Using Cantera from Fortran 77

Release 1.4

David G. Goodwin

April 23, 2003

California Institute of Technology
Email: dgoodwin@caltech.edu

1 Introduction

More than 20 years after its introduction in 1980, Fortran 77 remains widely used for scientific computing, even though the official Fortran standard is now Fortran 90/95. This is due to several factors, including the existence of a large body of Fortran 77 software, the efficiency of mature Fortran 77 compilers for numerical calculations, and the large number of scientists and engineers who are fluent in Fortran 77.

Many reacting flow codes are written in Fortran 77, so the question naturally arises whether Cantera can be used from Fortran 77. The answer is of course "yes," and in this document we'll describe how to do it.

If you have a Fortran 77 application program, or are planning to write one, there are many reasons why you might want to use Cantera from your program. Perhaps your program integrates equations that contain thermodynamic property coefficients or chemical source terms; if so, you can use Cantera to evaluate them. Perhaps you need to know the adiabatic flame temperature of a gas mixture; Cantera can be used to find it. Or perhaps you want to visualize reaction pathways at various points in a flowfield; Cantera can help with this too.

Of course, since Cantera is written in C++ and uses *objects* to represent high-level abstractions like gas mixtures, rate coefficients, reaction path diagrams, etc., you can't *directly* invoke Cantera from Fortran 77. Fortran 77 has concept of objects, nor does it know anything about C++. To get a Fortran 77 program to communicate with Cantera requires a "go-between" interface procedure — one that appears to your Fortran code to be just another Fortran subroutine or function, but that internally invokes Cantera in C++ to carry out the desired calculation.

We'll look at how to write such interface procedures by examining a complete, working example of a Fortran 77 program that uses Cantera. The files required to build and run this example (a Fortran 77 main program, a set of Fortran-callable interface procedures in C++, and a Makefile) are included in the Cantera installation, so you can build and run this example yourself. You can also use this example as a starting point to write a Makefile and the interface procedures for your application.

2 Program `demo` — an Example f77/Cantera Application

Program `demo` is designed to illustrate how to interface a Fortran 77 program to Cantera. To build it, copy the three files in '/usr/local/cantera/templates/f77' to a directory where you have write access. (If you installed Cantera somewhere other than '/usr/local', then substitute the installation directory for '/usr/local' here and below.)

A slightly condensed version of 'demo.f' is shown below.

```
      program demo
```

```
      implicit double precision (a-h,o-z)
      parameter (MAXSP = 20, MAXRXNS = 100)
      double precision q(MAXRXNS), qf(MAXRXNS), qr(MAXRXNS)
      double precision x(MAXSP), y(MAXSP), wdot(MAXSP)
      character*80 eq
c
      write(*,*) '**** Fortran 77 Test Program ****'
c
c---create the gas mixture
      call newIdealGasMix('h2o2.xml','')
c
c---set its initial state
      t = 1200.0
      p = 101325.0
      call setState_TPX_String(t, p,
     *     'H2:2, O2:1, OH:0.01, H:0.01, O:0.01')
c
c---set it to equilibrium at constant h and P
      call equilibrate('HP')
c
c---write some thermodynamic properties
      write(*,10) temperature(), pressure(), density(),
     *     enthalpy_mole(), entropy_mole(), cp_mole()
 10   format(//'Temperature:   ',g14.5,' K'/
     *         'Pressure:      ',g14.5,' Pa'/
     *         'Density:       ',g14.5,' kg/m3'/
     *         'Molar Enthalpy:',g14.5,' J/kmol'/
     *         'Molar Entropy: ',g14.5,' J/kmol-K'/
     *         'Molar cp:      ',g14.5,' J/kmol-K'//)
c
c---Reaction information
      irxns = nReactions()
      call getFwdRatesOfProgress(qf)
      call getRevRatesOfProgress(qr)
      call getNetRatesOfProgress(q)
      do i = 1,irxns
         call getReactionEqn(i,eq)
         write(*,20) eq,qf(i),qr(i),q(i)
 20      format(a20,3g14.5,' kmol/m3/s')
      end do
      stop
      end
```

The program is standard Fortran 77, except that the procedure names are longer than 8 characters. Most Fortran 77 compilers now allow long names, but if yours doesn't or you simply prefer short names, all you have to do is edit 'demo_ftnlib.cpp' and 'demo.f' to change the names.

To build the `demo` application, type

```
make -f demo.mak
```

This will compile the two source files `demo.f` and `demo_ftnlib.cpp` and link them to the Cantera libraries to create the executable program `demo`.

