UConnRCMPy: Python-based data analysis for Rapid Compression Machines

Bryan W. Weber[‡*], Chih-Jen Sung[‡]

https://youtu.be/tsjqkIAh8cw

Abstract—The ignition delay of a fuel/air mixture is an important quantity in designing combustion devices, and these data are also used to validate computational kinetic models for combustion. One of the typical experimental devices used to measure the ignition delay is called a Rapid Compression Machine (RCM). This paper presents UConnRCMPy, an open-source Python package to process experimental data from the RCM at the University of Connecticut. Given an experimental measurement, UConnRCMPy computes the thermodynamic conditions in the reaction chamber of the RCM during an experiment along with the ignition delay. UConnRCMPy relies on several packages from the SciPy stack and the broader scientific Python community. UConnRCMPy implements an extensible framework, so that alternative experimental data formats can be incorporated easily. In this way, UConnRCMPy improves the consistency of RCM data processing and enables reproducible analysis of the data.

Index Terms—rapid compression machine, engineering, kinetic models

Introduction

The world relies heavily on combustion to provide energy in useful and clean forms for human consumption; in particular, the transportation sector accounts for nearly 30% of the energy use in the United States and of that, more than 90% is supplied by combustion of fossil fuels [US 16]. Unfortunately, emissions from the combustion of traditional fossil fuels have been implicated in a host of deleterious effects on human health and the environment [ADF+02]. Two methods are being considered to reduce the impact of fossil fuel combustion in transportation on the environment, namely: 1) development of new fuel sources and 2) development of new engine technologies.

The challenge for engineers is that it is not straightforward to combine new fuels with newly designed engines. Employing computer-aided design and modeling of new engines with new fuels will be critical to develop advanced engines to be able to utilize multiple conventional and alternative fuels. The key to this process is the development of accurate and predictive combustion models.

These models of combustion are typically descriptions of the chemical kinetic pathways the hydrocarbon fuel and oxidizer undergo as they break down into carbon dioxide and water. There may be as many as several tens of thousands of pathways in the model for combustion of a particular fuel, with each pathway requiring several parameters to describe its rate. Therefore, it is important to thoroughly validate the operation of the model by comparison to experimental data collected over a wide range of conditions.

One type of data that is particularly relevant for transportation applications is the ignition delay. The ignition delay is a global combustion property depending on the interaction of many of the pathways present in the model. There are several methods to measure the ignition delay at engine-relevant conditions, including shock tubes and rapid compression machines (RCMs).

An RCM is typically designed with one or two pistons that rapidly compress a homogeneous fuel and oxidizer mixture inside a reaction chamber. After the end of compression (EOC), the piston(s) is (are) locked in place, creating a constant volume reaction chamber. The primary diagnostic in most RCM experiments is the pressure measured as a function of time in the reaction chamber. This pressure trace is then processed to extract the ignition delay.

In this paper, the design and operation of a software package to process the pressure data collected from RCMs is described. Our package, called UConnRCMPy [Web16], is designed to analyze the data acquired from the RCM at the University of Connecticut (UConn). Despite the initial focus on data from the UConn RCM, the package is designed to be extensible so that it can be used for data in different formats while providing a consistent interface to the user.

Recognizing that reproducible research is an important goal for the scientific community [Nat16], and that the code used to process experimental data is an important part of reproducing research, the primary goal of UConnRCMPy is to enable consistent, reproducible analysis of RCM data. Thus, UConnRCMPy offers all of the features required to process standard RCM data including:

- Filtering and smoothing the raw voltage generated by the pressure transducer
- Converting the voltage trace into a pressure trace using settings recorded from the RCM
- Processing the pressure trace to determine parameters of interest in reporting the experiments, including the ignition delay and machine-specific effects on the experiment
- Conducting simulations utilizing the experimental information to calculate the temperature during the experiment

Previous software used to analyze RCM data has generally

* Corresponding author: bryan.w.weber@gmail.com
‡ Mechanical Engineering Department, University of Connecticut, Storrs, CT 06269

Copyright © 2016 Bryan W. Weber et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

been undocumented and untested code specific to the researcher conducting the experiments. Moreover, the software typically used to estimate the temperature in the experiments is difficult to integrate with the data processing code. To the best of the authors' knowledge, UConnRCMPy is the first package for analysis of standard RCM data to be presented in detail in the literature, and it tightly integrates the temperature estimation routine into the workflow, reducing errors and inefficiencies.

This paper serves to describe some of the important aspects of RCM data processing, particularly the choices that the operator must make that are rarely documented. In addition, as a complement to the in-source documentation, this paper documents the design choices, interface, and flexibility of UConnRCMPy.

Background

The RCMs at the University of Connecticut have been described extensively elsewhere [DSZM12], [MS07], and will be summarized here for reference. The RCMs use a single pneumatically accelerated and hydraulically decelerated piston. In a typical experiment, the reaction chamber is evacuated to an absolute pressure near 1 Torr, measured by a high-accuracy static pressure transducer. Next, the reactants are filled in to the desired initial pressure (P_0), and a valve on the reaction chamber is closed. Compression is triggered by a digital control circuit. After compression, the piston is held in place to create a constant volume chamber in which reactions proceed. For appropriate combinations of pressure, temperature, and mixture composition, ignition will occur after some delay period. A single compression-delay-ignition sequence is referred to as an experiment or a run. Each experiment is repeated approximately 5 times at the same nominal initial conditions to ensure repeatability of the data, and this set of experiments is referred to in the following as a condition.

The primary diagnostic on the RCM is the reaction chamber pressure, measured by a dynamic pressure transducer (separate from the static transducer used to measure P_0). The pressure trace is processed to determine the quantities of interest, including the pressure and temperature at the EOC, P_C and T_C respectively, and the ignition delay, τ. The ignition delay is typically measured at several values of T_C for a given value of P_C and mixture composition; this is referred to in the following as a data set.

