# CanSen Documentation

***Release 1.1.0***

**Bryan W. Weber**

April 13, 2014

CanSen is a Python script that provides a SENKIN-like wrapper around the open-source Cantera package. The motivation for this project is to ease the transition from SENKIN to using Cantera. Many researchers have knowledge of how to build SENKIN input files, and many may have SENKIN input files available that they use. CanSen enables the use of SENKIN-formatted input files with Cantera.

CanSen can be used with any version of Python >= 2.6.

CanSen is hosted at GitHub, if you are interested in the source code and development. Please report any bugs there.

CHAPTER

ONE

# CONTENTS

## 1.1 CanSen Installation Guide

CanSen can be installed on any platform that supports Python and Cantera. This guide contains instructions for Windows and Ubuntu 12.04.

CanSen has several dependencies, including:

- Cantera
- NumPy
- PyTables

### 1.1.1 Windows

Python can be downloaded and installed from the Python.org page. Installation instructions for Cantera on Windows can be found on the Google code page. Make sure to download the correct version for your Python and 32- or 64-bit Windows, depending on which version your OS is. If NumPy is not already installed, download the proper version from the Windows Binaries page. From the same page, download the installer for PyTables and its dependency numexpr.

Then, download the most recent release of CanSen from GitHub. Unzip the zip file, and you're ready to go!

Alternatively, you can use Git to download the developer version. **WARNING:** The developer version of CanSen is not guaranteed to be working at any given commit. Proceed with caution.:

```
git clone git://github.com/bryanwweber/CanSen.git
```

will download the repository into a folder called `CanSen`.

### 1.1.2 Ubuntu

These instructions are for Ubuntu 12.04, but should work with only slight changes for most major releases of Linux. Optionally, download Python 3 from the apt repositories. At the same time, it is good to download some other dependencies:

```
sudo apt-get install python3 python3-dev libhdf5-serial-dev
```

Then, install `distribute` and `pip`:

```
wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py \
-O - | sudo python3.2
sudo easy_install-3.2 pip

or

wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py \
-O - | sudo python
sudo easy_install pip
```

Finally, with `pip` installed, install NumPy, Cython, numexpr, and finally, PyTables:

```
sudo pip-3.2 install numpy cython numexpr
sudo pip-3.2 install pytables

or

sudo pip install numpy cython numexpr
pip install pytables
```

Instructions for more complicated cases can be found on the PyTables documentation.

Compilation/installation instructions for Cantera can be found in the Cantera documentation.

Finally, get the most recent stable release of CanSen from GitHub. Untar the tarball, and you're ready to go!

```
tar -xzf CanSen-X.Y.Z.tar.gz
```

Alternatively, you can use Git to download the developer version. **WARNING:** The developer version of CanSen is not guaranteed to be working at any given commit. Proceed with caution.:

```
git clone git://github.com/bryanwweber/CanSen.git
```

will download the repository into a folder called `CanSen`.

## 1.2 Usage

The following are instructions for usage of CanSen.

### 1.2.1 Windows

CanSen can be run from the command line (`cmd.exe`) or from within IPython. From the command line, change into the directory with the CanSen script, and run:

```
py cansen.py [options]
```

In IPython, type:

```
In [1]: %run cansen.py [options]
```

### 1.2.2 Ubuntu

CanSen can be run either as an executable, or as a script with Python (2 or 3) or IPython. To run as an executable, change to the directory where CanSen is located, add the execute bit to cansen.py, and run:

```
chmod +x cansen.py
./cansen.py [options]
```

To run as a script, change to the directory where CanSen is located and:

```
python3 cansen.py [options]

or

python cansen.py [options]
```

Or, in IPython:

```
In [1]: %run cansen.py [options]
```

### 1.2.3 Options

All of the previous commands have shown `[options]` to indicate where command line options should be specified. The following options are available, and can also be seen by using the `-h` or `--help` options:

```
-i:
   Specify the simulation input file in SENKIN format. Required.
-o:
   Specify the text output file. Optional, default: ``output.out``
-x:
   Specify the binary save output file. Optional, default:
   ``save.hdf``
-c:
   Specify the chemistry input file, in either CHEMKIN, Cantera
   CTI or CTML format. Optional, default: ``chem.xml``
-d:
   Specify the thermodyanmic database. Optional if the
   thermodyanmic database is specified in the chemistry input
   file. Otherwise, required.
--convert:
   Convert the input mechanism to CTI format and quit. If
   ``--convert`` is specified, the SENKIN input file is optional.
-h, --help:
   Print this help message and quit.
```

## 1.3 Supported Input Keywords

The following is a list of the currently supported keywords in the input file. Keywords that include "CanSen specific keyword" should be placed after the 'END' keyword to maintain SENKIN compatibility, although CanSen has no preference for the order.

