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

January 15, 2015

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 used with flamelet models[1], chemical source term tabulation greatly reduces computational cost of large eddy simulations (LES) of turbulent reacting flows.

The challenge with these tabulation methods is determining the source term for statistics generated by the LES. 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 a linear combination 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.

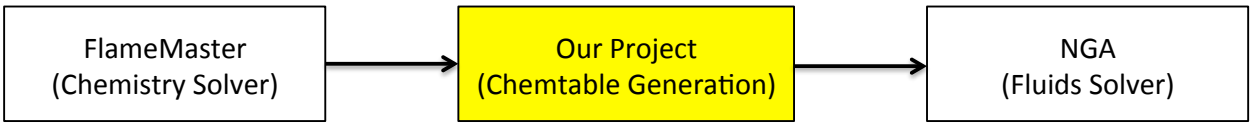

Figure 1: Context of project in relation to existing research codes.

Our project uses outputs from an existing research code called FlameMaster to facilitate and automate the selection of progress variables (see Fig. 1). Our project also processes FlameMaster data to create a chemtable through sorting based on the selected progress variable, convolution, and

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

interpolation. This chemtable can be used by the existing NGA codes to perform LES. A variety of interpolation schemes, integration methods, and PDFs are 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 codes exist to create chemtables after the progress variable has been defined, they lack generality and must be rewritten or substantially modified for different problems. A modular program for this task developed with well-defined interfaces will significantly improve the generality and usability of the chemtable code.

2 Design Process

The design process for this project began with identifying deficiencies in existing combustion research codes used by the Computational Reacting Turbulent Flow Laboratory (CTRFL). We identified several routines that could be modular: sorting, integrating, interpolating, and PDFs. Next, we outlined the overall process of our software by drawing flowcharts (Fig. 2 and 3). This helped structure our interfaces, as the flowcharts were used to identify inputs and outputs of the various routines. Then, we designed polymorphic classes in C++. This language choice was determined largely by our experience, but also for the ease of parallelization using OpenMP. Python was selected to provide the user interface with C++ functions, since it is well-suited to processing text inputs. Because our interfaces were well-designed before writing our code, only a few minor changes were made to the interfaces during software development. Specifics on the tools used during the design process (such as for testing and profiling) are provided in the Architecture section. I was also important to evaluate the pros and cons of writing our own codes rather than us existing codes. This is discussed below.

2.1 Make vs. Buy

Many routines required for our program exist in external libraries (e.g. interpolators, integrators, PDFs). The decision to write original code versus use libraries was determined by the simplicity of routine, our interfaces, and the expected effect on run time.

For simple integrators and interpolators, original codes were written. Some of the more complicated routines (e.g. quadrature integration or Hermite interpolation) use orthogonal polynomials that are far more challenging to write and test, so the external library AlgLib[2] was used.

Interfaces posed a slight problem for the PDFs. In our design review, it was suggested that we use external libraries to calculate values for the Beta PDF. This could have worked if we calculated PDF values on the fly during convolution. Instead, our approach required knowing full distributions over a range of statistics. It was far easier to calculate numerous PDF values at once and store them in a matrix.

Original routines had negligible influence on the total run time (see Profiling). There would have been negligible benefit to using better-optimized code.

3 Architecture

Our project consists of two connected parts: progress variable determination (Fig. 2) and chemtable generation (Fig. 3). Our program is written in C++ and Python 2.7. The following tools are used

[2] www.alglib.net

during software development:

- SWIG for interfacing C++ with Python
- Python Unittest framework for testing
- Doxygen for documentation
- Git for version control
- Valgrind for memory leak detection
- cProfile for profiling