The Makefile requires a 'make' utility that is compatible with GNU 'make.' On some platforms (Solaris) this may be installed as 'gmake.'

The output of this program is shown below.

```
 **** Fortran 77 Test Program ****


Temperature:       3187.7     K
Pressure:         0.10133E+06 Pa
Density:          0.53652E-01 kg/m3
Molar Enthalpy:   0.34369E+08 J/kmol
Molar Entropy:    0.26632E+06 J/kmol-K
Molar cp:          44717.     J/kmol-K


2 O + M <=> O2 + M     0.41979E-01   0.41979E-01   0.62450E-16 kmol/m3/s
O + H + M <=> OH + M   0.17494       0.17494        0.0000     kmol/m3/s
O + H2 <=> H + OH       5016.8        5016.8       0.14552E-10 kmol/m3/s
O + HO2 <=> OH + O2    0.70924       0.70924      -0.21094E-14 kmol/m3/s
O + H2O2 <=> OH + HO   0.10087       0.10087      -0.24980E-15 kmol/m3/s
H + 2 O2 <=> HO2 + O   0.19111E-01   0.19111E-01  -0.12490E-15 kmol/m3/s
H + O2 + H2O <=> HO2    4.2683        4.2683       0.10658E-13 kmol/m3/s
H + O2 + AR <=> HO2     0.0000        0.0000        0.0000     kmol/m3/s
H + O2 <=> O + OH       746.13        746.13       -0.45475E-11 kmol/m3/s
2 H + M <=> H2 + M     0.73995E-01   0.73995E-01  -0.12490E-15 kmol/m3/s
2 H + H2 <=> 2 H2      0.81805E-01   0.81805E-01   0.16653E-15 kmol/m3/s
2 H + H2O <=> H2 + H   0.84809       0.84809       0.16653E-14 kmol/m3/s
H + OH + M <=> H2O +    3.7171        3.7171        0.0000     kmol/m3/s
H + HO2 <=> O + H2O    0.28353       0.28353        0.0000     kmol/m3/s
H + HO2 <=> O2 + H2     3.0051        3.0051       0.23537E-13 kmol/m3/s
H + HO2 <=> 2 OH        6.0332        6.0332       -0.88818E-15 kmol/m3/s
H + H2O2 <=> HO2 + H   0.23480       0.23480      -0.22204E-15 kmol/m3/s
H + H2O2 <=> OH + H2   0.24584E-01   0.24584E-01  -0.52042E-16 kmol/m3/s
OH + H2 <=> H + H2O     7425.0        7425.0       -0.15461E-10 kmol/m3/s
2 OH (+ M) <=> H2O2     4.2464        4.2464       0.79936E-14 kmol/m3/s
2 OH <=> O + H2O        2876.7        2876.7       -0.13642E-11 kmol/m3/s
OH + HO2 <=> O2 + H2    1.3933        1.3933       0.73275E-14 kmol/m3/s
OH + H2O2 <=> HO2 +    0.90743E-02   0.90743E-02   0.57246E-16 kmol/m3/s
OH + H2O2 <=> HO2 +     79.432        79.432       0.49738E-12 kmol/m3/s
2 HO2 <=> O2 + H2O2    0.58786E-05   0.58786E-05  -0.84703E-21 kmol/m3/s
2 HO2 <=> O2 + H2O2    0.22081E-02   0.22081E-02  -0.43368E-18 kmol/m3/s
OH + HO2 <=> O2 + H2    28.781        28.781       0.14566E-12 kmol/m3/s
```

3 The Library demo_ftnlib

Although meant as a demonstration of writing an interface library rather than as an "official" Cantera Fortran 77 interface, in fact the 'demo_ftnlib' library contains many procedures that can be called to compute the properties of reacting ideal gas mixtures. These procedures may be all that many applications need.

Listed below are the functions and subroutines implemented in this library. Note that the capitalization of the procedure names is arbitrary, since Fortran is not case-sensitive. The capitalization used here is designed to promote readability, and follows the same conventions as those used in Cantera itself.

