The PYroMat User and Developer Handbook

Second Edition, v2.2.0 and later

Christopher R. Martin
Associate Professor of Mechanical Engineering
Penn State University

April 27, 2022

Contents

Foreword

Because this book is long and has never been professionally edited, it suffers from the kinds of typos and errata that plague any work of its size. That said, I have taken some pains to be certain that the information is accurate. Still, errors do creep in.

Corrections made since the first edition are:

1. Equation 6.9 originally read $p = 3mn\left\langle u^2\right\rangle$, which is an error. It now reads $p = \frac{1}{3}mn\left\langle u^2\right\rangle$.

Nomenclature

These variables and units are restricted to derivations in this document. Because PYroMat has a user-configurable unit system, any of these may be expressed in alternate units.

The following are parameters with units.

Symb.	Units	Description
a	m/s	Speed of sound
c	J/kg/K	Specific heat
e	J/kg	Internal energy
f	J/kg	Free energy
h	J/kg	Enthalpy
k	J/K	Boltzmann constant
M	kg	Mass
N	count	Number of moles
n	m^{-3}	Number density
p	Pa	Pressure
q	J kg^{-1}	Normalized heat addition
R	J kg^{-1} K^{-1}	Gas constant
R_u	J kmol^{-1} K^{-1}	Universal gas const.
s	J/kg/K	Entropy
T	K	Temperature
v	m^3/kg	Specific volume
W	kg/kmol	Molar mass
ρ	$\mathrm{kg/m}^3$	Density

The following are dimensionless parameters.

Symb.	Def.	Description
α	f/RT	ND Free energy
δ	ρ/ρ_c	ND Density
τ	T_c/T	ND Temperature
γ	c_p/c_v	Sp. heat ratio
π	p/p_c	ND Pressure

The following are subscripts used throughout the document.

Symb.	Description
$_c$	Critical point
$_f$	Formation property (e.g. enthalpy of formation)
$_p$	Constant-pressure (sp.heat)
$_t$	Triple point
$_v$	Constant-volume (sp.heat)
$^\circ$	Property at reference pressure

Special constants used in more fundamental calcuations are defined throughout this document. Many are listed explicitly in Table 4.1 in Chapter 4.

Chapter 1

Introduction

PYroMat is a Python-based package for calculating the thermodynamic properties of fluids. That includes liquids, gases, and plasmas. It is written in pure Python, and the core algorithms only depend on the Numpy package for back-end numerical and array support. This document is intended to serve as a reference manual for interpreting, using, or even writing your own PYroMat data sets.

Probably, no reader will need to digest this handbook in its entirity. The goal is to include all the important information in a single portable document. Chatper 2 on Getting Started is probably the one that most users will need, since it describes installation and basic use of the software.

For users who want a review of the thermodynamic properties, this introduction goes into detail on the fundamental definitions for the properties used. First, we talk briefly about the ideas of *primary* and *basic properties*, which PYroMat uses to organize its work on the back-end. Then, we will briefly define each property with a modest attempt to describe its origins and relevance.

For the curious or for developers who may want to add their own substance models, the inner workings of the package are described in some detail in Chapters 5, 6, 7, and 8.

1.1 The Properties

Here, we begin with a brief review of the thermodynamic properties of liquids and gases that PYroMat calculates. For a detailed development of these properties, the reader should consult an introductory text on thermodynamics[1].

In the author's opinion, contemporary undergraduate texts on thermodynamics do the reader a disservice by ignoring the deeply intuitive (and terribly important) underpinnings provided by the kinetic theory of gases. Kinetic theory and thermodynamics have a complicated relationship because they were developed in parallel (often in tragic contention with one another), but the limits imposed by many of the common assumptions (such as idea or perfect gas) cannot be deeply understood without kinetic theory. To the interested reader, the author would recommend Jeans's 1949 introductory text[2]. It is old, but it is brief, accessible, inexpensive, and approaches the subject only requiring that the reader have a grasp of calculus, geometry, and introductory mechanics.

PYroMat is primarily concerned with the *thermodynamic* properties listed in Table 1.1. There is no single set of units used to express these properties since PYroMat uses a user-configurable unit system. The units systems given in Table 1.1 are itemized in Table 1.2. All of Chapter 4 is devoted to describing PYroMat's treatment of units.

1.1.1 Primary Properties

The theory of thermodynamics can make equal use of any set of properties to discover the others, but it is almost always prudent to define a pair or more of core properties that are standard for defining the thermodynamic state. After version 2.2.0, PYroMat has been quite flexible in how it allows users to specify the thermodynamic state, but computational mysteries are likely to plague those users who do not have some appreciation for what is going on in the background. Some

[1] For example, Cengel and Boles, *Thermodynamics*, McGraw Hill. Any edition will do.

[2] James Jeans, *Introduction to the Kinetic Theory of Gases*, Cambridge University Press, 1949.

Table 1.1: Thermodynamic properties, their symbols, and their unit systems

Symbol	In-Code	Units	Description
T	`T`	T	Temperature
p	`p`	P	Pressure
ρ	`d`	M / V	Density
x	`x`	-	Quality
R	`R`	E / M / T	Ideal gas constant
W	`mw`	Ma / Mo	Molecular weight
e	`e`	E / M	Internal energy
h	`h`	E / M	Enthalpy
s	`s`	E / M / T	Entropy
c_p	`cp`	E / M / T	Constant-pressure specific heat
c_v	`cv`	E / M / T	Constant-volume specific heat
γ	`gam`	-	Specific heat ratio

Table 1.2: Classes of units, their entry in `pm.config`, and their defaults.

Unit	`config` entry	Default	Description
E	`unit_energy`	kJ	Energy
L	`unit_length`	m	Length
M	`unit_matter`	kg	Matter (molar or mass)
Ma	`unit_mass`	kg	Mass
Mo	`unit_molar`	kmol	Molar
P	`unit_pressure`	bar	Pressure
t	`unit_time`	s	Time
T	`unit_temperature`	K	Temperature
V	`unit_volume`	m^3	Volume

approaches produce more floating point error and are noticeably slower than others. Why?

Every thermodynamic model supported by PYroMat provides a means for calculating the substance's many properties in terms of temperature and/or density. For example, an ideal gas model allows entropy (see sec. 6.1.5) to be calculated explicitly in terms of temperature and pressure. While this theoretically provides a means for calculating temperature from entropy and pressure, $T(s,p)$, there is not guaranteed to be any such explicit relationship - in fact, in general, there is not. That means that specifying a state in this way is theoretically sound, but it requires a computationally expensive numerical iteration routine to invert $s(T,p)$. Chapter 8 discusses this process in some detail.

The *primary properties* for a thermodynamic model are the two that are used in the back-end to do the most basic thermodynamic state calculations. The performance will always be best if users can write their code to use these properties when they are available. Table 1.3 lists the current PYroMat substance classes and their primary properties.

For most problems of engineering relevance, temperature and pressure (T,p) are the favorite since both are readily measured, and both are of immediate importance to fluid machinery design. Fortunately, all of the common properties of ideal gases can be conveniently constructed directly from these two properties. Unfortunately, that is not the case with non-ideal fluids.

It is intuitive that when gases are compressed into tighter spaces, the distance between the molecules becomes a vitally important parameter for predicting the substance's properties. Pressure is not a convenient metric of that distance, but density is. For this reason, nearly all non-ideal gas properties are modeled in terms of (T,ρ), and pressure has to be calculated indirectly. This creates a substantial amount of work for the design engineer who may not wish to delve into the details of how properties are calculated.

1.1.2 Basic Properties

Though the most efficient means for specifying a thermodynamic state will always be in the primary property pair for whatever substance

Table 1.3: The PYroMat substance classes and their primary property pairs

Desription	Class	Prim.
Shomate Eqn.	`ig`	(T,p)
NASA poly.	`ig2`	(T,p)
IG mixture	`igmix`	(T,p)
Span & Wagner	`mp1`	(T,ρ)

is being used, the entire point of using a package like PYroMat is to handle this kind of computational meta-awareness automatically. To that end, it is also useful to think in terms of *basic properties*:

- temperature, T,
- pressure, p,
- density, ρ,
- specific volume, v,
- quality, x.

States may always be defined in terms of any pair of these with little additional computational cost.

Other properties, like

- internal energy, e,
- enthalpy, h,
- entropy, s,

can also be used to specify the thermodynamic state, but the user should be aware of their limitations. These may only be given one-at-a-time (i.e. they must be paired with a basic property), and specifying them forces PYroMat to use a slower back-end algorithm with inherent numerical precision limitations. For example, as of version 2.2.0, specifying entropy and enthalpy together is not supported by any of the property methods.

1.1.3 Density, ρ

It is convenient to begin our discussion of properties with density since it is the easiest to define.

Density is the quantity of a substance per unit volume it occupies in space. It can be described as a number of molecules or mass per unit space. When described with molar units, it is usually called concentration, but PYroMat does not make that distinction.

When the unit volume is very tiny, this is a poorly defined quantity. For example, as the volume shrinks to be about the same size as the distance between molecules, the density one might measure would vary hugely as individual molecules entered and left the region of space. However, as the volume grows within a thermodynamically homogeneous region, the ratio of matter to volume converges to a consistent well-defined value.

This introduces the idea that the study of thermodynamics is essentially a careful study of averages over a large population of mechanical bodies. It is important that we study a quantity of a substance large enough that the extremes of individual molecules do not weigh heavily in our measurements. On the other hand, our measurements must consider a region of space small enough that the properties that interest us do not vary significantly. We may consider a region to be thermodynamically homogeneous if it can be divided into two equal smaller regions that exhibit identical thermodynamic properties.

The density of the gas in molecules per unit volume is

$$n = \frac{N}{V}. \tag{1.1}$$

where V is the volume of the region, and N is the number of molecules. These numbers are extremely large, and early in the history of chemistry, there was no way to know the true number of molecules in a sample anyway. So, the use of molar quantities was of great utility.

The density, in molar units, is a slight variation on n,

$$\overline{\rho} = \frac{N}{N_a V} = \frac{\overline{N}}{V}, \tag{1.2}$$

where N_a is Avagadro's number, and $\overline{N}$ is the number of moles in the sample. As we will see in Section 4.5.4, other molar units exist, but

the same formula applies. In this way, the density of molecules may be expressed either in a literal count per unit volume, n, or in a number of moles per unit volume, $\overline{\rho}$.

In mass units, the density is merely multiplied by the molecular weight of the species,

$$\begin{aligned} \rho &= m_0 \frac{N}{V} \\ &= \frac{\overline{N} W}{V} = \frac{NW}{N_a V} \end{aligned} \tag{1.3}$$

when m_0 is the mass of a single molecule, and W is the more commonly used molecular weight in atomic mass units.

Since ρ is not available in the ASCII character set, PYroMat uses `d` to represent density.

1.1.4 Specific volume, v

Specific volume is the volume occupied by a unit of matter, and is calculated as the inverse of density.

$$v \equiv \frac{1}{\rho}. \tag{1.4}$$

Even though density is the property with the clearer fundamental definition, specific volume is mathematically identical to other intensive properties because it is formulated per unit matter.

PYroMat does not calculate specific volume directly, but deals in density instead. The user is called on to calculate `v=1/d` when specific volume is needed.

1.1.5 Temperature, T

Temperature scales were originally developed as quantitative means for describing hot and cold, but they were developed before we had more physical descriptions for their meaning. After all, the existence of molecules and atoms was still being hotly debated while temperature scales were already in wide scientific and engineering use.

We now understand temperature to be an observable measure of the molecular translational kinetic energy of a substance. In a gas, the molecules are free to translate through space, and temperature is proportional to their kinetic energy. In a liquid or solid, the same energy manifests as molecular vibration within the confines of the intermolecular forces.

For a gas,

$$\langle \frac{1}{2} m u^2 \rangle = \frac{3}{2} kT, \tag{1.5}$$

Here, m is the mass of an individual molecule, $\langle u^2 \rangle$ is the mean square of velocity, k is the Boltzmann constant, and T is the temperature in absolute units.

The ITS-90 temperature scale establishes an international standard for the definition of temperature in terms of the triple points of various pure substances. Many of the property models included in PYroMat were formulated when the previous ITPS-68 was the international temperature scale, but the changes were so minute that the uncertainties in the properties dominate[1, p.10]. As a result, they are treated as interchangeable for the purposes of this handbook.

1.1.6 Pressure, p

Pressure is the static force exerted by a fluid on a surface. It is quantified in force per unit area of the surface, and it always acts normal to the surface facing into the surface (away from the fluid).

In a gas, pressure is due to the impact of molecules on the surface. Pressure may be increased by increasing either their velocity (temperature) or their quantity (density). Because these effects are linear in a gas, this leads to the famous ideal gas relationship between density, temperature, and pressure.

In a liquid or solid, intermolecular forces that cause pressure are persistent instead of momentary (due to collisions in a gas). Under these conditions, even slight changes in intermolecular spacing causes huge changes in pressure, making the substance quite stiff in comparison to gases. In this case as well, increasing temperature makes molecules vibrate more violently in their equilibrium with each other,

so at a consistent average density, the force applied to a neighboring surface will increase. This is why substances appear to expand as they are heated.

1.1.7 Internal Energy, e

Energy can be stored in a fluid in many ways. In gases, for example, the molecules translate with great speed (see temperature), the molecules vibrate and rotate, and there is incredible energy stored in the chemical bonds of molecules. However, all of these are dwarfed by the energy contained in the forces binding the nucleus of each atom.

We account for *all* of these energies simultaneously with the internal energy, e, which has units energy per matter (e.g. J/kg). It is neither practical nor necessary to tally all of these energies in an absolute fashion. Especially since most applications will have no release of nuclear energy, it is practical, instead, to describe how the substance's energy changes relative to some reference state. This is not unlike defining a reference height for gravitational potential energy calculations in classical mechanics. The choice is arbitrary and has no bearing on the result, but it can drastically simplify the calculations.

Therefore, we say that the internal energy, e is the sum of vibrational, rotational, translational, chemical, and nuclear energies contained in a thermodynamically homogeneous unit matter, subtracted by the same sum at some reference state.

Some readers will be scandalized that PYroMat uses e in favor of the traditional u for internal energy. This is a genuine (if futile) attempt to untangle the web of contradictory variable use between thermodynamics and fluid mechanics. u is reserved for velocity, v for volume, and e for energy.

1.1.8 Enthalpy, h

When a fluid of any kind is flowing, it caries its internal energy with it, but it also does mechanical work as it flows. The mechanical work done by a moving fluid is $p\mathrm{d}V$, where p is the pressure exerted and $\mathrm{d}V$ is a differential volume displaced by the fluid. If we were to imagine that the volume were displaced while the fluid is expanding or contracting, the

same idea applies to a fluid whether it is flowing or not. Per unit mass, this can be expressed as $p\mathrm{d}v$ (when v is specific volume). Integrated over an isobaric (constant-pressure) process, this becomes simply pv.

It is a matter of convenience for engineers and physicists that deal with fluid power, that we define enthalpy as the sum of internal energy and fluid power,

$$h \equiv e + pv = e + \frac{p}{\rho}. \tag{1.6}$$

This term appears naturally when energy balances are applied to open systems (ones where flow is moving through the system). Internal energy accounts for all of the energy stored by the molecules in a fluid, and enthalpy additionally includes the fluid's capacity to do mechanical work as it flows.

Of course, most processes aren't *actually* isobaric, so extra terms will tend to appear (like $\mathrm{d}h - v\mathrm{d}p$), but that is a discussion for later.

Enthalpy is most commonly used in the analysis of any flowing fluid such as in heat exchangers, combustors and burners, chemical reactors, compressors, turbines, valves, etc... Even though it is derived from a property that might be argued to be more "fundamental," enthalpy is usually tabulated as a primary property because it is so useful.

Just like internal energy, a substance's enthalpy is defined relative to an arbitrary reference state. However, since internal energy and enthalpy share a common definition, their reference states *must* be consistent with one another. Furthermore, for any data set intended for the analysis of chemical reactions, the enthalpies of chemically related species (such as H_2, O_2, and H_2O) *must* be defined consistently so that the changes in molecule internal energy due to chemical reaction can be properly accounted for. As a result, there are certain *reference species* for which arbitrary reference state enthalpies are selected.

Since the PYroMat multi-phase models permit only pure substances, they do not need rigorously defined reference states. However, the ideal gas models are defined with this in mind to permit chemical action. See the ideal gas chapter (Chapter 6) for a detailed discussion on the enthalpy of formation.

1.1.9 Entropy, s

The idea of entropy is born with the Clausius statement of the Second Law of Thermodynamics, which says that a reversible cyclic series of processes that include heat transfer must obey

$$\oint \frac{\delta q}{T} \underset{rev.}{=} 0, \tag{1.7}$$

where T is the temperature of the substance and δq is the addition of heat. It is important to emphasize that this is only true of processes that result in a continuous cycle where the fluid ends at the same thermodynamic state from which it began (like in an engine or refrigeration cycle).

The first law would be satisfied by any cycle where the work and heat transfer merely summed to zero, but the Clausius equality implies something deeper. Heat is not a property of the substance, but heat added reversibly in ratio with the temperature consistently returns to zero when the substance returns to its original thermodynamic state. That implies the existence of a new (and terribly important) property. Clausius termed that property entropy,

$$\mathrm{d}s \underset{\text{rev.}}{\equiv} \frac{\delta q}{T} \tag{1.8}$$

From the first law of thermodynamics, $\delta q = \mathrm{d}e + p\mathrm{d}v$, and we have a way to calculate entropy from the other properties

$$T\mathrm{d}s = \mathrm{d}e + p\mathrm{d}v \tag{1.9}$$

$$= \mathrm{d}h - v\mathrm{d}p \tag{1.10}$$

Just like with temperature, there is a parallel (mathematically consistent) definition of entropy from statistical mechanics. Boltzmann found that Entropy can also be calculated from the probability of the substance obtaining a thermodynamically equivalent state. The probability of gas molecules spontaneously exhibiting a specific set of positions and velocities is astronomically low, but there is an emense quantity of dissimilar positions and velocities that would give the gas precisely the same temperature, internal energy, pressure, density, and

other properties. Meanwhile, the probability that gas molecules will spontaneously assemble themselves into a highly ordered crystal is astronomically low. When that probability is very high, so is the entropy. When that probability is very low, so is the entropy.

Just like internal energy, it is theoretically possible to tally all of the possible states and calculate the probability of each, but that task is neither practical nor necessary. Instead, it is convenient to define the entropy as zero at some reference state and then deal merely in changes in entropy as defined by (1.8). In the case of ideal gases, the entropy is known to be precisely zero at absolute zero temperature, but some multi-phase models (like water) set entropy (and internal energy) of the liquid phase to zero at the triple point.

1.1.10 Specific Heats, c_p c_v

Of the properties so far defined, only temperature, pressure, and density can be directly measured. The specific heats are vitally important to the study of a substance, because they can be conveniently directly measured and the other properties calculated from them.

Specific heat, c, is the amount of heat per mass of a substance, δq, required to obtain a small increase in temperature, δT,

$$c \equiv \frac{\delta q}{\delta T}. \tag{1.11}$$

Especially when dealing with gases, we have to be more specific because substances have a tendency to expand when they are heated. One measures a different value for specific heat depending on whether or not expansion is allowed.

For any process, energy will be conserved, so

$$\delta q = \mathrm{d}e + p\mathrm{d}v. \tag{1.12}$$

Here, δq is heat added per mass of the substance, $\mathrm{d}e$ is the change in internal energy, p is the pressure, and $\mathrm{d}v$ is the change in volume per unit mass.

When heat is applied in such a manner that the substance's volume is constant, $\mathrm{d}v = 0$, the mechanical work is zero, and all of the heat is

stored as internal energy.

$$\begin{aligned}\delta q|_v &= \mathrm{d}e \\ &= \left(\frac{\partial e}{\partial T}\right)_v \mathrm{d}T\end{aligned} \tag{1.13}$$

Thus, the constant-volume specific heat is, by definition,

$$c_v \equiv \left(\frac{\delta q}{\delta T}\right)_v = \left(\frac{\partial e}{\partial T}\right)_v . \tag{1.14}$$

In this process, the substance's pressure will rise (sometimes sharply) as it is heated in a rigid container.

When one considers addition of heat under constant pressure, a trick application of the chain rule for the term, pv, lets us transition the differential on volume into a differential on pressure. The definition for enthalpy (1.6) appears naturally.

$$\begin{aligned}\delta q &= \mathrm{d}e + p\mathrm{d}v + v\mathrm{d}p - v\mathrm{d}p \\ &= \mathrm{d}(e + pv) - v\mathrm{d}p\end{aligned} \tag{1.15}$$

When heat is added while pressure is constant (like in an atmospheric gas), $\mathrm{d}p = 0$, and

$$\begin{aligned}\delta q|_p &= \mathrm{d}h \\ &= \left(\frac{\partial h}{\partial T}\right)_p \mathrm{d}T\end{aligned} \tag{1.16}$$

Thus, constant-pressure specific heat is, by definition,

$$c_p \equiv \left(\frac{\delta q}{\delta T}\right)_p = \left(\frac{\partial h}{\partial T}\right)_p . \tag{1.17}$$

Chapter 2

Getting started

In this chapter, we discuss everything you should need to get your own installation of PYroMat working and providing properties.

2.1 Installation

PYroMat is distributed primarily through the Python package index (https://pypi.org/), but it can also be downloaded and installed "manually" if you have Python's setuptools installed. PYroMat is written in plain Python, so there is no need to compile to binaries. All data are encoded in `json` format (https://docs.python.org/3/library/json.html), and the configuration files are executable python code. The result is that all of the code, data, and configuration files are in human readable text using only the ASCII character set.

2.1.1 Prerequisites

Obviously, a Python interpreter needs to be installed on your system, but PYroMat is deliberately designed to minimize the number of prerequisites. Only Numpy version 1.7 or later is required.

It is worth noting that PYroMat was originally developed in Python 2, but support for Python 3 was designed in from the start. As of January 2020, Python 2 was officially depreciated, and PYroMat testing on Python 2 halted. On a number of popular Linux systems, Python

2 is still the system default, though, so users should take note. If your system has both Python 2 and Python 3 installed, it may be necessary to replace `python` in the instructions below with `python3` to specify the version.

2.1.2 Installation with `pip`

Python uses a package called `pip` (https://pip.pypa.io) to manage its own software. You can use it to automatically download and install PYroMat and its dependencies with a single command.

In an Anaconda installation start the "Anaconda Prompt" to bring up a terminal. See https://www.anaconda.com/ for more information.

In Windows with a system-wide Python interpreter, bring up a command prompt or type `cmd` in the "run" field to bring up the prompt. You may need to run the terminal as an administrator. If so, right click on the icon and open using the "run as administrator" option in the menu.

In Linux or MacOS bring up a terminal any way you like. In many popular installations `Ctrl+Alt+T` works by default.

First, it is important to make sure your python package manager is installed and updated. A Python installation without `pip` is ususual, but they do still appear from time to time. If you do not have `pip` on your system, follow the directions from the Python packaging authority to get it up and working: https://pip.pypa.io/en/stable/installation/.

On just about all systems, entering the following commands in a terminal will update `pip` and install PYroMat.

```
$ python -m pip install --upgrade pip
$ python -m pip install --upgrade pyromat
```

Note that you don't need to type the `$`; it represents the command line prompt. On a Windows system, it might appear `C:\>`, on most Unix-like terminals, it may appear `user@machine:~\$`. Your prompt may look a little different, and that's OK. If Python is already correctly installed these commands should get you going.