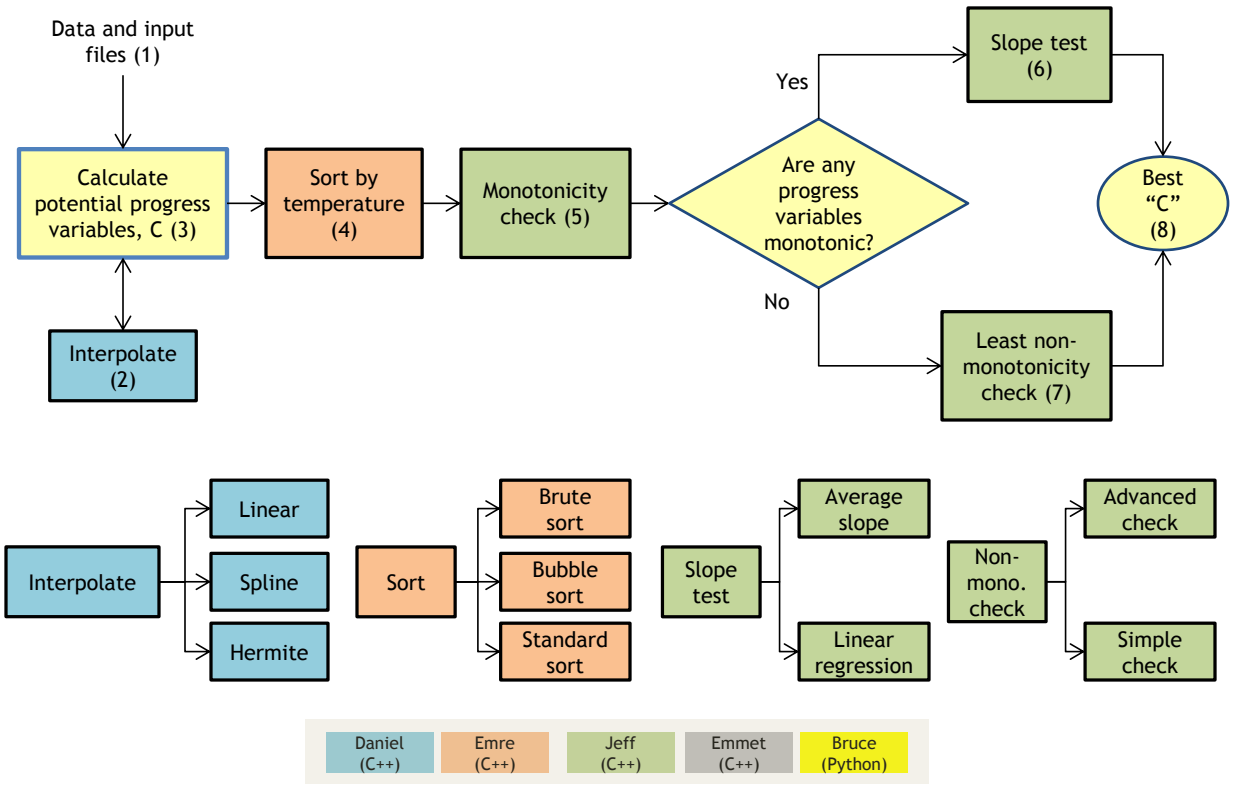

Figure 2: Flowchart for progress variable selection. Boxes are color-coded by programming language and task distribution

3.1 Progress Variable, C

As shown in Fig. 2, inputs to the progress variable section of our program include a text input file containing the user's selection of options to be used at run time and many combustion data files containing thermochemical state information. The final output of this section is the best progress variable, which is passed to the table generation section of our program in Fig. 3. Detailed descriptions of the processes in Fig. 2 are below.

