Cantera User’s Guide

Fortran Version

Release 1.2

D. G. Goodwin

November, 2001

Division of Engineering and Applied Science
California Institute of Technology
Pasadena, CA
E-mail: dgoodwin@caltech.edu
# CONTENTS

## 1 Getting Started

1.1 What is Cantera? ....................................... 3
1.2 Typical Applications ...................................... 3
1.3 Installing Cantera ....................................... 5
1.4 Running Stand-Alone Applications ............................ 6
1.5 The Cantera Fortran 90 Interface ................................ 6
1.6 The cantera Module . .................................. 6
1.7 Methods ............................................ 9
1.8 Units ............................................. 10
1.9 Useful Constants ....................................... 10
1.10 An Example Program .................................. 10
1.11 Summary ............................................ 11

## 2 Mixtures

2.1 Mixture Objects ....................................... 13
2.2 Procedures ........................................... 14
2.3 Constructors ....................................... 14
2.4 Utilities ............................................. 21
2.5 Number of Elements, Species, and Reactions ............. 25
2.6 Mixture Creation ....................................... 26
2.7 Properties of the Elements .................................. 27
2.8 Properties of the Species .................................. 29
2.9 Mixture Composition ....................................... 32
2.10 Mean Properties ....................................... 36
2.11 Procedures ........................................... 38
2.12 The Thermodynamic State .................................. 38
2.13 Species Ideal Gas Properties .................................. 48
2.14 Attributes of the Parameterization ............................ 49
2.15 Mixture Thermodynamic Properties .................................. 51
2.16 Potential Energy ....................................... 56
2.17 Critical State Properties .................................. 57
2.18 Saturation Properties .................................. 58
2.19 Equations of State ....................................... 59
This document is the Fortran 90 version of the Cantera tutorial. If you are interested in using Cantera from C++ or Python, versions of this document exist for those languages as well.
1.1 What is Cantera?

Cantera is a collection of object-oriented software tools for problems involving chemical kinetics, thermodynamics, and transport processes. Among other things, it can be used to conduct kinetics simulations with large reaction mechanisms, to compute chemical equilibrium, to evaluate thermodynamic and transport properties of mixtures, to evaluate species chemical production rates, to conduct reaction path analysis, to create process simulators using networks of stirred reactors, and to model non-ideal fluids. Cantera is still in development, and more capabilities continue to be added.

1.2 Typical Applications

Cantera can be used in many different ways. Here are a few.

**Use it in your own reacting-flow codes.** Cantera can be used in Fortran or C++ reacting-flow simulation codes to evaluate properties and chemical source terms that appear in the governing equations. Cantera places no limits on the size of a reaction mechanism, or on the number of mechanisms that you can work with at one time; it can compute transport properties using a full multicomponent formulation; and uses fast, efficient numerical algorithms. It even lets you switch reaction mechanisms or transport property models dynamically during a simulation, to adaptively switch between inexpensive/approximate models and expensive/accurate ones based on local flow conditions.

It is well-suited for numerical models of laminar flames, flow reactors, chemical vapor deposition reactors, fuel cells, engines, combustors, etc. Any existing code that spends most of its time evaluating kinetic rates (a common situation when large reaction mechanisms are used) may run substantially faster if ported to Cantera. (Cantera’s kinetics algorithm, in particular, runs anywhere from two to four times faster, depending on the platform, than that used in some other widely-used packages.)

**Use it for exploratory calculations.** Sometimes you just need a quick answer to a simple question, for example:

- if air is heated to 3000 K suddenly, how much NO is produced in 1 sec?
- What is the adiabatic flame temperature of a stoichiometric acetylene/air flame?
- What are the principal reaction paths in silane pyrolysis?
With Cantera, answering any of these requires only writing a few lines of code. If you are comfortable with Fortran or C++ you can use either of these, or you can write a short Python script, which has the advantage that it can be run immediately without compilation. Python can also be used interactively. Or you can use one of the stand-alone applications that come with Cantera, which requires no programming at all, other than writing an input file.

Use it in teaching. Cantera is ideal for use in teaching courses in combustion, reaction engineering, transport processes, kinetics, or similar areas. Every student can have his or her own copy and use it from whatever language or application he or she prefers. There are no issues of cost, site licenses, license managers, etc., as there are for most commercial packages. For this reason, Cantera-based applications are also a good choice for software that accompanies textbooks in these fields.

Run simulations with your own kinetics, transport, or thermodynamics models. Cantera is designed to be customized and extended. You are not locked in to using a particular equation of state, or reaction rate expressions, or anything else. If you need a different kinetics model than one provided, you can write your own and link it in. The same goes for transport models, equations of state, etc. [Currently this requires C++ programming, but in an upcoming release this will be possible to do from Fortran.]

Make reaction path diagrams and movies. One of the best ways to obtain insight into the most important chemical pathways in a complex reaction mechanism is to make a reaction path diagram, showing the fluxes of a conserved element through the species due to chemistry. But producing these by hand is a slow, tedious process. Cantera can automatically generate reaction path diagrams at any time during a simulation. It is even possible to create reaction path diagram movies, showing how the chemical pathways change with time, as reactants are depleted and products are formed.

Create stirred reactor networks. Cantera implements a general-purpose stirred reactor model that can be linked to other ones in a network. Reactors can have any number of inlets and outlets, can have a time-dependent volume, and can be connected through devices that regulate the flow rate between them in various ways. Closed-loop controllers may be installed to regulate pressure, temperature, or other properties.

Using these basic components, you can build models of a wide variety of systems, ranging in complexity from simple constant-pressure or constant-volume reactors to complete engine simulators.

Simulate multiphase pure fluids. Cantera implements several accurate equations of state for pure fluids, allowing you to compute properties in the liquid, vapor, mixed liquid/vapor, and supercritical states. (This will soon be expanded to include multiphase mixture models.)

These capabilities are only the beginning. Some new features scheduled to be implemented in an upcoming release are non-ideal and multiphase reacting mixtures, surface and interfacial chemistry, sensitivity analysis, models for simple reacting flows ( laminar flames, boundary layers, flow in channels, etc.), and electrochemistry. Other capabilities, or interfaces to other applications, may also be added, depending on available time, support, etc. If you have specific things you want to see Cantera support, you are encouraged to become a Cantera developer, or support Cantera development in other ways to make it possible. See http://www.cantera.org/support for more information.
1.3 Installing Cantera

If you’ve read this far, maybe you’re thinking you’d like to try Cantera out. All you need to do is download a version for your platform from http://www.cantera.org and install it.

1.3.1 Windows

If you want to install Cantera on a PC running Windows, download and run ‘cantera12.msi’. (If for some reason this doesn’t work, you can download ‘cantera12.zip’ instead.)

If you only intend to run the stand-alone applications that come with Cantera, then you don’t need a compiler. The executable programs are in folder ‘bin’ within the main Cantera folder.

If you intend to use Cantera in your own Fortran applications, you need to have Compaq (formerly Digital) Visual Fortran 6.0 or later. If you do, look in the ‘win32’ folder within the main Cantera folder. There you will find project and workspace files for Cantera.

To load the Cantera Fortran workspace into Visual Studio, open file ‘Cantera_F90.dsw’.

1.3.2 unix/linux

If you use unix or linux, you will need to download the source code and build Cantera yourself, unless the web site contains a binary version for your platform. To build everything, you will need both a C++ compiler and a Fortran 90 compiler. A configuration script is used to set parameters for Makefiles. In most cases, the script correctly configures Cantera without problem.

1.3.3 The Macintosh

Cantera has not been ported to the Macintosh platform, but there is no reason it cannot be. If you successfully port it to the Macintosh, please let us know and we will add the necessary files to the standard distribution.

Note that the Cantera Fortran interface requires Fortran 90, so if you are using the free g77 compiler, you will need to upgrade.

1.3.4 Environment Variables

Before using Cantera, environment variable CANTERA_ROOT should be set to the top-level Cantera directory. If you are using Cantera on a PC running Windows, you can alternatively set WIN_CANTERA_ROOT. (This is defined so that users of unix-like environments that run under Windows (e.g. CygWin) can define CANTERA_ROOT in unix-like format (/usr/local/cantera-1.2) and WIN_CANTERA_ROOT in DOS-like format (C:\CANtera-1.2).

In addition, you can optionally set CANTERA_DATA_DIR to a directory where input files should be looked for, if not found in the local directory. By default, Cantera will look in ‘CANTERA_ROOT/data’.

1.3. Installing Cantera
1.4 Running Stand-Alone Applications

Cantera is distributed with a few simple application programs. These are located in the ‘apps’ subdirectory. For example, one such program is zero, a general-purpose zero-dimensional kinetics simulator. All of these applications include the source code, unix Makefiles, and Visual Studio project files, so they can also function as starting points to build your own applications.

1.5 The Cantera Fortran 90 Interface

Cantera consists of a kernel written in C++, and a set of language interface libraries that allow using Cantera from other programming languages. The kernel is object-oriented, and defines a set of classes that represent important entities for a reacting-flow simulation. The kernel is described in more detail elsewhere.

Much of the object-oriented character of the C++ kernel is retained even in languages that are not themselves object-oriented, like Fortran. Fortran 90 actually does implement a number of things that allow some degree of object-oriented programming; as a front end for C++ objects it works very well.

From Fortran 90, you can create objects that represent reacting mixtures, or various types of reactors, or ODE integrators, or many other useful things. A great advantage of an object-oriented framework is that objects can be combined together to build more complex dynamic system and process models. For example, it is a simple matter to quickly put together a complex network of stirred reactors, complete with sensors, actuators, and closed-loop controllers, if desired.

1.6 The cantera Module

To use Cantera from Fortran 90, put the statement

\[ \text{use cantera} \]

at the top of any program unit where you need to access a Cantera object, constant, or procedure. Module cantera contains the Cantera Fortran 90 interface specification, and handles calling the C++ kernel procedures. It in turn uses other Cantera modules. You do not have to include a use statement for these others in your program, but the module information files must be available for them.

All module files are located in directory ‘Cantera/fortran/modules’. You should either tell your Fortran compiler to look for module files there, or else copy them all to the directory where your application files are.

1.6.1 Declaring an Object

Cantera represents objects in Fortran 90 using Fortran derived types. Derived types are not part of the older Fortran 77, which is one reason why Cantera implements a Fortran 90 interface, but not one for Fortran 77.

A few of the object types Cantera defines are

- **mixture_t**
  - Reacting mixtures.
Continuously-stirred tank reactors.

Flow control devices.

Equation of state managers.

Kinetics managers.

Transport managers.

By convention, all type names end in ‘_t’. This serves as a visual reminder that these are object types, and is consistent with the C/C++ naming convention (e.g., size_t, clock_t).

Objects may be declared using a syntax very similar to that used to declare simple variables:

```fortran
  type(mixture_t) mix1, mix2, a, b, c
  type(cstr_t) r1, r2, d, e, f
```

The primary difference is that the type name is surrounded by `type(...)`. 

1.6.2 Constructing An Object

The objects that are “visible” in your program are really only a Fortran facade covering an object belonging to one of the Cantera C++ classes (which we will call a “kernel object.”) The Fortran object is typically very small — in many cases, it holds only a single integer, which is the address of the C++ object, converted to a form representable by a Fortran variable (usually an integer). We call this internally-stored variable the handle of the C++ object.

When a Fortran derived-type object is declared, no initialization is done, and there is no underlying C++ object for the handle to point to. Therefore, the first thing you need to do before using an object is call a procedure that constructs an appropriate kernel object, stores its address (converted to an integer) in the Fortran object’s handle, and performs any other initialization needed to produce a valid, functioning object.

We refer to such procedures as constructors. In Cantera, constructors are Fortran functions that return an object. The syntax to construct an object is shown below.

```fortran
  type(mixture_t) gasmix
  ...
  gasmix = GRIMech30()
```

After the object is declared, but before its first use, it is assigned the return value of a constructor function. In this example, constructor GRIMech30 is called, which returns an object representing a gas mixture that conforms to the widely-used natural gas combustion mechanism GRI-Mech version 3.0 [Smith et al., 1997].

Object construction is always done by assignment, as shown here. The assignment simple copies the handle from the object returned by GRIMech30 () to object gasmix. Once this has been done, the handle inside gasmix points to a real C++ object, and it is one that models a gas mixture containing the 53 species and 325 reactions of GRI-Mech 3.0.

1.6. The cantera Module
We can check this by printing some attributes of the mixture:

```fortran
use cantera
type(mixture_t) mix
character*10 elnames(5), spnames(53)
character*20 rxn
mix = GRIMech30()
write(*,*), nElements(mix), nSpecies(mix), nReactions(mix)
call getElementNames(mix, elnames)
write(*,*) 'elements: '
write(*,*) (elnames(m), m=1, nElements(mix))
call getSpeciesNames(mix, spnames)
write(*,*) 'species: '
write(*,*) (spnames(k), k=1, nSpecies(mix))
call getReactionString(mix, i, rxn)
write(*,*) rxn
end do
```

The resulting output is

```
5 53 325

elements:
  O  H  C  N  AR

species:
  H2  H  O  O2  OH  H2O  HO2
  H2O2 C  CH  CH2  CH2(S) CH3  CH4
  CO  CO2 HCO  CH2O  CH2OH CH3O  CH3OH
  C2H C2H2 C2H3  C2H4  C2H5  C2H6 HCCO
  CH2CO HCCOH  N  NH  NH2  NH3  NNH
  NO  NO2 HNO  CN  HCN  H2CN
  HCNN HCNO  HOCN  HNCO  NCO  N2  AR
  C3H7 C3H8  CH2CHO  CH3CHO

first 10 reactions:
  2 O + M <= > O2 + M
  O + H + M <= > OH + M
  O + H2 <= > H + OH
  O + HO2 <= > OH + O2
  O + H2O2 <= > OH + HO
  O + CH <= > H + CO
  O + CH2 <= > H + HCO
  O + CH2(S) <= > H2 +
  O + CH2(S) <= > H + H
  O + CH3 <= > H + CH2O
```

These attributes are in fact correct for GRI-Mech 3.0.

Sometimes the constructor function can fail. For example, constructor GRIMech30 will fail if you don’t have CANTERA_ROOT set properly, since it uses this to find data files it needs to build the mixture.
Cantera implements a LOGICAL function ready to test whether or not an object is ready to use. Before calling a constructor for some object object, ready(object) returns .FALSE.. After successful construction, it returns .TRUE..

Some object constructors also write log files, which may contain warning or error messages. If the construction process fails for no apparent reason, check the log file if one exists.

1.7 Methods

In object-oriented languages, every class of objects has an associated set of methods or member functions that operate specifically on objects of that class. An object that has a “temperature” attribute might have method setTemperature(t) to set the value, and temperature() to read it.

The syntax to invoke these methods on an object in C++ or Java would be

```cpp
x.setTemperature(300.0); // set temp. of x to 300 K
y.setTemperature(x.temperature()); // set y temp. to x temp.
```

Here object y might belong to an entirely different class than x, and the process of setting its temperature might involve entirely different internal operations. Both classes can define methods with the same name, since it is always clear which one should be called — the one that belongs to the class of the object it is “attached” to.

Fortran is not an object-oriented language, and has only functions and subroutines, not object-specific methods. Therefore, the Cantera Fortran 90 interface uses functions and subroutines to serve the role of methods. Thus, the temperature attribute of an object is retrieved by calling function temperature(object).

What happens if you try to access a temperature attribute as object%temperature? Actually, the compiler will complain that the object has no member temperature and will abort. Property values are not stored in the Fortran derived type objects. They are either stored in the kernel object, or computed on-the-fly, depending on which attribute you request.

Mixtures and reactors are two of the Cantera object types that have a temperature attribute. We would like to be able to write something like this:

```fortran
use cantera
type(mixture_t) mix
type(cstr_t) reactor
write(*,*), temperature(mix), temperature(reactor)
```

This presents an immediate problem (but one that Fortran 90 fortunately provides a solution for). If we just define function temperature in the usual way, we would have to specify the argument type. If the function were then called with an argument of a different type, then (depending on the compiler and compile options) the compile might abort (good), or the compiler might warn you (ok), or it might accept it (bad), resulting in code that runs but gives erroneous results.

To fix this problem, we could define separate function names for each (mixture_temperature, cstr_temperature, etc. but this would be awkward at best. (This is precisely what we would have to do in Fortran 77.)
Fortunately, Fortran 90 provides a mechanism so that we can call functions `temperature(object)`, `setTemperature(object, t)` (or any other function) for any type of object. This is done using *generic names*. Cantera really *does* define different function names for each type of object — there is a function `mix_temperature()` that returns the temperature attribute of a `mixture_t` object, an entirely different function `reac_temperature()` that does the same for `cstr_t` objects, and so on. But in module `cantera`, `temperature` is declared to be a generic name that any of these functions can be called by. When a call to `temperature` is made, the compiler looks at the argument list, and then looks to see if any function that takes these argument types has been associated with this generic name. If so, the call to `temperature` is replaced by one to the appropriate function, and if not, a compile error results.

This procedure is analogous to using generic names for intrinsic Fortran function — if you call `sin(x)` with a double precision argument, function `dsin(x)` is actually called; if `x` has type `real(4)`, then function `asin(x)` is called. Fortran 90 simply extends this capability to any function.

### 1.8 Units

Cantera uses the SI unit system. The SI units for some common quantities are listed in Table 1.8, along with conversion factors to cgs units.