What are these commands doing? The first part, `python -m pip`, is executing Python's `pip` module with the options you pass next. Next, `install --upgrade`, tells `pip` that you want to install something over the internet from the python package index (https://pypi.org/), and that if there is a newer version of an existing package available, it should be upgraded. Finally, the last argument tells `pip` what to install. So, the first line is telling `pip` to upgrade itself, and the second line is actually installing `pyromat`.

It should be noted that many Linux systems still use Python 2 by default, so you may want to specify Python 3 specifically. In most systems, this will make certain that the installation is in the Python 3 interpreter and not Python 2.

```
$ python3 -m pip install --upgrade pip
$ python3 -m pip install --upgrade pyromat
```

Installing in system-wide Python installations has gone out of popularity in recent years, but if that's what you're doing, you may need to run `pip` as an administrator. Most installations these days are in virtual environments or some other needlessly complicated setup that obfuscates the whole thing, so if you're doing this the old-school global way, you have my respect, and I want to support you.

On a Windows system, you will need to restart the prompt as a priviledged user (administrator). Usually, you can right-click and select "run as administrator" when you start the command prompt.

On a Linux, Mac, or Unix system, you may need to run as a super user, so you should use

```
$ sudo -H python -m pip install --upgrade pip
$ sudo -H python -m pip install --upgrade pyromat
```

The `-H` switch is recommended by the Python package index to be certain that the `root` user's home directory is used during install.

2.1.3 Manual installation with `git`

So-called manual installation is also quite easy using `git` if you already have Python's `setuptools` (https://setuptools.readthedocs.io/en/latest/) installed and udpated.

Just clone the `git` repository, then navigate into the root directory and run the `setup.py` installation script. This is the stage where an error will be generated if there is a problem with `setuptools`.

```
$ git clone https://github.com/chmarti1/pyromat.git
$ cd pyromat
$ python setup.py install
```

2.1.4 Manual installation from Sourceforge

This method has largely fallen out of favor, but it is still a perfectly valid way to perform an installation. Like the GitHub method, this method only works if you have `setuptools` (https://setuptools.readthedocs.io/en/latest/) installed and updated.

Download the latest version of PYroMat from https://sourceforge.net/projects/pyromat/. Select whichever compression type that suits you (e.g. zip, bz2, gzip). Then, extract the package, creating a `pyromat` directory.

Bring up a command prompt (see above for how to do that on your system), and navigate to the extracted directory using the appropriate `cd` commands (https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/cd).

Finally, execute the Python setup script.

```
$ cd /path/to/your/dir
$ python setup.py install
```

2.2 Using PYroMat

This section assumes that users already have a basic familiarity with the Python command line. For users that are just getting started, try sampling the official Python Tutorial (https://docs.python.org/3/tutorial/index.html).

First, open a command prompt (terminal) and start Python in interactive mode.

```
$ python3  # On my system, I need to specify 3
Python 3.6.9 (default, Jan 26 2021, 15:33:00)
```

```
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for
    more information.
>>>
```

2.2.1 Importing

Import PYroMat like any other package.

```
>>> import pyromat as pm
```

It automatically loads its modules, and automatically seeks out and loads data.

Once loaded, there are only three functions in the base PYroMat package that most users will ever need: `get()`, `search()`, and `info()`. These are tools that interact with the PYroMat data collection to retrieve data class instances and print information about them. All of the rest of the functionality is offered by the class instances themselves.

2.2.2 Retrieving substance data

PYroMat identifies substances by a unique string called the substance identifier (ID). The substance ID string for ideal gas nitrogen is `'ig.N2'`. The `ig` prefix indicates that the substance model belongs to PYroMat's ideal gas collection. The rest of the string is the substance's chemica formula using Hill notation.

Hill notation removes the ambiguity in how a chemical formula is represented by mandating that all substances with the same atomic composition will be represented by the same strings. The atomic contents are listed in order of carbon, hydrogen, and then all others in alphabetical order.

By deliberately omitting all chemical structure from the notation, there will be collisions between compounds with the same atomic composition in dissimilar arrangements. So far, these collisions are rare in the PYroMat data collection, and they are addressed by appending an underscore and an index. For example, the CNN radical and methanetetraylbis-amidogen both have the same chemical formula, so the former has the ID string, `ig.CN2`, and the latter is listed under `ig.CN2_1`.

Given an ID string, the `get` function returns the corresponding class instance. This instance provides all of the property methods. This example calculates the ideal gas enthalpy of nitrogen at 492 K.

```
>>> n2 = pm.get('ig.N2')
>>> n2.h(T=492)
array([202.68455864])
```

In this example, the variable, `n2`, is an ideal gas class instance with all the methods (like `h()`) needed to calculate its properties.

2.2.3 Searching for substances with `search()`

There are about 1,000 unique substance models in PYroMat version 2.2.0, so manually skimming through the list is not a very efficient means of finding the species you want. The `search()` function returns sets of substances that match the criteria supplied. Those sets can either be used directly, or the `info()` function can display more information (see below).

Searching is done by specifying keywords to the `search()` function. Each one adds a new criterion that further narrows the search. The results are

The **name** keyword is used to search for an exact match with part of the substance ID or for a case-insensitive match with one of the substance's names. For example,

```
>>> pm.search(name='Acetylene')
{<ig2, ig.C2F2>, <ig2, ig.C2HCl>, <ig2, ig.C2HF>, ...
>>> pm.search(name='C2H')
{<ig2, ig.C2HCl>, <ig2, ig.C2HF>, <ig2, ig.C2HN>, ...
```

These show how a name string can be used to match part of one of a substance's common name *or* part of the substance ID string.

The **pmclass** is the name of the class that is used to implement the data model for each substance. As of version 2.2.0, there are four classes implemented: `ig`, `ig2`, `igmix`, and `mp1`. These are described in detail in sections 6.2.1, 6.2.2, 6.2.3, and 7.1.

The **collection** keyword allows users to specify the collection to which species must belong. This forces the substance id string to begin

with "`collection.`" For example `collection='ig'` forces all of the substances returned to belong to the ideal gas collection.

The **InChI**[?] and **CAS**[?] substance identifiers can also be used to narrow the search. Since these are unique substance identifiers, they will only return multiple entries when the same substance is represented in multiple collections. To specify these, use `inchi=...` or `cas=...` as keyword arguments. It is important to keep in mind that as of version 2.2.0, not all entries have had these values added to the data, so some results may not come back as expected.

Probably the most powerful of the search tools, the **contains** keyword allows users to specify the atomic contents of the substance. Values may be the string name of an element, or lists of element name strings, or dictionaries with element names and their precise amounts. For example, these queries return all substances (1) with any amount of iron, (2) with any amount of bromine and carbon, and (3) with two hydrogen, one oxygen, and any nonzero amount of carbon.

```
>>> pm.search(contains='Fe')
{<ig, ig.Br4Fe2>, <ig, ig.FFe>, <ig2, ig.H2FeO2>, ...
>>> pm.search(contains=['Br','C'])
{<ig, ig.CBrF3>, <ig, ig.CBr4>, <ig, ig.CBr>, ...
>>> pm.search(contains={'H':2, 'O':1, 'C':None})
{<ig2, ig.C2H2O>, <ig2, ig.CH2O>}
```

Because the `search()` function returns a Python `set` instance, applications that demand more nuanced search operations can use the set operations to combine the results of separate searches. For example, to find substances that contain two hydrogen *or* three carbon,

```
>>> h2 = pm.search(contains={'H':2})
>>> c3 = pm.search(contains={'C':3})
>>> h2.union(c3)
{<ig, ig.H2P>, <ig2, ig.C6H2>, <ig2, ig.CH2Cl2>, ...
```

For more tips and tricks with sets, see the Python documentation https://docs.python.org/3/tutorial/datastructures.html#sets.

2.2.4 Finding substances with `info()`

When called without any arguments, the `info()` function lists all of the substances available, and it lists all of the property methods available

for each. That probably won't be useful for most users, so the function also passes any keyword arguments to `search()` internally to narrow down the set.

```
>>> pm.info(contains=['Br','C'])
  PYroMat
Thermodynamic computational tools for Python
version: 2.2.0
--------------------------------------------------------------------------------
 ID       : class : name                  : properties
--------------------------------------------------------------------------------
 ig.CBr   :  ig   : Bromomethylidyne      : T p d v cp cv gam e h s mw R
 ig.CBr4  :  ig   : Carbon tetrabromide   : T p d v cp cv gam e h s mw R
 ig.CBrF3 :  ig   : Bromotrifluoromethane : T p d v cp cv gam e h s mw R
 ig.CBrN  :  ig   : Cyanogen bromide      : T p d v cp cv gam e h s mw R
```

Alternately, an iterable of substance instances (like the one generated by `search()` may be passed directly to `info()`.

The table shows each substance listed by its ID string and one of its common names (if one is included in the data). There is also a list of the properties currently offered in the data model.

If a substance ID string, a single substance instance, or a set (or other iterable) with only one entry is passed, the `info()` function prints detailed information on that substance.

```
>>> pm.info('ig.N2')
***
Information summary for substance: "ig.N2"
***

    N
     2

             Names : Nitrogen
                     Nitrogen gas
                     N2
                     UN 1066
                     UN 1977
                     Dinitrogen
                     Molecular nitrogen
                     Diatomic nitrogen
                     Nitrogen-14
  Molecular Weight : 28.01348
        CAS number : 7727-37-9
      InChI string : InChI=1S/N2/c1-2
        Data class : ig2
       Loaded from : /home/chris/Documents/pyromat/src/pyromat/data/ig2/N2.hpd
      Last updated : 21:11 April 20, 2022

The supporting data for this object were adapted from: B. McBride, S.
Gordon, M. Reno, "Coefficients␣for␣Calculating␣Thermodynamic␣and␣Transport
Properties␣of␣Individual␣Species," NASA Technical Memorandum 4513, 1993.
```

2.2.5 In-line documentation

While it is also designed to run efficiently in scripts, all aspects of PYroMat were designed with ease of use from the command line in

mind. Most users will first learn PYroMat through the command line and then go on to write scripts that automate their calculations.

With that in mind, every class instance, every method, and every module has in-line documentation that can be accessed using Python's built-in `help()` function. For example, try typing:

```
>>> help(n2)
>>> help(n2.h)
>>> help(pm)
```

2.3 Property interface

The property methods that belong to the many substance class instances use flexible arguments that are as standardized as is practical. Details about the individual properties and the theory behind them is included in chapters 6 and 7, but there are some general rules that apply to all substances in PYroMat.

2.3.1 Property method arguments

With a few special exceptions, a thermodynamic state can be specified with any two properties except specific heats. For example,

```
>>> n2 = pm.get('ig.N2')
>>> T = 452.
>>> p = 14.
>>> n2.s(T=T, p=p)
array([6.49072181])
>>> n2.s(T=T, d=n2.d(T=T, p=p))
array([6.49072181])
```

Or, with multi-phase data,

```
>>> n2 = pm.get('mp.N2')
>>> n2.s(T=T, p=p)
array([6.48696146])
>>> n2.s(T=T, d=n2.d(T=T, p=p))
array([6.48696146])
>>> n2.s(T=100, x=0.5)
array([4.18094321])
```

The last line uses quality to specify a 50/50 mixture of liquid and vapor nitrogen at 100 K.

This works with all basic properties (see section 1.1.2) T, p, ρ, v, and x (multi-phase only). Other properties like e, h, and s, may be specified as well, but only one at a time with a basic property. For example, PYroMat does not support property evaluations with h and s simultaneously. Instead s *or* h may be specified with one of the basic properties like T.

```
>>> n2.h(T=100,x=0.5)
array([7.27884341])
>>> n2.h(s=4.18090744, T=100)
array([7.27884313])
>>> n2.T(s=4.18090744, h=7.27884313)
  ...
pyromat.utility.PMParamError: Properties may not be
   specified together: s, h
```

2.3.2 Default values

It is not unusual that users may want vaguely defined properties. For example, "what is the specific heat of water?" That question has an infinite number of answers depending on the state of the water, so a property method really can't answer it. On the other hand, when users want properties without specifying a state, they usually mean "...at standard conditions."

All property methods apply default primary property values when properties are unspecified. For example,

```
>>> h2o = pm.get('mp.H2O')
>>> h2o.cp()
array([4.18131499])
```

tells the user that the specific heat of liquid water is about 4.18 kJ/kg/K at 273.15 K and 1.01325 bar. That default temperature and pressure can be changed by setting the `def_T` and `def_p` parameters in `pm.config`. See chapter 3 for more information on configuring PYroMat.

When only one property is specified, the second property will revert to a default value. If temperature is specified, then pressure reverts to

Table 2.1: Inverse property methods and their availability by class

	ig	ig2	mp1
T_h	☑	☑	☑
T_s	☑	☑	☑
d_s	☐	☐	☑
p_s	☑	☑	☐

its default. If any other property is specified, then temperature reverts to its default and pressure is unspecified. For example,

```
>>> h2o.cp(T=540)  # T=540K, p=1.01325bar
array([1.99666454])
>>> h2o.cp(p=0.01) # T=273.15K, p=0.01bar
array([1.87429701])
>>> h2o.cp(d=.01)  # T=273.15K, d=.01kg/m3
array([1.87870465])
```

Note that the default values are specified in whatever units PYroMat has been configured to use. By default, PYroMat uses K, kJ, kg, bar, m^3. If the unit system is changed, the default values should also be changed to reflect the intended values in the new unit system. See chapter 4 for more information on units and chapter 3 for more information on configuring PYroMat.

2.3.3 Inverse methods

Before version 2.2.0, the interface only accepted *basic properties* (see section 1.1.2) T, p, ρ, and x. To handle cases where users had secondary properties like entropy or enthalpy, methods like `T_s` were created to calculate "temperature from entropy" and one other property. Originally, `T_s`, `p_s`, `d_s`, `T_h`, and other *inverse property* methods were specific to the class and the property. The new interface introduced in version 2.2.0 obsoleted these methods, but they are still available for reverse compatibility. New codes should not use these methods, and they are scheduled to be removed in a future major release.

Table 2.3.3 shows the historical inverse property methods that are still available on the basic classes.

2.3.4 Tips and tricks

There are some guidelines that users can use to obtain dramatically better performance out of PYroMat.

Avoid calculating property values in a for loop. Instead, construct property values as arrays, lists, tuples, or other iterables and pass them to a single property method call. For example,

```
>>> import numpy as np
>>> import pyromat as pm
>>> h2o = pm.get('mp.h2o')
>>> ###### DON'T DO THIS ######
>>> h = []
>>> for T in np.linspace(300,1000,101):
...     h.append(h2o.h(T))
...
>>> ###### Do this instead ######
>>> T = np.linspace(300,1000,101)
>>> h = h2o.h(T)
```

On many systems, the first code segment can take four or five seconds to run! The second code segment consistently takes a small fraction of a second to run and makes for much cleaner code.

PYroMat is written to assume that all inputs and outputs are multi-dimensional arrays. Simple floating point scalars are taken to be special cases. Programming numerical codes like property evaluation in plain Python has a steep numerical penalty, but much of that can be regained by leaning on Numpy's (https://numpy.org) compiled back-end for efficient numerical methods on arrays.

Ideal gases are faster than the `mp1` model. Ideal gases only have to evaluate a polynomial. The multi-phase `mp1` model requires multiple parallel polynomials with computationally expensive exponential terms. Single equation-of-state multi-phase models sacrifice computational speed in favor of a single model that works well in liquid, gas, near critical, and super-critical states.

Prefer temperature and density when working with multi-phase substances. The `mp1` class calculates all properties (including pressure) in terms of temperature and density. When another combination of properties is specified (e.g. temperature and pressure) an

iterative routine has to run first to isolate temperature and density. That is why the example above takes so long to run; it requires two iterative algorithms run in series. When a series of properties are needed at a given state, it is much faster to calculate the density first and pass temperature and density to all of the successive property methods.

Using the `state()` method is faster than separate calls to properties. In all classes, the `state()` method minimizes the number of back-end polynomial evaluations to calculate the properties at a state. If a user needs more than two properties, chances are good that it is worth it to just go ahead and calculate them all. Multi-phase properties especially require the calculation of a number of intermediate parameters that can be re-used in successive property evaluations.

The `mp1` property methods can return quality too. When running a property method like `T()`, it is often a good idea to go ahead and calculate the quality by setting the optional keyword `quality=True`. All of these routines have to calculate the saturation properties anyway, so one might as well save the redudnant steps.

```
>>> T,x = h2o.T(s=4.331, p=1.01325, quality=True)
>>> print(T,x)
[373.12429581] [0.50004124]
```

Chapter 3

Configuration

There are parameters that most users will never need to change, but that some users may find irritating or even limiting. For example, no unit system ever seems to make everyone happy. There is a configuration system that allows users to alter settings that are honored by the PYroMat substance model classes.

3.1 The `pm.config` instance

The PYroMat configuration system is implemented by a custom class instance called `config`. It mimics a dictionary in that it has a series of string parameters that have values associated with them. Evoking the configuration object prints its contents in a table.

```
>>> import pyromat as pm
>>> pm.config
     config_file : ['/home/chris/Documents/PYroMat/
        pyromat/defaults.py']
  config_verbose : False
         dat_dir : ['/home/chris/Documents/PYroMat/
            pyromat/data']
 dat_exist_fatal : False
   dat_overwrite : True
   dat_recursive : True
     dat_verbose : False
    ...
```

Unlike a dictionary, the `PMConfig` instance enforces certain rules that are specific to each of the configuration entries.

- Configuration entries can be neither created nor deleted.
- The value of a read-only entry (marked "R" in Table 3.1) cannot be changed.
- The values of an appendable entry (marked "A" in Table 3.1) cannot be overwritten. Instead, new values are appended to the end of a list of values.
- Entry values must be of an appropriate data type.

Table 3.1 lists the configuration parameters, their hard-coded initial values when the configuration instance is first created, and a brief description for each. Entries to which special rules apply are marked with an (A) for appendable or (R) for read-only.

Table 3.1: Configuration parameters, their rules, and their descriptions. Parameters that are appendable are marked with (A), and parameters that are read-only are marked with (R). Paths are referenced relative to the PYroMat installation directory and not the user working directory.

Entry (Rule)	**Initial**	**Description**
`config_file` (A)	`['./config.py']`	String file paths to load for configuration values
`config_verbose`	`False`	Print descriptions of the configuration load process?
`dat_dir` (A)	`['./data']`	String paths to directories to scan for `.hpd` files
`dat_exist_fatal`	`False`	Raise an exception if there are two definitions for the same substance id?
`dat_overwrite`	`True`	Overwrite older substance ids with newer ones?
`dat_recursive`	`True`	Recurse into sub-directories when looking for data files?
`dat_verbose`	`False`	Print descriptions of the data load process?

Table 3.1: continued...

Entry (Rule)	Initial	Description
def_T	298.15	Default temperature in units, unit_temperature
def_oob	numpy.nan	Value to use when the state is out-of-bounds
def_p	1.01325	Default pressure in units, unit_pressure
error_verbose	True	Is verbose PYroMat error printing enabled?
install_dir (R)	'./'	Path to the PYroMat installation
reg_dir (A)	['./registry']	A list of directories to scan for .py files containing PYroMat classes
reg_exist_fatal	False	Raise an exception if there are two definitions for the same class?
reg_overwrite	True	Overwrite older class definitions with new ones?
reg_verbose	False	Print descriptions of the registry class discovery process?
unit_energy	'kJ'	Unit string for energy
unit_force	'N'	Unit string for force
unit_length	'm'	Unit string for length
unit_mass	'kg'	Unit string for mass
unit_matter	'kg'	Unit string for matter
unit_molar	'kmol'	Unit string for mole count
unit_pressure	'bar'	Unit string for pressure
unit_temperature	'K'	Unit string for temperature
unit_time	's'	Unit string for time
unit_volume	'm3'	Unit string for volume
version (R)	<version>	The PYroMat installation version string
warning_verbose	True	Is verbose PYroMat warning printing enabled?

3.1.1 Making temporary changes to config

Configuration entry values can be retrieved and written like dictionary values. In this example, we check the version string and change the temperature units to Farenheit.

```
>>> import pyromat as pm
>>> pm.config['version']
'2.2.0'
>>> pm.config['unit_temperature'] = 'F'
```

Parameters that are appendable do not behave the same way. When they are written to, they accumulate new values. For example,

```
>>> pm.config['config_file'] = '/etc/pyromat.py'
>>> pm.config['config_file']
['/home/chris/Documents/PYroMat/pyromat/config.py', '
   /etc/pyromat.py']
```

Observe that instead of overwriting the previous configuration file, the new value was appended to the list.

Undoing a configuration change can be done using the `restore_default()` method. When it is used as an entry method, it only applies to that entry.

```
>>> pm.config['unit_temperature']
'F'
>>> pm.config.restore_default('unit_temperature')
>>> pm.config['unit_temperature']
'K'
```

The "default" values honored by the `restore_default()` method are hard-coded, so if there is a flaw somewhere in the configuration process, the system can be restored to functional settings by calling `restore_default()` with no argument. Then, all of the parameters will be reset to their defaults.

In the next section, we will discuss how to write configuration files that override these defaults. They will be automatically loaded when PYroMat is imported, but the `config.load()` method will manually repeat the configuration load process.

3.2 Configuration files

To make changes to the configuration that persist when PYroMat is reloaded, values should be written to configuration files. PYroMat configuration files are Python scripts that are expected to define variables with the same names as the configuration parameters. Any variables

created that are recognized as valid PYroMat configuration parameters are read into the configuration. Any variables that are unrecognized or that have values with invalid types result in a `PMParamError` exception.

Loading all of its configuration files is the very first thing PYroMat does when it is loaded by the Python import system. First, `pm.config` is initialized with hard-coded initial values listed in Table 3.1. Even if all other aspects of the configuration process fail, the system will still be able to function with these basic parameters.

Then, the configuration load process begins. Configuration files are read one-by-one in the order they are listed in `config_file` until they have all been loaded. By default, the only path in `config_file` is `./config.py`, located in the PYroMat installation directory, but each time a configuration file is read in, new configuration files can be added. In this way, configuration files can add more configuration files. The loading algorithm protects users are protected against infinite loads by checking for files that reference each other or themselves.

The `config.py` file located in the base installation directory is heavily commented with instructions for administrators to make their own changes. Many administrators may want to add a configuration file in `/etc/` or in users' home directory somewhere. The example below shows a `config.py` file that admins might want to place in the PYroMat install directory to allow users to apply their own settings and write their own data models without a virtual environment.

This is an example of a configuration script that might appear in `config.py`.

```
# This adds two new configuration files.
# The first is a global configuration file in /etc.
# The second is in a hidden directory in the user's
# home directory.  The order is important.  Placing
# the user's settings last means that their options
# will overwrite the global options.
config = ['/etc/pyromat.py', '~/.pyromat/config.py']

# These add registry and data directories in the
# user's home directories.
reg_dir = '~/.pyromat/registry'
dat_dir = '~/.pyromat/data'
```

```
# These entries reconfigure PYroMat to use the
# imperial system of units.  These entries
# could appear anywhere in the configuration
# process.
unit_temperature = 'F'
unit_length = 'ft'
unit_volume = 'ft3'
unit_pressure = 'psi'
unit_energy = 'BTU'
unit_matter = 'lb'
unit_force = 'lb'
unit_molar = 'lbmol'
```

For more information on the units, see Chapter 4. For more information on the registry, data files, and the load process, see Chapter 5.