RCM Signal Processing Procedure

Signal measurement

The dynamic pressure transducer outputs a charge signal that is converted to a voltage signal by a charge amplifier with a nominal output of 0 V prior to the start of compression. In addition, the output range of 0 V to 10 V is set by the operator to correspond to a particular pressure range by setting a "scale factor". Typical values for the scale factor range between 10 bar/V and 100 bar/V.

The voltage output from the charge amplifier is digitized by a hardware data acquisition system (DAQ) and recorded into a plain text file by a LabView Virtual Instrument. The voltage is sampled at a rate chosen by the operator, typically between 50 kHz and 100 kHz. This provides sufficient resolution for events on the order of milliseconds; the typical ignition delay measured with this RCM approximately ranges from 5 ms to 100 ms.

Figure 1 shows a typical voltage trace measured from the RCM at UConn. Several features are apparent from this figure. First, the compression stroke takes approximately 30 ms to 40 ms and approximately 50% of the pressure rise occurs in the last 5 ms of compression. Second, there is a slow pressure decrease after the EOC due to heat transfer from the reactants to the relatively colder chamber walls. Third, after some delay period there is a spike in the pressure corresponding to rapid heat release due to combustion. Finally, the signal can be somewhat noisy, requiring filtering and/or smoothing to produce a useful pressure trace.

Filtering and Smoothing

In the current version of UConnRCMPy [Web16], the voltage is filtered using a low-pass filter with a cutoff frequency of 10 kHz. The filter is constructed using the `firwin()` function from the `signals` module of SciPy [JOPosh] with the Blackman window [BT58], [OSB99] and a filter order of $2^{14}-1$. The cutoff frequency, window type, and filter order were determined empirically, based on Fig. 2. Methods to select a cutoff frequency that optimizes the signal-to-noise ratio are currently being investigated.

After filtering, the signal is smoothed by a moving average filter with a width of 21 points. This width was selected empirically based on Fig. 1 to minimize the deviation of the smoothed voltage from the raw voltage during the ignition, and methods to automatically choose an optimal width are being investigated. It is desired that the signal remain the same length through this operation, but the convolution operation used to apply the moving average zero-pads the first and last 10 points. To avoid a bias in the initial voltage, the first 10 points are set equal to the value of the 11th point; the final 10 points are not important in the rest of the analysis and are ignored. The result of the filtering and smoothing operations is shown on Fig. 1.

Offset Correction and Pressure Calculation

In general, the voltage trace can be converted to a pressure trace by

$$P(t) = F \cdot \overline{V}(t) + P_0 \tag{1}$$

where $\overline{V}(t)$ is the filtered and smoothed voltage trace and F is the scale factor from the charge amplifier. However, as can be seen in Fig. 1b there is a small offset in the initial voltage relative to the nominal value of 0 V. To correct for this offset, it can be subtracted from the voltage trace

$$P(t) = F \cdot \left[\overline{V}(t) - \overline{V}(0)\right] + P_0 \tag{2}$$

where $\overline{V}(0)$ is the initial voltage of the filtered and smoothed signal. Assuming the noise in the signal has an equal probability of being above or below the mean voltage, choosing the initial point (i.e., $\overline{V}(0)$) to set the voltage offset is equivalent to choosing any other point prior to the start of compression. The result is a vector of pressure values that must be further processed to determine the time of the EOC and the ignition delay.

Finding the EOC

In the current version of UConnRCMPy [Web16], the EOC is determined by finding the local maximum of the pressure prior to ignition. This is done by searching backwards in time from the global maximum pressure in the pressure trace (typically, the global maximum of the pressure is due to ignition) until a minimum in the pressure is reached. Since the precise time of the minimum is not important for this method, the search is done by comparing the pressure at a given index i to the pressure at point $i-50$, starting with the index of the global maximum pressure. The comparison is not made to the adjacent point to avoid the influence of noise. If $P(i) \geq P(i-50)$, the index is decremented

