Zero-Dimensional Kinetics

David G. Goodwin

November 5, 2002

California Institute of Technology
Email: dgoodwin@caltech.edu

Contents

1 Introduction **1**

2 Getting Started **2**
2.1 Installing Python and NumPy 2
2.2 Installing the Cantera Python Package 2

3 Importing the Cantera Modules **2**

4 Reactors **2**
4.1 Methods `advance` and `step` 3
4.2 Writing CSV Files 4

5 Reservoirs **4**

6 Walls **4**
6.1 Heat Transfer 5
6.2 Volume Change 7

7 An Example **8**

8 Open Systems **8**

1 Introduction

This document is a very brief description of how to conduct zero-dimensional kinetics simulations using Cantera in Python. More extensive documentation will follow.

2 Getting Started

2.1 Installing Python and NumPy

If you don't have Python on your system, you need to first install it. If you are using a PC, go to `http://www.cantera.org`, select "Download" and then navigate to the 'Python/win32' directory. Download and run both installation programs found in this directory (Python first, then Numeric). Now you have Python installed on your system, and can skip this step in the future if you reinstall or upgrade the Cantera Python package.

If you are using a linux or unix workstation, go to `www.python.org` to get Python, and follow the links under "Scientific Computing" to get to the Numeric download site. (Be sure to get Numeric, not the newer NumArray package).

2.2 Installing the Cantera Python Package

If you are installing on a PC, go to the main download directory, navigate to '1.3/win32', and get 'PyCantera13r3.msi' (or a later release, if available). Run it, then follow the instructions in the README file.

If you are not using a PC, get the source distribution and build it.

3 Importing the Cantera Modules

The first thing to do is to start Python, and import the modules you will need.

```
>>> from Cantera import *
>>> from Cantera.Reactor import *
```

(This is written here as if you were running Python interactively; of course, it is usually better to write a script.)

4 Reactors

Cantera conducts zero-dimensional kinetics simulations using `Reactor` objects. Each reactor must have an associated object that represents the type of fluid the reactor contains. This object will be used to compute all required thermodynamic properties and species production rates, and therefore it must implement the reaction mechanism and equation of state desired for the reactor in question. Reactors that share the same mechanism may use the same object representing the fluid.

Suppose that the fluid is an ideal gas mixture, with species and reactions defined in reaction mechanism file 'mech.inp'. Species data not in this file may be found in file 'therm.dat'. The object representing the gas may be constructed as follows:

```
>>> gas = IdealGasMix('mech.inp','therm.dat')
```

For each reactor associated with this fluid object, set the state to the reactor initial conditions, and construct the reactor.

```
>>> gas.setState_TPX(1500.0, OneAtm, 'CH4:1, O2:2.2, AR:10')
>>> r = Reactor(gas)
>>> gas.setState_TPX(2000.0, 2.0*OneAtm, 'H2:1, O2:1, N2:3.76')
>>> r2 = Reactor(gas)
```

Unless connected to other reactors, a reactor is closed, adiabatic, and has a fixed volume. The volume defaults to 1 m^3, but may be changed.

```
>>> r2.setInitialVolume(4.0)
```

4.1 Methods `advance` and `step`

Now all that is required is to advance the state of the reactor(s) in time, and print out the quantities you are interested in.

```
>>> times = [1.e-6, 3.e-6, 1.e-5, 3.e-5, 1.e-4]
>>> for t in times:
...     r.advance(t)
...     print r.time(), r.temperature(), r.pressure()
...
1e-006 1999.39319026 101297.157964
3e-006 1998.27751397 101246.759955
1e-005 1995.35728266 101126.600882
3e-005 2039.55642698 103814.041611
0.0001 3090.10042832 168577.11871
```

If you are trying to catch a fast transient, use the `step` method, instead of `advance`:

```
>>> times = [1.e-6, 3.e-6, 1.e-5, 3.e-5, 1.e-4]
>>> for t in times:
...     tnow = -1.0
...     while tnow < t:
...         tnow = r.step(t)
...         print r.time(), r.temperature(), r.pressure()
...
1.51213178641e-017 2000.0 101325.0
1.51228299959e-013 1999.99999991 101324.999996
7.87176255724e-013 1999.99999951 101324.999978
(skipping 846 lines)
9.82632209872e-005 3090.28825544 168586.062026
9.92632209872e-005 3090.17798309 168580.811469
0.000100263220987 3090.07345531 168575.834388
```

The `step` method returns after every internal timestep. The array `times` is not really needed in this case, unless you want to insure that these times are among the set of output times.

