fluids
- Added surface tension, Lennard-Jones parameters for a lot of fluids
- Added enthalpy, entropy as inputs
- Added pressure, density as inputs
- CoolProp builds on Raspberry PI
- CoolProp works in MATLAB on OSX
- Python unit tests have been added in wrappers/CoolProp/CoolProp/tests - a work in progress

6.7 3.0 (revision 325)

- Added Tabular Taylor Series Expansion (see documentation)
- All the way to the critical point for almost all fluids
- Support added for Modelica, Python 3.x and Labview

6.8 2.5 (revision 247)

- Added EES wrapper (r245-r247)
- Saturation derivatives dhdp and d2hdp2 (r244)
- Caching of Helholtz derivatives in CPState.cpp (r243)
- Added Xylenes and EthylBenzene (r242)
- Added n-Dodecane, R23, DMC (r241)

6.9 2.4 (revision 240)

- Added the fluids R1234ze, DME, R143a, n-Pentane, n-Hexane, n-Octane, n-Heptane, CycleHexane, 1-Butene,trans-2-Butene, cis-2-Butene,IsoButene, MethylLinoleate, MethylLinolenate, MethylOleate, Methyl-Palmitate, MethylStearate
- Added C# wrappers (built for Windows) (r240)
- Added Phase_Trho() and Phase_Tp() functions (r240)
- Cleanup of the build process. svnrevision is saved to a file that is built in. Can access the svn revision through the functions get_svnrevision() and get_version()
- Added a genetic algorithm to build ancillaries to dev folder (r226)
- Added third partial derivatives of all the Helmholtz Energy terms (r238)
- **Bugfixes:**

1. Fixed Q(T,rho) (r237) (https://sourceforge.net/p/coolprop/tickets/42/)
2. dhdT and dhdrho added back (r232)
3. Surface tension now properly has the units of N/m as specified in the docs (r228)
4. Fixed bug from Reiner with V and Vda (r227)
5. Added a Brent solver to fix the solution for the saturation around the critical point (r220)(https://sourceforge.net/p/coolprop/tickets/38/)
6. Repaired saturation LUT (r214-r216)
7. Fixed bugs in IsFluidType as well as fixed bugs in Brine entropy calculations (r213)

6.10 2.3 (revision 212)

- Added updated correlations for brines and subcooled liquids from Melinder 2010 (r207)
- Added aliases to docs and python and DLL (r211)
- Excel wrapper updated to catch errors and output them to a message box
- Big speed update to p,Q as inputs (as fast as REFPROP now) (r202)
- Doxygen now gets updated as well (r200)
- **Bugfixes:**
    1. Updated inputs for brines (order doesn't matter) (r208)
    2. Fixed REFPROP with single-input props (r206)
    3. Fixed Manifest file for source distro (r206)
    4. Fixed bug with REFPROP mixtures not being properly parsed (r205 & r212)
    5. Added a backup Brent method for HAProps when solving at low humidity ratio: closed https://sourceforge.net/p/coolprop/tickets/32/ (r204)
    6. Added an example to show how to get version of CoolProp: closed https://sourceforge.net/p/coolprop/tickets/34/ (r204)
    7. Closed the bugs/issues in https://sourceforge.net/p/coolprop/tickets/35/ (r203)
    8. Resolved memory leak with ECS (r201)

6.11 2.2.5 (revision 199)

- P,h and p,s as inputs solve for almost all fluids under almost all conditions
- Octave modules for 3.6.1 and 3.6.2 now build and run properly for VS build on Windows
- Builds properly on Linux now
- **Bugfixes:**
    1. REFPROP.cpp bug with mixtures (r195)
    2. fixes around critical point (r198)
    3. Ancillaries for R134a updated in the vicinity of critical point

6.12 2.2.4 (revision 192)

- Does not die if pseudo-pure T,P are in the two-phase region
- Fixed bug with dewpoint as an input for dewpoints below 0C
- Added a CoolPropStateClass for elegantly handling inputs - internal codebase will soon transition to this entirely
- Fixed derivatives of drhodp|h and drhodh|p in two-phase region
- Improved ancillary equations for Siloxanes (were terrible!)
- Improved ancillary equations for Ethanol
- Improved ancillary equations for SES36
- Tmin is now an option for CoolProp and REFPROP fluids - ex: Props("REFPROP-MDM","Tmin") or Props("MDM","Tmin")
- T_hp is now faster than REFPROP
- Added Excel 2003 Add-in for CoolProp - not clear it is working though
- Improved the Distro builder

6.13 2.2.3 (revision 172)

- Added Ethylene, SF6, Ethanol, Methane, Ethane, n-Butane, Isobutane
- x(h,p) is much faster due to the avoidance of a lot of saturation routine calls
- x(p,Q) is about 200 times faster!!
- Added Quality 'Q' as an output
- Fixed properties for Air
- Fixed ancillaries for Siloxanes

6.14 2.2.2 (revision 169)

- Added MATLAB wrappers and compiled versions on Windows to batch
- Added plots to check solvers for (T,p) and (h,p) in subcooled liquid and superheated vapor regions

6.15 2.2.1 (revision 166)

- Added the fluid SES36
- HAProps added to CoolProp wrapper and added to Excel addin
- When using pseudo-pure fluid, saturation density are calculated based on solving for density given T,P and guess value given by ancillary for density
- Improved saturated vapor ancillary for SES36
- Changed default names: R717 -> Ammonia, R744 -> CarbonDioxide, R290 -> Propane

6.16 2.2.0 (revision 164)

- Added the Siloxanes (MM,MDM,MD2M,MD3M,MD4M,D4,D5,D6)
- Added a script that will build all the parts (Excel DLL, Python, MATLAB, etc.) and upload to Sourceforge
- Very-alpha code for use of CoolProp in Modelica
- Enthalpy and pressure are valid inputs for Brine fluids
- Added support for quantities package in Python code (If you provide quantities.Quantity instance to CoolProp.CoolProp.Props, the units will be converted to the default units for CoolProp; Default units can be obtained by calling get_index_units(iParam) as a std::string; If a string for the desired output units is passed to Props the units will be converted to the output units)
- Internals of CoolProp changed again, added a function called IProps that uses the integer indices for the input terms as well as the fluids - significant speedup. This is mostly for use with CoolProp.State.State in Python although the same principle can be used elsewhere
- Bug fixes for ECS

6.17 2.1.0 (revision 154)

- Added the fluids Hydrogen, Oxygen, and Helium
- Added the output term 'accentric' to get the accentric factor of the fluid
- Checking of input temperature now yields errors for bad temperatures below fluid min temp
- Fixed T(h,p) and T(s,p) in two-phase region
- Fixed Units on surface tension to N/m

6.18 2.0.6 (revision 147)