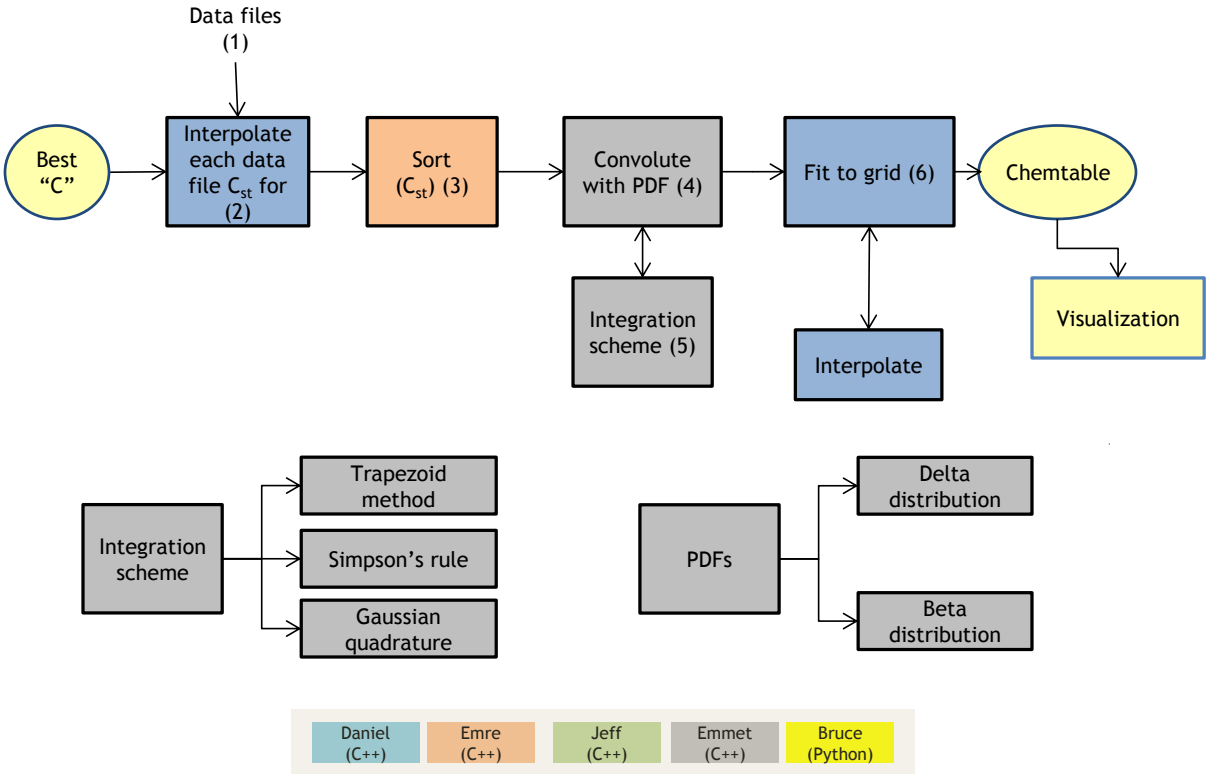

Figure 3: Flowchart for generating the chemtables, using the progress variable determined by the first section of code. Boxes are color-coded by programming language and task distribution.

1. The text input file contains a user-specified stoichiometric mixture fraction value, a user-specified list of species mass fractions, and options for function implementations. All combinations of mass fractions in the user-specified list are candidates for the progress variable. Data files are the FlameMaster data, containing thermochemical states represented as columns of temperature, species mass fraction, and chemical source terms in mixture fraction space.
2. Python code exports two rows from each data file to be interpolated at the stoichiometric mixture fraction.
3. Python code calculates all progress variable candidates using these interpolated values.
4. C++ code sorts all progress variables by the stoichiometric temperature, generating a matrix of temperature and all progress variables. This matrix contains the only data used for the rest of this section.
5. All progress variables are checked for monotonicity with temperature in C++ code.
6. Monotonic progress variables are ranked by various algorithms.
7. If no monotonic progress variables exist, the progress variables are tested to pick the least non-monotonic option.
8. The best progress variable is sent to the table generation section.

3.2 Table Generation

The table generation portion of our project is shown in Fig. 3. Inputs include the best progress variable from the previous section and a text input file containing the user's selection of options to be used at run time (same text input file from Fig. 2). The final output of this part of the code is a chemtable which is visualized through contour plots. Detailed descriptions of the processes in Fig. 3 are below.

1. This section of the program process the same FlameMaster data files used for progress variable selection.
2. The progress variable is calculated for all rows in each data file and interpolated to find the value at the stoichiometric mixture fraction C_{st}.
3. The data files are sorted by the stoichiometric progress variable.
4. The progress variable is convoluted with PDFs. The output is an array of values to be passed to the next process.
5. Numerical integration schemes are used in the convolution step.
6. These values are interpolated onto a grid as specified in the text input file. This is the chemtable that is the final output of our program.

3.3 Implemented Routines

As mentioned previously, many of the steps in both sections of the program are implemented using polymorphic classes. For example, specific interpolation methods are employed as classes which inherit from an abstract interpolation class. Multiple integration methods, sorting schemes, and PDFs are implemented in a similar manner. The polymorphic classes that we use in our code are listed here.

