Python Code for Simulation of Al/CuO Nanothermite Pellet Combustion

by

Joseph Mark Epps
mark_epps@hotmail.ca

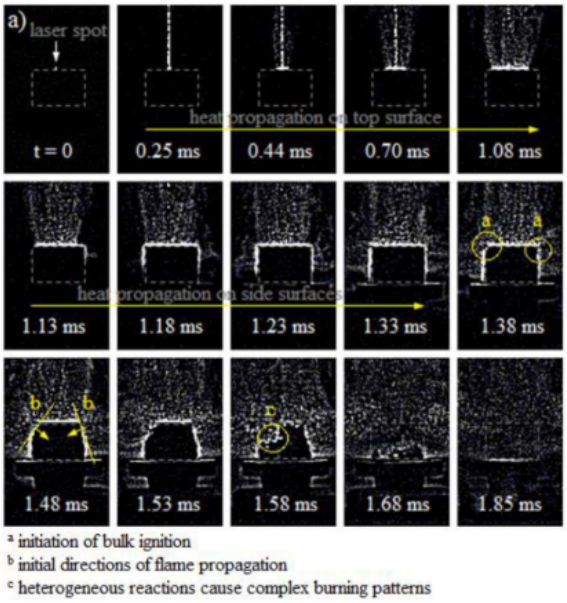

Figure 1: Combustion of nanothermite pellet [1]

June 17, 2020

Table of Contents

1 Introduction **1**
- 1.1 Message from Creator . . . 1
- 1.2 Motivation . . . 2

2 Governing Equations and Numerical Solution **4**
- 2.1 Governing Equations . . . 4
  - 2.1.1 Heat Mode . . . 4
  - 2.1.2 Species Mode . . . 5
- 2.2 Numerical Implementation . . . 8
  - 2.2.1 Flux Calculations . . . 9
  - 2.2.2 Time Advancement . . . 12
  - 2.2.3 Ignition and Burn Rate . . . 12
  - 2.2.4 Solver Procedure . . . 13

3 Using the Code **17**
- 3.1 Input Parameters . . . 17
  - 3.1.1 Modes and Domains . . . 21
  - 3.1.2 Thermal Properties . . . 22
  - 3.1.3 Porous Medium . . . 23
  - 3.1.4 Source Terms . . . 24
  - 3.1.5 Time Stepping . . . 25
  - 3.1.6 Boundary Conditions . . . 26
  - 3.1.7 Interpolation Schemes . . . 27
- 3.2 Code Execution . . . 28
- 3.3 Output . . . 28
- 3.4 Post-Processing . . . 30

4 Code Files and Structure **33**
- 4.1 Variable Types and Terminology . . . 33

4.2 Classes and Functions 34
4.2.1 FileClasses.py 35
4.2.2 GeomClasses.py 36
4.2.3 mpi_routines.py 38
4.2.4 MatClasses.py 39
4.2.5 SolverClasses.py 40
4.2.6 BCClasses.py 42
4.2.7 Source_Comb.py 42
4.2.8 Post.py 43
4.2.9 Post_timeEvolv.py 44
4.3 Parallelizing Routines 44
4.3.1 Process Dividing 44
4.3.2 Communication During Solve 45
4.3.3 Compiling Variable Data 46

Chapter 1

Introduction

This document contains all materials related to details of and running the nanothermite pellet combustion code. This chapter will be an introduction from the author and a brief statement of the motivation for the research. The governing equations that are solved are in the next chapter. Chapter 3will focus on how to use the code including modes and inputs. The last chapter will take a lower level look at the code including the classes, files, data types and procedures.

1.1 Message from Creator

Thank you for taking the time to read about my code. Although I am a graduate of Mechanical Engineering (University of Waterloo, Canada 2018), I enjoy programming. C++ and MATLAB were core languages covered and used (not C++ as much) during my degree, but I discovered Python on a co-op work term. I also taught myself batch programming in DOS which made doing repetitive tasks easy, but also gave me something new to learn and fun with. Object oriented programming was not a focus though, so I have been teaching myself those concepts in the last couple years.

Despite my enthusiasm for programming, I still consider myself a beginner in the programming world. Since objects were not a focus in my undergraduate degree,

this whole code was developed with my understanding of objects and through experimentation. Experienced programmers will probably look and laugh at the structure of my code or use of variables, but based on my understanding, it makes sense to me. It is likely not the most efficient code either, but I gave it a try.

Nevertheless, my love of programming, simulations (CFD, FEA) and teaching were why I decided to pursue a Master's degree. Given the choice of using commercial software packages (ANSYS and COMSOL were ones I have used before; I also worked at ANSYS!), academic codes or doing my own, I opted for the last option. I developed this code for my Master's degree thesis but also wanted to allow it to be used for simpler heat transfer problems. I shared it here, on Github, so others reading my publication (as of now, it is being submitted, so I have no link to it right now) or thesis can understand how I got the results. The results are in their own repository here. A lot of the information in this manual is from my thesis [2]; emphasis on the 'my thesis', please cite it appropriately if you wish to use the information for your uses (academic or otherwise).

One final note, there are lingering problems in the code that I didn't fix because a graduate degree has a schedule. It pained me to not fix them or I forgot they were there, but I wrote down issues that need attention as well as other features to add or how I would fix things up. I plan to do all these in my spare time, but in a separate repository. For future researchers, I want to preserve the code as it is so the results can be obtained by others and reverse engineered if needed.

1.2 Motivation

Nanothermite combustion is a fast, highly exothermic reaction where a metal and metal-oxide compound react to release a significant amount of heat. They are created by mixing nano-sized particles of each reactant using different mixing methods. Afterwards, they can be pressed into consolidated pellets (adjust density) for ignition. Their combustion characteristics can be tuned through a number of methods, which makes them useful in pyrotechnic applications such as igniters and propellants as well as welding applications. Table 1.1 was constructed from literature to give

an idea of some common thermite mixtures. Many experimental studies have been done to figure out what can be done to tune their characteristics and determine the governing mechanisms of the reaction. Numerical studies are not as prevalent and the ones that are done are simplified and do not focus on the porous nature of the pellets.

Table 1.1: Common thermite reaction equations, heat of combustion, adiabatic flame temperature and gas generation [3]

Chemical Equation	**Enthalpy of Combustion [kJ/g]**	**Adiabatic Flame Temperature [K]**	**Gas generation [g gas/g mix]**
$2\,Al + 3\,CuO \longrightarrow Al_2O_3 + 3\,Cu$	4.076	2843	0.3431
$2\,Al + MoO_3 \longrightarrow Al_2O_3 + Mo$	4.703	3253	0.2473
$2\,Al + WO_3 \longrightarrow Al_2O_3 + W$	2.914	3253	0.1463
$2\,Al + Bi_2O_3 \longrightarrow Al_2O_3 + 2\,Bi$	2.118	3253	0.8941
$2\,Al + 3\,NiO \longrightarrow Al_2O_3 + 3\,Ni$	3.441	3187	0.0063

A previous colleague in our research group (now a PhD holder) had performed experimental studies on Al/CuO at different packing densities and observed a dramatic increase in burn rate with packing densities. The objective of this research is to numerically quantify the mechanisms governing Al/CuO nanothermite combustion. Please refer to my thesis [2] for more information on the derivation of the model.

Chapter 2

Governing Equations and Numerical Solution

In this chapter, the model and numerical implementation will be explained. At its simplest, nanothermite combustion is a heat transfer problem, so the code has two modes; **Heat** and **Species**. A summary of the equations and basic background for each mode is summarized in this chapter. Options for specifying each parameter is covered in Chapter 3. For more details about the physics and explanations, go to my thesis [2]!

2.1 Governing Equations

2.1.1 Heat Mode

The most basic mode is the **Heat** mode, which is a porous solid heat conduction with source term solver. The governing equation solved is given by:

$$\frac{\partial \left(\rho_s C_{v,s} T\right)}{\partial t} = \nabla \cdot \left(\lambda_{eff} \nabla T\right) + \dot{Q}_{source} \tag{2.1}$$

where ρ_s is the solid density, $C_{v,s}$ is the specific heat at constant volume for the solid, λ_{eff} is the effective thermal conductivity and $\dot{Q}_{source}$ is heat generated.

There are two source term options; uniform or Arrhenius. Uniform is a constant volumetric heat generation while the Arrhenius is of the form:

$$\dot{Q}_{source} = \rho_s \dot{\eta} \Delta H \tag{2.2}$$

$$\dot{\eta} = A_0(1-\eta)\exp\left(-\frac{E_a}{RT}\right) \tag{2.3}$$

where η is the reaction progress, E_a is the activation energy, A_0 is the pre-exponential factor and ΔH is the enthalpy of combustion.