- Fixed entropy of humid air at above-atmospheric pressure (Typo in RP-1485)
- Added specific heat of humid air
- Changes to setup.py so that it will not build if cython version < 0.17 which is a requirement due to the use of STL containers
- Changes to plot module to allow for showing right after plot

6.19 2.0.5 (revision 143)

- Fixed wetbulb and dewpoint calculations - works correctly now
- Added wrappers for MATLAB and Octave to subversion code - not included in source distro

6.20 2.0.4 (revision 132)

- Fixed density for subcooled liquid
- Fixed setup.py for OSX (I think)
- Using cython for wrapping of CoolProp module
- CoolProp module - removed T_hp and h_sp - use Props instead
- Added IceProps function to HumidAirProps
- Added and fixed CO2 transport properties

6.21 2.0.1 (revision 122)

- Implemented the method of Akasaka to calculate the saturation state (works great). H/T to FPROPS for the recommendation
- Fixed the calculations for T(h,p) up to a subcooling of 50 K, works fine in superheated vapor
- Added the ideal-gas specific heat with key of C0

6.22 2.0.0 (revision 107)

- MAJOR revision to the internals of CoolProp
- Entropy added for humid air (Only fully validated at atmospheric pressure)
- Added the fluids R22, R1234yf and the 20 industrial fluids from Lemmon, 2000
- Added ECS model for calculation of transport properties (somewhat experimental)
- Added surface tension for all fluids. Property key is 'I' for surface tension
- **Some functions have been removed in order to better handle errors at the C++ level.** Tcrit(), Tsat() and pcrit() are gone, in Python call Props('R134a','Tcrit') for instance to get Tcrit
- Many other bug fixes.
- Documentation to follow.

6.23 1.4.0 (revision 75)

- Internal codebase rewritten in C++ to allow for better exception handling and function overloading
- All work now happens in CoolProp.cpp (inspired by FPROPS)
- Added 2-D lookup table (temperature and pressure) directly in CoolProp. Enable by calling UseSinglePhaseLUT(1) to turn on, UseSinglePhaseLUT(0) to turn off
- Compiled with the -builtin compilation flag
- Documentation updated for UseSinglePhaseLUT

6.24 1.3.2 (revision 49)

- Added functions to use Isothermal compressibility correlation UseIsothermCompressCorrelation and ideal gas compressibility UseIdealGasEnthalpyCorrelations

6.25 1.3.1 (revision 48)

- Updated documentation
- Added ability to use virial term correlations for Humid air by call to UseVirialCorrelation(1)

6.26 1.3 (revision 41):

- Added pseudo-pure fluid Air using EOS from Lemmon
- Added EOS for ice from IAPWS
- Updated Humid Air Thermo Props to use analysis from ASHRAE RP-1845, though IAPWS-1995 is used throughout for water vapor
- Enable the use of lookup tables for refrigerant saturation properties[call UseSaturationLUT(1) to turn on, and UseSaturationLUT(0) to turn off] Speed up is very significant!

6.27 1.2.2 (revision 35):

- Added some simple cycles for comparison of different working fluids
- Fixed quality calculations to agree with REFPROP

CHAPTER 7

CoolProp

7.1 CoolProp Module

CoolProp.CoolProp.**DerivTerms**(*str Output*, *double T*, *double rho*, *str Fluid*) → double

Call signature:

```
DerivTerms(OutputName, T, rho, Fluid) --> float
```

where `Fluid` is a string with a valid CoolProp fluid name, and `T` and `rho` are the temperature in K and density in kg/m ^{3} . The value `OutputName` is one of the strings in the table below:

OutputName	Description
dpdT	Derivative of pressure with respect to temperature at constant density [kPa/K]
dpdrho	Derivative of pressure with respect to density at constant temperature [kPa/(kg/m^3)]
Z	Compressibility factor [-]
dZ_dDelta	Derivative of Z with respect to reduced density [-]
dZ_dTau	Derivative of Z with respect to inverse reduced temperature [-]
VB	Second virial coefficient [m^3/kg]
dBdT	Derivative of second virial coefficient with respect to temperature [m^3/kg/K]
VC	Third virial coefficient [m^6/kg^2]
dCdT	Derivative of third virial coefficient with respect to temperature [m^6/kg^2/K]
phir	Residual non-dimensionalized Helmholtz energy [-]
dphir_dTau	Partial of residual non-dimensionalized Helmholtz energy with respect to inverse reduced temperature [-]
d2phir_dTau2	Second partial of residual non-dimensionalized Helmholtz energy with respect to inverse reduced temperature [-]
dphir_dDelta	Partial of residual non-dimensionalized Helmholtz energy with respect to reduced density [-]
d2phir_dDelta2	Second partial of residual non-dimensionalized Helmholtz energy with respect to reduced density [-]
d2phir_dDelta_dTau	First cross-partial of residual non-dimensionalized Helmholtz energy [-]
d3phir_dDelta2_dTau	Second cross-partial of residual non-dimensionalized Helmholtz energy [-]
phi0	Ideal-gas non-dimensionalized Helmholtz energy [-]
dphi0_dTau	Partial of ideal-gas non-dimensionalized Helmholtz energy with respect to inverse reduced temperature [-]
d2phi0_dTau2	Second partial of ideal-gas non-dimensionalized Helmholtz energy with respect to inverse reduced temperature [-]
dphi0_dDelta	Partial of ideal-gas non-dimensionalized Helmholtz energy with respect to reduced density [-]
d2phi0_dDelta2	Second partial of ideal-gas non-dimensionalized Helmholtz energy with respect to reduced density [-]
IsothermalCompressibility	Isothermal compressibility [1/kPa]

CoolProp.CoolProp.**DerivTermsU**(*in1*, *T*, *rho*, *fluid*, *units=None*)

Make the DerivTerms function handle different kinds of unit sets. Use kSI or SI to identify your desired unit system. Both input and output values have to be from the same unit set.

CoolProp.CoolProp.**F2K**(*double T_F*) → double

Convert temperature in degrees Fahrenheit to Kelvin

CoolProp.CoolProp.**FluidsList**() → list

Return a list of strings of all fluid names

Returns **FluidsList** : list of strings of fluid names

All the fluids that are included in CoolProp

Notes

Here is an example:

```
In [0]: from CoolProp.CoolProp import FluidsList

In [1]: FluidsList()
```

CoolProp.CoolProp.**IProps**(*long iOutput*, *long iInput1*, *double Input1*, *long iInput2*, *double Input2*, *long iFluid*) → double

This is a more computationally efficient version of the Props() function as it uses integer keys for the input and output codes as well as the fluid index for the fluid. It can only be used with CoolProp fluids. An example of how it should be used:

```
#  These should be run once in the header of your file
from CoolProp.CoolProp import IProps, get_Fluid_index
from CoolProp import param_constants
iPropane = get_Fluid_index('Propane')

#  This should be run using the cached values - much faster !
IProps(param_constants.iP,param_constants.iT,0.8*Tc,param_constants.iQ,1,iPropane)
```