3.1 Specifying the Gas Mixture

This library is designed to work with one gas mixture at a time. The first thing that should be done is to call `newIdealGasMix` to read in the mixture attributes from an input file. Cantera supports two file formats for this purpose: CTML, an XML-derived markup language that can be used to specify the properties of a general phase of matter, and a format compatible with the Chemkin [**?**Kee et al., 1996] software package that is restricted to specifying ideal gas mixtures.

subroutine newIdealGasMix(*inputFile, thermoFile***).** Read a specification for a reacting ideal gas mixture (or "reaction mechanism") from a file in CTML format or Chemkin-compatible format. Note that if a Chemkin-compatible file is used, an equivalent CTML file will also be written by this call. Both arguments are character strings; the first is the name of the file containig the specification, and the second is the name of a thermodynamic database file, which is only relevant if *inputFile* is in Chemkin-compatible format. If a CTML file is specified, the second argument should be an empty string.

This subroutine may be called multiple times. Each call removes the previously-loaded mixture model, and replaces it with the new one. The state is set to the default for the new mixture.

Examples:

```
call newIdealGasMix('chem.inp','therm.dat')
call newIdealGasMix('chem.xml','')
```

3.2 Constant attributes

Some attributes are constants, independent of the state. These functions return a few of them.

integer function nElements(). Number of elements.

integer function nReactions(). Number of reactions.

integer function nSpecies(). Number of species.

subroutine getReactionEqn(*i, eqn***).** Returns the reaction equation string for reaction number *i* in *eqn*. If string *eqn* is not large enough to contain the entire reaction equation string, the equation returned will be truncated.

3.3 Setting the State

These subroutines set the thermodynamic state of the gas mixture. All property functions return values for the state set by the last call to one of these subroutines.

subroutine setState_TPX(*T, P, X***).** Set the state by specifying the temperature, pressure, and mole fractions. The mole fractions are specified by an array, which must be at least as large as `nSpecies()`. The input mole fraction values will be scaled to sum to 1.0.

Example:

```
double precision T, P, X(100)
...
call setState_TPX(T, P, X)
```

subroutine setState_TPX_String(*T, P, X***).** Set the state by specifying the temperature, pressure, and mole fractions. The mole fractions are specified by a string, which must have a format like that shown below. Unspecified species are set to zero, and the rest are scaled to sum to 1.0.

```
call setState_TPX_String(T, P, 'CH4:1, O2:2, N2:7.52')
```

subroutine setState_TRY(*T, rho, Y***).** Set the state by specifying the temperature, density, and mass fractions. The mass fractions are specified by an array, which must be at least as large as `nSpecies()`. The input mass fraction values will be scaled to sum to 1.0.

3.4 Chemical Equilibrium

subroutine equilibrate(*XY***).** Set the state to a state of chemical equilibrium, holding two properties fixed at their values in the current state. The argument *XY* is a two-character string that specifies the two thermodynamic properties to be held fixed, and must be one of `'TP'`, `'TV'`, `'HP'`, `'UV'`, `'SV'`, or `'SP'`.

Example:

```
call setState_TPX_String(temp, pres, 'CH4:1, O2:2, N2:7.52')
call equilibrate('HP')   ! find adiabatic flame temp.
Tad = temperature()
```

3.5 Properties of the Current State

These functions all return properties of the current state, which is the state sset by the last call to one of the `setState` subroutines. As always in Cantera, the procedures return values in SI units (kmol, kg, s, m, K).

double precision function temperature(). The temperature [K].

double precision function pressure(). The pressure [Pa].

double precision function density(). The density [kg/m^3].

double precision function meanMolarMass(). The mean molar mass [kg/kmol].

double precision function enthalpy_mole(). The molar enthalpy [J/kmol].

double precision function entropy_mole(). The molar entropy [J/kmol/K].

double precision function intEnergy_mole(). The molar internal energy [J/kmol].

double precision function cp_mole(). The molar heat capacity at constant pressure [J/kmol/K].

double precision function gibbs_mole(). The specific Gibbs function [J/kg].

double precision function enthalpy_mass(). The specific enthalpy [J/kg].

double precision function entropy_mass(). The specific entropy [J/kg/K].

double precision function intEnergy_mass(). The specific internal energy [J/kg].

double precision function cp_mass(). The specific heat at constant pressure [J/kg/K].

double precision function gibbs_mass(). The specific Gibbs function [J/kg].

subroutine getFwdRatesOfProgress(*qfwd***).** Returns the reaction forward rates of progress [kmol/m^3/s] in array *qfwd*, which must be at least as large as `nReactions()`.

subroutine getRevRatesOfProgress(*qrev*). Returns the reaction reverse rates of progress [kmol/m^3/s] in array *qrev*, which must be at least as large as `nReactions()`.

subroutine getNetRatesOfProgress(*qnet*). Returns the reaction net rates of progress [kmol/m^3/s] in array *qnet*, which must be at least as large as `nReactions()`.

subroutine getNetProductionRates(*wdot*). Returns the species net production rates [kmol/m^3/s] in array *wdot*, which must be at least as large as `nSpecies()`.

subroutine getCreationRates(*cdot*). Returns the species creation rates [kmol/m^3/s] in array *cdot*, which must be at least as large as `nSpecies()`.

subroutine getDestructionRates(*ddot*). Returns the species destruction rates [kmol/m^3/s] in array *ddot*, which must be at least as large as `nSpecies()`.