<table>
<thead>
<tr>
<th>Property</th>
<th>SI unit</th>
<th>cgs unit</th>
<th>SI to cgs multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>K</td>
<td>K</td>
<td>1</td>
</tr>
<tr>
<td>Pressure</td>
<td>Pa</td>
<td>dyne/cm²</td>
<td>10</td>
</tr>
<tr>
<td>Density</td>
<td>kg/m³</td>
<td>gm/cm³</td>
<td>0.001</td>
</tr>
<tr>
<td>Quantity</td>
<td>kmol</td>
<td>mol</td>
<td>1000</td>
</tr>
<tr>
<td>Concentration</td>
<td>kmol/m³</td>
<td>mol/cm³</td>
<td>0.001</td>
</tr>
<tr>
<td>Viscosity</td>
<td>Pa·s</td>
<td>gm/cm·s</td>
<td>10</td>
</tr>
<tr>
<td>Thermal Conductivity</td>
<td>W/m·K</td>
<td>erg/cm·s·K</td>
<td>10⁵</td>
</tr>
</tbody>
</table>

### 1.9 Useful Constants

The `cantera` module defines several useful constants, including the ones listed here.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OneAtm</td>
<td>$1.01325 \times 10^5$</td>
<td>Atmospheric pressure in Pascals.</td>
</tr>
<tr>
<td>Avogadro</td>
<td>$6.022136736 \times 10^{26}$</td>
<td>Number of particles in one kmol.</td>
</tr>
<tr>
<td>GasConstant</td>
<td>8.3140</td>
<td>The universal ideal gas constant $\hat{R}$.</td>
</tr>
<tr>
<td>StefanBoltz</td>
<td>$5.6705 \times 10^{-8}$</td>
<td>The Stephan-Boltzmann constant $\sigma$ [W/m²·K⁴].</td>
</tr>
<tr>
<td>Boltzmann</td>
<td>GasConstant / Avogadro</td>
<td>Boltzmann’s constant $k$ [J/K].</td>
</tr>
</tbody>
</table>

### 1.10 An Example Program

To see how this all works, let’s look at a complete, functioning Cantera Fortran 90 program. This program is a simple chemical equilibrium calculator.
program eq
use cantera
implicit double precision (a-h,o-z)
character*50 xstring
character*2 propPair
type(mixture_t) mix

mix = GRIMech30()
if (.not.ready(mix)) then
   write(*,*) 'error encountered constructing mixture.'
   stop
end if

write(*,*) 'enter T [K] and P [atm]:'
read(*,*) temp, pres
pres = OneAtm * pres ! convert to Pascals

! the initial composition should be entered as a string of name:<moles> pairs
! for example ‘CH4:3, O2:2, N2:7.52’. The values will be normalized
! internally
   write(*,*) 'enter initial composition:'
   read(*, 10) xstring
10 format(a)

! set the initial mixture state
   call setState_TPX(temp, pres, xstring)

! determine what to hold constant
   write(*,*) 'enter one of HP, TP, UV, or SP to specify the
   write(*,*) 'properties to hold constant:'
   read(*,10) propPair

! equilibrate the mixture
   call equilibrate(mix, propPair)

! print a summary of the new state
   call printSummary(mix)

stop
end

Each of the functions or subroutines used here is described in detail in later chapters.

1.11 Summary

So in summary, the procedure to write a Cantera-based Fortran 90 program is simple:
1. Put the statement `use cantera` at the top of any program unit where you want to use Cantera.

2. Declare one or more objects.

3. Construct the objects that you declared by assigning them the return value of a constructor function.

4. Use the object in your program.

In the following chapters, the objects Cantera implements, their constructors, and their methods will be described in detail. Many examples will also be given. You should try programming a few of these yourself, to verify that you get the same results as shown here. Once you have done that, you’re ready to use Cantera for your own applications.
Some of the most useful objects Cantera provides are ones representing mixtures. Mixture objects have a modular structure, allowing many different types of mixtures to be simulated by combining different mixture components. You can build a mixture object to suit your needs by selecting an equation of state, a model for homogeneous kinetics (optional), and one for transport properties (also optional). You can then specify the elements that may be present in the mixture, add species composed of these elements, and add reactions among these species. Mixture objects also can be used to model pure substances, by adding only one species to the "mixture."

If you like, you can carry out this process of mixture construction by calling Fortran procedures for each step. But Cantera also provides a set of turn-key constructors that do all of this for you. For example, if you want an object representing pure water (including vapor, liquid, and saturated states), all you need to do is call constructor \texttt{Water()} . It takes care of installing the right equation of state and installing a single species composed of two atoms of hydrogen and one of oxygen.

Or if you want an object that allows you to do kinetics calculations in a way that is compatible with the CHEMKIN™ package, just call constructor \texttt{CKGas} with your reaction mechanism input file as the argument. \texttt{CKGas} installs the appropriate kinetics model, equation of state, and pure species property parameterizations, and then adds the elements, species, and reactions specified in your input file. Using \texttt{CKGas}, a complete object representing your reaction mechanism, ready for use in kinetics simulations, can be constructed with a single function call.

In this chapter, we will take a first look at objects representing mixtures (and pure substances). We will have much more to say about mixture objects in the next few chapters, where we will cover their thermodynamic properties, kinetic rates, and transport properties.

\section{2.1 Mixture Objects}

\subsection{2.1.1 The Fortran Mixture Object}

The basic object type used to represent matter is type \texttt{mixture_t}. As this name suggests, Cantera regards all material systems as \textit{mixtures}, although the term is used somewhat loosely, so that this type can represent pure substances too. A pure substance is simply regarded as a mixture that happens to contain only one species.

Objects representing mixtures are implemented in Fortran 90 by the derived type \texttt{mixture_t}. Like all Cantera objects, the \texttt{mixture_t} object itself is very thin. It contains only 4 integers – the handle that is
used to access the underlying C++ object created by the constructor, the number of elements, the number of species, and the number of reactions. All other information about the mixture is stored in the kernel-level C++ object.

2.1.2 The Kernel Mixture Object

As mentioned above, the kernel mixture object has a modular structure, with interchangeable parts. The basic structure is shown in Figure 2.2. A mixture object contains three major components, each of which has specific responsibilities. The thermodynamic property manager handles all requests for thermodynamic properties of any type; the transport property manager is responsible for everything to do with transport properties, and the kinetics manager deals with everything related to homogeneous kinetics. We refer to these three internal objects as the “property managers.”

The mixture object itself does not compute thermodynamic, transport, or kinetic properties; it simply directs incoming requests for property values to the appropriate manager. It also mediates communication between the property managers — if the transport manager needs a thermodynamic property, the mixture object gets it from the thermodynamic property manager and hands it to the transport manager. The managers have no direct links, making it possible to swap one out for another one of the same type without disrupting the operation of the other property managers.

The mixture object does handle a few things itself. It serves as the interface to the application program, maintains lists of species and element properties, stores and retrieves data specifying the state (internally represented by temperature, density, and mass fractions), and orchestrates property updating when the state has changed.

```
mixture_t
    Object type for mixtures.

    nel        integer . Number of elements
    nsp        integer . Number of species
    nrxn       integer . Number of reactions
    hndl       integer . Handle encoding pointer to the C++ object
```

2.2 Procedures

2.3 Constructors

2.3.1 Reacting Gas Mixtures

To carry out a kinetics simulation with Cantera, an object representing a chemically-reacting mixture is required. While such an object can be created using the basic `Mixture` constructor and explicitly adding all necessary components, it is usually more convenient to call a constructor that does all that for you and returns a finished object, ready to use in simulations.

In this section, constructors that return objects that model reacting gas mixtures are described. These constructors typically parse an input file or files, which specify the elements, species, and reactions to be included, and provide all necessary parameters needed by kinetic, thermodynamic, and transport managers.
Figure 2.1: Top-level structure of mixture objects. Each
Release Note: In this early release, the only reacting mixture model implemented is one compatible with the model described in Kee et al. [1989]. The capability to define other models is present in the kernel, however.

1) CKGas

Constructor CKGas implements a mixture model consistent with that described by Kee et al. [1989]. This model is designed to be used to represent low-density, reacting gas mixtures. The mixture components installed by this constructor are listed below.

Equation of State Manager. The IdealGasEOS equation of state manager is installed.

Species Thermodynamic Properties. The NASA polynomial parameterization is used to represent the thermodynamic properties of the individual species. See ??

Kinetics. The CK_Kinetics kinetics manager is used for homogeneous kinetics. See Chapter ??

Transport. The NullTransport transport manager is installed, disabling evaluation of transport properties until another manager is installed by calling setTransport. Other compatible transport managers include MultiTransport and MixTransport.

Homogeneous Kinetics. The CK_Kinetics model for homogeneous kinetics is used. See chapter ?? for more information on this kinetics model.

Transport Properties. Transport properties are disabled by default, since many applications do not require them. Any supported transport property model can be used, including MultiTransport and MixTransport, which are compatible with the multicomponent and mixture-averaged models, respectively, described by Kee et al. [1986], ?.

Syntax

\[
\text{object} = \text{CKGas}(\text{inputFile, thermoFile, iValidate})
\]

Arguments

inputFile (Input) Input file. The input file defines the elements, species, and reactions the mixture will contain. It is a text file, and must be in the format\(^1\) specified in Kee et al. [1989]. This file format is widely used to describe gas-phase reaction mechanisms, and several web sites have reaction mechanisms in CK format available for download. A partial list of such sites is maintained at the Cantera User’s Group site http://groups.yahoo.com/group/cantera. If omitted, an empty mixture object is returned. Cantera actually defines several extensions to the format of Kee et al. [1989]. See Appendix ?? for a complete description of CK format with Cantera extensions. [character*(*)].

thermoFile (Input) CK-format species property database. Species property data may be specified either in the THERMO block of the input file, or in thermoFile. If species data is present in both files for some species, the input file takes precedence. Note that while thermoFile may be any CK-format file, only its THERMO section will be read. If thermoFile is an empty string, then all species data must be contained in the input file. [character*(*)].

\(^1\)We refer to this format as “CK format”
iValidate  (Input) This flag determines the degree to which the reaction mechanism is validated
as the file is parsed. If omitted or zero, only basic validation that can be performed quickly is
done. If non-zero, extensive and possibly slow validation of the mechanism will be performed.
[integer].

Result
Returns an object of type mixture_t.

Example

type(mixture_t) mix1, mix2, mix3
! all species data in 'mymech.inp'
mix1 = CKGas('mymech.inp')
! look for species data in 'thrm.dat' if not in 'mech2.inp'
mix2 = CKGas('mech2.inp', 'thrm.dat')
! turn on extra mechanism validation
mix3 = CKGas('mech3.inp', '', 1)

See Also
addDirectory, GRIMech30

2) GRIMech30

This constructor builds a mixture object corresponding to the widely-used natural gas combustion mecha-
nism GRI-Mech version 3.0 [Smith et al., 1997]. The mixture object created contains 5 elements, 53 species,
and 325 reactions.

Syntax
object = GRIMech30()

Result
Returns an object of type mixture_t.

Description

The natural gas combustion mechanism GRI-Mech version 3.0 Smith et al. [1997] is widely-used for
general combustion calculations. [more...]

At present, this constructor simply calls the CKGas constructor with input file ‘grimech30.inp’. This
file is located in directory ‘data/CK/mechanisms’ within the ‘cantera-1.2’ tree. For this to work
correctly, environment variable CANTERA_ROOT must be set as described in Chapter 1.

In a future release, calling this constructor will generate a hard-coded version of GRI-Mech
3.0, that may be faster than the one generated by calling CKGas.

For more information on GRI-Mech, see http://www.me.berkeley.edu/gri_mech
2.3.2 Pure Substance Constructors

In addition to the ideal gas mixtures discussed above, Cantera implements several non-ideal pure fluids. These use accurate equations of state, and are valid in the vapor, liquid, and mixed liquid/vapor regions of the phase diagram. In most cases, the equation of state parameterizations are taken from the compilation by Reynolds [Reynolds, 1979].

The standard-state enthalpy of formation and absolute entropy have been adjusted to match the values in the JANAF tables. These objects should therefore be thermodynamically consistent with most gas mixture objects that use thermodynamic data obtained from fits to JANAF data.

3) Water

Constructs an object representing pure water, H$_2$O.

**Syntax**
```
object = Water()
```

**Result**
Returns an object of type `mixture_t`.

**Description**
The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is ?. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

**Example**
```
type(substance_t) h2o
h2o = Water()
```

4) Nitrogen

Construct an object representing nitrogen, N$_2$.

**Syntax**
```
object = Nitrogen()
```

**Result**
Returns an object of type `mixture_t`.

```
Description
The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is Jacobsen et al. [1972]. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

Example
```csharp
type(substance_t) subst
subst = Nitrogen()
```

<table>
<thead>
<tr>
<th>5)</th>
<th>Methane</th>
</tr>
</thead>
</table>

Construct a pure substance object representing methane, CH₄.

Syntax
```csharp
object = Methane()
```

Result
Returns an object of type mixture_t.

Description
The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is ?. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

Example
```csharp
type(substance_t) subst
subst = Methane()
```

<table>
<thead>
<tr>
<th>6)</th>
<th>Hydrogen</th>
</tr>
</thead>
</table>

Construct a pure substance object representing hydrogen, H₂.

Syntax
```csharp
object = Hydrogen()
```

Result
Returns an object of type mixture_t.

Description
The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is ?. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

2.3. Constructors
Example

```plaintext
type(substance_t) subst
subst = Hydrogen()
```

7) Oxygen

Construct a pure substance object representing oxygen, O\(_2\).

Syntax

```plaintext
object = Oxygen()
```

Result

Returns an object of type `mixture_t`.

Description

The equation of state parameterization is taken from the compilation by Reynolds [1979]. The original source is ?. The standard-state enthalpy of formation and the absolute entropy have been adjusted to match the values in the JANAF tables.

Example

```plaintext
type(substance_t) subst
subst = Oxygen()
```

Example 2.1

Constant-pressure lines for nitrogen are shown on a volume-temperature plot in Fig. 2.2. The data for this plot were generated by the program below. Note that the state of the nitrogen object may correspond to liquid, vapor, or saturated liquid/vapor.

```plaintext
program plotn2
use cantera
implicit double precision (a-h,o-z)
double precision pres(5), v(5)
data pres/1.d0, 3.d0, 10.d0, 30.d0, 100.d0/
type(substance_t) s
s = Nitrogen()
open(10, file='plot.csv', form='formatted')
t = minTemp(s)
do n = 1, 100
   t = t + 1.0
   do j = 1, 5
      call setState_TP(s, t, OneAtm*pres(j))
      v(j) = 1.0/density(s)
   end do
```

Chapter 2. Mixtures
Figure 2.2: Isobars at 1, 3, 10, 30, and 100 atm for nitrogen.

```fortran
write(10,20) t,(v(j), j = 1,5)
20 format(6(e14.5,', '))
end do
close(10)
stop
end
```

2.4 Utilities

The procedures in this section provide various utility functions.

8) **addDirectory**

Add a directory to the path to be searched for input files.

**Syntax**

```fortran```
call addDirectory(*dirname*)
```

**Arguments**

- `dirname` (Input) Directory name [character*(*)].
Description

Constructor functions may require data in external files. Cantera searches for these files first in the local directory. If not found there and environment variable $CANTERA\_DATA\_DIR$ is set to a directory name, then this directory will be searched. Additional directories may be added to the search path by calling this subroutine.

Example

call addDirectory('/usr/local/my_filea')

9) delete

Deletes the kernel mixture object.

Syntax

call delete(mix)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Description

When a mixture constructor is called, a kernel mixture object is dynamically allocated. This object stores all data required to compute the mixture thermodynamic, kinetic, and transport properties. For mixtures of many species, or ones with a large reaction mechanism, the size of the object may be large. Calling this subroutine frees this memory. After deleting a mixture, the object must be constructed again before using it.

10) ready

Returns true if the mixture has been successfully constructed and is ready for use.

Syntax

result = ready(mix)

Arguments

mix (Input) Mixture object [type(mixture_t)].

Result

Returns a logical value.
11) **copy**

Copies one mixture object to another.

**Syntax**

```
call copy(dest, src)
```

**Arguments**

- `dest` (Output) Mixture object [type(mixture_t)].
- `src` (Input) Mixture object [type(mixture_t)].

**Description**

Performs a “shallow” copy of one mixture object to another. Note that the kernel object itself is not copied, only the handle. Therefore, after calling `copy`, both objects point to the same kernel object. The contents of `dest` are overwritten. Assigning one mixture to another invokes this procedure.

**Example**

```plaintext
use cantera
type(mixture_t) fluid1, fluid2, fluid3
fluid1 = Water()
fluid2 = Methane()
write(*,*) 'fluid1 ready: ', ready(fluid1)
call delete(fluid1) ! fluid1 is now empty
write(*,*) 'fluid1 ready: ', ready(fluid1)
call copy(fluid1, fluid2) ! now fluid1 is Methane too
write(*,*) 'fluid1 ready: ', ready(fluid1)
fluid3 = fluid2 ! assignment calls ‘copy’
write(*,*) critTemperature(fluid1)
write(*,*) critTemperature(fluid2)
write(*,*) critTemperature(fluid3)
```

**Output:**