ADD ATLS ATOL BORE CMPR CONP CONT CONV COTV CPROD
CRAD DEG0 DELT DTIGN DTSV END EQUI FUEL ICEN IGNBREAK
LOLR OXID PRES REAC RODL RPM RTLS RTOL SENS STPT
STROKE TEMP TIME TLIM TPRO TTIM VOL VOLC VOLD VPRO
VTIM

---

ADD: Mole fractions of species that should be included in the initial composition but excluded from the calculation of the equivalence ratio. Only valid when the equivalence ratio option is used to specify the composition. See CPROD, EQUI, FUEL, OXID, REAC.

Example:

```
ADD Ar 0.1
```

---

ATLS: Absolute tolerance of the accuracy of the sensitivity coefficients. Optional keyword, default: 1E-06

Example:

```
ATLS 1E-06
```

---

ATOL: Absolute tolerance of the accuracy of the solution. Should be set smaller than the smallest meaningful species mass fraction. Optional keyword, default: 1E-20

Example:

```
ATOL 1E-20
```

---

BORE: CanSen specific keyword. Bore diameter of the engine cylinder. Units: cm.

Example:

```
BORE 1.0
```

---

CMPR: Specify the compression ratio for the internal combustion engine model. Defined as the maximum total volume in the cylinder divided by the clearance volume. See the *documentation*. See also: VOLC, VOLD.

Example:

```
CMPR 10.0
```

---

CONP: Solve a constant pressure reactor with the energy equation on. One of CONP, CONT, CONV, COTV, ICEN, TPRO, TTIM, VPRO, or VTIM must be specified.

---

CONT: Solve a constant pressure reactor with the energy equation off. One of CONP, CONT, CONV, COTV, ICEN, TPRO, TTIM, VPRO, or VTIM must be specified.

---

CONV: Solve a constant volume reactor with the energy equation on. One of CONP, CONT, CONV, COTV, ICEN, TPRO, TTIM, VPRO, or VTIM must be specified.

---

COTV: Solve a constant volume reactor with the energy equation off. One of CONP, CONT, CONV, COTV, ICEN, TPRO, TTIM, VPRO, or VTIM must be specified.

---

CPROD: Complete products of stoichiometric combustion for the given fuel and oxidizer compositions. Only valid when the equivalence ratio option is used to specify the composition. All of the elements specified in the FUEL and OXID must be included in the set of species specified in CPROD. See ADD, EQUI, FUEL, OXID, REAC.

Example:

```
CPROD H2O
CPROD CO2
```

---

CRAD: CanSen specific keyword. Specify the crank radius. Units: cm.

Example:

```
CRAD 3.5
```

---

DEG0: Specify the initial crank angle of the simulation. Units: degrees. Default: 180 deg.

Example:

```
DEG0 180
```

---

DELT: Time interval for printing to the screen and the text output file. Optional keyword, default: TIME/100.Units: seconds.

Example:

```
DELT 1E-03
```

---

DTIGN: Temperature threshold used to determine the ignition delay. Ignition temperature is the initial temperature TEMP plus this value. Will be ignored for cases with the energy equation turned off. If both DTIGN and TLIM are specified, TLIM will override DTIGN. See TLIM. Optional keyword, default: 400. Units: K.

Example:

```
DTIGN 400
```

---

DTSV: Time interval for saving to the binary save file. Values are stored at the nearest time step to the save time interval. Optional keyword, by default, all time points are saved to the binary save file. Units: seconds.

Example:

```
DTSV 1E-05
```

---

END: Signifies the end of the input file in SENKIN. It is included in CanSen for compatibility with SENKIN input files, but does not do anything. Any CanSen specific keywords can be placed after END and the same input file can be used with SENKIN with no changes.