2.1.2 Species Mode

Species mode is built around a two phase model that was adopted to focus on the gas and non-gas phases; termed *gas* and *solid* phases respectively. The link between the phases is given by the reaction equation:

$$\mathrm{A_{(s)}} \longrightarrow \mathrm{B_{(}}_{s,l\mathrm{)}} + \mathrm{C_{(g)}} \tag{2.4}$$

where A is the *solid* phase pre-combustion, B is the *solid* phase post-combustion and C is the *gas* phase. Although B can be in liquid or solid states, the *solid* phase is assumed to behave like a solid. A visual description is in Figure 2.1.

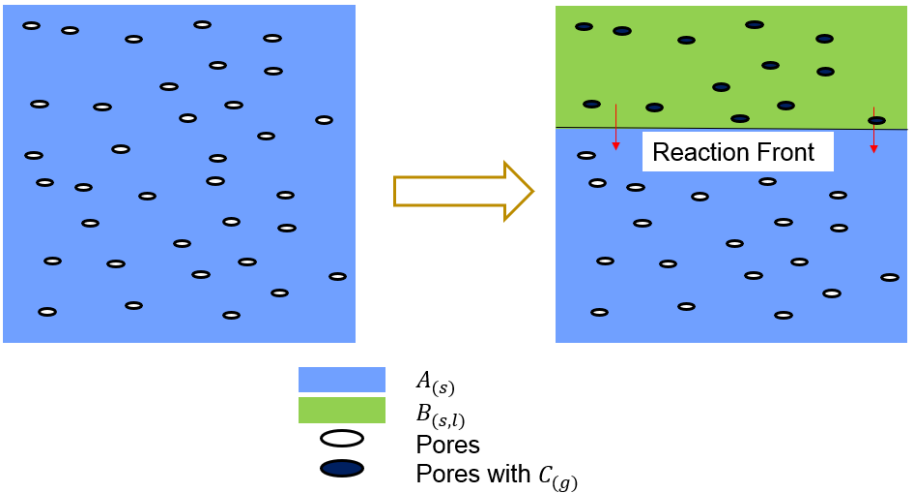

Figure 2.1: Visualization of 2-phase mass model

The conservation equations for the phases, denoted with the subscripts s (*solid*) and g (*gas*), are:

$$\frac{\partial(\rho_s)}{\partial t} = \dot{\rho}_{s,source} \tag{2.5}$$

$$\frac{\partial(\rho_g)}{\partial t} + \nabla \cdot (\rho_g \mathbf{u}) = \dot{\rho}_{g,source} \tag{2.6}$$

$$\frac{\partial\left(\rho C_{eff} T\right)}{\partial t} + \nabla \cdot (\rho_g \mathbf{u} C_{p,g} T) = \nabla \cdot (\lambda_{eff} \nabla T) + \dot{Q}_{source} \tag{2.7}$$

where $\mathbf{u}$ is the vector of the fluid velocity, $\dot{\rho}_{(g,s),source}$ are the source terms for each phase, $(\rho C_{eff} T)$ is the energy storage term, $\rho C_{eff} = \rho_s C_{v,s} + \rho_g C_{v,g}$ is the energy storage contribution of the *solid* and *gas* species, $C_{v,(g,s)}$ is the specific heat at constant volume for each phase, λ_{eff} is the effective thermal conductivity, $C_{p,g}$ is the specific heat at constant pressure for the *gas*, and $\dot{Q}_{source}$ is heat generated due to the reaction. The *gas* and *solid* are assumed to be in thermal equilibrium and thus have equal temperatures.

The advection of *gas* within the porous pellet is driven by the pressure gradient

resulting from the reaction and uses Darcy's law:

$$\mathbf{u} = -\frac{K}{\mu}\nabla P \tag{2.8}$$

where P is the pressure, K is the permeability of the solid, and μ is the viscosity of the gas.

The permeability is calculated using the Kozeny-Carmen equation [4] given by:

$$K = \frac{\phi^3 d_c^2}{k_k(1-\phi)^2} \tag{2.9}$$

where ϕ is the porosity, d_c is the ratio between solid volume and fluid-solid interfacial surface area [5] (will be referred to as characteristic pore size) and k_k is the Kozeny constant.

The *gas* is assumed to behave like an ideal gas law which allows the calculation of pressure by:

$$P = \frac{\rho_g RT}{\phi} \tag{2.10}$$

where R is the specific gas constant. Since the variable ρ_g is per volume of continuum, porosity must be included to reflect the pressure inside the pore.

$\dot{Q}_{source}$ follows the Arrhenius form (only) given by:

$$\dot{Q}_{source} = \rho_{s,0}\dot{\eta}\Delta H \tag{2.11}$$

$$\dot{\eta} = A_0(1-\eta)\exp\left(-\frac{E_a}{R_u T}\right) \tag{2.12}$$

where $\rho_{s,0}$ is the initial density of *solid*, ΔH is the enthalpy of combustion, A_0 is the pre-exponential factor, η is the reaction progress variable, Ea is the activation energy, and R_u is the universal gas constant.

From equation (2.12), the source terms for Equations (2.5) and (2.6) can be derived. Using the *gas* to define the extent of reaction:

$$\eta = \frac{\rho_g - \rho_{g,0}}{\rho_{g,f}} \tag{2.13}$$

where $\rho_{g,0}$ is the amount of *gas* present in the pores pre-combustion and $\rho_{g,f}$ is the amount of *gas* post-reaction, taking the time derivative allows both source terms to be calculated easily by utilizing Equation (2.12) in:

$$\dot{\rho}_{g,source} = \dot{\eta}\rho_{g,f} \tag{2.14}$$

The amount of *gas* post-reaction is calculated as a percentage of the initial *solid* density:

$$\rho_{g,f} = c\rho_{s,0} \tag{2.15}$$

c is a value between 0 and 1. For my research, this value was based on the amount of gas generated reported in Fischer and Grubelich [3] for Al/CuO reaction. At any stage in the reaction (assuming no advection or diffusion of mass), the total mass of each phase has to be conserved. Based on equation (2.4), the relationship between the source terms is given by:

$$0 = \dot{\rho}_{s,source} + \dot{\rho}_{g,source} \tag{2.16}$$

making the source terms for equations (2.5) and (2.6) equal in magnitude.

Note that the definition of η is used to derive the *gas* source term, however the reaction progress is only dependent on Equation (2.12) and the pressure relationship of Equation (2.10) has no impact.

2.2 Numerical Implementation

This code is a first-order finite volume method solution to the governing equations with vertex centred nodes. This section will outline the relevant concepts of the Finite Volume Method in the current framework as well as details regarding time stepping, stability, boundary and initial conditions, ignition criteria and code progression. Axisymmetric or 2D planar implementations can be solved using this code and as a quick calculus lesson, here are some definitions for converting the gradient form of

the governing equations to individual partial derivatives:

$$\nabla \cdot \mathbf{A} = \frac{\partial \mathbf{A}}{\partial x} + \frac{\partial \mathbf{A}}{\partial y} \tag{2.17a}$$

$$= \frac{\partial \left(r\mathbf{A}\right)}{r\partial r} + \frac{\partial \mathbf{A}}{\partial z} \tag{2.17b}$$

where equation (2.17a) is the 2D planar implementation and equation (2.17b) is the axisymmetric implementation.

2.2.1 Flux Calculations

The Finite Volume Method is a well established method of numerical solution to partial differential equations, so only the basic principles are outlined here to show you how the code operates. Since the nature of my research was for cylindrical pellets, the description will be for the axisymmetric implementation. A finite volume approximation to the axisymmetric implementation of equation (2.6) requires approximating each spatial derivative with a flux function:

$$\frac{\partial}{r\partial r}\left(r\rho_g u\right) \approx \frac{1}{r_{i,j}H_r}\left(m''_{i+\frac{1}{2},j} - m''_{i-\frac{1}{2},j}\right) \tag{2.18}$$

$$\frac{\partial}{\partial z}\left(\rho_g v\right) \approx \frac{1}{H_z}\left(m''_{i,j+\frac{1}{2}} - m''_{i,j-\frac{1}{2}}\right) \tag{2.19}$$

where i and H_r are the node number and control volume length in the radial direction respectively and j and H_z are the node number and control volume length in the axial directions, respectively. To compute the convective face flux in Equation (2.6), a first-order averaging of the properties between nodes are used to approximate the

properties at the control surface:

$$m''_{i+\frac{1}{2},j} = f(\rho_{i+1,j}, \rho_{i,j}) r_{i+\frac{1}{2},j} u_{i+\frac{1}{2},j} \tag{2.20}$$

$$m''_{i-\frac{1}{2},j} = f(\rho_{i,j}, \rho_{i-1,j}) r_{i-\frac{1}{2},j} u_{i-\frac{1}{2},j} \tag{2.21}$$

$$m''_{i,j+\frac{1}{2}} = f(\rho_{i,j+1}, \rho_{i,j}) v_{i,j+\frac{1}{2}} \tag{2.22}$$

$$m''_{i,j-\frac{1}{2}} = f(\rho_{i,j}, \rho_{i,j-1}) v_{i,j-\frac{1}{2}} \tag{2.23}$$