The reason that this function is significantly faster than Props is that it skips all the string comparisons which slows down the Props function quite a lot. At the C++ level, IProps doesn't use any strings and operates on integers and floating point values

CoolProp.CoolProp.**IsFluidType**(*string Fluid*, *string Type*) → bool

Check if a fluid is of a given type

Valid types are:

- `Brine`
- `PseudoPure` (or equivalently `PseudoPureFluid`)
- `PureFluid`

CoolProp.CoolProp.**K2F**(*double T_K*) → double

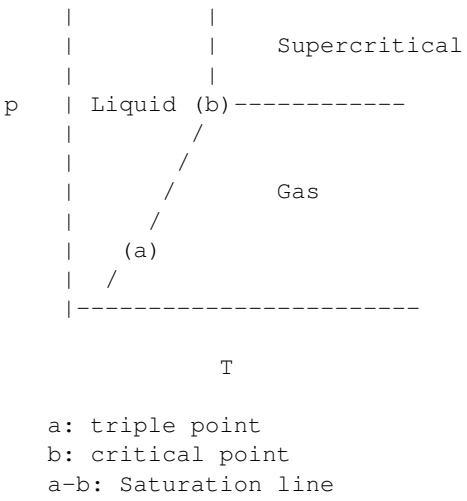
Convert temperature in Kelvin to degrees Fahrenheit

CoolProp.CoolProp.**Phase**(*str Fluid*, *double T*, *double p*) → string

Given a set of temperature and pressure, returns one of the following strings

- Gas
- Liquid
- Supercritical
- Two-Phase

Phase diagram:



CoolProp.CoolProp.**Phase_Tp**(*str Fluid*, *double T*, *double p*) → string

CoolProp.CoolProp.**Phase_Trho**(*str Fluid*, *double T*, *double rho*) → string

CoolProp.CoolProp.**Props**(*str in1*, *str in2*, *in3=None*, *in4=None*, *in5=None*, *in6=None*, *in7=None*)

Call Type #1:

```
Props(Fluid,PropName) --> float
```

Where `Fluid` is a string with a valid CoolProp fluid name, and `PropName` is one of the following strings:

`Tcrit`	Critical temperature [K]
`Treduce`	Reducing temperature [K]
`pcrit`	Critical pressure [kPa]
`rhocrit`	Critical density [kg/m3]
`rhoreduce`	Reducing density [kg/m3]
`molemass`	Molecular mass [kg/kmol]
`Ttriple`	Triple-point temperature [K]
`Tmin`	Minimum temperature [K]
`ptriple`	Triple-point pressure [kPa]
`accentric`	Accentric factor [-]
`GWP100`	Global Warming Potential 100 yr
`ODP`	Ozone Depletion Potential

This type of call is used to get fluid-specific parameters that are not dependent on the state

Call Type #2:

Alternatively, Props can be called in the form:

```
Props(OutputName,InputName1,InputProp1,InputName2,InputProp2,Fluid) --> float
```

where `Fluid` is a string with a valid CoolProp fluid name. The value `OutputName` is either a single-character or a string alias. This list shows the possible values

`OutputName`	**Description**
`Q`	Quality [-]
`T`	Temperature [K]
`P`	Pressure [kPa]
`D`	Density [kg/m3]
`C0`	Ideal-gas specific heat at constant pressure [kJ/kg]
`C`	Specific heat at constant pressure [kJ/kg]
`O`	Specific heat at constant volume [kJ/kg]
`U`	Internal energy [kJ/kg]
`H`	Enthalpy [kJ/kg]
`S`	Entropy [kJ/kg/K]
`A`	Speed of sound [m/s]
`G`	Gibbs function [kJ/kg]
`V`	Dynamic viscosity [Pa-s]
`L`	Thermal conductivity [kW/m/K]
`I` or *SurfaceTension*	Surface Tension [N/m]
`w` or *accentric*	Accentric Factor [-]

The following sets of input values are valid (order doesn't matter):

InputName1	InputName2
T	P
T	D
P	D
T	Q
P	Q
H	P
S	P
S	H

Python Only : InputProp1 and InputProp2 can be lists or numpy arrays. If both are iterables, they must be the same size.

If *InputName1* is *T* and *OutputName* is `I` or `SurfaceTension`, the second input is neglected since surface tension is only a function of temperature

Call Type #3: New in 2.2 If you provide InputName1 or InputName2 as a derived class of Quantity, the value will be internally converted to the required units as long as it is dimensionally correct. Otherwise a ValueError will be raised by the conversion

`CoolProp.CoolProp.`**`PropsU`**(*in1*, *in2*, *in3=None*, *in4=None*, *in5=None*, *in6=None*, *in7=None*)
Make the Props function handle different kinds of unit sets. Use kSI or SI to identify your desired unit system. Both input and output values have to be from the same unit set.

`CoolProp.CoolProp.`**`add_REFPROP_fluid`**(*str FluidName*)
Add a REFPROP fluid to CoolProp internal structure

example:

```
add_REFPROP_fluid("REFPROP-PROPANE")
```

`CoolProp.CoolProp.`**`cair_sat`**(*double T*) → double
The derivative of the saturation enthalpy cair_sat = d(hsat)/dT

`CoolProp.CoolProp.`**`conductivity_background`**(*str Fluid*, *double T*, *double rho*)

`CoolProp.CoolProp.`**`conductivity_critical`**(*str Fluid*, *double T*, *double rho*)

`CoolProp.CoolProp.`**`conformal_Trho`**(*str Fluid*, *str ReferenceFluid*, *double T*, *double rho*) → tuple

`CoolProp.CoolProp.`**`disable_TTSE_LUT`**(*char *FluidName*) → bool

`CoolProp.CoolProp.`**`disable_TTSE_LUT_writing`**(*char *FluidName*) → bool

`CoolProp.CoolProp.`**`enable_TTSE_LUT`**(*char *FluidName*) → bool

`CoolProp.CoolProp.`**`enable_TTSE_LUT_writing`**(*char *FluidName*) → bool

`CoolProp.CoolProp.`**`fromSI`**(*str in1*, *in2=None*, *str in3='kSI'*)
Call fromSI(Property, Value, Units) to convert from SI units to a given set of units. At the moment, only kSI is supported. This convenience function is used inside both the PropsU and DerivTermsU functions.

`CoolProp.CoolProp.`**`get_ASHRAE34`**(*str Fluid*)
Return the safety code for the fluid from ASHRAE34 if it is in ASHRAE34

`CoolProp.CoolProp.`**`get_BibTeXKey`**(*str Fluid*, *str key*) → string
Return the BibTeX key for the given fluid.