Chapter 4

Units

PYroMat handles a wide variety of units automatically, but interested users are also welcome to use the back-end algorithms for their own purposes. All of the core substance methods deal in their own native units, but the `pyromat.units` module contains the tools that are responsible for converting to and from the units specified in the `pyromat.config` system (see Chapter 3).

A list of all currently available units is available by typing

```
>>> import pyromat as pm
>>> help(pm.units)
```

To obtain information specific to one of the unit classes (length, for example), type

```
>>> help(pm.units.length)
```

The unit systems used by the property methods and their corresponding configuration entries are listed in Tables 1.1 and 1.2.

This chapter infuses a little history along with the information PYroMat users are likely to find useful when interacting with the unit conversion system.

4.1 Unit definitions

The Bureau International des Poids et Mesures (BIPM) International Committee for Weights and Measures (CIPM) is the entity responsible

for the definition of the fundamental units on which the SI system is based. Nearly all other units in wide use are now derived from the SI units in some way, so rigorous treatment of their formal definition is worth some attention. There are countless web-based unit conversion calculators, but few use sufficient care in the derivation of their conversion factors to be trusted for serious work.

The second (time), meter (length), kilogram (mass), ampere (electrical current), kelvin (temperature), mole (quantity), and candela (luminous intensity) are the fundamental units in terms of which all other measures are derived. Contemporary definitions for these units are constructed so that certain important physical constants (like Plank's or Boltzmann's constants) may be represented exactly with finite precision, so the conversion factors used in PYroMat are, by definition, exact.

Before the turn of the twentieth century, the United States and the British governments had adopted the international yard and avoirdupois pound. This move re-defined the pound and the yard (and all those based on them) in terms of the meter and kilogram using an exact relationship with finite precision. The contemporary definitions for those fundamental units are constructed from highly stable natural phenomena may be independently reproduced in a laboratory, but this idea is quite recent. Abandoning the tradition of defining units from a standard bar or mass did not happen until the latter half of the twentieth century. [2]

This establishes a web of precise definitions of units for US customary, imperial, and SI systems. However, there are still popular measures (like the inch water column for pressure or scf for quantity of a gas) that depend on establishing so-called "standard" conditions whose precise definitions are often neglected (and are far from standard).

For example, water and mercury column measures for pressure depend on the density of a fluid and the local strength of gravity, both of which are subject to change. So-called "standard" pressure is often 1atm (1.01325bar), but is precisely 1bar for the NIST-JANAF data. Worse still, standard temperature usually refers to 273.15K (0°C), but the NIST-JANAF tables use 298.15K (25°C), and there are some engineering applications that still use 70°F. There are at least a half-dozen

definitions for the calorie, which are construed from the specific heat of water at different conditions. These inconsistencies are far too severe to ignore, so great care is required when pulling data from multiple sources into a single database.

4.2 Setup

To maintain transparency in regard to the standards used to derive unit conversions, the PYroMat units system includes a `setup` function that is responsible for constructing all of the unit conversion routines. Users can inspect the constants it defines, and users may change them at any time by re-calling the `setup` function with new arguments. Its behavior is fully documented and can be found by evoking `help` as below.

```
>>> import pyromat as pm
>>> pm.units.setup(Tstd=273.15, pstd=1.01325, \
...     g=9.80665, dh2o = 999.972, dhg=13595.1)
>>> help(pm.units.setup)
```

It should be emphasized that there is a difference between the standard parameters used to construct the PYroMat unit system and the default parameters determined by `pm.config`. The standard conditions are only used to construct the unit conversion system, they are always given in the units documented below, and they are ignored outside of the `units` module. On the other hand, the default parameters are used in property methods when arguments are omitted, and they are interpreted in whatever units are currently configured. See the configuration chapter (Chapter 3) for more information.

Standard temperature and pressure, `Tstd` and `pstd`, *must* be entered in units of Kelvin and bar respectively (regardless of the units configured in `pm.config`). The standard atmosphere was set to precisely 1.013 25 bar by the BIPM General Conference on Weights and Measures (CGPM) in 1901 [3], and has been almost universally adopted. The default standard temperature is less universal, but is usually 273.15 K, the freezing point of water at one atmosphere of pressure.

The CGPM did not establish a standard acceleration due to gravity for the purpose of weights and measures until 1954. It is set to be

precisely 9.80665 m s^{-2} [4]. This is a mean of gravitational forces experienced at sea level at a latitude of 45 degrees. This value can be found to agree precisely with the official conversions for pounds mass, pounds force, kilogram, and newton.

The densities of water and mercury, `dh2o` and `dhg`, must be entered in units of kg m^{-3}. The default for water is its density at atmospheric pressure and 4°C. The default for mercury is its density at atmospheric pressure and 0°C.

4.3 Constants

The `setup` function is called when the `units` module is first imported to initialize the PYroMat unit constants listed in Table 4.1 and declare all the unit conversion routines. These constants are used widely throughout the PYroMat system, so altering them is only recommended for advanced users who can anticipate the effects they will have.

Regardless of experience, constants should never be changed manually. They should only ever be changed be re-calling the `setup` function since the unit conversion routines will need to be updated to reflect the new value.

The values in this table are, by the definitions of units, exact (unless noted with ≈). Similarly, the conversions in the following sections are also exact unless they are otherwise noted.

Table 4.1: PYroMat constants and their default values. Values with a * are affected by calls to `setup`.

Sym.	Name	Value		Description
g^*	`const_g`	9.806 65	m s^{-2}	Gravity at 45° lat.
h	`const_h`	6.626 070 15$\times 10^{-34}$	J s	Plank constant
k	`const_k`	1.380 649 $\times 10^{-23}$	J K^{-1}	Boltzmann constant
N_a	`const_Na`	6.022 140 857$\times 10^{23}$	mol^{-1}	Avagadro number
N_c	`const_Nc`	6.241 509 34$\times 10^{18}$	C^{-1}	Electrons per Coulumb
$\overline{\rho}_{std}{}^*$	`const_nstd`	44.615 048 197 83	mol m^{-3}	Standard molar density
$p_{std}{}^*$	`const_pstd`	1.013 25	bar	Standard pressure
q	`const_q`	1.602 176 634 $\times 10^{-19}$	C	Fundamental charge
R_u	`const_Ru`	$\approx$ 8.314 462 618	J mol^{-1} K^{-1}	Universal gas constant
$T_{std}{}^*$	`const_Tstd`	273.15	K	Standard temperature
$\rho_{H_2O}{}^*$	`const_dh2o`	999.972	kg m^{-3}	Std. density of water
$\rho_{Hg}{}^*$	`const_dhg`	1,3595.1	kg m^{-3}	Std. density of mercury

4.4 `Conversion` Class

Unit conversions are performed by a callable `Conversion` class. With the exception of the `matter` and `temperature_scale` functions (described below), each of the conversions is performed by a callable `Conversion` class instance defined by the `setup` function at import.

As a callable, `Conversion` instances mimic a function that accepts up to five arguments,

```
>>> conversion_instance(value=1., from_units=None, \
...     to_units=None, exponent=None, inplace=None)
```

The value may be a scalar or a numpy array of values to be converted. By default, it is 1, so that if no value is specified, the result will automatically be a conversion factor between the given units.

The `from_units` and `to_units` keywords accept strings identifying the units in the conversion. In the event that one of them is omitted, each `Conversion` instance is initialized to check for an entry establishing an default unit in the `pm.config` system. In the sections that follow, each of the unit classes is listed with the name of its instance, the configuration entry that identifies its default unit, and the default that is configured in PYroMat at installation.

The `exponent` keyword allows for powers of units that are not unity. For example, units of velocity are not explicitly included, but it is still easy to convert from meters per second to miles per hour by

```
>>> import pyromat as pm
>>> speed_ms = 10.
>>> temp = pm.units.length(speed_ms, 'm', 'mile')
>>> speed_mph = pm.units.time(temp, 's', 'hr',
   exponent=-1)
>>> print(speed_mph)
22.369362920544024
```

Most users will never need the `inplace` keyword, but when it is set to `True` and the `value` is a numpy array, the original values will be overwritten with the result of the conversion.

`Conversion` instances also support item recall, so that the conversion factor for any unit may be obtained simply

```
>>> print(pm.units.volume['m3'] / pm.units.volume['L'
    ])
1000.
```

This indicates that there are 1000 liters per cubic meter. Most users will not need to use this capability, but for those that do, it is important to always use ratios of units of the same class. Each item recall returns a scalar value indicating the *relative* size of the respective unit, but relative to what? In the case of volume, each value indicates the unit's value in cubic meters, but that is an arbitrary choice.

For a complete list of the supported units in a unit class, use the `get()` method.

```
>>> pm.units.energy.get()
dict_keys(['J', 'kJ', 'cal', 'kcal', 'BTU_ISO', 'eV',
    'BTU'])
```

4.5 Fundamental Units

Fundamental units are those that have precise definitions based on observable phenomena in nature. They are defined by the BIPM.

4.5.1 Time

`Conversion` instance:	`pm.units.time`
`pyromat.config` entry:	`'unit_time'`
Default:	`'s'`

To alter the PYroMat unit for time,

```
>>> import pyormat as pm
>>> pm.config['unit_time'] = 'min'
```

Time is the most fundamental measure defined by the BIPM. "[The second] is defined by taking the fixed numerical value of the caesium frequency $\Delta\nu_{\mathrm{Cs}}$, the unperturbed ground-state hyperfine transition frequency of the caesium-133 atom, to be 9,192,631,770 when expressed in the unit Hz, which is equal to s^{-1}."[5, p.130] Most of the other fundamental units are constructed in terms of it.

Table 4.2: Time units recognized by PYroMat

Setting	Value	Description
`year`	31,536,000 s	year
`day`	86,400 s	day
`hr`	3,600 s	hour
`min`	60 s	minute
`s`	1 s	second
`ms`	0.001 s	millisecond
`us`	10^{-6} s	microsecond
`ns`	10^{-9} s	nanosecond

The other units of time recognized by the PYroMat system are well known, and have simple definitions in terms of the second. It is worth emphasizing that the year and the day recommended by NIST [6] presume precisely 24 hours to a day and 365 days to a year. This ignores the roughly quarter-day annual disagreement between the year and the sidereal year (the time for the Earth to orbit the sun).

Their configuration strings, their values in terms of the second, and their names are listed in Table 4.2.

Time is implemented in the unit conversion system for completeness, but as of PYroMat version 2.1.0, it is not used directly by any of the property methods.

Listing 4.1: Time Conversion Example

```
>>> pm.units.time(30, 's', 'min')
0.5
```

4.5.2 Length

`Conversion` instance:	`pm.units.length`
`pyromat.config` entry:	`unit_length`
Default:	`'m'`

To alter the PYroMat unit for length,

```
>>> import pyormat as pm
>>> pm.config['unit_length'] = 'in'
```

"[The meter] is defined by taking the fixed numerical value of the speed of light in vacuum, c, to be 299,792,458 when expressed in the unit m s^{-1} [...]."[5, p.131] All other metric length units are trivial to derive from the meter.

The international yard (and by extension, the inch, foot, and mile) were defined exactly in terms of the meter before the end of the nineteenth century, but their contemporary values were not adopted until the middle of the twentieth century.[2] Now one yard is defined to be precisely 0.914 4 meters, which results in the better known relationship, one inch is precisely 25.4 millimeters.

The meter was once defined to be a fixed ratio of a great circle around the Earth. This made it of great use for navigation, and for the same reason, the nautical mile is still in common use. The meter was one ten millionth of one quarter of a meridian, making the circumference of the Earth precisely 40,000 km by definition [2]. The nautical mile was was also primarily used for navigation, but it was set to the distance covered by one arc minute of latitude (of which there are 21,600 in a circle). Therefore, the nautical mile was exactly 40,000,000 m / 21,600 or approximately 1,851.851 85 m. In the middle of the twentieth century, the United States Departments of Commerce and Defense jointly declared the nautical mile to be redefined as precisely 1,852m [7].

The length scales supported by PYroMat and their values in terms of the meter are listed in table 4.3.

Example:

```
>>> pm.units.length(12, 'in', 'm')
.127
```

4.5.3 Mass and Weight

`Conversion` instance:	`pm.units.mass`
`pyromat.config` entry:	`unit_mass`
Default:	`'kg'`

To alter the PYroMat unit for mass,

```
>>> import pyormat as pm
```

Table 4.3: Length units recognized by PYroMat

Setting	Value	Description
`km`	1,000 m	kilometer
`m`	1 m	meter
`cm`	0.01 m	centimeter
`mm`	0.001 m	millimeter
`um`	1×10^{-6} m	micrometer
`nm`	1×10^{-9} m	nanometer
`A`	1×10^{-10} m	Angstrom
`in`	0.025 4 m	inch
`ft`	0.304 8 m	foot
`yd`	0.914 4 m	yard
`mile`	1,609.344 m	statute mile
`mi`	Alternate of `mile`	
`nmi`	1,852 m	nautical mile

```
>>> pm.config['unit_mass'] = 'lb'
```

"[The kilogram] is defined by taking the fixed numerical value of the Planck constant h to be $6.62607015 \times 10^{-34}$ when expressed in the unit J s, which is equal to kg m^2 s^{-1}" [5, p.131]. Like many of the remaining fundamental units, the kilogram is defined in terms of length and time, making it dependent on their definitions as well.

The atomic mass unit (u or amu) is defined as precisely 1/12 the mass of a carbon 12 atom at rest. However, in PYroMat the value of one u is calculated in kilograms by enforcing that 1000 moles of a substance with 1u of mass will have 1kg of mass. These two definitions of the atomic mass unit are equivalent to the precision of their definition[5, p.146].

Definitions for the pound and units derived from it are often confused by conflicting definitions of the term "weight." For example, in NIST special publications it is possible to find "[...] the weight of a body in a particular reference frame is defined as the force that gives the body an acceleration equal to the local acceleration of free fall in that reference frame" [6, p.23] and "In general usage, the term 'weight' nearly always means mass, and this is the meaning given the term in

Table 4.4: Mass units recognized by PYroMat

Setting	Value	Description
`kg`	1 kg	kilogram
`g`	0.001 kg	gram
`mg`	1×10^{-6} kg	milligram
`lbm`	0.453 592 37 kg	pound-mass
`lb`	Alternate form of `lbm`	
`oz`	0.028 349 523 125 kg	ounce
`slug`	$\approx$ 14.593 902 9 kg	slug
`u`	$\approx$ 1.660 539 06 $\times 10^{-27}$ kg	atomic mass unit
`amu`	Alternate form of `u`	

U.S. laws and regulations." [8, p.10]

However, at the end of the nineteenth century, the definition of the avoirdupois pound was set to a fixed fraction of the kilogram, formalizing the definition of the pound as a measure of mass and not force. The distinction became important as the precision of measuring instruments exceeded the consistency of the strength of gravity over the Earth's surface. The contemporary value for the avoirdupois pound was established in the middle of the twentieth century, and is precisely 0.453 592 37 kg of mass.[7]

The troy pound and the troy ounce predate the avoirdupois pound as a measure primarily used for quantities of precious metal in coinage [2, p.6], but today they are rarely used outside of these specialized applications. They are omitted from PYroMat to avoid confusion and because they are not often used in scientific or engineering applications.

Table 4.4 shows the mass units recognized by PYroMat and their values in kilograms. Note that only the atomic mass unit and the slug have been rounded. The rest of the relationships are precise by definition.

Example:

```
>>> pm.units.mass(2, 'lb', 'kg')
0.90718474
```

4.5.4 Molar

`Conversion` instance:	`pm.units.molar`
`pyromat.config` entry:	`unit_molar`
Default:	`'kmol'`

To alter the PYroMat unit for molar quantities,

```
>>> import pyormat as pm
>>> pm.config['unit_molar'] = 'lbmol'
```

"One mole contains exactly 6.022 140 76 $\times 10^{23}$ elementary entities. This number is the fixed numerical value of the Avogadro constant, N_a, when expressed in the unit mol^{-1} and is called the Avogadro number."[5][p.134]. Because a mole is a unit of counting, it is equally valid to refer to a mole of planets (about right for our observable universe) as it is to refer to a mole of gas.

The mole is the quantity of a substance with a total mass in grams numerically equal to the mass of the elementary entities expressed in atomic mass units (see Section 4.5.3). This relationship is mirrored by more recently invented alternatives such as the kilogram-mole (kmol) or the pound-mole (lbmol), which enjoy the same relationships with their mass-based namesakes.

If the mass of a single molecule is m_0, a mass, m, of many such molecules must contain

$$N = \frac{m}{m_0} \tag{4.1}$$

molecules.

By definition, when $N = N_a$, the mass of the group expressed in grams will equal the mass of the molecule expressed in atomic mass units. Therefore, the gram-mole is

$$N_{\mathrm{g}} = N_a = \frac{1\mathrm{g}}{1\mathrm{u}}. \tag{4.2}$$

The same process may be applied for any other unit of mass, so the quantity required for m to be expressed in kilograms is

$$N_{\mathrm{kg}} = \frac{1\mathrm{kg}}{1\mathrm{u}}. \tag{4.3}$$

Therefore,

$$\frac{N_{\mathrm{kg}}}{N_{\mathrm{g}}} = \frac{1\mathrm{kg}}{1\mathrm{g}} = 1000. \tag{4.4}$$

The moles are converted by the same conversion factors as their mass equivalents, so a kilogram-mole contains precisely 1000 times as many elementary particles as the gram-mole. To express a quantity, N, in molar units, its quantity only needs to be divided by the quantity of the corresponding mole type. For example, in kilogram moles,

$$\overline{N} = \frac{N}{N_{kg}}. \tag{4.5}$$

For describing quantities of a gas, standard (or normal) volumetric units are used so widely that the problematic aspects of their definitions are worth tolerating. A standard (or normal) volume is the quantity of an ideal gas that would occupy that volume at standard conditions. US Customary and imperial units use the word "standard" and metric units use the word "normal," but the meaning is the same.

Regardless of the mass of the elementary particles, an ideal gas has a consistent number density (concentration) at given conditions,

$$\overline{\rho}_{std} = \frac{p_{std}}{R_u T_{std}}, \tag{4.6}$$

given in mole count per unit volume. Obviously, precise and consistent definitions for standard conditions are essential here. By default PYroMat uses $p_{std} = 1.013\ 25$ bar and $T_{std} = 273.15$ K. When $\overline{\rho}_{std}$ is expressed in gram-moles per cubic meter, it is approximately 44.615 033 4.

To calculate the number of moles in a standard volume, one need only multiply by the volume in question. So, a normal liter contains .044 615 033 4 moles (when standard temperature and pressure are as above).

Table 4.5 shows the molar units and their values expressed in kilogram-moles. The values marked with * are dependent on the standard conditions provided when `setup()` was last called.

Even though the mole is the BIPM standard for quantity of a substance, PYroMat uses the kmol or kilogram-mole as the default molar

Table 4.5: Molar units recognized by PYroMat

Setting	Value	Description
`kmol`	1 kmol	kilogram-mole
`mol`	0.001 kmol	gram-mole
`lbmol`	0.453 592 37 kmol	pound-mole
`n`	$\approx 1.660\ 539\ 06 \times 10^{-27}$ kmol	count
`Nm3`*	$\approx 0.044\ 615\ 033\ 4$ kmol	normal cubic meters
`Ncum`*	Alternate form of `Nm3`	
`NL`*	$\approx 44.615\ 033\ 4 \times 10^{-6}$ kmol	normal liters
`Ncc`*	$\approx 44.615\ 033\ 4 \times 10^{-9}$ kmol	normal cubic centimeters
`scf`*	$\approx 0.001\ 263\ 357\ 06$ kmol	standard cubic feet
`sci`*	$\approx 0.731\ 109\ 408 \times 10^{-6}$ kmol	standard cubic inches

unit for self consistency. For example, the molecular weight property methods return mass per molar units. If the mass units were set to kg and the molar units were set to mol, then the molar mass of diatomic nitrogen would be reported as something near 28,000. Setting the two consistently prevents this unusual result.

Example:

```
>>> pm.units.molar(1.5, 'scf', 'kmol')
0.0018950355849594869
```

4.5.5 Matter

Function:	`pm.units.matter`
`pyromat.config` entry:	`unit_matter`
Default:	`'kg'`

To alter the PYroMat unit for matter quantities,

```
>>> import pyormat as pm
>>> pm.config['unit_matter'] = 'lbm'
```

Molar and mass units provide parallel methods for quantifying amounts of matter. Thermodynamic properties can be expressed in either, so PYroMat uses a third class of units, *matter*, which is an amalgamation of all molar and mass units. A unit matter may be any

of the units listed in Tables 4.4 and 4.5. Since every molar unit can be related to the kilogram-mole and every mass unit can be related to the kilogram, it is only important that we establish how the kilogram is related to the kilogram-mole.

The ratio between kilograms and kilogram moles is a property of the molecule, and is called its molar or molecular weight, W. It is typically expressed in terms of atomic mass units per molecule, but it is the same in kilograms per kilogram moles. The mass, m of a molar quantity, $\overline{N}$, in kilogram moles then, is

$$m = \overline{N}W. \tag{4.7}$$

Unlike any of the other unit conversions, this one depends on the properties of the substance itself. As a result, it is implemented in a custom function that adds molecular weight (in u per molecule), `mw`, as a mandatory argument in addition to the other standard `Conversion` arguments.

```
>>> matter(value, mw, from_units=None, \
... to_units=None, exponent=None, inplace=False)
```

If the to- and from-units are in the same molar or mass class, then the function merely calls the appropriate `Conversion` instance.

These examples consider a molecule with molecular weight exactly 2.0 u (H_2):

```
>>> import pyromat as pm
>>> pm.units.matter(1, 2, 'kg', 'kmol')
0.5
>>> pm.units.matter(1, 2, 'kg, 'lbmol')
1.1023113109243878
>>> pm.units.matter(1,2,'kg','lb')
2.2046226218487757
>>> pm.units.mass(1, 'kg', 'lb')
2.2046226218487757
```

4.5.6 Temperature

`Conversion` instance:	`pm.units.temperature`
Scale function:	`pm.units.temperature_scale`
`pyromat.config` entry:	`unit_temperature`
Default:	`'K'`

To alter the PYroMat unit for temperature quantities,

```
>>> import pyormat as pm
>>> pm.config['unit_temperature'] = 'K'
```

This history of temperature as a measure of hot and cold extends back longer than thermodynamics as a rigorous theory. Without knowing what underlying principles caused substances to be hot or cold, a Celsius scale could still be reliably constructed on a thermometer by marking its readings in ice water and boiling water and by dividing the space between into one hundred equal increments.

With the discovery that temperature indicates the mean translational kinetic energy of the molecules of an ideal gas, the kinetic theory of gasses gives the relationship,

$$\frac{1}{2}m_0 \left\langle u^2 \right\rangle = \frac{3}{2}kT. \tag{4.8}$$

The coefficient, k, which is now known as Boltzmann's constant, establishes the proportionality between kinetic energy and temperature. This also provides the idea of an *absolute* temperature scale; one in which zero temperature corresponds to zero energy rather than an arbitrary choice (like the freezing point of water at standard pressure). In this way, Boltzmann's constant completely determines an absolute temperature scale in terms of the other units.

The BIPM stipulates that "[the kelvin] is defined by taking the fixed numerical value of the Boltzmann constant k to be 1.380649×10^{-23} when expressed in the unit J K^{-1}, which is equal to kg m^2 s^{-2} K^{-1} [...]"[5, p.133] This makes the value for the Boltzmann constant exact by definition.

The realization that temperature is a measure of certain portions of a system's energy is especially useful in plasma physics and related fields. In these studies, it is useful to express temperature directly

as an energy, or electron-volts (eV). The temperature of a substance expressed in electron-volts is the electrical potential that can be built up due to thermal motions of electrons, and is equal to kT/q, when q is the fundamental charge.