where the functions are described in section 3.1.7. The velocities are directly calculated from discretizing Equation (2.8):

$$u_{i+\frac{1}{2},j} = \frac{K}{\mu} \frac{P_{i+1,j} - P_{i,j}}{\Delta r} \tag{2.24}$$

$$u_{i-\frac{1}{2},j} = \frac{K}{\mu} \frac{P_{i,j} - P_{i-1,j}}{\Delta r} \tag{2.25}$$

$$v_{i,j+\frac{1}{2}} = \frac{K}{\mu} \frac{P_{i,j+1} - P_{i,j}}{\Delta z} \tag{2.26}$$

$$v_{i,j-\frac{1}{2}} = \frac{K}{\mu} \frac{P_{i,j} - P_{i,j-1}}{\Delta z} \tag{2.27}$$

where Δr and Δz are the distances between nodes in the r and the z directions respectively. Since the pressure is calculated at the nodes, a second-order central difference scheme is used to approximate the pressure gradient that yields a velocity on the control surface; no further interpolation is required. The same flux approximation routine applies for Equation (2.7):

$$\frac{\partial}{r\partial r}\left(r\rho_g u C_{p,g} T\right) \approx \frac{1}{r_{i,j} H_r}\left(m''_{i+\frac{1}{2},j} h_{i+\frac{1}{2},j} - m''_{i-\frac{1}{2},j} h_{i+\frac{1}{2},j}\right) \tag{2.28}$$

$$\frac{\partial}{r\partial r}\left(r\frac{\partial T}{\partial r}\right) \approx \frac{1}{r_{i,j} H_r}\left(q''_{i+\frac{1}{2},j} - q''_{i-\frac{1}{2},j}\right) \tag{2.29}$$

$$\frac{\partial}{\partial z}\left(\rho_g v C_{p,g} T\right) \approx \frac{1}{H_z}\left(m''_{i,j+\frac{1}{2}} h_{i,j+\frac{1}{2}} - m''_{i,j-\frac{1}{2}} h_{i,j+\frac{1}{2}}\right) \tag{2.30}$$

$$\frac{\partial}{\partial z}\left(\frac{\partial T}{\partial z}\right) \approx \frac{1}{H_z}\left(q''_{i,j+\frac{1}{2}} - q''_{i,j-\frac{1}{2}}\right) \tag{2.31}$$

The diffusion flux functions (in either equation (2.1) or (2.7)) use second-order central difference schemes to approximate the temperature gradient at the face and are calculated by:

$$q''_{i+\frac{1}{2},j} = g(\lambda_{i+1,j}, \lambda_{i,j}) r_{i+\frac{1}{2},j} \frac{T_{i+1,j} - T_{i,j}}{\Delta r} \tag{2.32}$$

$$q''_{i-\frac{1}{2},j} = g(\lambda_{i,j}, \lambda_{i-1,j}) r_{i-\frac{1}{2},j} \frac{T_{i,j} - T_{i-1,j}}{\Delta r} \tag{2.33}$$

$$q''_{i,j+\frac{1}{2}} = g(\lambda_{i,j+1}, \lambda_{i,j}) \frac{T_{i,j+1} - T_{i,j}}{\Delta z} \tag{2.34}$$

$$q''_{i,j-\frac{1}{2}} = g(\lambda_{i,j}, \lambda_{i,j-1}) \frac{T_{i,j} - T_{i,j-1}}{\Delta z} \tag{2.35}$$

where the interpolation functions $g()$ are outlined in section 3.1.7. The enthalpy for the convective terms follow the same interpolation methods as seen in the mass flux functions:

$$h_{i+\frac{1}{2},j} = f(C_{i+1,j}, C_{i,j}) f(T_{i+1,j}, T_{i,j}) \tag{2.36}$$

$$h_{i-\frac{1}{2},j} = f(C_{i,j}, C_{i-1,j}) f(T_{i,j} + T_{i-1,j}) \tag{2.37}$$

$$h_{i,j+\frac{1}{2}} = f(C_{i,j+1} + C_{i,j}) f(T_{i,j+1} + T_{i,j}) \tag{2.38}$$

$$h_{i,j-\frac{1}{2}} = f(C_{i,j} + C_{i,j-1}) f(T_{i,j} + T_{i,j-1}) \tag{2.39}$$

I want to point out that the averaging schemes implemented are not the best and given more time, I would've improved it.

It is worth noting that the values for H_r and Δr are equal in this implementation because the domain is uniformly discretized in the radial direction. The axial direction is also uniformly discretized, so the same applies for H_z and Δz, however the radial and axial directions can have different discretizations. I intended to code non-uniformly discretized domains, in which case, these values would differ due to the contribution of adjacent Δr values on H_r. Time did not permit further development, but much of the framework is in place for non-uniformly discretized rectangular grids.

2.2.2 Time Advancement

An explicit time scheme was selected as the time advancement scheme due to its simple implementation. Using the time term from equation (2.5) as an example, the time derivative is discretized as:

$$\frac{\partial(\rho_s)}{\partial t} \approx \frac{\rho_s^{k+1} - \rho_s^k}{\Delta t} \tag{2.40}$$

where k is the time step and Δt is the time step size. In each of the governing equations, all other terms are evaluated at time step k in order to recompute the density or temperature at the time step $k+1$. Stability is addressed through Fourier and Courant-Friedrichs-Lewry (CFL) numbers:

$$Fo = \frac{\lambda_{eff}\Delta t}{(\rho C)_{eff}(\Delta r)^2} + \frac{\lambda_{eff}\Delta t}{(\rho C)_{eff}(\Delta z)^2} < 0.5 \tag{2.41}$$

$$CFL = \frac{u\Delta t}{\Delta r} + \frac{v\Delta t}{\Delta z} < 1 \tag{2.42}$$

where u and v are the Darcy velocities in r and z respectively.

2.2.3 Ignition and Burn Rate

When the Arrhenius source term (Equation (2.2)) is implemented, ignition occurs when the reaction becomes self-sustaining. In this code, the heat generated comes from the source term, $\dot{Q}_{source}$, and ignition will occur when this value exceeds the conductive and convective losses:

$$\dot{Q}_{source} > \nabla \cdot (\rho_g \mathbf{u} C_{p,g} T) - \nabla \cdot (\lambda_{eff} \nabla T) \tag{2.43}$$

Since the equations are implemented numerically, the source term should be a specified factor larger than the losses and should hold in a specified number of control volumes. This method of defining ignition is very crude and very problem dependent; I spent lots of time just trying to figure out the best numbers for this to work for my research. Once ignition has occurred, the laser flux boundary condition is replaced

with the same convective boundary conditions on the other faces. More details on specifying ignition conditions are in section 3.1.4.

Each successive time step after ignition will calculate the instantaneous burn rate:

$$V_{cw} = \frac{d}{dt} \int_0^L \eta dy \tag{2.44}$$

where L is the height of the domain (in y or z). The average burn rate is calculated using an arithmetic average of all instantaneous burn rates.

2.2.4 Solver Procedure

When the code is initialized, a series of declarations are done as illustrated in Figure 2.2. The domain object contains the arrays of variable data (energy, pressure, densities), discretizations and functions to calculate thermal properties. The solver object contains the functions to solve the governing equations by manipulating the variables directly in the domain object. Multiple cores are implemented using Message Passing Interface (MPI) to speed up the solving process by splitting the domain into equal sized sections that each process focuses on.

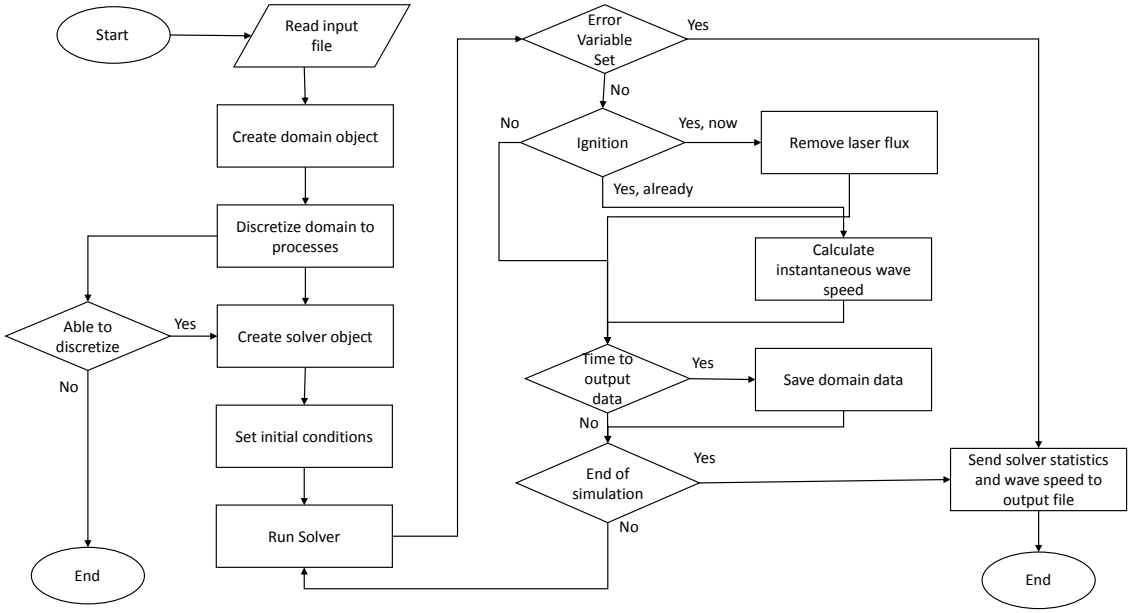

Figure 2.2: Flowchart of main code procedure

Inside the 'Run Solver' box of Figure 2.2, the governing equations are solved in a logical procedure to ensure a consistent solution. A flow chart of this procedure is given in Figure 2.3.