3.3.1 Sorting

- Brute sort
- Bubble sort
- C++ Standard Library Sort

3.3.2 Integration

- Trapezoidal method
- Simpson's rule
- Gauss-Legendre quadrature

3.3.3 Interpolation

- Linear
- Cubic spline
- Cubic-Hermite interpolation

3.3.4 Probability Density Function

- Delta function
- Beta distribution

3.3.5 Slope Test

- Average slope
- Linear regression

3.3.6 Least Nonmonotonicity Check

- Simple check
- Advanced check

3.4 External Libraries

We have used the following external libraries to facilitate the development of our software:

- C++ Standard Library - sorting
- AlgLib -
- Matplotlib - contour plots of chemtable results
- Numpy - matrix and vector calculations in Python
- numpy.i - SWIG typemapping

3.5 SWIG Implementation

C++ is interfaced with Python using SWIG, and the interface compiler that connects C/C++ functions with scripting languages like Python, Perl or Ruby. An example of how our C++ functions are wrapped is shown in Fig. 4. A *.i file was written for each abstract class, which allows all derived classes to be called from Python. Compilation with SWIG generates *_wrap.cxx, *.py and *.so files. Generated shared objects can be imported into Python scripts.

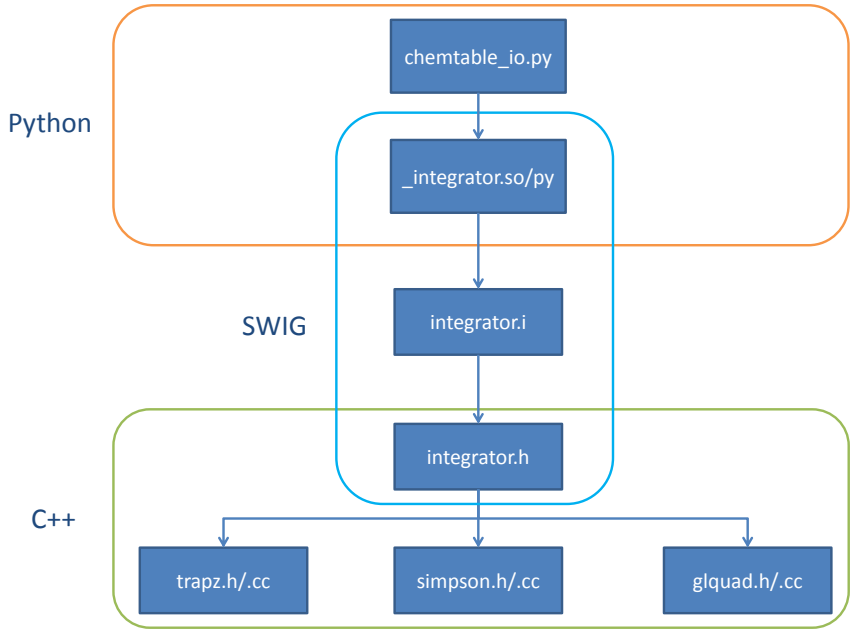

Figure 4: Role of SWIG in the interface between Python and C++ as shown using the integrator class as an example.

4 Versions

4.1 Prototype - 12/5/2014

This version included at least one derived class per abstract class. Each class worked independently. C++ code was compiled separately and no Makefiles were submitted. A primitive Python demo that read flamelet files, created candidate progress variables, and plot progress variable against temperature was also included.

4.2 Alpha version - 12/12/2014

This version included C++ code for at least one derived class per abstract class. SWIG was used to create one module per derived class. SWIG typecasting was done using standard vectors. All code was compiled using separate Makefiles for each derived class, each called by a single executable bash script. Python code interacted with a text input file, called each module accordingly and printed out tabulated data in the terminal.

4.3 Beta version - 01/15/2015

All derived classes have been written. SWIG now generates one module per abstract class. Python code interacts with a text input file, uses full data sets, calls each module accordingly, prints out tabulated data in a text file, and generates contour plots of the tabulated data. All C++ and Python code have been tested using Python Unittest. All files are organized in separate directories

containing source code, objects, modules, test functions, data, and output. All code is now compiled from a single Makefile in the home directory. Parallelization using OpenMP was implemented to reduce the run time of the grid-fitting step.