The rankine scale is the US customary and imperial equivalent to the kelvin, with one 1.8 R per 1 K. The Celsius scale uses the same increments as the kelvin scale, but its zero is set 273.15 K, the freezing point of water at 1 atmosphere. The Fahrenheit scale uses the same increment as the rankine, but its zero is set so that the freezing point of water at standard pressure occurs at precisely 32°F.

Scales with offsets defy the unit conversion rules implemented by the `Conversion` instance; that all measurements can be converted between the units merely by multiplying by a factor. Obviously, when converting values for a temperature, it is important to handle these offsets correctly, but when considering derivatives involving temperature or other changes in temperature, the offsets must be ignored. As a result, there are two temperature-based unit conversion tools.

The `pm.units.temperature` is a `Conversion` instance should only be used in cases where changes or derivatives in temperature are being considered (e.g. in specific heat or entropy). In these cases, offsets can be ignored, and the conversion factor rules are observed.

When temperatures are being converted between scales so that the offsets between them must be respected, the `pm.units.temperature_scale` function should be used instead. It is a function that mimics the `Conversion` call signature, but that is specially written to handle temperature scales.

```
>>> pm.units.temperature_scale(value, \
...     from_units=None, to_units=None,\
...     inplace=False)
```

Just like the `Conversion` instances, it respects the `unit_temperature` configuration parameter for any unspecified units. However, it makes no sense to refer to a temperature scale with any exponent other than 1, so the exponent parameter is absent.

The temperature scales and their respective differential values in kelvin are listed in Table 4.6. Note that the offsets are not included in this table.

Table 4.6: Temperature units recognized by PYroMat

Setting	Value	Description
`K`	1 K	kelvin
`C`	1 K	degree Celsius
`R`	5/9 K	rankine
`F`	5/9 K	degree Fahrenheit
`eV`	$\approx 86.173\ 332\ 6 \times 10^{-6}$ K	electron-volt

Examples:

```
>>> pm.units.temperature(2, 'C', 'F')
3.6
>>> pm.units.temperature_scale(2, 'C', 'F')
array(35.6)
```

4.6 Derived Units

Derived units are ones that are defined in terms of the fundamental units. They have no fundamental definition in nature, but are instead calculated in terms of the fundamental units.

4.6.1 Force

`Conversion` instance:	`pm.units.force`
`pyromat.config` entry:	`unit_force`
Default:	`'N'`

To alter the PYroMat unit for force quantities,

```
>>> import pyormat as pm
>>> pm.config['unit_force'] = 'kgf'
```

Force is a concept that originates with Newton's laws of motion, so it is fitting that the principle force unit be called the Newton. It is equivalent to one kg m s^{-2} [5, p.137], corresponding to mass times acceleration.

Force units that are derived from measures of mass (like kilogram-force and pound-force) can be calculated in terms of Newtons by cal-

Table 4.7: Force units recognized by PYroMat

Setting	Value	Description
`N`	1 N	newton
`kN`	1,000 N	kilonewton
`lbf`*	≈ 4.448 221 62 N	pound-force
`lb`*	Alternate for `lbf`	
`oz`*	≈ 0.278 013 851 N	ounce

culating their weight (in Newtons) under Earth gravity. These units depend on the value assigned to g in the last call to `setup()` (see Section 4.2 of this chapter).

Table 4.7 lists the supported force units and their values in Newtons.

4.6.2 Energy

`Conversion` instance:	`pm.units.energy`
`pyromat.config` entry:	`unit_energy`
Default:	`'kJ'`

To alter the PYroMat unit for energy quantities,

```
>>> import pyormat as pm
>>> pm.config['unit_energy'] = 'kcal'
```

The Joule is the SI measure of energy, and it is defined as one N m, or one kg m^2 s^{-2} [5, p.137]. The calorie is of historical importance to the scientific community, but its use has fallen out in favor of the Joule. The British thermal unit (BTU) is still broadly used in some industries (especially heating and refrigeration), but few authoritative definitions for unit systems recognize either in their contemporary standards.

The calorie has a number of alternate definitions that has made the unit somewhat problematic. One set of definitions is the energy required to raise a gram of water one degree Celsius at different "standard" conditions. Another is 1/100 of the energy required to raise a gram of water from its ice point to boiling at atmospheric pressure. In 1956, the Fifth International Conference on the Properties of Steam set the "international table calorie" to be precisely 4.1868 J. The calorie in broadest contemporary use is the thermochemical calorie, which

Table 4.8: Energy units recognized by PYroMat

Setting	Value	Description
`J`	1 J	joule
`kJ`	1,000 J	kilojoule
`cal`	4.184 J	calorie
`kcal`	4,184 J	kilocalorie
`BTU`	$\approx$ 1,054.350 26 J	British thermal unit
`eV`	1.602 176 634 $\times 10^{-19}$ J	electron-volt

seems to be universally understood to be precisely 4.184 J [6], though an original source for that definition is difficult to find. Because it is the unit adopted by the International Unions of Pure and Applied Chemistry and Physics (IUPAC and IUPAP) [9] PYroMat also adopts the thermochemical calorie.

Whatever definition is used for the calorie, the BTU may be derived from it by adjusting the quantity of water to be one pound and the temperature rise to be one degree Fahrenheit. The BTU so derived from the thermochemical calorie is approximately 1,054.350 26 J.

It is important to emphasize that it is difficult to find contemporary authorities that certify the BTU or calorie for commercial use. The ISO standard that historically defined the calorie (ISO 31-4) was withdrawn and superseded by ISO 80000-5, which makes no mention of the calorie or the BTU. NIST's special publication number 811 from which the conversion is lifted specifically lists the calorie as an "unacceptable unit."

Finally, the electron-volt is the energy required to move a single electron through a one volt potential. It is expressed in Joules as 1V $\times q$.

4.6.3 Pressure

`Conversion` instance:	`pm.units.pressure`
`pyromat.config` entry:	`unit_pressure`
Default:	`'bar'`

To alter the PYroMat unit for pressure quantities,

```
>>> import pyormat as pm
>>> pm.config['unit_pressure'] = 'Pa'
```

Pressure is a force exerted per unit area. In addition to its typical use describing fluid forces on surfaces, pressure units are also used to describe stresses and surface loads in solids. The SI unit for pressure is the pascal, which is defined as one kg m^{-1} s^{-2} [5, p.137]. The bar is precisely 100,000 Pa, and is in broad use thanks to its proximity to atmospheric pressure.

The "standard atmosphere" was adopted by the (CGPM) in 1954 to be precisely 101,325 newtons per square meter [4]. It is intended to be indicative of a global mean barometric pressure (adjusted to sea level). This definition is in broad use both scientifically [9] and commercially [6] as value of one atm. The standard atmosphere can be adjusted using the `pstd` parameter of `setup` (see Section 4.2 of this chapter).

Liquid column units for pressure are defined as the increase in pressure observed beneath a column of liquid of some height. Like bourdon tube gauges, these pressures are measured relative to the ambient, and are also sometimes called gauge pressure. Provided sufficient measures are taken to avoid the impacts of meniscus, the liquid column pressure, p, may be calculated for a liquid with mass density, ρ, under a uniform gravitational acceleration, g, with a column height, h,

$$p = \rho g h. \tag{4.9}$$

For measures of high precision, it is obviously necessary to define a standard gravity and standard densities for the column fluids.

By default, PYroMat uses the acceleration of free fall in Earth gravity, g, to be precisely 9.8065 m s^{-2}. This value was internationally adopted in 1901 by the third General Conference on Weights and Measures (CGPM)[3] and is still in broad use as a "standard" value [8, p.45] [6, p.5]. However, it should be emphasized that actual pressures produced by liquid columns will vary significantly with altitude, proximity to dense geological formations, and especially latitude.

It is common to calculate the mmH_2O with a convenient and easy-to-remember 1,000 kg m^{-3} water density. The so-called 4°C mmH_2O is based on a water density of 999.972 kg m^{-3}, which is close enough to the convention for most purposes. However, actual water density in the

"ambient" range 20°C to 25°C can be as low as 997 kg m^{-3}, so practical realizations of this unit in a laboratory setting are unlikely to ever be more precise than 0.3%. By default PYroMat uses the 4°C mmH_2O, but users may change this convention by altering the `dh2o` parameter in the `setup` function (see Section 4.2 of this chapter). When in doubt, users should calculate their own water column pressures in Pa or bar based on the known conditions in the lab.

There is also a variety of conventional values established around the mmHg column. The definition most commonly used is the 0°C mmHg, which is adopted by PYroMat by default. A density of 13,595.1 kg $^{-3}$ for mercury with the above value for g results in a conversion factor identical to the value officially adopted by NIST [6, p.52] to the precision given.

Because they are identical to practical precision, the Torr and the mmHg are often treated as identical units, but, there is a difference in definition. The Torr is defined so that 1 atm is precisely 760 Torr. As seen in Table 4.9, they are not quite identical, but they are so close, few practical measurements will ever mandate a distinction. However, the distinction means that the Torr does not depend on the choice for the density of mercury or gravity. Instead, it only depends on the definition of the standard atmosphere.

As mentioned above in the discussion on water column, "gauge" measurements of pressure are often made relative to the ambient conditions. These kinds of pressures should never be used in thermodynamic calculations, so to avoid confusion, PYroMat only works in "absolute" pressure, which is actual force per actual area. To convert back-and-forth between absolute pressure and the gauge measurements that are easier in the laboratory, the `units` module includes the `abs_to_gauge` and `gauge_to_abs` functions. See their in-line documentation for more information.

4.6.4 Volume

`Conversion` instance:	`pm.units.volume`
`pyromat.config` entry:	`unit_volume`
Default:	`'m3'`

Table 4.9: Pressure units recognized by PYroMat

Setting	Value	Description
`Pa`	1 Pa	pascal
`kPa`	1,000 Pa	kilopascal
`MPa`	1,000,000 Pa	megapascal
`bar`	100,000 Pa	bar
`atm`*	101,325 Pa	atmosphere
`Torr`*	≈ 133.322 368 Pa	Torr
`mmHg`*	≈ 133.322 387 Pa	mm mercury column
`inHg`*	≈ 3,386.388 64 Pa	inches mercury column
`mmH2O`*	≈ 9.806 375 41 Pa	mm water column
`inH2O`*	≈ 249.081 936 Pa	inches water column
`psi`	≈ 6,894.757 29 Pa	pounds per square inch
`ksi`	≈ 6,894,757.29 Pa	kips per square inch
`psf`	≈ 47.880 259 0 Pa	pounds per square foot

To alter the PYroMat unit for volumetric quantities,

```
>>> import pyormat as pm
>>> pm.config['unit_volume'] = 'gal'
```

Volume is the measure of the size of a region in space. SI volumetric units are entirely derived from cubes of the linear units, with the liter being equivalent to 0.001 m^3.

Historically the English units for fluid ounce, pint, quart, and gallon were based on the volume occupied by specific weights of water. The English wine gallon was first implemented in 1707 as precisely 231 cubic inches, which was quite close to the gallon based on quantities of water. Though the United Kingdom abandoned it in 1824, it was adopted as the official gallon by the United States Treasury Department in 1832 [2, p.]. Today, the US gallon is still defined as precisely 231 cubic inches.

The Imperial gallon, which the UK and Canada adopted in place of the wine gallon, was the volume occupied by 10 pounds of liquid water at atmospheric conditions. Today, the imperial gallon is precisely 4.546 09 liters, which is roughly consistent with its original definition.

The system of US liquid quart (1/4 gallon) and liquid pint (1/8 gallon) follow from the definition of the gallon, but it should be emphasized

Table 4.10: Volumetric units recognized by PYroMat

Setting	Value	Description
`cum`	1 cum	cubic meter
`m3`	Alternate for `cum`	
`cc`	$\times 10^{-6}$ cum	cubic centimeter
`cm3`	Alternate for `cc`	
`cumm`	1×10^{-9} cum	cubic millimeter
`mm3`	Alternate for `cumm`	
`L`	.001 cum	liter
`mL`	1×10^{-6} cum	milliliter
`uL`	1×10^{-9} cum	microliter
`cuin`	$0.163\ 870\ 64 \times 10^{-5}$ cum	cubic inch
`in3`	Alternate for `cuin`	
`cuft`	0.028 316 846 591 cum	cubic foot
`ft3`	Alternate for `cuft`	
`USgal`	0.003 785 411 783 cum	US gallon
`gal`	Alternate for `USgal`	
`qt`	$0.946\ 352\ 946 \times 10^{-3}$ cum	liquid quart
`pt`	$0.473\ 176\ 473 \times 10^{-3}$ cum	liquid pint
`UKgal`	0.004 546 09 cum	imperial gallon

that there is a vast and nuanced system of specialized volumetric units that are not included in PYroMat (e.g. fluid ounce, dry quart, dry int, bushel, etc.). This decision is primarily to avoid confusion, but also to keep the focus on units of relevance to the engineering and scientific community.

Table 4.10 shows the volumetric units recognized by PYroMat.

Chapter 5

The PYroMat modules

The core of PYroMat's functionality is split across four modules; `reg`, `dat`, `utilty`, and `units`. The `units` module already has its own chapter (Chapter 4), so this chapter is devoted to a description of the backend and how it retrieves the models and their data.

When the PYroMat package is first imported, the load process is completed in three steps: (1) configuration, (2) registry, and (3) data. In the configuration stage, PYroMat loads configuration files, which are described in detail in Chapter 3. In the registry stage, PYroMat searches for Python code that defines the classes that handle the substance models. Finally, in the data stage PYroMat searches for the `*.hpd` files that define the substance data.

5.1 The class registry module, `reg`

The `reg` module only has three members of interest to a user; the `registry` dictionary, the `regload` function, and the `__basedata__` class. As a module, `reg` is responsible for maintaining the `reg.registry` dictionary, which contains all of the classes that provide the substance models. Each entry of this dictionary is a child class of the `reg.__basedata__` class. The key for each member of the dictionary is the same as its name and serves as the string used to identify the class. For example, in the default installation, `reg.registry['ig2']`, recalls the `ig2` class, so `'ig2'` can be used as a value for the `class` entry of a data file.

When PYroMat is first imported, it calls `reg.regload()` with no arguments, which causes it to discover a list of possible registry directories from the `config['reg_dir']` entry. If the `config['reg_recursive']` is set to `True`, the `reg.regload()` function will also descend into subdirectories. Calling `reg.regload()` at any time causes it to repeat this process. See the in-line documentation for how `reg.regload()` arguments can be used to override configuration entries when manually repeating the registry load process from the command line.

Great care should be taken when specifying potential registry directories. The `reg.regload()` function executes all Python codes in the registry directories to check for definition of `reg.__basedata__` classes, so registry directories that define codes globally (for all users) should be protected just like system files, and users should be disallowed from loading each other's registries.

5.2 The data module, `dat`

The `dat` module is responsible for maintaining the `dat.data` dictionary of all available substance models. It includes tools to discover, load, and manipulate the model data. The data dictionary is where `get()` finds the substance instances requested by the user.

5.2.1 The `load()` function

The `dat.load()` function is the most important of `dat`'s functions. It is responsible for loading all of the substance data and creating the class instances with which users will interact. When PYroMat is first imported, `dat.load()` is called with no arguments, which prompts it to discover a list of possible data file directories from `config['dat_dir']`. Every file with a `.hpd` file extension is loaded and used to initialize an appropriate class instance. Each instance is added to the `dat.data` dictionary with its "id" as the keyword string. See section 5.2.2 for more information.

When it is called with a directory or a path to a specific file, `dat.load()` will only open the contents of that directory or that file. This is useful for manually adding or re-loading files under development

that are not yet in the data directories listed in `config`.

The `dat.load()` function also has an optional keyword, `check`, that prompts a data file integrity check when set to `True`. When run in check mode, `dat.load()` returns a dictionary with six entries describing the results of the load process. If a user has changed the data contained in the `dat.data` dictionary, added a new element, or deleted an existing element, it will be discovered by comparing the current data dictionary against a repeated load process.

```
>>> import pyromat as pm
>>> # This does NOT affect pm.dat.data
>>> result = pm.load(check=True)
>>> result['changed']    # modified substances
>>> result['added']      # new substances
>>> result['removed']    # substances removed
>>> result['redundant']  # redundant files
>>> result['suppressed'] # excluded files
>>> result['data']       # new data dict
```

The `changed`, `added`, and `removed` elements of the `result` dict are lists of id strings for the species that are affected. This lets developers carefully inspect the impermanent changes they have made to the data during a command line session.

The `redundant` dictionary member is a dictionary of substance ids with more than one definition found in the bank of `*.hpd` files. This happens often when multiple data sources have their own contradictory models for the same substances. The keys to the `redundant` dictionary are the substance ids, and the values are lists of paths to the multiple files that specified the same substance id.

The `suppressed` dictionary member is a list of paths to files in the search path with a `*.hpd` file extension. These files that have been removed from the load process (probably to resolve a redundancy), but that the user may want to be able to locate to re-activate them.

See section 5.2.3 for advanced tools for working with this information.

Table 5.1: Required data keywords in all PYroMat files

Keyword	Type	Description
'id'	str	The string used by the get function to recognize the species and its collection. It should be in the format <collection>.<formula>
'class'	str	The string name of the class from the reg module that will be used to interpret the data.
'doc'	str	A long string that is used to describe the data and cite its original source.
'atom'	dict	A dictionary with element symbols for keywords, and the numerical quantity per molecule of each for values. This is used by info to search for species by contents. It can also be useful for calculating the molecular weights of isotopes.

5.2.2 Data files

Files are in the JavaScript Object Notation (JSON) data format, which only requires an ASCII character set, so any UTF-N extension will do. The file should define a dictionary with keyword names the describe the essential data elements of the substance. The sections in Chapters 6 and 7 describe the various substance classes and their required data elements, but there are also certain basic keywords used by PYroMat itself.

5.2.3 Tools for working with data files

For users who want to develop their own models or change existing models, the dat module includes tools to make that easier.

The dat.clear() function empties the current dat.data dictionary. Since successive calls to dat.load() merely overwrite or add to the dictionary, if users really want to start from scratch, they should

call `dat.clear()` before running `dat.load()`.

The `dat.new()` function creates a new entry in the `dat.data` dictionary from a dictionary like one that might be loaded directly from a `*.hpd` file. The intent is to allow users to write scripts that build their own data dictionaries from scratch, test them in PYroMat and only save them permanently when they have been tested.

The `dat.updatefiles()` function runs `dat.load()` in check mode and allows the user to resolve differences between the current members of the `dat.data` dictionary and the currently available data files. When run verbosely, the user will be prompted with a choice of actions for each file. When run with the `verbose=False`, it will automatically operate on all findings. See the inline documentation for more information.

5.3 The utility module, `utility`

The `utility` module is a container for a number of back-end code to which users should not need direct access. For example, it is where all of the PYroMat error types are defined, and there are a number of back-end helper functions.

5.3.1 PYroMat error types

There are special error types defined to be unique to PYroMat. These are intended to help users write their own scripts with meaningful exceptions when things go wrong. Table 5.2 shows all of the error types and describes their use.

5.3.2 Redundancy tools

The `utility.revive_file()` and `utility.suppress_file()` automatically add or remove a ‘ ’ at the end of a file extension to add it to or remove it from the load process. The `utility.red_repair()` is an automatic interactive file redundancy repair function that calls these if necessary. Interested users can read their inline documentation for more information.

Table 5.2: PYroMat error types

Error Type	Description
`PMAnalysisError`	A numerical routine has failed. This probably means that an iteration has not converged or a calculation gave an unexpected illegal result. These are very unusual.
`PMDataError`	A substance data set is corrupt. This usually appears when a required data element is not defined or the data are incorrect data types. Not all of these errors are handled gracefully, so the property methods may simply crash with a basic Python exception.
`PMFileError`	There was an error working with a file. This is probably because the user does not have permission to work one of the `*.hpd` files.
`PMParamError`	There was a problem with an argument passed to a method or function. This is common when the user specifies a state out of the model's range or some otherwise invalid combination of properties.

5.3.3 Other tools

The `utility` module is also where messaging and error handling helper functions, `print_lines()`, `print_warning()`, and `print_error()` reside. These automatically add line breaks and headers in appropriate locations so all terminal communication will be in a standard format.

Finally, `load_file()` is a helper function that does back-end work for the `dat.load()` function. It is a wrapper function that loads and checks the `json` code from data files, and it returns the dictionary they create. The `dat.load()` is responsible for passing them to the appropriate class initializer and adding them to the data dictionary.

Chapter 6

Ideal Gases

The ideal gas is one in which bombarding molecules do not exert significant forces on each other except in collisions. Under these conditions, the distance between molecules (the gas's density) is unimportant for determining thermodynamic properties. Only the gas's temperature (the speed of the molecules) is important. It is worth emphasizing that transport properties (like conductivity and diffusivity) are still impacted by density.

There are two classes in PYroMat that implement the two most widely used models: the `ig` class manages the Shomate equation of state, and `ig2` manages the so-called NASA polynomials equation of state. In either case, constant-pressure specific heat, c_p, is constructed purely as a function of temperature. Here, we re-develop the thermodynamic properties from first principles to demonstrate how c_p is sufficient to calculate them.

6.1 Properties of ideal gases

6.1.1 Ideal gas law

When they are spread so sparsely that forces between molecules are small, ideal gases are well described by the relations

$$p = nkT \tag{6.1a}$$
$$p = \rho RT \tag{6.1b}$$
$$p = \overline{\rho} R_u T. \tag{6.1c}$$

Here, p and T are the pressure and temperature of the gas. The densities are expressed in number density, n, mass density, ρ, and molar density, $\overline{\rho}$. It is clear, then, that the Boltzmann constant, k, and the ideal gas constants, R and R_u, are related,

$$kN_a = RW = R_u. \tag{6.2}$$

The Boltzmann constant is exact by definition thanks to the contemporary definition of temperature (see Section 4.5.6), so this relationship provides the authoritative definitions for the gas constants, R, and R_u.

The ideal gas law is empirical in its origins. It was originally formulated as an amalgamation of the independent laws of Gay-Lussac, Charles, and Boyle. The kinetic theory of gases eventually provided an independent formulation based entirely in Newton's laws. Given its importance to the study of matter and its deeply intuitive nature, it is surprising that virtually no introductory text on thermodynamics gives the subject any treatment. It is worth a brief summary here.

First, let us take that a gas is comprised of molecules that translate freely in space. They may be imagined to follow straight paths of constant velocity unless they collide with a containing wall or another molecule.

Pressure, then, is due to a series of impacts on a surface so rapid that they appear to be continuous. Pressure only has meaning as a property when the density of molecules is sufficiently high that their individual impacts are imperceptible. That means that pressure is a kind of average force determined by a series of may individual random impacts. It is possible to quantify its magnitude by describing the

individual impacts through Newtonian mechanics. If the forces due to a series of impacts in time is $F(t)$, a surface with area, A, will experience a pressure, p, based on the rate of increase of total impulse,

$$p \equiv \frac{1}{At} \int_0^t F(\tau) \mathrm{d}\tau. \tag{6.3}$$

This formulation should have a well defined limit when t is larger than period between individual impacts and shorter than the period over which the local properties change.

In a Cartesian coordinate system with z normal to a surface and positive *into* the surface, the velocity of any single molecule will have three components, $\vec{u}_1 = u_x\hat{i} + u_y\hat{j} + u_z\hat{k}$. After an elastic collision with the surface, the velocity will be $\vec{u}_2 = u_x\hat{i} + u_y\hat{j} - u_z\hat{k}$.