The possible keys are

- `EOS`
- `CP0`

- VISCOSITY
- CONDUCTIVITY
- ECS_LENNARD_JONES
- ECS_FITS
- SURFACE_TENSION

BibTeX keys refer to the BibTeX file in the trunk/CoolProp folder

Returns **key, string** :

empty string if Fluid not in CoolProp, "Bad key" if key is invalid

CoolProp.CoolProp.**get_CAS_code**(*string Fluid*) → string
Return a string with the CAS number for the given fluid

CoolProp.CoolProp.**get_Fluid_index**(*str Fluid*) → long
Gets the integer index of the given CoolProp fluid (primarily for use in `IProps` function)

CoolProp.CoolProp.**get_REFPROPname**(*str Fluid*) → string
Return the REFPROP compatible name for the fluid (only useful on windows)

Some fluids do not use the REFPROP name. For instance, ammonia is R717, and propane is R290. You can still can still call CoolProp using the name ammonia or R717, but REFPROP requires that you use a limited subset of names. Therefore, this function that returns the REFPROP compatible name. To then use this to call REFPROP, you would do something like:

```
In [0]: from CoolProp.CoolProp import get_REFPROPname, Props

In [1]: Fluid = 'REFPROP-' + get_REFPROPname('R290')

In [2]: Props('D', 'T', 300, 'P', 300, Fluid)
```

CoolProp.CoolProp.**get_TTSESinglePhase_LUT_range**(*char *FluidName*) → tuple
Get the current range of the single-phase LUT

Returns **tuple of hmin,hmax,pmin,pmax** :

CoolProp.CoolProp.**get_TTSE_mode**(*string fluid*) → str
Get the mode of the TTSE table, one of `"TTSE"` or `"BICUBIC"`

CoolProp.CoolProp.**get_aliases**(*str Fluid*)
Return a comma separated string of aliases for the given fluid

CoolProp.CoolProp.**get_debug_level**()
Return the current debug level as integer

Returns **level** : int

If level is 0, no output will be written to screen, if >0, some output will be written to screen. The larger level is, the more verbose the output will be

CoolProp.CoolProp.**get_errstr**() → string
Return the current error string

CoolProp.CoolProp.**get_fluid_param_string**(*str fluid*, *str param*)

CoolProp.CoolProp.**get_global_param_string**(*str param*)

CoolProp.CoolProp.**get_standard_unit_system**() → int

CoolProp.CoolProp.**isConstant**(*what*)
Get the boolean telling you if the given key matches with a fluid constant.

CoolProp.CoolProp.**isenabled_TTSE_LUT**(*char *FluidName*) → bool

CoolProp.CoolProp.**isenabled_TTSE_LUT_writing**(*char *FluidName*) → bool

CoolProp.CoolProp.**psatL_anc**(*signatures*, *args*, *kwargs*, *defaults*)

CoolProp.CoolProp.**psatV_anc**(*signatures*, *args*, *kwargs*, *defaults*)

CoolProp.CoolProp.**rebuildState**(*d*)

CoolProp.CoolProp.**rhosatL_anc**(*signatures*, *args*, *kwargs*, *defaults*)

CoolProp.CoolProp.**rhosatV_anc**(*signatures*, *args*, *kwargs*, *defaults*)

CoolProp.CoolProp.**set_TTSESat_LUT_size**(*char *FluidName*, *int N*) → bool

CoolProp.CoolProp.**set_TTSESinglePhase_LUT_range**(*char *FluidName*, *double hmin*, *double hmax*, *double pmin*, *double pmax*) → bool

CoolProp.CoolProp.**set_TTSESinglePhase_LUT_size**(*char *FluidName*, *int Np*, *int Nh*) → bool

CoolProp.CoolProp.**set_TTSE_mode**(*char *FluidName*, *char *Value*) → bool
Set the mode of the TTSE table, one of `"TTSE"` or `"BICUBIC"`

CoolProp.CoolProp.**set_debug_level**(*int level*)
Set the current debug level as integer in the range [0,10]

Parameters **level** : int

If level is 0, no output will be written to screen, if >0, some output will be written to screen. The larger level is, the more verbose the output will be

CoolProp.CoolProp.**set_reference_state**(*str FluidName*, **args*)
Accepts one of two signatures:

Type #1:

set_reference_state(FluidName,reference_state)

FluidName The name of the fluid param reference_state The reference state to use, one of

Type #2:

set_reference_state(FluidName,T0,rho0,h0,s0)

`FluidName` The name of the fluid

`T0` The temperature at the reference point [K]

`rho0` The density at the reference point [kg/m^3]

`h0` The enthalpy at the reference point [J/kg]

`s0` The entropy at the reference point [J/kg]

CoolProp.CoolProp.**set_standard_unit_system**(*int unit_system*)

CoolProp.CoolProp.**toSI**(*str in1*, *in2=None*, *str in3='kSI'*)
Call toSI(Property, Value, Units) to convert from a given set of units to SI units. At the moment, only kSI is supported. This convenience function is used inside both the PropsU and DerivTermsU functions.

CoolProp.CoolProp.**viscosity_dilute**(*str Fluid*, *double T*)

CoolProp.CoolProp.**viscosity_residual**(*str Fluid*, *double T*, *double rho*)

7.2 HumidAirProp Module

CoolProp.HumidAirProp.**HAProps**(*str OutputName*, *str Input1Name*, *double Input1*, *str Input2Name*, *double Input2*, *str Input3Name*, *double Input3*) → double

Copyright Ian Bell, 2011 email: ian.h.bell@gmail.com

The function is called like

HAProps('H','T',298.15,'P',101.325,'R',0.5)

which will return the enthalpy of the air for a set of inputs of dry bulb temperature of 25C, atmospheric pressure, and a relative humidity of 50%.

This function implements humid air properties based on the analysis in ASHRAE RP-1845 which is available online: http://rp.ashrae.biz/page/ASHRAE-D-RP-1485-20091216.pdf

It employs real gas properties for both air and water, as well as the most accurate interaction parameters and enhancement factors. The IAPWS-95 formulation for the properties of water is used throughout in preference to the industrial formulation. It is unclear why the industrial formulation is used in the first place.

Since humid air is nominally a binary mixture, three variables are needed to fix the state. At least one of the input parameters must be dry-bulb temperature, relative humidity, dew-point temperature, or humidity ratio. The others will be calculated. If the output variable is a transport property (conductivity or viscosity), the state must be able to be found directly - i.e. make sure you give temperature and relative humidity or humidity ratio. The list of possible input variables are

String	Aliases	Description
T	Tdb	Dry-Bulb Temperature [K]
B	Twb	Wet-Bulb Temperature [K]
D	Tdp	Dew-Point Temperature [K]
P		Pressure [kPa]
V	Vda	Mixture volume [m3/kg dry air]
R	RH	Relative humidity in (0,1) [-]
W	Omega	Humidity Ratio [kg water/kg dry air]
H	Hda	Mixture enthalpy [kJ/kg dry air]
S	Sda	Mixture entropy [kJ/kg dry air/K]
C	cp	Mixture specific heat [kJ/kg dry air/K]
M	Visc	Mixture viscosity [Pa-s]
K		Mixture thermal conductivity [W/m/K]

There are also strings for the mixture volume and mixture enthalpy that will return the properties on a total humid air flow rate basis, they are given by 'Vha' [units of m^3/kg humid air] and 'Cha' [units of kJ/kg humid air/K] and 'Hha' [units of kJ/kg humid air] respectively.