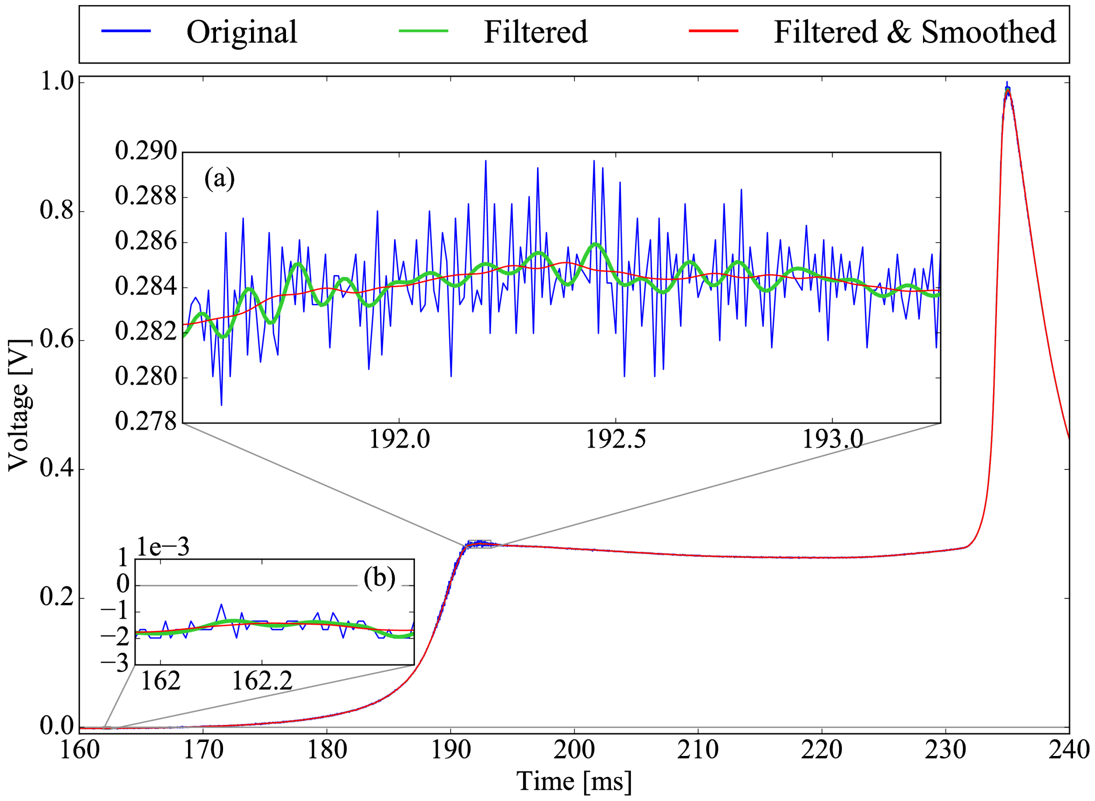

***Fig. 1:** Raw voltage trace and the voltage trace after filtering and smoothing from a typical RCM experiment. Note that the voltage in the figure varies from 0 V to 1 V because the scale factor is 100 bar/V and the maximum pressure for this case is near 100 bar. (a): Close up of the time around the EOC, demonstrating the fidelity of the smoothed and filtered signal with the original signal. (b): Close up of the time before the start of compression, demonstrating the offset of the initial voltage slightly below 0 V.*

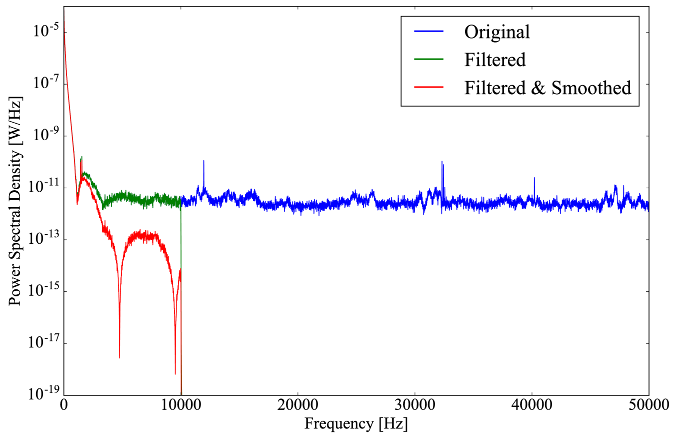

***Fig. 2:** Power spectral density profiles of the original, filtered, and filtered and smoothed signals, showing the peaks of noise above 10 kHz.*

and the process is repeated until $P(i) < P(i-50)$. This value of i is approximately at the minimum of pressure prior to ignition, so the maximum of the pressure in points to the left of the minimum will be the EOC.

This method is generally robust, but it fails when there is no minimum in the pressure between the EOC and ignition, or the minimum pressure is very close to the EOC pressure. This may be the case for short ignition delays, on the order of 5 ms or less. In these cases, the comparison offset (which is set to 50 points by default) can be reduced to improve the granularity of the search; if the method still fails, manual intervention is necessary to determine the EOC. In either case, the value of the pressure at the EOC, P_C, is recorded and the time at the EOC is taken to be $t = 0$.

Calculating Ignition Delay

The ignition delay is determined as the time difference between the EOC and the point of ignition. There are several definitions of the point of ignition; the most commonly used in RCM experiments is the inflection point in the pressure trace due to ignition. As before, finding zero crossings of the second time derivative of the pressure to define the inflection point is difficult due to noise; however, finding the maximum of the first derivative is trivial, particularly since the time before and shortly after the EOC can be excluded to avoid the peak in the derivative around the EOC.

In the current version of UConnRCMPy [Web16], the first derivative of the experimental pressure trace is computed by a second-order forward differencing method. The derivative is then smoothed by the moving average algorithm with a width of 151 points. This value for the moving average window was chosen empirically.

For some conditions, the reactants may undergo two distinct stages of ignition. These cases can be distinguished by a pair of peaks in the first time derivative of the pressure. For some two-stage ignition cases, the first-stage pressure rise, and consequently the peak in the derivative, are relatively weak, making it hard to distinguish the peak due to ignition from the background noise.

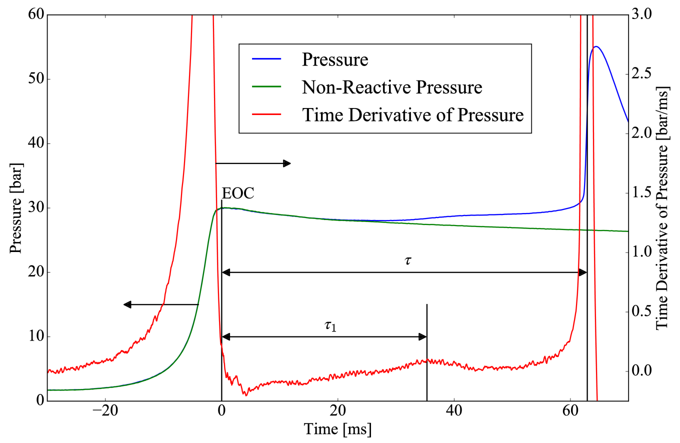

***Fig. 3:** Illustration of the definition of the ignition delay in a two-stage ignition case.*

This is currently the area requiring the most manual intervention, and one area where significant improvements can be made by refining the differentiation and filtering/smoothing algorithms. An experiment that shows two clear peaks in the derivative is shown in Fig. 3 to demonstrate the definition of the ignition delays.

