CONTENTS

ABSTRACT

This report presents an algorithm to calculate the equilibrium state of ideal gas species of fixed volume and internal energy by minimizing the system's Gibbs free energy, specifically solving the temperature and chemical species abundance. Two algorithms are considered: the first a reduced non-stoichiometric algorithm based on the formulation of Zelenznik & Gordon and Gordon & McBride and in their CEA program [1] [2], and the second an independently developed stoichiometric algorithm that makes use of the Nelder Mead optimization algorithm. The equilibrium problem for ideal gases is considered to be a solved problem [3], and so this report acts as a reiteration of existing well established methods in the literature rather than an investigation into novel approaches to the problem.

The algorithms and their applications are specifically formulated for equilibrium flow calculations in Eilmer, the University of Queensland's finite volume computational fluid dynamics program. The system constraints and algorithm formulation are provided with this application in mind, and some suggestions are made for effective integration of the algorithms into the main Eilmer program.

The two proposed methods are prototyped, tested and validated against the results of CEA, and are compared to against one another in terms of efficiency, reliability and ease of use, with validation results presented for the ion-free dissociation of nitrogen. The "Lagrangian Newton Raphson" approach, modelled after a simplified version of CEA's iteration algorithm, was found to consistently outperform the stoichiometric algorithm by every relevant metric, and so is selected as the best option for future use and development. This algorithm is found to be fast, reliable, accurate and robust when used in systems that meet the physical conditions that are of interest in equilibrium flow calculations.

1 INTRODUCTION

"Eilmer" (pronounced "ell-merr") is a computational fluid dynamics (CFD) program currently maintained by The University of Queensland (UQ) that uses a finite volume approach to simulate complex fluid flows, including those at high temperature. Chemical reactions can often occur at an appreciable rate in such systems, and so it becomes important to track how the fluid composition evolves throughout the flow. In very high temperature regions, this becomes difficult to achieve practically through direct simulation of the reaction rates, with the integration timescale becoming prohibitively small and the underlying reaction rate data becoming increasingly unreliable [4] [3]. To avoid these pitfalls, Eilmer frequently makes use of an "equilibrium flow" assumption at certain points in the flow, in which the chemical reactions are idealized to be so high that each point in the flow has settled out to the state of local thermodynamic equilibrium.

Figure 1.1-Eilmer Logo

The solving of chemical equilibrium is a separate and well established problem, which can be approached independently of many of the complexities of CFD simulation. At present, Eilmer calculates equilibrium conditions with an external program, NASA's 'Chemical Equilibrium with Applications' (CEA), a publically

available program that can calculate chemical equilibrium for a wide variety of system types. However, this 'outsourcing' of equilibrium calculations is restrictive and unwieldy, and so it is of practical interest to develop an equilibrium calculator integrated into Eilmer itself.

This report outlines the development of a standalone iteration equilibrium solver for ideal gasses, aimed to fulfil this objective. Presented within this report is:

- An overview of the history and types of general chemical equilibrium algorithms;
- An introduction to the principles, both physical and mathematical, required to understand the the basics of the chemical equilibrium problem;
- The formulation of a Newton-Raphson equilibrium solver algorithm, modelled after the process used by CEA;
- The attempted formulation of an alternative algorithm to make use of Eilmer's existing Nelder Mead optimization module; and
- An analysis and comparison of the performance of the two methods.

Calculating equilibrium for a system of ideal gases is generally considered to be a solved problem [3] [5] [6], and so this report does not contain much novel information regarding equilibrium solvers. It is instead intended to act as a focused analysis of the methods best suited to the specific requirements of Eilmer, and as an introductory resource for related work in the future.

2 Review of Literature

In this section, a broad overview of chemical equilibrium solvers is provided: their history, the different types of solvers and which methods are most commonly adopted. This is not, by any measure, intended to be an exhaustive review of the field, and is instead intended to act only as an introduction for the uninitiated to provide an understanding of the already well-established methods and principles relevant to this report and its contents. For the specific physical and mathematical principles at play in this report, see section 4.1.