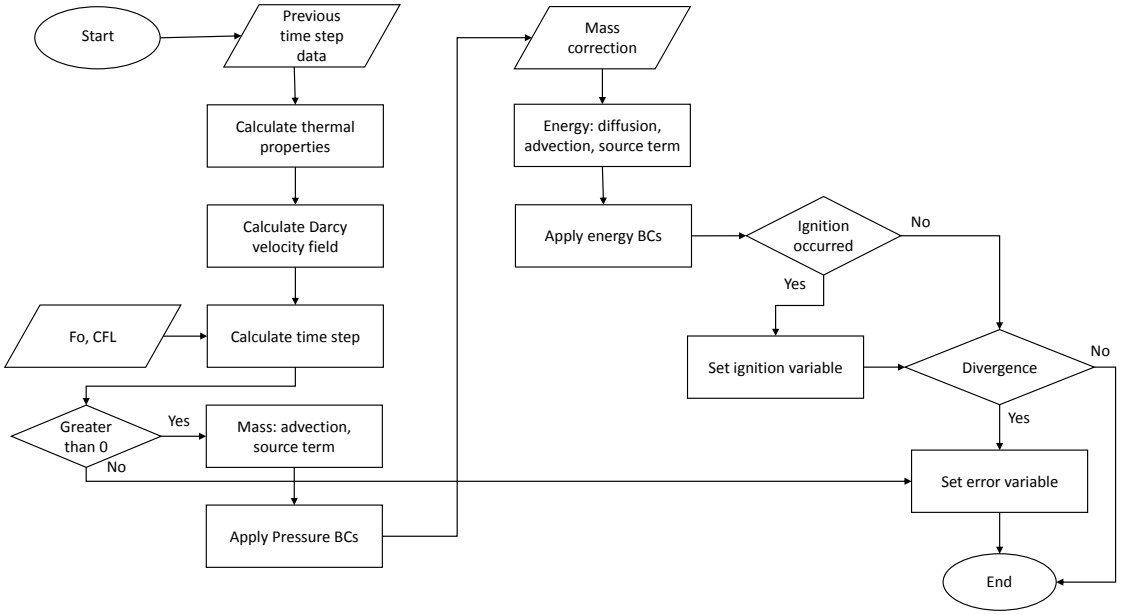

Figure 2.3: Flowchart of 'Run Solver' operation

The temperature-dependent thermophysical properties and the Darcy velocity fields are re-computed at the start of each time-step and are assumed constant over the small, Δt of time-advancement step. Similarly, the source terms for the energy and mass equations are computed at the start of the time-step. Using the computed source terms and velocity field, the conservation of mass is then applied. Due to the time independent nature of the momentum equation, it is used directly in the computation of the mass conservation. The boundary conditions are then applied on the pressure field which sets the ambient pressure of the combustion. As the thermodynamic conditions are fully prescribed at the external bounds of the pellet, mass advection at the boundary nodes is necessary to enforce the pressure and density. This permits the advective mass loss through the porous boundaries of the nanothermite pellet and maintains a consistent numerical boundary condition in the problem. The energy equation is then solved by applying the diffusion and advection for both phases. The laser heating and natural convection with the ambient boundary conditions are applied last.

After computing the updated values, if ignition has not occurred previously, the conditions outlined in section 2.2.3 are checked. If the conditions are satisfied, the laser flux boundary condition is replaced with a natural convection boundary condition. If ignition occurred previously, the instantaneous burn rate is calculated and the average burn rate is updated.

The divergence check in Figure 2.3 is intended to catch any signs of instability or non-physical values in the domain. Conditions that were coded to be checked include the temperature becoming undefined or negative, reaction progress is outside the bounds $0 \leq \eta \leq 1$ and the values of density of each phase. The density checks were commented out (not deleted) due its false divergence catches; the likely cause was divergence in a ghost node. If the divergence checks are triggered or the calculated time step is negative, the solver box is terminated and an error sent to the main code loop where the 'Error Variable Set' box in Figure 2.2 will terminate the rest of the code safely.

Chapter 3

Using the Code

In this section, details on how to use the code including available options and cautions/warnings when specifying inputs are explored. All numbers specified are in standard units of m, kg, K, W or J unless otherwise specified. For example, specific heat should be specified in $\mathrm{J\,kg^{-1}\,K^{-1}}$.

3.1 Input Parameters

The input file is read by the main script and contains all parameters needed to solve the equations described in the previous sections. Any line beginning with # will be ignored when reading in. Each line that is read in will have the name of the variable then a colon and then the information needed to specify that parameter. Any parameter that requires a floating point number to be specified can be written as a decimal (0.001) or in scientific notation (1e-3). A comprehensive list with all inputs, brief description and relevant reference sections is given below:

Table 3.1: Comprehensive list of input file parameters

Parameter	**Description**
Domain	Domain type for solver. See section 3.1.1
Model	Heat or Species modes. See section 3.1.1

Table 3.1: (continued)

Parameter	**Description**
Length	Length of domain
Width	Width of domain
Nodes_x	Discretization along length (x-direction for 'Planar', r-direction for 'Axisymmetric')
bias_type_x	Discretization method in x or r direction; LEAVE AS 'None' since the solver is not coded correctly
Nodes_y	Discretization along width (y-direction for 'Planar', z-direction for 'Axisymmetric')
bias_type_y	Discretization method in y or z direction; LEAVE AS 'None' since the solver is not coded correctly
rho_IC	Initial density of each specie in the domain. If in **Heat** mode, indicate initial density of solid phase with no porosity. If in **Species** mode, indicate initial densities of *gas* and *solid* phases separated by a comma.
Temperature_IC	Initial temperature of domain
Thermal Properties - see section 3.1.2	
Cv_s	Specific heat at constant volume for solid or *solid* phase
Cv_g	Specific heat at constant volume for *gas* phase
Cp_g	Specific heat at constant pressure for *gas* phase
k_s	Thermal conductivity of solid or *solid* phase
k_g	Thermal conductivity of *gas* phase
k_model	Thermal conductivity calculation method when **Species** mode is active
Porous Medium - see section 3.1.3	

Table 3.1: (continued)

Parameter	**Description**
Porosity	Volume fraction of domain that is void space. If 0, is regular solid domain. DO NOT INDICATE 1
Carmen_diam	Length scale in Carmen-Kozeny equation.
Kozeny_const	Kozeny constant in Carmen-Kozeny equation.
Darcy_mu	Dynamic viscosity used in Carmen-Kozeny equation.
gas_constant	Specific gas constant used in pressure calculation; ($\mathrm{J\,kg^{-1}\,K^{-1}}$)
Source Terms - see section 3.1.4	
Source_Uniform	Specify uniform heat generation in $\mathrm{W\,m^{-3}}$
Source_Kim	Enable Arrhenius source term model for combustion or not
Ea	Activation energy for Arrhenius source term model ($\mathrm{J\,mol^{-1}}$)
A0	Pre-exponential factor for Arrhenius source term model ($\mathrm{s^{-1}}$)
dH	Enthalpy of combustion for Arrhenius source term model
Ignition	Ignition criteria to change north boundary condition to convective BC and start calculating combustion speed
gas_gen	Gas generated for mass source terms
Time Advancement - see section 3.1.5	
Fo	Fourier number
CFL	CFL number
dt	Time step size
total_time_steps	Total number of time steps to simulate

Table 3.1: (continued)

Parameter	**Description**
total_time	Maximum simulated time
Time_Scheme	Time scheme to be used; 'Explicit' is only option right now
Restart	Unique sequence of numbers found in the file name to restart from; decimals can be in sequence; if not restarting, indicate 'None'
Convergence	Convergence criteria when obtaining temperature (if specific heat is temperature dependent)
Max_iterations	Maximum number of iterations when obtaining temperature (if specific heat is temperature dependent)
Number_Data_Output	Number of data files to output during the simulation
Boundary Conditions - see section 3.1.6	
bc_left_E	Boundary conditions of Equation (2.1) or (2.7) for left boundary (smallest x or r value)
bc_right_E	Boundary conditions of Equation (2.1) or (2.7) for right boundary (largest x or r value)
bc_north_E	Boundary conditions of Equation (2.1) or (2.7) for top boundary (largest y or z value)
bc_south_E	Boundary conditions of Equation (2.1) or (2.7) for bottom boundary (smallest y or z value)
bc_left_rad	Radiation boundary conditions of Equation (2.1) or (2.7) for left boundary (smallest x or r value)
bc_right_rad	Radiation boundary conditions of Equation (2.1) or (2.7) for right boundary (largest x or r value)
bc_north_rad	Radiation boundary conditions of Equation (2.1) or (2.7) for top boundary (largest y or z value)