For more information, go to http://coolprop.sourceforge.net

CoolProp.HumidAirProp.**HAProps_Aux**(*str OutputName*, *double T*, *double p*, *double w*) → tuple

Allows low-level access to some of the routines employed in HumidAirProps

Returns tuples of the form `(Value, Units)` where `Value` is the actual value and `Units` is a string that describes the units

The list of possible inputs is

- Baa [First virial air-air coefficient]
- Caaa [Second virial air coefficient]
- Bww [First virial water-water coefficient]
- Cwww [Second virial water coefficient]

- Baw [First cross virial coefficient]
- Caww [Second air-water-water virial coefficient]
- Caaw [Second air-air-water virial coefficient]
- beta_H
- kT
- vbar_ws [Molar saturated volume of water vapor]
- p_ws [Saturated vapor pressure of pure water (>=0.01C) or ice (<0.01 C)]
- f [Enhancement factor]

7.3 Plots Package

7.3.1 CoolProp.Plots Package

7.3.2 CoolProp.Plots.Common Module

class CoolProp.Plots.Common.**BasePlot**(*fluid_ref*, *graph_type*, ***kwargs*)
Bases: object

AXIS_LABELS = {'D': ['Density', '[kg/m^3]'], 'H': ['Enthalpy', '[kJ/kg]'], 'P': ['Pressure', '[kPa]'], 'S': ['Entropy', '[

COLOR_MAP = {'D': 'DarkBlue', 'H': 'DarkGreen', 'Q': 'black', 'P': 'DarkCyan', 'S': 'DarkOrange', 'T': 'Darkred'}

LINE_IDS = {'PS': ['H', 'T', 'D'], 'PT': ['D', 'P', 'S'], 'HS': ['P'], 'TS': ['P', 'D'], 'PD': ['T', 'S', 'H'], 'TD': ['P'], 'PH':

SYMBOL_MAP = {'D': ['$\\rho = ', '$ kg/m^3'], 'H': ['$h = ', '$ kJ/kg'], 'Q': ['$x = ', '$'], 'P': ['$p = ', '$ kPa'], 'S': ['$s

grid(*b=None*, ***kwargs*)

set_axis_limits(*limits*)

show()

title(*title*)

xlabel(*xlabel*)

ylabel(*ylabel*)

7.3.3 CoolProp.Plots.Plots Module

CoolProp.Plots.Plots.**InlineLabel**(*xv*, *yv*, *x=None*, *y=None*, *axis=None*, *fig=None*)
This will give the coordinates and rotation required to align a label with a line on a plot

class CoolProp.Plots.Plots.**IsoLine**
Bases: object

class CoolProp.Plots.Plots.**IsoLines**(*fluid_ref*, *graph_type*, *iso_type*, ***kwargs*)
Bases: CoolProp.Plots.Common.BasePlot

draw_isolines(*iso_range*, *num=None*, *rounding=False*)
Draw lines with constant values of type 'which' in terms of x and y as defined by 'plot'. 'iMin' and 'iMax' are minimum and maximum value between which 'num' get drawn.

There should also be helpful error messages...

get_isolines(*iso_range=[ ]*, *num=None*, *rounding=False*)

This is the core method to obtain lines in the dimensions defined by 'plot' that describe the behaviour of fluid 'Ref'. The constant value is determined by 'iName' and has the values of 'iValues'.

'iValues' is an array-like object holding at least one element. Lines are calculated for every entry in 'iValues'. If the input 'num' is larger than the amount of entries in 'iValues', an internally defined pattern is used to calculate an appropriate line spacing between the maximum and minimum values provided in 'iValues'.

Returns lines[num] - an array of dicts containing 'x' and 'y' coordinates for bubble and dew line. Additionally, the dict holds the keys 'label' and 'opts', those can be used for plotting as well.

CoolProp.Plots.Plots.**PT**(*Ref*, *Tmin=None*, *Tmax=None*, *show=False*, *axis=None*, **args*, ***kwargs*)

Make a pressure-temperature plot for the given fluid

Note `CoolProps.Plots.PT()` will be deprecated in future releases and replaced with `CoolProps.Plots.PropsPlot()`

Parameters **Ref** : str

The given reference fluid

Tmin : float, Optional

Minimum limit for the saturation line

Tmax : float, Optional

Maximum limit for the saturation line

show : bool, Optional

Show the current plot (Default: False)

axis : `matplotlib.pyplot.gca()`, Optional

The current axis system to be plotted to. (Default: create a new axis system)

Examples

```
>>> from CoolProp.Plots import PT
>>> PT('R290', show=True)

>>> from CoolProp.Plots import PT
>>> PT('R290', show=True, Tmin=200, Tmax=300)

>>> from matplotlib import pyplot
>>> fig = pyplot.figure(1)
>>> ax = fig.gca()
>>> PT('R290', show=True, axis=ax)
```

CoolProp.Plots.Plots.**Ph**(*Ref*, *Tmin=None*, *Tmax=None*, *show=False*, *axis=None*, **args*, ***kwargs*)

Make a pressure-enthalpy plot for the given fluid

Note `CoolProps.Plots.Ph()` will be deprecated in future releases and replaced with `CoolProps.Plots.PropsPlot()`

Parameters **Ref** : str

The given reference fluid

Tmin : float, Optional

Minimum limit for the saturation line

Tmax : float, Optional

Maximum limit for the saturation line

show : bool, Optional

Show the current plot (Default: False)

axis : `matplotlib.pyplot.gca()`, Optional

The current axis system to be plotted to. (Default: create a new axis system)

Examples

```
>>> from CoolProp.Plots import Ph
>>> Ph('R290', show=True)

>>> from CoolProp.Plots import Ph
>>> Ph('R290', show=True, Tmin=200, Tmax=300)

>>> from matplotlib import pyplot
>>> fig = pyplot.figure(1)
>>> ax = fig.gca()
>>> Ph('R290', show=True, axis=ax)
```

`CoolProp.Plots.Plots.`**`Prho`**(*Ref*, *Tmin=None*, *Tmax=None*, *show=False*, *axis=None*, **args*, ***kwargs*)

Make a pressure-density plot for the given fluid

Note `CoolProps.Plots.Prho()` will be deprecated in future releases and replaced with `CoolProps.Plots.PropsPlot()`