5 Division of Work and Git Repository Log

Figures 2 and 3 show how C++/Python codes were divided among the group. Each team member was responsible for testing his own code. There were several other tasks completed by all team members:

- Swig interface files (*.i)
- Test functions
- Alpha version Makefiles
- Design Document/Reports/Presentations

SWIG interface files were particularly challenging to write, especially for our implementation (for abstract classes rather than derived classes). As our project critically depended on making this work, all team members spent large amounts of time learning SWIG for the Alpha version. Specifically, Daniel made a breakthrough by implementing numpy.i for type-recasting. This first appears in only lininterp.i in the Alpha version, and was adopted for all SWIG interface files for the Beta version. Emmet made a breakthrough in generating modules for abstract classes, rather than generating modules for derived classes. Again, this format was adapted by all for the Beta version.

Other tasks were assigned to individual team members:

- Doxygen documentation: Emre
- Beta version Makefiles: Emmet
- Beta version README: Bruce

There are discrepancies between team members in the Git repository commit log. We attribute this to various styles of using Git. Emmet frequently committed minor changes to the repository, while Daniel and Jeff committed larger sections of code more infrequently.

6 Results

The chemtable generation program has been successfully applied to two data sets: a complete set of FlameMaster outputs for combustion of C_2H_4, and a truncated subset of this data chosen to be a well-behaved test case. These data sets were chosen to be representative of real data that would be analyzed to make a chemtable in turbulent combustion research. The input files used to run these example cases are shown in Appendix A. Additionally, the outputs printed by the program to the terminal when running these cases are shown in Appendix B. Note that a warning is printed that the input "plot all progress variables:" has not been specified and the default value, "yes", is being used.

Figure 5 shows the results of the progress variable selection portion of the program, using the truncated and full data sets. The species examined for inclusion in the progress variable are H_2O, H_2, CO_2, and OH. For the truncated data case, almost all candidate progress variables increase monotonically with temperature, and the program selects the candidate with the maximum slope (Y-H2O + Y-H2 + Y-CO2 + Y-OH). The program prints out this progress variable and lists separately all other monotonic progress variables. For the full case, all progress variables are non-monotonic, and the program correctly identifies the lest non-monotonic progress variable (Y-H2O + Y-H2 + Y-CO2). In fact, the non-monotonic region is small enough to be indistinguishable in Fig. 5(b). To notify the user, the program issues a warning that no completely monotonic progress variable was found.

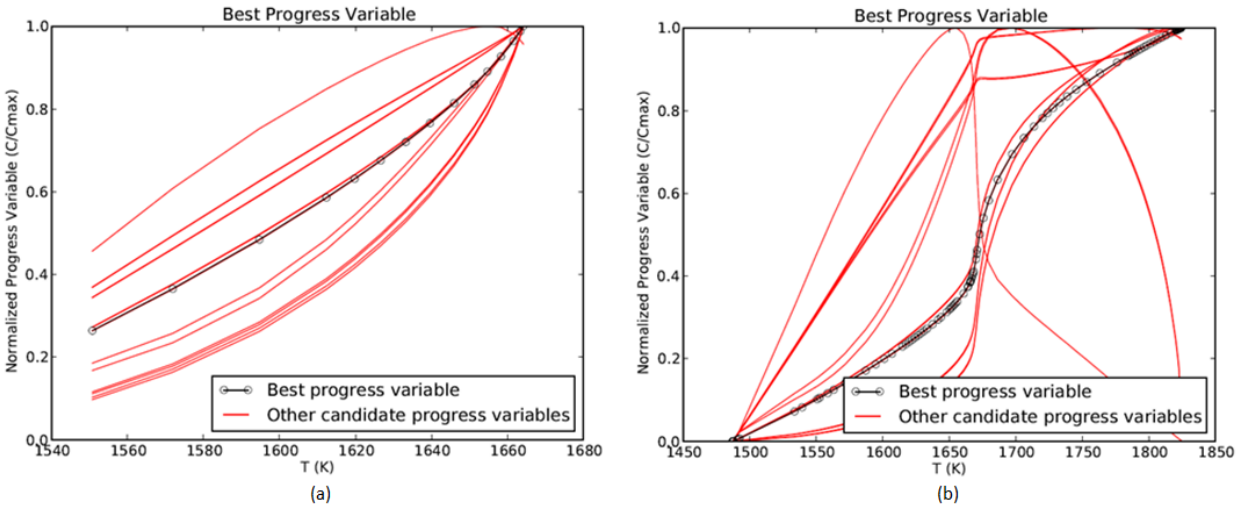

Figure 5: Plots of the selected progress variable and other candidate progress variables as a function of temperature for the (a) truncated and (b) full data sets.

