Cantera C++ User's Guide

David G. Goodwin

October 3, 2002

California Institute of Technology
Email: dgoodwin@caltech.edu

CONTENTS

Part I

User's Guide

‘

CHAPTER

ONE

Introduction

1.1 What is Cantera?

Cantera is a collection of object-oriented software tools for problems involving chemical kinetics, thermodynamics, and transport processes. It consists of a "kernel" that provides the core numerical capabilities, and language interfaces that allow accessing the kernel from various environments, including MATLAB, Python, Fortran 90, and C++.

Cantera works with *objects* that represent components of a simulation — gas mixtures, reactors, kinetics models, surfaces, equations of state, flames, ODE integrators, reaction path diagrams, and so on. Numerical models are constructed in a physical, intuitive way, by creating and assembling components.

Cantera provides the advanced capabilities needed for use in research, including fast, efficient algorithms, the capability to work with large reaction mechanisms, and interfaces for Fortran and C++. But Cantera is also designed to be easy to learn and use, and can even be used interactively from Python or MATLAB. It is also open-source software, which means that you can always find out exactly what Cantera is doing internally, should you need to know. It also means that you can customize it as necessary for your application.

Cantera's efficient kinetics algorithms can result in substantial performance gains for codes that spend much of their time evaluating chemical production rates. For typical real-world applications (for example, rich flame simulations) increases in speed of a factor of two or more have been achieved simply by using Cantera to evaluate the production rates, leaving the rest of the code (in Fortran) untouched.

Cantera is also designed to be used in teaching. The Python and MATLAB interfaces, in particular, allow students to quickly solve problems involving chemical equilibrium, thermochemistry, kinetics in well-stirred reactors, one-dimensional flames, and much more. Problems that previously involved too much computation for effective use in class now can be easily solved, freeing students to focus on the concepts being taught, instead of carrying out long calculations by hand. In addition, more interesting problems involving design or optimization are feasible, and (unlike point-and-click, what-you-see-is-all-you-get programs, the work is documented and saved in a script.

1.2 The Langage Interfaces

Cantera currently supports four programming languages: Python, MATLAB, Fortran 90, and C++. Python is a popular object-oriented scripting language, and MATLAB is a widely-used problem-solving environment that has its own scripting language. Both have good support for graphics and array operations, are relatively easy to learn and use, and can be used interactively. Python is available free for most major platforms from `http://www.python.org`. MATLAB is a product of the Mathworks, Inc. Many universities have MATLAB site licenses, and an inexpensive student version is also available.

Fortran 90 and C++ are, of course, widely-used compiled programming languages. Fortran has traditionally been the dominant language used for scientific computing, especially the older Fortran 77 version of the language. Fortran 90 adds many modern features found in other programming languages to Fortran, including partial support for object-

oriented programming. The use of C++ in scientific computing is growing rapidly, due to the development of efficient C++ compilers that have largely eliminated the performance penalty of C++ compared to Fortran.

The language interfaces are designed so that Cantera has a similar look and feel in all environments. As an example, the statements required to create an object representing an ideal gas mixture, set its state, and print out its molar enthalpy are shown below for each language.

Fortran 90

```
use Cantera
gasmix_t gas
gas = IdealGasMix('chem.xml')
call setState_TPX(gas, 300.0, OneAtm, 'CH4:1, O2:2')
write(*,*) 'molar enthalpy = ',enthalpy_mole(gas)
```

Python

```
from Cantera import *
gas = IdealGasMix("chem.xml")
gas.setState_TPX(300.0, OneAtm, "CH4:1, O2:2")
print "molar enthalpy = ", gas.enthalpy_mole()
```

MATLAB

```
gas = IdealGasMix('chem.xml');
setState_TPX(gas, 300.0, OneAtm, 'CH4:1, O2:2');
disp('molar enthalpy = ', enthalpy_mole(gas));
```

C++

```
include "Cantera.h"
IdealGasMix gas("chem.xml");
gas.setState_TPX(300.0, OneAtm, "CH4:1, O2:2");
cout << "molar enthalpy = ", gas.enthalpy_mole();
```

The structure of Cantera is shown schematically in Fig. 1.1. If Cantera is used from any language other than C++, access to the kernel goes through a C-callable library that provides functions to create, link, manipulate, and destroy kernel objects. These functions are "wrapped" by language-specific interface code that uses the capabilities of each language to represent Cantera objects.

1.3 The Kernel

Cantera is built on a "kernel" written in C++. The kernel consists of classes and functions that provide the core capabilities that can be accessed from all language environments. It is implemented as a static library.

The kernel has a modular structure, and can be configured with only those features desired. For example, if Cantera is only going to be used to evaluate thermodynamic and transport properties, then the modules implementing kinetics,

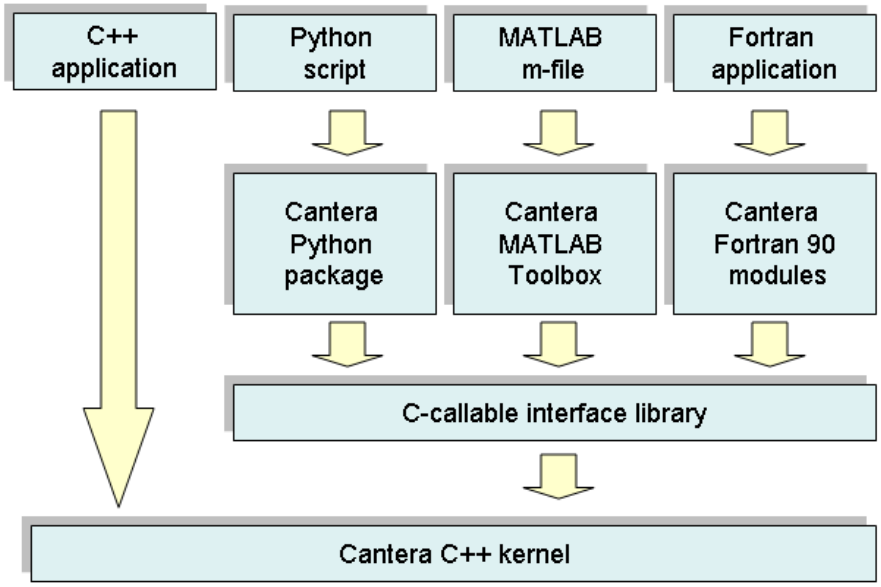


Figure 1.1: Cantera internal structure.

one-dimensional flows, ODE solvers, and other capabilities can be excluded. This cuts down both on the size of the kernel library, and on its compile time.

1.4 Versions of this Document

For each language, there is a version of this manual. The one you are reading is the C++ version. For information about using Cantera from any of the other supported languages, consult the appropriate version of the manual.

CHAPTER

TWO

Cantera in C++

Cantera is itself written in C++, and this makes using it in C++ programs different than using it to write programs in any of the other supported languages. It also makes this version of the user's manual different than the other versions too, since it includes a discussion of procedures to modify and extend Cantera, and a description of some of Cantera's internal features.

2.1 Installing Cantera

To use Cantera in C++ programs, download the Cantera source distribution and compile it on your system. The latest version of Cantera is always available from `http://www.cantera.org`.

To compile Cantera, both a C++ compiler and a Fortran 77 compiler are required, since a few external routines that Cantera uses are written in Fortran 77. Cantera has been built successfully on PCs running several flavors of the Windows operating system, and on most common unix or unix-like platforms.

2.1.1 Environment Variables

Before running applications that use Cantera, the environment variable CANTERA_ROOT should be set to the top-level Cantera directory. For example, if Cantera is located at '/usr/local/cantera-1.3', then CANTERA_ROOT should be set to `/usr/local/cantera-1.3`.

Also, the 'CANTERA_ROOT/bin' subdirectory should be put on the `PATH`, since this contains some useful utility programs.

On a Windows PC, environment variables can be set from the Control Panel, by double-clicking on the System icon, and selecting the tab labeled Advanced.

On a unix system, they may be set using shell commands. These may be put in a startup file (for example, '.cshrc' or '.profile'. A typical `csh` unix script might look like this:

```
#!/bin/csh
setenv CANTERA_ROOT /usr/local/cantera-1.3;
setenv PATH $PATH:$CANTERA_ROOT/bin
```

The equivalent commands for the GNU `bash` shell would be:

```
#!/bin/bash
export CANTERA_ROOT=/usr/local/cantera-1.3;
export PATH=$PATH:$CANTERA_ROOT/bin
```

2.1.2 Windows Build Procedure

Under Windows, Cantera can be built using Microsoft Visual C++ and Compaq Visual Fortran. Both share the same Developer Studio development environment. The Cantera distribution comes with Developer Studio workspace and project files configured to build Cantera. Simply open the workspace 'cantera.dsw' in folder 'win32', set the Active Configuration on the Build menu to `examples - Win32 Release`, and build the project. This will build the Cantera kernel static libraries and a test program `examples.exe`. Execute 'examples.exe' to verify that Cantera has been correctly installed.

2.1.3 Unix Build Procedure

On a unix system, go to CANTERA_ROOT and edit file 'configure'. Set the options appropriate for your system, save the file, and then run it by typing at the shell prompt

```
./configure
```

This will cause a set of 'Makefiles' to be written that are configured appropriately for your system.

After 'configure' has completed, type

```
make
```

to build Cantera, and then

```
make install
```

to install it. You may need to have the "superuser" for your system help you with the last step, depending on how your system's file priviliges are set.

Be sure to use a `make` program compatible with GNU `make`. This may be called something other than `make` on your system.

When `make` finishes, you should build the example programs to verify that Cantera is working properly. Type

```
cd examples/cxx
make
```

to build them.

If 'CANTERA_ROOT/bin' is on your PATH, then the examples may be run from any directory by typing

```
cxx_examples
```

Note that doing so will generate several output files — be sure to run this command from a directory where you want them to be written.

This unix build procedure also works on linux systems, and on Windows PCs running the Cygwin[1] unix-like environment. If the 'configure' script is left unmodified, it's settings are appropriate to build Cantera on a linux or cygwin system using the GNU compilers.

[1] available from `http://www.cygwin.com`

2.1.4 Compilers

Cantera can be built with recent versions of most C++ compilers. The compiler must support namespaces, the `bool` datatype, and have good support for templates, including the Standard Template Library (STL). If your C++ compiler is several years old, you may need to upgrade it first.

Different compilers come with different implementations of the STL, some of which are better than others. A good implementation that should work with most compilers is available from `http://www.sgi.com/tech/stl/download.html`.

If you are using a recent compiler, but still get errors when you try to compile, please post a message at the Cantera User's Group stating the problem; someone else may have experienced the same problem and knows the solution, or perhaps there is an incompatibility in the Cantera source code or Makefiles that can be modified to eliminate the problem. If you found a problem and were able to modify something to fix it, please post a message with your fix. We would like Cantera to compile "out of the box" on as many systems as possible, and will incorporate your fix into Cantera if practical to do so.

