Tabulation of Chemical Source Terms for Turbulent Combustion Simulations

Emmet Cleary: emcleary@princeton.edu Daniel Floryan: dfloryan@princeton.edu
Jeffry Lew: jklew@princeton.edu Bruce Perry: bperry@princeton.edu
Emre Turkoz: eturkoz@princeton.edu

21 November 2014

1 Introduction

Turbulent combustion simulations require closure of chemical source terms (reaction rates). This is not trivial, as the chemical source terms follow highly nonlinear Arrhenius kinetics and can depend on numerous stiffly-coupled chemical reactions. One approach, rather than evaluating these terms on the fly, is to calculate a set of thermochemical states *a priori* and use these when solving conservation equations. When combined with flamelet models[1], chemical source term tabulation greatly facilitates large eddy simulations (LES) of turbulent reacting flows.

The challenge with these tabulation methods is knowing how to identify each term as needed. This is done by tabulating source terms against a predetermined variable. The trick is to select a single variable that uniquely identifies each thermochemical state. Temperature is the most obvious one: the more a reaction proceeds, the more heat is released. However, filtering the energy conservation equation for LES leads to a closure problem. It is much simpler to filter species conservation equations, but a single chemical species is usually insufficient to identify the thermochemical state uniquely. Rather, one must take linear combinations of several species, called a progress variable, to define a mass-based conservation equation that is suitable for turbulent combustion simulations. Once a progress variable is chosen, the thermochemical states can be sorted, convoluted with a probability density function (PDF) to calculate filtered quantities for LES, and interpolated as needed to generate a table for chemical source terms (a chemtable) on a predefined grid.

Our project will use outputs from an existing research code called FlameMaster to facilitate and automate the selection of progress variables. Our project will also process output data from FlameMaster to create a chemtable through sorting based on the selected progress variable, convoluting, and interpolating. A variety of interpolation schemes, integration methods, and PDFs will be incorporated into the code to ensure that it is easily adaptable. The automated selection of progress variable has not been implemented in any existing code. Furthermore, although there exists code that can create chemtables after the progress variable has been defined, it lacks generality and must be rewritten or substantially modified when requirements for the chemtable change. A modular program for this task developed with a well-defined interfaces will significantly improve the generality and usability of the chemtable code.

[1]Pierce *et al.*, J. Fluid Mech. (2004) vol. 504, pp. 73-97.

2 Architecture

Our project consists of two related parts: determining the best progress variable (Figure 1) and generating the chemtables (Figure 2). Our program will be based in C++. SWIG will be used to interface the C++ functions to a Python wrapper that includes the user interface. The following tools will be used during software development:

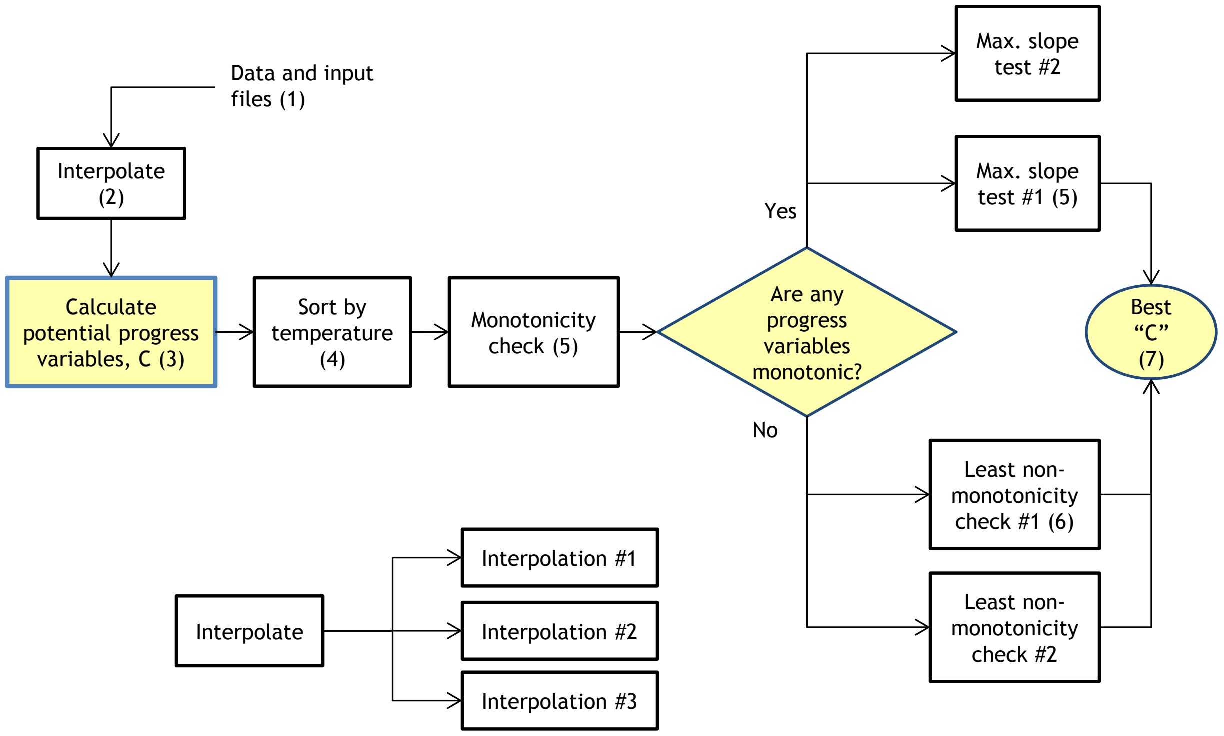

Figure 1: Flowchart depicting key steps in determining the best progress variable. Colored boxes indicate tasks that will be accomplished in Python and white blocks indicate tasks that will be accomplished with C++ functions.

2.1 Progress Variable

As shown in Fig. 1, inputs to the progress variable section of our program include a text input file containing the users selection of options to be used during run time and many combustion data files containing thermochemical state information. The final output in Fig. 1 is the best progress variable, which is passed to the table generation section of our program in Fig. 2. Detailed descriptions of the processes in Fig. 1 are below.

1. Input files contain a user-specified stoichiometric mixture fraction value, a user-specified list of species mass fractions, and a set of full thermochemical states represented as columns of temperature, species mass fraction, and chemical source terms in mixture fraction space. All combinations of mass fractions in the user-specified list are candidates for the progress variable.