2.1 An overview of Solution Types

The ability to efficiently calculate chemical equilibrium is a very practical problem, with any number of applications in fields ranging from propulsion to materials science [3] [4]. As such, there is a long standing historical pursuit, closely tied to the development of digital computers [3], to find efficient and generalized algorithms for calculating CEC.

Speaking very broadly, the equilibrium state can be described in and solved for in one of two ways. The first is to consider equilibrium as the state at which the net reaction rate of all chemical species goes to zero; finding which is an exercise in multivariate nonlinear root-finding of the reaction equations [4] [3]:

$$\frac{d}{dT}n_i = \sum_k R_k(\vec{n}, T) \quad \forall\, i$$

However, this approach is hard to generalize beyond moderate temperature systems of low complexity, as it requires precise knowledge of every possible reaction between all simulated chemical species, something that becomes increasingly unreliable at higher temperatures [3] [4]. In this way, it inherits many of the issues associated with time-marching simulation of the reactions.

In the second approach, equilibrium is formulated as the state that minimizes the system's Gibbs Free Energy, while still obeying the systems constraints (e.g. atomic element abundance, volume, internal energy) [4] [5] [3] [7] [1]. Unlike reaction-balancing, this 'energy minimization' approach only requires knowledge of the steady-state thermodynamic data of each chemical species, something that is much more reliably available in the high temperature regime [4] [8] [5].

Speaking generally, a thermodynamic system's free energy is a function of its chemical composition and any two intensive properties, such as temperature or pressure [9]. For single phase systems, such as the well mixed ideal gasses usually considered in Eilmer, it can be shown that this constrained function is convex for any closed system [10] [9]. As such, the problem of equilibrium becomes the straightforward numerical problem of:

- Locating the state (i.e. chemical species abundance and any two state properties) that minimizes free energy, while also:
    - Obeying all constraints on atomic abundance;
    - Obeying any two additional constraints (e.g. fixed temperature, entropy, internal energy); and
    - Ensuring the calculated state is physical (e.g. positive temperature and species abundance)

For the formulation of the free energy and constraints relevant to the problem of finite volume CFD, see section 4.1. This problem rarely has an analytical solution, and must instead be found, even in relatively simple cases, by iterative numerical methods. This brings the equilibrium problem to an exercise in multivariate non-linear numerical optimization [1] [3] [11] [12] [5].

These 'energy minimization' methods are broadly grouped into two categories:

- **Stoichiometric Algorithms**, in which each iteration obeys the system constraints. This uses the system constraints to reduce the degrees of freedom over which optimization is performed.
- **Non-Stoichiometric Algorithms**, in which only the final iteration is required to obey the system constraints, and the interim iterations are unconstrained.

As the number of constraints (i.e. chemical elements) increases, stoichiometric algorithms become increasingly constrained, decreasing the degrees of freedom to optimize over. For a system of 'I' chemical species and 'J' atomic species, a non-stoichiometric algorithm will have:

$$I - J$$

Degrees of freedom to optimize over. Conversely, non-stoichiometric algorithms, which are typically formulated by way of the method of Lagrangian multipliers (see section 4.2.1), become more complex as additional elements are added For 'I' chemical species and 'J' atomic elements, a non-stoichiometric algorithm will, at first analysis, require optimization over:

$$I + J + 2$$

Degrees of freedom, or up to two less if the constraints define the system properties directly (e.g. fixed pressure or temperature).

At first analysis, stoichiometric algorithms would seem to scale better with the system complexity, but this is not true when considering the family of 'reduced' non-stoichiometric algorithms. The structure of the iteration equations for non-stoichiometric algorithms allow them be simplified algebraically such that the computational cost of each iteration scales predominantly with the number of atomic species, a step that is sometimes called 'reduction'. Though the degrees of freedom over which the optimization is performed are unchanged, the cost-dominant part of the iteration now scales with:

$$J + 2$$

Rather than $I + J + 2$. This means that additional chemical species may be added to the system for only a small increase in computational expense.