If you don't have C++ and Fortran 77 compilers specifically for your system, you can use the free GNU `g++` and `g77` compilers (`http://www.gnu.org/software/gcc/gcc.html`). If you have these on your system, the `configure` script will select them automatically unless you edit the configure script to specify different compilers.

> If you do use the GNU compilers, it is strongly recommended that you use version 3.0 or later (preferably 3.2 or later). Earlier versions of `g++` will compile Cantera, but may be slow and do not provide adequate support for exception handling. Throwing an exception is likely to cause your program to terminate, even if you supply code to catch the exception.

2.2 Creating a Cantera Application

Cantera comes with a utility program `ctsetup` that generates properly-configured `Makefiles` or Visual Studio project files for your application. It can also create a prototype main program file.

To begin a new application, type at the command line

```
ctsetup
```

(If '$CANTERA_ROOT/bin' is not on your path, then you will need to type the full path instead.)

When you run `ctsetup`, it will ask you for a project (i.e program) name, the directory where files should be written, and the type of output file to generate (a unix Makefile, or a Developer Studio project file).

If there is no file named '<project>.cpp' present, one will be created, with the following contents:

```
#include "Cantera.h"
// include additional header files here if needed

main(int argc, char** argv) {
    try {
        // your code goes here
    }
    catch (CanteraError) {
        showErrors(cerr);
        cerr << "program terminating." << endl;
    }
```

```
    }
```

Simply fill in the `try` block with your code.

If you generate a unix Makefile, it has the project name and the extension '.mak'. If your program is named `react`, for example, then you would build the program by typing

```
make -f react.mak
```

If your program consists of more than one source file, you will need to first edit the Makefile to add the other object file names.

If you generate a Developer Studio project file, simply open it and build the project. The project file is set up so that the links to the Cantera libraries will be made.

2.3 Exception Handling

As can be seen from the prototype program above, Cantera uses the exception-handing mechanism of C++ to report error conditions. Many different possible errors result in a `CanteraError` exception being thrown. When this occurs, an error message is written to an internal buffer, and, if thrown within a `try` block, transfers execution to the corresponding `catch` block.

In the prototype program above, this simply prints out the error message and terminates execution. If necessary, additional clean-up actions could be taken in this `catch` block.

2.4 An Example Cantera C++ Application

An example of a C++ Cantera application is shown below. This program creates an object representing an ideal gas mixture, reads an input file specifying its attributes, and prints a table listing the species names, their molecular weights, standard-state enthalpy of formation at 25 C, and elemental composition.

Listing 2.1: A program to list species data.

```
#include "Cantera.h"
#include <stdio.h>

void printSpeciesInfo(IdealGasMix& gas) {
    int nsp = gas.nSpecies();
    int nel = gas.nElements();
    int k, m;

    // print header
    printf("  %10s  %10s  %10s","Species","MolWt",
        "h0_298");
    for (m = 0; m < nel; m++)
        printf("%3s",gas.elementName(m).c_str());
    printf("\n  %10s  %10s  %10s"," ","[g/mol]",
        "[kJ/mol]");
    printf("\n\n");

    // print data for each species
    string x;
```

```
        for (k = 0; k < nsp; k++) {
            x = gas.speciesName(k)+":1";
            gas.setState_TPX(298.15, OneAtm, x);
            printf("  %10s  %10.4f  %10.4f",
                gas.speciesName(k).c_str(),
                gas.molecularWeight(k),
                1.e-6*gas.enthalpy_mole());
            for (m = 0; m < nel; m++)
                printf("%3d",int(gas.nAtoms(k,m)));
            printf("\n");
        }
    }

    int main(int argc, char** argv) {
      try {
          if (argc < 2)
              cout << "usage: speciesinfo <filename>" << endl;
          else {
              string fname = string(argv[1]);
              IdealGasMix g(fname);
              printSpeciesInfo(g);
          }
          return 0;
      }
      catch (CanteraError) {
          showErrors(cerr);
          cerr << "program terminating." << endl;
          return -1;
      }
    }
```

This program is invoked from the command line with a filename, as follows:

```
speciesinfo gri30.xml
```

The resulting output looks like this:

```
Species       MolWt      h0_298  O  H  C  N AR
             [g/mol]    [kJ/mol]

     H2      2.0159      0.0000  0  2  0  0  0
      H      1.0080    217.9851  0  1  0  0  0
      O     15.9994    249.1598  1  0  0  0  0
     O2     31.9988      0.0000  2  0  0  0  0
     OH     17.0074     39.3447  1  1  0  0  0
    H2O     18.0153   -241.8112  1  2  0  0  0
    HO2     33.0068     12.5512  2  1  0  0  0
   H2O2     34.0147   -135.8717  2  2  0  0  0
      C     12.0112    716.6360  0  0  1  0  0
     CH     13.0191    597.3014  0  1  1  0  0
    CH2     14.0271    392.3113  0  2  1  0  0
 CH2(S)     14.0271    429.8661  0  2  1  0  0
```

```
   CH3     15.0351    146.8910  0  3  1  0  0
   CH4     16.0430    -74.5954  0  4  1  0  0
    CO     28.0106   -110.5232  1  0  1  0  0
   CO2     44.0100   -393.4859  2  0  1  0  0
   HCO     29.0185     41.9974  1  1  1  0  0
  CH2O     30.0265   -108.5733  1  2  1  0  0
 CH2OH     31.0345    -14.6270  1  3  1  0  0
  CH3O     31.0345     16.3028  1  3  1  0  0
 CH3OH     32.0424   -200.9277  1  4  1  0  0
   C2H     25.0303    566.1707  0  1  2  0  0
  C2H2     26.0382    228.1860  0  2  2  0  0
  C2H3     27.0462    299.7218  0  3  2  0  0
  C2H4     28.0542     52.4968  0  4  2  0  0
  C2H5     29.0622    118.6510  0  5  2  0  0
  C2H6     30.0701    -83.8464  0  6  2  0  0
  HCCO     41.0297    177.4009  1  1  2  0  0
 CH2CO     42.0376    -47.6971  1  2  2  0  0
 HCCOH     42.0376     78.2540  1  2  2  0  0
     N     14.0067    472.6510  0  0  0  1  0
    NH     15.0147    356.8964  0  1  0  1  0
   NH2     16.0226    192.0384  0  2  0  1  0
   NH3     17.0306    -45.8957  0  3  0  1  0
   NNH     29.0214    249.5017  0  1  0  2  0
    NO     30.0061     91.2594  1  0  0  1  0
   NO2     46.0055     34.1911  2  0  0  1  0
   N2O     44.0128     81.5950  1  0  0  2  0
   HNO     31.0141    106.2523  1  1  0  1  0
    CN     26.0179    438.6564  0  0  1  1  0
   HCN     27.0258    130.8079  0  1  1  1  0
  H2CN     28.0338    247.3193  0  2  1  1  0
  HCNN     41.0325    462.0938  0  1  1  2  0
  HCNO     43.0252    171.0257  1  1  1  1  0
  HOCN     43.0252    -11.8014  1  1  1  1  0
  HNCO     43.0252   -118.0712  1  1  1  1  0
   NCO     42.0173    131.7890  1  0  1  1  0
    N2     28.0134      0.0014  0  0  0  2  0
    AR     39.9480     -0.0000  0  0  0  0  1
  C3H7     43.0892    100.4942  0  7  3  0  0
  C3H8     44.0972   -103.8476  0  8  3  0  0
CH2CHO     43.0456     25.1008  1  3  2  0  0
CH3CHO     44.0536   -166.1798  1  4  2  0  0
```

We'll look at many more examples in subsequent chapters.

2.5 Summary

This chapter has described the basics of how to download and install Cantera, and how to build a C++ application. With this as background, we are now ready to begin looking at how to use Cantera to solve problems.

CHAPTER

THREE

Phases of Matter

Some of the most useful classes Cantera provides are those that model phases of matter in various forms — gases, liquids, and solids. In this chapter, we'll introduce these classes and work through several examples showing how to use them.

3.1 Ideal Gas Mixtures

Let's begin by looking at how ideal gas mixtures are represented.

The Cantera class representing ideal gas mixtures is class `IdealGasMix`. A simple program that uses this class is shown below.

Listing 3.1: A simple program using class `IdealGasMix`.

```
#include "Cantera.h"

int main(int argc, char** argv) {
  try {
      IdealGasMix g("silane.xml");
      g.setState_TPX(2000.0, 100.0, "SIH4:0.01,H2:0.99");
      equilibrate(g, TP);
      printSummary(g, cout);
      return 0;
  }
  catch (CanteraError) {
      showErrors(cerr);
      cerr << "program terminating." << endl;
      return -1;
  }
}
```

This program creates an ideal gas mixture from a specification in an input file, sets its initial state, finds the chemical equilibrium state at the same temperature and pressure, and then prints out a summary of the results. This program requires adding only four statements to the boilerplate code generated by `ctsetup`.

The output looks like this:

```
       temperature            2000  K
          pressure             100  Pa
           density     1.33182e-05  kg/m^3
  mean mol. weight         2.21455  amu
```

```
                        1 kg            1 kmol
                     -----------      ------------
        enthalpy     3.03206e+07        6.715e+07     J
  internal energy    2.28121e+07        5.052e+07     J
         entropy          111327        2.465e+05     J/K
   Gibbs function   -1.92333e+08       -4.259e+08     J
heat capacity c_p         15109.6        3.346e+04     J/K
heat capacity c_v         11355.4        2.515e+04     J/K

                           X                Y
                     -------------     ------------
              H2   9.402287e-01     8.559033e-01
               H   5.023175e-02     2.286333e-02
            SIH4   1.904176e-10     2.761643e-09
              SI   9.493714e-03     1.204037e-01
             SIH   2.610042e-05     3.428973e-04
            SIH2   2.437203e-06     3.312835e-05
            SIH3   5.801418e-09     8.149792e-08
         H3SISIH   3.881514e-15     1.055211e-13
           SI2H6   9.861394e-20     2.770644e-18
        H2SISIH2   4.058658e-14     1.103369e-12
             SI2   1.619905e-05     4.108876e-04
             SI3   1.121605e-06     4.267413e-05
```