Table 3.1: (continued)

Parameter	**Description**
bc_south_rad	Radiation boundary conditions of Equation (2.1) or (2.7) for bottom boundary (smallest y or z value)
bc_left_P	Pressure boundary conditions for left boundary (smallest x or r value)
bc_right_P	Pressure boundary conditions for right boundary (largest x or r value)
bc_north_P	Pressure boundary conditions for top boundary (largest y or z value)
bc_south_P	Pressure boundary conditions for bottom boundary (smallest y or z value)
Interpolation Schemes - see section 3.1.7	
diff_interpolation	Interpolation scheme for diffusion terms of governing equations
conv_interpolation	Interpolation scheme for convective terms of governing equations

3.1.1 Modes and Domains

There are two modes: **Heat** and **Species** modes which are specified with the *Model* input. The equations solved in each mode are in sections 2.1.1 and 2.1.2. In either mode, a 2D rectangular or axisymmetric geometry can be solved for the equations. The x coordinate becomes r in axisymmetric and y becomes z. To select either domain, use *Domain* input to specify 'Planar' or 'Axisymmetric'.

Meshing options are coded into the solver, however the solver does not properly account for non-uniform discretizations in a given direction. The discretization sizes in x and y can be different, but are uniform throughout the domain. These options MUST be left as 'None':

```
bias_type_x:None
bias_type_y:None
```

Failure to do so will result in errors in calculations. The framework was put into the code during development, but was abandoned due to time constraints of the research. The other biasing options can be any number since they will not be referenced as long as the previous inputs are 'None':

```
bias_size_x:0.003
bias_size_y:1e-06
```

3.1.2 Thermal Properties

The properties of specific heat and thermal conductivity in equations (2.1) or (2.7) can be a constant, temperature based or reaction progress, η based. Specific heat can be calculated based on reaction progress:

$$C_v = C_{v,0}\left(1-\eta\right) + C_{v,1}\left(\eta\right) \tag{3.1}$$

This would be specified in the input file like:

```
Cv_s:eta,500,800
```

where 500 is $C_{v,0}$ and 800 is $C_{v,1}$. Note that the Arrhenius source term must be active (section 3.1.4) otherwise reaction progress remains zero always. Another method would be temperature based for a specified species in the **MatClasses** class. This can take one of two forms:

```
Cv_g:Air,Temp
Cv_g:O2,Temp,600
```

where the first line specifies temperature dependent specific heat of air and the second line specifies the specific heat of oxygen gas at 600 K. The temperature dependent options require the desired species data to be in **MatClasses**. As of now, the only species available are aluminum (solid, liquid or gas states), copper (solid,

liquid or gas states), alumina (solid and liquid states), copper(II) oxide (solid or gas states), argon, air and oxygen gas. See section 4.2.4 for more details. If temperature dependent options are chosen, the inputs *Convergence* and *Max_iterations* are used when iterating to determine the temperature.

For thermal conductivity, the same options are available; constant, temperature or reaction progress based. As of now, the only temperature dependent data in **MatClasses** is for air or oxygen gas.

Heat *Cv_s* is the only input read by the solver regarding the specific heat of solid, $C_{v,s}$. λ_{eff} is calculated only based on the input *k_s*.

Species $C_{v,s}$, $C_{v,g}$ and $C_{p,g}$ are read in through the input values for *Cv_s*, *Cv_g* and *Cp_g* respectively. λ_{eff} is calculated based on the input *k_s*, *k_g* and *k_model*. *k_s* and *k_g* specify the thermal conductivities of the *solid* and *gas* phases respectively. When *k_model* is 'Parallel', the effective thermal conductivity in Equation (2.7) is calculated by:

$$\lambda_{eff} = k_g\phi + k_s(1-\phi) \tag{3.2}$$

For 'Series':

$$\lambda_{eff} = \left(\frac{\phi}{k_g} + \frac{(1-\phi)}{k_s}\right)^{-1} \tag{3.3}$$

For 'Geometric':

$$\lambda_{eff} = k_s\left(\frac{k_g}{k_s}\right)^{\phi} \tag{3.4}$$

3.1.3 Porous Medium

Porosity has been seen in all the previous section because it is a characteristic of a porous medium. It has an impact on the density, thermal properties, permeability and pressure. Porosity, ϕ, is a constant as specified in the input file by *Porosity*. This value can be 0 if desired, but should never be 1.

Heat Porosity will be used to calculate the density of the solid by:

$$\rho_{s,0} = (1 - \phi)\rho_0 \tag{3.5}$$

where ρ_0 is specified by *rho_IC*.

Species Porosity will be used to calculate the initial densities of the *gas* and *solid* by:

$$\rho_{g,0} = \phi\rho_1 \tag{3.6}$$

$$\rho_{s,0} = (1 - \phi)\rho_2 \tag{3.7}$$

where ρ_1 and ρ_2 are the first and second entry of *rho_IC*, respectively. The permeability (Equation (2.9)) requires inputs for d_c and k_k via *Carmen_diam* and *Kozeny_const* respectively. R in the ideal gas law (Equation (2.10) is specified in the input file with *gas_constant*.

3.1.4 Source Terms

The value of $\dot{Q}_{source}$ has two options in either **Heat** or **Species** modes. The first is uniformly generated which is specified by the option *Source_Uniform*. If deactivated, it should read 'None', otherwise specify a number (in $\mathrm{W\,m^{-3}}$).

The other option is of an Arrhenius form given by Equations (2.2) and (2.3):

$$\dot{Q}_{source} = \rho_{s,0}\dot{\eta}\Delta H$$

$$\dot{\eta} = A_0(1 - \eta)\exp\left(-\frac{E_a}{R_u T}\right)$$

where $\rho_{s,0}$ is the initial density of *solid*, ΔH is the enthalpy of combustion, A_0 is the pre-exponential factor, η is the reaction progress variable, E_a is the activation energy, and R_u is the universal gas constant (8.314). E_a and A_0 are specified constants from inputs *Ea* and *A0* respectively. ΔH can be specified on a density or volume basis as seen below:

```
dH:rho,2.78e6
dH:vol,3e5
```

When this option is active, the option *Ignition* will be checked to see if ignition has occurred. It is specified as:

```
Ignition:10,15
```

where the source term must be 10 times the conduction/boundary condition losses and must occur at 15 nodes. Refer to section 2.2.3 for more information. Once this condition is met, the time is recorded, all variables are saved to the output directory and the north boundary condition is changed to that of the right boundary.

Species When the species mode is active, the Arrhenius source term is automatically active and η represents the conversion of the solid species to gas species (see section 2.1.2 for more information). The final amount of *gas* is expressed as a percentage of the initial amount of *solid*:

$$\rho_{g,f} = c\rho_{s,0} \tag{3.8}$$

where c is based on the input *gas_gen*.

3.1.5 Time Stepping

Heat Time steps sizes can be specified through the Fourier number or time step size via inputs *Fo* or *dt*, respectively. If both are specified, the smallest time step value is used. The final time or number of time steps can be specified by *total_time* or *total_time_steps* respectively. If both are specified, the solver will follow the specified number of time steps.

Species Time steps sizes can be specified through the Fourier or CFL numbers or time step size via inputs *Fo*, *CFL* or *dt*, respectively. If all are specified, the smallest time step value is used. The final time or number of time steps can be specified

by *total_time* or *total_time_steps* respectively. If both are specified, the solver will follow the specified number of time steps. When implementing both CFL and Fo, it is clear that the Fourier number is more restrictive, however tests have shown that the Darcy velocity can reaches very high values that it impacts stability.

3.1.6 Boundary Conditions

Being a 2D code, there are 4 boundaries; north, south, left and right. Why didn't I just call them east and west? I don't know if I can answer that looking back; I tend to complicate things sometimes. The left boundary is where $x = 0$ or $r = 0$ and the south boundary is where $y = 0$ or $z = 0$.