Solver Scheme	Effect on Cost of…	
	Increasing No. Species *'I'*	Increasing No. Elements *'J'*
Stoichiometric	Increased	Decrease
Non-Stoichiometric	Increased	Increase
Non-Stoichiometric (Reduced)	No effect	Decrease

Table 2.1.1-Simplified verview of Solver Families

This has clear practical benefits: a single set of reacting species will have a fixed number of available elements, but these may be combined in to any number of molecular products. Take, for example, the burning of some hydrocarbon in oxygen: only three elements are present (O,H and C) but dozens of chemical species may be formed in some amount, e.g.

$$O, H, C, O_2, H_2, CO, CO_2, O_3, CH_4, CH_4O \ldots$$

This efficient scaling has made reduced non-stoichiometric CEC solvers the dominant class of all modern practical methods. One such algorithm is presented in this report (section 5.1) as well as a simple non-stoichiometric algorithm for comparison (section 5.2).

2.2 A Brief History of CEC Solvers

The efficient scaling of the reduced non-stoichiometric algorithms has made them the predominant means of approaching the general equilibrium problem since the mid-20th century [3] [5]. The first general purpose algorithm of this type was introduced by Brinkley [8], and was soon after joined by the alternative methods of White [11] and Huff [13]. These three algorithms have since come to be considered foundational works for CEC solvers, with most modern solvers having either directly or indirectly evolved from them in some way [14] [5]. A small set of linear programming algorithms have been developed across the years [15] [16], but these see little widespread use after the establishment of the aforementioned Newton Raphons approach.

Huff, White and Brinkley describe the problem and their approaches to it with very different language, which has historically led to some confusion about their similarity [3]. Brinkley for example speaks of 'reaction constants' despite utilizing a variation of the energy minimization method, while White uses similar language as is used in this report. An overview by Gordon and McBride [2] [7] highlighted the similarities between the three methods, and found that, with some slight modifications, all three were either instances of or could be recovered from the same formulation:

1. Define a free energy state function from the available thermo chemistry data
2. Use the method of Lagrangian multipliers and the prescribed system constraints to develop a system of equations whose mutual solution defines the equilibrium state
3. Location this solution via a Newton Raphson iteration method

This general process is used by Zeleznik and Gordon in the formulation of their CEA program [1], and has been used as the basis of the solver outlined in this project (see section 5.1 for a specific derivation).

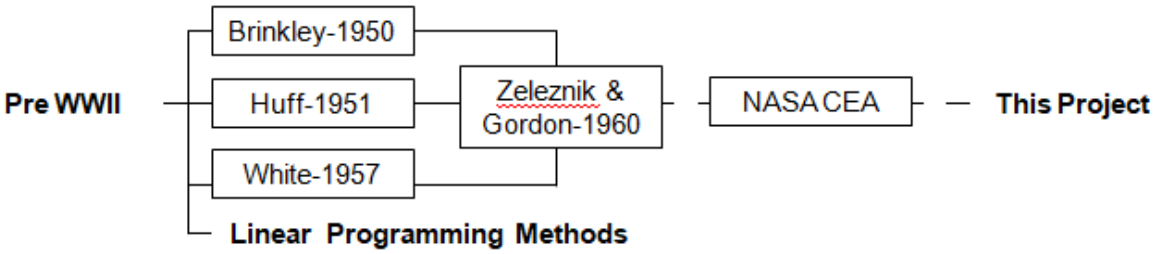

Figure 2.2.1-A Simplified Timeline of CEC Solvers

This general algorithm has been well established since the 1950s, and as such the solving of single-phase equilibrium, particularly for ideal gasses, is considered a 'solved' problem [3] [10] [5]. Despite this general similarity of approach, there remains some freedom as to how specifically how iteration equations are formalized. For example:

- The constraints on system temperature may be introduced in the formulation of the Lagrangian potential, or simply introduced as an additional equation to be solved for during the rootfinding stage
- The system of equations that is solved by the Newton Raphson algorithm may be formalized in any arbitrary way
- The Newton Raphson correction variables may be selected arbitrarily, provided that they adequately represent the degree of freedom

We have specifically adopted the methodology of Gordon & McBride in their CEA program [1], namely their use of logarithmic correction variables and unitless equations in the Newton Raphson rootfinding stage, though some modifications have been made to apply the method to the specific constraints of CFD. This specific approach is referred to as the "Lagrangian Newton Raphson", or "LNR" method.