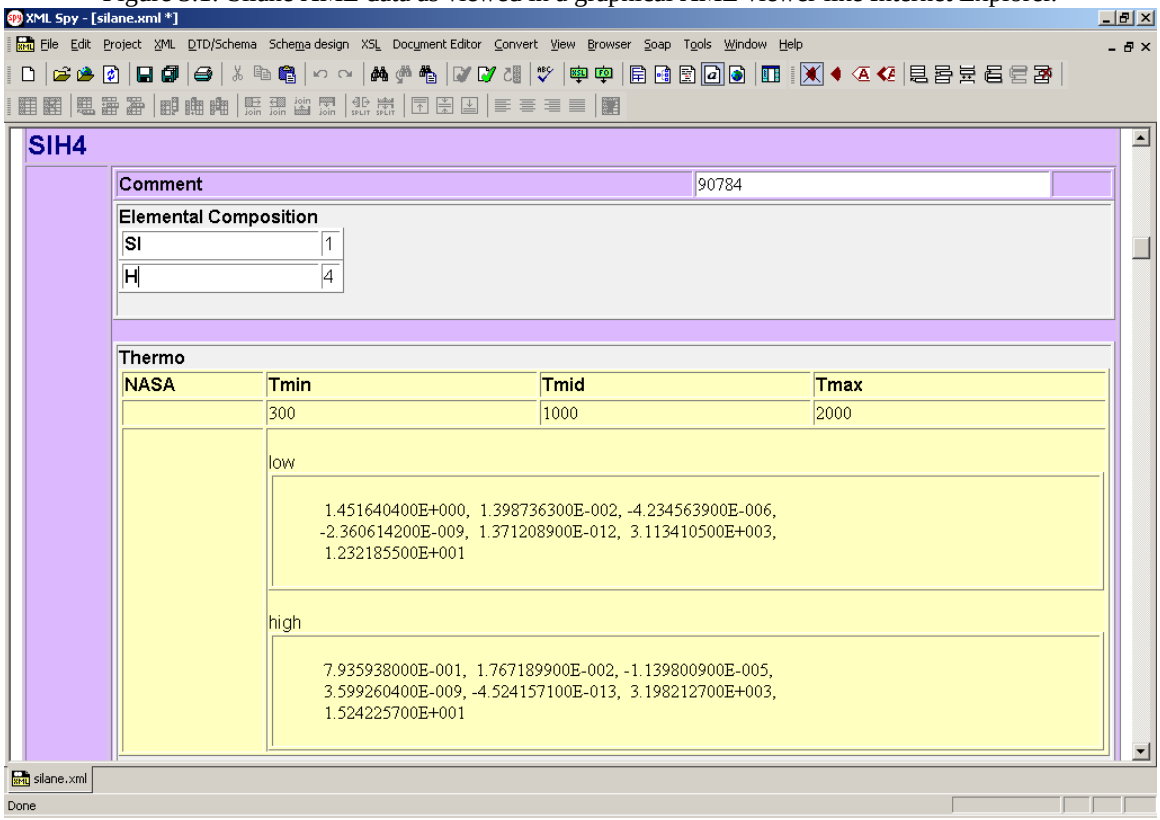
This output is all produced by the call to `printSummary`. This is a very useful way to get a quick overview of the state of the gas mixture.

3.2 Input Files

It is clear from the output above that the ideal gas mixture this object represents contains multiple species. We did not explicitly specify these species and their properties in program statements (although we could have). Instead, they were read in from file 'silane.xml'. This file contains an entry for each species similar to the one shown below. (It also contains entries for every reaction.)

```
<!-- SIH4 -->
<species id="this_s_SIH4" name="SIH4" phaseType="G">
  <string title="comment">90784</string>
  <atomArray>
    <integer title="SI">1</integer>
    <integer title="H">4</integer>
  </atomArray>
  <thermo>
    <NASA Tmax="2000" Tmid="1000" Tmin="300">
      <floatArray size="7" title="low">
       1.451640400E+000,  1.398736300E-002, -4.234563900E-006,
      -2.360614200E-009,  1.371208900E-012,  3.113410500E+003,
       1.232185500E+001
      </floatArray>
      <floatArray size="7" title="high">
       7.935938000E-001,  1.767189900E-002, -1.139800900E-005,
```

Figure 3.1: Silane XML data as viewed in a graphical XML viewer like Internet Explorer.




```
        3.599260400E-009, -4.524157100E-013,  3.198212700E+003,
        1.524225700E+001
      </floatArray>
    </NASA>
  </thermo>
</species>
```

The input file is written in an XML-based format we call CTML, for “Cantera Markup Language.” It is a plain text file, and may be edited with any text editor. But CTML files are meant to be viewed and edited with graphical tools. If the file containing this silane entry is viewed in Internet Explorer or any other XML viewer, the silane record appears as shown in Fig. 3.2. And if this file is opened in an XML editor like XML Spy, species records can be added, deleted, or modified graphically. This is a big advantage of using XML data files: graphical XML viewers and editors already exist, so there is no need to write special graphical tools for this purpose, as would be necessary if some non-standard file format were used.

> The graphical view may not work in the current release.

In addition to CTML, input files in the format used in the Chemkin software package [Kee et al., 1989] may be used. We will call this format “CK format.” Files in this format often have extensions ‘.inp’ or ‘.dat’. In any case, Cantera looks at the file contents, not the extension, to determine the file type, so any extension may be used.

The CK format specification allows species thermodynamic data to be taken from a separate database file. This file may be specified as a second parameter, if it is required. If a database file is specified, then it must exist, whether or not it is actually needed.

All of the following are valid ways to call the `IdealGasMix` constructor:

```
 // CTML file with one phase specification
IdealGasMix g1("silane.xml");

 // CTML file with multiple phases
IdealGasMix g1("silane.xml/vapor");

// self-contained CK file
IdealGasMix g1("silane.inp");

// CK file with an external database
IdealGasMix g1("silane.inp", "therm.dat");
```

A utility program `ck2ctml` is included in the distribution that converts files in this format into CTML. For example

```
ck2ctml -i chem.inp -t therm.dat -o mech.xml
```

converts Chemkin input file 'chem.inp' into CTML file 'mech.xml', using file 'therm.dat' to resolve undefined species in the input file.

Specifying a CK-format input file in the call to `IdealGasMix` automatically calls `ck2ctml` also to generate an equivalent CTML file. For example, if the input file is specified as 'chem.inp', then the call to `IdealGasMix` will create a file 'chem.xml'. This file can optionally be used later instead of 'chem.inp'. One advantages of doing so is that it allows viewing/editing the file graphically. Also, CTML files are self-contained, with all thermodynamic data for each species.

`ck2ctml.exe` for Windows PCs may be downloaded from
`http://blue.caltech.edu/cantera_dist/tools/ck2ctml.exe`

3.2.1 The Search Path

Whenever a file name is specified, Cantera first searches the local directory for a file with the specified name. If one is not found, then Cantera data directories are searched, including directory 'CANTERA_ROOT/data/inputs', where the input files used in the examples are located.

If you want to use your own input files in application programs in different directories, you can put the files into 'CANTERA_ROOT/data/inputs', instead of copying the input file to every directory where you want to use it.

Alternatively, additional directories can be added to the Cantera search path. This is done by calling function `addDirectory`:

```
addDirectory("/usr/local/inputs");
```

The format of the directory string depends on the operating system. On a PC running Windows, the call to `addDirectory` might look like this:

```
addDirectory("c:\\inputs");
```

Note that two backslashes are required to produce a literal '
' in the string.

3.3 Methods

Class `IdealGasMix` has a large number of methods, most of which are inherited from base classes. We'll look at some of the most useful methods here; others are described in the Reference section.

3.3.1 Element, Species, and Reaction Information

A few useful methods to retrieve information about the elements, species, and reactions for the mixture are shown below.

```
int nel = gas.nElements();     // number of elements
int nsp = gas.nSpecies();      // number of species
int nrx = gas.nReactions();    // number of reactions
```

The elements and species have an *index number*, which is assigned in sequence as they are added to the mixture object. Therefore, the index number corresponds to the order in which they are declared in the input file.

Since indexing of arrays in C/C++ begins with zero, the first element listed in the input file has index number 0, and the last one has index number `nElements() - 1`. Similarly, the first species has index number 0, and the last one has index number `nSpecies() - 1`.

Note that this convention differs in the other Cantera language interfaces, since Cantera uses the default indexing scheme for the local environment. Thus, in Python indexing also begins with 0, but in MATLAB and Fortran it begins with 1.

```
int m, k, i, j;
string s;
for (m = 0; m < nel; m++) {
    s = gas.elementName(m);       // index number -> name
    j = gas.elementIndex(s);      // name -> index number
    a = gas.atomicWeight(m);      // atomic wt.
}
for (k = 0; k < nsp; k++) {
    s = gas.speciesName(k);       // index number -> name
    j = gas.speciesIndex(s);      // name -> index number
    a = gas.molecularWeight(m);   // mol. wt.
}
```

While elements and species may be referenced by name or by index number, reactions can only be referenced by number. As with elements and species, numbers are assigned in order as reactions are added to the mixture.

```
bool rev;
for (i = 0; i < nrx; i++) {
    s = gas.reactionString(i);    // rxn equation
    rev = gas.isReversible(i);    // reversible rxn?
    j = gas.rxnType(i);           // reaction type flag
}
```

3.3.2 Setting the State

The methods discussed in the last section are examples of methods that return the values of constant, "read-only" mixture attributes like the number of species, their names, the coefficients that are used to compute their properties, etc. Once set during the construction phase (that is, while building the object from the input file specification), these

attributes are frozen and cannot be changed.

Other methods return the values of physical properties that may change during the course of a simulation. Examples include methods that return thermodynamic properties, transport properties, and kinetic rates. These all depend on the *thermodynamic state* of the gas mixture, which is fully specified by the values of any two independent thermodynamic properties of the mixture plus the chemical composition. If we consider only the intensive state (the state of a unit amount), then the composition is specified by $K - 1$ mole or mass fraction values, since the last one can be obtained from the summation conditions

$$\sum_k X_k = 1, \tag{3.1}$$

and

$$\sum_k Y_k = 1. \tag{3.2}$$

We will use the notation $\sum_k$ as shorthand for $\sum_{k=1}^{K}$, where K is the total number of species in the mixture.

If there are K species in the mixture, then the intensive thermodynamic state is a function of $K + 1$ parameters — two thermodynamic properties, and $K - 1$ mole fractions, mass fractions, or concentrations.

Cantera allows the state to be set by specifying the value of one quantity from each of the three columns in the table below. The methods to set the state have names that begin with `setState_`, which is followed by three uppercase letters. The letters denote the properties that are being specified, and corrrepond to the letters in parentheses in the table. The three letters must be in the same order as the columns of this table — the first letter must be T, H, or S; the second one P or R, and the third one X or Y. For example, to set the temperature, density, and mole fractions, you would call method `setState_TRX`; to set the specific entropy, pressure, and mass fractions, you would call method `setState_SPY`.