---

EQUI: Equivalence ratio desired for the initial mixture. If EQUI is specified, all of CPROD, FUEL, and OXID also must be specified, and ADD can be optionally specified. If EQUI is not specified, the reactants must be specified with REAC. See ADD, CPROD, FUEL, OXID, REAC.

Example:

```
EQUI 1.0
```

---

FUEL: Relative mole fractions of components in the fuel mixture for equivalence ratio calculations. The sum of the fuel mole fractions should be 1.0; if they are not, they will be normalized and a warning message will be printed. If EQUI is specified, FUEL must be specified. See ADD, CPROD, EQUI, OXID, REAC.

Example:

```
FUEL CH4 1.0
```

---

ICEN: Specify the internal combustion engine model be used. See *the documentation for the model* for information on the derivation. See also BORE, CMPR, CRAD, DEG0, LOLR, RODL, RPM, STROKE, VOLD, and VOLC. One of CONP, CONT, CONV, COTV, ICEN, TPRO, TTIM, VPRO, or VTIM must be specified.

---

IGNBREAK: CanSen specific keyword. Indicates that the simulation should exit when ignition is encountered, instead of continuing until the end time TIME is reached. The criterion for ignition is specified by DTIGN or TLIM. Optional keyword.

---

LOLR: Specify the ratio of the connecting rod length, $\ell$, to the crank radius, $a$. See RODL, CRAD.

Example:

```
LOLR 3.5
```

---

OXID: Relative mole fractions of components in the oxidizer mixture for equivalence ratio calculations. The sum of the oxidizer mole fractions should be 1.0; if they are not, they will be normalized and a warning message will be printed. If EQUI is specified, OXID must be specified. See ADD, CPROD, EQUI, FUEL, REAC.

Example:

```
OXID O2 1.0
OXID N2 3.76
```

---

PRES: Initial reactor pressure. Required keyword. Units: atmospheres.

Example:

```
PRES 1.0
```

---

REAC: Initial mole fraction of a reactant gas in the reactor. Required keyword if EQUI is not specified; however, only one of REAC or EQUI may be specified. If the mole fractions of the components given on REAC lines do not sum to 1.0, they will be normalized and a warning message will be printed.

Example:

```
REAC CH4 1.0
REAC O2 1.0
REAC N2 3.76
```

---

RODL: CanSen specific keyword. Specify the connecting rod length, $\ell$. Units: cm.

Example:

```
RODL 5.0
```

---

RPM: Specify the rotation rate of the engine in revolutions per minute.

Example:

```
RPM 1500
```

---

RTLS: Relative tolerance of the accuracy of the sensitivity coefficients. Optional keyword, default: 1E-04

Example:

```
RTLS 1E-04
```

---

RTOL: Relative tolerance of the accuracy of the solution. Can be interpreted roughly as the number of significant digits expected in the solution. Optional keyword, default: 1E-08

Example:

```
RTOL 1E-08
```

---

SENS: Calculate sensitivity coefficients for the solution variables. The sensitivity coefficients are stored in a 2-D array, with dimensions of (number of solution variables, number of reactions). For CONV, COTV, VPRO and VTIM cases, the order of the sensitivity coefficients (i.e. the rows) is:

```
- 0  - mass
- 1  - volume
- 2  - temperature
- 3+ mass fractions of the species
```

For CONP, CONT, TPRO, and TTIM cases, the order of the sensitivity coefficients (i.e. the rows) is

```
- 0  - mass
- 1  - temperature
- 2+ - mass fractions of the species
```

---

STPT: Maximum internal time step for the solver. Optional keyword. If any of DELT, DTSV, or STPT are specified, the minimum of these is used as the maximum internal time step. Otherwise, the default maximum time step is the end time TIME/100.

Example:

```
STPT 1E-5
```

---

STROKE: CanSen specific keyword. Specify the stroke length of the engine, $L$. Units: cm.

Example:

```
STROKE 7.0
```

---

TEMP: Initial reactor temperature. Required keyword. Units: K.

Example:

```
TEMP 800
```

---

TIME: End time for the integration. Unless, IGNBREAK is specified and its condition satisfied, the solver will integrate until TIME is reached. Required keyword. Units: seconds.