The impulse imparted to the surface by one such collision will be

$$\int_0^t F \mathrm{d}\tau = 2mu_z, \tag{6.4}$$

when m is the mass of the molecule. When collisions occur due to many identical molecules, their impulses accumulate

$$\begin{aligned} \int_0^t F \mathrm{d}\tau &= 2m \left(u_{z,1} + u_{z,2} + u_{z,3} + \ldots\right) \\ &= 2m \sum_i u_{z,i} \end{aligned} \tag{6.5}$$

Note that for a molecule to collide with the surface, its z-component velocity before the collision, u_z, must be positive. This results in a purely positive force (into the surface). Ideal gas molecules experiencing elastic collisions have no mechanism to pull on the surface.

The next step to predict the force on the surface requires a prediction for the rate at which collisions occur. Because faster moving molecules will travel more distance in the time interval, their collisions will be more numerous. Let the number density of molecules with z-component velocity between u_z and $u_z + \mathrm{d}u_z$ be $n'(u_z)\mathrm{d}u_z$. Here, n' is a population density function for a population of molecules based on one component of their velocity. So, n' has units number per volume

per velocity, and its integral over all velocities is precisely n, the total number density of molecules.

The number of molecules with a certain velocity that will strike an area, A, in time t will be determined by the size of the volume that can be traversed by molecules traveling at that velocity. Molecules traveling with z-component velocity u_z will traverse a length $u_z t$ in the time interval. So, the total volume occupied by molecules with that velocity that will strike the surface is Atu_z. The total number of collisions is $Atu_z n'(u_z)\mathrm{d}u_z$. So, the impulse becomes

$$\begin{aligned}\int_0^t F\mathrm{d}\tau &= 2m\int_0^\infty Atu_z{}^2 n'(u_z)\mathrm{d}u_z \\ &= mAt\int_{-\infty}^\infty u_z{}^2 n'(u_z)\mathrm{d}u_z \end{aligned} \tag{6.6}$$

Note that only half of the gas's population will have a velocity component in the positive direction, and the other half will not cause pressure. Therefore, the 2 coefficient is canceled when the bounds of the integral are extended to include all molecules.

The integral is merely a calculation of the average value of $u_z{}^2$ over the population of molecules, and may be simplified to $n\langle u_z{}^2\rangle$ when n is simply the total number density of the molecules and $\langle u_z{}^2\rangle$ is the mean square of z-component velocity. Finally, we have obtained an expression for pressure in terms of the outward velocity component,

$$p = mn\langle u_z{}^2\rangle. \tag{6.7}$$

The last simplification to this relationship comes when we assert that gas velocity statistics are isotropic; velocity statistics are the same in all directions and do not depend on the coordinate system. That implies that $\langle u_x{}^2\rangle = \langle u_y{}^2\rangle = \langle u_z{}^2\rangle$, so

$$\langle \vec{u}^2\rangle = \langle u_x{}^2 + u_y{}^2 + u_z{}^2\rangle = 3\langle u_z{}^2\rangle. \tag{6.8}$$

Therefore, pressure is proportional to the mean square of molecular translational velocity,

$$p = \frac{1}{3}mn\langle u^2\rangle. \tag{6.9}$$

When this relationship is substituted into the ideal gas law above, it provides a purely mechanical interpretation for temperature as well,

$$\langle \frac{1}{2} m u^2 \rangle = \frac{3}{2} kT. \tag{6.10}$$

Temperature is a measure of the average *thermal* energy of the gas. Higher temperature means higher kinetic energy, and the Boltzmann constant relates the two.

This development was quick and it neglects to address mixtures of molecules of different masses. However, the same approach may be used quite intuitively to show that Dalton's law for partial pressures also follows from these basic assumptions.

6.1.2 Internal energy

What happens to heat and work as they are added to an ideal gas depends on the structure of the gas molecule. If there are chemical, atomic, or phase changes, the species can be modeled as vanishing and being replaced by a new substance with its own properties. The ideal property models are, therefore, concerned with how energy is stored in a molecule that is neither changing its fundamental structure nor changing phase. This remaining energy will be exhibited entirely as the thermal energy in (6.10) and internal vibration, rotation, or electrical motions inside the molecule, for which we have not yet accounted.

Perfect gases are ideal gases, but not all ideal gases are perfect. The molecules of an ideal gas collide elastically and have no further interactions with their surroundings. However, the molecules of a perfect gas are further assumed to neither spin nor vibrate. As a result, any molecule more complicated than a single atom does not form a perfect gas.

When a gas composed of a monoatomic molecule like argon is heated (without atomic, chemical, or phase changes) the energy can only be stored in the thermal translational energy described in (6.10). A sample of N molecules of such a gas would have thermal energy $N\frac{3}{2}kT$, so the

energy of one kmol or one kg is

$$\overline{e} - \overline{e}_0 = N_{kg}\frac{3}{2}kT = \frac{3}{2}R_u T \tag{6.11}$$

$$e - e_0 = \frac{N_{kg}}{W}kT = \frac{3}{2}RT. \tag{6.12}$$

See section 4.5.4 for definitions of the molar mass, W, and kilogram-mole count, N_{kg}. Here, e_0 is the internal energy due to the atomic, chemical or phase changes we have not yet considered. Note that the gas constants, R_u and R, appear naturally in terms of the Boltzmann constant, k.

Non-perfect ideal gases are made of more complicated molecules, and they can store energy in more complicated ways; they vibrate, they rotate (spin), and they have means of storing energy electrically. They do this because the can; they have more degrees of freedom.

The 3 in (6.12) first appeared in (6.8) because all ideal gases are free to translate in three directions. These three thermal degrees of freedom are the only ones that contribute to measurements of temperature, but complex molecules have more degrees of freedom that do not involve translation through space of the molecule's bulk. If these are included in the internal energy as well, a molecule that is free to vibrate, rotate, and translate in f degrees of freedom will have an internal energy

$$e - e_0 = \frac{N_{kg}}{W}\frac{f}{2}kT = \frac{f}{2}RT. \tag{6.13}$$

The intuitive assumption might be that f is constant and a property of each molecule. This assumption implicitly asserts that all modes of vibration, rotation, and translation have an equal fraction of the total energy so it is called the *equipartition* assumption. It results in an elegant formulation, but it completely fails to predict the actual behavior of molecules in gases.

Instead, it is necessary to stipulate that f is a statistical quantity that can change with the state. It can be thought of as the number of "active" degrees of freedom of each molecule. Its minimum is 3, but it can increase to a maximum that will depend on the complexity of the molecule and the energy of the collisions it endures. For example, at very low temperatures, molecules may not vibrate much; instead

they may just bang around and spin like rigid bodies (see Figure 6.1 below). However, at higher temperatures, there may be enough energy to excite vibrations inside the molecules.

Fortunately, since all ideal gases (perfect or not) are presumed to collide elastically, the equilibrium distribution of energy should depend on neither the rate of collisions nor the distance separating the molecules. Obviously, this assumption will break when molecules are packed in closely with one another. However, so long is that is true, the active degrees of freedom and the internal energy will both only be a function of temperature (the average kinetic energy).

$$e(T) - e_0 = \frac{f(T)}{2} RT \tag{6.14}$$

Better insights into the behavior of f will be seen when we examine specific heats below.

6.1.3 Enthalpy

As established above, we will construct all of the properties of the ideal gas in terms of its specific heat, but in order to derive reliable relations for the specific heats, it is important to first re-examine enthalpy.

Recall from Section 1.1.8 that the definition for enthalpy is $e + pv$. For an ideal gas, pv may be substituted with RT, so an ideal gas will have enthalpy

$$h(T) = e(T) + RT. \tag{6.15}$$

For all ideal gases, enthalpy and internal energy are only functions of temperature, and they are related by RT.

6.1.4 Specific heats

Recall from Section 1.1.10 that the constant-volume and constant-pressure specific heats are

$$c_v = \left(\frac{\partial e}{\partial T} \right)_v$$
$$c_p = \left(\frac{\partial h}{\partial T} \right)_p$$

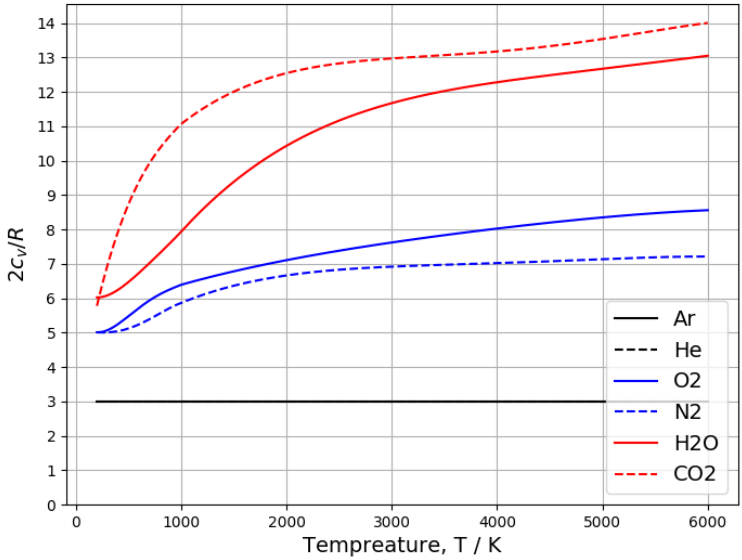

Figure 6.1: Approximate mean active degrees of freedom in selected ideal gas models

Because both internal energy and enthalpy are only functions of temperature, this relationship is relatively simple,

$$c_v = \frac{f(T)}{2} R \left(1 + T \frac{f'(T)}{f(T)}\right) \approx \frac{f(T)}{2} R \tag{6.16}$$

$$c_p = c_v + R \tag{6.17}$$

The approximation in (6.16) only holds when $f(T)$ is very nearly constant.

Figure 6.1 shows the quantities $2c_v/R$ for six gases with increasing molecular complexities. The perfect gases, He and Ar, behave precisely as predicted with exactly three degrees of freedom. All four polyatomic molecules start with quasi-rigid motion at low temperatures before adopting higher degrees of freedom at high temperatures.

For H_2O, rigid motion means six degrees of freedom: three coordinates of translation and three axes of rotation. For diatomic molecules like O_2 and N_2, however, their symmetry robs them of one of their axes of rotation, so they only exhibit five degrees of freedom at low temperatures. Since CO_2 is also an axisymmetric molecule, it can be seen asymptotically approaching 5 at low temperatures as well.

6.1.5 Entropy and enthalpy revisited

Recall from Section 1.1.9 that the entropy of any substance changes like

$$T\mathrm{d}s = \mathrm{d}h - v\mathrm{d}p.$$

For the ideal gas, this can be simplified by substituting RT/p for v and integrating to obtain

$$s - s_0 = \int c_p(T)\frac{\mathrm{d}T}{T} - R\ln\left(\frac{p}{p^\circ}\right). \tag{6.18}$$

when p° is a reference pressure. For a perfect gas,

$$s - s_0 = c_p\ln\left(\frac{T}{T_0}\right) - R\ln\left(\frac{p}{p^\circ}\right). \tag{6.19}$$

when T_0 is a reference temperature.

Evaluating the pressure portion of (6.19) is straightforward, but it will be seen below that the temperature dependence is more nuanced. For that reason, it is often treated alone,

$$s^\circ(T) = s_0 + \int c_p(T)\frac{\mathrm{d}T}{T}, \tag{6.20}$$

where s° is the entropy at the reference pressure, p°.

An identical approach can be taken for enthalpy, but with an even simpler result.

$$h - h_0 = \int c_p(T)\mathrm{d}T \tag{6.21}$$

For a perfect gas,

$$h - h_0 = c_p \left(T - T_0 \right). \tag{6.22}$$

In the NIST-JANAF tables, the specific heat is calculated theoretically from the molecular and atomic structure of each atom, but specific heat is also readily validated by calorimetry. Entropy's integration constant, s_0, is chosen so the values agree with absolute calculations for the species entropy from Boltzmann's statistical model for entropy. Since no such model exists for enthalpy, h_0 is determined by a separate approach described in Section 6.1.7.

The models in the `ig` and `ig2` classes use systems of coefficients to form piece-wise polynomials for $c_p(T)$ instead of using the detailed models recorded in the NIST-JANAF tables. These are ideal for computational codes because they give good numerical performance, but they are not valid down to the low temperatures included by the original tables.

6.1.6 Other properties

Once $c_p(T)$ is well defined, it is also possible to evaluate internal energy,

$$e(T) = h(T) - RT, \tag{6.23}$$

constant-volume specific heat,

$$c_v(T) = c_p(T) - R, \tag{6.24}$$

specific heat ratio,

$$\gamma(T) = \frac{c_p(T)}{c_p(T) - R}, \tag{6.25}$$

speed of sound

$$a(T) = \sqrt{\gamma(T) R T}, \tag{6.26}$$

and others.

6.1.7 Enthalpy of formation

When a substance is formed either by nuclear, chemical or phase change, energy is nearly always released or consumed. This energy is accounted for by a property called the *enthalpy of formation*. It is the energy required to form a substance, so exothermic reactions have negative enthalpies of formation. The ideal gas data on which PYroMat's ideal gas classes do not consider nuclear reactions; only chemical reactions and phase changes.

One might imagine a reactor with a mixture of reactants flowing in and a mixture of products flowing out. In the simplest case, we should imagine the products to be made entirely of the substance we wish to study, so the reactants will be only those that are absolutely necessary for forming it and in the correct proportions.

$$a\mathrm{A} + b\mathrm{B} + \ldots \rightarrow \mathrm{A}_a\mathrm{B}_b\ldots \tag{6.27}$$

Many such chemical reactions release or consume vast amounts of energy. Usually, this results in cooling or heating of the substance as it changes. When heat and work are neither added nor removed from the system, an energy balance mandates an isenthalic process,

$$\begin{aligned} h_{\text{reactants}} &= h_{\text{products}} \\ a\overline{h}_{\mathrm{A}} + b\overline{h}_{\mathrm{B}} + \ldots &= \overline{h}_{\mathrm{A}_a\mathrm{B}_b\ldots} \end{aligned} \tag{6.28}$$

when $\overline{h}$ is enthalpy in molar units.

This result is often counter intuitive. Since ideal gas enthalpy is only a function of temperature, it seems like an isenthalpic process should also be isothermal. However, when chemical or phase changes take place, the enthalpy curves of the products and reactants can be dramatically different.

Fig. 6.2 shows enthalpy curves for a hypothetical set of reactants and products. Not only may the two have dissimilar slopes, but the curves may have large offsets separating them. When the process is isenthalpic, these offsets cause temperature changes shown in red. In the case shown, the reactants have higher enthalpy than the products, so the additional energy is absorbed thermally by the products, causing an increase in temperature. On a molecular level, the effect is like

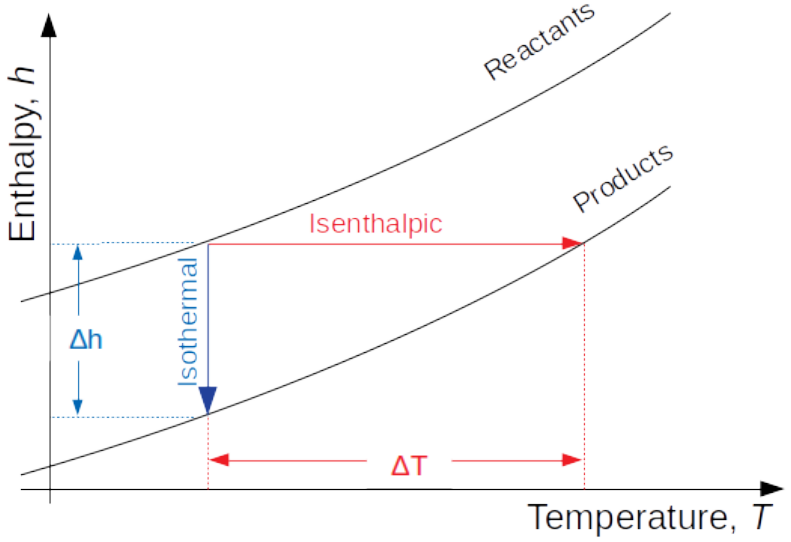

Figure 6.2: Isenthalpic and isothermal reactions on an h-T diagram.

allowing two magnetic marbles to roll near one another on a flat table. Even if neither has much velocity to begin with, after they collide, they will be sent off quickly spinning and rolling. The same happens in an exothermic reaction.

If the hypothetical reactor were modified to add or extract heat so that the temperature of the reaction were constant, the process would adopt the vertical blue line instead. In an isothermal reaction, the amount of thermal energy exhibited by the substance is constant, so any release or consumption of heat will have come from a chemical reaction (including a change in the substance's active degrees of freedom, f). Therefore, the change in enthalpy between the two curves is the enthalpy consumed by the chemical reaction reaction.

In the diagram, the enthalpy is seen decreasing in order to maintain the temperature, so heat was removed, and the reaction is called exothermic. For such a reaction, the energy balance would be

$$a\overline{h}_{\mathrm{A}} + b\overline{h}_{\mathrm{B}} + \ldots + \Delta h = \overline{h}_{\mathrm{A}_a\mathrm{B}_b\ldots} \tag{6.29}$$

Here, the change in enthalpy, Δh, is positive when energy is added to maintain a constant temperature throughout the reaction, so in the exothermic reaction depicted in fig. 6.2, Δh would be negative. In general Δh is also a function of temperature because the properties of the two gas mixtures are not parallel.

The value of $\Delta h(T)$ also depends on which reactants are selected to form the product. For example, one might select atomic oxygen and atomic hydrogen to form water, but it would be just as valid to chose the more common diatomic hydrogen and oxygen. These two reactions would exhibit different Δh values because the energies of the starting substances are different. If one wishes to establish a value for Δh that is clearly defined and indicates the energy required to form the substance, it is necessary to specify a standard set of "reference" substances, relative to which all other substances are derived.

Every element is given exactly one **reference substance**, which may include phase changes (like in the case of the metals). The enthalpy of formation for the reference substance is arbitrarily declared to be zero at all temperatures, since it is imagined to be a fundamental building block from which compounds are built.

The *enthalpy of formation*, $\Delta_f h$, is the energy consumed by the reaction when all of the reactants are reference substances in the proper proportions to produce the product.

Many of the reference substances are atomic gases or pure liquids and crystals (like argon and aluminum). Others are selected to be diatomic gases because of their abundance (like hydrogen, oxygen, and nitrogen).

6.1.8 Properties of mixtures

The discussion so far has applied only to properties of pure ideal gases. The composition of a gas mixture is conventionally defined in either mass or molar quantities of the constituent pure gases. In this section, we establish methods by which properties can be calculated from the properties of the components.

In extensive units, a volume, V, might contain many individual gas species, each with total mass, m_i, or total mole count, N_i. They are

related by the pure gas's molecular weight,

$$m_i = W_i N_i. \tag{6.30}$$

Mass and mole fractions are the fractions of a gas composed of a single pure substance. The total mass and count of gas in the volume is merely the sum of all the constituents, $m = \sum m_i$, and $N = \sum N_i$. These let us more conveniently express the composition in mass and mole fractions, respectively,

$$y_i \equiv \frac{m_i}{\sum m_k} \tag{6.31a}$$

$$\chi_i = \equiv \frac{N_i}{\sum N_k}. \tag{6.31b}$$

Note that the fractional quantities, y_i and χ_i, are not dependent on the choice of units for mass or mole count. Observe that, by definition, $\sum y_i = \sum \chi_i = 1$.

Mixture density is the total mass or mole count of all species divided by the volume they occupy.

$$\rho \equiv \frac{\sum m_i}{V} \tag{6.32a}$$

$$\overline{\rho} \equiv \frac{\sum N_i}{V} \tag{6.32b}$$

If ρ_i and $\overline{\rho}_i$ were the densities of constituent i at the same temperature and pressure as the total mixture, then it is important to emphasize that m_i/V and N_i/V are *not* ρ_i and $\overline{\rho}_i$. That question is addressed below.

Molecular weight of a gas mixture has the same definition as the molecular weight of a pure gas; it is the mass per mole count of total gas. It can be calculated from the constituent molecular weights using

either mole fractions or mass fractions,

$$W \equiv \frac{\sum m_i}{\sum N_j} \tag{6.33a}$$

$$= \frac{\sum N_i W_i}{\sum N_j}$$

$$= \sum \chi_i W_i \tag{6.33b}$$

$$= \frac{\sum m_i}{\sum m_j / W_j}$$

$$= \left(\sum \frac{y_j}{W_j} \right)^{-1}. \tag{6.33c}$$

Partial pressure, p_i, is the pressure force exerted on a surface due only to collisions of a single constituent, i. Section 6.1.1 will help understand what is meant by this idea. It is the pressure force that would be measured if all other constituent gases were removed.

$$p_i \equiv \frac{N_i}{V} R_u T$$

$$= \chi_i \frac{N}{V} R_u T$$

$$= \chi_i \overline{\rho} R_u T \tag{6.34}$$

The same pressure may be calculated from mass fraction,

$$p_i = \frac{N_i}{V} R_u T$$

$$= \frac{m_i}{W_i V} \rho R_u T$$

$$= y_i \rho R_i T \tag{6.35}$$

The ideal gas constant, R_i, is the mass-based gas constant for only that constituent, $R_i = R_u / W_i$.

Total pressure, p, is the pressure force actually experienced by a surface due to all of the constituent gases. By definition,

$$p = \sum p_i$$

$$= \overline{\rho} R_u T \tag{6.36}$$

Observe, also, that

$$\chi_i = \frac{p_i}{p}. \tag{6.37}$$

The total pressure can also be calculated in terms of mass density. Using (6.33c),

$$\begin{aligned} p &= \sum p_i \\ &= \sum y_i \rho R_i T \\ &= \sum \frac{y_i}{W_i} \rho R_u T \\ &= \rho R T. \end{aligned} \tag{6.38}$$

The total mixture gas constant appears naturally.

The mixture gas constant, R, appears naturally when calculating total pressure from total mass density, and it is calculated in precisely the same way as a pure gas constant, but the mixture molecular weight is used instead,

$$R \equiv \frac{R_u}{W} \tag{6.39a}$$

$$= \frac{R_u}{\sum \chi_i W_i}$$

$$= \left(\frac{\chi_i}{R_i} \right)^{-1} \tag{6.39b}$$

$$= R_u \sum \frac{y_i}{W_i}$$

$$= \sum y_i R_i \tag{6.39c}$$

Densities can be calculated from the same properties of the com-

ponent species using the ideal gas law. For mass density,

$$\begin{aligned}
\rho &= \frac{p}{RT} \\
&= \frac{p}{\sum y_i R_i T} \\
&= \left(\sum \frac{y_i}{\rho_i(T,p)} \right)^{-1} && \text{(6.40a)} \\
&= \frac{p}{\left(\sum \chi_i / R_i\right)^{-1} T} \\
&= \sum \chi_i \rho_i(T,p) && \text{(6.40b)}
\end{aligned}$$

For molar density, no such complexity is needed, since the universal gas constant is the same for all gases,

$$\overline{\rho} = \frac{p}{R_u T}$$

Internal energy, enthalpy, entropy, and other bulk properties are merely the sum of all the values contributed by each of the constituent gases. By definition in an ideal gas, the presence of other gases do not affect the behavior of each of the pure constituents. Therefore, a bulk property may be calculated from the properties of the pure substances under equivalent temperature and pressure,

$$\begin{aligned}
\left(\sum m_j \right) \phi(T,p) &= \sum m_i \phi_i(T,p) \\
\left(\sum N_j \right) \overline{\phi}(T,p) &= \sum N_i \overline{\phi}_i(T,p)
\end{aligned}$$

Therefore,

$$\begin{aligned}
\phi(T,p) &= \sum y_i \phi_i(T,p) \\
\overline{\phi}(T,p) &= \sum \chi_i(T,p) \overline{\phi}_i
\end{aligned}$$

This is hardly a proof, but it is sufficient for the scope of this document to assert that it is true for bulk properties.