Figure 6 shows sample contour plots generated from the truncated and full data. The contour plot generated from the truncated data matches what we expect for canonical counter-flow non-premixed flames[3]. The contour plot for the full data set is qualitatively similar, but has artifacts near where there are large gradients in chemical source term. These artifacts are a result of the slight non-monotonicity in the chosen progress variable and are unavoidable. Overall, these plots show that the chemtable generation program performs as expected and the data it produces is reliable.

6.1 Profiling

Our code was profiled using cProfile. For all cases, we used realistic numbers of FlameMaster data files as well a grid sizes. The three dimensions of the grid are Zmean, Zvar, and C. Zmean and Zvar refer to the mean and variance of mixture fraction, Z.

Our results point to the grid-fitting step as the bottleneck in our code. Results from tests with 201 grid points for Zmean and C, and 26 grid points for Zvar took the grid-fitting step 126

[3]Knudsen and Pitsch. Combust. Flame. (2009) vol. 156, pp. 678-696

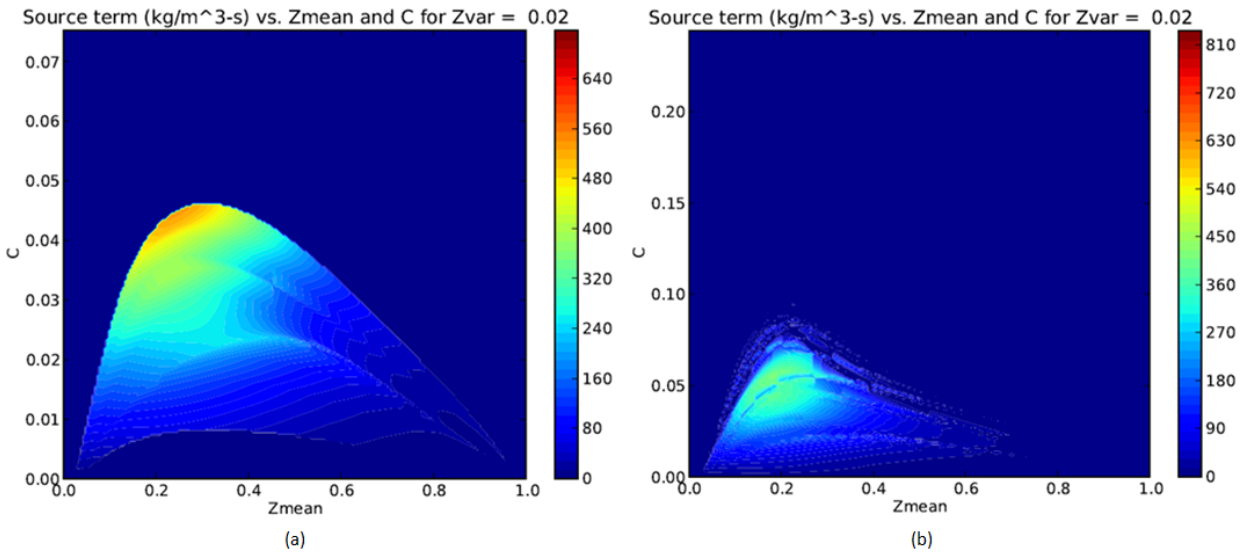

Figure 6: Contour plots of chemical source term as a function of progress variable (C) and mixture fraction (Zmean) at Zvar = 0.02 for the (a) truncated and (b) full data sets.

s to run and the entire program 142 s. Grid-fitting performs two tasks, namely interpolation and extrapolation. We attribute the long run times to the extrapolation part of this function, which contains 5 nested for loops. These for loops are necessary to search through a 3-dimensional matrix, and then for each point, to search through the same matrix again for the points nearest neighbor. Our only choice to speed up this part of the code was through parallelizing these loops. We chose to parallelize using OpenMP for its simplicity. Furthermore, even after all future work is implemented (see Future Work section), our expected run time should not exceed several hours. This run time is negligible compared to any LES that will use these chemtables, so parallelizing with MPI to use > 16 nodes was deemed unnecessary.