This code allows for multiple boundary conditions to be specified along a given side. A boundary condition, for any equation, is specified in the following format: *[Type]*, *[values]*, *[first node number]*, *[final node number]*, ...

where
[Type] is the type of boundary condition
[values] is the value associated with that boundary condition
[first node number] is the node number where this boundary condition begins
[final node number] is the node number where this boundary condition ends

Each character or number must be separated by commas. The following example for the north boundary specifies a $200\,\mathrm{MW\,m^{-2}}$ flux on nodes 0 to 20 and convective boundary condition (coefficient of $30\,\mathrm{W\,m^{-2}\,K^{-1}}$ to ambient of $300\,\mathrm{K}$) from nodes 21 to 100. For the axisymmetric solver, the left boundary will automatically be a zero flux boundary condition unless a constant temperature is specified.

```
bc_north_E:F, 200e6, 0, 20,C, 30, 300, 20, 100
```

Radiation boundary conditions are possible by specifying the format as: *[emissivity], [surrounding temperature]* if active or 'None' if not active. The following example uses an emissivity of 0.4 to a surrounding temperature of $298\,\mathrm{K}$ on the right

boundary.

```
bc_right_rad:0.4, 298
```

For equations (2.1) or (2.7), *[Type]* can be T for a constant temperature, F is a constant flux, and C is a convective flux. *[values]* would specify the constant temperature or flux for the first two types or the heat transfer coefficient and free stream temperature in the case of a convective boundary condition (separated by commas).

Species For Equation (2.6), the pressure dictates the boundary conditions. The options for *[Type]* are P for constant pressure, *grad* for pressure gradient or *none* for no condition. *[values]* simply specifies the value of the pressure or gradient. These boundary conditions are implemented in a way such that mass leaves the boundary in order to satisfy the given condition (except when no condition is specified where nothing happens).

3.1.7 Interpolation Schemes

The calculation of properties for the convective or diffusion terms in the governing equations requires an approximation scheme. There are two options in this code: *Harmonic* or *Linear*. The *Linear* scheme is a simple averaging of neighbouring node values (linear variation between control volumes) to approximate the values at the control surfaces as is given by:

$$\rho_{i+\frac{1}{2}} = \frac{\rho_{i+1} + \rho_i}{2} \tag{3.9}$$

The *Harmonic* scheme (parallel thermal resistances formula) is given by:

$$\rho_{i+\frac{1}{2}} = \frac{\rho_{i+1}\rho_i}{\rho_{i+1} + \rho_i} \tag{3.10}$$

The inputs *diff_interpolation* and *conv_interpolation* indicate the interpolation schemes for the effective thermal conductivity in section 2.2.1 and the convective terms outlined in section 2.2.1, respectively. The intention was to allow different interpolation schemes for the properties that are spatially dependent. Upwind-Difference Scheme (UDS) was an alternate scheme that I would've coded if I had time.

3.2 Code Execution

This code was used extensively in Windows, but can be used in Unix (see section 4.2.1); use at your own risk. To run the code, open a terminal at the directory where the code files are located. Next, activate the necessary Python environment (if using Anaconda). Then run the following line:

```
mpiexec -n [n] python main.py [Input file] [Output directory]
```

where:
[n] is the number of processes to run on
[Input file] is the relative path and the name of the input file
[Output directory] is the relative path to the desired directory to output the data.

In the following example, 6 processes will be used to solve the problem outlined in the input file named *Input_File_axi.txt* located in the current directory and the data sent to the relative directory Tests\1:

```
mpiexec -n 6 python main.py Input_File_axi.txt Tests\1
```

If the output directory does not exist, it will be created. If you wish to suppress Python warning messages, use '-W ignore' between 'python' and 'main.py'.

3.3 Output

As the code executes, messages are printed to the console. At the beginning of every 1000 time steps, the time step number, time step size and total time elapsed

are output to the console. When ignition occurs, a message 'Ignition occurred at t=' will also be output with the time elapsed. When the solver stops (successful or divergence), the solve time per 1000 time steps, total time steps, final time step size, ignition time and average wave speed are output. If divergence occurred, a message is output with the error code is also output.

As the variable data is saved during the solve, the input file will be modified to contain transient information. The name of this input file is fixed to *Input_file.txt*. If this name is changed, post-processing will not work. The transient information include the wave speed calculated att that time step and cumulative average to that point. When the solver stops (successful or divergence), the same information sent to the console is also sent to the input file.

After the main script reads in the inputs from the input file, they are saved into the appropriate dictionaries for solving but also output to another text file. All information related to divergence, solver time and number of time steps are also output to the file. Below is an example of output data to the text file after a run diverged:

```
#################### Solver aborted ######################
Time step 1948879, Time elapsed=0.000109, error code=2;
Error codes: 1-time step, 2-Energy, 3-reaction progress, 4-Species
balance
Final time step size: 0.001000 ms
Ignition time: 0.100822 ms
Solver time per 1000 time steps: 0.010371 min
Total time steps: 1948879
Average wave speed: 21.41 m/s
```

In the following example, the solver completed successfully, but the Arrhenius source term was not active:

```
Final time step size: 328.312882 ms
Ignition time: 0.000000 ms
Solver time per 1000 time steps: 0.004712 min
```

```
Total time steps: 10000
Average wave speed: 0 m/s
```

The variable data is output as numpy array files in the form:

```
[q]_[time].npy
```

where [q] is the variable name and [time] is the time in the simulation in milliseconds. Temperature (T), reaction progress (eta), density of the solid (rho_s), density of gas (rho_g) and pressure (P) are the possible variable names.

Heat Temperature is output by default, but reaction progress will also be output if the Arrhenius source term is used.

Species Temperature, densities of each species, pressure and reaction progress are output.

3.4 Post-Processing

Post-processing has its own input file to specify details of the contour plots of the variables. Like the solver input file, any line beginning with # will be ignored. The inputs used are summarized in the table below:

Table 3.2: Comprehensive list of post-processing input file parameters

Parameter	**Description**
Directory	Relative directory (from where **Post.py** is) to variable data
Times	Time portion of file names to be processed (must be exact matches). Use *ALL* for all times in directory.
x_min	Smallest x value to plot of domain.

Table 3.2: (continued)

Parameter	**Description**
x_max	Largest x value to plot of domain. Use *ALL* for all x.
y_min	Smallest y value to plot of domain.
y_max	Largest y value to plot of domain. Use *ALL* for all y.
Temp_min	Minimum temperature for colour plot to colour.
Temp_max	Maximum temperature for colour plot to colour.
Temp_pts	Number of temperature points to indicate between minimum and maximum temperatures.
eta_pts	Number of contour plot legend points for reaction progress contour plots
1D_Plots	Output variable data along centreline of domain. Indicate 'None' if not desired.
Phi_Plots	Output reaction rate contours according to equation (2.3). Indicate 'None' if not desired.
Darcy_vel	Output Darcy velocity contours 1 each for u and v. Indicate 'None' if not desired. *Contour_Plots* must indicate True to occur.
Contour_Plots	Output contour plots of all default and selected variables.
Time_Temp_Pos	Node numbers for position in domain to plot a temperature-time profile (**Post_timeEvolv.py** file only)
Variable	Variable to plot using **Post_timeEvolv.py**. Options include Temperature, Pressure, eta, rho_g, rho_s.

A file called 'Post_processing.txt' is output when post-processing. The directory

of the data is written as the second line. As each time is processed, the time in milliseconds and percentage mass loss is output. If the Darcy velocity contours are selected, the maximum and minimum velocity values are also output. After all times are processed, maximum pressure, average density of *gas*, average axial Darcy velocity obtained in the these times are output. The Peclet, Damköhler, non-dimensional burn rate and ignition delay are output last. If no contours are output, these numbers are still output to the file. If the file already exists from a previous run, it will be modified below the existing output. The following example processed 3 times with no Darcy velocity contours:

```
Post processing results:
..\Nanothermite\Data\1_base\4

Time = 0.560001
     Mass balance residual: -0.102
Time = 0.600000
     Mass balance residual: -0.113
Time = 0.640001
     Mass balance residual: -0.125

Max pressure: 268900043
avg rho_g: 55.3573
avg Darcy v: 0.4553
Pe_y: 0.000000470359
Da_y: 205142503.231598
Burn rate: 49223.013727
Ignition delay: 0.002069
```

Chapter 4

Code Files and Structure

This chapter contains the lower-level details about how the code operates. The main packages required are numpy, mpi4py, string, copy, os, sys and time. There should also be some multi-thread program installed on the computer for the code to be run in parallel. Please note that the code was developed in Windows and only partially tested during development on a Linux based system. Using Unix based systems should be approached with caution. If using Unix, it is imperative to view section 4.2.1 otherwise input files will not be read properly.

This code is parallel in nature and can be run on supercomputer clusters. My first runs were run on a 6 core gaming computer which were sufficient for 60 000 nodes, but then I used SciNet (Niagara) based out of the University of Toronto, Canada for 300 000 nodes. The modules required were *intel/2018.2*, *intelmpi/2018.2* and *intelpython2/2018.2*. Runs were done with either 20 or 40 cores; using anymore resulted in significantly slower solving times.

4.1 Variable Types and Terminology

Most of the code is built around manipulating 2D-numpy float arrays with each time step. All variables and properties that are spatially dependent (energy variable, coordinates, discretizations, thermal conductivity, pressure, temperature etc.) have

their own 2D numpy array representing that variable in the whole simulation domain.