The reactor properties are those at the end of the last call to `advance` or `step`. The state of the gas object may be reset without affecting these property values, which are stored within each reactor object. The properties that class `Reactor` defines may be determined using Python's help facility:

```
>>> help(Reactor)
```

To access other properties, set the state of the fluid object to the reactor state, and then examine the desired property of the gas.

```
>>> gas.setState_TPX(r.temperature(), r.pressure(), r.moleFractions())
>>> print gas.creationRates(), gas.fwdRatesOfProgress(), gas.equilibriumConstants()
```

4.2 Writing CSV Files

If you want to postprocess your data in Excel, use function `writeCSV` to write a file in comma-separated-value format:

```
>>> times = [1.e-6, 3.e-6, 1.e-5, 3.e-5, 1.e-4]
>>> f = open('react.csv','w')
>>> writeCSV(f,['time (s)', 'T (K)'] + list(gas.speciesNames()))
>>> for t in times:
...    tnow = -1.0
...    while tnow < t:
...        tnow = r.step(t)
...        writeCSV(f, [r.time(), r.temperature(),
...                     r.pressure()]+list(r.moleFractions()))
...
... f.close()
```

5 Reservoirs

A reservoir is a special type of reactor that has a constant state. Once set, its temperature, pressure, and chemical composition are constant, no matter how much heat or species are extracted or added. A reservoir can be used to provide a constant upstream source for open reactors, or to define the properties of the environment. A reservoir can be used wherever a reactor can be used.

```
env = Reservoir(Air(T = 300.0, P = OneAtm))
```

This statement also uses the `Air` function, which returns an object representing air (and including some high-temperature reactions).

6 Walls

Reactors and/or reservoirs may share common walls. A wall between two reactors (or reservoirs) may be defined as follows:

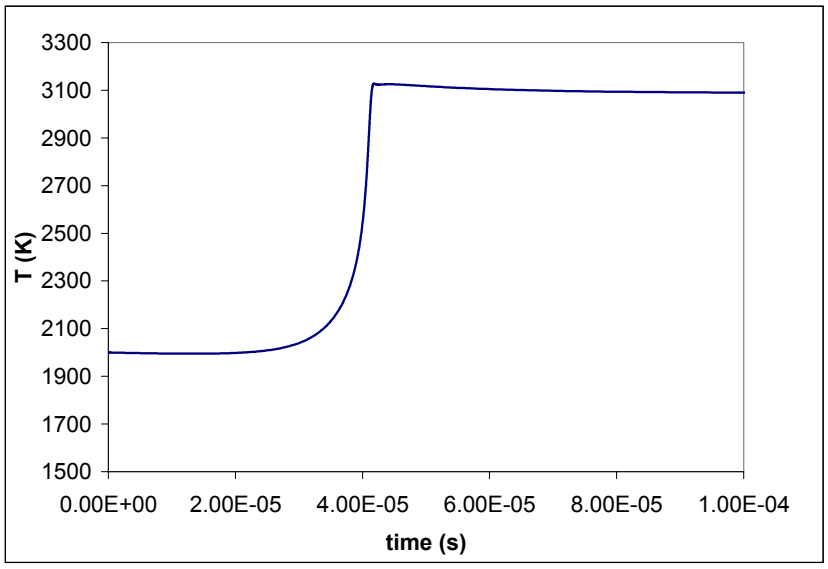

Figure 1: Computed temperature vs. time.

```
wall1 = Wall(r, env)
```

By default, walls are adiabatic and rigid. But they may be set to have other properties, as described below.

6.1 Heat Transfer

Heat may flow between the two reactors on either side of the wall, if they differ in temperature. The heat transfer rate is computed from

$$Q_{12} = UA(T_1 - T_2) \tag{1}$$

where A is the wall area (m^2) and U is the overall heat transfer coefficient (W/m^2/K). Both may be set using method `set`:

```
wall1.set(area = 5.0, U = 1.0e5)
```

With this heat-conducting wall providing a path for heat loss from the reactor to the environment, now the simulation predicts that the temperature falls instead of rises.