Temperature (T)	Density (R)	Mole Fractions (X)
Specific Enthalpy (H)	Pressure	Mass Fractions (Y)
Specific Entropy (S)		

These methods take three arguments, which must be the chosen property values in the same order. The first two arguments are given as floating-point numbers, and the third argument specifying the chemical composition must be either an array, or a string.

3.3.3 Specifying the Composition

Using an Array

If the composition is specified with an array, it must contain values (mole fractions or mass fractions) for each species, ordered by species index number. Although only $K - 1$ values are actually required due to the summation condition, these methods require that all K values be input. This is done for convenience, so that *relative* values may be specified that do not satisfy the summation conditions. The input values will be scaled by dividing by their sum, so that the scaled values are guaranteed to sum to one.

Therefore, if you want to set the species mole fractions all to the same value, it is sufficient to set them all to 1.0 (or any other non-zero value):

```
vector_fp x(gas.nSpecies(), 1.0);
gas.setState_TPX(300.0, OneAtm, x.begin());
```

Sometimes it is desired to set the mole or mass fractions to values that do not sum to 1.0. Often this occurs when computing small perturbations to a system of equationa about a point where the summation condition is satisfied. Two methods are provided that set the state without scaling the input values: `setState_TPX_NoNorm` and `setState_TPY_NoNorm`.

3.3.4 Using a String

In cases where only a few species have non-zero mole or mass fractions, it is sometimes more convenient to specify the composition with a string. Versions of the `setState_abc` methods are also provided that take a string rather than an array for the third argument. The format of the string is a list of comma-separated name:value pairs, for example

```
"SI2H6:2, SIH4:1.2, O2:0.1"
```

The colon separating the name and the value is required, as is the comma separating the species. The format of the number can be anything that the C function `atof` can parse correctly. The species name is case-sensitive, and must match a species in the mixture exactly.

All unspecified species are set to zero, and if a species is specified that is not present in the mixture an exception is thrown.

```
nsp = gas.nSpecies();
vector_fp x(nsp, 1.0/nsp);
double t = 300.0, p = OneAtm, rho = 0.001;

gas.setState_TPX(t, p, x);          // set T, P, and X
gas.setState_TRX(t, rho, x);        // set T, density, and X

string xstring = "CH4:0.2, N2: 0.5, O2:0.3";
gas.setState_TPX(t, p, xstring);   // set T, P, and X

vector_fp y(nsp, 1.0/nsp);
gas.setState_TPY(t, p, y);         // set T, P, and Y
gas.setState_TRY(t, rho, y);       // set T, density, and Y

string ystring = "CH4:0.2, N2: 0.5, O2:0.3";
gas.setState_TPY(t, p, ystring);   // set T, P, and Y
```

3.4 Updating the State

The methods of the previous section set the complete thermodynamic state. After one of them has been invoked, the state of the object is entirely independent of its previous state. In contrast, the methods discussed here only *update* the current state, by changing the values of some but not all properties.

In implementing methods that set only part of the state, there is some ambiguity about which unspecified property is held fixed. For example, if method `setState_TX` is called to set the temperature and mass fractions, is the pressure held constant, or the density? A convention must be adopted, and this is the one Cantera uses:

1. If neither T, H, or S is specified, then T is held to its current value
2. If neither P nor R is specified, then the *density* is held to its current value.
3. If neither X nor Y is specified, then the composition is held fixed.

```
      nsp = gas.nSpecies();
      vector_fp x(nsp, 1.0);
      double t = 300.0, p = OneAtm, rho = 0.001;

      gas.setState_TP(t,p);           // set T and P, hold comp.
      gas.setState_PX(p,x);           // set P and X, hold T
      gas.setState_TR(t,rho);         // set T and density, hold comp.

      gas.setTemperature(t);          // set T, hold *density* and comp.
      gas.setDensity(rho);            // set density, hold T and comp.
      gas.setPressure(p);             // set P, hold T and comp.

      gas.setMoleFractions(x)         // set X, hold T and density
      gas.setMoleFractions_NoNorm(x); // set raw X, hold T and density

      string xstring = "CH4:0.2, N2: 0.5, O2:0.3";
      gas.setState_TPX(t, p, xstring); // set T, P, and X

      vector<double> x(nsp, 1.0/nsp);
      gas.setState_PY(p,y);           // set P and Y, hold T

      gas.setMassFractions(x)         // set Y, hold T and density
      gas.setMassFractions_NoNorm(x); // set raw Y, hold T and density
```

The methods that set two properties have the same `setState_` form that those that set three do. But the methods that set only one property deviate from this form. For example, the method to set the temperature only (holding density and composition fixed) is called `setTemperature`, not `setState_T`.

3.5 Thermodynamic Properties

The following methods return mixture thermodynamic properties. Those with names ending in `_mole` return molar properties, and those ending in `_mass` return specific properties.

```
      double v;
      v = gas.enthalpy_mole();   // molar enthalpy [J/kmol]
      v = gas.entropy_mole();
      v = gas.gibbs_mole();
      v = gas.cp_mole();
      v = gas.cv_mole();

      v = gas.enthalpy_mass();   // specific enthalpy [J/kg]
      v = gas.entropy_mass();
      v = gas.gibbs_mass();
      v = gas.cp_mass();
      v = gas.cv_mass();
```

Property values associated with individual species may also be determined. The *partial molar* properties represent the contribution of each species to the mixture properties. The *pure molar* properties are those of the pure species at the mixture temperature and pressure.

> These may not work in the current release.

```
      int nsp = gas.nSpecies();
      vector<double> output(nsp);

      gas.getChemPotentials(output);   // chemical potentials

      // partial molar properties
      gas.getPartialEnthalpies_mole(output);
      gas.getPartialEntropies_mole(output);
      gas.getPartialGibbs_mole(output);
      gas.getPartialCp_mole(output);

      // pure species properties
      gas.getPureEnthalpies_mole(output);
      gas.getPureEntropies_mole(output);
      gas.getPureGibbs_mole(output);
      gas.getPureCp_mole(output);
```

3.6 Kinetics

Objects of type `IdealGasMix` represent *reacting* ideal gas mixtures. The most useful methods for kinetics are introduced here.

```
      int nsp = gas.nSpecies();
      int nrx = gas.nReactions();
      vector_fp rxndata(nrx);
      vector_fp spdata(nsp);

      gas.getFwdRatesOfProgress(rxndata.begin());
      gas.getRevRatesOfProgress(rxndata.begin());
      gas.getNetRatesOfProgress(rxndata.begin());

      gas.getCreationRates(spdata.begin());
      gas.getDestructionRates(spdata.begin());
      gas.getNetProductionRates(spdata.begin());
```

These examples show only a fraction of the methods available in class `IdealGasMix`. These and others will be described in more detail in subsequent chapters.

3.7 Transport Properties

Currently transport properties are not integrated into class `IdealGasMix`, although they may be in subsequent release. See Chapter **??** for information on calculation of transport properties.

3.8 Structure

We saw in the last few sections that class `IdealGasMix` provides a rich set of methods to simulate the behavior of reacting ideal gas mixtures. Many of them, of course, would be useful for simulating dense gases, liquids, or even solids. It would be very inefficient to have to implement the entire set from scratch for each class representing some material phase.

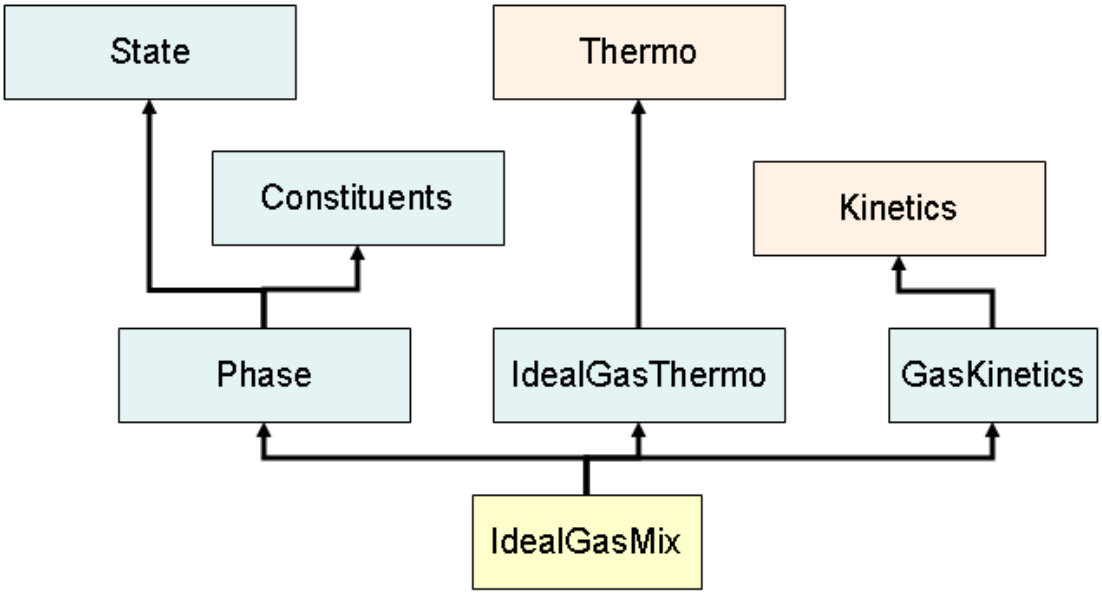


Figure 3.2: Inheritance diagram for class `IdealGasMix`.

In fact, most of the methods discussed in previous sections are not defined in class `IdealGasMix` at all. They are defined in more elementary base classes from which `IdealGasMix` derives. Class `IdealGasMix` is derived from three lower-level classes — `Phase`, `GasKinetics`, and `IdealGasThermo`. It defines relatively few methods of its own. A simplified version of its definition is shown below, and the inheritance diagram is shown in Fig. **??**.

```
class IdealGasMix : public Phase, public IdealGasThermo,
                    public GasKinetics
{
public:
      IdealGasMix();
      IdealGasMix(string infile, string dbase="",
                  bool validate=true);
      virtual ~IdealGasMix();

      // a few utility methods
      // ...
};
```

Each of the base classes of `IdealGasMix` handles a specific portion of the required tasks, as described below.

Phase. Class `Phase` provides a basic no-frills representation of any phase. It provides methods to set or get the

temperature, density, and composition (in several forms), and handles constant data for the elements and species. It itself derives from two more-elementary classes (`State` and `Constituents`).

IdealGasThermo. This class provides the thermodynamic property methods. It is a member of a family of classes that derive from class `Thermo`. Each family member has the same set of public methods, but implements them using expressions appropriate for a particular thermodynamic model. Class `IdealGasThermo` is the family member that computes properties using expressions valid for ideal gas mixtures.