```
fluid1 ready: T
fluid1 ready: F
fluid1 ready: T
190.555000000000
190.555000000000
190.555000000000
```

12) **saveState**

Write the internal thermodynamic state information to an array.

---

2.4. Utilities 23
Syntax
   call saveState(mix, state)

Arguments

    mix       (Input) Mixture object [type(mixture_t)].
    state     (Output) Array where state information will be stored [double precision array]. Must be dimensioned at least as large as the number of species in mix + 2.

Description

calling saveState writes the internal data defining the thermodynamic state of a mixture object to an array. It can later be restored by calling restoreState.

Example

use cantera
real(8) state(60) ! dimension at least (# of species) + 2
type(mixture_t) mix
mix = GRIMech30()
call setState_TPX(mix, 400.d0, OneAtm, 'N2:0.2, H2:0.8')
write(*,*), entropy_mole(mix), moleFraction(mix,'N2')
call saveState(mix, state)
call setState_TPX(mix, 1000.d0, 1.d3, 'N2:0.8, CH4:0.2')
write(*,*), entropy_mole(mix), moleFraction(mix,'N2')
call restoreState(mix, state)
write(*,*), entropy_mole(mix), moleFraction(mix,'N2')

Output:

   155555.372874368  0.200000000000000
   236274.031245184  0.800000000000000
   155555.372874368  0.200000000000000

13) restoreState

Restore the mixture state from the data in array state. written by a prior call to saveState.

Syntax
   call restoreState(mix, state)

Arguments

    mix       (Input) Mixture object [type(mixture_t)].
    state     (Output) Array where state information will be stored [double precision array]. Must be dimensioned at least as large as the number of species in mix + 2.

Example  see the example for procedure 12.
2.5 Number of Elements, Species, and Reactions

14) nElements

Number of elements.

Syntax
\[
\text{result} = \text{nElements}(\text{mix})
\]

Arguments
\[
\text{mix} \quad \text{(Input) Mixture object [type(mixture_t)].}
\]

Result
Returns a integer value.

Example
\[
\text{nel} = \text{nElements}(\text{mix})
\]

15) nSpecies

Number of species.

Syntax
\[
\text{result} = \text{nSpecies}(\text{mix})
\]

Arguments
\[
\text{mix} \quad \text{(Input) Mixture object [type(mixture_t)].}
\]

Result
Returns a integer value.

Example
\[
\text{use cantera}
\text{type(mixture_t) mix}
\text{mix = GRIMech30()}
\text{write(*,*) ‘number of species = ‘,nSpecies(mix)}
\]

Output:
\[
\text{number of species} = 53
\]
16) \textit{nReactions}

Number of reactions.

 Syntax 
 \[
\text{result} = \text{nReactions}(\text{mix})
\]

 Arguments 
 \textit{mix} \hspace{1cm} \text{(Input) Mixture object \{type(mixture_t}\}.}

 Result 
 Returns a integer value.

 Example 
 \begin{verbatim}
use cantera
type(mixture_t) mix
mix = GRIMech30()
write(*,*) 'number of reactions = ', \text{nReactions}(\text{mix})
\end{verbatim}

 Output: 
 number of reactions = 325


2.6 Mixture Creation

The procedures in this section are designed to allow creation of arbitrary custom mixtures from Fortran. To use these procedures, first construct a skeleton mixture object using constructor \textit{Mixture}. This returns an “empty” mixture — one that has no constituents, and with do-nothing “null” property managers installed. By adding elements and species, and installing managers for thermodynamic properties, kinetics, and transport properties, arbitrary mixtures can be constructed.

\textbf{Release Note:} These capabilities are not yet fully functional in the Fortran yet. More procedures will be added soon.

17) \textit{Mixture}

Construct an empty mixture object with no constituents or property managers.

 Syntax 
 \[
\text{object} = \text{Mixture}()
\]

 Result 
 Returns an object of type \texttt{mixture_t}. 


Description

This constructor creates a skeleton object, to which individual components may be added by calling the procedures below.

Example

type(mixture_t) mix
mix = Mixture()

18) addElement

Add an element to the mixture.

Syntax

call addElement(mix, name, atomicWt)

Arguments

| mix      | (Input) Mixture object [type(mixture_t)]. |
| name     | (Input) Element name [character*(*)].    |
| atomicWt | (Input) Atomic weight in amu [double precision]. |

Example

type(mixture_t) mix
mix = Mixture()
call addElement(‘Si’, 28.0855)
call addElement(‘H’, 1.00797)
... ! add species, reactions, etc. (not yet implemented) ...

2.7 Properties of the Elements

The elements in a mixture are the entities that species are composed from, and must be conserved by reactions. Usually these correspond to elements of the periodic table, or one of their isotopes. However, for reactions involving charged species, it is useful to define some charged entity (usually an electron) as an element, to insure that all reactions conserve charge.

19) getElementNames

Get the names of the elements.
Syntax
call getElementNames(mix, enames)

Arguments

- **mix**  
  (Input) Mixture object [type(mixture_t)].
- **enames**  
  (Output) Element names [character(*) array]. Must be dimensioned at least as large as the number of elements in *mix*.

Description
The element names may be strings of any length. If they names are longer than the number of characters in each entry of the *enames* array, the returned names will be truncated.

---

20) **elementIndex**

Index number of an element.

Syntax
result = elementIndex(mix, name)

Arguments

- **mix**  
  (Input) Mixture object [type(mixture_t)].
- **name**  
  (Input) Element name [character(*)].

Result
Returns a integer value.

If the element is not present in the mixture, the value -1 is returned.

Description
The index number corresponds to the order in which the element was added to the mixture. The first element added has index number 1, the second has index number 2, and so on. All arrays of element properties (e.g., names, atomic weights) are ordered by index number.

Example

```fortran
! v1 contains element properties ordered as listed in 'mech1.inp'.
! copy these values to the the positions in array v2 which is
! ordered as in 'mech2.inp'
type(mixture_t) mix, mix2
mix1 = CKGas('mech1.inp')
mix2 = CKGas('mech2.inp')
doi = 1, nElements(mix1)
    ename = elementName(i)
    loc2 = elementIndex(mix2, ename)
    v2[loc2] = v1[i]
end do
```
See Also

speciesIndex

21) atomicWeight

The atomic weight of the $m^{th}$ element [kg/kmol].

Syntax

\[
\text{result} = \text{atomicWeight}(\text{mix}, m)
\]

Arguments

- \textit{mix} (Input) Mixture object [type(mixture_t)].
- \textit{m} (Input) Element index [integer].

Result

Returns a double precision value.

Example

\begin{verbatim}
character(20) names(10)
call getElementNames(mix, names)
do m = 1, nElements(mix)
   write(*,*) names(m), atomicWeight(mix, m)
end do
\end{verbatim}

2.8 Properties of the Species

The species are defined so that any achievable chemical composition for the mixture can be represented by specifying the mole fraction of each species. In gases, the species generally correspond to distinct molecules, atoms, or ions, and each is composed of integral numbers of atoms of the elements.

However, if the gas mixture contains molecules with populated metastable states, then to fully represent any possible mixture composition, these states (or groups of them) must also be defined as species. Common examples of such long-lived metastable states include singlet $\Delta$ oxygen and singlet methylene.

In solid or liquid solutions, it may be necessary to define species composed of non-integral numbers of elements. This is acceptable, as long as all realizable states of the mixture correspond to some set of mole fractions for the species so defined.

22) nAtoms

Number of atoms of element $m$ in species $k$
Syntax
result = nAtoms(mix, k, m)

Arguments

mix (Input) Mixture object [type(mixture_t)].
k (Input) Species index [integer].
m (Input) Element index [integer].

Result
Returns a double precision value.
Non-integral and/or negative values are allowed. For example, if an electron is defined to be an element, then a positive ion has a negative number of “atoms” of this element.

23) charge

Electrical charge of the \( k^{th} \) species, in multiples of \( e \), the magnitude of the electron charge.

Syntax
result = charge(mix, k)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].
k (Input) Species index [integer].

Result
Returns a double precision value.
In most cases, the charge is integral, but non-integral values are allowed.

24) speciesIndex

Index number of a species.

Syntax
result = speciesIndex(mix, name)

Arguments

mix (Input) Mixture object [type(mixture_t)].
name (Input) Species name [character\(^{(*)}\)].
Result
Returns an integer value.

Description
The index number corresponds to the order in which the species was added to the mixture. All species property arrays are ordered by index number.

25) getSpeciesNames

Get the array of species names.

Syntax
call getSpeciesNames(mix, snames)

Arguments
- **mix** (Input) Mixture object [type(mixture_t)].
- **snames** (Output) Array of species names [character*(*) array]. Must be dimensioned at least as large as the number of species in mix.

Description
The species names may be strings of any length. If the species names are longer than the number of characters in each element of the snames array, the returned names will be truncated.

26) molecularWeight

The molecular weight of species \( k \) [amu].

Syntax
result = molecularWeight(mix, k)

Arguments
- **mix** (Input) Mixture object [type(mixture_t)].
- **k** (Input) Species index [integer].

Result
Returns a double precision value.
27) getMolecularWeights

Get the array of species molecular weights.

Syntax

call getMolecularWeights(mix, molWts)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].
molWts (Output) Species molecular weights [double precision array]. Must be dimensioned at least as large as the number of species in mix.

2.9 Mixture Composition

The mixture composition may be specified by mole fractions, mass fractions, or concentrations. The procedures described here set or get the mixture composition. Additional procedures that set the composition are described in the next chapter.

28) setMoleFractions

Set the species mole fractions, holding the temperature and mass density fixed.

Syntax

call setMoleFractions(mix, moleFracs)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].
moleFracs (Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in mix.

29) setMoleFractions_NoNorm

Set the species mole fractions without normalizing them to sum to 1.

Syntax

call setMoleFractions_NoNorm(mix, moleFracs)

Arguments
mix  (Input / Output) Mixture object [type(mixture_t)].
moleFracs (Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in mix.

Description
This procedure is designed for use when perturbing the composition, for example when computing Jacobians.

30) setMassFractions

Set the species mass fractions, holding the temperature and mass density fixed.

Syntax
    call setMassFractions(mix, massFracs)

Arguments
    mix   (Input / Output) Mixture object [type(mixture_t)].
    massFracs (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in mix.

31) setMassFractions_NoNorm

Set the species mass fractions without normalizing them to sum to 1.

Syntax
    call setMassFractions_NoNorm(mix, y)

Arguments
    mix   (Input / Output) Mixture object [type(mixture_t)].
    y    (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in mix.

Description
This procedure is designed for use when perturbing the composition, for example when computing Jacobians.
Set the species concentrations, holding the temperature fixed.

**Syntax**

```plaintext
call setConcentrations(mix, conc)
```

**Arguments**

- `mix` (Input / Output) Mixture object [type(mixture_t)].
- `conc` (Input) Species concentrations [kmol/m³] [double precision array]. Must be dimensioned at least as large as the number of species in `mix`.

---

Mole fraction of one species.

**Syntax**

```plaintext
result = moleFraction(mix, name)
```

**Arguments**

- `mix` (Input) Mixture object [type(mixture_t)].
- `name` (Input) Species name [character*(*)].

**Result**

Returns a double precision value.

**Description**

The mole fraction of the species with name `name` is returned. If no species in the mixture has this name, the value zero is returned.

---

Mass fraction of one species.

**Syntax**

```plaintext
result = massFraction(mix, name)
```

**Arguments**

- `mix` (Input) Mixture object [type(mixture_t)].
name  (Input) Species name [character*(*)].

Result
Returns a double precision value.

Description
The mass fraction of the species with name name is returned. If no species in the mixture has this name, the value zero is returned.

35) getMoleFractions

Get the array of species mole fractions.

Syntax
call getMoleFractions(mix, moleFracs)

Arguments
mix  (Input) Mixture object [type(mixture_t)].
moleFracs  (Input / Output) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in mix.

36) getMassFractions

Get the array of species mass fractions.

Syntax
call getMassFractions(mix, massFracs)

Arguments
mix  (Input) Mixture object [type(mixture_t)].
massFracs  (Input / Output) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in mix.

37) getConcentrations

Get the array of species concentrations.
Syntax
    call getConcentrations(mix, concentrations)

Arguments
    mix   (Input) Mixture object [type(mixture_t)].
    concentrations (Input / Output) Species concentrations. [kmol/m³] [double precision array]. Must be dimensioned at least as large as the number of species in mix.

2.10 Mean Properties

These procedures return weighted averages of pure-species properties.

38) \( \text{mean}_X \)

Return \( \sum_k Q_k X_k \), the mole-fraction-weighted mean value of pure species molar property \( Q \).

Syntax
    result = mean_X(mix, Q)

Arguments
    mix   (Input) Mixture object [type(mixture_t)].
    Q     (Input) Array of species molar property values [double precision array]. Must be dimensioned at least as large as the number of species in mix.

Result
    Returns a double precision value.

39) \( \text{mean}_Y \)

Return \( \sum_k Q_k Y_k \), the mass-fraction-weighted mean value of pure species specific property \( Q \).

Syntax
    result = mean_Y(mix, Q)

Arguments
    mix   (Input) Mixture object [type(mixture_t)].
    Q     (Input) Array of species specific property values [double precision array]. Must be dimensioned at least as large as the number of species in mix.

Result
    Returns a double precision value.
The mean molecular weight [amu].

Syntax

\[ \text{result} = \text{meanMolecularWeight}(\text{mix}) \]

Arguments

- \( \text{mix} \) (Input) Mixture object [type(mixture_t)].

Result

Returns a double precision value.

Description

The mean molecular weight is defined by

\[ \bar{M} = \sum_k M_k X_k. \]  \hspace{1cm} (2.1)

An equivalent alternative expression is

\[ \frac{1}{\bar{M}} = \sum_k \frac{Y_k}{M_k}. \]  \hspace{1cm} (2.2)

See Also

molecularWeight

---

Given a vector \( Q \), returns \( \sum_k X_k \log Q_k \).

Syntax

\[ \text{result} = \text{sum_xlogQ}(\text{mix}, Q) \]

Arguments

- \( \text{mix} \) (Input) Mixture object [type(mixture_t)].
- \( Q \) (Output) Array of species property values [double precision array]. Must be dimensioned at least as large as the number of species in \( \text{mix} \).

Result

Returns a double precision value.
In the last chapter, we saw how to create a basic mixture object. In this chapter, we’ll look at how to set the thermodynamic state in various ways, and compute various thermodynamic properties.

2.11 Procedures

2.12 The Thermodynamic State

Although a long list of properties of a mixture can be written down, most of them are not independent — once a few values have been specified, all the rest are fixed.

The number of independent properties is equal to the number of different ways any one property can be altered. If we consider the Gibbs equation written for $dU$

$$dU = TdS - PdV + \sum_k \mu_k N_k,$$

(2.3)

we see that $U$ can be changed by heat addition ($TdS$) or by compression ($-PdV$), or by changing any one of $K$ mole numbers. Therefore, the number of degrees of freedom is $K+2$.

However, if we are interested only in the intensive state, there are only $K+1$ independently variable parameters, since one degree of freedom simply sets the total mass or number of moles.

The thermodynamic state is determined by specifying any $K+1$ independent property values. Of these, $K-1$ specify the composition, and are usually taken to be either mole fractions or mass fractions, although the chemical potentials can be used too. The two remaining can be any two independent mixture properties.

These procedures described here set the thermodynamic state of a mixture, or modify its current state.

Procedures that set the mole fractions or mass fractions take an array of $K$ values, even though only $K-1$ are independent. Unless otherwise noted, these input values will be internally normalized to satisfy

$$\sum_k X_k = 1$$

(2.4)

or

$$\sum_k Y_k = 1.$$

(2.5)

For those procedures that take as an argument the internal energy, enthalpy, or entropy, the value per unit mass must be used.
2.12.1 Setting the State

The procedures described below set the full thermodynamic state of a mixture. They take as arguments two thermodynamic properties and an array or string that specifies the composition. The state of the mixture after calling any of these is independent of its state prior to the call.

42) setState_TPX

Set the temperature, pressure, and mole fractions.

Syntax

call setState_TPX(mix, t, p, moleFracs)

Arguments

- **mix** (Input / Output) Mixture object [type(mixture_t)].
- **t** (Input) Temperature [K] [double precision].
- **p** (Input) Pressure [Pa] [double precision].
- **moleFracs** (Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in **mix**.

Description

There are two versions of **setState_TPX**. This one uses an array to specify the mole fractions, and the other (below) uses a string.

Example

```fortran
double precision x(100)
...
t = 1700.d0
p = 0.1 * OneAtm
do k=1, nSpecies(mix)
    x(k) = 0.d0
end do
i_nh3 = speciesIndex(mix, 'NH3')
i_h2 = speciesIndex(mix, 'H2')
x(i_nh3) = 2.d0
x(i_h2) = 1.d0
call setState_TPX(mix, t, p, x)
```

43) setState_TPX

Set the temperature, pressure, and mole fractions.
Syntax
   call setState_TPX(mix, t, p, moleFracs)

Arguments
   mix   (Input / Output) Mixture object [type(mixture_t)].
   t     (Input) Temperature [K] [double precision].
   p     (Input) Pressure [Pa] [double precision].
   moleFracs (Input) Species mole fraction string [character*(*)].

Description
   There are two versions of setState_TPX. This one uses a string to specify the mole fractions, and the other (above) uses an array. The string should contain comma-separated name/value pairs, each of which is separated by a colon, such as 'CH4:1, O2:2, N2:7.52'. The values do not need to sum to 1.0 - they will be internally normalized. The mole fractions of all other species in the mixture are set to zero.

Example
   real(8) t, p
   character(50) x
   t = 300.d0
   p = OneAtm
   x = 'H2:3, SIH4:0.1, AR:60'
   call setState_TPX(t, p, x)

44) setState_TPY

Set the temperature, pressure, and mass fractions.

Syntax
   call setState_TPY(mix, t, p, massFracs)

Arguments
   mix   (Input / Output) Mixture object [type(mixture_t)].
   t     (Input) Temperature [K] [double precision].
   p     (Input) Pressure [Pa] [double precision].
   massFracs (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in mix.

Description
   There are two versions of setState_TPY. This one uses an array to specify the mass fractions, and the other (below) uses a string.
45) **setState_TPY**

Set the temperature, pressure, and mass fractions.

**Syntax**

```fortran
call setState_TPY(mix, t, p, massFracs)
```

**Arguments**

- `mix` (Input / Output) Mixture object [type(mixture_t)].
- `t` (Input) Temperature [K] [double precision].
- `p` (Input) Pressure [Pa] [double precision].
- `massFracs` (Input) Species mass fraction string [character*(*)].

**Description**

There are two versions of `setState_TPY`. This one uses a string to specify the mole fractions, and the other (above) uses an array. The string should contain comma-separated name/value pairs, each of which is separated by a colon, such as `CH4:2, O2:1, N2:10` (the values do not need to sum to 1.0 - they will be internally normalized). The mass fractions of all other species in the mixture will be set to zero.

**Example**

```fortran
real(8) t, p
character(50) x

x = 'H2O:5, NH3:0.1, N2:20'

call setState_TPY(t, p, x)
```

46) **setState_TRX**

Set the temperature, density, and mole fractions.

**Syntax**