Example:

```
TIME 1E-03
```

---

TLIM: Ignition temperature. Ignition is considered to have occurred when this temperature is exceeded. If both DTIGN and TLIM are specified, TLIM overrides DTIGN. Optional keyword, default: TEMP + 400. Units: K.

Example:

```
TLIM 1200
```

---

TPRO: Warning: TPRO is broken in CanSen v1.1 due to incompatibilites with Cantera 2.1. Specify the reactor temperature as a function of time. Multiple invocations of this keyword build a profile of the temperature over the given times. This profile is linearly interpolated to set the reactor temperature at any solver time step. When the end time of the profile is exceeded, the temperature remains constant at the last specified value. One of CONP, CONT, CONV, COTV, ICEN, TPRO, TTIM, VPRO, or VTIM must be specified. Units: seconds, K.

Example:

```
TPRO 0.0 800
TPRO 0.1 900
```

---

TTIM: Warning: TTIM is broken in CanSen v1.1 due to incompatibilites with Cantera 2.1. Specify the reactor temperature as a user-provided function of time. To use this keyword, the user must edit the TemperatureFunctionTime class in the user_routines file. Any parameters to be read from external files should be loaded in the __init__ method so that they are not read on every time step. The parameters should be stored in the self instance of the class so that they can be accessed in the __call__ method. The __call__ method should contain the actual calculation and return of the temperature given the input time.One of CONP, CONT, CONV, COTV, ICEN, TPRO, TTIM, VPRO, or VTIM must be specified. Units: K.

---

VOL: Initial volume of the reactor. Optional keyword, default: 1E6 cm**3. Units: cm**3.

Example:

```
VOL 1.0
```

---

VOLC: Specify the clearance volume, $V_c$. Units: cm**3. See CMPR, VOLD.

Example:

```
VOLC 1.0
```

---

VOLD: Specify the swept or displaced volume, $V_d$. Units: cm**3. See CMPR, VOLC.

Example:

```
VOLD 10.0
```

---

VPRO: Specify the reactor volume as a function of time. Multiple invocations of this keyword build a profile of the volume over the given times. This profile is linearly interpolated to set the reactor volume at any solver time step. When the end time of the profile is exceeded, the volume remains constant at the last specified value. One of CONP, CONT, CONV, COTV, ICEN, TPRO, TTIM, VPRO, or VTIM must be specified. Units: seconds, m**3.

Example:

```
VPRO 0.0 1E-5
VPRO 0.1 1E-6
```

---

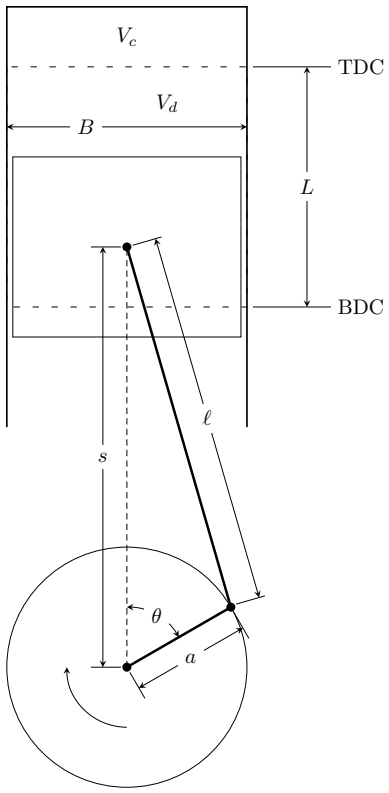
VTIM: Specify the reactor volume as a user-provided function of time. To use this keyword, the user must edit the VolumeFunctionTime class in the user_routines file. Any parameters to be read from external files should be loaded in the __init__ method so that they are not read on every time step. The parameters should be stored in the self instance of the class so that they can be accessed in the __call__ method. The __call__ method should contain the actual calculation and must return the velocity of the wall given the input time. One of CONP, CONT, CONV, COTV, ICEN, TPRO, TTIM, VPRO, or VTIM must be specified. Units: m/s.