GasKinetics. Class `GasKinetics` provides the methods that compute reaction rates of progress and species production rates. It uses rate expressions valid for elementary reactions in ideal gas mixtures. Like `IdealGasThermo`, `GasKinetics` is part of a family of clases that derive from a common base class (`Kinetics`).

Each of these classes is described in more detail in later chapters.

The advantage of this structure is that it is easy to create new classes representing different types of gases, liquids, or solids. For example, suppose we want to create a class representing gas mixtures that obey the van der Waals equation of state. For simplicity, we will not do kinetics simulations with this class, so only need to derive a new class for the thermodynamic properties.

All we need to do is first derive a class from `Thermo`:

```
class VdwThermo : public Thermo {
public:
      VdwThermo(phase_t& phase);
      virtual ~VdwThermo();

      // new methods
      void setVdwConstants(double a, double b);

      // overloaded methods
      virtual double pressure() { ... }
      virtual double enthalpy_mole() { ... }
      // add other virtual methods
};
```

Then simply fill in the body of the methods with the appropriate van der Waals expressions. The easiest way to do this would be to copy file `IdealGasThermo.h` and make the necessary modifications.

We have added a method to set the van der Waals a and b parameters. A better model would use a mixture rule to compute these from data for individual species. But those are design decisions that specify the model to be implemented; the procedure is the same whatever the model.

Once this class is finished, define class `VdwGasMix` as

```
class VdwGasMix : public Phase, public VdwThermo {
public:
      VdwGasMix(string infile);
      virtual ~VdwGasMix();
};
```

The same code could be used to import the gas specification from a CTML file, and then method `setVdwConstants` called to set a and b. Alternatively, the function that reads CTML files (`importCTML`) could be modified to recognize XML tags that specify van der Waals parameters in the input file.

By building up classes from more elementary ones in this way, it is possible to re-use a great deal of code, and only code the differences between the new derived class and its base class(es). The resulting class can be used wherever one of its base classes can be used. Instances of class `VdwGasMix` can, for example, be operated on using the same functions and methods used for `IdealGasMix` (except for kinetics, which we didn't implement).

CHAPTER

FOUR

Utilities

This chapter describes some classes, templates, and functions that have nothing to do with kinetics, thermodynamics, or transport processes, but are useful just the same. They are used internally within Cantera, and may be useful in user programs too.

4.1 Arrays

Many methods and functions in Cantera take array arguments of type `double*`. Although there are many different implementations in C++ of classes for arrays, the POD[1] type `double*` is used for this purpose in Cantera, since these methods must be accessible from other languages too. Fortran arrays, MATLAB arrays, and NumPy arrays in Python all can be accessed through pointers of type `double*`. If the method arguments were instead a C++ array class such as `std::vector<double>`, then the language interfacing code would have to copy the native array data into C++ objects before invoking the function or method.

Still, when writing C++ code, it is convenient to use an array class that handles memory allocation and deallocation automatically, defines `oerator«` so that arrays can be easily printed for debugging, and can dynamically grow by adding elements to the end. As long as the underlying data structure is a `double` array, and a method is provided to return a pointer to the first element in this array, the use of such a class is entirely compatible with methods that take `double*` arguments.

Until recently, Cantera used the Standard Template Library `vector<double>` class for this purpose. It was convenient to use, and method `begin()` returned a `double*` pointer to the underlying contiguous data array. However, some recent implementations of `vector<double>` define method `begin()` to return an iterator of type `vector<double>::iterator` which is *not* convertible into a `double*` pointer. (Method `begin()` returns an iterator in all implementations, but in older ones `vector<double>::iterator` is only a `typedef` for `double*`.

The solution to this problem is to move away from using `vector<double>` to represent arrays. Cantera now defines its own lightweight class that defines only those methods that are really needed (far fewer than are defined in the `std::vector` template), and guarantees that `begin()` returns a `double*` pointer to the first element. This class is accessed as `vector_fp` within Cantera, which is simply a `typedef` for class `ct::ctvector_fp`, declared in file 'ctvector.h'.

This class is used exclusively within Cantera when `double` arrays are needed, and can also be used in user programs. Some examples of its use are shown below.

```
#include "Cantera.h"
...
// create an array of length 10 and fill it with 0.0
vector_fp  x(10, 0.0);
```

[1] "plain old data"

```
    // create a zero-length array
    vector_fp y;

    // append array elements
    int i;
    for (i = 0; i < 10; i++) y.push_back(x[i]);

    // stream output
    cout << x << end << y << endl;
    cout << x.size() << " " << y.size() << endl;

    // resizing
    x.resize(500);

    // passing an array to Fortran
    ftnfunction_(x.begin());
```

CHAPTER

FIVE

Writing Functions in C++ for Use in Fortran and C

Cantera provides an interface for Fortran 90, but you can also write your own interface functions in C++ that can be called from Fortran and C. This allows you to use Cantera in Fortran 77 applications, for example.

Let's consider a simple example. Suppose you need a function that returns the species molar production rates, given the temperature, pressure, and array of mass fractions. If you were writing this in C++ with no consideration of being able to call it from any other language, you might write something like this:

```
static IdealGasMix* pGas = 0;

// create the gas object
void setup(string mechfile) {
   pGas = new IdealGasMix(mechfile);
}

// call to delete the gas object
void cleanup() {
  delete pGas;
}

void getWdot(double t, double p, double* y, double* wdot) {
  pGas->setState_TPY(t, p, y);
  pGas->getNetProductionRates(wdot);
}
```

A function `setup` is first called to create an `IdealGasMix` object. A pointer to this object is stored in a global variable for later access by function `getWdot`. A function `cleanup` is also provided to destroy this object when it is no longer needed. (Since the memory will be reclaimed anyway when the program exits, calling `cleanup` is only needed if it is desired to reclaim memory before the program finishes.)

These functions, as written, cannot be called from Fortran, or even from C. There are several problems, the first of which is *name mangling*. C++ compilers change the names of functions they compile to incorporate information about the argument types. Since Fortran and C don't know anything about C++ name mangling, they will be unable to find the functions in the object file, and the link will fail. This problem can be remedied by surrounding every function that is to be exported from the object file by `extern "C" {...}`.

Another problem is that the `setup` function takes a `string` argument, which is a C++ class, and therefore unknown to Fortran and C. For C, a solution is to rewrite the function to take a pointer to a NULL-terminated array of `char` (a "C-string"). The `IdealGasMix` constructor takes a `string` argument, but this can be constructed from the C-string

within the function.

For use in C programs, the following rewrite of the above functions will work:

```
    static IdealGasMix* pGas = 0;

    extern "C" {
      // create the gas object
      void setup(char* mechfile) {
         pGas = new IdealGasMix(string(mechfile));
      }

      // call to delete the gas object
      void cleanup() {
        delete pGas;
      }

      void getWdot(double t, double p, double* y, double* wdot) {
        pGas->setState_TPY(t, p, y);
        pGas->getNetProductionRates(wdot);
      }
    }
```

If we want to use these functions in Fortran 77, we still have some work to do, however. Unlike Fortran 90, Fortran 77 does not support the `INTERFACE` statement, so it is not possible to tell the compiler that the first two parameters to `getWdot` should be passed by value, not by reference. Nor can we specify that `setup` expects a C-string, not a Fortran string.

So we need to modify these functions to put them in the form Fortran 77 expects to see.

Fortran passes strings to functions by passing *two* arguments — a character array, and a hidden integer length parameter. Most Fortran compilers pass all string length parameters at the end of the argument list, but some, including Compaq Visual Fortran (CVF), pass them immediately after each string argument. (CVF can optionally pass them at the end, however.)

Finally, most Fortran compilers modify the function names before writing them to the object file by adding a trailing underscore, and changing the name to lowercase. Functions written in C/C++ for use in Fortran must also have lowercase names with a trailing underscore.

The following modified version is now Fortran callable.

```
    static IdealGasMix* pGas = 0;

    extern "C" {
      // create the gas object
      void setup_(char* mechfile, int n) {
         string fname(n+1,'\0');
         copy(mechfile, mechfile + n, fname.begin());
         delete pGas;
         pGas = new IdealGasMix(string(mechfile));
      }

      // call to delete the gas object
      void cleanup_() {
        delete pGas;
```

```
    pGas = 0;
  }

  void getwdot_(double *t, double *p, double *y, double *wdot) {
    pGas->setState_TPY(*t, *p, y);
    pGas->getNetProductionRates(wdot);
  }
}
```

Once compiled, the functions are equivalent to Fortran 77 subroutines with the following interface:

```
      SUBROUTINE SETUP(MECHFILE)
      CHARACTER*(*) MECHFILE
      END

      SUBROUTINE GETWDOT(T, P, Y, WDOT)
      DOUBLE PRECISION T, P, Y(1), WDOT(1)
      END

      SUBROUTINE CLEANUP
      END
```

They may be called from Fortran as follows:

```
      PROGRAM REACT
      DOUBLE PRECISION T(1000), P, Y(100), WDOT(100)
      ...
      CALL SETUP('mech.xml')
      ...
      DO I = 1,1000
          ...
          CALL GETWDOT(T(I), P, Y, WDOT)
          ...
      END DO

C     finished with GETWDOT, so release memory
      CALL CLEANUP
```

These Fortran-callable functions can of course also be used from C. To do so, a header file should be written that declares these functions as shown below.

```
#ifndef MY_FUNC_LIB_H
#definE MY_FUNC_LIB_H

void setup_(char* mechfile, int n);
void getwdot_(double* t, double* p, double* y, double* wdot);
void cleanup_();

#endif
```

Part II

Reference

CHAPTER

SIX

Class `State` — Temperature, Density, and Composition

6.1 Purpose

The thermodynamic state is uniquely specified by the values of any two thermodynamic properties and the composition. Class `State` stores the the temperature, the mass density, and the mass fractions for this purpose.

It also stores a pointer to a vector of species molecular weights, and provides methods to convert between mole fractions, mass fractions, and concentrations. Class `State` contains no other information about the species.

Class `State` is meant to be used only as a base class.

Its methods are not virtual, and are not meant to be overloaded in subclasses. They are defined in the header file, so that they may be inlined by the compiler.

Classes derived from `State` include `Phase` and `IdealGasMix`.

6.2 Theory

The thermodynamic state may be specified by any two thermodynamic state variables, but some choices are more convenient than others. Instances of class `State` specify the state by storing the temperature, mass density, and mole fractions for all species. This choice of independent variables is made since many quantities depend on T, and ρ is always independent of T, unlike P.

6.3 Construction and Initialization

`State``()`
: The constructor takes no arguments.

`initState``(`*const vector_fp& mw*`)`
: Initialize the instance. The argument is a reference to a vector of species molecular weights. A local copy of this vector is made, and the number of species is set based on the length of this vector. **Note:** This method is declared `protected`, and therefore can only be accessed from within subclasses of `State`

6.4 Temperature and Density

`double` **`setTemperature`**`()`
Set the temperature [K].