This identity may be applied to internal energy, enthalpy, entropy, and all the properties derived from them, so

$$e(T) = \sum y_i e_i(T) \tag{6.41a}$$

$$h(T) = \sum y_i h_i(T) \tag{6.41b}$$

$$s(T,p) = \sum y_i s_i(T,p) \tag{6.41c}$$

and

$$\overline{e}(T) = \sum \chi_i \overline{e}_i(T) \tag{6.42a}$$

$$\overline{h}(T) = \sum \chi_i \overline{h}_i(T) \tag{6.42b}$$

$$\overline{s}(T,p) = \sum \chi_i \overline{s}_i(T,p). \tag{6.42c}$$

6.2 The ideal gas collection

There are several classes that implement various data models for ideal gas properties. Their interfaces are all standardized so that very little difference should be apparent to the user, except that the ideal gas mixture class has some extra methods associated with its composition.

All property methods accept any two of temperature, density, or pressure to specify the state.

6.2.1 The Shomate equation: `ig`

PYroMat's `ig` class is built on the Shomate equation for constant-pressure specific heat c_p. This is the formulation used by the NIST webbook [10]. Despite the wise range of substances represented, it has the advantage of using a simple standard piece-wise formulation for specific heat. However, it does suffer from certain limitations.

The Shomate equation takes the form

$$\theta = \frac{T}{T_s} \tag{6.43}$$

$$c_p(t) = c_0 + c_1\theta + c_2\theta^2 + c_3\theta^3 + \frac{c_4}{\theta^2}, \tag{6.44}$$

where the scaling temperature, T_s is 1000K for all species. The decision to scale the temperature by a large value has the effect of scaling τ so that it will not be much larger than 5 or 6. That helps reduce numerical errors in high-order polynomials.

Because of its simplicity, the Shomate equations lack the degrees of freedom to express specific heat over wide ranges, so data are usually given in piece-wise formulations. For example, tungsten dioxide (WO_2), has a set of coefficients for 298K $\leq T <$ 1100K and 1100K $\leq T \leq$ 6000K.

The enthalpy can be explicitly calculated from (6.21),

$$\begin{aligned} h(T) &= h_0 + \int c_p(T)\mathrm{d}T \\ &= h_0 + T_s \int c_p(\theta)\mathrm{d}\theta \\ &= T_s \left(c_0\theta + \frac{c_1}{2}\theta^2 + \frac{c_2}{3}\theta^3 + \frac{c_3}{4}\theta^4 - \frac{c_4}{\theta} + c_5 \right). \end{aligned} \tag{6.45}$$

It is important to emphasize that h_0 is not the same as the enthalpy of formation, Δh_f°. Instead, it is merely an integration constant, which can be alternately expressed as a new coefficient, c_5.

Because of the temperature term in the denominator, no multiple of T_s appears in entropy when the integration is changed to θ,

$$\begin{aligned} s^\circ(T) &= s_0 + \int \frac{c_p(T)}{T}\mathrm{d}T \\ &= s_0 + \int \frac{c_p(\theta)}{\theta}\mathrm{d}\tau \\ &= c_0 \ln\theta + c_1\theta + \frac{c_2}{2}\theta^2 + \frac{c_3}{3}\theta^3 - \frac{c_4}{2\theta^2} + c_6 \\ s(T,p) &= s^\circ(T) - R\ln\left(\frac{p}{p^\circ}\right) \end{aligned} \tag{6.46}$$

Just like in the enthalpy integral, a new coefficient, c_6, has been introduced to represent the integration constant.

Internal energy is readily calculated from the definition of enthalpy in (6.21),

$$e(T) = h(T) - RT \tag{6.47}$$

There is a similarly simple relationship to determine constant-volume specific heat and specific heat ratio,

$$c_v(T) = c_p(T) - R \tag{6.48}$$

$$\gamma(T) = \frac{c_p(T)}{c_p(T) - R} \tag{6.49}$$

Table 6.1 lists the data elements that define the `ig` class. For more information on how `.hpd` files are stored, see Section 5.2.2. Most of the essential data elements are self explanatory, but the coefficients and temperature limits must have compatible sizes. Even though the `ig` class only uses seven coefficients, there are eight provided in the NIST data sets. If there are N sets of coefficients defined over N temperature ranges, there must be N arrays of eight coefficients and $N+1$ temperature limit values. 5 For example, in a data set with two temperature ranges, the following would define a data set valid between `<T0>` and `<T2>`, and the transition between the two piece-wise data sets is at `<T1>`.

```
"Tlim":[<T0>, <T1>, <T2>]
"C":[
  [ <c0>, <c1>, <c2>, <c3>, <c4>, <c5>, <c6>, <c7> ],
  [ <c0>, <c1>, <c2>, <c3>, <c4>, <c5>, <c6>, <c7> ],
]
```

6.2.2 The NASA polynomial: `ig2`

The so-called "NASA polynomials" are a piece-wise empirical formulation to the specific heat of an ideal gas. They are taken from [11] predate the latest formulation of the NIST-JANAF tables, and are even used for reference. Unlike the Shomate equation, there is no $1/t^2$ term, there is no attempt to scale temperature prior to evaluating the polynomial, and properties are scaled with respect to the ideal gas constant.

$$c_p(T) = R\left(c_0 + c_1 T + c_2 T^2 + c_3 T^3 + c_4 T^4\right) \tag{6.50}$$

Table 6.1: HPD data file elements for the `ig` class

Name	Type	Description
`id`	`str`	Unique substance identifier string
`class`	`='ig'`	Class identifier string
`doc`	`str`	Documentation string
`atoms`	`dict`	Keys are elemental atoms, and values are their integer quantities in the substance.
`mw`	`float`	The molecular weight.
`Tlim`	`list`	A sorted list of $N+1$ floating point temperatures. Middle values define the boundaries between piece-wise coefficient ranges. High and low values define the limits of the model.
`C`	nested `list`	The value, $C[i][j]$, corresponds to coefficient c_j in the temperature interval `Tlim[i]` to `Tlim[i+1]`.
`TAB`	nested `list`	Optional table of truth values published by NIST used for validation

There are nearly identical formulations for enthalpy,

$$h(T) = R\left(c_0 T + \frac{c_1}{2}T^2 + \frac{c_2}{3}T^3 + \frac{c_3}{4}T^4 + \frac{c_4}{5}T^5 + c_5\right), \tag{6.51}$$

and entropy

$$s^\circ(T) = R\left(c_0 \ln(T) + c_1 T + \frac{c_2}{2}T^2 + \frac{c_3}{3}T^3 + \frac{c_4}{4}T^4 + c_6\right). \tag{6.52}$$

Here, just as in the Shomate equations, c_5 and c_6 are introduced as integration constants in enthalpy and entropy.

Internal energy is readily calculated from the definition of enthalpy in (6.21),

$$e(T) = h(T) - RT \tag{6.53}$$

There is a similarly simple relationship to determine constant-volume specific heat and specific heat ratio,

$$c_v(T) = c_p(T) - R \tag{6.54}$$

$$\gamma(T) = \frac{c_p(T)}{c_p(T) - R} \tag{6.55}$$

There is some overlap in the species offered by the two classes. A mild preference has been given to the older NASA polynomials for two reasons:

- Most of the NASA models have wider ranges of validity than the Shomate models.
- Some of the Shomate data have been found to suffer from discontinuity errors at the piecewise boundaries.

Otherwise, the two have been found to be sufficiently equivalent that most users will not find a reason to wonder.

Table 6.2 lists the data elements that define the `ig2` class. For more information on how `.hpd` files are stored, see Section 5.2.2. Most of the essential data elements are self explanatory, but the coefficients and temperature limits must have compatible sizes. Even though the

Table 6.2: HPD data file elements for the `ig2` class

Name	Type	Description
`id`	`str`	Unique substance identifier string
`class`	`='ig2'`	Class identifier string
`doc`	`str`	Documentation string
`atoms`	`dict`	Keys are elemental atoms, and values are their integer quantities in the substance.
`pref`	`float`	Reference pressure in Pascals
`mw`	`float`	The molecular weight.
`Tlim`	`list`	A sorted list of $N+1$ floating point temperatures: Middle values define the boundaries between piece-wise coefficient ranges. High and low values define the limits of the model.
`C`	nested `list`	The value, $C[i][j]$, corresponds to coefficient c_j in the temperature interval `Tlim[i]` to `Tlim[i+1]`.

`ig2` class only uses seven coefficients, there are eight provided in the NIST data sets. If there are N sets of coefficients defined over N temperature ranges, there must be N arrays of eight coefficients and $N+1$ temperature limit values.

For example, in a data set with two temperature ranges, the following would define a data set valid between `<T0>` and `<T2>`, and the transition between the two piece-wise data sets is at `<T1>`.

```
"Tlim":[<T0>, <T1>, <T2>]
"C":[
  [<c0>, <c1>, <c2>, <c3>, <c4>, <c5>, <c6>, <c7>],
  [<c0>, <c1>, <c2>, <c3>, <c4>, <c5>, <c6>, <c7>],
]
```

6.2.3 The ideal gas mixture: `igmix`

The PYroMat ideal gas mixture class defines methods to calculate properties of a static mixture of ideal gases. For improved computational

efficiency, properties like the mixture molecular weight, mole fractions, and mass fractions, are calculated ahead of time and stored in class instances for later use.

Section 6.1.8 already shows how properties of a mixture can be calculated from the properties of the constituent gases. The implementation of all but entropy is quite straightforward. In all of the ideal gas codes, the entropy at the reference pressure, s°, is calculated in a separate efficient internal method. For that reason, the `igmix` class calls on these methods directly and then treats the pressure dependence separately.

When calculating in molar units,

$$\overline{s}(T,p) = \sum \chi_i \; \overline{s}_i^\circ(T) - \sum \chi_i R_u \ln\left(\frac{p}{p_i^\circ}\right)$$

Note that in this formulation, the constituents are not required to have the same reference pressure. This can be dealt with by calculating an effective mixture reference pressure.

$$\begin{aligned}\sum \chi_i \ln\left(\frac{p}{p_i^\circ}\right) &= \sum \chi_i \left(\ln p - \ln p_i^\circ\right) \\ &= \ln p - \sum \chi_i \ln p_i^\circ \\ &= \ln\left(\frac{p}{p_{mix}^\circ}\right)\end{aligned}$$

when

$$p_{mix}^\circ = \exp\left(\sum \chi_i \ln p_i^\circ\right). \tag{6.56}$$

This is a log-weighted average of the constituent reference pressures. In the event that they are all the same, the mixture reference pressure is also the same.

It is less direct, but the same equivalent reference pressure can be derived from the mass-unit properties as well. So,

$$s(T,p) = \sum \chi_i \overline{s}_i^\circ(T) - R \ln\left(\frac{p}{p_{mix}^\circ}\right) \tag{6.57a}$$

$$\overline{s}(T,p) = \sum \chi_i \overline{s}_i^\circ(T) - R_u \ln\left(\frac{p}{p_{mix}^\circ}\right) \tag{6.57b}$$

Table 6.3: HPD data file elements for the `igmix` class

Name	Type	Description
`id`	`str`	Unique substance identifier string
`class`	`='igmix'`	Class identifier string
`doc`	`str`	Documentation string
`bymass`	`bool`	Indicates whether contents are listed by mass or by mole count
`contents`	`dict`	Specifies the mixture composition: Keys are pure ideal gas id strings, values are quantities. The quantities will be normalized after load, so they do not need to add to one.

Chapter 7

Multi-phase substance models

Unlike ideal gas properties, it is sensible to formulate multi-phase properties in terms of temperature and density instead of temperature and pressure.

Modeling phase changes with temperature and pressure as independent variables requires a discontinuity at the phase change. On the other hand, using density requires no such discontinuity. Using T, ρ also opens the possibility of extending the model to govern meta-stable states, which would require two values from a single T, p pair. Finally, in the molecular view of substances, density and temperature are descriptive of a fluid's local properties, while pressure is somewhat more contrived.

7.1 General formulation for `mp1`

The models supported by the `mp1` class have explicit formulations for the Holmholtz free energy (or simply free energy) in terms of temperature and density, and all other properties are calculated from that

formulation. Free energy is defined as

$$a(T,\rho) \equiv e(T,\rho) - Ts(T,\rho) \tag{7.1}$$

$$= h - \frac{p}{\rho} - Ts. \tag{7.2}$$

By constructing the formulation from a bank of terms inspired by theoretical formulae for the intermolecular forces, there is greater hope of reducing the number of terms needed.

The formulation that we describe in this chapter is sometimes referred to as a "Span and Wagner" fit for Helmholtz free energy. An expansive body of papers by Span, Wagner, Lemon, Jacobsen, and others provides a library of formulations for substances that use a standard bank of terms to construct the free energy equation.

7.1.1 Nondimensionalization

It is generally sound practice to nondimensionalize formulae in all but the most trivial numerical problems. Ensuring that parameters vary on the order of unity helps reduce the severity of numerical errors, and if the nondimensionalization is performed with special attention to the underlying physics, then it is highly likely that the complexity of the formulae required will be reduced.

All of the substance models in the first multi-phase class use

$$\alpha(\tau,\delta) = \frac{a(T,\rho)}{RT} \tag{7.3a}$$

$$\tau = \frac{T_c}{T} \tag{7.3b}$$

$$\delta = \frac{\rho}{\rho_c}. \tag{7.3c}$$

The free energy is normalized by the quantity RT, which is motivated by (6.14) from the study of ideal gases. Normalizing temperature and density by the critical point values acknowledges that the critical point is a natural scale for the important phenomena in this substance. The choice to make τ scale like the inverse of temperature is motivated by the Boltzmann and Maxwell distributions for molecular velocity, in which temperature appears in a denominator.

The formulation for dimensionless free energy, α, is split into two parts: the ideal gas part, α^o, and the residual part, α^r. So, the total free energy is

$$\alpha(\tau, \delta) = \alpha^o(\tau, \delta) + \alpha^r(\tau, \delta) \tag{7.4}$$

This approach separates the problem of needing to model the energy contained in molecular vibration and translation of a large number of independent oscillators (α^o) from the problem of modeling the effects of intermolecular forces with increasing density (α^r).

7.1.2 Ideal gas portion of free energy

The ideal gas portion of the free energy can be constructed from a specific heat model, just as other properties were for the ideal gas substances in Chapter 6. The ideal gas enthalpy is merely the integral of specific heat, and ideal gas entropy can be similarly constructed in terms of specific heat, temperature, and density,

$$h = h_0 + \int_{T_0}^{T} c_p \mathrm{d}T \tag{7.5a}$$

$$s = s_0 + \int_{T_0}^{T} \frac{c_p}{T} \mathrm{d}T - R \ln\left(\frac{\rho}{\rho_0}\right) - R \ln\left(\frac{T}{T_0}\right). \tag{7.5b}$$

When these are transposed into the dimensionless parameters, τ and δ,

$$h = h_0 - T_c \int_{\tau_0}^{\tau} \frac{c_p}{\tau^2} \mathrm{d}\tau$$

$$s = s_0 - \int_{\tau_0}^{\tau} \frac{c_p}{\tau} \mathrm{d}\tau - R \ln\left(\frac{\delta \tau_0}{\delta_0 \tau}\right).$$

When these are used to calculate the dimensionless free energy,

$$\begin{aligned} \alpha^o &= \frac{h - RT - Ts}{RT} = \frac{h\tau}{RT_c} - 1 - \frac{s}{R} \\ &= \frac{h_0}{RT_c}\tau - \frac{s_0}{R} - 1 + \ln\left(\frac{\delta\tau_0}{\delta_0\tau}\right) + \left(-\tau \int_{\tau_0}^{\tau} \frac{c_p}{R\tau^2} \mathrm{d}\tau + \int_{\tau_0}^{\tau} \frac{c_p}{R\tau} \mathrm{d}\tau\right) \end{aligned} \tag{7.6}$$

Note that, for ease of interpretation, we have adopted a lazy notation with τ both inside and outside of the integrals. We will adopt a more strict notation below.

The difficulty in formulating the specific heat of even an ideal gas is introduced in Section 1.1.7. It is, briefly, that portions of energy that a substance stores in translation (thermal energy) versus the many internal forms of vibration depends heavily on the temperature (and intermolecular distance for a real substance). A fully detailed model would need to include molecular vibration, electronic, magnetic energies, and quantom mechanical effects. Approaches to this problem range from a few relatively complicated terms derived from first principles to purely empirical approaches with many simple polynomial terms. In this discussion, we will spend a little time addressing those approaches that are most common to the ideal gas portion of the `mp1` model.

In the ideal gas models described so far in Chapter 6, generic empirical polynomials do a reasonable job of matching the phenomena that dominate there with the advantage that a couple of standard algorithms can consistently evaluate them. The caveat is that they require piece-wise definitions to adequately capture all of the relevant features of the curve, but the `mp1` model covers the entire temperature range with a single expression. Even though the residual terms are responsible for the intermolecular forces, low-temperature gas behaviors still need to be considered in the ideal gas portion. As a result, many high-performing published models tend to include a mix of polynomial and physics-based terms derived from quantum theory.

Models that include terms derived from quantum mechanics can compactly match the decrease in specific heat seen at low temperature. The specific heat of many independent oscillators assuming a distribution of discrete energies adopts the form

$$\frac{c_p(\tau)}{R} = \ldots + bm^2 \frac{\tau^2 \exp(m\tau)}{(\exp(m\tau) - 1)^2} + \ldots,$$

where b represents the magnitude of the feature, and m is a dimensionless temperature (normalized by T_c) at which the feature occurs.

The integrals of specific heat can be collectively simplified by inverting the integration-by-parts procedure with a series of substitutions, so

that

$$-\tau\int_{\tau_0}^{\tau}\frac{c_p(t)}{Rt^2}\mathrm{d}t+\int_{\tau_0}^{\tau}\frac{c_p(t)}{Rt}\mathrm{d}t=-\int_{\tau_0}^{\tau}\left[\int_{\tau_0}^{t}\frac{c_p(t')}{Rt'^2}\mathrm{d}t'\right]\mathrm{d}t. \tag{7.7}$$

This has the fortunate effect of canceling the τ^2 in the quantum component of specific heat, making its integral straightforward.

$$\begin{aligned}
&-\int_{\tau_0}^{\tau}\int_{\tau_0}^{t}\frac{bm^2\exp(mt')}{\left(\exp(mt')-1\right)^2}\mathrm{d}t'\mathrm{d}t\ldots\\
&\quad=\int_{\tau_0}^{\tau}\left[\frac{bm}{\exp(mt)-1}-\frac{bm}{\exp(m\tau_0)-1}\right]\mathrm{d}t\\
&\quad=\int_{\tau_0}^{\tau}\frac{bm\exp(-mt)}{1-\exp(-mt)}\mathrm{d}t-\frac{bm}{\exp(m\tau_0)-1}(\tau-\tau_0)\\
&\quad=b\ln(1-\exp(-m\tau))-b\ln(1-\exp(-m\tau_0))+\ldots\\
&\qquad-\frac{bm}{\exp(m\tau_0)-1}(\tau-\tau_0)
\end{aligned} \tag{7.8}$$

The same approach could be applied to the integration of the polynomial, but it is probably more easily dealt with in separate terms. For a sum of terms with arbitrary exponents,

$$\begin{aligned}
&-\tau\int_{\tau_0}^{\tau}\sum_k c_k t^{k-2}\mathrm{d}t+\int_{\tau_0}^{\tau}\sum_k c_k t^{k-1}\mathrm{d}t\ldots\\
&\quad=-\tau\left[c_1\ln t+\sum_{k\neq 1}\frac{c_k t^{k-1}}{k-1}\right]_{\tau_0}^{\tau}+\left[c_0\ln t+\sum_{k\neq 0}\frac{c_k t^k}{k}\right]_{\tau_0}^{\tau}\\
&\quad=-\tau c_1\ln\tau+\tau c_1\ln\tau_0-\left[\sum_{k\neq 1}\frac{c_k\tau^k}{k-1}\right]+\tau\left[\sum_{k\neq 1}\frac{c_k\tau_0^{k-1}}{k-1}\right]+\ldots\\
&\qquad+c_0\ln\tau-c_0\ln\tau_0+\left[\sum_{k\neq 0}\frac{c_k\tau^k}{k}\right]-\left[\sum_{k\neq 0}\frac{c_k\tau_0^k}{k}\right].
\end{aligned} \tag{7.9}$$

Note that this analysis does not require that values of k be limited to integers.

Collectively, these integrals motivate the form that is now commonplace in contemporary free-energy-based models,

$$\alpha^o(\tau,\delta) = \ln\delta + (c_0 + c_1\tau)\ln\tau + \alpha_0^o(\tau) + \alpha_1^o(\tau),. \tag{7.10a}$$

In many models, the $\tau\ln\tau$ term is omitted, implying that there was no linear term on τ (proportional to $1/T$) in the formulation of specific heat, but permitting it in the `mp1` model makes it highly extensible. The remaining terms are a polynomial on τ, α_0^o, and the quantum terms, α_1^o. which are only functions of temperature. The α_1^o term includes a sum of all of the quantum functions,

$$\alpha_1^o(\tau) = \sum_j b_j \ln\left(1 - \exp(-m_j\tau)\right), \tag{7.10b}$$

and the $\alpha_0^o(\tau)$ term is merely a polynomial. In many models, it is a constant and a linear term only, representing the terms that appear in (7.6).

Because much of the complexity of the specific heat is captured by the α_1^o terms, the polynomial, α_0^o, may only contain a constant and a linear term. Still, exponentials and logarithms are numerically expensive, so this model is usually substantially slower than the polynomial ideal gas models.

In the data files, the ideal gas terms are contained in a dictionary called the `"AOgroup"`. An `"AOgroup"` entry in the file might appear like the example below. For more information on data files, see Section 5.2.2. For more information on polynomial coefficient lists, see Section 8.2.

```
"AOgroup" : {
    "Tscale" : <Tc>,
    "dscale" : <dc>,
    "logt" : <c0>,
    "tlogt" : <c1>,
    "coef0" : <p coefficients>,
    "coef1" : [[<b0>, <m0>], [<b1>, <m1>], <...>]
}
```

7.1.3 Residual portion of free energy

The residual (or real-fluid) portion of free energy accounts for the intermolecular forces that defy the ideal gas assumption. Many of these terms become especially important when the substance density is high. The model is divided into three groups of terms,

$$\alpha^r = \alpha_0^r(\tau,\delta) + \alpha_1^r(\tau,\delta) + \alpha_2^r(\tau,\delta) \tag{7.11a}$$

Many of these terms are physically motivated, but that discussion is not included here.

The first group of terms is a series of polynomials multiplied by exponentials of powers of density,

$$\alpha_0^r = \sum_{k=0}^{K} \exp\left(-\delta^k\right) p_k(\tau,\delta) \tag{7.11b}$$

$$p_k = \sum_i c_i \tau_i^a \delta_i^b. \tag{7.11c}$$

When $k = 0$ the exponential term is constant and may alternately be absorbed into the corresponding polynomial, $_0$. The polynomials rarely (if ever) have terms where the δ exponent, b, is zero, so α_0^r terms where $k \neq 0$ tend to vanish near $\delta \to 0$ and $\delta \to \infty$. These terms model the complicated behaviors that occur near the phase change (when δ is on the order 1).