## 1.4 Internal Combustion Engine Model

The internal combustion engine model in CanSen is included to enable simulations of a reciprocating internal combustion engine. The equation of motion for the piston follows from [1], Ch. 2.

Assuming the piston roughly appears as:

[1] John B. Heywood. *Internal Combustion Engine Fundamentals.* New York: McGraw Hill, 1988. Print.




where $s$ is distance from the crank axis to the piston pin axis, $\ell$ is the connecting rod length, $a$ is the crank radius, $\theta$ is the crank angle, with $0^\circ$ at the top of the crank, $L$ is the stroke length, $B$ is the cylinder bore, $V_d$ is the swept, or displacement, volume, $V_c$ is the clearance volume, and TDC and BDC are top dead center and bottom dead center respectively (i.e. the top and bottom of the stroke).

The compression ratio of the cylinder is defined as:

$$r_c = \frac{V_d + V_c}{V_c}$$

The swept volume can either be specified directly, or can be calculated from the cylinder bore and stroke length:

$$V_d = L * \pi * \frac{B^2}{4}$$

The initial volume of the cylinder (the volume at BDC) is:

$$V_0 = V_d + V_c = r_c * V_c = \frac{V_d}{r_c - 1} + V_d$$

---

The distance from the crank center to the piston pin is given by:

$$s = a\cos(\theta) + \sqrt{\ell^2 - a^2\sin^2(\theta)}$$

Cantera expects a moving wall to be given a velocity, so we find the piston velocity by differentiating with respect to time:

$$\frac{ds}{dt} = -a\sin(\theta)\frac{d\theta}{dt} + \frac{a^2\sin(\theta)\cos(\theta)\frac{d\theta}{dt}}{\sqrt{\ell^2 - a^2\sin^2(\theta)}}$$

Defining $\frac{d\theta}{dt}$, the angular velocity of the crank, as $\omega$, and using the definition of the stroke length $L$ and the connecting rod length to crank radius ratio $R$:

$$L = 2a$$
$$R = \frac{\ell}{a}$$

the equation for the velocity can be simplified to:

$$\frac{ds}{dt} = -\omega \frac{L}{2} \sin(\theta) \left[ 1 + \frac{\cos(\theta)}{\sqrt{R^2 - \sin^2(\theta)}} \right]$$

In CanSen, the angular velocity of the crank is input in revolutions per minute, so it must be converted to radians per second:

$$\frac{rad}{s} = RPM * \frac{\pi}{30}$$

By default, the starting crank angle is $180°$, or BDC, $0°$ is TDC and the piston reaches BDC again at $-180°$. The starting crank angle can be set with the *DEG0* keyword, so $\theta$ is calculated as a function of time by:

$$\theta = \frac{DEG0 * \pi}{180} - \omega * t$$

## 1.5 Postprocessing

CanSen saves the solution information to a binary save file in a standard format called HDF5. Many programming and scripting languages have interfaces for HDF5 files, including C++, MATLAB, Fortran 90 and Python. Notably, these are all of the interfaces that Cantera supports. The Python interface will be demonstrated in this tutorial, but the structure of the data and thus the main content of this tutorial will remain the same for all of the interfaces.

There are several Python interfaces for HDF5, but the one we will be using is called PyTables. The documentation for PyTables can be found on their GitHub page.

Note that on the following lines, the >>> indicates that you should type the text at a Python prompt, not including the >>>. First, we will import the necessary libraries:

```
>>> import tables
>>> import cantera as ct
```

If either of these don't work, make sure that PyTables and Cantera are both properly installed.

To print information about the save file, just type the name of its variable

```
>>> save_file
```

The data is saved in the save file with the Table format. Each Row in the Table represents one time step. Each Row further consists of a number of Columns where the data is stored. The Columns can be of arbitrary shape - thus, the entire 2-D sensitivity array is saved in one Column on each time step (i.e. in each row).