4 Building your Application

Now that you have built the `demo` application and verified that it works, you are ready to write your own Fortran 77 application program and build it in the same way. Let's take the simplest case first. Suppose that the procedures in the `demo_ftnlib` library are all you require for your application. In this case, all that is really required is to edit the Makefile to specify your program file(s), instead of 'demo.f'.

First, copy file 'demo.mak' to 'Makefile', so that you can simply type `make` to build your application. Now edit 'Makefile' to specify your program files and any other libraries your application may require. You can also change other parameters, if you like.

For example, if your application program is called `react`, and consists of program files 'react.f', 'sub1.f', and 'sub2.f', and uses additional procedures from external libraries `linpack` and `blas`, then the first few lines of 'Makefile' would look like this:

```
# the name of the executable program to be created
PROG_NAME = react

# the object files to be linked together.
OBJS = react.o sub1.o sub2.o demo_ftnlib.o

# additional flags to be passed to the linker. If your program
# requires other external libraries, put them here
LINK_OPTIONS = -llinpack -lblas
```

After editing the Makefile, typing `make` at the command prompt should now build your application and link it to Cantera.

Note that 'demo.mak' was created when Cantera was first configured for your system, and so contains system-specific information. For example, it specifies the directory where Cantera libraries may be found. It also specifies some libraries required by Fortran that must be added explicitly to the link command, since the C++ compiler is used for the linking step. (The Fortran compiler knows it needs these libraries; the C++ compiler doesn't. On the other hand, C++ also uses additional libraries that Fortran does not know about.) If you build your program on another computer system, it may be easiest to get the version of 'demo.mak' on that system and modify it as you did on your system.

5 Inside `demo_ftnlib`

If the example library 'demo_ftnlib' does not have interface functions for some features of Cantera, you can modify the file to add new procedures, or write an entirely new file. To do so requires a bit of C++ programming. However,

since these functions don't *do* anything other than act as the "glue" between your Fortran application and the Cantera kernel, they are easy to write. Many have bodies of only one line.

Let's look at file 'demo_ftnlib.cpp' to see how it works. Here is a condensed version, showing only a few of the function definitions.

```
#include "IdealGasMix.h"
#include "equilibrium.h"

static IdealGasMix* _gas = 0;

extern "C" {

    // This is the Fortran main program
    extern int MAIN__();

    void newidealgasmix_(char* file, char* thermo,
        ftnlen lenfile, ftnlen lenthermo) {
        string fin = string(file, lenfile);
        string fth = string(thermo, lenthermo);
        if (_gas) delete _gas;
        _gas = new IdealGasMix(fin, fth);
        _init();
    }

    integer nelements_() { return _gas->nElements(); }
...
    doublereal enthalpy_mass_() {
        return _gas->enthalpy_mole();
    }
...
    void getnetratesofprogress_(doublereal* q) {
        _gas->getNetRatesOfProgress(q);
    }
...
    // This C++ main program simply calls the Fortran main program.
    int main() {
      try {
          return MAIN__();
      }
      catch (CanteraError) {
          showErrors(cerr);
          return -1;
      }
      catch (...) {
          cout << "An␣exception␣was␣trapped.␣Program␣terminating." << endl;
          return -1;
      }
  }
}
```

5.1 Cantera Header Files

At the top of the file, Cantera header files are included. Since this library is designed to provide functions to compute properties of ideal gas mixtures, Cantera C++ class `IdealGasMix` will be used. Also, a chemical equilibrium function will be implemented, which will call the function `equilibrate` declared in header file 'equilibrium.h'.

```
#include "IdealGasMix.h"
#include "equilibrium.h"
```

5.2 Storing a Pointer

An object of class `IdealGasMix` will be used to define the state and compute the properties. This object needs to be stored somewhere; the approach taken here is to use a global pointer variable that points to a dynamically-created object.

```
static IdealGasMix* _gas = 0;
```

This statement creates a pointer in global storage that can point to an `IdealGasMix` object. Initially, however, it doesn't point to anything – it is initialized to zero (NULL). It is important to initialize the pointer, since otherwise on the first call to function `newIdealGasMix`, it would attempt to delete an object at the location where `_gas` happens to point, even though none is there. This would almost certainly generate a segentation fault, and cause your program to crash.