Most options (axisymmetric domain, interpolation routine etc.) are stored as individual strings. Lists are also common for specification of an input such as specific heat (e.g. reaction progress dependent specific heat from section 3.1.2) or the boundary conditions for a given face.

Dictionaries are used to store all inputs from the input file. Once the input file is read, they are stored in one of four dictionaries (more details in section 4.2.1). Boundary conditions are contained in one dictionary with a keyed index for each boundary and equation (one, each, for the energy and pressure boundary conditions on each face). In each entry is a list containing the boundary condition information as it is specified in the input file (see section 3.1.6).

In this section, certain terminologies will be thrown around in describing how the code operates. The 'global' domain refers to the simulation domain as defined in the input file. If MPI features are used, each process (CPU core with a rank number) will solve the equations for a part of the 'global' domain or their 'local' domain. The 'adjacent process' refers to the process simulating a part of the 'global domain' that is outside its 'local domain' in a particular direction.

4.2 Classes and Functions

The code structure begins with the main script (**main.py**). Each stage outlined in Figure 2.2 involves one of a few class files, but everything starts and ends with the main script. All settings for the solver are defined here and passed into the appropriate classes for processing. This file calls the solver function for each time step until the number of time steps or total time has been achieved based on the input file. Depending on the frequency defined in the input file, the variables are output at required intervals from this script. They are output as numpy array files to be post-processed by the post-processing script (described in later section). When ignition occurs, the main script changes the boundary condition at the north boundary (section 3.1.4).

Solver time is calculated by recording the start time of the script execution and

when the solver is finished. Note that the global domain will refer to the geometry intended to be simulated while the local domain will be the portion of the global domain that a given process will solve.

This section will describe each file in detail. All files except **FileClasses** and **MatClasses** contain one class, so individual subsections are allocated for each function.

4.2.1 FileClasses.py

The required classes and functions to read the input file and copy them to the output file are contained in this file. All settings from the input file are read and saved to one of four dictionary variables and sent back to the main script. The name of the variable in the input file is also the key that is used in each dictionaries and summarized in lists at the top of the Python file. The *Species* dictionary contains the gas phase inputs for specific heat and thermal conductivity (section 3.1.2). The *Sources* dictionary contains all inputs from section 3.1.4. *BCs* contains all boundary conditions from section 3.1.6. The *settings* dictionary contains all other inputs not found in the other dictionaries.

LINUX NOTE: Due to differences in Linux and Windows having ending lines with CRLF versus LF, this file must be modified if the code is to be run in a Unix based operating system. The variable *newline_check* should be modified to be the carriage return character '/r', then the input file will be read correctly and the rest of the code should function. If the input file is created in Linux, this change might not be needed. Please proceed with caution when using Linux.

FileIn

This class handles reading the input file and saving the data to the appropriate dictionary. Upon declaration, a read-only file stream is opened and the input file is read in one line at a time. The only function is **Read_Input** which will read the line and based on the key (for dictionary) convert them to the desired variable type (float, integer etc.) and save them to their appropriate dictionary. The file stream is

then closed. Any line beginning with # is ignored and the function reads the next line.

FileOut

This class outputs a copy of the input parameters to another file. Upon declaration, an write-only stream is created an opened.

Write_single_line: This function takes in a string and writes it to the filestream with a newline character '/n' at the end. The rest of this class will use this function instead of using the python function 'print'.

header_cond: Prints a header on the output file. At the moment, it is mostly # characters with a couple mentions of myself, type of code and a title specified in the main script. This function is called from the main script, not within this class. I had a more customizable message in mind, but never got around to doing anything with it.

input_writer_cond: This function receives all four dictionaries from the main script and writes them to the output file.

close: This function closes the file stream.

4.2.2 GeomClasses.py

The domain class is contained in this file. In this class, all numpy arrays of all variables including outputs of the simulation (pressure, energy storage term, thermal conductivity, reaction progress, permeability, etc.) and characteristics of the domain (x and y coordinate arrays, discretization arrays dx and dy). All settings pertaining to thermal property calculation are stored as well. Parallel coding data including the adjacent process numbers, global process arrangement and rank are also saved. Upon declaration, parallel coding variables are declared, but only the coordinate

and discretization arrays and energy stoarge variables are created. These arrays are modified by the functions in mpi_routines.py and the remaining variables are created with **create_var**. The functions within this class are described below.

mesh

This function creates the coordinate and discretization arrays depending on biasing requirements from the *settings* dictionary. Currently, this function does work and will output a non-uniform discretization grid if specified, however accounting for a non-uniform grid are not fully implemented in the solver class (**SolverClasses.py**), so results will likely be wrong. The options that coded here include specifying the smallest element size and having the sizes linear grow in size from one end to the other. They can also grow from the middle outwards or outside inwards.

create_var

During the initial setup, this function will create all the remaining numpy variable arrays. This function exists because only some of the necessary arrays are created upon declaration. After all processes have been assigned a part of the global domain to solve, their respective coordinate and discretization arrays are split up accordingly. This function is then called to create the remaining variables (reaction progress, pressure, density) with the correct sizes for the arrays.

CV_dim

This function returns the dimensions (individually) of control volumes in each direction. h_x is the width wise dimensions and h_y are the length wise dimensions.

calcProp

Since the energy storage term is saved, this function returns the temperature field. Given the options previously outlined for thermal properties, the density and specific heat are calculated based on the specified options before calculating the temperature

field. Once calculated, the temperature field is used to calculate the specific heat $C_{p,g}$ and thermal conductivities to be used in the solver. This function will return the temperature field, heat capacity (ρC_{eff}), effective thermal conductivity and specific heat $C_{p,g}$. For specific heat or thermal conductivity using species in **MatClasses.py**, the appropriate class from **MatClasses.py** is used to calculate said properties. If temperature dependency is specified, an iterative procedure is done until a converged temperature is obtained.

4.2.3 mpi_routines.py

This file contains the routines needed to discretize the global domain among available processes as well as communication between each process. All functions in this file are called from the main script.

split_var

This function splits a specified domain class variable based on where its local domain is relative to the global domain. Using the position of the local domain relative to the global domain, the appropriate slice of that variable is saved to the specified local domain class. This is used during the initial MPI setup before the solver begins.

MPI_discretize

When the domain objects are first declared, each process contains a domain class with the global domain. This function will determine the best way to discretize the global domain, uniformly, into the given number of processes. More information is in section 4.3. If a problem occurs, it will throw an error to the main script halting all processes.

If successful, **split_var** is called to split the local coordinate arrays, energy storage term and discretization arrays for each process' domain class according to its local domain. Based on its position in the global domain, the ranks its adjacent process numbers are saved to the variables in the domain class. If it has a boundary on that side, the value is -1.

update_ghosts

With the global domain evenly divided up, each local boundary shared with an adjacent process will have an array of 'ghost' nodes that contain data from the adjacent process. This function updates those nodes for all variables (temperature, pressure, densities etc.) and is called from the main script before each solve run. Each process sends and receives data to its neighbouring process so flux values are properly calculated during the solve run.

compile_var

This function compiles a variable (such as temperature, reaction progress etc.) from each process into a single array that reflects the global domain. The numpy function *block* is used extensively here. All processes at the beginning of a row of processes (corresponds to lowest x coordinate in global domain) will append a local variable array (only exists in this function) to accumulate the data from that row of processes. Process 0 will then append its array with all the other process data. The final array is returned to process 0 only.

save_data

This function will call **compile_var** to compile each variable (depending on mode; Heat or Species) and output to the working directory as numpy files (.npy). If the Species mode is active, all variables are saved to the directory. If in Heat mode, only temperature is saved. If Heat mode with Arrhenius source term, temperature and reaction progress are saved.

4.2.4 MatClasses.py

The classes to calculate specific heats or thermal conductivity are contained in this file. There is also a diffusion coefficient class with similar structure, but it is not used in the solver.

Cp

This is the specific heat class containing functions for calculating specific heats of different species. There is a central function (**get_Cp** or **get_Cv**) that calls the appropriate function to calculate the specific heat depending on the specified setup. There are separate functions to calculate the specific heat at constant pressure (**get_Cp**) or constant volume (**get_Cv**). The specific heat data contained are for aluminum (solid, liquid or gas states), copper (solid, liquid or gas states), alumina (solid and liquid states), copper(II) oxide (solid or gas state; data is sketchy), argon, air and oxygen gas. The specific heat data varies with temperature and are fit with standard NASA fourth order polynomials obtained from the JANAF tables or textbooks. The origin of the data is commented in each function.

therm_cond

This is the class for thermal conductivity, which only holds data for air and argon as a function of temperature. It is called in the same way as specific heat with **get_k**. Currently, only argon and air are the only species that are available.