The format of the save file is hierarchical. The Table with each time step is stored in a Group, which is stored in the Root. It can be thought of as nested directories, with the Root as the top directory, then the Group, then the Table, like so:

```
Root
    |-Group
          |-Table
```

To access the information in the Table, it should be stored in a variable for quick access. The name of the Group in the save files from CanSen is `reactor`.

```
>>> table = save_file.root.reactor
```

The Table can now be used like any other class instance. In particular, the Table class defines a number of useful functions and attributes, such as `nrows`, which prints the number of rows in the Table.

```
>>> table.nrows
```

PyTables provides a method to iterate over the rows in a table automatically, called `iterrows`. Here we introduce one way to access information in a particular Column in the Table, by using natural name indexing. In this case, we print the value of the time at each time step.

```
>>> for row in table.iterrows():
        print(row['time'])
```

Note that numerical indexing is also supported. The following is equivalent to the above:

```
>>> for row in table.iterrows():
        print(row[0])
```

The information stored by CanSen is written into case-sensitive columns named:

0. `time`
1. `temperature`
2. `pressure`
3. `volume`
4. `massfractions`
5. `sensitivity`

Columns 0-3 have a single value in each row. Column 4 (`massfractions`) contains a vector with length of the number of species in the mechanism. Column 5 is optional and included only if the user requested sensitivity analysis during the simulation. The dimensions of Column 5 are `(n_vars, n_sensitivity_params)`.

In addition to the method of iterating through Rows, entire Columns can be accessed and stored in variables. First, all of the Columns can be stored in a variable.

```
>>> all_cols = table.cols
```

In this method, different Columns are accessed by their numerical index. The first index to `all_cols` gives the row and the second index gives the column number. Remembering that Python is zero-based, to access the mass fractions on the 4th time step, do

```
>>> mass_fracs_4 = all_cols[3][4]
```

Individual Columns can be stored in variables as well. This is done by the natural naming scheme.

```
>>> all_mass_fracs = table.cols.massfractions
```

This stores an instance of the Column class in `all_mass_fracs`. It may be more useful to store the data in a particular column in a variable. To do that, get a slice of the column by using the index and the colon operator. For instance, to store all of the mass fraction data in a variable

```
>>> all_mass_fracs = table.cols.massfractions[:]
```

**Or, to store the fifth through tenth time steps**

```
>>> mass_fracs_5_10 = table.cols.massfractions[4:9]
```

Or, to store every other time step from the sixth through the 20th

```
>>> mass_fracs_alt = table.cols.massfractions[5:19:2]
```

Once the data has been extracted from the save file, we need to actually be able to do something with it. Fortunately, Cantera offers a simple way to do this, by initializing a Solution to the desired conditions.

```
>>> gas = ct.Solution('mech.xml')
>>> for row in table.iterrows():
        gas.TPY = row['temperature'], row['pressure'], row['massfractions']
        print(gas.creation_rates)
```

This will print the creation rates of each species at each time step. Any method or parameter supported by the Solution class can be used to retrieve data at any given time step.

Further information about the PyTables package can be found at http://pytables.github.io/usersguide/index.html and information about Cantera can be found at http://cantera.github.io/docs/sphinx/html/index.html

## 1.6 Code Documentation

### 1.6.1 cansen module

`cansen.`**`main`**(*filenames*, *convert*, *version*)

The main driver function of CanSen.

**Parameters**

- **filenames** – Dictionary of filenames related to the simulation.
- **convert** – Boolean indicating that the user wishes only to convert the input mechanism and quit.
- **version** – Version string of CanSen.

`cansen.`**`cansen`**(*argv*)

CanSen - the SENKIN-like wrapper for Cantera written in Python.

**Usage:**

**-i:** Specify the simulation input file in SENKIN format. Required.

**-o:** Specify the text output file. Optional, default: `output.out`

**-x:** Specify the binary save output file. Optional, default: `save.hdf`

**-c:** Specify the chemistry input file, in either CHEMKIN, Cantera CTI or CTML format. Optional, default: `chem.xml`

**-d:** Specify the thermodyanmic database. Optional if the thermodyanmic database is specified in the chemistry input file. Otherwise, required.

**–convert:** Convert the input mechanism to CTI format and quit. If `--convert` is specified, the SENKIN input file is optional.

**-h, –help:** Print this help message and quit.

### 1.6.2 printer module

**class** printer.**Tee**(*name*, *mode*)
Bases: builtins.object

Write to screen and text output file

Initialize output.

**Parameters**

- **name** – Output file name.
- **mode** – Read/Write mode of the output file.