```fortran
call setState_TRX(mix, t, density, moleFracs)
```

**Arguments**

- `mix` (Input / Output) Mixture object [type(mixture_t)].
- `t` (Input) Temperature [K] [double precision].
- `density` (Input) Density [kg/m³] [double precision].
- `moleFracs` (Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in `mix`.
Set the temperature, density, and mass fractions.

Syntax
   call setStateTRY(mix, t, density, massFracs)

Arguments
   mix  (Input / Output) Mixture object [type(mixture_t)].
   t    (Input) Temperature [K] [double precision].
   density  (Input) Density [kg/m$^3$] [double precision].
   massFracs  (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in mix.

2.12.2 Modifying the Current State

The procedures in this section set only one or two properties, leaving the temperature, density, and/or composition unchanged.

Set the temperature, holding the mass density and composition fixed.

Syntax
   call setTemperature(mix, temp)

Arguments
   mix  (Input / Output) Mixture object [type(mixture_t)].
   temp  (Input) Temperature [K]. [double precision].

Set the density, holding the temperature and composition fixed.

Syntax
   call setDensity(mix, density)

Arguments
   mix  (Input / Output) Mixture object [type(mixture_t)].
   density  (Input) Density [kg/m$^3$] [double precision].
50) **setState**\_TP

Set the temperature and pressure, holding the composition fixed.

**Syntax**

call **setState**\_TP(mix, t, p)

**Arguments**

- **mix** (Input / Output) Mixture object [type(mixture\_t)].
- **t** (Input) Temperature [K] [double precision].
- **p** (Input) Pressure [Pa] [double precision].

51) **setState**\_PX

Set the pressure and mole fractions, holding the temperature fixed.

**Syntax**

call **setState**\_PX(mix, p, x)

**Arguments**

- **mix** (Input / Output) Mixture object [type(mixture\_t)].
- **p** (Input) Pressure [Pa] [double precision].
- **x** (Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in **mix**.

52) **setState**\_PY

Set the pressure and mass fractions, holding the temperature fixed.

**Syntax**

call **setState**\_PY(mix, p, massFracs)

**Arguments**

- **mix** (Input / Output) Mixture object [type(mixture\_t)].
- **p** (Input) Pressure [Pa] [double precision].
- **massFracs** (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in **mix**.
setPressure

Set the pressure, holding the temperature and composition fixed.

Syntax
   call setPressure(mix, p)

Arguments
   mix  (Input / Output) Mixture object [type(mixture_t)].
   p    (Input) Pressure [Pa] [double precision].

setState_TR

Set the temperature and density, holding the composition fixed.

Syntax
   call setState_TR(mix, t, rho)

Arguments
   mix  (Input / Output) Mixture object [type(mixture_t)].
   t    (Input) Temperature [K] [double precision].
   rho  (Input) Density [kg/m^3] [double precision].

setState_TX

Set the temperature and mole fractions, holding the density fixed.

Syntax
   call setState_TX(mix, t, moleFracs)

Arguments
   mix  (Input / Output) Mixture object [type(mixture_t)].
   t    (Input) Temperature [K] [double precision].
   moleFracs (Input / Output) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in mix.
56) setState\_TY

Set the temperature and mass fractions, holding the density fixed.

**Syntax**

```plaintext
call setState\_TY(mix, t, massFracs)
```

**Arguments**

- `mix` (Input / Output) Mixture object [type(mixture\_t)].
- `t` (Input) Temperature [K] [double precision].
- `massFracs` (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in `mix`.

57) setState\_RX

Set the density and mole fractions, holding the temperature fixed.

**Syntax**

```plaintext
call setState\_RX(mix, density, moleFracs)
```

**Arguments**

- `mix` (Input / Output) Mixture object [type(mixture\_t)].
- `density` (Input) Density [kg/m\(^3\)] [double precision].
- `moleFracs` (Input) Species mole fractions [double precision array]. Must be dimensioned at least as large as the number of species in `mix`.

58) setState\_RY

Set the density and mass fractions, holding the temperature fixed.

**Syntax**

```plaintext
call setState\_RY(mix, density, y)
```

**Arguments**

- `mix` (Input / Output) Mixture object [type(mixture\_t)].
- `density` (Input) Density [kg/m\(^3\)] [double precision].
- `y` (Input) Species mass fractions [double precision array]. Must be dimensioned at least as large as the number of species in `mix`.  

2.12. The Thermodynamic State
setState_HP

Set the specific enthalpy and pressure, holding the composition fixed.

Syntax

\[
\text{call setState_HP}(\text{mix}, t, p)
\]

Arguments

- \( \text{mix} \) (Input / Output) Mixture object [type(mixture_t)].
- \( t \) (Input) Specific enthalpy [J/kg] [double precision].
- \( p \) (Input) Pressure [Pa] [double precision].

Description

This procedure uses a Newton method to find the temperature at which the specific enthalpy has the desired value. The iteration proceeds until the temperature changes by less than 0.001 K. At this point, the state is set to the new temperature value, and the procedure returns. The iterations are done at constant pressure and composition.

This algorithm always takes at least one Newton step. Therefore, small perturbations to the input enthalpy should produce a first-order change in temperature, even if it is well below the temperature error tolerance, allowing its use in routines that calculate derivatives or Jacobians. To test this, in the example below, the specific heat is calculated numerically using this procedure, and compared to the value obtained directly from its parameterization (here a polynomial). (Note that this is not the most direct way to calculate \( c_p \) numerically – normally, \( T \) would be perturbed, not \( h \).)

Example

```fortran
use cantera
real(8) h0, t1, t2, state(100)
type(mixture_t) mix
mix = GRIMech30()

call setState_TPX(mix,2000.d0,'OneAtm','CH4:1,O2:2')
call saveState(mix, state)
h0 = enthalpy_mass(mix)
call setState_HP(mix, h0, 'OneAtm')
t1 = temperature(mix)
call setState_HP(mix, h0 + h0*1.d-8, 'OneAtm')
t2 = temperature(mix)
call restoreState(mix, state)
write(*,*) (h0*1.d-8)/(t2 - t1), cp_mass(mix)
return
```

Output:

```
2199.12043414729     2199.12045953925
```

46 Chapter 2. Mixtures
60) setState_UV

Set the specific internal energy and specific volume, holding the composition constant.

**Syntax**
call setState_UV(mix, t, p)

**Arguments**
- mix (Input / Output) Mixture object [type(mixture_t)].
- t (Input) Specific internal energy [J/kg] [double precision].
- p (Input) Specific volume [m³/kg] [double precision].

**Description**
See the discussion of procedure *setState_HP*, procedure number 59.

61) setState_SP

Set the specific entropy and pressure, holding the composition constant.

**Syntax**
call setState_SP(mix, t, p)

**Arguments**
- mix (Input / Output) Mixture object [type(mixture_t)].
- t (Input) Specific entropy [J/kg/K] [double precision].
- p (Input) Pressure [Pa] [double precision].

**Description**
See the discussion of procedure *setState_HP*, procedure number 59.

62) setState_SV

Set the specific entropy and specific volume, holding the composition constant.

**Syntax**
call setState_SV(mix, t, p)

**Arguments**
- mix (Input / Output) Mixture object [type(mixture_t)].
2.13 Species Ideal Gas Properties

The procedures in this section return non-dimensional ideal gas thermodynamic properties of the pure species at the standard-state pressure \( P_0 \) and the mixture temperature \( T \). As discussed in Section ??, the properties of any non-ideal mixture can be determined given the mechanical equation of state \( P(T, \rho) \) and the pure-species properties evaluated for the low-density, ideal-gas limit. The procedures below return arrays of pure species ideal gas properties. For those that depend on \( \ln \rho \), the values are scaled to a density \( \rho = P_0 / RT \), where \( P_0 \) is the standard-state pressure.

63) \texttt{getEnthalpy\_RT}

Get the array of pure-species non-dimensional enthalpies

\[
\hat{h}_k(T, P_0) / \hat{R}T, \quad k = 1 \ldots K.
\] (2.6)

Syntax

\begin{verbatim}
call getEnthalpy\_RT(mix, h\_RT)
\end{verbatim}

Arguments

- \texttt{mix} (Input) Mixture object [type(mixture\_t)].
- \texttt{h\_RT} (Output) Array of non-dimensional pure species enthalpies [double precision array]. Must be dimensioned at least as large as the number of species in \texttt{mix}.

64) \texttt{getGibbs\_RT}

Get the array of pure-species non-dimensional Gibbs functions

\[
\hat{g}_k(T, P_0) / \hat{R}T, \quad k = 1 \ldots K.
\] (2.7)

Syntax

\begin{verbatim}
call getGibbs\_RT(mix, g\_RT)
\end{verbatim}
Arguments

\texttt{mix} (Input) Mixture object [type(mixture_t)].
\texttt{g\_RT} (Output) Array of non-dimensional pure species Gibbs functions [double precision array]. Must be dimensioned at least as large as the number of species in \texttt{mix}.

65) \texttt{getCp\_R}

Get the array of pure-species non-dimensional heat capacities at constant pressure
\[ \hat{c}_{p,k}(T, P_0)/\hat{R}, \quad k = 1 \ldots K. \] (2.8)

Syntax
\texttt{call getCp\_R(mix, cp\_R)}

Arguments

\texttt{mix} (Input) Mixture object [type(mixture_t)].
\texttt{cp\_R} (Output) Array of non-dimensional pure species heat capacities at constant pressure [double precision array]. Must be dimensioned at least as large as the number of species in \texttt{mix}.

66) \texttt{getEntropy\_R}

Get the array of pure-species non-dimensional entropies
\[ \hat{s}_k(T, P_0)/\hat{R}, \quad k = 1 \ldots K. \] (2.9)

Syntax
\texttt{call getEntropy\_R(mix, s\_R)}

Arguments

\texttt{mix} (Input) Mixture object [type(mixture_t)].
\texttt{s\_R} (Output) Array of non-dimensional pure species entropies [double precision array]. Must be dimensioned at least as large as the number of species in \texttt{mix}.

2.14 Attributes of the Parameterization

These procedures return attributes of the thermodynamic property parameterization used.
67) minTemp

Lowest temperature for which the thermodynamic properties are valid [K].

Syntax
result = minTemp(mix)

Arguments
mix (Input) Mixture object [type(mixture_t)].

Result
Returns a double precision value.

68) maxTemp

Highest temperature for which the thermodynamic properties are valid [K].

Syntax
result = maxTemp(mix)

Arguments
mix (Input) Mixture object [type(mixture_t)].

Result
Returns a double precision value.

69) refPressure

The reference (standard-state) pressure $P_0$. Species thermodynamic property data are specified as a function of temperature at pressure $P_0$. Usually, $P_0$ is 1 atm, but some property data sources use 1 bar. Any choice is acceptable, as long as the data for all species in the mixture use the same one.

Syntax
result = refPressure(mix)

Arguments
mix (Input / Output) Mixture object [type(mixture_t)].

Result
Returns a double precision value.
2.15 Mixture Thermodynamic Properties

2.15.1 Temperature, Pressure, and Density

70) \textbf{temperature}

The temperature [K].

\textbf{Syntax}
\[ \text{result} = \text{temperature}(\text{mix}) \]

\textbf{Arguments}
\[ \text{mix} \quad \text{(Input) Mixture object \{type(mixture_t)\}.} \]

\textbf{Result}
Returns a double precision value.

71) \textbf{density}

The density [kg/m$^3$].

\textbf{Syntax}
\[ \text{result} = \text{density}(\text{mix}) \]

\textbf{Arguments}
\[ \text{mix} \quad \text{(Input) Mixture object \{type(mixture_t)\}.} \]

\textbf{Result}
Returns a double precision value.

72) \textbf{molarDensity}

The number of moles per unit volume [kmol/m$^3$].

\textbf{Syntax}
\[ \text{result} = \text{molarDensity}(\text{mix}) \]

\textbf{Arguments}
\[ \text{mix} \quad \text{(Input) Mixture object \{type(mixture_t)\}.} \]

\textbf{Result}
Returns a double precision value.
73) pressure

The pressure [Pa].

Syntax

\[ \text{result} = \text{pressure}(\text{mix}) \]

Arguments

\[ \text{mix} \quad \text{(Input) Mixture object [type(mixture_t)].} \]

Result

Returns a double precision value.

2.15.2 Molar Mixture Properties

The procedures in this section all return molar thermodynamic properties. The names all end in \textit{mole}, which should be read as “per mole.”

74) enthalpy\_mole

Molar enthalpy [J/kmol]

Syntax

\[ \text{result} = \text{enthalpy\_mole}(\text{mix}) \]

Arguments

\[ \text{mix} \quad \text{(Input / Output) Mixture object [type(mixture\_t)].} \]

Result

Returns a double precision value.

75) intEnergy\_mole

Molar internal energy [J/kmol]

Syntax

\[ \text{result} = \text{intEnergy\_mole}(\text{mix}) \]

Arguments

\[ \text{mix} \quad \text{(Input / Output) Mixture object [type(mixture\_t)].} \]
Molar entropy [J/kmol/K]

**Syntax**

```plaintext
result = entropy_mole(mix)
```

**Arguments**

- `mix` (Input / Output) Mixture object [type(mixture_t)].

**Result**

Returns a double precision value.

---

Molar Gibbs function [J/kmol/K]

**Syntax**

```plaintext
result = gibbs_mole(mix)
```

**Arguments**

- `mix` (Input / Output) Mixture object [type(mixture_t)].

**Result**

Returns a double precision value.

---

Molar heat capacity at constant pressure [J/kmol/K]

**Syntax**

```plaintext
result = cp_mole(mix)
```

**Arguments**

- `mix` (Input) Mixture object [type(mixture_t)].

**Result**

Returns a double precision value.
Molar heat capacity at constant volume [J/kmol/K]

**Syntax**

\[
\text{result} = \text{cv\_mole}(\text{mix})
\]

**Arguments**

- \( \text{mix} \) (Input) Mixture object [type(mixture_t)].

**Result**

Returns a double precision value.

---

Get the species chemical potentials [J/kmol].

**Syntax**

\[
\text{call getChemPotentials\_RT(} \text{mix, mu}\text{)}
\]

**Arguments**

- \( \text{mix} \) (Input) Mixture object [type(mixture_t)].
- \( \text{mu} \) (Output) Species chemical potentials [double precision array]. Must be dimensioned at least as large as the number of species in \( \text{mix} \).

---

### 2.15.3 Specific Mixture Properties

The procedures in this section all return thermodynamic properties per unit mass. The names all end in \_mass, which should be read as “per (unit) mass.”

---

Specific enthalpy [J/kg]

**Syntax**

\[
\text{result} = \text{enthalpy\_mass}(\text{mix})
\]

**Arguments**

- \( \text{mix} \) (Input) Mixture object [type(mixture_t)].

**Result**

Returns a double precision value.
82) \texttt{intEnergy\_mass}

Specific internal energy [J/kg]

**Syntax**

\[ \text{result} = \text{intEnergy}\_\text{mass}(\text{mix}) \]

**Arguments**

\( \text{mix} \) (Input) Mixture object [type(mixture\_t)].

**Result**

Returns a double precision value.

83) \texttt{entropy\_mass}

Specific entropy [J/kg/K]

**Syntax**

\[ \text{result} = \text{entropy}\_\text{mass}(\text{mix}) \]

**Arguments**

\( \text{mix} \) (Input) Mixture object [type(mixture\_t)].

**Result**

Returns a double precision value.

84) \texttt{gibbs\_mass}

Specific Gibbs function [J/kg]

**Syntax**

\[ \text{result} = \text{gibbs}\_\text{mass}(\text{mix}) \]

**Arguments**

\( \text{mix} \) (Input) Mixture object [type(mixture\_t)].

**Result**

Returns a double precision value.
85) \texttt{cp\_mass}

Specific heat at constant pressure \([\text{J/kg/K}]\)

\textbf{Syntax}
\[ \text{result} = \text{cp\_mass}(\text{mix}) \]

\textbf{Arguments}
\[ \text{mix} \quad \text{(Input) Mixture object [type(mixture\_t)]} \]

\textbf{Result}
Returns a \textit{double precision} value.

86) \texttt{cv\_mass}

Specific heat at constant volume \([\text{J/kg/K}]\)

\textbf{Syntax}
\[ \text{result} = \text{cv\_mass}(\text{mix}) \]

\textbf{Arguments}
\[ \text{mix} \quad \text{(Input) Mixture object [type(mixture\_t)]} \]

\textbf{Result}
Returns a \textit{double precision} value.

2.16 Potential Energy

For problems in which external conservative fields play an important role, it may be desirable to assign potential energies to the species. The potential energy adds to the internal energy, but does not affect the entropy. Species-specific potential energies (for example, proportional to charge) can alter the reaction equilibrium constants, and therefore both kinetics and the chemical equilibrium composition. These effects are important in electrochemistry.

87) \texttt{setPotentialEnergy}

Set the potential energy of species \(k\) \([\text{J/kmol}]\) due to external conservative fields (electric, gravitational, or other). Default: 0.0.
Syntax
    call setPotentialEnergy(mix, k, pe)

Arguments
    mix  (Input / Output) Mixture object [type(mixture_t)].
    k    (Input) Species index [integer].
    pe   (Input) Potential energy [double precision].

The potential energy of species $k$.

Syntax
    result = potentialEnergy(mix, k)

Arguments
    mix  (Input / Output) Mixture object [type(mixture_t)].
    k    (Input) Species index [integer].

Result
    Returns a double precision value.

2.17 Critical State Properties

These procedures return parameters of the critical state. Not all equation of state managers implement these, in which case the value undefined is returned.

Syntax
    result = critTemperature(mix)

Arguments
    mix  (Input) Mixture object [type(mixture_t)].

Result
    Returns a double precision value.
90) **critPressure**

The critical pressure [Pa].

**Syntax**

```plaintext
result = critPressure(mix)
```

**Arguments**

- **mix** (Input) Mixture object [type(mixture_t)].

**Result**

Returns a **double precision** value.

**Example**

```plaintext
use cantera
type(mixture_t) h2o
h2o = Water()
write(*,*), 'Pcrit = ', critPressure(h2o)
```

**Output:**

`Pcrit = 22089000.000000`

### 2.18 Saturation Properties

These procedures are only implemented by models that treat saturated liquid/vapor states. Calling them with any other equation of state manager installed will result in an error.

91) **satTemperature**

The saturation temperature at pressure $p$ [K]. An error results if $P > P_{crit}$.

**Syntax**

```plaintext
result = satTemperature(mix, p)
```

**Arguments**

- **mix** (Input) Mixture object [type(mixture_t)].
- **p** (Input) Pressure [Pa] [double precision].

**Result**

Returns a **double precision** value.

**Example**