Calculating the EOC Temperature

The final parameter of interest presently is the EOC temperature, T_C. This temperature is often used as the reference temperature when reporting ignition delays. In general, it is difficult to measure the temperature as a function of time in the reaction chamber of the RCM, so methods to estimate the temperature from the pressure trace are generally used.

The law of conservation of energy written for the ideal gases in the reaction chamber is:

$$c_v \frac{dT}{dt} = -P\frac{dv}{dt} - \sum_k u_k \frac{dY_k}{dt} \tag{3}$$

where c_v is the specific heat at constant volume of the mixture, v is the specific volume, u_k and Y_k are the specific internal energy and mass fraction of the species k, and t is time. For a constant-area piston, the rate of change of the volume is equal to the piston velocity. In UConnRCMPy, Eq. 3 is integrated by Cantera [GMS16].

In Cantera, intensive thermodynamic information about the system is stored in an instance of the `Solution` class. The `Solution` classes used in this study model simple, compressible systems and require two independent properties, plus the composition, to fix the state. The two properties must be intensive (i.e., not dependent on system size), and are typically chosen from the pressure, temperature, and density. The thermodynamic information for each species is read from a file in the CTI format, described in the Cantera documentation [GMS16], when a `Solution` instance is created.

In addition to evaluating thermodynamic data, Cantera [GMS16] contains several objects used to model homogeneous reacting systems; the two used in UConnRCMPy are the `Reservoir` and the `IdealGasReactor`, which are subclasses of the generic `Reactor` class. A `Solution` object is installed in each `Reactor` subclass instance to manage the state information and evaluate thermodynamic properties. The difference between the `Reservoir` and the `IdealGasReactor` is simply that the state (i.e., the pressure, temperature, and chemical composition) of the `Solution` in a `Reservoir` is fixed.

Integrating Eq. 3 requires knowledge of the volume of the reaction chamber as a function of time. To calculate the volume as a function of time, it is assumed that there is a core of gas in the reaction chamber that undergoes an isentropic compression [LH98]. Furthermore, it is assumed that there is negligible reactant consumption during the compression stroke.

Constructing the volume trace is triggered by the user by running the `create_volume_trace()` method that implements the following procedure. A Cantera `Solution` object is initialized at the initial temperature, pressure, and composition of the reaction chamber. After initialization, UConnRCMPy stores the initial mass-specific entropy (s_0) and density (ρ_0). The initial volume is arbitrarily taken to be $V_0 = 1.0\,\mathrm{m}^3$. The initial volume used in constructing the volume trace is arbitrary provided that the same value is used for the initial volume in the simulations described below. However, extensive quantities such as the total heat release during ignition cannot be compared to experimental values.

The measured pressure at each point in the pressure trace (P_i) is used with the previously recorded initial entropy (s_0) to set the state of the `Solution` object sequentially. At each point, the volume is computed by applying the ideal gas law:

$$V_i = V_0 \frac{\rho_0}{\rho_i} \tag{4}$$

where ρ_i is the density at each point computed by the Cantera `Solution`. This procedure effects a constant composition isentropic compression process.

Once the volume trace has been generated, it can be utilized in the `IdealGasReactor` and the solution of Eq. 3 by installing an instance of the `Wall` class. `Walls` must be installed between instances of `Reactors`, so in UConnRCMPy a `Wall` is installed between the `IdealGasReactor` that represents the reaction chamber and an instance of the `Reservoir` class. By specifying the velocity of the `Wall` using the volume trace, the `IdealGasReactor` will proceed through the same states as the reaction chamber in the experiment. The velocity of the `Wall` is specified by using an instance of the `VolumeProfile` class from the CanSen software [Web15], which computes the first forward difference of the volume as a function of time.

Finally, the `IdealGasReactor` is installed into an instance of `ReactorNet` from Cantera [GMS16]. The `ReactorNet` implements the interface to the solver CVODES. CVODES is an adaptive-time-stepping solver, distributed as part of the SUNDIALS suite [HBG+05].

Two simulations can be triggered by the user that utilize this procedure. In the first, the multiplier for all the reaction rates is set to zero, to simulate a constant composition (non-reactive) process. In the second, the reactions are allowed to proceed as normal. Only the non-reactive simulation is necessary to determine T_C, which is defined as the simulated temperature at the EOC time.

When a reactive simulation is conducted, the user must compare the temperature traces from the two simulations to verify that the inclusion of the reactions does not change T_C, validating the assumption of adiabatic, constant composition compression. Although including reactions during the compression stroke does not affect the value of T_C, it does allow for the buildup of a small pool of radicals that can affect processes after the EOC [MCSD08]. Thus, it is critical to include reactions during the compression stroke when conducting simulations to compare a kinetic model to experimental results.

Simulating Post-EOC Processes

As can be seen in Fig. 3, the pressure decreases after the EOC due to heat transfer from the higher temperature reactants to the reaction chamber walls. This process is specific to the machine that carried out the experiments, and to the conditions under which the experiment was conducted. Therefore, the rate of pressure decrease should be modeled and included in simulations that compare predicted ignition delays to the experimental values.

To conduct this modeling, a non-reactive experiment is conducted, where O_2 in the oxidizer is replaced with N_2 to maintain a similar specific heat ratio but suppress the oxidation reactions that lead to ignition. The pressure trace from this non-reactive experiment should closely match that from the reactive experiment during the compression stroke, further validating the assumption of adiabatic, constant composition compression. Furthermore, the non-reactive pressure trace should closely match the reactive pressure trace after the EOC until exothermic reactions cause the pressure in the reactive experiment to begin to increase.