**close**()
Close output file and restore standard behavior

### 1.6.3 profiles module

**class** profiles.**VolumeProfile**(*keywords*)
Bases: builtins.object

Set the velocity of the piston by using a user specified volume profile. The initialization and calling of this class are handled by the Func1 interface of Cantera. Used with the input keyword *VPRO*

Set the initial values of the arrays from the input keywords.

The time and volume are read from the input file and stored in the keywords dictionary. The velocity is calculated by assuming a unit area and using the forward difference, calculated by numpy.diff. This function is only called once when the class is initialized at the beginning of a problem so it is efficient.

**Parameters** **keywords** – Dictionary of keywords read from the input file

**__call__**(*t*)
Return the velocity when called during a time step.

Using linear interpolation, deterimine the velocity at a given input time t.

**Parameters** **t** – Input float, current simulation time.

**class** profiles.**TemperatureProfile**(*keywords*)
Bases: builtins.object

Set the temperature of the reactor by using a user specified temperature profile. The initialization and calling of this class are handled by the Func1 interface of Cantera. Used with the input keyword *TPRO*

Set the initial values of the arrays from the input keywords.

The time and temperature are read from the input file and stored in the keywords dictionary as lists. This function is only called once when the class is initialized at the beginning of a problem so it is efficient.

**__call__**(*t*)
Return the temperature when called during a time step.

Using linear interpolation, determine the temperature at a given input time t.

**Parameters** **t** – Input float, current simulation time.

**class** profiles.**ICEngineProfile**(*keywords*)
Bases: builtins.object

Set the velocity of the wall according to the parameters of a reciprocating engine. The initialization and calling of this class are handled by the Func1 interface of Cantera. Used with the input keyword *ICEN*.

Set the initial values of the engine parameters.

The parameters are read from the input file into the `keywords` dictionary.

**__call__**(*time*)
Return the velocity of the piston when called.

The function for the velocity is given by Heywood. See *Internal Combustion Engine Model*.

**Parameters** **time** – Input float, current simulation time

**class** `profiles.`**PressureProfile**
Bases: `builtins.object`

Dummy class for the pressure profile, to be implemented in CanSen v2.0

### 1.6.4 run_cases module

**class** `run_cases.`**SimulationCase**(*filenames*)
Bases: `builtins.object`

Class that sets up and runs a simulation case.

Initialize the simulation case.

Read the SENKIN-format input file is read into the `keywords` dictionary.

**Parameters** **filenames** – Dictionary containing the relevant file names for this case.

**setup_case**()
Sets up the case to be run. Initializes the `ThermoPhase`, `Reactor`, and ReactorNet according to the values from the input file.

**run_case**()
Actually run the case set up by `setup_case`. Sets binary output file format, then runs the simulation by using `ReactorNet.step(self.tend)`.

**run_simulation**()
Helper function that sequentially sets up the simulation case and runs it. Useful for cases where nothing needs to be changed between the setup and run. See *setup_case* and *run_case*.

**reactor_state_printer**(*state*, *end=False*)
Produce pretty-printed output from the input reactor state.

In this function, we have to explicitly pass in the reactor state instead of using `self.reac` because we might have interpolated to get to the proper time.

**Parameters**

- **state** – Vector of reactor state information.
- **end** – Boolean to tell the printer this is the final print operation.

### 1.6.5 user_routines module

**class** `user_routines.`**VolumeFunctionTime**
Bases: `builtins.object`

Calculate the volume of the reactor as a user specified function of time.

Set up the function to be calculated.

The init function should be used to import any parameters for the volume as a function of time from external files. Do not calculate the volume as a function of time here. Store all of the parameters in the `self` instance. The reason for this is to avoid running the `__init__` function on every time step. See the example below.

Example to load polynomial parameters from a file:

```
# Read the file into the list ''self.params''. The lines of
# the file are read as strings.
with open('file.txt','r') as input_file:
    self.params = input_file.readlines()
# Convert the strings to floats.
self.params = [float(param) for param in self.params]
self.area = 1 # m**2
```

**__call__**(*time*)

Return the velocity of the piston at the given time.

The call function should be where the implementation of the volume as a function of time is done. Cantera expects the velocity to be returned - v = dV/dt * 1/A. Get all of the needed parameters that were stored in the `self` instance. See the example below.