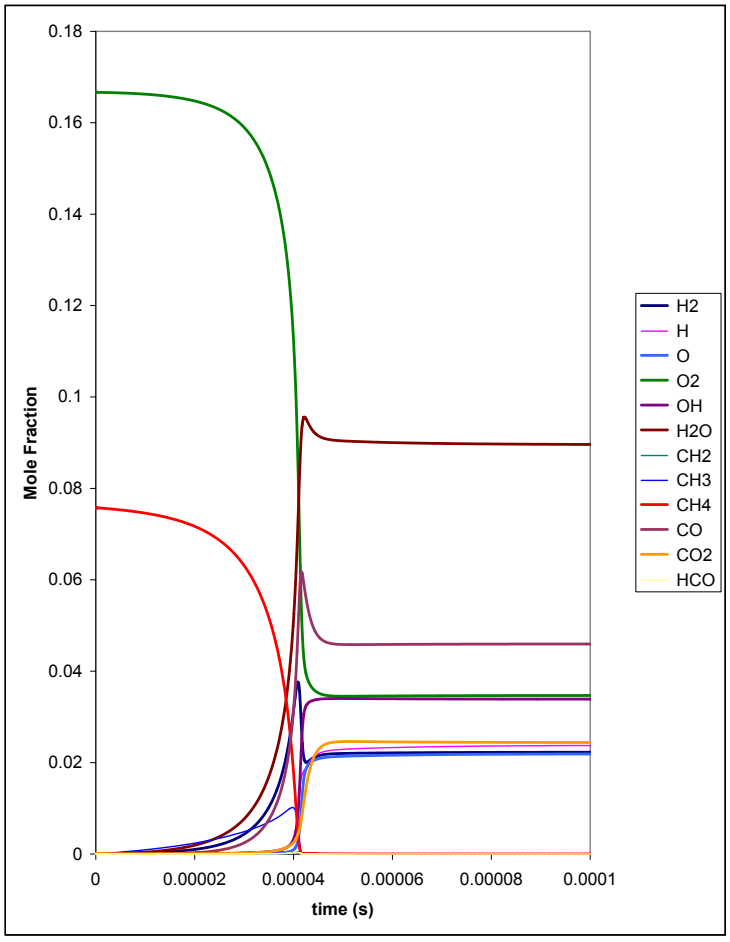

Figure 2: Selected mole fractions vs. time.

```
>>> times = [1.e-6, 3.e-6, 1.e-5, 3.e-5, 1.e-4]
>>> for t in times:
...     r.advance(t)
...     print r.time(), r.temperature(), r.pressure()
...
1e-006 1992.88888714 100967.523257
3e-006 1978.93880002 100266.029955
1e-005 1932.3358458 97921.9632879
3e-005 1816.19367259 92098.0028338
0.0001 1608.03356677 82117.1181917
```

Alternatively, the heat flow through the wall may be specified as a function of time. To do so, first create a function object that evaluates the desired function, using a class from module `Cantera.Func`.

```
>>> from Cantera.Func import *
>>> heatrate = Polynomial([-1.0e4, 0.0, +1.0e3])
>>> print heatrate(2.0)
-6000
```

Objects of class `Polynomial` behave like functions of one variable. The one defined here evaluates

$$1000t^2 - 10000. \tag{2}$$

Once the function of time is defined, simply set the wall heat flow Q to this function:

```
>>> wall1.set(Q = heatrate)
```

Note that the direction for positive heat flow depends on the order of the reactors in the argument list to the `Wall` constructor. Positive heat flow is always from the first reactor to the second one, and negative heat flow corresponds to heat flow in the opposite direction.

6.2 Volume Change

A wall may also move in response to a pressure difference, or according to a prescribed displacement rate. Walls have a parameter K, defined such that the rate of volume change of the reactors that is due to motion of the wall is

$$\dot{V}_1 = -\dot{V}_2 = KA(P_1 - P_2). \tag{3}$$

The default setting is $K = 0$ (rigid wall), but this may be changed at any time.

```
>>> wall1.set(K = 10.0)
```

It is also possible the specify the rate of volume change as a function of time.

```
>>> vrate = Polynomial([5.0, -1.0, 10.0])
>>> wall1.set(Vdot = vrate)
```