Of course, multiple pointers could be stored, if it were desired to access more than one Cantera object.

5.3 `extern "C"`

C++ normally modifies the names of procedures when it writes them to the object file ("name mangling") by encoding the argument types in the name. This allows a C++ program to define multiple functions that have the same name, as long as the arguments differ in number or type. There is no ambiguity, since the names in the object file are different.

Unfortunately, C, Fortran, and most other applications know nothing about C++ name mangling, and can't deduce what "mangled" name the should look for in the object file. Name mangling can be turned off, however, so that C++ can generate code that has the same external interface as C. Therefore, any procedure that is meant to be callable from Fortran or C must be eclosed in an `extern C¨` block:

```
extern "C" {
     void sub1_(double* x, double* y) { ... };
     double func1_(int* n, double* x) { ... };
}
```

5.4 Procedure Names

If you look at the definition of any of the procedures in 'demo_ftnlib.cpp', you will notice two things about the names: they are all lowercase, and they have a trailing underscore appended. They are defined this way because this is how most Fortran compilers write procedure names to the object file.

Unlike C, which writes names to the object file exactly as they appear in the source file, Fortran compilers modify the names. Many popular unix Fortran compilers (GNU g77, Solaris, DEC) translate names as shown here – they are converted to lower case, and a trailing underscore is appended. Other Fortran compilers (AIX, HP, Compaq Visual Fortran) have a different default behavior, but can optionally translate names in this way if certain flags are set.

5.5 Procedure Arguments

C and Fortran pass arguments to procedures differently. C passes most variables by value (except arrays and structures), while Fortran always passes variables by reference. As a result, the C equivalent of the Fortran procedure

```
      SUBROUTINE SUB2(X, Y, ARRAY1)
      DOUBLE PRECISION X, Y, ARRAY1(*)
```

is

```
void sub2_(double* x, double* y, double* array1);
```

Passing strings is done differently in Fortran and C. In C, strings are NULL-terminated (the last character is `'\0'`.) This character acts as a sentinel denoting the end of the string. Fortran strings are not null-terminated, and therefore an additional length parameter must be passed to a procedure taking a string argument, so that it knows how many characters should be read from memory. This parameter is passed after all other parameters have been specified; if multiple strings are passed, the length parameter for each one is appended to the end of the argument list, in order.

5.6 The Procedure Body

Most of the procedures in this library have very simple bodies; many are only one line. For the most part, they simply invoke an appropriate method of class `IdealGasMix` on the object pointed to by the pointer `_gas`.

5.7 The `main` Procedure

To use this library, it must of course be compiled with a C++ compiler, and then be linked to your program, which is compiled with a Fortran 77 compiler. The link step must also be done using the C++ compiler, so that C++-specific initializations are done.

During linking, the C++ compiler will look for a procedure named `main` as the entry point. (The Fortran compiler, on the other hand, uses the name `MAIN__` for the main program.) Since the Fortran application does not contain a routine named `main`, the library provides one. It simply calls the Fortran entry point `MAIN__` within a `try...catch` block to handle exceptions.

6 Adding New Procedures

xxx

6.1 Data Types

As shown here, it is assumed that Fortran type `DOUBLE PRECISION` corresponds to `double` in C, `INTEGER` corresponds to `int`, and the hidden string length parameters also have type `int`. While this is the case for most 32-bit systems, it may not always be true. To account for this, Cantera defines three preprocessor symbols that resolve to the appropriate C type, as shown below. The names of these symbols are the same as used by the `f2c` Fortran-to-C translator. (If in doubt about what the correct values should be, run `f2c` and see how it translates a small test Fortran program.)

These symbols are defined in Cantera header file ‘kernel/ct_defs.h’, which is included by virtually every other Cantera header file. Therefore, they can be used in your interface library to provide a degree of portability to other architectures.

Symbol	Fortran Type	Default Value
doublereal	DOUBLE PRECISION	double
integer	INTEGER	int
ftnlen	(hidden)	int

7 Summary

xxx

References

R. J. Kee, F. M. Rupley, E. Meeks, and J. A. Miller. Chemkin-III: A Fortran chemical kinetics package for the analysis of gas-phase chemical and plasma kinetics. Technical Report SAND96-8216, Sandia National Laboratories, 1996.