Parameters **Ref** : str

The given reference fluid

Tmin : float, Optional

Minimum limit for the saturation line

Tmax : float, Optional

Maximum limit for the saturation line

show : bool, Optional

Show the current plot (Default: False)

axis : `matplotlib.pyplot.gca()`, Optional

The current axis system to be plotted to. (Default: create a new axis system)

Examples

```
>>> from CoolProp.Plots import Prho
>>> Prho('R290', show=True)

>>> from CoolProp.Plots import Prho
>>> Prho('R290', show=True, Tmin=200, Tmax=300)
```

```
>>> from matplotlib import pyplot
>>> fig = pyplot.figure(1)
>>> ax = fig.gca()
>>> Prho('R290', show=True, axis=ax)
```

class CoolProp.Plots.Plots.**PropsPlot**(*fluid_ref*, *graph_type*, ***kwargs*)

Bases: CoolProp.Plots.Common.BasePlot

Create graph for the specified fluid properties

Parameters **fluid_ref** : str

The refence fluid

graph_type : str

The graph type to be plotted

axis : matplotlib.pyplot.gca(), Optional

The current axis system to be plotted to. Default: create a new axis system

fig : matplotlib.pyplot.figure(), Optional

The current figure to be plotted to. Default: create a new figure

Examples

```
>>> from CoolProp.Plots import PropsPlot
>>> plt = PropsPlot('Water', 'Ph')
>>> plt.show()

>>> plt = PropsPlot('n-Pentane', 'Ts')
>>> plt.set_axis_limits([-0.5, 1.5, 300, 530])
>>> plt.draw_isolines('Q', [0.1, 0.9])
>>> plt.draw_isolines('P', [100, 2000])
>>> plt.draw_isolines('D', [2, 600])
>>> plt.show()
```

Note: See the online documentation for a the available fluids and graph types

---

draw_isolines(*iso_type*, *iso_range*, *num=10*, *rounding=False*)

CoolProp.Plots.Plots.**Ps**(*Ref*, *Tmin=None*, *Tmax=None*, *show=False*, *axis=None*, **args*, ***kwargs*)

Make a pressure-entropy plot for the given fluid

Note CoolProps.Plots.Ps() will be deprecated in future releases and replaced with CoolProps.Plots.PropsPlot()

Parameters **Ref** : str

The given reference fluid

Tmin : float, Optional

Minimum limit for the saturation line

Tmax : float, Optional

Maximum limit for the saturation line

show : bool, Optional

Show the current plot (Default: False)

axis : `matplotlib.pyplot.gca()`, Optional

The current axis system to be plotted to. (Default: create a new axis system)

Examples

```
>>> from CoolProp.Plots import Ps
>>> Ps('R290', show=True)

>>> from CoolProp.Plots import Ps
>>> Ps('R290', show=True, Tmin=200, Tmax=300)

>>> from matplotlib import pyplot
>>> fig = pyplot.figure(1)
>>> ax = fig.gca()
>>> Ps('R290', show=True, axis=ax)
```

`CoolProp.Plots.Plots.`**`Trho`**(*Ref*, *Tmin=None*, *Tmax=None*, *show=False*, *axis=None*, **args*, ***kwargs*)

Make a temperature-density plot for the given fluid

Note `CoolProps.Plots.Trho()` will be deprecated in future releases and replaced with `CoolProps.Plots.PropsPlot()`

Parameters **Ref** : str

The given reference fluid

Tmin : float, Optional

Minimum limit for the saturation line

Tmax : float, Optional

Maximum limit for the saturation line

show : bool, Optional

Show the current plot (Default: False)

axis : `matplotlib.pyplot.gca()`, Optional

The current axis system to be plotted to. (Default: create a new axis system)

Examples

```
>>> from CoolProp.Plots import Trho
>>> Trho('R290', show=True)

>>> from CoolProp.Plots import Trho
>>> Trho('R290', show=True, Tmin=200, Tmax=300)

>>> from matplotlib import pyplot
>>> fig = pyplot.figure(1)
>>> ax = fig.gca()
>>> Trho('R290', show=True, axis=ax)
```

`CoolProp.Plots.Plots.`**`Ts`**(*Ref*, *Tmin=None*, *Tmax=None*, *show=False*, *axis=None*, **args*, ***kwargs*)

Make a temperature-entropy plot for the given fluid

Note `CoolProps.Plots.Ts()` will be deprecated in future releases and replaced with `CoolProps.Plots.PropsPlot()`

Parameters **Ref** : str

The given reference fluid

Tmin : float, Optional

Minimum limit for the saturation line

Tmax : float, Optional

Maximum limit for the saturation line

show : bool, Optional

Show the current plot (Default: False)

axis : `matplotlib.pyplot.gca()`, Optional

The current axis system to be plotted to. (Default: create a new axis system)

Examples

```
>>> from CoolProp.Plots import Ts
>>> Ts('R290', show=True)

>>> from CoolProp.Plots import Ts
>>> Ts('R290', show=True, Tmin=200, Tmax=300)

>>> from matplotlib import pyplot
>>> fig = pyplot.figure(1)
>>> ax = fig.gca()
>>> Ts('R290', show=True, axis=ax)
```

`CoolProp.Plots.Plots.`**`drawIsoLines`**(*Ref*, *plot*, *which*, *iValues=[]*, *num=0*, *show=False*, *axis=None*)

Draw lines with constant values of type 'which' in terms of x and y as defined by 'plot'. 'iMin' and 'iMax' are minimum and maximum value between which 'num' get drawn.

Note `CoolProps.Plots.drawIsoLines()` will be depreciated in future releases and replaced with `CoolProps.Plots.IsoLines()`

Parameters **Ref** : str

The given reference fluid

plot : str

The plot type used

which : str

The iso line type

iValues : list

The list of constant iso line values

num : int, Optional

The number of iso lines (Default: 0 - Use iValues list only)

show : bool, Optional

Show the current plot (Default: False)

axis : `matplotlib.pyplot.gca()`, Optional

The current axis system to be plotted to. (Default: create a new axis system)

Examples

```
>>> from matplotlib import pyplot
>>> from CoolProp.Plots import Ts, drawIsoLines
>>>
>>> Ref = 'n-Pentane'
>>> ax = Ts(Ref)
>>> ax.set_xlim([-0.5, 1.5])
>>> ax.set_ylim([300, 530])
>>> quality = drawIsoLines(Ref, 'Ts', 'Q', [0.3, 0.5, 0.7, 0.8], axis=ax)
>>> isobars = drawIsoLines(Ref, 'Ts', 'P', [100, 2000], num=5, axis=ax)
>>> isochores = drawIsoLines(Ref, 'Ts', 'D', [2, 600], num=7, axis=ax)
>>> pyplot.show()
```

`CoolProp.Plots.Plots.`**`drawLines`**(*Ref*, *lines*, *axis*, *plt_kwargs=None*)

Just an internal method to systematically plot values from the generated 'line' dicts, method is able to cover the whole saturation curve. Closes the gap at the critical point and adds a marker between the two last points of bubble and dew line if they reach up to critical point. Returns the an array of line objects that can be used to change the colour or style afterwards.