To apply the effect of the post-compression heat loss into the simulations, the reaction chamber is modeled as undergoing an adiabatic volume expansion. Since the post-compression time is modeled as an isentropic expansion, the same procedure is used as in the computation of T_C to compute a volume trace for the post-EOC time. The only difference is that the non-reactive pressure trace is used after the EOC instead of the reactive pressure trace. Once the volume trace is generated, it can be applied to a simulation by concatenating the volume trace of the compression stroke and the post-EOC volume trace together and following the procedure outlined in Calculating the EOC Temperature. For consistency, the ignition delay in a reactive simulation is defined in the same manner as in the reactive experiments, as the maxima of the time derivative of the pressure trace. This procedure has been validated experimentally by measuring the temperature in the reaction chamber during and after the compression stroke. The temperature of the reactants was found to be within ± 5 K of the simulated temperature [DUS12], [UDS12].

Implementation of UConnRCMPy

UConnRCMPy is constructed in a hierarchical manner. The main user interface to UConnRCMPy is through the `Condition` class, the highest level of data representation. The `Condition` class contains all of the information pertaining to the experiments at a given condition. The intended use of this class is in an interactive Python interpreter (the author prefers the Jupyter Notebook with an IPython kernel [PG07]). `Condition` also contains all the methods that make up the user interface:

- `add_experiment()`
- `create_volume_trace()`
- `compare_to_sim()`

The usage of these methods will be described in detail in the Usage Example section. In general, the user will conduct several experiments and, using the `add_experiment()` method, will trigger UConnRCMPy to create instances of the `Experiment` class and extract the ignition delay.

All of the information about a particular experimental run is stored in the `Experiment` class. When initialized, the `Experiment` expects an instance of the `pathlib.Path` class; if none is provided, it prompts the user to enter a file name that is expected to be in the current working directory. The file name should point to a tab-delimited plain text file that contains the voltage trace recorded by LabView from one experimental run. Then UConnRCMPy creates an instance of `VoltageTrace`, followed by an instance of `ExperimentalPressureTrace`. The pressure trace from the latter is processed to extract the ignition delay(s).

The lowest level representation of data in UConnRCMPy is the `VoltageTrace` that contains the raw voltage signal and timing recorded from the DAQ, as well as the filtered and smoothed voltage traces. The filtering and smoothing algorithms are implemented as separate methods so they can be reused in other situations and are run automatically when the `VoltageTrace` is initialized.

One step up from the `VoltageTrace` is the `ExperimentalPressureTrace` class. This class consumes a `VoltageTrace` and processes it into a pressure trace, given the multiplication factor from the charge amplifier and the initial pressure. This class also contains methods to compute the derivative of the experimental pressure trace, as discussed in Calculating Ignition Delay.

When all the experiments are conducted and processed, `create_volume_trace()` further processes the experiments to create the volume trace necessary to run the simulations to determine T_C. The actual computation of the volume trace is done by the `VolumeFromPressure` class. First, the volume trace of the pre-EOC portion is generated using the pre-EOC pressure trace, the experimental initial temperature, and an initial volume of $V_0 = 1.0\,\mathrm{m}^3$, as discussed in Calculating the EOC Temperature. A temperature trace is also constructed for the pre-EOC pressure trace using the `TemperatureFromPressure` class.

For the post-EOC volume trace, the initial temperature is estimated as the final value of the temperature trace constructed for the pre-EOC period. Furthermore, the initial volume of the post-EOC volume trace is taken to be the final value of the pre-EOC volume trace, so that although there may be small mismatches in P_C, the volume trace will be consistent.

After generation, `create_volume_trace()` writes the volume trace out to a CSV file so that the volume trace can be used in other software. The reactive pressure trace is also written to a tab-separated file. Before writing, the volume and pressure traces are both downsampled by a factor of 5. This reduces the computational time of a simulation and does not have any effect on the simulated results. `create_volume_trace()` also generates a figure that plots the complete reactive pressure trace, a non-reactive pressure trace generated from the volume trace using the `PressureFromVolume` class, and a linear fit to the constant pressure period prior to the start of compression. This linear fit aids in determining a suitable compression time. Finally, the value of the pressure at the beginning of compression is put on the system clipboard to be pasted into a spreadsheet to record the P_0 used for simulations. This may differ slightly from the P_0 read from the static transducer due to noise in the signal.

The final step is to use the volume trace in a simulation to determine T_C. To begin the simulations, the user calls the `compare_to_sim()` method of the `Condition`. The `compare_to_sim()` method relies on the `run_simulation()` method, which in turn adds instances of the class `Simulation` to the `Condition` instance. Instances

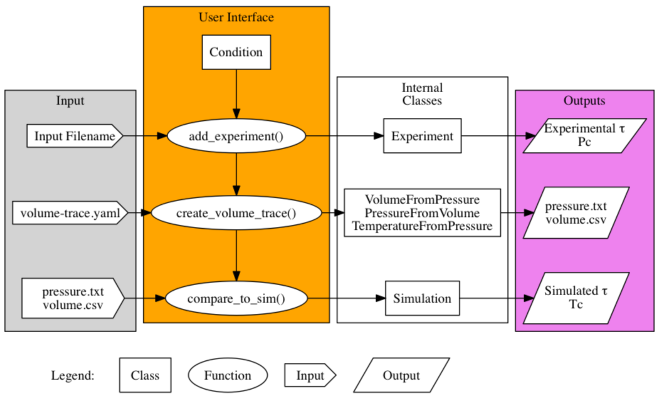

Fig. 4: *Flowchart of information in UConnRCMPy.*

of `Simulation` can represent a reactive or a non-reactive experiment; if either type of simulation has already been added to the `Condition` instance, the user is asked whether they would like to overwrite the existing simulation.