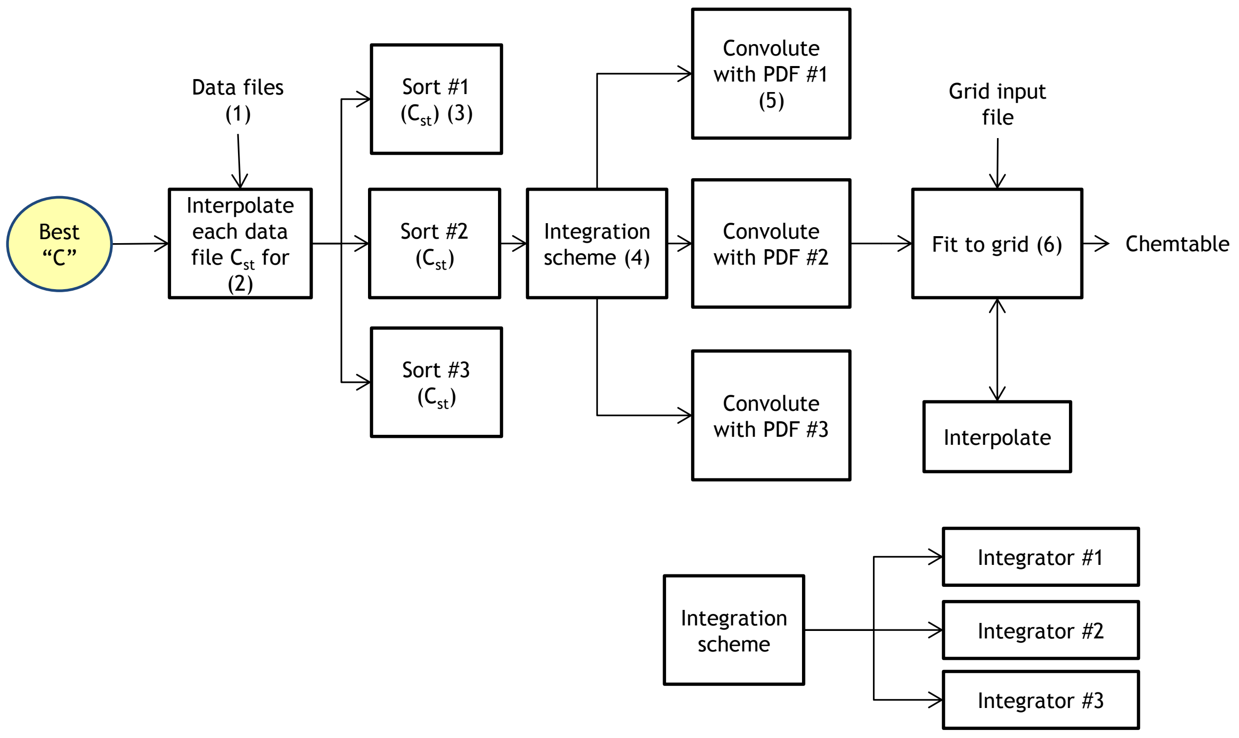

Figure 2: Flowchart depicting key steps in generating from the chemtables, using the definition of progress variable determined by the first section of code. All functions for this section of the program will be written in C++, but as with the progress variable section these functions will be called from a Python program that provides the user interface.

2. Python code exports two rows from each thermochemical state to be interpolated at the stoichiometric mixture fraction.

3. Python code calculates all progress variable candidates at these interpolated values.

4. C++ code sorts all progress variables by the stoichiometric temperature through various sort functions, generating arrays of temperature and all progress variables. These arrays are the only data used for the rest of this section.

5. All progress variables are tested for monotonicity with temperature in C++ code.

6. Monotonic progress variables will be ranked by various tests, *e.g.* an average slope test.

7. If no monotonic progress variables exist, the variables will be tested to pick the one that is least non-monotonic, *e.g.* monotonic over the greatest range.

8. The best progress variable will be sent to the table generation section.

2.2 Table Generation

The table generation portion of our project is shown in Fig. 2. Inputs include the best progress variable from the previous section, thermochemical states (data files), a grid input file, and a text input file containing the users selection of options to be used during run time (same text input file from Fig. 1). The final output of this part of the code is a chemtable, which can be fed into another program for further processing. Detailed descriptions of the processes in Fig. 2 are below.

1. The input data files are the same inputs used for the progress variable selection. The best progress variable, as determined by the processes depicted in Fig. 1, is also an input.

2. Progress variable is calculated for each data file and interpolated to find the value at the stoichiometric mixture fraction C_{st}. Various interpolation schemes will be used in the required sections of the program. These interpolation schemes are conducive to polymorphism - specific styles of interpolation will be employed as classes which inherit from an abstract interpolation class. User selection of the interpolation methods for each process will be determined from the text input file which will contain the chosen scheme.

3. The data files will be sorted by the stoichiometric progress variable, again using inheritance and polymorphism for the sorting functions. The algorithm used will be read from the text input file.

4. Numerical integration schemes will be used in the convolution step. Like interpolation and sorting, these numerical integration schemes exploit inheritance and polymorphism. Again, the numerical integration scheme is determined by the user in the text input file.

5. The progress variable will be convoluted with PDFs. The user will specify which PDF to use for convolution. The implementation will be polymorphic and the PDF to be used will be specified in the text input file. The output will be an array of values to be passed to the next process.