`CoolProp.Plots.Plots.`**`hs`**(*Ref*, *Tmin=None*, *Tmax=None*, *show=False*, *axis=None*, **args*, ***kwargs*)

Make a enthalpy-entropy plot for the given fluid

Note `CoolProps.Plots.hs()` will be deprecated in future releases and replaced with `CoolProps.Plots.PropsPlot()`

Parameters **Ref** : str

The given reference fluid

Tmin : float, Optional

Minimum limit for the saturation line

Tmax : float, Optional

Maximum limit for the saturation line

show : bool, Optional

Show the current plot (Default: False)

axis : `matplotlib.pyplot.gca()`, Optional

The current axis system to be plotted to. (Default: create a new axis system)

Examples

```
>>> from CoolProp.Plots import hs
>>> hs('R290', show=True)
```

```
>>> from CoolProp.Plots import hs
>>> hs('R290', show=True, Tmin=200, Tmax=300)
```

```
>>> from matplotlib import pyplot
>>> fig = pyplot.figure(1)
>>> ax = fig.gca()
>>> hs('R290', show=True, axis=ax)
```

7.3.4 CoolProp.Plots.PsychChart Module

This file implements a psychrometric chart for air at 1 atm

class CoolProp.Plots.PsychChart.**EnthalpyLines**(*H_values*)
Bases: object

plot(*ax*)

class CoolProp.Plots.PsychChart.**HumidityLabels**(*RH_values*, *h*)
Bases: object

plot(*ax*)

class CoolProp.Plots.PsychChart.**HumidityLines**(*RH_values*)
Bases: object

plot(*ax*)

class CoolProp.Plots.PsychChart.**PlotFormatting**
Bases: object

plot(*ax*)

class CoolProp.Plots.PsychChart.**SaturationLine**
Bases: object

plot(*ax*)

7.3.5 SimpleCycles Module

CoolProp.Plots.SimpleCycles.**EconomizedCycle**(*Ref*, *Qin*, *Te*, *Tc*, *DTsh*, *DTsc*, *eta_oi*, *f_p*, *Ti*, *Ts_Ph='Ts'*, *skipPlot=False*, *axis=None*, ***kwargs*)

This function plots an economized cycle, on the current axis, or that given by the optional parameter *axis*

Required parameters:

- Ref : Refrigerant [string]
- Qin : Cooling capacity [W]
- Te : Evap Temperature [K]
- Tc : Condensing Temperature [K]
- DTsh : Evaporator outlet superheat [K]
- DTsc : Condenser outlet subcooling [K]
- eta_oi : Adiabatic efficiency of compressor (no units) in range [0,1]
- f_p : fraction of compressor power lost as ambient heat transfer in range [0,1]
- Ti : Saturation temperature corresponding to intermediate pressure [K]

Optional parameters:

- Ts_Ph : 'Ts' for a Temperature-Entropy plot, 'Ph' for a Pressure-Enthalpy
- axis : An axis to use instead of the active axis
- skipPlot : If True, won't actually plot anything, just print COP

CoolProp.Plots.SimpleCycles.**SimpleCycle**(*Ref*, *Te*, *Tc*, *DTsh*, *DTsc*, *eta_a*, *Ts_Ph='Ph'*, *skipPlot=False*, *axis=None*)

This function plots a simple four-component cycle, on the current axis, or that given by the optional parameter *axis*

Required parameters:

- Ref : A string for the refrigerant
- Te : Evap Temperature in K
- Tc : Condensing Temperature in K
- DTsh : Evaporator outlet superheat in K
- DTsc : Condenser outlet subcooling in K
- eta_a : Adiabatic efficiency of compressor (no units) in range [0,1]

Optional parameters:

- Ts_Ph : 'Ts' for a Temperature-Entropy plot, 'Ph' for a Pressure-Enthalpy
- axis : An axis to use instead of the active axis
- skipPlot : If True, won't actually plot anything, just print COP

CoolProp.Plots.SimpleCycles.**TwoStage**(*Ref*, *Q*, *Te*, *Tc*, *DTsh*, *DTsc*, *eta_oi*, *f_p*, *Tsat_ic*, *DTsh_ic*, *Ts_Ph='Ph'*, *prints=False*, *skipPlot=False*, *axis=None*, ***kwargs*)

This function plots a two-stage cycle, on the current axis, or that given by the optional parameter *axis*

Required parameters:

- Ref : Refrigerant [string]
- Q : Cooling capacity [W]
- Te : Evap Temperature [K]
- Tc : Condensing Temperature [K]
- DTsh : Evaporator outlet superheat [K]
- DTsc : Condenser outlet subcooling [K]
- eta_oi : Adiabatic efficiency of compressor (no units) in range [0,1]
- f_p : fraction of compressor power lost as ambient heat transfer in range [0,1]
- Tsat_ic : Saturation temperature corresponding to intermediate pressure [K]
- DTsh_ic : Superheating at outlet of intermediate stage [K]

Optional parameters:

- Ts_Ph : 'Ts' for a Temperature-Entropy plot, 'Ph' for a Pressure-Enthalpy
- prints : True to print out some values
- axis : An axis to use instead of the active axis
- skipPlot : If True, won't actually plot anything, just print COP

7.4 State Module

class CoolProp.State.**State**(*str Fluid*, *dict StateDict*, *double xL=-1.0*, *Liquid=None*, *phase=None*)

A class that contains all the code that represents a thermodynamic state

The motivation for this class is that it is useful to be able to define the state once using whatever state inputs you like and then be able to calculate other thermodynamic properties with the minimum of computational work.

Let's suppose that you have inputs of pressure and temperature and you want to calculate the enthalpy and pressure. Since the Equations of State are all explicit in temperature and density, each time you call something like:

```
h = Props('H','T',T,'P',P,Fluid)
s = Props('S','T',T,'P',P,Fluid)
```

the solver is used to carry out the T-P flash calculation. And if you wanted entropy as well you could either intermediately calculate `T`, `rho` and then use `T`, `rho` in the EOS in a manner like:

```
rho = Props('D','T',T,'P',P,Fluid)
h = Props('H','T',T,'D',rho,Fluid)
s = Props('S','T',T,'D',rho,Fluid)
```

Instead in this class all that is handled internally. So the call to update sets the internal variables in the most computationally efficient way possible

Parameters **Fluid, string** :

StateDict, dictionary :

The state of the fluid - passed to the update function

phase, string :

The phase of the fluid, it it is known. One of `Gas`,``Liquid``,``Supercritical``,``TwoPhase``

xL, float :

Liquid mass fraction (not currently supported)

Liquid, string :

The name of the liquid (not currently supported)