Figure 7 shows run times using 1, 2, 4, and 8 threads. Again, grid sizes were taken to be 201 points for Zmean and C, and 26 points for Zvar. For all cases, the total run time is about 16 s longer than the grid-fitting times. For 1, 2, and 4 threads used, grid-fitting run times roughly decrease by half as the number of threads doubles, indicating an $\mathcal{O}(n^{-1})$ dependence on thread number, as expected. The times for 8 threads deviates from this trends simply because the cost of parallelizing to 8 threads becomes comparable to the speedup.

Figure 8 shows the scaling of run time with the three dimensions of the grid. Based on the trends shown, it is apparent that the run time scales quadratically with grid size for Zmean and C, but linearly with grid size for Zvar. This behavior matches with expectations based on the design of the program. In the rate limiting grid fitting function, the nested loops must iterate over Zmean and C twice, but only once over Zvar. This plot indicates that higher grid resolution would significantly increase run time, but the tested resolution is representative of what is required for combustion research, so this should not be an issue.

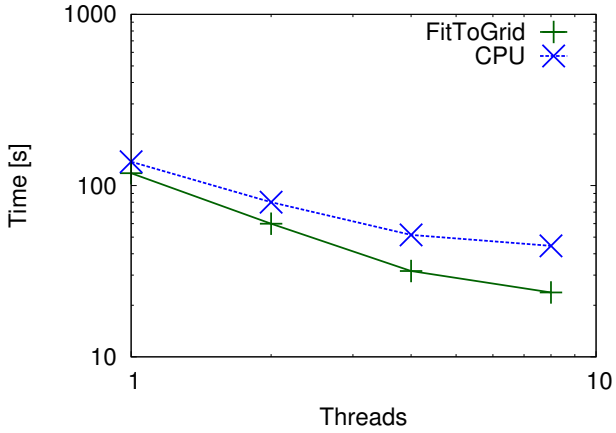

Figure 7: Comparison of run times for the gird-fitting function (FitToGrid) and total run time (CPU).

6.2 Interpolator Accuracy

The linear interpolation gave accurate results, but Hermite and cubic interpolation scheme gave nonsensical results. An illustrative example is shown in figure 9. Steep gradients in the data cause the interpolating polynomials to have steep gradients as well, leading to the enormous inaccuracies observed. Linear interpolation is, of course, insensitive to such effects. This leads us to question the appropriateness of interpolation schemes which are susceptible to such effects.

7 Lessons Learned

For most team members, this project was their first experience coding collaboratively. This was no easy feat: version control and interface design are particularly challenging in collaborative projects. We benefited greatly from designing our interfaces early in the design process and sticking to them. Using a Git repository greatly facilitated sharing and debugging code.

This was also our first experience working on a project with so many files. A project of this scale was difficult and confusing to maintain for the Alpha version, and would have been even worse for the Beta version had we not sorted our source code, modules, test, etc. into different directories. Thus, we also learned the value of maintaining a clean and organized working directory.

None of our team members had ever interfaced languages. This part of the project was probably the most troublesome to get working. SWIG was difficult to learn, but was incredibly useful. By interfacing C++ with Python, we were able to handle IO text processing in Python, as well as test our C++ code using Python Unittest.

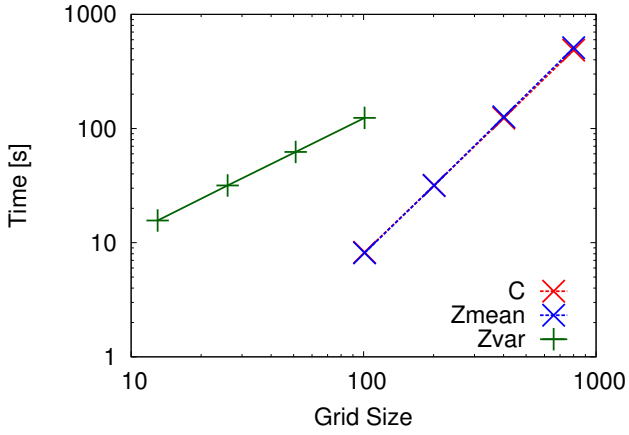

Figure 8: Sensitivity of parallelized code to grid size.