6. These values will be interpolated onto a table (grid) that is specified by the grid input file. This is the chemtable that will be the final output of our program.

2.3 Implemented Schemes/PDFs

Here we give some examples for the polymorphic classes that we will use in our code in different sections.

2.3.1 Sort

- Bubble sort
- Binary sort
- C++ Standard Library

2.3.2 Integration

- Trapezoid method
- Simpsons rule
- Gaussian quadrature

2.3.3 Interpolation

- Linear
- Cubic spline
- Bilinear

2.3.4 Probability Density Function

- Delta function
- Beta distribution
- Most probable distribution

2.4 External Functions

We will use the following outside functions and packages to facilitate the development of our software:

1. C++ Standard Library
2. Numpy - matrix and vector calculations in Python
3. Matplotlib - contour plots of chemtable results
4. SWIG - interface between Python and C++
5. FlameMaster (existing research software) - our code will not interact with FlameMaster, but the data files generated by FlameMaster will serve as inputs to our code

3 Milestones

Prototype - 12/5/2014: This version will have one element from each step working separately. Mathematical routines will be coded in C++. The two sections of the code will be developed separately. The tasks will be divided as follows:

- Interface between functions (All)
- Interpolation routines (Daniel)
- Sorting algorithms (Emre, Jeffry)

- Monotonicity check and maximum slope tests (Bruce, Jeffry)
- Integration schemes (Emmet, Daniel)
- Probability density functions (Emmet)
- I/O text processing (Emre, Bruce)

Alpha version - 12/12/2014: This version will include one element from each step at the presented pipeline and will be able to run with a "nice" configuration.

- Preliminary Python wrapper will be implemented using SWIG and will allow for some user interaction. (Emre, Emmet)
- One routine from each step (1 interpolator, 1 sorting algorithm, the basic monotonicity check, maximum slope check, 1 integration scheme along with 1 PDF) will be implemented in C++. (All)
- The link between the progress variable selection and chemtable generation sections will be implemented with Python. (Daniel, Jeffry)
- Fully-functional program that works on well-behaved data sets. (All)
- Basic plotting capability will be incorporated for visualization of the preliminary results. (Bruce)

Beta version - 01/6/2015: This is the version ready for submission.

- Fully-operational Python wrapper allows for autonomous and interactive use.
- Based on the interface we will already have built for the alpha function, we will implement alternative functions for interpolation, least non-monotonicity checking, integration, PDFs, and sorting.
- Fully-functional program that works on real FlameMaster output data.
- Interactive and advanced plotting capabilities will be incorporated for visualization.
- Error-checking functionality.
- Speed of key functions will be optimized.

For the beta version each team member will extend their own sections from the prototype and alpha version.

4 Risks and Open Issues

One of our foremost issues is completing the alpha version within the given project time constraints. The groundwork for our project must be laid with the completion of the alpha version.

We also think that the runtime of our code may be an issue for large data sets and for the cases which require large number progress variable candidates. Also for our research purposes, the data sets that our program will deal with are considerably large. If time turns out to be a bigger issue, we may consider implementing multi-threading parallelization using OpenMP.

Identifying a truly bijective mapping between progress variable and thermochemical state is among the most challenging and arbitrary parts of turbulent combustion simulations using flamelet models. Furthermore, even perfectly monotonic progress variables may have wildly varying slopes over temperature, so the highest average slope may not necessarily give the best progress variable. In fact, one may not even exist. For our project to be incorporated into real research codes it must still be able to deal with these challenging situations. In the worst cases, this can only be dealt with through plotting numerous graphs and requiring the user to visually determine the best progress variable. Regardless, our code will still greatly facilitate this process.

Both sections of our project are independently useful. Even if one is not successful, our project will still provide a useful tool for turbulent combustion simulations. And due to the modular structure of our project, even if one component cannot be completed, the remaining portions will not be affected by this problem. Furthermore, the modular structure of our project allows us to develop the various components in parallel and rapidly iterate through versions of our code.