```plaintext
use cantera
type(mixture_t) h2o
h2o = Water()
write(*,*), 'Tsat = ', satTemperature(h2o, 101325.0)
```

**Output:**

`Tsat = 373.150000`
use cantera
type(mixture_t) h2o
h2o = Water()
write(*,*) 'Tsat @ 1 atm = ', satTemperature(h2o, OneAtm)

Output:

Tsat @ 1 atm = 373.177232956890

92) satPressure

The saturation pressure (vapor pressure) at temperature $t$ [K]. An error results if $T > T_{crit}$.

Syntax
result = satPressure(mix, t)

Arguments
mix (Input) Mixture object [type(mixture_t)].
t (Input) Temperature [K] [double precision].

Result
Returns a double precision value.

Example
use cantera
type(mixture_t) h2o
h2o = Water()
write(*,*) 'Psat @ 300 K = ', satPressure(h2o, 300.d0)

Output:

Psat @ 300 K = 3528.21380534104

2.19 Equations of State

93) equationOfState

Return the equation of state object.

Syntax
result = equationOfState(mix)
Arguments

\textit{mix} (Input / Output) Mixture object \([\text{type}(\text{mixture}_t)]\).

Result

Returns a \textit{type} \((\text{eos}_t)\) value.

94) \begin{verbatim}
setEquationOfState
\end{verbatim}

Set the equation of state.

Syntax

\begin{verbatim}
call setEquationOfState(\textit{mix}, \textit{eos})
\end{verbatim}

Arguments

\begin{itemize}
  \item \textit{mix} (Input / Output) Mixture object \([\text{type}(\text{mixture}_t)]\).
  \item \textit{eos} (Input / Output) Equation of state object \([\text{type}(\text{eos}_t)]\).
\end{itemize}
Chemical Equilibrium

A discussion of chemical equilibrium, the solver algorithm, etc. will soon replace this box...

95) equilibrate

Chemically equilibrate a mixture.

Syntax

call equilibrate(mix, propPair)

Arguments

mix (Input / Output) Mixture object [type(mixture_t)].
propPair (Input) Property pair to hold fixed [character*(*)].

Description

This subroutine sets the mixture to a state of chemical equilibrium, holding two thermodynamic properties fixed at their initial values. The pair of properties is specified by a two-character string, as shown in the table below.

<table>
<thead>
<tr>
<th>String</th>
<th>Property 1</th>
<th>Property 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td>temperature</td>
<td>pressure</td>
</tr>
<tr>
<td>TV</td>
<td>temperature</td>
<td>specific volume</td>
</tr>
<tr>
<td>HP</td>
<td>specific enthalpy</td>
<td>pressure</td>
</tr>
<tr>
<td>SP</td>
<td>specific entropy</td>
<td>pressure</td>
</tr>
<tr>
<td>SV</td>
<td>specific entropy</td>
<td>specific volume</td>
</tr>
<tr>
<td>UV</td>
<td>specific internal energy</td>
<td>specific volume</td>
</tr>
</tbody>
</table>

Example

use cantera
type(mixture_t) mix
mix = GRIMech30()
call setState_TPX(mix, 300.d0, OneAtm,
$ \log ('CH_4:0.9, O_2:2, N_2:7.52')$
call equilibrate(mix,'HP')
call printSummary(mix, 6)

Output:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>temperature</td>
<td>2134.24 K</td>
</tr>
<tr>
<td>pressure</td>
<td>101325 Pa</td>
</tr>
<tr>
<td>density</td>
<td>0.158033 kg/m^3</td>
</tr>
<tr>
<td>mean mol. weight</td>
<td>27.6748 amu</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enthalpy</td>
<td>-230208 J/kg, -6.371e+006 J/kmol</td>
</tr>
<tr>
<td>internal energy</td>
<td>-871370 J/kg, -2.412e+007 J/kmol</td>
</tr>
<tr>
<td>entropy</td>
<td>9727.6 J/kg, 2.692e+005 J/Kk</td>
</tr>
<tr>
<td>Gibbs function</td>
<td>-2.09912e+007 J/kg, -5.809e+008 J/kmol</td>
</tr>
<tr>
<td>heat capacity c_p</td>
<td>1484.63 J/kg, 4.109e+004 J/Kk</td>
</tr>
<tr>
<td>heat capacity c_v</td>
<td>1184.22 J/kg, 3.277e+004 J/Kk</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Substance</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>H_2</td>
<td>9.339031e-004</td>
<td>6.802905e-005</td>
</tr>
<tr>
<td>H</td>
<td>1.174265e-004</td>
<td>4.276898e-006</td>
</tr>
<tr>
<td>O</td>
<td>2.387901e-004</td>
<td>1.380495e-004</td>
</tr>
<tr>
<td>O_2</td>
<td>1.847396e-002</td>
<td>2.136037e-002</td>
</tr>
<tr>
<td>OH</td>
<td>2.687186e-003</td>
<td>1.651391e-003</td>
</tr>
<tr>
<td>H_2O</td>
<td>1.699745e-001</td>
<td>1.106474e-001</td>
</tr>
<tr>
<td>HO_2</td>
<td>9.949030e-007</td>
<td>1.186585e-006</td>
</tr>
<tr>
<td>H_2O_2</td>
<td>6.557154e-008</td>
<td>8.059304e-008</td>
</tr>
<tr>
<td>C</td>
<td>4.158314e-019</td>
<td>1.804749e-019</td>
</tr>
<tr>
<td>CH</td>
<td>4.275343e-020</td>
<td>2.011257e-020</td>
</tr>
<tr>
<td>CH_2</td>
<td>1.037590e-019</td>
<td>5.259063e-020</td>
</tr>
<tr>
<td>CH_2(S)</td>
<td>0.000000e+000</td>
<td>0.000000e+000</td>
</tr>
<tr>
<td>CH_3</td>
<td>5.912788e-019</td>
<td>3.212273e-019</td>
</tr>
<tr>
<td>CH_4</td>
<td>2.440321e-019</td>
<td>1.414648e-019</td>
</tr>
<tr>
<td>CO</td>
<td>2.327608e-003</td>
<td>2.355844e-003</td>
</tr>
<tr>
<td>CO_2</td>
<td>8.382804e-002</td>
<td>1.333077e-001</td>
</tr>
<tr>
<td>HCO</td>
<td>7.389023e-011</td>
<td>7.747780e-011</td>
</tr>
<tr>
<td>CH_2O</td>
<td>8.885994e-013</td>
<td>9.641078e-013</td>
</tr>
<tr>
<td>CH_2OH</td>
<td>1.132705e-018</td>
<td>1.270211e-018</td>
</tr>
<tr>
<td>CH_3O</td>
<td>1.686218e-020</td>
<td>1.890919e-020</td>
</tr>
<tr>
<td>CH_3OH</td>
<td>7.627499e-020</td>
<td>8.831259e-020</td>
</tr>
<tr>
<td>C_2H</td>
<td>0.000000e+000</td>
<td>0.000000e+000</td>
</tr>
<tr>
<td>C_2H_2</td>
<td>0.000000e+000</td>
<td>0.000000e+000</td>
</tr>
<tr>
<td>C_2H_3</td>
<td>0.000000e+000</td>
<td>0.000000e+000</td>
</tr>
<tr>
<td>C_2H_4</td>
<td>0.000000e+000</td>
<td>0.000000e+000</td>
</tr>
<tr>
<td>C_2H_5</td>
<td>0.000000e+000</td>
<td>0.000000e+000</td>
</tr>
</tbody>
</table>

Chapter 3. Chemical Equilibrium
<table>
<thead>
<tr>
<th>Species</th>
<th>Concentration</th>
<th>Uncertainty</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2H6</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>HCCO</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>CH2CO</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>HCCOH</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>N</td>
<td>4.723857e-009</td>
<td>2.390824e-009</td>
</tr>
<tr>
<td>NH</td>
<td>5.297876e-010</td>
<td>2.874303e-010</td>
</tr>
<tr>
<td>NH2</td>
<td>1.590979e-010</td>
<td>9.211144e-011</td>
</tr>
<tr>
<td>NH3</td>
<td>4.043624e-010</td>
<td>2.488376e-010</td>
</tr>
<tr>
<td>NNH</td>
<td>2.194905e-010</td>
<td>2.301699e-010</td>
</tr>
<tr>
<td>NO</td>
<td>3.077021e-003</td>
<td>3.336222e-003</td>
</tr>
<tr>
<td>NO2</td>
<td>1.281909e-006</td>
<td>2.130992e-006</td>
</tr>
<tr>
<td>N2O</td>
<td>1.649231e-007</td>
<td>2.622863e-007</td>
</tr>
<tr>
<td>HNO</td>
<td>2.693730e-008</td>
<td>3.018755e-008</td>
</tr>
<tr>
<td>CN</td>
<td>2.467159e-015</td>
<td>2.319442e-015</td>
</tr>
<tr>
<td>HCN</td>
<td>7.246531e-013</td>
<td>7.076590e-013</td>
</tr>
<tr>
<td>H2CN</td>
<td>8.061030e-020</td>
<td>8.165586e-020</td>
</tr>
<tr>
<td>HCNN</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>HCNO</td>
<td>6.903304e-018</td>
<td>1.073236e-017</td>
</tr>
<tr>
<td>HOCN</td>
<td>1.190410e-013</td>
<td>1.850695e-013</td>
</tr>
<tr>
<td>HCNO</td>
<td>5.138356e-011</td>
<td>7.988446e-011</td>
</tr>
<tr>
<td>NCO</td>
<td>2.247139e-012</td>
<td>3.411713e-012</td>
</tr>
<tr>
<td>N2</td>
<td>7.183390e-001</td>
<td>7.271270e-001</td>
</tr>
<tr>
<td>AR</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>C3H7</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>C3H8</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>CH2CHO</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>CH3CHO</td>
<td>0.0000000e+000</td>
<td>0.0000000e+000</td>
</tr>
</tbody>
</table>
CHAPTER
FOUR

Homogeneous Kinetics

A discussion of homogeneous kinetics models, kinetics managers, etc. will soon replace this box...

4.1 Procedures

getReactionString

Reaction equation string.

Syntax

\[
\text{call getReactionString}(\text{mix}, \ i, \ \text{rxnString})
\]

Arguments

\[
\begin{align*}
\text{mix} & : (\text{Input}) \ \text{Mixture object [type(mixture_t)].} \\
i & : (\text{Input}) \ \text{Reaction index [integer].} \\
\text{rxnString} & : (\text{Output}) \ \text{Reaction equation string [character*(*)].}
\end{align*}
\]

Description

This procedure generates the reaction equation as a character string. The notation of Kee et al. [1989] is used. In particular,

1. The reactants and products are separated by ‘\( <=> \)’ for reversible reactions, and by ‘\( => \)’ for irreversible ones;

2. For pressure-dependent falloff reactions in which any species can act as a third-body collision partner, the participation of the third body is denoted by ‘\( (+ \ M) \)’. If only one species acts as the third body, the species name replaces \( M \), for example, ‘\( (+ \ \text{H}_2\text{O}) \)’.

3. For reactions involving a third body but which are always in the low-pressure limit (rate proportional to third body concentration), the participation of the third body is denote by ‘\( + \ M \)’. 
Example

```fortran
use cantera
character(30) str
type(mixture_t) mix
mix = GRIMech30()
do i = 1,20
call getReactionString(mix,i,str)
write(*,10) i,str
10 format(i2,' ',a)
end do
return
```

Output:

1) \( 2 \text{ O} + \text{ M} \leftrightarrow \text{ O}_2 + \text{ M} \)
2) \( \text{ O} + \text{ H} + \text{ M} \leftrightarrow \text{ OH} + \text{ M} \)
3) \( \text{ O} + \text{ H}_2 \leftrightarrow \text{ H} + \text{ OH} \)
4) \( \text{ O} + \text{ HO}_2 \leftrightarrow \text{ OH} + \text{ O}_2 \)
5) \( \text{ O} + \text{ H}_2\text{O}_2 \leftrightarrow \text{ OH} + \text{ HO}_2 \)
6) \( \text{ O} + \text{ CH} \leftrightarrow \text{ H} + \text{ CO} \)
7) \( \text{ O} + \text{ CH}_2 \leftrightarrow \text{ H} + \text{ HCO} \)
8) \( \text{ O} + \text{ CH}_2(\text{S}) \leftrightarrow \text{ H}_2 + \text{ CO} \)
9) \( \text{ O} + \text{ CH}_2(\text{S}) \leftrightarrow \text{ H} + \text{ HCO} \)
10) \( \text{ O} + \text{ CH}_3 \leftrightarrow \text{ H} + \text{ CH}_2\text{O} \)
11) \( \text{ O} + \text{ CH}_4 \leftrightarrow \text{ OH} + \text{ CH}_3 \)
12) \( \text{ O} + \text{ CO} (+ \text{ M}) \leftrightarrow \text{ CO}_2 (+ \text{ M}) \)
13) \( \text{ O} + \text{ HCO} \leftrightarrow \text{ OH} + \text{ CO} \)
14) \( \text{ O} + \text{ HCO} \leftrightarrow \text{ H} + \text{ CO}_2 \)
15) \( \text{ O} + \text{ CH}_2\text{O} \leftrightarrow \text{ OH} + \text{ HCO} \)
16) \( \text{ O} + \text{ CH}_2\text{OH} \leftrightarrow \text{ OH} + \text{ CH}_2\text{O} \)
17) \( \text{ O} + \text{ CH}_3\text{O} \leftrightarrow \text{ OH} + \text{ CH}_2\text{O} \)
18) \( \text{ O} + \text{ CH}_3\text{OH} \leftrightarrow \text{ OH} + \text{ CH}_2\text{OH} \)
19) \( \text{ O} + \text{ CH}_3\text{OH} \leftrightarrow \text{ OH} + \text{ CH}_3\text{O} \)
20) \( \text{ O} + \text{ C}_2\text{H} \leftrightarrow \text{ CH} + \text{ CO} \)

97) **reactantStoichCoeff**

The stoichiometric coefficient for species \( k \) as a reactant in reaction \( i \).

**Syntax**

```
result = reactantStoichCoeff(mix, k, i)
```

**Arguments**

- **mix** (Input) Mixture object [type(mixture_t)].
\( k \)  
(Input) Species index [integer].

\( i \)  
(Input) Reaction index [integer].

**Result**

Returns a double precision value.

**Description**

A given species may participate in a reaction as a reactant, a product, or both. This function returns the number of molecules of the \( k \)th species appearing on the reactant side of the reaction equation. It does not include any species \( k \) molecules that act as third-body collision partners but are not written in the reaction equation. Non-integral values are allowed.

**Example**