`double` **`temperature`**`() const`
The temperature [K].

`double` **`setDensity`**`()`
Set the density [kg/m^3].

`double` **`density`**`() const`
The mass density [kg/m^3].

`double` **`molarDensity`**`() const`
The molar density [kmol/m^3].

`double` **`meanMolecularWeight`**`() const`
The mean molecular weight [kg/m^3].

6.5 Setting the Composition

`void` **`setMassFractions`**`(`*double* yin*`) const`
Set the species mass fractions to the values in array *yin* after normalizing them so that $\sum_k Y_k = 1$.

`void` **`setMassFractions_NoNorm`**`(`*double* yin*`) const`
Set the species mass fractions to the values in array *yin without* normalizing them so that $\sum_k Y_k = 1$.

`void` **`setMoleFractions`**`(`*double* xin*`) const`
Set the species mole fractions to the values in array *xin* after normalizing them so that $\sum_k X_k = 1$.

`void` **`setMoleFractions_NoNorm`**`(`*double* xin*`) const`
Set the species mole fractions to the values in array *xin without* normalizing them so that $\sum_k X_k = 1$.

See also: Sectionphasesetcomp

6.6 Getting the Composition

These methods retrieve composition data, either for one species or for all of them. For those that take an array argument, the array must have length of at least K.

`void` **`getConcentrations`**(*double* c*) `const`
Return the species concentrations [kmol/m^3] in array *c*.

`void` **`getMassFractions`**(*double* y*) `const`
Return the species mass fractions in *x*.

`void` **`getMoleFractions`**(*double* x*) `const`
Return the species mole fractions in *x*.

`double` **`massFraction`**(*int k*) `const`
Mass fraction of species k.

`double` **`moleFraction`**(*int k*) `const`
Mole fraction of species k.

6.7 Utilities

`bool` **`ready`**() `const`
Return `true` if the instance has been initialized and is ready for use.

`double` **`mean_X`**(*double* Q*) `const`
Return the mole-fraction-weighted mean of the entries in array Q. The return value R is computed from

$$R = \sum_k X_k Q_k. \tag{6.1}$$

`double` **`mean_Y`**(*double* Q*) `const`
Return the mass-fraction-weighted mean of the entries in array Q. The return value R is computed from

$$R = \sum_k Y_k Q_k. \tag{6.2}$$

`double` **`sum_xlogx`**() `const`
Return $R = \sum_k X_k \log X_k$.

`double` **`sum_xlogQ`**(*double* Q*) `const`
Return $R = \sum_k X_k \log Q_k$.

CHAPTER

SEVEN

Class `Constituents` — Elements and Species

7.1 Purpose

Class `Constituents` handles basic properties of the elements and species. It is meant to be used as a mix-in class, and is not usually instantiated directly.

Class `Phase` derives from `Constituents`, and so the methods described here can be invoked on objects of class `Phase` and its subclasses.

7.2 Adding Elements and Species

`void` **`addElement`**`(`*string name, double atomicWt*`) const`

Add an element. Both the name and the atomic weight must be specified.

`void` **`addSpecies`**`(`*string name, double* atoms*[, double charge]*`) const`

Add a species with name *name*. The atoms array must contain atom numbers for each element, in the order the elements were added. The molecular weight is constructed from the atom numbers and the element atomic weights. If the species is charged, the charge may be entered as an optional third parameter.

```
    Phase s;
    s.addElement('H', 1.00797);
    s.addElement('O', 15.9996);
    double h2o_atoms[] = {2.0, 1.0};
    s.addSpecies('H2O', h2o_atoms);
    double oh_atoms[] = {1.0, 1.0};
    s.addSpecies('hydroxyl', oh_atoms);
    s.addSpecies('oh_plus', oh_atoms, 1.0);
```

7.3 Element Attributes

`double` **`atomicWeight`**`(`*int m*`) const`

Atomic weight of element m.

`const vector_fp& atomicWeights() const`
Return a read-only reference to the vector of atomic weights.

`int elementIndex(string name) const`
Index of element named *name*. The index is an integer assigned to each element in the order it was added, beginning with 0 for the first element. If *name* is not the name of an element in the set, then the value -1 is returned.

`string elementName(int m) const`
Name of element with index m. If $m < 0$ or $m >=$nElements(), an exception is thrown.

`const vector<string>& elementNames() const`
Return a read-only reference to the internal vector of element names.

`int nElements() const`
Number of elements.

```
    int nel = s.nElements();
    cout << 'Number of elements: ' << nel << endl;
    for (int m = 0; m < nel; m++) {
        printf('%d  %s  %10.4f  ', m,
               s.elementName(m), s.atomicWeight(m));
        }
    }
```

7.4 Species Attributes

`double charge(int k) const`
The electrical charge of species k, in multiples of the electron charge ($|e| = 1.602 \times 10^{-19}$ Coulombs).

`double molecularWeight(int k) const`
Molecular weight of species k [kg/kmol].

`const vector_fp& molecularWeights() const`
Return a read-only reference to the vector of molecular weights [kg/kmol].

`double nAtoms(int k, int m) const`
The number of atoms of element m in species k.

`int nSpecies() const`
Number of species.

`double` **`speciesIndex`**(*string name*) `const`

Index of species named *name*. The index is an integer assigned to each species in the order it is added, beginning with 0 for the first species. If *name* is not the name of a species in the set, then the value -1 is returned.

`string` **`speciesName`**(*int* k) `const`

Name of species with index k. If $k < 0$ or $k >=$nSpecies(), an exception is thrown.

`const vector<string>&` **`speciesNames`**() `const`

Return a read-only reference to the vector of species names.

This program takes an input file name from the command line, and prints a list of species with their molecular weights and charges, and the elemental composition matrix.

```
    #include 'Cantera.h'
    main(int argc, char** argv) {
        if (argc < 2) {
            cout << 'missing filename' << endl;
            exit(-1);
        }
        IdealGasMix g(string(argv[1]));
        int nsp = g.nSpecies();
        int nel = g.nElements();
        cout << 'Number of species: ' << nsp << endl;
        int k, m;
        for (k = 0; k < nsp; k++) {
           printf('%d  %s  %10.4f  %10.4f \n', k,
                  g.speciesName(k), g.molecularWeight(k),
                  g.charge(k));
        }

        // print elemental composition matrix
        printf('          ');
        for (m = 0; m < nel; m++) printf('%10s',g.elementName(m));
        for (k = 0; k < nsp; k++) {
            printf('%10s',g.speciesName(k));
            for (m = 0; m < nel; m++)
                printf('%10.4f',g.nAtoms(k,m));
            printf('\n');
        }
    }
```

7.5 Setting the State

Setting the thermodynamic state requires specifying two thermodynamic properties and the chemical composition.

7.5.1 Setting the Full State

These methods specify all state information. The final state is completely determined by the input values, and is independent of the initial state.

7.5.2 Updating the Current State

These methods change some properties of the current state, leaving others unchanged. The properties of the current state that are held constant if not specified are the temperature, density, and mass fractions. Note in particular that the density, not the pressure, is held constant. If it is desired to hold the pressure fixed, one of the methods of the previous section should be used instead.

`void` **`setTemperature`**(*double t*)
Set the temperature T [K], holding density and composition fixed.

`void` **`setDensity`**(*double rho*)
Set the density ρ [kg/m^3], holding temperature and composition fixed.

`void` **`setPressure`**(*double p*)
Set the pressure P [Pa], holding temperature and composition fixed.

`void` **`setState_PX`**(*double p, const double* x*)
Set the pressure [Pa] and mole fractions, holding the temperature fixed.

`void` **`setState_PY`**(*double p, const double* y*)
Set the pressure [Pa] and mass fractions, holding the temperature fixed.

`void` **`setState_TP`**(*double t, double p*)
Set the temperature [K] and pressure [Pa], holding the composition fixed.

7.6 Setting the Composition

The composition may be set by specifying species mole fractions, mass fractions, or concentrations. The methods of class `State` described here allow setting the co

This may be done with an array of values for each species, or with a string. The string format is a comma-separated list of name/value pairs, as shown below:

```
        gas1.setMoleFractions("N2:2.1, NH3:0.3, OH:0.002");
        gas2.setMassFractions("AR:0.5, HE:0.5");
```

Species not listed in the string are set to zero. The string format is patrticularly convenient for setting the initial or inlet composition, where only a few species are non-zero.

If a large number of species values must be specified, then the array format may be used. This is also the most convenient form when the composition information is taken from external flowfield data.

`void` **`setMassFractions`**(*double* yin*) `const`
Set the species mass fractions to the values in array *yin* after normalizing them so that $\sum_k Y_k = 1$.

`void` **`setMassFractions`**(*string yin*) `const`
Set the species mass fractions to the values specified in string *yin* after normalizing them so that $\sum_k Y_k = 1$.

`void` **`setMassFractions_NoNorm`**`(`*double* yin*`) const`
Set the species mass fractions to the values in array *yin without* normalizing them so that $\sum_k Y_k = 1$.

`void` **`setMoleFractions`**`(`*double* xin*`) const`
Set the species mole fractions to the values in array *xin* after normalizing them so that $\sum_k X_k = 1$.

`void` **`setMoleFractions`**`(`*string xin*`) const`
Set the species mole fractions to the values specified in string *xin* after normalizing them so that $\sum_k X_k = 1$.

`void` **`setMoleFractions_NoNorm`**`(`*double* xin*`) const`
Set the species mole fractions to the values in array *xin without* normalizing them so that $\sum_k X_k = 1$.

See also: Section 7.5

7.7 Getting the Composition

These methods retrieve composition data, either for one species or for all of them. For those that take an array argument, the array must have length of at least K.

`void` **`getConcentrations`**`(`*double* c*`) const`
Return the species concentrations [kmol/m^3] in array *c*.

`void` **`getMassFractions`**`(`*vector_fp& y*`) const`
Return the species mass fractions in *x*.

`void` **`getMoleFractions`**`(`*vector_fp& x*`) const`
Return the species mole fractions in *x*.

`double` **`massFraction`**`(`*int k*`) const`
Mass fraction of species k.

`double` **`moleFraction`**`(`*int k*`) const`
Mole fraction of species k.

CHAPTER

EIGHT

Class `Thermo` — Thermodynamic Properties

8.1 Purpose

Class `Thermo` is the base class for the family of *thermodynamic property manager* classes. Class `Thermo` only defines the interface — if any of its methods are actually called, an exception is thrown. The virtual public methods should be overloaded in derived classes to implement specific models.

These classes may be used as mix-in classes to provide thermodynamic properties. For an example of how to do this, see Chapter **??**.

8.2 Subclasses

`IdealGasThermo`, `ConstDensityThermo`.