The `Simulation` class sets up the simulation in Cantera and importantly, sets the maximum time step to be the time step used in the volume trace, so that the solver does not take steps larger than the resolution of the velocity. Larger time steps may result in incorrect calculation of the state if the velocity is not properly applied to the reactor. The time, temperature, pressure, and simulated volume are stored in NumPy arrays [vCV11] and the derivative is computed using second order Lagrange polynomials, as suggested by Chapra and Canale [CC10] because the time step is not constant in the simulation. Finally, the calculated value of T_C is placed into the system clipboard. If the reactive simulation is conducted, the overall ignition delay is also copied into the system clipboard. The first stage ignition delay must be found manually because determining peaks in the derivative is currently unreliable, as mentioned in Calculating Ignition Delay for experiments.

The `compare_to_sim()` method also plots the experimental pressure trace and any of the simulated pressure traces that have been generated. If the simulated reactive pressure trace is generated, the time derivative of the pressure is also plotted, where the derivative is scaled by the maximum pressure in the reactive simulation.

The general flow of the user interaction with UConnRCMPy is shown in Fig. 4. The Inputs are required input from the user, while the User Interface are classes and functions called by the user during processing.

UConnRCMPy is documented using standard Python docstrings for functions and classes. The documentation is converted to HTML files by the Sphinx documentation generator [Bra16]. The format of the docstrings conforms to the NumPy docstring format so that the autodoc module of Sphinx can be used. The documentation is available on the web at https://bryanwweber.github.io/UConnRCMPy/.

Usage Example

In the following, two examples of using UConnRCMPy are given, first with the standard interface and second utilizing a slightly modified interface corresponding to a different data format. Both examples assume the user is running in a Jupyter Notebook with an IPython kernel.

Standard Interface

These experiments were conducted with mixtures of propane, oxygen, and nitrogen [DRW+16]. The CTI file necessary to run this example can be found in the Supplementary Material of the work by Dames et al. [DRW+16]. It must be named exactly `species.cti` and placed in the current working directory. Then, the composition of the mixture under consideration must be added to the `initial_state` parameter of the `ideal_gas()` method:

```
ideal_gas(
    name='gas',
    elements=...,
    species=...,
    reactions='all',
    initial_state=state(
        temperature=300.0, pressure=OneAtm,
        mole_fractions=(
            'C3H8:0.0403,O2:0.1008,N2:0.8589')))
```

Ellipses indicate input that was truncated to save space; the truncated input is present in the file available with the work of Dames et al. The initial temperature and pressure are arbitrary, since those are set based on information stored in the filename of the experiment, but the `mole_fractions` must be set to the appropriate values. The condition in this example is for a fuel rich mixture, with a target P_C of 30 bar. The user creates the `Condition`, then conducts a reactive experiment with the RCM and adds the experiment to the `Condition` using the `add_experiment()` method. This method creates an instance of class `Experiment` for each experiment passed in. As each experiment is processed by UConnRCMPy, the information from that run is added to the system clipboard for pasting into some spreadsheet software. In the current version, the information copied is the time of day of the experiment, the initial pressure, the initial temperature, the pressure at the EOC, the overall and first stage ignition delays, an estimate of the EOC temperature, and some information about the compression ratio of the reactor. This process is repeated 5 times to ensure repeatable data is obtained.

```
from uconnrcmpy import Condition
from pathlib import Path
%matplotlib

cond_00_in_02_mm = Condition()
# Conduct reactive experiment #1 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1285t-100x-19-Jul-15-1620.txt'))
# Conduct reactive experiment #2 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1282t-100x-19-Jul-15-1626.txt'))
# Conduct reactive experiment #3 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1282t-100x-19-Jul-15-1633.txt'))
# Conduct reactive experiment #4 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1282t-100x-19-Jul-15-1640.txt'))
# Conduct reactive experiment #5 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1282t-100x-19-Jul-15-1646.txt'))
```

This sequence generates one figure showing all of the experiments together and one figure per experiment comparing the pressure and the time derivative of the pressure. Matplotlib is used for plotting [Hun07]. The plots are optional, and are controlled by passing a boolean keyword argument `plotting` when the `Condition` is initialized. The figures showing each experiment look similar to Fig. 3, but the non-reactive trace is not plotted and the EOC and ignition delays are not labeled.

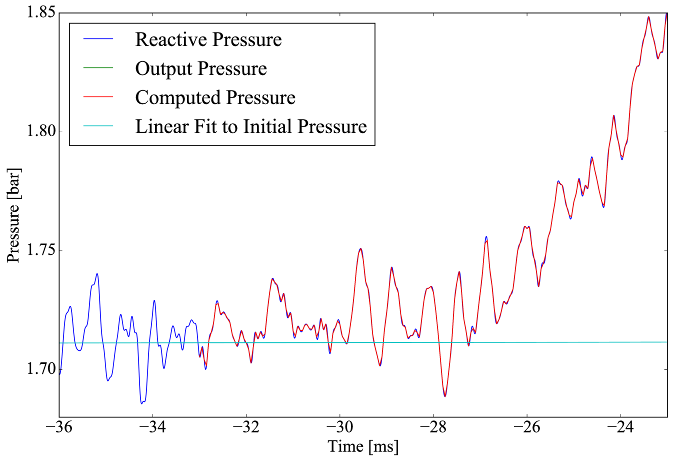

Fig. 5: *Comparison of the reactive pressure trace, the pressure trace output to the text file, the pressure trace computed from the volume trace, and the linear fit to the initial pressure demonstrating the choice of compression time. The dark blue, green, and red lines follow each other nearly exactly after the start of compression, so only the red line is visible. This is the desired result, indicating that the pressure traces agree.*