7 An Example

An example that makes use of all of the features discuss so far is listed below. This script simulates the following situation. A closed cylinder is divided into two parts by a piston. One side s filled with argon at 20 atm, and the other with a combustible methane/air mixture at 0.1 atm. At t = 0 the piston is released and begins to move, compressing and heating the methane/air mixture, which eventually detonates. At the same time, the argon cools. The methane/air mixture also loses heat to the environment.

```
from Cantera import *
from Cantera.Reactor import *

gri3 = GRI30()
gri3.setState_TPX(1000.0, 20.0*OneAtm, 'AR:1')

r1   = Reactor(gri3)
env  = Reservoir(Air())

gri3.setState_TPX(500.0, 0.1*OneAtm, 'CH4:1.1, O2:2, N2:7.52')
r2 = Reactor(gri3)

# add a flexible wall (a piston) between r2 and r1
w = Wall(r2, r1)
w.set(area = 2.0, K=1.e-4)

# heat loss to the environment
w2 = Wall(r1, env)
w2.set(area = 0.5, U=100.0)

time = 0.0
f = open('piston.csv','w')
for n in range(300):
    time += 4.e-5
    r1.advance(time)
    r2.advance(time)
    writeCSV(f, [r2.time(), r2.temperature(), r2.pressure(), r2.volume(),
                 r1.temperature(), r1.pressure(), r1.volume()])
f.close()
```

The temperature and pressure in reactor 2 are shown in Fig. 3

8 Open Systems

Multiple reactors may be connected together to form a network. The flow rate between reactors is regulated by a flow control device. A `MassFlowController` provides a constant mass flow rate, independent of upstream or downstream conditions. A `Valve` provides a mass flow rate proportional to the pressure drop, or zero if the pressure rises.

A CSTR can be implemented as follows. The mass flow rate, valve coefficient, and downstream pressure can be adjusted to give the desired residence time and reactor pressure.

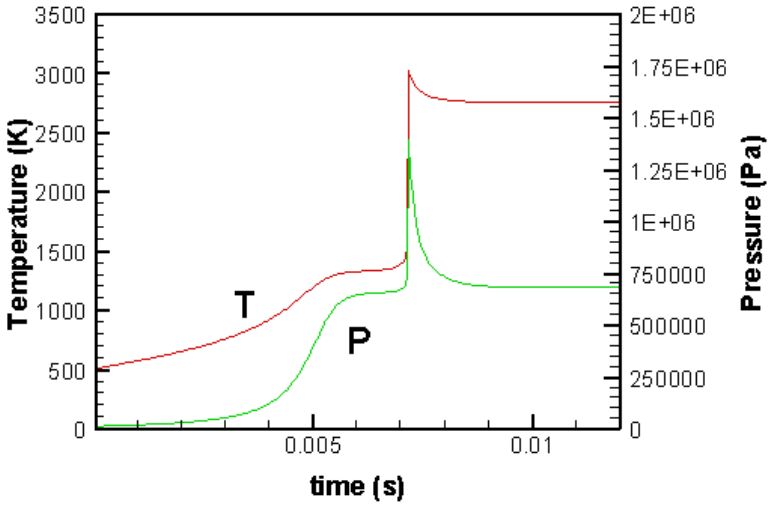

Figure 3: Temperature and pressure vs. time.

```
>>> gas.setState_TPX(1500.0, 2.0*OneAtm, 'H2:1,O2:1,AR:6')
>>> r1 = Reservoir(gas)
>>> reactor = Reactor(gas)
>>> gas.setState_TPX(300.0, 1.0*OneAtm, 'AR:1')
>>> r2 = Reservoir(gas)
>>> m1 = MassFlowController(r1, reactor)
>>> m1.setMassFlowRate(1.e-4)
>>> v1 = Valve(reactor, r2)
>>> v1.setValveCoeff(1.0)
>>> reactor.advance(100.0)
```

With multiple reactors, method `advance` must be called for each reactor. Since the state of each reactor is integrated in time separately, keeping the properties of the other reactors fixed at their values after the last call to `advance`, specifying too large a time interval in `advance` may result in instabilities. In this case, use a smaller interval.

If method `step` is used, care must be taken to keep the reactors synchonized in time.

Class `CSTR` can also be used. For more information, type

```
>>> from Cantera import CSTR
>>> help(CSTR)
```