```fortran
use cantera
real(8) rcoeff
type(mixture_t) mix
mix = GRIMech30()
inh = speciesIndex(mix,'NH')
write(*,*) 'Reactions in GRI-Mech 3.0 with NH as a reactant:'
do i = 1, nReactions(mix)
   rcoeff = reactantStoichCoeff(mix,inh,i)
   if (rcoeff .gt. 0.d0) write(*,*) ' reaction ',i
end do
```

**Output:**

Reactions in GRI-Mech 3.0 with NH as a reactant:
- reaction 190
- reaction 191
- reaction 192
- reaction 193
- reaction 194
- reaction 195
- reaction 196
- reaction 197
- reaction 198
- reaction 199
- reaction 280

**See Also**

productStoichCoeff

98) productStoichCoeff

The stoichiometric coefficient for species \( k \) as a product in reaction \( i \).
Syntax

\[
\text{result} = \text{productStoichCoeff}(\text{mix}, k, i)
\]

Arguments

- \textit{mix} (Input) Mixture object [\text{type(mixture\_t)}].
- \textit{k} (Input) Species index [integer].
- \textit{i} (Input) Reaction index [integer].

Result

Returns a double precision value.

Description

A given species may participate in a reaction as a reactant, a product, or both. This function returns the number of molecules of the \(k^{th}\) species appearing on the product side of the reaction equation. It does not include any species \(k\) molecules that act as third-body collision partners but are not written in the reaction equation. Non-integral values are allowed.

Example

\begin{verbatim}
use cantera
data real(8) pcoeff
type(mixture_t) mix
mix = GRIMech30()
inh = speciesIndex(mix,'NH')
write(*,*) 'Reactions in GRI-Mech 3.0 with NH as a product:'
do i = 1,nReactions(mix)
   pcoeff = productStoichCoeff(mix,inh,i)
   if (pcoeff .gt. 0.d0) write(*,*) ' reaction ',i
end do
\end{verbatim}

Output:

Reactions in GRI-Mech 3.0 with NH as a product:
reaction 200
reaction 202
reaction 203
reaction 208
reaction 223
reaction 232
reaction 242
reaction 243
reaction 262
reaction 269

\textbf{99) netStoichCoeff}

The net stoichiometric coefficient for species \(k\) in reaction \(i\).
Syntax
result = netStoichCoeff(mix, k, i)

Arguments

mix (Input) Mixture object [type(mixture_t)].
k (Input) Species index [integer].
i (Input) Reaction index [integer].

Result
Returns a double precision value.

Description
This function returns the difference between the product and reactant stoichiometric coefficients.

100) getFwdRatesOfProgress

Get the forward rates of progress $Q_i^{(f)}$ for the reactions [kmol/m$^3$/s].

Syntax
call getFwdRatesOfProgress(mix, fwdrop)

Arguments

mix (Input) Mixture object [type(mixture_t)].
fwdrop (Output) Forward rates of progress [double precision array]. Must be dimensioned at least as large as the number of reactions in mix.

Description
The expressions for the reaction rates of progress are determined by the kinetics model used. See section ??.

Example see the example for procedure ??.

101) getRevRatesOfProgress

Get the reverse rates of progress $Q_i^{(r)}$ for the reactions [kmol/m$^3$/s].

Syntax
call getRevRatesOfProgress(mix, revrop)

Arguments

mix (Input) Mixture object [type(mixture_t)].
revrop (Output) Reverse rates of progress [double precision array]. Must be dimensioned at least as large as the number of reactions in mix.

Description
The reverse rate of progress is non-zero only for reversible reactions. The expressions for the reaction rates of progress are determined by the kinetics model used. See section ??.

Example see the example for procedure ??.

102) getNetRatesOfProgress

Get the net rates of progress for the reactions [kmol/m$^3$/s].

Syntax
```
call getNetRatesOfProgress(mix, netrop)
```

Arguments
- `mix` (Input) Mixture object [type(mixture_t)].
- `netrop` (Output) Net rates of progress [double precision array]. Must be dimensioned at least as large as the number of reactions in mix.

Description
The net rates of progress are the difference between the forward and the reverse rates:

$$Q_i = Q_i^{(f)} - Q_i^{(r)}$$

(4.1)

The expressions for the rates of progress are determined by the kinetics model used. See section ??.

In the example below, a mixture is first created containing hydrogen, oxygen, and argon, and then set to the chemical equilibrium state at 2500 K and 1 atm. The temperature is then lowered, and the forward, reverse, and net rates of progress are computed for all reactions, and those above a threshold are printed. (If the temperature had not been changed, then the forward and reverse rates of progress would be equal for all reversible reactions).getFwdRatesOfProgress, getRevRatesOfProgress

Example
```
use cantera
real(8)  frop(500),  rrop(500),  netrop(500)
type(mixture_t)  mix
character*40  rxn
mix = GRIMech30()
call setState_TPX(mix,2500.d0,OneAtm,‘H2:2,O2:1.5,AR:8’)
call equilibrate(mix,’TP’)
call setState_TP(mix, 2000.d0,OneAtm)
call getFwdRatesOfProgress(mix,frop)
call getRevRatesOfProgress(mix,rrop)
call getNetRatesOfProgress(mix,netrop)
```
do i = 1, nReactions(mix)
    if (abs(frop(i)) .gt. 1.d-10 .or.
        abs(rrop(i)) .gt. 1.d-10) then
        call getReactionString(mix,i,rxn)
        write(*,20) rxn, frop(i), rrop(i), netrop(i)
    end if
end do

Output:

<table>
<thead>
<tr>
<th>Reaction</th>
<th>K1</th>
<th>K2</th>
<th>K3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 O + M &lt;=&gt; O2 + M</td>
<td>0.44094E-03</td>
<td>0.94026E-06</td>
<td>0.44000E-03</td>
</tr>
<tr>
<td>O + H + M &lt;=&gt; OH + M</td>
<td>0.51156E-03</td>
<td>0.23776E-05</td>
<td>0.50819E-03</td>
</tr>
<tr>
<td>O + H2 &lt;=&gt; H + OH</td>
<td>0.36968E+01</td>
<td>0.41079E-03</td>
<td>0.12904E-01</td>
</tr>
<tr>
<td>O + H2O2 &lt;=&gt; OH + H2O</td>
<td>0.13797E-01</td>
<td>0.41079E-03</td>
<td>0.12904E-01</td>
</tr>
<tr>
<td>H + O2 + M &lt;=&gt; HO2 + M</td>
<td>0.16806E-02</td>
<td>0.12072E-03</td>
<td>0.15599E-02</td>
</tr>
<tr>
<td>H + 2 O2 &lt;=&gt; HO2 + O2</td>
<td>0.13767E-02</td>
<td>0.98895E-04</td>
<td>0.12779E-02</td>
</tr>
<tr>
<td>H + O2 + H2O &lt;=&gt; HO2 + H2O</td>
<td>0.11154E+00</td>
<td>0.80119E-02</td>
<td>0.10352E+00</td>
</tr>
<tr>
<td>H + O2 + AR &lt;=&gt; HO2 + AR</td>
<td>0.21897E-01</td>
<td>0.15729E-02</td>
<td>0.20324E-01</td>
</tr>
<tr>
<td>H + O2 &lt;=&gt; O + OH</td>
<td>0.66073E+01</td>
<td>0.14401E+02</td>
<td>0.77937E+01</td>
</tr>
<tr>
<td>2 H + M &lt;=&gt; H2 + M</td>
<td>0.19043E-03</td>
<td>0.79650E-06</td>
<td>0.18964E-03</td>
</tr>
<tr>
<td>2 H + H2 &lt;=&gt; 2 H2</td>
<td>0.32781E-05</td>
<td>0.13711E-07</td>
<td>0.32644E-05</td>
</tr>
<tr>
<td>2 H + H2O &lt;=&gt; H2 + H2O</td>
<td>0.55931E-03</td>
<td>0.23394E-05</td>
<td>0.55697E-03</td>
</tr>
<tr>
<td>H + OH + M &lt;=&gt; H2O + M</td>
<td>0.28712E-01</td>
<td>0.57896E-02</td>
<td>0.28654E-01</td>
</tr>
<tr>
<td>H + HO2 &lt;=&gt; O + H2O</td>
<td>0.13263E+00</td>
<td>0.81148E-04</td>
<td>0.12452E+00</td>
</tr>
<tr>
<td>H + HO2 &lt;=&gt; O2 + H2</td>
<td>0.13544E-01</td>
<td>0.78862E-03</td>
<td>0.12755E-01</td>
</tr>
<tr>
<td>H + H2O2 &lt;=&gt; OH + H2O</td>
<td>0.28319E-01</td>
<td>0.39935E-02</td>
<td>0.24325E-01</td>
</tr>
<tr>
<td>H + H2O2 &lt;=&gt; HO2 + H2</td>
<td>0.22868E-03</td>
<td>0.89198E-04</td>
<td>0.13948E-03</td>
</tr>
<tr>
<td>H + H2O2 &lt;=&gt; OH + H2O</td>
<td>0.70669E-04</td>
<td>0.18741E-05</td>
<td>0.68795E-04</td>
</tr>
<tr>
<td>OH + H2 &lt;=&gt; H + H2O</td>
<td>0.21063E+02</td>
<td>0.10908E+02</td>
<td>0.10352E+00</td>
</tr>
<tr>
<td>2 OH (+ M) &lt;=&gt; H2O2 (+ M)</td>
<td>0.15268E+00</td>
<td>0.11609E-01</td>
<td>0.14107E+00</td>
</tr>
<tr>
<td>2 OH &lt;=&gt; O + H2O</td>
<td>0.32276E+02</td>
<td>0.14003E+02</td>
<td>0.18273E+02</td>
</tr>
<tr>
<td>OH + H2O2 &lt;=&gt; O2 + H2O</td>
<td>0.23253E-03</td>
<td>0.43728E-04</td>
<td>0.18881E-03</td>
</tr>
<tr>
<td>OH + H2O2 &lt;=&gt; HO2 + H2O</td>
<td>0.13444E+00</td>
<td>0.25281E-01</td>
<td>0.10916E+00</td>
</tr>
<tr>
<td>2 HO2 &lt;=&gt; O2 + H2O2</td>
<td>0.18695E+00</td>
<td>0.52480E-02</td>
<td>0.18170E+00</td>
</tr>
<tr>
<td>2 HO2 &lt;=&gt; O2 + H2O2</td>
<td>0.26438E-06</td>
<td>0.39466E-07</td>
<td>0.22492E-06</td>
</tr>
</tbody>
</table>

Get the equilibrium constants of the reactions in concentration units \([\text{kmol/m}^3]^{\Delta n}\), where \(\Delta n\) is the net change in mole numbers in going from reactamts to products.  

4.1. Procedures
Syntax
   call getEquilibriumConstants(mix, kc)

Arguments
   mix     (Input) Mixture object [type(mixture_t)].
   kc      (Output) Equilibrium constants [double precision array]. Must be dimensioned at least as large as the number of species in mix.

104) getCreationRates

Get the species chemical creation rates [kmol/m³/s].

Syntax
   call getCreationRates(mix, cdot)

Arguments
   mix     (Input) Mixture object [type(mixture_t)].
   cdot    (Output) Creation rates [double precision array]. Must be dimensioned at least as large as the number of species in mix.

Description
   The creation rate of species $k$ is
   $$\dot{C}_k = \sum_i \left( \nu_{ki}^{(p)} Q_{f,i} + \nu_{ki}^{(r)} Q_{r,i} \right)$$  
   (4.2)

Example  see the example for procedure 106.

105) getDestructionRates

Get the species chemical destruction rates [kmol/m³/s].

Syntax
   call getDestructionRates(mix, ddot)

Arguments
   mix     (Input) Mixture object [type(mixture_t)].
   ddot    (Output) Destruction rates [double precision array]. Must be dimensioned at least as large as the number of species in mix.
Description
The destruction rate of species $k$ is

$$\dot{D}_k = \sum_i \left( \nu_{ki}^r Q_{f,i} + \nu_{ki}^p Q_{r,i} \right)$$  \hspace{1cm} (4.3)$$

Example see the example for procedure 106.

106) getNetProductionRates

Get the species net chemical production rates [kmol/m$^3$/s].

Syntax
\[
\text{call getNetProductionRates}(\text{mix, wdot})
\]

Arguments

- \(\text{mix}\) (Input) Mixture object [type(mixture)].
- \(\text{wdot}\) (Output) Net production rates [double precision array]. Must be dimensioned at least as large as the number of species in \(\text{mix}\).

Description
This procedure returns the net production rates for all species, which is the difference between the species creation and destruction rates:

$$\dot{\omega}_k = \dot{C}_k - \dot{D}_k,$$

$$= \sum_i \nu_{ki} Q_i$$  \hspace{1cm} (4.5)$$

If only the net rates are required, it is usually more efficient to call this procedure than to first compute the creation and destruction rates separately and take the difference.

In the example below, the creation, destruction, and net production rates are computed for the same conditions as in the example for procedure 106. Also shown is the characteristic chemical time, defined as the concentration divided by the destruction rate.

Example

\[
\text{use cantera}
\]
\[
\text{real(8) cdot(100), ddot(100), wdot(100), conc(100)}
\]
\[
\text{type(mixture_t) mix}
\]
\[
\text{character*10 names(100)}
\]
\[
\text{mix = GRIMech30()}
\]
\[
\text{call setState_TPX(mix,2500.d0,OneAtm,'H2:2,O2:1.5,AR:8')}
\]
\[
\text{call equilibrate(mix,'TP')}
\]
\[
\text{call setState_TP(mix, 2000.d0,OneAtm)}
\]
\[
\text{call getCreationRates(mix,cdot)}
\]
\[
\text{call getDestructionRates(mix,ddot)}
\]

4.1. Procedures
call getNetProductionRates(mix, wdot)
call getSpeciesNames(mix, names)
call getConcentrations(mix, conc)
do k = 1, nSpecies(mix)
   if (wdot(k) .gt. 1.d-10) then
      write(*, 20) names(k), cdot(k), ddot(k), wdot(k),
      $ conc(k)/ddot(k)
   end if
end do

Output:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0.39176E+02</td>
<td>0.21081E+02</td>
<td>0.18095E+02</td>
<td>0.51077E-06</td>
</tr>
<tr>
<td>O</td>
<td>0.42994E+02</td>
<td>0.32117E+02</td>
<td>0.10877E+02</td>
<td>0.58471E-06</td>
</tr>
<tr>
<td>O2</td>
<td>0.14675E+02</td>
<td>0.67536E+01</td>
<td>0.79217E+01</td>
<td>0.40874E-04</td>
</tr>
<tr>
<td>H2O</td>
<td>0.53859E+02</td>
<td>0.24310E+02</td>
<td>0.29549E+02</td>
<td>0.44237E-04</td>
</tr>
<tr>
<td>H2O2</td>
<td>0.17831E+00</td>
<td>0.14701E+00</td>
<td>0.31294E-01</td>
<td>0.11045E-07</td>
</tr>
</tbody>
</table>

**Example 4.1**

In the example below, getNetProductionRates is used to evaluate the right-hand side of the species equation for a constant-volume kinetics simulation.

\[
\frac{dY_k}{dt} = \omega_k M_k / \rho.
\]

type (mixture_t) :: mix
parameter (KMAX = 100)
double precision wdot(KMAX)
double precision wt(KMAX)
...
call getNetProductionRates(mix, wdot)
call getMolecularWeights(mix, wt)
rho = density(mix)
do k = 1, nSpecies(mix)
   ydot(k) = wdot(k) * wt(k) / rho
end do
CHAPTER FIVE

Transport Properties

A discussion of the theory of transport properties and the transport models implemented in Cantera will soon replace this box...

5.1 Derived Types

**transport_t**

Derived type for transport managers.

- **hdl**: integer. Handle encoding pointer to the C++ object.
- **iok**: integer. Internal flag used to signify whether or not the object has been properly constructed.

5.2 Procedures

107) **MultiTransport**

Construct a new multicomponent transport manager.

**Syntax**

object = MultiTransport(mix, database, logLevel)

**Arguments**

- **mix**: (Input / Output) Mixture object whose transport properties the transport manager will compute. [type(mixture_t)].
- **database**: (Input) Transport database file. [character*(*)].
- **logLevel**: (Input) A log file in XML format ‘transport_log.xml’ is generated during construction of the transport manager, containing polynomial fit coefficients, etc. The value of logLevel (0 - 4) determines how much detail to write to the log file. [integer].
Result

Returns an object of type `transport_t`.

Description

This constructor creates a new transport manager that implements a multicomponent transport model for ideal gas mixtures. The model is based on that of? and Kee et al. [1986]. See Section ** for more details.

Example

```plaintext
type(mixture_t) mix
type(transport_t) tr
mix = GRIMech30()
tr = MultiTransport(mix, 'gri30_tran.dat', 0)
```

108) MixTransport

Construct a new transport manager that uses a mixture-averaged formulation.

Syntax

`object = MixTransport(mix, database, logLevel)`

Arguments

- `mix` (Input / Output) Mixture object whose transport properties the transport manager will compute. [type(mixture_t)].
- `database` (Input) Transport database file. [character*(*)].
- `logLevel` (Input) A log file in XML format ‘transport_log.xml’ is generated during construction of the transport manager, containing polynomial fit coefficients, etc. The value of `logLevel` (0 - 4) determines how much detail to write to the log file. [integer].

Result

Returns an object of type `transport_t`.

Description

This constructor creates a new transport manager that implements a 'mixture-averaged' transport model for ideal gas mixtures. The model is very similar to the mixture-averaged model described by Kee et al. [1986]. In general, the results of this model are less accurate than those of `MultiTransport`, but are far less expensive to compute.

This manager does not implement expressions for thermal diffusion coefficients. If thermal diffusion is important in a problem, use the `MultiTransport` transport manager.

Example

```plaintext
type(mixture_t) mix
type(transport_t) tr
mix = GRIMech30()
tr = MixTransport(mix, 'gri30_tran.dat', 0)
```
Set the transport manager to one that was previously created by a call to `initTransport`.

**Syntax**

```call setTransport(mix, transportMgr)```

**Arguments**

- `mix` (Input / Output) Mixture object [type(mixture_t)].
- `transportMgr` (Input / Output) Transport manager [type(transport_t)].

**Description**

Transport managers may be swapped in or out of a mixture object. This capability makes it possible, for example, to use a multicomponent transport formulation in flow regions where gradients are large, and switch to a much less expensive model for other regions.

This procedure re-installs a transport manager previously created with `initTransport`, and saved with `transport`.

Note that transport managers cannot be shared between mixture objects.

**Example**

```use cantera
type(mixture_t) mix
type(transport_t) tr1, tr2
mix = GRIMech30()
call initTransport(mix, Multicomponent, 'gri30_tran.dat', 0)
tr1 = transport(mix)
write(*,*) thermalConductivity(mix)
call initTransport(mix, CK_Multicomponent, 'gri30_tran.dat', 0)
tr2 = transport(mix)
write(*,*) thermalConductivity(mix)
call setTransport(mix, tr1)
write(*,*) thermalConductivity(mix)
return```

**Output:**