In general, for a given condition, the user will conduct and process all of the reactive experiments before conducting any non-reactive experiments. Then, the user chooses one of the reactive experiments as the reference experiment for the condition (i.e., the one whose ignition delay(s) and T_C are reported) by inspection of the data in the spreadsheet. The reference experiment is defined as the experimental run whose overall ignition delay is closest to the mean overall ignition delay among the experiments at a given condition. To select the reference experiment, the user puts the file name of the reference experiment into a YAML format file called `volume-trace.yaml` with the key `reacfile`. For this case, the reference experiment is the run that took place at 16:33:

```
reacfile: >
  00_in_02_mm_373K-1282t-100x-19-Jul-15-1633.txt
```

Note that the file must be named exactly `volume-trace.yaml` and it must be located in the current working directory. Once the reference reactive experiment is selected, the user runs non-reactive experiments at the same initial conditions as the reference experiment. The user adds non-reactive experiments to the `Condition` by the same `add_experiment()` method and UConnRCMPy automatically determines whether the experiment is reactive or non-reactive.

```
# Conduct non-reactive experiment #1 on the RCM
cond_00_in_02_mm.add_experiment(Path(
'NR_00_in_02_mm_373K-1278t-100x-19-Jul-15-1652.txt'))
```

UConnRCMPy determines that this is a non-reactive experiment and generates a new figure that compares the current non-reactive case with the reference reactive case as specified in `volume-trace.yaml`. If the user adds a non-reactive experiment before creating the `volume-trace.yaml` file, or if the file referenced in the `reacfile` key is not present in the current working directory, UConnRCMPy throws a `FileNotFound` exception. For this particular example, the pressure traces are shown in Fig. 3. In this case, the non-reactive pressure agrees very well with the reactive pressure and no further experiments are necessary; in principle, any number of non-reactive experiments can be conducted and added to the figure for comparison. Since there is good agreement between the non-reactive and reactive pressure traces, the user adds the non-reactive reference file name to `volume-trace.yaml`.

```
reacfile: >
  00_in_02_mm_373K-1282t-100x-19-Jul-15-1633.txt
nonrfile: >
  NR_00_in_02_mm_373K-1278t-100x-19-Jul-15-1652.txt
```

Then, the user specifies the rest of the parameters in `volume-trace.yaml`, including the compression time and the end times for the reactive and non-reactive experiments. The reactive end time (`reacend`) determines the length of the output pressure trace, while the non-reactive end time (`nonrend`) determines the length of the volume trace. The length of the volume trace is also determined by the compression time (`comptime`), which should be set to a time such that the starting point is before the beginning of the compression. All three times should be specified in milliseconds. `comptime` is determined by comparison with the fit to the initial pressure, as shown in Fig. 5. In this case, the compression has started at approximately $t > -28$ ms. The time prior to that where the pressure appears to stabilize around the initial pressure is approximately $t = -33$ ms, giving a compression time of 33 ms. `reacend` is typically chosen to be shortly after the main pressure peak due to ignition, about 80 ms in this case, and `nonrend` is typically chosen to be 400 ms.

```
reacfile: >
  00_in_02_mm_373K-1282t-100x-19-Jul-15-1633.txt
nonrfile: >
  NR_00_in_02_mm_373K-1278t-100x-19-Jul-15-1652.txt
comptime: 33
nonrend: 400
reacend: 80
```

This sample represents a complete, minimal example of the necessary information in the `volume-trace.yaml` file. In addition, two optional parameters can also be specified in `volume-trace.yaml`. These are offset parameters used to control the precise point where the switch from the reactive pressure trace to the non-reactive pressure trace occurs in the volume trace. These parameters may be necessary if the determination of the EOC does not result in aligned compression strokes for the reactive and non-reactive experiments, but they are not generally necessary.

Once the `volume-trace.yaml` file is completed, the `create_volume_trace()` method can be run. Then, the final step is to conduct the simulations to calculate T_C and the simulated ignition delay. This is done by the user by running the `compare_to_sim()` function. This function takes two optional arguments, `run_reactive()` and `run_nonreactive()`, both of which are booleans. These determine which type of simulation should be conducted; by default, `run_reactive()` is `False` and `run_nonreactive()` is `True` because the reactive simulations may take substantial time (~5 min). There is no restriction on combinations of values for the arguments; either or both may be `True` or `False`.

```
cond_00_in_02_mm.create_volume_trace()
cond_00_in_02_mm.compare_to_sim(
    run_reactive=True,
    run_nonreactive=True,
)
```

At this point, the user has completed one experimental condition. Now, further conditions should be studied, either by changing T_0 or the compression ratio of the RCM to reach a different value of T_C for a given P_C.

Modified Interface

It is also possible to replace parts of the processing interface by using the features of Python to overload class methods. Due to the modular nature of UConnRCMPy, small parts of the interface can be replaced without sacrificing consistent analysis for the critical parts of the code, such as computing the ignition delay. For instance, ongoing work involves processing RCM data collected by several operators of the RCM. Each user has their own file naming strategy that must be parsed for information about the experiment. To process this "alternate" data format, two new classes called `AltCondition` and `AltExperiment` are created that inherit from the `Condition` and `Experiment` classes, respectively. The `AltCondition` class only needs to overload the `add_experiment()` method, to create an `AltExperiment`, instead of a regular `Experiment`.

```python
class AltCondition(Condition):
    def add_experiment(self, file_name=None):
        exp = AltExperiment(file_name)
        # Omit the plotting code...
```

Then, the `AltExperiment` overloads the `parse_file_name()` method of the `Experiment` class to parse the alternate format. The user must make sure the new `parse_file_name()` method returns the expected values as defined in the docstring for the original `parse_file_name()` method, or else overload other methods that consume the file name information.

```python
class AltExperiment(Experiment):
    def parse_file_name(self, file_path):
        # Parse the file name for information...
        return file_name_information
```

In this way, consistent definitions for important research quantities can be used, while providing flexibility in the data format and naming conventions.