The first group (α_0^r) usually constitutes the largest number of terms (often dozens). The exponents in the polynomial expansion are usually integers, but they are occasionally rationals, which permits efficient numerical evaluation. Many models do not include the terms in α_1^r or α_2^r, and most that do will only permit a few of them.

The second and third groups are Gaussian functions multiplied by powers of τ, δ, or function, Δ. The second group, α_1^r, centers the Gaussian function around temperature and density, γ and ϵ. These are usually set near the critical point. The third group, α_2^r, is a gaussian centered on exactly the critical point, but the Δ function is a scaled

distance from the critical point.

$$\alpha_1^r = \sum_{l=0}^{L} c_l \delta^{d_l} \tau^{t_l} \exp\left(-a_l(\delta - \epsilon_l)^2 - b_l(\tau - \gamma_l)^2\right) \tag{7.11d}$$

$$\alpha_2^r = \sum_{m=0}^{M} c_m {\Delta_m}^{b_m} \delta \exp\left(-C_m(\delta - 1)^2 - D_m(\tau - 1)^2\right) \tag{7.11e}$$

$$\Delta_m = \left[1 - \tau + A_m\left((\delta - 1)^2\right)^{1/2\beta_m}\right]^2 + B_m\left((\delta - 1)^2\right)^{a_m}. \tag{7.11f}$$

At first glance, these formulations appear to be an inefficient tangled mess of nested exponents. However, quantities like $\tau - 1$ and $\delta - 1$ may be positive or negative, and the exponents are not integers. It is numerically elegant to square these quantities (requiring just a multiplication operation) before raising them to decimal or fractional powers.

In the data files, these terms are encoded in a format shown below. The `coef0` member contains a list of 2D polynomial coefficients. Each polynomial, p_k, corresponds to an exponential term, $\exp(-\delta^k)$. The `coef1` and `coef2` members are nested lists with each containing the necessary coefficients and exponents to construct the terms of α_1^r and α_2^r respectively.

```
"ARgroup" : {
    "Tscale" : <Tc>,
    "dscale" : <dc>,
    "coef0" : [<p0 coef>, <p1 coef>, ...]
    "coef1" : [[<t>, <d>, <b>, <a>, <gamma>, <epsilon
       >, <c>], ...]
    "coef2" : [[<a>, <b>, <beta>, <A>, <B>, <C>, <D>,
        <c>], ...]
}
```

For more information on polynomial coefficients, see Section 8.1. It is worth emphasizing that `coef0` is a four-deep nested list.

```
"coef" : [# This begins the list of polynomials
            [# This begins the first polynomial
                [# First sub-polynomial
                    [<a>, <b>],
```

```
                    [<alpha>, <beta>],
                    [<i0>, <j0>, <c0>],
                    <...>
                ], <...more sub-polynomials?...>
            ],
            [# This begins the exp(-delta) term
                <...polynomial definition...>
            ],
            [# This begins the exp(-delta**2) term
                <...polynomial definition...>
            ],
            <...>
        ]
```

7.2 Calculation of properties

The `mp1` class calculates all substance properties from temperature, density, α, and its derivatives. In this section, the formulae that are used to calculate the various properties are developed from first principles.

Validating a code to efficiently and reliably calculate derivatives of the above functinos is a substantial task. Some discussion is afforded the problem of evaluating polynomials in Section 8.1, but term-by-term details for how this is done in the `mp1` class is beyond the scope of this document. For readers concerned with this question, in-line comments in the original code are intended to be instructional.

In this development, it will be useful to have this form of the first law:

$$T\mathrm{d}s = \mathrm{d}e - \frac{p}{\rho^2}\mathrm{d}\rho. \tag{7.12}$$

By definition,

$$a = e - Ts. \tag{7.13}$$

Its derivatives with respect to temperature and density may be simpli-

fied with some help from (7.12),

$$\left(\frac{\partial a}{\partial T}\right)_\rho = \left(\frac{\partial e}{\partial T}\right)_\rho - s - T\left(\frac{\partial s}{\partial T}\right)_\rho = -s \tag{7.14}$$

$$\left(\frac{\partial a}{\partial \rho}\right)_T = \left(\frac{\partial e}{\partial \rho}\right)_T - T\left(\frac{\partial s}{\partial \rho}\right)_T = \frac{p}{\rho^2}. \tag{7.15}$$

The properties below are calculated in a dimensionless form. In this way, this document is relieved from needing to be sensitive to units – including molar versus mass.

7.2.1 Pressure

The pressure (and compressibility factor) can be calculated explicitly from (7.15),

$$\begin{aligned}\frac{p}{\rho RT} &= \frac{\rho}{RT}\left(\frac{\partial a}{\partial \rho}\right)_T \\ &= \delta\alpha_\delta.\end{aligned} \tag{7.16}$$

7.2.2 Entropy

Entropy appears explicitly in (7.14), so it only needs to be transformed into terms of α. First, it is helpful to observe that derivatives with respect to temperature may be transposed into derivatives with respect to τ

$$\frac{\partial}{\partial T} = -\frac{\tau}{T}\frac{\partial}{\partial \tau}.$$

Therefore,

$$\begin{aligned}s &= -\left(\frac{\partial a}{\partial T}\right)_\rho \\ &= -\frac{\partial}{\partial T}RT\alpha \\ &= -R\alpha + R\tau\alpha_\tau\end{aligned} \tag{7.17}$$

So, normalized by R, entropy is

$$\frac{s}{R} = \tau\alpha_\tau - \alpha \tag{7.18}$$

7.2.3 Internal energy

Internal energy can be calculated in terms of a and s from (7.13) and (7.18),

$$\frac{e}{RT} = \frac{a}{RT} + \frac{s}{R} = \tau\alpha_\tau. \tag{7.19}$$

7.2.4 Enthalpy

Enthalpy is trivial to construct from (7.16) and (7.19). By definition,

$$\begin{aligned}\frac{h}{RT} &= \frac{e}{RT} + \frac{p}{\rho RT} \\ &= \tau\alpha_\tau + \delta\alpha_\delta.\end{aligned} \tag{7.20}$$

7.2.5 Specific heats

Constant-volume specific heat is obtained by differentiating internal energy by temperature.

$$\begin{aligned}c_v &= \frac{\partial e}{\partial T} = e_\tau \frac{\mathrm{d}\tau}{\mathrm{d}T} \\ &= -e_\tau \frac{\tau}{T}\end{aligned}$$

From (7.19), e is simply $RT_c\,\alpha_\tau$, so

$$\frac{c_v}{R} = -\tau^2\alpha_{\tau\tau}. \tag{7.21}$$

Constant-pressure specific heat is not so simply obtained since pressure is not one of the independent variables of the formulation. Derivations for c_p usually begin with enthalpy, but it will become aparent that the calculation benefits from the prior derivation for c_v. Therefore, from (1.12) we may write

$$c_p = \left(\frac{\partial e}{\partial T}\right)_p - \frac{p}{\rho^2}\left(\frac{\partial \rho}{\partial T}\right)_p.$$

However, because the `mp1` formulation is always in terms of temperature and density, derivatives with constant pressure must be transposed to derivatives on T and ρ,

$$\left(\frac{\partial e}{\partial T}\right)_p = \left(\frac{\partial e}{\partial T}\right)_\rho + \left(\frac{\partial e}{\partial \rho}\right)_T \left(\frac{\partial \rho}{\partial T}\right)_p.$$

Of course, the same must be done for the derivative of density,

$$\left(\frac{\partial \rho}{\partial T}\right)_p = -\frac{\left(\frac{\partial p}{\partial T}\right)_\rho}{\left(\frac{\partial p}{\partial \rho}\right)_T}.$$

Substituting,

$$c_p = \left(\frac{\partial e}{\partial T}\right)_\rho - \left(\left(\frac{\partial e}{\partial \rho}\right)_T - \frac{p}{\rho^2}\right) \frac{\left(\frac{\partial p}{\partial T}\right)_\rho}{\left(\frac{\partial p}{\partial \rho}\right)_T}.$$

The first term (the derivative of interneral energy with respect to temperature) is merely c_v. The second term may be simplified using (7.12), so

$$c_p = c_v - T\left(\frac{\partial s}{\partial \rho}\right)_T \frac{\left(\frac{\partial p}{\partial T}\right)_\rho}{\left(\frac{\partial p}{\partial \rho}\right)_T}.$$

Finally, this is in a form that can be conveniently substituted for free energy. Using (7.14) and (7.15),

$$\begin{aligned}
\left(\frac{\partial s}{\partial \rho}\right)_T &= -\frac{\partial^2 a}{\partial T \partial \rho} = -\frac{R}{\rho}\left(\delta\alpha_\delta - \tau\delta\alpha_{\tau\delta}\right) \\
\left(\frac{\partial p}{\partial T}\right)_\rho &= \rho^2 \frac{\partial^2 a}{\partial T \partial \rho} = R\rho\left(\delta\alpha_\delta - \tau\delta\alpha_{\tau\delta}\right) \\
\left(\frac{\partial p}{\partial \rho}\right)_T &= 2\rho\left(\frac{\partial a}{\partial \rho}\right)_T + \rho^2\left(\frac{\partial^2 a}{\partial \rho^2}\right)_T \cdots \\
&= RT\left(2\delta\alpha_\delta + \delta^2\alpha_{\delta\delta}\right)
\end{aligned}$$

So,

$$\frac{c_p}{R} = \frac{c_v}{R} + \frac{\left(\delta\alpha_\delta - \tau\delta\alpha_{\tau\delta}\right)^2}{2\delta\alpha_\delta + \delta^2\alpha_{\delta\delta}} \tag{7.22}$$

7.2.6 Liquid-vapor line

Technically, the Maxwell conditions for phase equilibrium provide enough information for an algorithm to iteratively calculate the temperature and densities (and all other properties) from these properties. However, the process is sufficiently numerically expensive to demand the use of additional empirical curves so the saturation line can be efficiently calculated. This means that there are tiny numerical discrepancies between the saturation properties and the *actual* saturation properties that one would predict using the Maxwell criteria.

Formulae are provided for

- liquid saturation density given temperature,
- vapor saturation density given temperature,
- saturation pressure given temperature.

From them, all other properties can be obtained along the saturation line. The `mp1` class exposes these properties through methods: `ds`, `ps`, `Ts` (not to be confused with `T_s`), `es`, `hs`, and `ss`. All of these depend on the three basic empirical formulae.

These formulae are calculated from dimensionless temperature,

$$\theta = \frac{T}{T_c}, \tag{7.23}$$

which is the inverse of τ. Each of the formulae for a saturation property, p, take one of four types:

0. A polynomial on θ

$$p(\theta) = \sum_k c_k \theta^{a_k} \tag{7.24}$$

This form is commonly used for liquid-solid saturation lines (not yet implemented in version 2.1.0).

1. A polynomial on $1 - \theta$

$$p(\theta) = \sum_k c_k (1-\theta)^{a_k} \tag{7.25}$$

This form is usually used for the saturation pressure.

2. An exponential of a polynomial on $1-\theta$

$$p(\theta) = \exp\left(\sum_k c_k (1-\theta)^{a_k}\right) \tag{7.26}$$

This form is usually used for the liquid saturation density.

3. An exponential of a polynomial on $1-\theta$ with a $1/\theta$ multiplier

$$p(\theta) = \exp\left(\frac{1}{\theta}\sum_k c_k (1-\theta)^{a_k}\right) \tag{7.27}$$

This form is usually used for the vapor saturation density.

In data, there are three groups that are used to define the saturation pressure and densities.

```
"PSgroup" : {
    "fn" : <fn index>,
    "pscale" : <p scale>,
    "Tscale" : <T scale>,
    "coef" : <polynomial coefficients>
}

"DSLgroup" : {
    "fn" : <fn index>,
    "dscale" : <d scale>,
    "Tscale" : <T scale>,
    "coef" : <polynomial coefficients>
}

"DSVgroup" : {
    "fn" : <fn index>,
    "dscale" : <d scale>,
    "Tscale" : <T scale>,
    "coef" : <polynomial coefficients>
}
```

The `fn` parameter is an index (0-3) that selects the form of the formula to use. The `pscale` and `dscale` parameters are used to re-scale the dimensionless property calculated by the selected formula.

The `Tscale` parameter is used to normalize temperature to calculate θ. Finally, `coef` is a polynomial coefficient list described in Section 8.2.

Chapter 8

Numerical routines

PYroMat is designed with a number of custom back-end numerical routines. Even though low-level custom numerical routines composed in plain Python are unlikely to ever compete well with compiled numerical binaries that have been developed over decades, there are distinct advantages to this approach.

- Reducing the number of dependencies makes the installation simpler and less likely to exhibit mysterious problems on other systems,
- The installation footprint is minimal - not requiring entire tertiary libraries,
- Algorithms can be customized to conform to the inputs and outputs required by the PYroMat.
- Numerical algorithms can be (and have been) tuned specifically to provide consistent convergence for the types of numerical problems encountered in thermodynamic property evaluation.
- The PYroMat numerical routines are specially designed to iterate on *parts* of a Numpy array at a time while still benefiting from Numpy's compiled algorithm speed.

A package written in plain Python is unlikely to ever be used in numerically intense applications like computational fluid dynamics, so

we have consistently prioritized reliability over speed. Still, that's no reason to be wasteful with clock cycles! Even on arrays with thousands of elements, the most cumbersome of PYroMat property calculations reliably converge in fractions of a second in testing. The author has yet to hear of a use case for which this performance is unacceptably slow. Until that changes, this is how the PYroMat back-end will continue to do things.

8.1 Polynomials of two variables

The evaluation of the multi-phase models requires efficient evaluations of polynomials of two variables. These expansions are typically of the form

$$P(X,Y) = \sum_{i,j} c_{i,j} X^i Y^j \tag{8.1}$$

where a and b are real coefficients such that i and j are integer indices.

8.1.1 Modifying polynomials for non-integer and negative powers

Fractional and negative exponents are also possible within this framework if we were to accept input values x, and y, and adjust them according to pre-exponentials a and b,

$$X = x^a \tag{8.2}$$

$$Y = y^b. \tag{8.3}$$

Then, if the polynomial were also multiplied by post-exponential terms, x^α, and y^β, the new polynomial formed is

$$p(x,y) = x^\alpha y^\beta P(X,Y). \tag{8.4}$$

This results in a polynomial,

$$p(x,y) = \sum_{i,j} c_{i,j} x^{ai+\alpha} y^{bj+\beta}, \tag{8.5}$$

but in a form that can be efficiently evaluated using integer exponents, i and j.

Non-integer exponents are problematic to polynomials, because they require the use of a `power` function, while integer exponents can be evaluated exclusively with multiplication. For example, consider a polynomial,

$$\begin{aligned} p(x,y) &= 2x^{-2} + x^{0.1}y + xy^2. \\ X &= x^{0.1} \\ p(x,y) &= x^{-2}\left(2 + X^{21}y + X^{40}y^2\right) \end{aligned}$$

Using scaling values $a = 0.1$ and $\alpha = -20$, the polynomial can be adjusted to be evaluated with only integer exponents.

However, this also illustrates a potential problem in the approach. For individual elements a call to Numpy's `power` funciton costs about double a single multiplication, but for larger arrays (when the difference is most important) the difference diverges. Figure 8.1 shows a timing study of `power`, `*`, and ** operations on arrays of random positive floating point numbers. For small arrays, the interpreter's overhead dominates the measurement, but above 100 elements, the two adopt classical power curves.

Because of the dominance of numerical overhead, repeated multiplication operations can be more expensive than a call to `power` on small arrays. However, it should be emphasized that judicious use of computational effort is really only critical on large arrays, anyway. For arrays with 100 elements, `power` operations are about 10 times as expensive, but `power` function calls on 10,000-element arrays are roughly equivalent to 100 multiplications.

This leads to a careful choice for arrays with large exponents (or tiny fractional values). The large exponents in the last example can be mitigated at the cost of added overhead and complexity by splitting the polynomial into two separate polynomials, each with its own scaling.

$$\begin{aligned} p(x,y) &= 2x^{-2} + x^{0.1}y + xy^2. \\ &= x^{-2}\left(2 + x^3y^2\right) + x^{0.1}y \end{aligned} \tag{8.6}$$

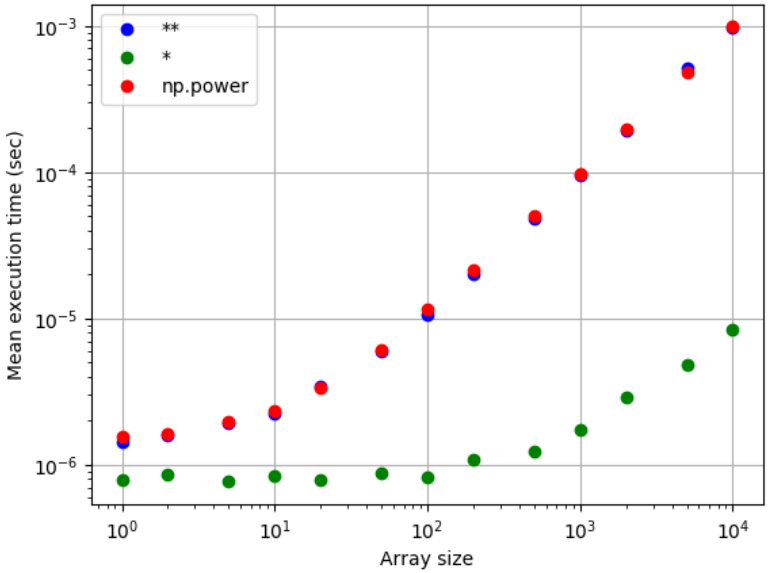

Figure 8.1: Approximate calculation time of the numpy `power` function, `**` operation, and the `*` operation on arrays of random positive floating point values.

8.1.2 Representation of polynomials in data

The `json` data representation of a polynomial on two dimensions uses three-deep series of nested lists to describe the pre- and post-exponents, the coefficients, and their corresponding exponents for multiple polynomial parts. For example, the nested lists below define a polynomial with multiple parts.

```
poly = [# This begins the list of sub-polynomials
    [# This begins the first sub-polynomial
        [<a>,<b>],
        [<alpha>, <beta>],
        [<i0>, <j0>, <c0>],
        [<i1>, <j1>, <c1>],
        <...>
    ],
    [# This begins the second sub-polynomial
        [<a>,<b>],
        [<alpha>, <beta>],
        [<i0>, <j0>, <c0>],
        [<i1>, <j1>, <c1>],
        <...>
    ],
    <...more sub-polynomials...>
]
```

The exponents `i0`, `j0`, `i1`, etc., must be non-negative integers listed in descending order, sorted by i first and then sorted by j. An alternative approach is to define a two-dimensional polynomial with a matrix of coefficients, each row and column corresponding to a power of x or y, but that method suffers when there are only a few terms with high powers. This approach might be called a "sparse coordinate matrix" approach to defining a polynomial.

8.1.3 Efficient evaluation of the polynomial

Once a polynomial is adjusted to use only non-negative integer exponents.

$$p(x, y) = \sum_{i,j} c_{i,j} x^i y^j \tag{8.7}$$

However, evaluating each term individually requires two expensive calls to a `pow` function and two floating point multiplications.

The widely accepted method for evaluating a polynomial of one variable is to construct a recursive expansion

$$q(y) = c_0 + y(c_1 + y(c_2 + y(\dots \tag{8.8}$$

If there are n coefficients, then this amounts to only n multiplications with no `pow` calls. In order to extend this algorithm to two variables, more elegant notation will be helpful. If we name the intermediate value calculated in the process of these recursions q, then a polynomial with n terms implies the series

$$q_n = c_n \tag{8.9}$$
$$q_j(y) = c_j + y\, q_{j+1}(y) \tag{8.10}$$
$$q_0(y) = q(y). \tag{8.11}$$

This is a series beginning with $j = n$ and proceeding backwards to $j = 0$. The final function in the series, q_0 is also the desired polynomial, $q(y)$. In practice, there is no need to keep the old values of q, so a single register may be used to hold the latest value.

How can this be extended to a polynomial of two variables? We may consider the polynomials to be nested; the evaluation of a polynomial on Y determines the individual coefficients for a polynomial on X.

$$p(x, y) = \sum_i q_i(y) x^i \tag{8.12}$$
$$q_i(y) = \sum_j c_{i,j} y^j \tag{8.13}$$

We only need a minor modification to the intermediate values for the x polynomial since there will be a separate expansion for each value of i. If there are n j terms,

$$q_{i,n}(y) = c_{n,j} \tag{8.14a}$$
$$q_{i,j}(y) = c_{i,j} + y\, q_{i+1,j}(y) \tag{8.14b}$$
$$q_{i,0}(y) = q_i(y). \tag{8.14c}$$

If there are m x-terms,

$$p_m(x,y) = q_m(y) \tag{8.15a}$$
$$p_i(x,y) = q_i(x) + y\, p_{i+1}(x,y) \tag{8.15b}$$
$$p_0(x,y) = p(x,y). \tag{8.15c}$$

8.1.4 Efficient evaluation of derivatives

The partial derivatives of the polynomial can be efficiently evaluated along with the polynomial itself. To relax the already cumbersome notation, the functional dependencies (y) and (x,y) will be dropped. For the purpose of thermodynamic property evaluation, the first two derivatives will suffice.

Let us begin with the simpler task of calculating the derivatives of q_i with respect to y.

$$q_{i,n|y} = 0 \tag{8.16a}$$
$$q_{i,j|y} = q_{i+1,j} + y\, q_{i+1,j|y} \tag{8.16b}$$
$$q_{i,0|y} = q_{i|y} \tag{8.16c}$$

$$q_{i,n|yy} = 0 \tag{8.17a}$$
$$q_{i,j|yy} = 2q_{i,j+1|y} + y\, q_{i,j+1|yy} \tag{8.17b}$$
$$q_{i,0|yy} = q_{i|yy} \tag{8.17c}$$

The derivatives on p are constructed somewhat differently because they can be in both x and y. Beginning with y,

$$p_{n|y} = 0 \tag{8.18a}$$
$$p_{j|y} = q_{i|y} + x\, p_{j+1|y} \tag{8.18b}$$
$$p_{0|y} = p_y \tag{8.18c}$$

$$p_{m|yy} = 0 \tag{8.19a}$$
$$p_{i|yy} = q_{i|yy} + x\, q_{i+1|yy} \tag{8.19b}$$
$$p_{0|yy} = p_{yy} \tag{8.19c}$$

The derivatives on x appear

$$p_{m|x} = 0 \tag{8.20a}$$
$$p_{i|x} = p_{i+1} + x\, p_{i+1|x} \tag{8.20b}$$
$$p_{0|x} = p_x \tag{8.20c}$$

$$p_{n|xx} = 0 \tag{8.21a}$$
$$p_{i|xx} = 2p_{i+1|x} + x\, p_{i+1|xx} \tag{8.21b}$$
$$p_{0|xx} = p_{xx} \tag{8.21c}$$

Finally, the cross-term (both x and y) appears

$$p_{n|xy} = 0 \tag{8.22a}$$
$$p_{i|xy} = p_{i+1|y} + y\, p_{i+1|xy} \tag{8.22b}$$
$$p_{0|xy} = p_{xy} \tag{8.22c}$$

8.1.5 Implementation of the algorithm

In practice, this cumbersome notation can be drastically simplified in code because it is not necessary to distinguish between the subscripts of p and q, provided care is taken not to overwrite a value before it is needed.