4.2.5 SolverClasses.py

This file contains the solver class to solve all governing equations. This class requires the domain class and *settings* and *Sources* dictionaries upon initialization in order to directly access the domain variables and its functions. Functions include **mult_BCs**, **getdt**, **interpolate**, and **Advance_Soln_Cond**.

mult_BCs

This function converts the global domain boundary condition dictionary into boundary conditions for the local domain. Any local boundary that is shared with another process (not a global boundary) is set to a zero flux boundary along the entire boundary. Boundary conditions can vary along a given boundary, so each boundary

condition is read and determined whether it applies to this local domain. The relevant conditions are saved in the boundary condition dictionary for the local domain.

getdt

This function determines the time step for this given local domain. It requires the thermal conductivity, heat capacity and Darcy velocities as inputs. Using the settings for *Fo*, *CFL* and *dt*, the smallest time step is chosen and output. Any Darcy velocities of 0 are set to 10^{-9} (in this function, not for the simulation) to ensure no divide by zero errors.

interpolate

This is the function to interpolate properties at control faces. The options available are outlined in section 3.1.7. This function is used extensively during solving of the equations.

Advance_Soln_Cond

This is the main function that solves the conservation equations (mass, momentum and energy) and advances time one time step. It references the domain variables and modifies them directly after copying them if needed. In solving each time step, the time step is calculated with **getdt** and then the smallest value between processes is broadcast to all processes. If the time step is negative or infinite, an error code is output to the main script which will then be sent to the input file copy at the working directory.

Diffusion and convective terms (where applicable) are calculated directly for incoming and outgoing fluxes for each control volume. Interpolation for the flux function is done by **interpolate**. After solving the equations, the source terms are added to the conservative variable. The source terms are calculated from functions in **Source_Comb.py** and will be described in another section. After the source terms, the boundary conditions are applied from a function inside **BCClasses.py**.

After boundary conditions have been applied, the variables are checked for signs of divergence or instabilities. Conditions checked are if the energy is infinite or less than or equal to zero, or reaction progress variable is negative or above 1. There are checks for density of the species being infinite or negative, but they are commented out due to being unreliable. If any condition is met, the solver sends an error to the main script which will cancel the rest of the solve.

4.2.6 BCClasses.py

This class contains all boundary condition functions for each variable or conservative equation. Each function has the same architecture, but is specific to temperature or pressure. For initialization, the boundary condition dictionary, spatial discretizations and domain type (planar or axisymmetric) are needed.

Energy

This function applies the boundary conditions for the energy storage variable. Each boundary condition is applied sequentially, using for loops, along the boundary based on the boundary condition dictionary. A zero heat flux is implied on the left boundary for an axisymmetric case unless a constant temperature is specified.

P

This function calculates the amount of *gas* phase that must be removed at the boundary to enforce the specified pressure boundary condition. The pressure and temperature inputs (from solver) is compared to the boundary condition, and then the difference in density returned to the solver where the associated mass and energy loss is applied. For an axisymmetric domain, nothing is done with the left boundary.

4.2.7 Source_Comb.py

This file contains the required functions to calculate the source terms for energy and mass equations. During initialization, the appropriate source term information from

the *Sources* dictionary is saved as object variables. The uniform generation source term is calculated by passing in the generation value to **Source_Uniform**. The resulting value is returned. For the Arrhenius source term, the density, temperature and reaction progress are passed in from the solver class to **Source_Comb_Kim**. The resulting numpy array of values is returned to the solver class. For the mass source terms (Species mode), the reaction rate from the Arrhenius source term calculation and the total initial mass are passed in to **Source_mass**. The resulting mass generated/destroyed for gas and solid phases are then returned as a tuple in that order.

4.2.8 Post.py

Since the variables are output as numpy array files for certain time steps, this script will read in the variable and output a formatted contour plot and post-processing file. An input file is required to specify the directory with the numpy files, times steps to process, graphs to output and certain formats of the contours. The times to process must match the time stamps on the file name EXACTLY, otherwise an error will occur. When the Heat mode was used, the temperature is the only output, so only temperature contours or 1D plots can be output from this script. When in the Species mode, contours can be output of temperature, pressure, densities, reaction progress, reaction rate and Darcy velocities. The coordinate bounds on the contours can be specified to only display a specific clip of the domain. The maximum, minimum temperatures and number of colorbar bins for the temperature contour can be customized. All contours except the reaction rate and Darcy velocity plots are automatic (when contours are set to be output); these need to be set to output. The post-processing file outputs a mass balance residual at each time step processed which is the total mass missing normalized with the initial mass. The maximum magnitudes of the Darcy velocities are also output to the console with each time step if the contours were output too. After each processed time, a summary of the characteristic variables are output at the end of the post-processing file.

4.2.9 Post_timeEvolv.py

This post-processing tool plots a 1D graph showing a time evolution of temperature and another variable at a specific location in the global domain. Like **Post.py**, an input file is used to specify the directory, times to process, domain location (specified by node numbers) and second variable. The second variable can be pressure, densities or reaction progress.

4.3 Parallelizing Routines

The code is parallelized, all printed messages to the console will occur through the process with rank 0 (herein referred to as process 0). Any communication between processes is also done through process 0. The Python package *mpi4py* was used to allow parallel programming.

4.3.1 Process Dividing

This routine works by listing all process ranks (0 to $n-1$ ranks where n is the number of processes) in a 1D numpy array and then reshaping the array to a 2D array. For example, if 6 processes are to be used, a 1D array of length 6 will be created containing the numbers from 0 to 5 (ranks of each process). This array is then reshaped using the numpy *reshape* function with the column dimension (m) being the truncated square root of the number of processes plus one. For 6 processes, the column dimension would be $m = 3$, which would result in a row dimension of $n = 2$ (n x m arrangement). This is the first proposed process arrangement of processes to divide the domain into with m being the number processes in x and n the number of processes being the division in y. n processes in x and m processes in y is checked second. The dimensions of this 2D array determines the local domain size of each process in each direction.

These dimensions are then tested with the number of nodes in x and y to ensure they can be divided evenly. If the domain grid is 120 nodes in x and y, then the 2 by 3 arrangement would work because 120 is divisible by 2 and 3. If this test fails in

any one dimension, then m is reduced by 1, and the process arrangement is reshaped again. For this example, the 2 by 3 arrangement would become 3 by 2 and the check is done again. This process continues until the first arrangement passes this check. If m becomes 0, then the solver exits with an error message sent to all processes. The number of nodes in x and y needs to be chosen carefully for a given number of processes.

Since boundary condition can vary along a boundary, a function was added to the SolverClasses.py to ensure the proper boundary conditions dictionary is set in BCClasses.py.

4.3.2 Communication During Solve

When the main script is executing, all processes contain the relevant data for the local domain within the previously described classes. Interior processes contain an extra row/column of data that act as ghost nodes to represent the missing domain data from the adjacent processes. The ghost nodes for all processes are updated from the main script before entering into the solver.

Once the solver is called, the time step is calculated first. Since each process might calculate a different time step, each process must communicate this with all the other processes so a single time step is used to advance the solution. Each process will calculate a time step, and then process 0 will collect all the values and broadcast the minimum value to all processes using *reduce* and *bcast* functions.

This same procedure occurs with error checking at the end of each time advancement. If divergence occurs at one process, the error codes are collected in process 0 and the maximum error code is broadcast to all processes so the run halts on all processes.

Ignition can occur at any process, so the minimum and maximum ignition variable states are broadcast to each process. When the minimum is 0 (no ignition before this time step) and the maximum is 1 (ignition occurred somewhere), then the boundary conditions will changed where appropriate and the ignition variable will be set to 1 and broad cast to all processes.

4.3.3 Compiling Variable Data

When variable data needs to be saved as per the input file settings, each process at the left boundary (near $x = 0$), will compile the variable data (not including ghost nodes) from that row of processes using the numpy *block* function. This will yield full data sets in x for a given y range. Each process that compiled the variable data will then send it to process 0 where it will be saved to numpy files in the output directory.

To calculate the wave speed, the variable compiling routine was done for the reaction progress variable and then integrated in y to obtain the wave speed.

Bibliography

[1] F. Saceleanu, M. Idir, N. Chaumeix, and J. Z. Wen, "Combustion Characteristics of Physically Mixed 40 nm Aluminum/Copper Oxide Nanothermites Using Laser Ignition," *Frontiers in Chemistry*, vol. 6, pp. 1–10, 2018.

[2] Epps, Joseph, "Continuum modelling of al/cuo nanothermite pellet combustion," 2020.

[3] S. Fischer and M. Grubelich, "A survey of combustible metals thermites and intermetallics for pyrotechnic applications," in *32nd AIAA Joint Propulsion Conference*, p. 15, 07 1996.

[4] M. M. Kaviany, *Principles of heat transfer in porous media*. Mechanical engineering series, New York: Springer-Verlag, 2nd ed. ed.

[5] T. Ozgumus, M. Mobedi, and U. Ozkol, "Determination of kozeny constant based on porosity and pore to throat size ratio in porous medium with rectangular rods," *Engineering Applications of Computational Fluid Mechanics*, vol. 8, no. 2, pp. 308–318, 2014.