Fluid

Liquid

PFC

PFC: CoolProp.CoolProp.PureFluidClass

Phase(*self*) → long

Returns an integer flag for the phase of the fluid, where the flag value is one of iLiquid, iSupercritical, iGas, iTwoPhase

These constants are defined in the phase_constants module, and are imported into this module

Prandtl

The Prandtl number (cp*mu/k) [-]

Props(*self*, *long iOutput*) → double

Q

The quality [-]

T
The temperature [K]

Tsat
The saturation temperature (dew) for the given pressure, in [K]

copy(*self*) → State
Make a copy of this State class

cp
The specific heat at constant pressure [kJ/kg]

cv
The specific heat at constant volume [kJ/kg]

dpdT

get_MM(*self*) → double
Get the mole mass [kg/kmol] or [g/mol]

get_Q(*self*) → double
Get the quality [-]

get_T(*self*) → double
Get the temperature [K]

get_Tsat(*self*, *double Q=1*) → double
Get the saturation temperature, in [K]

Returns `None` if pressure is not within the two-phase pressure range

get_cond(*self*) → double
Get the thermal conductivity, in [kW/m/K]

get_cp(*self*) → double
Get the specific heat at constant pressure [kJ/kg]

get_cp0(*self*) → double
Get the specific heat at constant pressure for the ideal gas [kJ/kg]

get_cv(*self*) → double
Get the specific heat at constant volume [kJ/kg]

get_dpdT(*self*) → double

get_h(*self*) → double
Get the specific enthalpy [kJ/kg]

get_p(*self*) → double
Get the pressure [kPa]

get_rho(*self*) → double
Get the density [kg/m^3]

get_s(*self*) → double
Get the specific enthalpy [kJ/kg/K]

get_speed_sound(*self*) → double
Get the speed of sound [m/s]

get_subcooling(*self*) → double
Get the amount of subcooling below the saturation temperature corresponding to the pressure, in [K]

Returns `None` if pressure is not within the two-phase pressure range

get_superheat (*self*) → double
Get the amount of superheat above the saturation temperature corresponding to the pressure, in [K]

Returns `None` if pressure is not within the two-phase pressure range

get_u (*self*) → double
Get the specific internal energy [kJ/kg]

get_visc (*self*) → double
Get the viscosity, in [Pa-s]

h
The specific enthalpy [kJ/kg]

hasLiquid

is_CPFluid

k
The thermal conductivity, in [kW/m/K]

p
The pressure [kPa]

phase

rho
The density [kg/m^3]

s
The specific enthalpy [kJ/kg/K]

set_Fluid (*self*, *str Fluid*)

speed_test (*self*, *int N*)

subcooling
The amount of subcooling below the saturation temperature corresponding to the pressure, in [K]

Returns `None` if pressure is not within the two-phase pressure range

superheat
The amount of superheat above the saturation temperature corresponding to the pressure, in [K]

Returns `None` if pressure is not within the two-phase pressure range

u
The internal energy [kJ/kg]

update (*self*, *dict params*, *double xL=-1.0*)
Parameters params, dictionary

A dictionary of terms to be updated, with keys equal to single-char inputs to the Props function, for instance `dict(T=298, P = 101.325)` would be one standard atmosphere

update_Trho (*self*, *double T*, *double rho*)
Just use the temperature and density directly for speed

Parameters **T: float** :

Temperature [K]

rho: float :

Density [kg/m^3]

update_ph (*self*, *double p*, *double h*)
Use the pressure and enthalpy directly

Parameters **p: float** :

Pressure (absolute) [kPa]

h: float :

Enthalpy [kJ/kg]

visc
The viscosity, in [Pa-s]

class CoolProp.State.**PureFluidClass**

T (*self*) → double

TL (*self*) → double

TV (*self*) → double

cp (*self*) → double

cv (*self*) → double

d2Tdp2_along_sat (*self*) → double

d2hdT2_constp (*self*) → double

d2hdT2_constrho (*self*) → double

d2hdTdp (*self*) → double

d2hdp2_along_sat_liquid (*self*) → double

d2hdp2_along_sat_vapor (*self*) → double

d2hdp2_constT (*self*) → double

d2hdrho2_constT (*self*) → double

d2hdrhodT (*self*) → double

d2pdT2_constrho (*self*) → double

d2pdrho2_constT (*self*) → double

d2pdrhodT (*self*) → double

d2rhodT2_constp (*self*) → double

d2rhodTdp (*self*) → double

d2rhodh2_constp (*self*) → double

d2rhodhdQ (*self*) → double

d2rhodhdp (*self*) → double

d2rhodp2_along_sat_liquid (*self*) → double

d2rhodp2_along_sat_vapor (*self*) → double

d2rhodp2_constT (*self*) → double

d2rhodpdQ (*self*) → double

d2sdT2_constp (*self*) → double

d2sdT2_constrho(*self*) → double

d2sdTdp(*self*) → double

d2sdp2_along_sat_liquid(*self*) → double

d2sdp2_along_sat_vapor(*self*) → double

d2sdp2_constT(*self*) → double

d2sdrho2_constT(*self*) → double

d2sdrhodT(*self*) → double

dTdp_along_sat(*self*) → double

dhdT_constp(*self*) → double

dhdT_constrho(*self*) → double

dhdp_along_sat_liquid(*self*) → double

dhdp_along_sat_vapor(*self*) → double

dhdp_constT(*self*) → double

dhdrho_constT(*self*) → double

dhdrho_constp(*self*) → double

disable_TTSE_LUT(*self*)

dpdT_consth(*self*) → double

dpdT_constrho(*self*) → double

dpdrho_constT(*self*) → double

dpdrho_consth(*self*) → double

drhodT_along_sat_liquid(*self*) → double

drhodT_along_sat_vapor(*self*) → double

drhodT_constp(*self*) → double

drhodp_along_sat_liquid(*self*) → double

drhodp_along_sat_vapor(*self*) → double

drhodp_constT(*self*) → double

dsdT_constp(*self*) → double

dsdT_constrho(*self*) → double

dsdp_along_sat_liquid(*self*) → double

dsdp_along_sat_vapor(*self*) → double

dsdp_constT(*self*) → double

dsdrho_constT(*self*) → double

dsdrho_constp(*self*) → double

enable_TTSE_LUT(*self*)

h(*self*) → double

hL(*self*) → double

hV (*self*) → double

isenabled_TTSE_LUT (*self*) → bool

keyed_output (*self*, *long iOutput*) → double

p (*self*) → double

pL (*self*) → double

pV (*self*) → double

phase (*self*) → long

rho (*self*) → double

rhoL (*self*) → double

rhoV (*self*) → double

s (*self*) → double

sL (*self*) → double

sV (*self*) → double

speed_sound (*self*) → double

update (*self*, *long iInput1*, *double Value1*, *long iInput2*, *double Value2*)

Python Module Index

C