In most practical polynomials of two variables of order m on x and order n on y, there are $(m+1)(n+1)$ possible coefficients, but many (if not most) of them may be zero. As a result, storing the coefficient array in a 2D array (or matrix) format is not efficient. Instead, PYroMat takes an approach closer to coordinate sparse matrix storage.

If we have one-dimensional arrays of polynomial coefficients, c_k, and exponents, i_k and j_k, the polynomial will be constructed as

$$p(X,Y) = \sum_{k}^{N-1} c_k X^{i_k} Y^{i_k}. \tag{8.23}$$

In this way, the polynomial

$$p(X,Y) = -0.1X^2 + XY + 0.5Y^2 - Y - 0.2 \tag{8.24}$$

may be represented by

$$i = [2,\ 1,\ 0,\ 0,\ 0] \tag{8.25}$$
$$j = [0,\ 1,\ 2,\ 1,\ 0] \tag{8.26}$$
$$c = [-0.1,\ 1,\ 0.5,\ -1,\ 0.2] \tag{8.27}$$

For the algorithm to function efficiently, it is reasonable to impose some prior sorting of the exponent values. Since the series developed in the previous section requires that we interact with higher-order terms first, let us assert that the polynomial should be expressed in order of descending exponents on X and then Y.

In Algorithm 1, an outer loop over the range of values of i and an inner loop on the range of values of j considers all of the possible terms in the polynomial. If coefficients are absent from the arrays (if the i, j pair is not found), then the term is not included in the polynomial (the coefficient is zero). Starting with the maximum value for each exponent, the indices are reduced incrementally until the i, j combination corresponding to the next row (k) is found.

8.2 Polynomials in one dimension

Polynomials of one dimension benefit from the same approach as two-dimensional polynomials, but the algorithm is far simpler. All the formulae and algorithms that apply to q in the last section may simply be repurposed for a single dimensional polynomial.

The only topic that really requires special treatment is the format used in data. One-dimensional polynomials are identical to two-dimensional polynomials, except (1) the pre- and post-exponent values are scalars instead of two-element lists, and (2) each term only needs two parameters instead of three.

```
coef = [ # begins the polynomial
    [ # begins a sub-polynomial
        <a>,
        <alpha>,
        [ <i0>, <c0> ],
        [ <i1>, <c1> ],
        <...>
```

Algorithm 1 Efficient evaluation of a polynomial of two variables

1: **procedure** _POLY2(x, y, i, j, c) ▷ i, j, c are arrays
2: $p, p_x, p_y, p_{xx}, p_{xy}, p_{yy} \leftarrow 0$ ▷ Initialize the results with zero
3: $i_{max} \leftarrow i[0]$ ▷ Detect the maximum i value
4: $k \leftarrow 0$ ▷ k is our place in the coefficient array
5: **for** $ii = i_{max}$ to 0 **do** ▷ All possible i values
6: **if** $k < N$ and $i[k] == ii$ **then** ▷ Is there an x^{ii} term?
7: $j_{max} \leftarrow j[k]$ ▷ Detect the maximum j value
8: $q, q_y, q_{yy} \leftarrow 0$ ▷ Initialize the inner q result
9: **for** $jj = j_{max}$ to 0 **do** ▷ All possible j values
10: $q_{yy} \leftarrow 2q_y + yq_{yy}$
11: $q_y \leftarrow q + yq_y$
12: **if** $k < N$ and a_k is i and b_k is j **then** ▷ Is there a $x^{ii}y^{jj}$ term?
13: $q \leftarrow c_k + yq$
14: $k \leftarrow k + 1$
15: **else** ▷ Term not in polynomial
16: $q \leftarrow yq$
17: **end if**
18: **end for**
19: $p_{yy} \leftarrow q_{yy} + xp_{yy}$
20: $p_{xx} \leftarrow 2p_x + xp_{xx}$
21: $p_{xy} \leftarrow p_y + xp_{xy}$
22: $p_x \leftarrow p + xp_x$
23: $p_y \leftarrow q_y + xp_y$
24: $p \leftarrow q + xp$
25: **else** ▷ Term not in polynomial
26: $p_{yy} \leftarrow xp_{yy}$
27: $p_{xx} \leftarrow 2p_x + xp_{xx}$
28: $p_{xy} \leftarrow p_y + xp_{xy}$
29: $p_x \leftarrow p + xp_x$
30: $p_y \leftarrow xp_y$
31: $p \leftarrow xp$
32: **end if**
33: **end for**
34: **return** $p, p_x, p_y, p_{xx}, p_{xy}, p_{yy}$
35: **end procedure**

```
    ],
    [ # begins a second sub-polynomial
        <a>,
        <alpha>,
        [ <i0>, <c0> ],
        [ <i1>, <c1> ],
        <...>
    ],
    <... more sub-polynomials?...>
]
```

8.3 Iteration with `iter1`

Most of the classes have a member that implements some variation of the Newton-Rhapson variant `_iter1` method. It is fast, and it has excellent convergence characteristics in most applications except the numerically irksome cases found in the multi-phase models.

Given a function on one variable, f, the `_iter1` algorithm seeks a value, x, between upper and lower limits, x_{max} and x_{min} so that $f(x)$ is equal to a reference value, y.

Given a guess, x_k, traditional Newton-Rhapson iteration calculates a next guess, x_{k+1} by extrapolating linearly,

$$\Delta x = \frac{y - f(x_k)}{f'(x_k)} \tag{8.28}$$

$$x_{k+1} = x_k + \Delta x. \tag{8.29}$$

This is simply repeated until $y - f(x_k)$ is sufficiently small.

However, in problems with substantial nonlinearities, the iteration can diverge badly like in Figure 8.2. Worse still, these intermediate guesses can even wander to illegal property values.

The `_iter1` algorithm addresses the problem by allowing the calling property method to establish upper and lower boundaries on the value, x. If a guess wanders outside of the valid range, the size of the step, Δx, is halved repeatedly until the next guess is back in the valid range. The algorithm is usually initialized externally with some first guess for x.

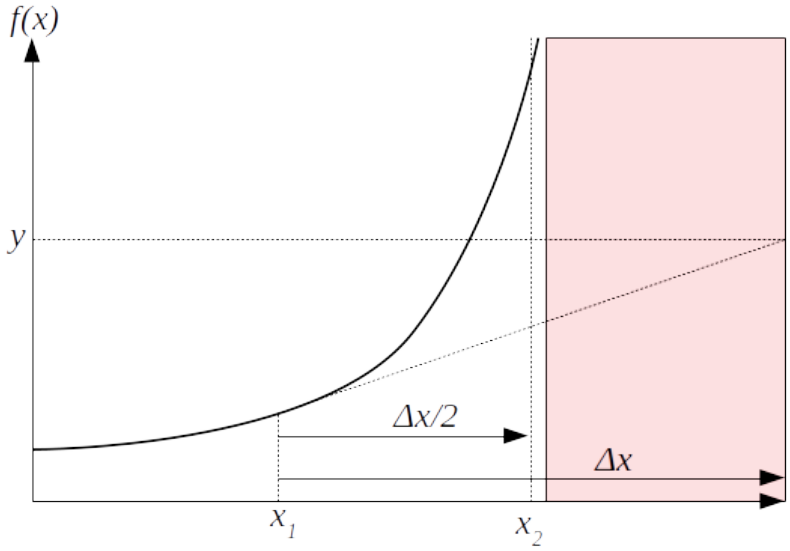

Figure 8.2: An example of a system where classical Newton iteration diverges to illegal values. The

Algorithm 2 ITER1: Modified bounded Newton-Rhapson iteration, $y = f(x)$

procedure _ITER1$(f, y, x_{min}, x_{max}, x, \epsilon)$
  $x_k \leftarrow x$ ▷ Initialize the current guess
  $y_k \leftarrow f(x)$ ▷ Evaluate the function
  $y'_k \leftarrow f'(x)$
  while $|y - y_k| > \epsilon$ **do** ▷ While error is large
    $\Delta x \leftarrow (y - y_k)/y'_k$ ▷ Calculate step size
    $x_{k+1} \leftarrow x_k + \Delta x$ ▷ Calculate the next guess
    ▷ If x_k is out of bounds
    while $x_{k+1} > x_{max}$ or $x_{k+1} < x_{min}$ **do**
      $\Delta x \leftarrow \Delta x/2$ ▷ Halve Δx
      $x_{k+1} \leftarrow x_k + \Delta x$ ▷ Update the guess
    end while
    $x_k \leftarrow x_{k+1}$ ▷ Accept the new guess
    $y_k \leftarrow f(x_k)$ ▷ Evaluate the function
    $y'_k \leftarrow f'(x_k)$
  end while
  return x_k
end procedure

Algorithm 2 gives a simplified version of the algorithm implemented in PYroMat. See the in-line documentation and comments for a more detailed description.

8.4 Iteration with `hybrid1`

Multi-phase properties present severe numerical challenges since the phase change represents an abrupt jump in properties, and the iteration is inherently two-dimensional. The `_iter1` algorithm prevents the solution from diverging, but it often fails to converge in multi-phase substance inversion problems.

8.4.1 Bisection iteration

Bisection is a classic approach to nasty inversion problems where Newton iteration fails. If upper and lower bounds, x_a and x_b are found so that $f(x_a) < y < f(x_b)$, then a continuous function must have a value somewhere between x_a and x_b such that $f(x) = y$. If we were to divide the domain in half $x_c = (x_a + x_b)/2$ and evaluate the function there, its value will either be above or below y, so one of the two values could be replaced by x_c. In this way, the domain between x_a and x_b is reliably cut in half with every iteration step, no matter how bizarrely $f(x)$ behaves.

This process just has to be repeated until the distance between the upper and lower bounds have shrunk to be so small that the numerical uncertainty is acceptable. Figure 8.3 shows three steps using this process. The vertical space occupied by each blue box represents the shrinking uncertainty for the value of $f(x)$ and the horizontal space represents the shrinking uncertainty in x.

While Newton iteration might converge in only a few steps on a nearly linear function, bisection can take tens of steps depending on the ratio of the initial domain and the acceptable error range. Since the domain is divided by two every time, it is easy to calculate the number of iterations to obtain a certain domain size.

$$N = \log_2 \frac{|x_b - x_a|}{\epsilon} \tag{8.30}$$

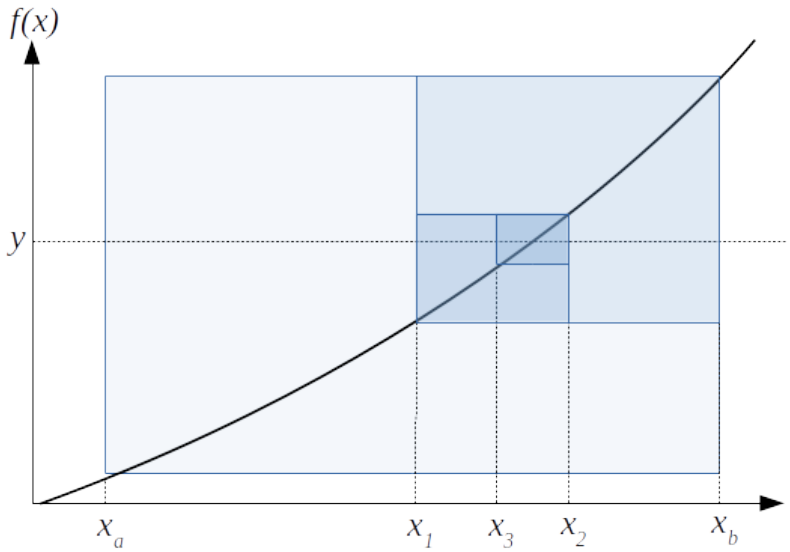

Figure 8.3: A depiction of the bisection iteration algorithm on a hypothetical function.

If the initial domain were a temperature range of 0 to 1000 Kelvin, and the acceptable error were .01 Kelvin, 17 iterations are required. If the acceptable error were tightened to .001 Kelvin, 20 iterations are required!

8.4.2 The `hybrid1` candidates

Literature on numerical methods is littered with different flavors of hybrid algorithms that try to benefit from the guaranteed convergence of bisection and the speed of Newton-Rhapson. This algorithm is specifically designed for application on the types of functions that appear in thermodynamic properties.

This hybrid approach begins by bracketing a solution with guesses, x_a and x_b, just like a bisection algorithm. Then, we calculate three candidates for the next guess, x_c:

1. Newton iteration from x_a:

$$x_1 = x_a + \frac{y - f(x_a)}{f'(x_a)} \tag{8.31}$$

2. Newton iteration from x_b:

$$x_2 = x_b + \frac{y - f(x_b)}{f'(x_b)} \tag{8.32}$$

3. Bisection iteration from x_a and x_b:

$$x_3 = \frac{x_a + x_b}{2} \tag{8.33}$$

These three parallel processes for calculating candidates are shown in Figure 8.4. As the next sections will explore, the next step is to select one. Once the function is evaluated there, the boundaries can be updated (just like in bisection) and convergence can be tested.

One could pose dozens of ideas for how information about these three candidates might be used to select a best guess, but only two are implemented in PYroMat. They are discussed in the sections below.

It may appear that each iteration requires two function evaluations, but that is not so. The next guess will be used to replace either the upper or lower bound (and its candidate). The function values and the candidate for the other bound can be retained.

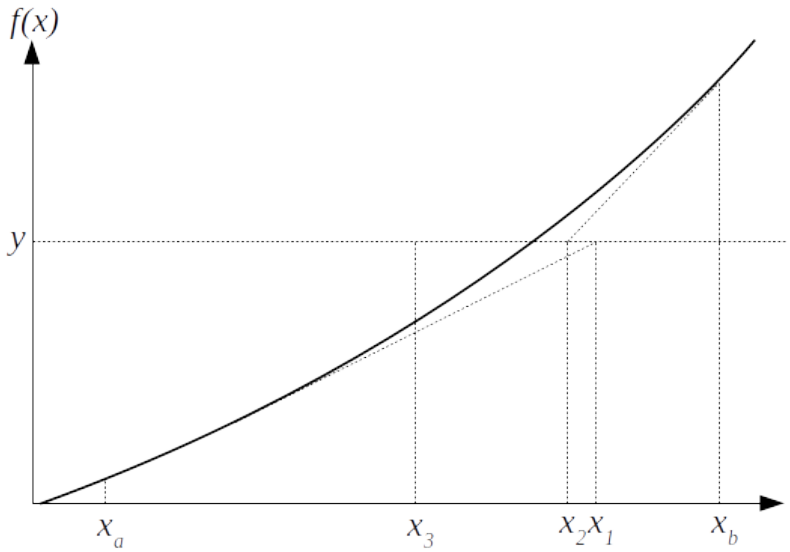

Figure 8.4: A depiction of the bisection iteration algorithm on a hypothetical function.

8.4.3 Standard candidate selection

Once the candidate guesses are calculated, the next step is to select a single candidate for the next step of iteration. In general, the Newton steps at the extremes of the domain are unlikely to be very good until the algorithm is very close to the solution. If the two Newton iteration steps produce a guess very close to each other, there is a good chance that they can be trusted. Otherwise, it is probably best to rely on the bisection step. Above all, it is vital that no guess ever lie outside of the boundaries where a solution is known to exist.

In standard candidate selection, the median (middle) of the three guesses is selected. If the median lies outside of x_a and x_b, then the next guess reverts to the bisection. In this way, if the two guesses at the edge agree, the more conservative of the two (the one closer to the center of the domain) will be selected. If they disagree or if they both diverge from the domain, the algorithm reverts to bisection.

The standard candidate selection process gives excellent performance virtually everywhere in multi-phase property domains except very near the critical point. Testing revealed that standard candidate selection is vulnerable to slow convergence near the critical point due to the extreme numerical stiffness there. It should be emphasized that the algorithm still converges; just very slowly.

The problem is similar to one that happens in the secant iteration algorithm. If a solution lies in a numerically stiff regime like in Figure 8.3, the boundary at the stiff edge will produce a guess very close to itself while the other guess will produce a guess that diverges outside the boundary. Median candidate selection will always favor the candidate at the stiff edge even though it only barely moves from the one prior.

8.4.4 Paranoid candidate selection

So-called "paranoid" candidate selection utterly distrusts the Newton method if either of the guesses lie outside of the domain. The idea is that if the domain is so nonlinear that either of the guesses leaves the domain, the algorithm should revert to bisection until it looks like the function is locally linear.

In practice, paranoid hybrid iteration is slower than standard hy-

Algorithm 3 HYBRID1: Hybrid bisection and Newton iteration

procedure _HYBRID1$(f, y, x_a, x_b, \epsilon_x, \epsilon_y)$
  $y_a \leftarrow f(x_a)$ ▷ Evaluate f at the initial bounds
  $y'_a \leftarrow f'(x_a)$
  $y_b \leftarrow f(x_b)$
  $y'_b \leftarrow f'(x_b)$
  if $y_a > y_b$ **then**
    $x_a, y_a, y'_a \leftrightarrow x_b, y_b, y'_b$ ▷ Swap the bounds
  end if
  if NOT $y_a < y < y_b$ **then**
    return "ERROR: The solution is not bracketed"
  end if
  $x_1 \leftarrow x_a + (y - y_a)/y'_a$ ▷ The first candidates
  $x_2 \leftarrow x_b + (y - y_b)/y'_b$
  $x_3 \leftarrow (x_a + x_b)/2$
  while $|x_b - x_a| > \epsilon_x$ **do** ▷ Halt if the interval is small
    $x_c \leftarrow \text{median}(x_1, x_2, x_3)$ ▷ Select the median
    if $x_c \leq \min(x_a, x_b)$ OR $x_c \geq \max(x_a, x_b)$ **then** ▷ If out of bounds...
      $x_c \leftarrow x_3$ ▷ ...use bisection instead
    end if
    $y_c \leftarrow f(x_c)$ ▷ Evaluate f at the new guess
    $y'_c \leftarrow f'(x_c)$
    if $|y_c - y| < \epsilon_y$ **then** ▷ Test for Newton convergence
      return x_c
    end if
    if $y_c < y$ **then** ▷ If this is a lower bound
      $x_a \leftarrow x_c$ ▷ Replace the lower bound
      $y_a \leftarrow y_c$
      $y'_a \leftarrow y'_c$
      $x_1 \leftarrow x_a + (y - y_a)/y'_a$ ▷ A new lower candidate
      ...

Algorithm 4 HYBRID1: Hybrid bisection continued

```
        ...
        else                                   ▷ If this is an upper bound
            x_b ← x_c                          ▷ Replace the upper bound
            y_b ← y_c
            y'_b ← y'_c
            x_2 ← x_b + (y − y_b)/y'_b         ▷ A new upper candidate
        end if
        x_3 = (x_a + x_b)/2                    ▷ A new bisection candidate
    end while
    return x_3                                 ▷ Convergence by bisection
end procedure
```

brid iteration. However, in testing, it has never failed to converge on any of the multi-phase models. Under the most difficult conditions, it might take ten or more iterations to converge. In better conditions, it typically converges in five or so iterations.

Algorithm 5 HYBRID1: Paranoid hybrid iteration

procedure _HYBRID1$(f, y, x_a, x_b, \epsilon_x, \epsilon_y)$
  $y_a \leftarrow f(x_a)$ ▷ Evaluate f at the initial bounds
  $y'_a \leftarrow f'(x_a)$
  $y_b \leftarrow f(x_b)$
  $y'_b \leftarrow f'(x_b)$
  if $y_a > y_b$ **then**
    $x_a, y_a, y'_a \leftrightarrow x_b, y_b, y'_b$ ▷ Swap the bounds
  end if
  if NOT $y_a < y < y_b$ **then**
    return "ERROR: The solution is not bracketed"
  end if
  $x_1 \leftarrow x_a + (y - y_a)/y'_a$ ▷ The first candidates
  $x_2 \leftarrow x_b + (y - y_b)/y'_b$
  $x_3 \leftarrow (x_a + x_b)/2$
  while $|x_b - x_a| > \epsilon_x$ **do** ▷ Halt if the interval is small
    ▷ If either guess is out of bounds...
    if $x_1 \notin (x_a, x_b)$ OR $x_2 \notin (x_a, x_b)$ **then**
      $x_c \leftarrow x_3$ ▷ ...use bisection instead
    else
      $x_c \leftarrow \text{median}(x_1, x_2, x_3)$ ▷ Select the median
    end if
    $y_c \leftarrow f(x_c)$ ▷ Evaluate f at the new guess
    $y'_c \leftarrow f'(x_c)$
    if $|y_c - y| < \epsilon_y$ **then** ▷ Test for Newton convergence
      return x_c
    end if
    if $y_c < y$ **then** ▷ If this is a lower bound
      $x_a \leftarrow x_c$ ▷ Replace the lower bound
      $y_a \leftarrow y_c$
      $y'_a \leftarrow y'_c$
      $x_1 \leftarrow x_a + (y - y_a)/y'_a$ ▷ A new lower candidate
      ...

Algorithm 6 HYBRID1: Paranoid hybrid iteration continued

```
        ...
        else                                  ▷ If this is an upper bound
            x_b ← x_c                         ▷ Replace the upper bound
            y_b ← y_c
            y'_b ← y'_c
            x_2 ← x_b + (y − y_b)/y'_b        ▷ A new upper candidate
        end if
        x_3 = (x_a + x_b)/2                   ▷ A new bisection candidate
    end while
    return x_3                                ▷ Convergence by bisection
end procedure
```

Bibliography

[1] M. W. Chase, *NIST-JANAF thermochemical tables fourth edition; part I, Al-Co*. Gaithersburg, Maryland 20899-0001: American Chemical Society, American Institute of Physics, and National Institute of Standards and Technology, 1998.

[2] L. V. Judson, "Weights and measures standards of the united states, a brief history," Tech. Rep. SP447, National Bureau of Standards, Washington DC, 20402, Mar 1976.

[3] M. B. D. L. Grye, M. W. Foerster, M. R. Siegel, M. V. V. Lang, M. W. Marek, and M. L. B. D. Zagon, "Resolution2: Declaration on the unit of mass and on the definition of weight; conventional value of g_n," in *Proceedings of the 3rd CGPM*, (Paris), 1901.

[4] M. DeBroglie, R. Vieweg, G. Zickner, U. Stille, E. Padelt, and E. Blechschmidt, "Resolution4: Definition of the standard atmosphere," in *Proceedings of the 10th CGPM*, (Paris), 1954.

[5] B. Inglis, J. Ulrich, and M. Milton, *The International System of Units*. Creative Commons: BIPM, 9th ed., 2019.

[6] A. Thompson and B. N. Taylor, "Guide for the use of the international system of units," Tech. Rep. SP811, NIST, Gaithersburg, MD 20899, Mar 2008.

[7] A. V. Astin, H. A. Karo, and F. H. Mueller, "Refinement of values for the yard and the pound," in *Federal Register*, (Washington DC), National Bureau of Standards, Jul 1959.

[8] K. Butcher, L. Crown, and E. J. Gentry, "The international system of units (si) - conversion factors for general use," Tech. Rep. SP1038, Weights and Measures Division, NIST, May 2006.

[9] J. R. Rumble, *CRC Handbook of Chemistry and Physics*. Boca Raton, FL: Taylor and Francis, 98th ed., 2018.

[10] National Institute for Standards and Technology, https://webbook.nist.gov/. accessed: Jun, 2021.

[11] B. J. McBride, S. Gordon, and M. A. Reno, "Coefficients for calculating thermodynamic and transport properties of individual species," Tech. Rep. 4513, NASA, 1993.