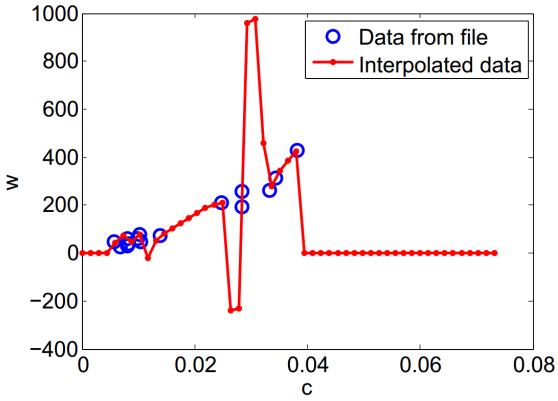

Figure 9: Interpolating along a grid using cubic Hermite interpolation.

8 Conclusion/Future Work

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. This software greatly facilitates the progress variable selection process and will likely be used by researchers in CTRFL. While this software meets all of our project objectives, there remains future work that could enhance the functionality and usability of this code:

- Software should be extended to filter all chemical species individually in addition to filtering chemical source terms as it already does. Only filtered chemical source terms are needed to run LES, but filtered species mass fractions and temperature are needed for visualization of LES results.
- Mathematically rigorous algorithms for finding the least non-monotonic progress variable can be incorporated based on current mathematics research. For example, an index of non-

decreasingness[4] can be implemented. Their applicability to progress variable selection is, to date, unknown, but would be an interesting topic of research.

- Manual selection of the best progress variable from a plot generated at the end of the section shown in Fig. 2. Currently, our program allows manual selection of the best progress variable at the very beginning, where the user is unable to view all the possible options.

[4]Qoyyimi *et al.*, SSRN. (2014), pp. 1-23.

Appendix A: Input File

The input file used to run the sample cases described in the Results section is shown below. This input file is for the full case, but the input file for the truncated case is identical except that data/C2H4 is replaced by data/C2H4truncated.

```
data file directory: data/C2H4
test species: Y-H2O Y-H2 Y-CO2 Y-OH
output file name: data_out
plot all progress variables:
skip progress variable optimization: no
extrapolate in fittogrid: no

number of threads: 4

Zmean_grid: 201
Zvar_max: 0.25
Zvar_grid: 26

Zpdf: beta
sort method: standard
StoichMassFrac: 0.05
interp method: linear
max slope test: linear regression
integrator: trapezoid
least nonmonotonic check: simple

glq Number of Nodes: 10
length Cgrid: 201
```

Appendix B: Output

Output for the full data case:

using default input for plot all progress variables: ['yes']
Sorting PROGVARS by temperature using standard sort
Sorting FILESMATRIX by temperature
Testing monotonicity

Finding least non-monotonic progress variable using simple nonmono check
WARNING: no monotonic progress variables found, but proceeding with best alternative.

The least non-monotonic progress variable is C = Y-H2O + Y-H2 + Y-CO2
The column numbers of these species are [8, 7, 9] , respectively.

Sorting filesmatrix by C using standard sort
Generating PDF matrix with beta PDF
PDF calculated
Convoluting using trapezoid integration
Convolution completed
Maximum value of progress variable is: 0.162988071
Fitting final data to grid using linear interpolation
No extrapolation used to fit to grid

Final data written to file: output/data_out
10 contour plots created in output directory

Output for the truncated data case:

using default input for plot all progress variables: ['yes']
Sorting PROGVARS by temperature using standard sort
Sorting FILESMATRIX by temperature
Testing monotonicity

Testing max slope using linear regression
The chosen progress variable is C = Y-H2O + Y-H2 + Y-CO2 + Y-OH
The column numbers of these species are [8, 7, 9, 6] , respectively.

Other candidate monotonic progress variables are:
C = Y-H2O
C = Y-CO2
C = Y-OH
C = Y-H2O + Y-H2
C = Y-H2O + Y-CO2
C = Y-H2O + Y-OH
C = Y-H2 + Y-CO2
C = Y-H2 + Y-OH
C = Y-CO2 + Y-OH
C = Y-H2O + Y-H2 + Y-CO2
C = Y-H2O + Y-H2 + Y-OH
C = Y-H2O + Y-CO2 + Y-OH
C = Y-H2 + Y-CO2 + Y-OH

Sorting filesmatrix by C using standard sort
Generating PDF matrix with beta PDF
PDF calculated
Convoluting using trapezoid integration
Convolution completed
Maximum value of progress variable is: 0.050205925
Fitting final data to grid using linear interpolation
No extrapolation used to fit to grid

Final data written to file: output/data_out
10 contour plots created in output directory