8.3 Theory

This section is under construction.

8.3.1 Solution Properties

The thermodynamic properties must satisfy the following relationships.

$$P = -\left(\frac{\partial f}{\partial v}\right)_T \tag{8.1}$$

$$s = -\left(\frac{\partial f}{\partial T}\right)_v \tag{8.2}$$

$$u = f + Ts \tag{8.3}$$

$$h = u + Pv \tag{8.4}$$

$$g = f + Pv \tag{8.5}$$

$$c_v = T\left(\frac{\partial s}{\partial T}\right)_v \tag{8.6}$$

$$\beta = \frac{1}{v}\left(\frac{\partial v}{\partial T}\right)_P \tag{8.7}$$

$$\kappa_T = -\frac{1}{v}\left(\frac{\partial v}{\partial P}\right)_T \tag{8.8}$$

$$c_p = c_v + \frac{Tv\beta^2}{\kappa_T} \tag{8.9}$$

These relationships are written for the specific properties, but analogous expressions apply for the molar properties.

8.3.2 Partial Molar Properties

For any extensive thermodynamic property $B(T, P, N_1, \ldots, N_K)$, the partial molar property $\bar{b}_k$ for species k is defined by

$$\bar{b}_k = \left(\frac{\partial B}{\partial N_k}\right)_{T,P,N_{j\neq k}}, \tag{8.10}$$

It may be shown [1] that

$$B = \sum_k \bar{b}_k N_k \tag{8.11}$$

The chemical potential μ_k is identical to the partial molar Gibbs function.

$$\mu_k = \left(\frac{\partial G}{\partial N_k}\right)_{T,P,N_{j\neq k}}. \tag{8.12}$$

8.4 Construction and Destruction

`Thermo`(*phase_t* phase*)

A pointer is stored to object `phase`. The properties are evaluated for the temperature, density, and composition stored in this object. Note that while class `Thermo` can be instantiated if desired, its virtual methods throw exceptions if called. This constructor is meant to be called by the constructor of a subclass that overloads the virtual methods with ones that return property values. See the subclass documentation for its constructor parameters.

[1] Use the fact that B is extensive to write

$$B(T, P, \lambda N_1, \ldots, \lambda N_K) = \lambda B(T, P, N_1, \ldots, N_K).$$

Now Taylor-expand the left hand side in lambda, and equate the linear terms.

Table 8.1: Values returned by method `eosType`.

Subclass	Value
`Thermo`	0
`IdealGasThermo`	1
`IdealSolnThermo`	2

`Thermo``()`
Destructor. Does nothing.

8.5 Utilities

`virtual int` **`eosType`**`() const`
An integer flag identifying the equation of state type. Class `Thermo` returns zero. Subclasses should return a unique value. The values for currently-implemented classes are listed in Table **??**

`phase_t&` **`phase`**`()`
Return a reference to the `Phase` object.

`const phase_t&` **`phase`**`() const`
Return a read-only reference to the `Phase` object.

8.6 Partial Molar Properties

The methods that get the partial molar properties are listed below. In all cases, the output array must be supplied as an argument, and must have length greater than or equal to the number of species.

`virtual void` **`getPartialMolarVolumes`**`(`*double* vbar*`) const`
Get the array of species partial molar volumes [m^3/kmol],

$$\overline{V}_k = \left(\frac{\partial V}{\partial N_k}\right)_{T,P,N_{j\neq k}}. \tag{8.13}$$

`virtual void` **`getPartialMolarEnthalpies`**`(`*double* hbar*`) const`
Get the array of species partial molar enthalpies [J/kmol],

$$\overline{H}_k = \left(\frac{\partial H}{\partial N_k}\right)_{T,P,N_{j\neq k}}. \tag{8.14}$$

`virtual void` **`getPartialMolarEntropies`**`(`*double* sbar*`) const`
Get the array of species partial molar entropies [J/kmol/K]

$$\overline{S}_k = \left(\frac{\partial S}{\partial N_k} \right)_{T,P,N_{j \neq k}} . \tag{8.15}$$

`virtual void` **`getPartialMolarCp`**`(`*double* cpbar*`) const`
Get the array of species partial molar heat capacities at constant pressure [J/kmol/K]

$$\overline{C}_{p,k} = \left(\frac{\partial C_p}{\partial N_k} \right)_{T,P,N_{j \neq k}} . \tag{8.16}$$

`virtual void` **`getChemPotentials`**`(`*double* mu*`) const`
Get the array of chemical potentials (partial molar Gibbs functions) [J/kmol/K].

8.6.1 Limits

The parameterizations used to evaluate the properties may only be valid for a limited range of temperature, density, or pressure. These methods return those limits. These methods accept an optional species index parameter. If supplied, then the returned values represent the limits for the data for that species. Otherwise, the limits are for all species.

`double` **`maxTemp`**`([`*int k*`]) const`
The highest temperature for which the parameterization(s) are valid.

`double` **`minTemp`**`([`*int k*`]) const`
The lowest temperature for which the parameterization(s) are valid.

8.7 Molar Thermodynamic Properties

These methods return molar properties. They are all virtual methods, and must be overloaded in subclasses.

`virtual double` **`enthalpy_mole`**`() const`
Molar enthalpy $\hat{h}$ [J/kmol].

`virtual double` **`intEnergy_mole`**`() const`
Molar internal energy $\hat{u}$ [J/kmol].

`virtual double` **`entropy_mole`**`() const`
Molar entropy $\hat{s}$ [J/kmol-K].

`virtual double` **`gibbs_mole`**`() const`
Molar Gibbs function $\hat{g}$ [J/kmol].

`virtual double` **`cp_mole`**`() const`
Molar heat capacity at constant pressure $\hat{c}_p$ [J/kmol-K].

`virtual double` **`cv_mole`**`() const`
Molar heat capacity at constant volume $\hat{c}_v$ [J/kmol-K].

8.8 Specific Thermodynamic Properties

These methods return property values per unit mass. They are evaluated by dividing the corresponding molar property by the mean molecular weight. The are not declared virtual, and do not need to be overloaded in subclasses.

`double` **`enthalpy_mass`**`() const`
Specific enthalpy h [J/kg].

`double` **`intEnergy_mass`**`() const`
Specific internal energy u [J/kg].

`double` **`entropy_mass`**`() const`
Specific entropy s [J/kg-K].

`double` **`gibbs_mass`**`() const`
Specific Gibbs function g [J/kg].

`double` **`cp_mass`**`() const`
Specific heat at constant pressure c_p [J/kg-K].

`double` **`cv_mass`**`() const`
Specific heat at constant volume c_v [J/kg-K].

8.9 Setting the State

For classes that derive from both `Phase` and `Thermo`, these methods add to the set of `setState_XYZ` methods defined in `Phase`. These methods are defined here, since they involve setting the pressure, which requires the pressure equation of state. These methods are not virtual, and should not be overridden in subclasses.

`void` **`setState_TPX`**`(`*`double t, double p, const double* x`*`)`
Set the temperature [K], pressure [Pa], and mole fractions.

`void` **`setState_TPX`**`(`*`double t, double p, string x`*`)`
Set the temperature [K], pressure [Pa], and mole fractions.

`void` **`setState_TPY`**(*double t, double p, double* y*)
Set the temperature [K], pressure [Pa], and mass fractions.

`void` **`setState_TPY`**(*double t, double p, string y*)
Set the temperature [K], pressure [Pa], and mass fractions.

8.10 Setting the Enthalpy, Internal Energy, or Entropy

These methods set the specific enthalpy, specific internal energy, or specific entropy at specified pressure or specific volume. These are implemented using Newton iteration in temperature to find the state with the desired property. The optional *tol* argument specifies the error tolerance to use in the Newton iteration (default value = 10^{-8}).

`void` **`setState_HP`**(*double h, double p* [, *double tol*])
et the specific enthalpy [J/kg] and pressure [Pa].

`void` **`setState_UV`**(*double u, double v* [, *tol*])
et the specific internal energy [J/kg] and specific volume [m^3/kg].

`void` **`setState_SP`**(*double s, double p* [, *tol*])
et the specific entropy [J/kg-K] and pressure [Pa].

`void` **`setState_SV`**(*double s, double v,* [, *tol*])
et the specific entropy [J/kg-K] and specific volume [m^3/kg].

8.11 Setting to Equilibrium

Method `setToEquilState` is used by class `ChemEquil` to compute the chemical equilibrium state.

For a chemical equilibrium state (only), the species chemical potentials may be computed from a set of potentials associated with each element – the *element potentials*. For any element potential vector $[\lambda_1, \ldots, \lambda_M]$, the chemical potential vector $[\mu_1, \ldots, mu_K]$ defined by

$$\mu_k = \sum_m A_{k,m} \lambda_m, \tag{8.17}$$

along with the temperature, defines a chemical equilibrium state. This method sets the state to the equilibrium state so defined.

`void` **`setToEquilState`**(*double* lambda_RT*)
Set to the chemical equilibrium state at the current temperature and nondimensional element potentials.

CHAPTER

NINE

Class `IdealGasThermo` — Ideal Gas

9.1 Purpose

Class `IdealGasThermo` is the thermodynamic property manager class for ideal gas mixtures.

9.2 Base Classes

`Thermo` (Chapter 8).

9.3 Subclasses

None.

9.4 Theory

The pressure is computed using the ideal gas law

$$P = n\hat{R}T, \tag{9.1}$$

where $n = \rho/\overline{M}$ and $\hat{R} = 8314.0$ J/kmol/K.

For an ideal gas, the heat capacities, internal energy, and enthalpy are functions only of temperature, and are linear combinations of the pure-species values.

$$\begin{aligned}
\hat{c}_p(T) &= \sum_k \hat{c}_{p,k}(T) X_k, & (9.2)\\
\hat{h}(T) &= \sum_k \hat{h}_k(T) X_k, & (9.3)\\
\hat{u}(T) &= \hat{h}(T) - \hat{R}T, & (9.4)\\
\hat{c}_v(T) &= c_p(T) - \hat{R}. & (9.5)\\
& & (9.6)
\end{aligned}$$

The entropy and Gibbs function have a logarithmic dependence on partial pressure:

$$\hat{s}(T,P) = \sum_k s_k^0(T) - \hat{R}\sum_k X_k \log X_k - \hat{R}\log(P/P_0), \tag{9.7}$$

$$\hat{g}(T,P) = \sum_k g_k^0(T) + \hat{R}T \sum_k X_k \log X_k + \hat{R}T \log(P/P_0), \tag{9.8}$$

The pure species properties are related by

$$\hat{h}_k(T) = \hat{h}_k(T_0) + \int_{T_0}^{T} \hat{c}_p(T')dT' \tag{9.9}$$

$$\hat{s}_k^0(T) = \hat{s}_k(T_0,P_0) + \int_{T_0}^{T} \frac{\hat{c}_p(T')}{T'}dT' \tag{9.10}$$

CHAPTER

TEN

Species Thermodynamic Properties

The classes discussed in this chapter manage the standard-state properties of the pure species. These classes are designed for internal use by thermodynamic property managers (classes derived from `Thermo`).