Conclusions and Future Work

UConnRCMPy provides a framework to enable consistent analysis of RCM data. Because it is open source and extensible, UConnRCMPy can help to ensure that RCM data in the community can be analyzed in a reproducible manner; in addition, it can be easily modified and used for data in any format. In this sense, UConnRCMPy can be used more generally to process any RCM experiments where the ignition delay is the primary output.

Future plans for UConnRCMPy include the development of a robust test suite to prevent regressions and document correct usage of the framework, as well as the development of a method to determine the optimal cutoff frequency in the filtering algorithm.

Acknowledgements

This paper is based on material supported by the National Science Foundation under Grant No. CBET-1402231.

References

[ADF+02] Maureen D Avakian, Barry Dellinger, Heidelore Fiedler, Brian Gullet, Catherine Koshland, Stellan Marklund, Günter Oberdörster, Stephen Safe, Adel Sarofim, Kirk R Smith, David Schwartz, and William A Suk. The origin, fate, and health effects of combustion by-products: a research framework. *Environmental Health Perspectives*, 110(11):1155–1162, November 2002. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1241073&tool=pmcentrez&rendertype=abstract.

[Bra16] Georg Brandl. Overview — Sphinx 1.4.1 documentation, 2016. URL: http://www.sphinx-doc.org/en/stable/.

[BT58] Ralph Beebe Blackman and John Wilder Tukey. *The Measurement of Power Spectra*. Dover, 1958. URL: https://archive.org/details/TheMeasurementOfPowerSpectra.

[CC10] Steven C. Chapra and Raymond P. Canale. *Numerical methods for engineers*. McGraw-Hill Higher Education, Boston, 6th ed edition, 2010.

[DRW+16] Enoch E. Dames, Andrew S. Rosen, Bryan W. Weber, Connie W. Gao, Chih-Jen Sung, and William H. Green. A detailed combined experimental and theoretical study on dimethyl ether/propane blended oxidation. *Combustion and Flame*, 168:310–330, June 2016. doi:10.1016/j.combustflame.2016.02.021.

[DSZM12] Apurba Kumar Das, Chih-Jen Sung, Yu Zhang, and Gaurav Mittal. Ignition delay study of moist hydrogen/oxidizer mixtures using a rapid compression machine. *International Journal of Hydrogen Energy*, 37(8):6901–6911, April 2012. doi:10.1016/j.ijhydene.2012.01.111.

[DUS12] Apurba Kumar Das, Mruthunjaya Uddi, and Chih-Jen Sung. Two-line thermometry and H2O measurement for reactive mixtures in rapid compression machine near 7.6μm. *Combustion and Flame*, 159(12):3493–3501, December 2012. doi:10.1016/j.combustflame.2012.06.020.

[GMS16] David G. Goodwin, Harry K. Moffat, and Raymond L. Speth. Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes, 2016. URL: http://www.cantera.org.

[HBG+05] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, September 2005. doi:10.1145/1089014.1089020.

[Hun07] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[JOPosh] Eric Jones, Travis Oliphant, Pearu Peterson, and others. *SciPy: Open source scientific tools for Python*. 2001–. [Online; accessed 2016-05-19]. URL: http://www.scipy.org/.

[LH98] Daeyup Lee and Simone Hochgreb. Rapid Compression Machines: Heat Transfer and Suppression of Corner Vortex. *Combustion and Flame*, 114(3-4):531–545, August 1998. doi:10.1016/S0010-2180(97)00327-1.

[MCSD08] Gaurav Mittal, Marcos Chaos, Chih-Jen Sung, and Frederick L. Dryer. Dimethyl ether autoignition in a rapid compression machine: Experiments and chemical kinetic modeling. *Fuel Processing Technology*, 89(12):1244–1254, December 2008. doi:10.1016/j.fuproc.2008.05.021.

[MS07] Gaurav Mittal and Chih-Jen Sung. A Rapid Compression Machine for Chemical Kinetics Studies at Elevated Pressures and Temperatures. *Combustion Science and Technology*, 179(3):497–530, 2007. doi:10.1080/00102200600671898.

[Nat16] Reality check on reproducibility. *Nature*, 533(7604):437–437, May 2016. doi:10.1038/533437a.

[OSB99] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing*. Prentice Hall, Upper Saddle River, N.J, 2nd ed edition, 1999.

[PG07] Fernando Pérez and Brian E. Granger. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. doi:10.1109/MCSE.2007.53.

[UDS12] Mruthunjaya Uddi, Apurba Kumar Das, and Chih-Jen Sung. Temperature measurements in a rapid compression machine using mid-infrared H2O absorption spectroscopy near 7.6 μm. *Applied Optics*, 51(22):5464–5476, August 2012. URL: http://www.ncbi.nlm.nih.gov/pubmed/22859037.

[US 16] US Energy Information Administration. EIA Monthly Energy Review. Technical Report DOE/EIA-0035(2016/4), April 2016. URL: http://www.eia.gov/totalenergy/data/monthly/archive/00351604.pdf.

[vCV11] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, March 2011. doi:10.1109/MCSE.2011.37.

[Web15] Bryan William Weber. CanSen, June 2015. URL: https://github.com/bryanwweber/CanSen.

[Web16] Bryan William Weber. UConnRCMPy, May 2016. URL: https://github.com/bryanwweber/UConnRCMPy.