Example to use the previously stored polynomial parameters:

```
volume = (self.params[0] + self.params[1]*time +
    self.params[2]*time**2 + self.params[3]*time**3) # m**3
dvoldt = (self.params[1] + 2*self.params[2]*time +
    3*self.params[3]*time**2) # m**3/s
velocity = dvoldt/self.area # m/s
return velocity
```

**class** user_routines.**TemperatureFunctionTime**

Bases: builtins.object

Calculate the temperature of the reactor as a user specified function of time.

Set up the function to be calculated.

The init function should be used to import any parameters for the temperature as a function of time from external files. Do not calculate the temperature as a function of time here. Store all of the parameters in the `self` instance. The reason for this is to avoid running the `__init__` function on every time step. See the example below.

Example to load polynomial parameters from a file:

```
# Read the file into the list ''self.params''. The lines of
# the file are read as strings.
with open('file.txt','r') as input_file:
    self.params = input_file.readlines()
# Convert the strings to floats.
self.params = [float(param) for param in self.params]
```

**__call__**(*time*)

Return the velocity of the piston at the given time.

The call function should be where the implementation of the temperature as a function of time is done. Get all of the needed parameters that were stored in the `self` instance. See the example below. Note: `None` is not a valid return value, so this function does not work as written.

Example to use the previously stored polynomial parameters:

```
        temperature = (self.params[0] + self.params[1]*time +
            self.params[2]*time**2 + self.params[3]*time**3) # K
        return temperature
```

### 1.6.6 utils module

utils.**convert_mech**(*mech_filename*, *thermo_filename*)
Convert a mechanism and return a string with the filename.

Convert a CHEMKIN format mechanism to the Cantera CTI format using the Cantera built-in script *ck2cti*.

**Parameters**

- **mech_filename** – Filename of the input CHEMKIN format mechanism. The converted CTI file will have the same name, but with *.cti* extension.
- **thermo_filename** – Filename of the thermodynamic database. Optional if the thermodynamic database is present in the mechanism input.

utils.**read_input_file**(*input_filename*)
Read a formatted input file and return a dictionary of keywords.

**Parameters** **input_filename** – Filename of the SENKIN input file.

utils.**cli_parser**(*argv*)
Parse command line interface input.

**Parameters** **argv** – List of command line options.

utils.**reactor_interpolate**(*interp_time*, *state1*, *state2*)
Linearly interpolate the reactor states to the given input time.

**Parameters**

- **interp_time** – Time at which the interpolated values should be calculated
- **state1** – Array of the state information at the previous time step.
- **state2** – Array of the state information at the current time step.

utils.**equivalence_ratio**(*gas*, *eq_ratio*, *fuel*, *oxidizer*, *complete_products*, *additional_species*)
Calculate the mixture mole fractions from the equivalence ratio.

Given the equivalence ratio, fuel mixture, oxidizer mixture, the products of complete combustion, and any additional species for the mixture, return a string containing the mole fractions of the species, suitable for setting the state of the input ThermoPhase.

**Parameters**

- **gas** – Cantera ThermoPhase object containing the desired species.
- **eq_ratio** – Equivalence ratio
- **fuel** – Dictionary of molecules in the fuel mixture and the fraction of each molecule in the fuel mixture
- **oxidizer** – Dictionary of molecules in the oxidizer mixture and the fraction of each molecule in the oxidizer mixture.
- **complete_products** – List of species in the products of complete combustion.
- **additional_species** – Dictionary of molecules that will be added to the mixture. The mole fractions given in this dictionary are as a fraction of the total mixture.

CHAPTER

TWO

# NOTICE

I have tested this software to the best of my abilities. Any user uses this software with the express understanding that there may be bugs, non-working features, and other gremlins that prevent a user from using the software to their specification. If you find problems, please report them in the issue tracker.

ALL USERS ARE ENCOURAGED TO CHECK THEIR RESULTS WITH AN INDEPENDENT PROGRAM! Researchers especially are encouraged *not* to treat this software as a black box. Always remember: Garbage in equals garbage out!

CHAPTER

THREE

# LICENSE

Also available in LICENSE.txt

CHAPTER

FOUR

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

## c



## p



## r



## u