Many models of the thermodynamic properties of solutions express solution properties in terms of properties for the pure species that depend on temperature but are evaluated at a specified reference or standard-state pressure. The temperature dependence is typically parameterized as a function of temperature for some temperature range (T_{min}, T_{max}). The parameterization might be a polynomial in T, for example.

10.1 Class `SpeciesThermo`

Class `SpeciesThermo` is the base class that defines the interface. Subclasses of `SpeciesThermo` implement specific parameterizations.

`virtual void` **`install`**`(`*int index, int type, const double* coeffs,*
double minTemp, double maxTemp, double refPressure`)`

Install a parameterization for the standard-state thermodynamic properties of the species with index number *index*. The parameterization type is indicated by the *type* flag, and must correspond to a parameterization suppported by the subclass. The coefficients of the parameterization must be provided in array *coeffs*. The parameterization is valid in the temperature range (*minTemp,maxTemp*), and is for pressure *refPressure*.

`virtual void` **`update`**`(`*double t, vector_fp& cp_R,*
vector_fp& h_RT, vector_fp& s_R`) const=0`

Compute the species non-dimensional standard-state properties at temperature *t*. On return,

$$\texttt{cp\_R[k]} = \frac{\hat{c}_{p,k}(T)}{\hat{R}}, \tag{10.1}$$

$$\texttt{h\_RT[k]} = \frac{\hat{h}_k(T)}{\hat{R}T}, \tag{10.2}$$

$$\texttt{s\_R[k]} = \frac{\hat{s}_k(T)}{\hat{R}}, \tag{10.3}$$

`double` **`maxTemp`**`(`[*int k*]`) const`

The highest temperature for which the parameterization(s) are valid.

`double` **`minTemp`**`([`*int k*`]) const`
The lowest temperature for which the parameterization(s) are valid.

`double` **`refPressure`**`() const`
The lowest temperature for which the parameterization(s) are valid.

10.2 Class `NasaThermo` — the NASA Polynomial Parameterization

A widely-used parameterization is the one first used in the NASA chemical equilibrium program, which consists of two fourth-order polynomials in T for c_p. The first one is used for $T_{min} \leq T < T_{mid}$, and the second one for $T_{mid} \leq T \leq T_{max}$.

The midpoint temperature T_{mid} must be specified along with the two sets of coefficients.

The standard-state properties are given by

$$\frac{\hat{c}_p^0(T)}{\hat{R}} = a_0 + a_1 T + a_2 T^2 + a_3 T^3 + a_4 T^4, \tag{10.4}$$

$$\frac{\hat{h}^0(T)}{\hat{R}T} = a_0 + \frac{a_1}{2} T + \frac{a_2}{3} T^2 + \frac{a_3}{4} T^3 + \frac{a_4}{5} T^4 + a_5, \tag{10.5}$$

$$\frac{\hat{s}^0(T)}{\hat{R}} = a_0 \ln T + a_1 T + \frac{a_2}{2} T^2 + \frac{a_3}{3} T^3 + \frac{a_4}{4} T^4 + a_6. \tag{10.6}$$

The coefficient array supplied to method `init` must set up as shown in the Table below.

coefficient	**value**
`coef[0]`	T_{mid}
`coef[1]-coef[7]`	$a_0, \ldots, a_6$ for the low temperature range
`coef[8]-coef[14]`	$a_0, \ldots, a_6$ for the high temperature range

```
// coefficients for O2
double o2data[15] = {1000.0,                                  // Tmid
    3.78245636e+00, -2.99673416e-03,  9.84730201e-06,   // low T a0-a6
   -9.68129509e-09,  3.24372837e-12, -1.06394356e+03,
    3.65767573e+00,
    3.28253784e+00,  1.48308754e-03, -7.57966669e-07,   // high T a0-a6
    2.09470555e-10, -2.16717794e-14, -1.08845772e+03,
    5.45323129e+00};

NasaThermo nasa;   // constructor takes no parameters

// install a parameterization for O2 as species 12
nasa.install(12, NASA, o2data, 200.0, 3500.0, OneAtm);
```

10.3 Class `ShomateThermo` — the Shomate Parameterization

This parameterization is used in the NIST Chemistry WebBook.

Seven coefficients $(a_0, \ldots, a_6)$ are used to represent $c_p^0(T)$, $h^0(T)$, and $s^0(T)$ as polynomials in T:

$$\hat{c}_p^0(T) = A + Bt + Ct^2 + Dt^3 + \frac{E}{t^2}, \tag{10.7}$$

$$\hat{h}^0(T) = At + \frac{Bt^2}{2} + \frac{Ct^3}{3}\frac{Dt^4}{4} - \frac{E}{t} + F, \tag{10.8}$$

$$s^0(T) = A\log t + Bt + \frac{Ct^2}{2} + \frac{Dt^3}{3} - \frac{E}{2t^2} + G. \tag{10.9}$$

Here $t = T/1000.0$.

The coefficient array supplied to method `init` must set up as shown in the Table below.

coefficient	**value**
`coef[0]`	T_{mid}
`coef[1]-coef[7]`	$a_0, \ldots, a_6$ for the low temperature range
`coef[8]-coef[14]`	$a_0, \ldots, a_6$ for the high temperature range

```
// coefficients for CH4
double ch4data[15] = {1300.0,
   -0.703029,  108.4773, -42.52157, 5.862788,
    0.678565, -76.84376,  158.7163,
    85.81217,  11.26467, -2.114146, 0.138190,
   -26.42221, -153.5327,  224.4140};

ShomateThermo sh;   // constructor takes no parameters

// install a parameterization for CH4 as species 5
sh.install(5, NASA, ch4data, 298.0, 6000.0, OneAtm);
```

APPENDIX

A

Glossary

closed
A system that does not exchange mass with the environment. Since mass is conserved, the total mass of a closed system is constant in time. If nuclear reactions can be neglected, the total mass of each element is also constant for a closed system. **See also:** isolated.

isolated
A system is isolated if it is closed and in addition does not exchange energy in any form with the environment. The total mass and the total energy are both constant for an isolated system. **See also:** closed.

open
A system that exchanges mass with the environment.

substance
A macroscopic sample of matter with a precise, characteristic composition. Pure water is a substance, since it is always made up of H_2O molecules; any pure element is also a substance.

compound
A substance containing more than one element. Sodium chloride, water, and silicon carbide are all compounds.

mixture
A macroscopic sample of matter made by combining two or more substances, usually (but not necessarily) finely-divided and intermixed. Liquid water with dissolved oxygen and nitrogen is a mixture, as are sand, air, wood, beer, and most other everyday materials. In fact, since even highly-purified "substances" contain measurable trace impurities, it is exceptionally rare to encounter anything that is *not* a mixture, i.e., anything that is *truly* a substance.

solution
A mixture in which the constituents are fully mixed on a molecular scale. In a solution, all molecules of a given constituent (or all sites of a given type) are statistically equivalent, and the configurational entropy of the mixture is maximal. Mixtures of gases are solutions, as are mixtures of mutually-soluble liquids or solids. For example, silicon and germanium form a crystalline solid solution Si_xGe_{1-x}, where x is continuously variable over a range of values.

phase
A macroscopic sample of matter with a homogeneous composition and structure that is stable to small perturbations. Example: water at a temperature below its critical temperature and above its triple point can exist in two stable states: a low-density state (vapor) or a high-density one(liquid). There is no stable homogeneous state at intermediate densities, and any attempt to prepare such a state will result in its spontaneous segregation into liquid and vapor regions. Liquid water and water vapor are two phases of water below its critical temperature. (Above it, these two phases merge, and there is only a single phase.)

Note that a phase does not need to be *thermodynamically* (globally) stable. Diamond is a valid phase of carbon at atmospheric pressure, even though graphite is the thermodynamically stable phase.

APPENDIX

B

Web Resources

under construction

B.1 Reaction Mechanism Files

Prof. J. E. Shepherd's Mechanism Library. Professor J. E. Shepherd at Caltech has posted a number of combustion-related reaction mechanism files at `http://www.galcit.caltech.edu/EDL/mechanisms/library/library.html`.

GRI-Mech 3.0. (description)

APPENDIX

C

Thermodynamic Property Managers

These classes are derived from class `Thermo`, and implement thermodynamic properties for particular equations of state.

C.1 Class `IdealGasThermo` — Ideal Gas

C.2 Purpose

Class `IdealGasThermo` is the thermodynamic property manager class for ideal gas mixtures.

C.3 Base Classes

`Thermo` (Chapter 8).

C.4 Subclasses

None.

C.5 Theory

The pressure is computed using the ideal gas law

$$P = n\hat{R}T, \tag{C.1}$$

where $n = \rho/\overline{M}$ and $\hat{R} = 8314.0$ J/kmol/K.

For an ideal gas, the heat capacities, internal energy, and enthalpy are functions only of temperature, and are linear combinations of the pure-species values.

$$\hat{c}_p(T) = \sum_k \hat{c}_{p,k}(T) X_k, \tag{C.2}$$

$$\hat{h}(T) = \sum_k \hat{h}_k(T) X_k, \tag{C.3}$$

$$\hat{u}(T) = \hat{h}(T) - \hat{R}T, \tag{C.4}$$

$$\hat{c}_v(T) = c_p(T) - \hat{R}. \tag{C.5}$$

$$\tag{C.6}$$

The entropy and Gibbs function have a logarithmic dependence on partial pressure:

$$\hat{s}(T,P) = \sum_k s_k^0(T) - \hat{R}\sum_k X_k \log X_k - \hat{R}\log(P/P_0), \tag{C.7}$$

$$\hat{g}(T,P) = \sum_k g_k^0(T) + \hat{R}T\sum_k X_k \log X_k + \hat{R}T\log(P/P_0), \tag{C.8}$$

The pure species properties are related by

$$\hat{h}_k(T) = \hat{h}_k(T_0) + \int_{T_0}^{T} \hat{c}_p(T')dT' \tag{C.9}$$

$$\hat{s}_k^0(T) = \hat{s}_k(T_0,P_0) + \int_{T_0}^{T} \frac{\hat{c}_p(T')}{T'}dT' \tag{C.10}$$

BIBLIOGRAPHY

R. J. Kee, F. M. Rupley, and J. A. Miller. Chemkin-II: A Fortran chemical kinetics package for the analysis of gas-phase chemical kinetics. Technical Report SAND89-8009, Sandia National Laboratories, 1989.

INDEX