```
0.186896997850329
0.186818124230130
0.186896997850329
```

**See Also**

transport, initTransport
Delete a transport manager

**Syntax**
```
call delete(transportMgr)
```

**Arguments**
- `transportMgr` (Input / Output) Transport manager object [type(transport_t)].

**Description**
Deleting a transport manager releases the memory associated with its kernel object.

Return the transport manager object.

**Syntax**
```
result = transportMgr(mix)
```

**Arguments**
- `mix` (Input) Mixture object [type(mixture_t)].

**Result**
Returns a `type(transport_t)` value.

**Description**
This function returns a reference to the currently-installed transport manager object. It may be used to store a transport manager before installing a different one.

**Example** see the example for procedure ??.

The dynamic viscosity [Pa-s].

**Syntax**
```
result = viscosity(mix)
```

**Arguments**
- `mix` (Input) Mixture object [type(mixture_t)].
Result

- Returns a double precision value.

Description

The stress tensor in a Newtonian fluid is given by the expression

\[ \tau_{ij} = -P \delta_{ij} + \mu \left( \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right) + \delta_{ij} \lambda_b \text{div} v. \]  

(5.1)

where \( \mu \) is the viscosity and \( \lambda_b \) is the bulk viscosity. This function returns the value of \( \mu \) computed by the currently-installed transport manager.

113) getSpeciesViscosities

Get the pure species viscosities [Pa-s].

Syntax

call getSpeciesViscosities(mix, visc)

Arguments

- mix
  (Input) Mixture object [type(mixture_t)].
- visc
  (Input / Output) Array of pure species viscosities [double precision array]. Must be dimensioned at least as large as the number of species in mix.

Description

A common approach to computing the viscosity of a mixture is to use a mixture rule to generate a weighted average of the viscosities of the pure species. This procedure returns the viscosities of the pure species that are used in the mixture rule. Transport managers that do not compute viscosity using a mixture rule may not implement this procedure.

114) getSpeciesFluxes

Compute the species net mass fluxes due to diffusion.

Syntax

call getSpeciesFluxes(mix, ndim, grad_X, grad_T, fluxes)

Arguments

- mix
  (Input) Mixture object [type(mixture_t)].
- ndim
  (Input) Dimensionality (1, 2, or 3) [integer].
grad_\textit{X} (Input / Output) Array of species mole fraction gradients. The \((k,n)\) entry is the gradient of species \(k\) in coordinate direction \(n\). [double precision].

\textit{grad_\textit{T}} (Input) Temperature gradient array. By default, thermal diffusion is included, using the temperature gradient specified here. Set to zero to compute the result without thermal diffusion. [double precision].

\textit{fluxes} (Output) Array of species diffusive mass fluxes [double precision].

\textbf{Description}

This procedure returns the diffusive mass fluxes for all species, given the local gradients in mole fraction and temperature. The diffusive mass flux vector for species \(k\) is given by

\[
j_k = \rho V_k Y_k,
\]

(5.2)

where \(V_k\) is the diffusion velocity of species \(k\). The diffusive mass flux satisfies

\[
\sum_k j_k = 0.
\]

(5.3)

The inputs to this procedure are a mixture object, which defines the local thermodynamic state, the array of mole fraction \textit{gradients} with elements

\[
\text{grad}_X(k,n) = \left( \frac{\partial X_k}{\partial x_n} \right)
\]

(5.4)

and the one-dimensional temperature gradient array

\[
\text{grad}_T(n) = \left( \frac{\partial T}{\partial x_n} \right).
\]

(5.5)

Thermal diffusion is included if any component of the temperature gradient is non-zero.

There are several advantages to calling this procedure rather than evaluating the transport properties and then forming the flux expression in your application program. These are:

\textbf{Performance.} For multicomponent transport models, this procedure uses an algorithm that does not require evaluating the multicomponent diffusion coefficient matrix first, and thereby eliminates the need for matrix inversion, resulting in somewhat better performance.

\textbf{Generality.} The expression for the flux in terms of mole fraction and temperature gradients is not the same for all transport models. Using this procedure allows switching easily between multicomponent and Fickian diffusion models.

\textbf{Example}

\begin{verbatim}
use cantera
implicit double precision (a-h,o-z)
type(mixture_t) mix
character*80 infile, thermofile, xstring, model, trdb
double precision gradx(10,3), gradt(3), fluxes(10,3)
double precision x1(8), x2(8), x3(8)
character*20 name(10)
data x1/8*0.1d0/
\end{verbatim}
data x2/0.d0, 0.2d0, 0.d0, 0.2d0, 0.d0, 0.2d0, 0.d0,
$ 0.2d0/
data x3/0.1d0, 0.2d0, 0.3d0, 0.4d0, 0.d0, 0.d0, 0.d0,
$ 0.d0/
mix = CKGas('air.inp','therm.dat')
call getSpeciesNames(mix, name)
call initTransport(mix, Multicomponent,'gri30_tran.dat', 0)
nspec = nSpecies(mix)

! generate dummy gradient data

do k = 1, nsp
  gradx(k,1) = x1(k) - x2(k) ! dX(k)/dx1
  gradx(k,2) = x2(k) - x3(k) ! dX(k)/dx2
  gradx(k,3) = x3(k) - x1(k) ! dX(k)/dx3
  x1(k) = 0.5*(x1(k) + x2(k))
end do

gradt(1) = 1000.d0 ! dT/dx1
gradt(2) = -500.d0 ! dT/dx2
gradt(3) = 800.d0 ! dT/dx3

call setState_TPX(mix, 1800.d0, OneAtm, x1)
call getSpeciesFluxes(mix, 3, gradx, gradt, fluxes)
do n = 1,3
  write(*,10) n,n,gradt(n),n
10 format(//'coordinate direction ',i1,':'/'dT/dx',i1,' = ',g13.5,
      '/species ',15x,' dX/dx',i1,' mass flux')
do k = 1,nsp
  write(*,20) name(k),gradx(k,n),fluxes(k,n)
20 format(a,2e14.5)
end do
end do

Output:
coordinate direction 1:
 dT/dx1 = 1000.0

species          dX/dx1          mass flux
O        0.10000E+00 -0.72034E-05
O2       -0.10000E+00  0.88312E-05
N        0.10000E+00 -0.56038E-05
NO       -0.10000E+00  0.81268E-05
NO2      0.10000E+00 -0.14005E-04
N2O      -0.10000E+00  0.86180E-05
N2       0.10000E+00 -0.90551E-05
AR       -0.10000E+00  0.10291E-04
coordinate direction 2:
\[ \frac{dT}{dx_2} = -500.00 \]
<table>
<thead>
<tr>
<th>species</th>
<th>( \frac{dX}{dx_2} )</th>
<th>mass flux</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>-0.10000E+00</td>
<td>-0.67305E-05</td>
</tr>
<tr>
<td>O_2</td>
<td>0.00000E+00</td>
<td>0.21224E-05</td>
</tr>
<tr>
<td>N</td>
<td>-0.30000E+00</td>
<td>0.18372E-04</td>
</tr>
<tr>
<td>NO</td>
<td>-0.20000E+00</td>
<td>0.19739E-04</td>
</tr>
<tr>
<td>NO_2</td>
<td>0.00000E+00</td>
<td>0.32196E-05</td>
</tr>
<tr>
<td>N_2O</td>
<td>0.20000E+00</td>
<td>-0.17896E-04</td>
</tr>
<tr>
<td>N_2</td>
<td>0.00000E+00</td>
<td>0.18281E-05</td>
</tr>
<tr>
<td>AR</td>
<td>0.20000E+00</td>
<td>-0.20653E-04</td>
</tr>
</tbody>
</table>

coordinate direction 3:
\[ \frac{dT}{dx_3} = 800.00 \]
<table>
<thead>
<tr>
<th>species</th>
<th>( \frac{dX}{dx_3} )</th>
<th>mass flux</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>0.00000E+00</td>
<td>0.15068E-04</td>
</tr>
<tr>
<td>O_2</td>
<td>0.10000E+00</td>
<td>-0.10916E-04</td>
</tr>
<tr>
<td>N</td>
<td>0.20000E+00</td>
<td>-0.11766E-04</td>
</tr>
<tr>
<td>NO</td>
<td>0.30000E+00</td>
<td>-0.27733E-04</td>
</tr>
<tr>
<td>NO_2</td>
<td>-0.10000E+00</td>
<td>0.97117E-05</td>
</tr>
<tr>
<td>N_2O</td>
<td>-0.10000E+00</td>
<td>0.82877E-05</td>
</tr>
<tr>
<td>N_2</td>
<td>-0.10000E+00</td>
<td>0.74848E-05</td>
</tr>
<tr>
<td>AR</td>
<td>-0.10000E+00</td>
<td>0.98637E-05</td>
</tr>
</tbody>
</table>

115) getMultiDiffCoeffs

Get the multicomponent diffusion coefficients \([\text{n}^2/\text{s}]\). This procedure may not be implemented by all transport managers. If not, the `multiDiff` array will be left unchanged, and `ierr` will be set to a negative number.

**Syntax**
```
call getMultiDiffCoeffs(mix, ldim, multiDiff)
```

**Arguments**
- `mix`  (Input) Mixture object [type(mixture_t)].
- `ldim` (Input) Leading dimension of `multiDiff`. Must be at least as large as the number of species in `mix` [integer].
- `multiDiff` (Input) Two-dimensional array of multicomponent diffusion coefficients [double precision array]. Must be dimensioned at least as large as the number of species in `mix`. 

Chapter 5. Transport Properties
116) \textbf{thermalConductivity}

Thermal conductivity [W/m/K].

\textbf{Syntax}

\begin{verbatim}
result = thermalConductivity(mix)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \textit{mix} (Input) Mixture object [type(mixture_t)].
\end{itemize}

\textbf{Result}

Returns a double precision value.

117) \textbf{getThermalDiffCoeffs}

Get the array of thermal diffusion coefficients.

\textbf{Syntax}

\begin{verbatim}
call getThermalDiffCoeffs(mix, dt)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \textit{mix} (Input) Mixture object [type(mixture_t)].
  \item \textit{dt} (Output) Thermal diffusion coefficients [double precision array]. Must be dimensioned at least as large as the number of species in \textit{mix}.
\end{itemize}

118) \textbf{setOptions}

Set transport options.

\textbf{Syntax}

\begin{verbatim}
call setOptions(linearSolver, GMRES_m, GMRES_eps)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \textit{linearSolver} (Input) Linear algebra solver to use. Valid values are 'LU' (LU decomposition), and 'GMRES' (GMRES iterative solver). This setting only affects those transport managers that solve systems of linear equations in order to compute the transport properties. Specifying GMRES may significantly accelerate transport property evaluation in some cases. However, if transport properties are evaluated within functions for which numerical derivatives are computed, then using any iterative method may cause convergence difficulties. In many cases, transport property evaluation can be done outside the function. If not, then LU decomposition should be used. Default: LU decomposition. [character*].
\end{itemize}
**GMRES**\_\_m  (Input) GMRES ‘m’ parameter. Number of ‘inner’ iterations between ‘outer’ iterations. Default: 100 [integer].

**GMRES**\_\_eps (Input) GMRES convergence parameter. Default: $10^{-4}$ [double precision].

**Description**

This procedure sets various transport options. All arguments but the first one are optional. The recommended way to call this procedure is using keywords, as shown in the example.

**Example**

```plaintext
use cantera
transport_t tr
...
call setOptions(tr, linearSolver = ‘GMRES’)  ! set any one or more

call setOptions(tr, GMRES\_eps = 1.e-3, GMRES\_m = 50)
```
6.1 Reactor Objects

6.1.1 Type cstr_t

Objects representing zero-dimensional, stirred reactors are implemented in Fortran 90 by the derived type cstr_t, defined below.

**cstr_t**
- Stirred reactors.
  - **hdl** integer . Handle encoding pointer to the C++ object
  - **mix** type(mixture_t) . Mixture object

6.2 Procedures

6.3 Constructors

119) StirredReactor

Construct a stirred reactor for zero-dimensional kinetics simulations.

**Syntax**

object = StirredReactor()

**Result**

Returns an object of type cstr_t.
Description

Objects created using **StirredReactor** represent generic stirred reactors. Depending on the components attached or the values set for options, it may represent a closed batch reactor, a reactor with a steady flow through it, or one reactor in a network.

The reactor object internally integrates rate equations for volume, species, and energy in primitive form, with no assumption regarding the equation of state. It uses the CVODE stiff ODE integrator, and can handle typical large reaction mechanisms.

A reactor object can be viewed as a cylinder with a piston. Currently, you can set a time constant of the piston motion. By choosing an appropriate value, you can carry out constant volume simulations, constant pressure ones, or anything in between.

The reactor also has a heat transfer coefficient that can be set. A value of zero (default) results in an adiabatic simulation, while a very large value produces an isothermal simulation.

In C++, more general control of the volume and heat transfer coefficient vs. time is possible, allowing construction of engine simulators, for example. (Surface chemistry can be included also.) This will soon be added to the Fortran interface.

The reactor may have an arbitrary number of inlets and outlets, each of which may be connected to a "flow device" such as a mass flow controller, a pressure regulator, etc. Additional reactors may be connected to the other end of the flow device, allowing construction of arbitrary reactor networks.

If an object constructed by **StirredReactor()** is used with default options, it will simulate an adiabatic, constant volume reactor with gas-phase chemistry but no surface chemistry.

More documentation coming soon.

Example

```c
use cantera
type(cstr_t) reactr
reactr = StirredReactor()
```

Reservoir

A reservoir is like a reactor, except that the state never changes after being initialized. No chemistry occurs, and the temperature, pressure, composition, and all other properties of the contents remain at their initial values unless explicitly changed. These are designed to be connected to reactors to provide specified inlet conditions.

Syntax

```
object = Reservoir()
```

Result

Returns an object of type **cstr_t**.
Example

```c
use cantera
type(cstr_t) reacr
type(cstr_t) upstr
type(flowdev_t) mfc
! install a reservoir upstream from the reactor,
! and connect them through a mass flow controller
reactr = StirredReactor()
upstr = Reservoir()
mfc = MassFlowController()
call install(mfc, upstr, reactr)
```

6.4 Assignment

121) *copy*

Copy one reactor object to another. The contents of `dest` are overwritten.

**Syntax**

```c
call copy(src, dest)
```

**Arguments**

- `src` (Input) Reactor object [type(mixture_t)].
- `dest` (Output) Reactor object [type(mixture_t)].

**Description**

This procedure performs a shallow copy – the handle is copied, but not the object it points to.

6.5 Setting Options

These procedures set various optional reactor characteristics.

122) *setInitialVolume*

Set the initial reactor volume [m$^3$]. By default, the initial volume is 1.0 m$^3$.

**Syntax**

```c
call setInitialVolume(reac, vol)
```

**Arguments**

- `reac` (Input) Reactor object.
- `vol` (Input) Initial volume [m$^3$].
**setInitialTime**

Set the initial time [s]. Default = 0.0 s.

**Syntax**

```lua
call setInitialTime(reac, time)
```

**Arguments**

- `reac` (Input / Output) Reactor object. [type(cstr_t)].
- `time` (Input) Initial time [double precision].

**Description**

Restarts integration from this time using the current substance state as the initial condition. Note that this routine does no interpolation. It simply sets the clock to the specified time, takes the current state to be the value for that time, and restarts.

**Example**

```lua
use cantera
type(cstr_t) r
r = StirredReactor()
call setInitialTime(r, 0.02)
```

---

**setMaxStep**

Set the maximum step size for integration.

**Syntax**

```lua
call setMaxStep(reac, maxstep)
```

**Arguments**

- `reac` (Input / Output) Reactor object. [type(cstr_t)].
maxstep  (Input) Maximum step size [double precision].

Description
Note that on the first call to 'advance,' if the maximum step size has not been set, the maximum step size is set to not exceed the specified end time. This effectively keeps the integrator from integrating out to a large time and then interpolating back to the desired final time. This is done to avoid difficulties when simulating problems with sudden changes, like ignition problems. If used, it must be called before calling advance for the first time.

Example
use cantera
type(cstr_t)  r
  r = StirredReactor()
callsetMaxStep(r,1.d-3)

setArea

Set the reactor surface area [m²]. Can be changed at any time.

Syntax
call setArea(reac, area)

Arguments
  reac  (Input / Output) Reactor object [type(cstr_t)].
  area  (Input) Area [double precision].

Description
Note that this does not update automatically if the volume changes, but may be changed manually at any time. The area affects the total heat loss rate.

Example
use cantera
type(cstr_t)  r
  r = StirredReactor()
! a spherical reactor
  radius = 2.0
  vol = (4.d0/3.d0)*Pi*radius**3
  ar = 4.d0*Pi*radius**2
call setInitialVolume(r, vol)
call setArea(r, ar)
126) setExtTemp

Set the external temperature $T_0$ used for heat loss calculations. The heat loss rate is calculated from

$$\dot{Q}_{out} = hA(T - T_0) + \epsilon A(T^4 - T_{0,R}^4).$$  \hspace{1cm} (6.1)

See also setArea, setEmissivity, setExtRadTemp. Can be changed at any time.

**Syntax**

```call setExtTemp(reac, ts)```

**Arguments**

- **reac** (Input / Output) Reactor object [type(cstr_t)].
- **ts** (Input) External temperature [double precision].

**Example**

```use cantera
type(cstr_t) r
r = StirredReactor()
call setExtTemp(r, 500.d0)```

127) setExtRadTemp

Set the external temperature for radiation. By default, this is the same as the temperature set by setExtTemp. But if setExtRadTemp is called, then subsequent calls to setExtTemp do not modify the value set here. Can be changed at any time. See Eq. (6.1)

**Syntax**

```call setExtRadTemp(reac, trad)```

**Arguments**

- **reac** (Input / Output) Reactor object [type(cstr_t)].
- **trad** (Input) External temperature for radiation [double precision].

**Example**

```use cantera
type(cstr_t) jet
jet = StirredReactor()
call setExtTemp(jet, 1000.d0)
call setExtTemp(jet, 300.d0)
call setExtRadTemp(jet, 300.d0)```
128) setHeatTransferCoeff

Set the heat transfer coefficient \([\text{W/m}^2/\text{K}]\). Default: 0.d0 See Eq. (6.1). May be changed at any time.

Syntax

call setHeatTransferCoeff(react, h)

Arguments

- `react` (Input / Output) Reactor object \([\text{type(cstr\_t)}]\).
- `h` (Input) Heat transfer coefficient \([\text{double precision}]\).

Example

```java
use cantera
type(cstr\_t) r
r = StirredReactor()
call setHeatTransferCoeff(r,10.d0)
```

129) setVDotCoeff

The equation used to compute the rate at which the reactor volume changes in response to a pressure difference is

\[
\dot{V} = K \left( \frac{P - P_{\text{ext}}}{P_{\text{ext}}} \right) V_i
\]

where \(V_i\) is the initial volume. The inclusion of \(V_i\) in this expression is done only so that \(K\) will have units of \(s^{-1}\). May be changed at any time.

Syntax

call setVDotCoeff(react, k)

Arguments

- `react` (Input / Output) Reactor object \([\text{type(cstr\_t)}]\).
- `k` (Input) Coefficient determining rate at which volume changes in response to a pressure difference \([\text{double precision}]\).

Description

The default value is zero, which results in the volume being held constant. To conduct a constant pressure simulation, set \(P_{\text{ext}}\) to the initial pressure, and set \(K\) faster than characteristic rates for your problem. It is always a good idea to examine the constancy of the pressure in the output.

Example

```java
use cantera
type(cstr\_t) r
r = StirredReactor()
call setVDotCoeff(r,1.d8)
```
130) setEmissivity

Set the emissivity. May be changed at any time. See Eq. (6.1). May be changed at any time.

Syntax

```plaintext
call setEmissivity(react, emis)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>react</td>
<td>(Input / Output) Reactor object [type(cstr_t)].</td>
</tr>
<tr>
<td>emis</td>
<td>(Input) Emissivity [double precision].</td>
</tr>
</tbody>
</table>

Description

The treatment of radiation here is approximate at best. The emissivity is defined here such that the net radiative power loss from the gas to the walls is \( \epsilon A \sigma (T^4 - T_{ext}^4) \). Note that this is the emissivity of the gas, not the walls, and will depend on a characteristic reactor dimension (the mean beam length) in the optically thin limit. In the very optically thick limit, it is unlikely that the radiative loss can be computed with a stirred reactor model, since the wall boundary layers may be optically thick.

Example

```plaintext
use cantera
type(cstr_t) r
r = StirredReactor()
call setEmissivity(r, 0.001)
```

131) setExtPressure

Set the external pressure [Pa].

Syntax

```plaintext
call setExtPressure(react, p0)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>react</td>
<td>(Input / Output) Reactor object [type(cstr_t)].</td>
</tr>
<tr>
<td>p0</td>
<td>(Input) External pressure [double precision].</td>
</tr>
</tbody>
</table>

Description

This pressure is used to form the pressure difference used to evaluate the rate at which the volume changes.

Example

```plaintext
use cantera
type(cstr_t) r
r = StirredReactor()
call setExtPressure(r, OneAtm)
```
6.6 Specifying the Mixture

Specify the mixture contained in the reactor.

**Syntax**

```plaintext
call setMixture(reac, mix)
```

**Arguments**

- `reac` (Input / Output) Reactor object [type(cstr_t)].
- `mix` (Input / Output) Mixture object [type(mixture_t)].

**Description**

Note that a pointer to this substance is stored, and as the integration proceeds, the state of the mixture is modified. Nevertheless, one mixture object can be used for multiple reactors. When `advance` is called for each reactor, the mixture state is set to the appropriate state for that reactor.

**Example**

```plaintext
use cantera
type(mixture_t) gas
type(cstr_t) reactr1, reactr2

! create the gas mixture and set its state
gas = CKGas('mymech.inp')
if (.not.ready(mix)) stop
call setState_TPX(mix, 1200.d0, OneAtm,
 & 'C2H2:1.0, NH3:2.0, O2:0.5')

! create a reactor object, and insert the gas
reactr1 = StirredReactor()
call setMixture(reactr1, gas)

! use the same gas object for another reactor
call setState_TPX(mix, 100.d0, OneAtm,
 & 'C2H2:1.0, NH3:2.0, O2:0.5')
reactr2 = StirredReactor()
call setMixture(reactr2, gas)
```
Syntax
\[ \text{result} = \text{contents} (\text{react}) \]

Arguments
- \textit{react} (Input / Output) Reactor object [type(cstr_t)].

Result
Returns a \textit{type} (mixture_t) value.

6.7 Operation

\begin{verbatim}
134) advance
\end{verbatim}

Advance the state of the reactor forward in time to \textit{time}.

Syntax
\[ \text{call advance} (\text{react}, \text{time}) \]

Arguments
- \textit{react} (Input / Output) Reactor object [type(cstr_t)].
- \textit{time} (Input) Final time [s] [double precision].

Description
Note that this method changes the state of the mixture object. On the first call, initialization is carried out and the maximum integrator step size is set. By default, this is set to \textit{time}. To specify a different maximum step size, precede the call to \textit{advance} with a call to \textit{setMaxStep}. Note that this cannot be reset after \textit{advance} has been called.

6.8 Reactor Attributes

\begin{verbatim}
135) residenceTime
\end{verbatim}

The residence time [s].

Syntax
\[ \text{result} = \text{residenceTime} (\text{react}) \]

Arguments
- \textit{react} (Input / Output) Reactor object. [type(cstr_t)].

Result
Returns a double precision value.
The current time [s].

**Syntax**

\[
\text{result} = \text{time}(\text{reac})
\]

**Arguments**

- \( \text{reac} \) (Input / Output) Reactor object. [type(cstr_t)].

**Result**

Returns a double precision value.

---

The reactor volume [m\(^3\)].

**Syntax**

\[
\text{result} = \text{volume}(\text{reac})
\]

**Arguments**

- \( \text{reac} \) (Input / Output) Reactor object. [type(cstr_t)].

**Result**

Returns a double precision value.

---

### 6.9 Mixture Attributes

These procedures return properties of the reactor contents at the time reached by the last call to `advance`. These values are stored in the reactor object, and are not affected by changes to the mixture object after `advance` was last called.

The density [kg/m\(^3\)].

**Syntax**

\[
\text{result} = \text{density}(\text{reac})
\]

**Arguments**
\textit{reac} \hspace{1cm} (Input / Output) Reactor object. [type(cstr\_t)].

\textbf{Result}

Returns a double precision value.

\begin{verbatim}
139) temperature
\end{verbatim}

The temperature [K].

\textbf{Syntax}

\begin{verbatim}
result = temperature(reac)
\end{verbatim}

\textbf{Arguments}

\begin{verbatim}
reac \hspace{1cm} (Input / Output) Reactor object. [type(cstr\_t)].
\end{verbatim}

\textbf{Result}

Returns a double precision value.

\begin{verbatim}
140) enthalpy\_mass
\end{verbatim}

The specific enthalpy [J/kg].

\textbf{Syntax}

\begin{verbatim}
result = enthalpy\_mass(reac)
\end{verbatim}

\textbf{Arguments}

\begin{verbatim}
reac \hspace{1cm} (Input / Output) Reactor object. [type(cstr\_t)].
\end{verbatim}

\textbf{Result}

Returns a double precision value.

\begin{verbatim}
141) intEnergy\_mass
\end{verbatim}

The specific internal energy [J/kg].

\textbf{Syntax}

\begin{verbatim}
result = intEnergy\_mass(reac)
\end{verbatim}

\textbf{Arguments}

\begin{verbatim}
\end{verbatim}
reac (Input / Output) Reactor object. [type(cstr_t)].

Result
Returns a double precision value.

142) pressure

The pressure [Pa].

Syntax
result = pressure(reac)

Arguments
reac (Input / Output) Reactor object. [type(cstr_t)].

Result
Returns a double precision value.

143) mass

The total mass $\rho V$.

Syntax
result = mass(reac)

Arguments
reac (Input / Output) Reactor object. [type(cstr_t)].

Result
Returns a double precision value.

144) enableChemistry

Enable chemistry.

Syntax
call enableChemistry(reac)

Arguments
reac (Input / Output) Reactor object. [type(cstr_t)].
disableChemistry

Disable chemistry.

**Syntax**

call disableChemistry(*reac*)

**Arguments**

*reac*  
(Input / Output) Reactor object. [type(cstr_t)].
**Flow Devices**

“Flow devices” are objects that regulate fluid flow. These objects are designed to be used in conjunction with the stirred flow.

Perhaps these is a better name for these than “flow devices.” Any suggestions?

**flowdev_t**
Object type used to represent flow control devices.

- **hndl** integer . Handle encoding pointer to the C++ object
- **upstream** type(cstr_t) . Upstream reactor
- **downstream** type(cstr_t) . Downstream reactor
- **type** integer . Flow controller type

### 7.1 Procedures

### 7.2 Constructors

These functions construct new flow devices.

146) MassFlowController

Create a new mass flow controller. A mass flow controller maintains a specified mass flow rate, independent of all other conditions.

**Syntax**

```object = MassFlowController()```

**Result**

Returns an object of type `flowdev_t`. 
147) PressureController

Create a new pressure controller.

**Syntax**

```python
object = PressureController()
```

**Result**

Returns an object of type `flowdev_t`.

**Description**

This device attempts to maintain a specified *upstream* pressure by adjusting its mass flow rate. It is designed to regulate the exhaust flow rate from a reactor, not the inlet flow rate to a reactor (i.e., to decrease the pressure, the valve opens wider) It should be installed on an outlet of the reactor to be controlled. This device will only function correctly if the reactor also has one or more inlets with positive flow rates. It also acts as a check valve, and does not permit reverse flow under any conditions.

7.3 Assignment

148) copy

Perform a shallow copy of one flo device to another. The contents of `dest` are overwritten.

**Syntax**

```python
call copy(src, dest)
```

**Arguments**

- `src` (Input) Flow controller object [type(flowdev_t)].
- `dest` (Output) Flow controller object [type(flowdev_t)].

7.4 Setting Options

149) setSetpoint

Set the setpoint. The units depend on the type of flow control device. May be called at any time.

**Syntax**

```python
call setSetpoint(flowDev, setpnt)
```

**Arguments**
flowDev  (Input / Output) Flow controller object. [type(flowdev_t)].
setpnt  (Input) Setpoint. [double precision].

Description
For mass flow controllers, this sets the mass flow rate [kg/s]. For pressure controllers, it sets the pressure setpoint [Pa].

Syntax

```
call install(flowDev, upstream, downstream)
```

Arguments

- **flowDev** (Input / Output) Flow controller object. [type(flowdev_t)].
- **upstream** (Input / Output) Upstream reactor object. [type(cstr_t)].
- **downstream** (Input / Output) Downstream reactor object. [type(cstr_t)].

Syntax

```
result = upstream(flowDev)
```

Arguments

- **flowDev** (Input / Output) Flow controller object. [type(flowdev_t)].

Result

Returns a type (cstr_t) value.
152) **downstream**

Return a reference to the downstream reactor.

**Syntax**

```c
result = downstream(flowDev)
```

**Arguments**

- `flowDev` (Input / Output) Flow controller object. [type(flowdev_t)].

**Result**

Returns a `type(cstr_t)` value.

153) **massFlowRate**

Mass flow rate [kg/s].

**Syntax**

```c
result = massFlowRate(flowDev)
```

**Arguments**

- `flowDev` (Input / Output) Flow controller object. [type(flowdev_t)].

**Result**

Returns a double precision value.

154) **update**

Update the state of the flow device. This only needs to be called for those flow controllers that have an internal state, such as pressure controllers that use a PID controller.

**Syntax**

```c
call update(flowDev)
```

**Arguments**

- `flowDev` (Input / Output) Flow controller object. [type(flowdev_t)].
Reset the controller. Call this procedure before operating any flow controller that uses a PID controller.

Syntax

call reset(flowDev)

Arguments

flowDev  (Input / Output) Flow controller object. [type(flowdev_t)].

Returns true if flow controller is ready for use.

Syntax

result = ready(flowDev)

Arguments

flowDev  (Input / Output) Flow controller object. [type(flowdev_t)].

Result

Returns a integer value.

Some flow control devices use a PID (proportional / integral / derivative) controller. This procedure can be used to set the gains for the PID controller.

Syntax

call setGains(flowDev, gains)

Arguments

flowDev  (Input / Output) Flow controller object. [type(flowdev_t)].

gains  (Input) gains [double precision array]. Must be dimensioned at least 4.
getGains

Some flow control devices use a PID (proportional / integral / derivative) controller. This procedure can be used to retrieve the current gain settings for the PID controller.

Syntax
   call getGains(flowDev, gains)

Arguments
   flowDev (Input / Output) Flow controller object. [type(flowdev_t)].
   gains (Output) gains [double precision array]. Must be dimensioned at least 4.

maxError

Maximum absolute value of the controller error signal (input - setpoint) since the last call to 'reset'.

Syntax
   result = maxError(flowDev)

Arguments
   flowDev (Input / Output) Flow controller object. [type(flowdev_t)].

Result
   Returns a double precision value.
CHAPTER EIGHT

Utility Functions
Utilities

9.1 Introduction

These procedures carry out various utility functions.

9.2 Procedures

9.3 Procedures

160) printSummary

Write to logical unit lu a summary of the mixture state.

Syntax

\text{call printSummary}(mix, lu)

Arguments

\begin{itemize}
  \item \texttt{mix} \hspace{1cm} \text{(Input) Mixture object [\text{mixture_t}].}
  \item \texttt{lu} \hspace{1cm} \text{(Input) Fortran logical unit for output [integer].}
\end{itemize}
Glossary

substance
A macroscopic sample of matter with a precise, characteristic composition. Pure water is a substance, since it is always made up of $\text{H}_2\text{O}$ 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 $\text{Si}_x\text{Ge}_{1-x}$, 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.
BIBLIOGRAPHY


INDEX

addDirectory subroutine, 21
addElement subroutine, 27
advance subroutine, 94
atomicWeight function, 29
charge function, 30
contents function, 93
copy subroutine, 23, 87, 100
cp_mass function, 56
cp_mole function, 53
critPressure function, 58
critTemperature function, 57
cstr_t constructors, 85, 86
cv_mass function, 56
cv_mole function, 54
delete subroutine, 22, 78
density function, 51, 95
disableChemistry subroutine, 98
downstream function, 102
elementIndex function, 28
enableChemistry subroutine, 97
enthalpy_mass function, 54, 96
enthalpy_mole function, 52
entropy_mass function, 55
entropy_mole function, 53
equationOfState function, 59
equilibrate subroutine, 61
flowdev_t constructors, 99, 100
getchemPotentials_RT subroutine, 54
getchConcentrations subroutine, 35
getc_p_R subroutine, 49
getchCreationRates subroutine, 72
getchDestructionRates subroutine, 72
getchElementNames subroutine, 27
getchenthalpy_RT subroutine, 48
getchEntropy_R subroutine, 49
getchEquilibriumConstants subroutine, 71
getchFwdRatesOfProgress subroutine, 69
getGains subroutine, 104
getGibbs_RT subroutine, 48
getMassFractions subroutine, 35
getMoleFractions subroutine, 35
getMolecularWeights subroutine, 32
getMultiDiffCoeffs subroutine, 82
getNetProductionRates subroutine, 73
getNetRatesOfProgress subroutine, 70
getReactionString subroutine, 65
getRevRatesOfProgress subroutine, 69
getSpeciesFluxes subroutine, 79
getSpeciesNames subroutine, 31
getSpeciesViscosities subroutine, 79
getThermalDiffCoeffs subroutine, 83
gibbs_mass function, 55
gibbs_mole function, 53
install subroutine, 101
intEnergy_mass function, 55, 96
intEnergy_mole function, 52
massFlowRate function, 102
massFraction function, 34
mass function, 97
maxError function, 104
maxTemp function, 50
meanMolecularWeight function, 37
mean_X function, 36
mean_Y function, 36
minTemp function, 50
mixture_t constructors, 16–20, 26
molarDensity function, 51
moleFraction function, 34
molecularWeight function, 31
nAtoms function, 29
nElements function, 25
nReactions function, 26
nSpecies function, 25
netStoichCoeff function, 68
potentialEnergy function, 57
pressure function, 52, 97
printSummary subroutine, 107
productStoichCoeff function, 67
reactantStoichCoeff function, 66
ready function, 22, 103
refPressure function, 50
reset subroutine, 103
residenceTime function, 94
restoreState subroutine, 24
satPressure function, 59
satTemperature
  function, 58
saveState
  subroutine, 23
setArea
  subroutine, 89
setConcentrations
  subroutine, 34
setDensity
  subroutine, 42
setEmissivity
  subroutine, 92
setEquationOfState
  subroutine, 60
setExtPressure
  subroutine, 92
setExtRadTemp
  subroutine, 90
setExtTemp
  subroutine, 90
setGains
  subroutine, 103
setHeatTransferCoeff
  subroutine, 91
setInitialTime
  subroutine, 88
setInitialVolume
  subroutine, 87
setMassFractions_NoNorm
  subroutine, 33
setMassFractions
  subroutine, 33
setMaxStep
  subroutine, 88
setMixture
  subroutine, 93
setMoleFractions_NoNorm
  subroutine, 32
setMoleFractions
  subroutine, 32
setOptions
  subroutine, 83
setPotentialEnergy
  subroutine, 56
setPressure
  subroutine, 44
setSetpoint
  subroutine, 100
setState_HP
  subroutine, 46
setState_PX
  subroutine, 43
setState_PY
  subroutine, 43
setState_RX
  subroutine, 45
setState_RY
  subroutine, 45
setState_SP
  subroutine, 47
setState_SV
  subroutine, 47
setState_TPX
  subroutine, 39
setState_TPY
  subroutine, 40, 41
setState_TP
  subroutine, 43
setState_TRX
  subroutine, 41
setState_TRY
  subroutine, 42
setState_TR
  subroutine, 44
setState_TX
  subroutine, 44
setState_TY
  subroutine, 45
setState_UV
  subroutine, 47
setTemperature
  subroutine, 42
setTransport
  subroutine, 77
setVDotCoeff
  subroutine, 91
speciesIndex
  function, 30
sum_xlogQ
  function, 37
temperature
  function, 51, 96
thermalConductivity
  function, 83
time
  function, 95
transportMgr
function, 78
transport_t
constructors, 75, 76
update
subroutine, 102
upstream
function, 101
viscosity
function, 78
volume
function, 95
bulk viscosity, 79
$CANTERA_DATA_DIR, 22
chemical equilibrium, 61
chemical potential, 54
CK format, 16
congcentration, 51
constructors, 7
CKGas, 16
GRIMech30, 17
Hydrogen, 19
MassFlowController, 99
Methane, 19
MixTransport, 76
Mixture, 26
MultiTransport, 75
Nitrogen, 18
Oxygen, 20
PressureController, 100
Reservoir, 86
StirredReactor, 85
Water, 18
density, 51
density
molar, 51
setting, 42
energy
internal, 47
potential, 56
enthalpy
molar, 52
non-dimensional, 48
pure species, 48
setting, 46
specific, 54
entropy
molar, 53
non-dimensional, 49
setting, 47
specific, 55
environment variables
$CANTERA_DATA_DIR, 22
equation of state, 59
equation of state
specifying, 60
equilibrium
chemical, 61
format
CK, 16
free energy
Gibbs, 48
Gib, 55
molar Gibbs, 53
generic names, 10
Gibbs function
molar, 53
non-dimensional, 48
pure species, 48
specific, 55
handle, 7
heat capacity
constant pressure, 53
constant volume, 54
molar, 53, 54
internal energy
molar, 52
setting, 47
specific, 55
Jacobian, 46
kernel, 6
kinetics manager, 14
member functions, 9
methods, 9
mixture rule, 79
mixtures, 13
Newton, 46

Index
potential
  chemical, 54
potential energy, 56
pressure, 52
pressure
  reference, 50
  setting, 47
  standard, 50
properties
  molar, 52
  species thermodynamic, 48
  specific, 54
  thermodynamic, 38

specific heat
  constant pressure, 49, 56
  constant volume, 56
  non-dimensional, 49
state
  thermodynamic, 38

temperature, 51
temperature
  maximum, 50
  minimum, 50
  setting, 42
thermodynamic property manager, 14
thermodynamic state, 38
thermodynamic state
  setting, 39
transport property manager, 14

viscosity, 79
viscosity
  bulk, 79
volume
  setting, 47