Development in the field since the establishment of the general method has been focused on dealing with non-ideal behaviour: the mixing of gas and condensed matter, chemical interaction the presence of multiple phases and the effects of miscibility gaps. A major issue facing multi-phase systems is the existence of local minima in the energy function, corresponding to pseudo-stable physical states [16] [5]. There also exist methods for approaching non-ideal behaviour, particularly near-ideal gas mixtures, but these are not discussed here.

3 SCOPE & GOALS

This project is based around successfully accomplishing a single goal: developing a fixed volume/internal energy equilibrium solver for integration in the Eilmer CFD program. To this end, its scope is limited and its goals are strictly design oriented. As a design project, the project has included:

1. The prototyping of both a stoichiometric solver and reduced non-stoichiometric solver in python, with the later in the style of the NASA CEA program
2. The development and refinement of the non-stoichiometric solver in the 'D' programming language
3. The testing of this integrated solver for a number of systems to validate and measure performance

To this end, the report has a limited scope, only examines:

- A specific implementation of the reduced non-stoichiometric algorithm
- A specific stoichiometric formulation for use with the Nelder Mead potential
- Systems of well mixed ideal gasses
- Systems for which the empirical steady state thermodynamic data is readily available

Not included in the scope of this report is any examination of:

1. Any alternate formulations of the reduced non-stoichiometric method
2. Alternate stoichiometric formulations or optimization methods
3. Any investigation of non-ideal, multi-phase or rate limited equilibrium systems
4. Extension of the models into high temperature or high density domains

4 Overview of Theory

This section will act as an outline of the relevant chemical physics and numerical methods that are employed in the two equilibrium solvers presented in this report. Section 4.1 describes the physics of equilibrium and the specific constraint conditions and idealizations that are used in this project's modelling of equilibrium flow, while section 4.2 acts as a reference for the general forms of the numerical methods that are employed throughout the methods proposed in section 5.

4.1 Physical Description of Equilibrium

In the equilibrium flow limit, reaction rates are much faster than the local mass flow rate, making each finite volume element effectively a well-mixed system of fixed (given) volume and internal energy:

$$U = U_0$$
$$V = V_0$$

Each element then must conserve some given abundance of each atomic element present in the system. This element conservation constraint is expressed as a linear sum:

$$b_j = b_{j,0} = \sum_i n_i C_{ij}$$

Where "n_i" is the number of moles of species 'i', "b_j" is the number of moles of element 'j', and "C_{ij}"is the number of atoms of element 'j' in species 'i'. In some sources, this is sometimes expressed in linear algebra form as:

$$\vec{b} = C\vec{n}$$

In this report, we additionally assume that all reaction species are ideal gasses, i.e. following the ideal gas law and law of partial pressures [7] [9]:

$$PV = NRT$$
$$P_i V = n_i RT$$

Where $N = \sum_i n_i$ is the total number of moles in the system. Additionally, the species are assumed to have well defined molar enthalpies (h_i), internal energies (u_i) and standard entropies ($s_{T,i}$) as functions of temperature only, another property of ideal gasses [17] [9] [17] [18]:

$$u_i(T), \qquad h_i(T), \qquad s_{T,i}(T)$$

And the entropy of a species not at standard pressure having a partial pressure based correction [1] [9]:

$$\begin{aligned} s_i &= s_{T,i}(T) - Rln\left|\frac{P_i}{P_0}\right| \\ &= s_{T,i}(T) - Rln\left|\frac{P}{P_0}\frac{n_i}{N}\right| \end{aligned}$$

Where P_0 is the standard pressure (usually 1 bar or 1 atm) for the available thermodynamic data. This also gives the chemical potential (Gibbs free energy per particle) of each species:

$$\begin{aligned} \mu_i &= h_i + Ts_i \\ &= \mu_{T,i} + RT\ ln\left|\frac{P_i}{P_0}\right|, \qquad \mu_{T,i} = h_i + s_{T,i} \end{aligned}$$

Such that the total internal energy, enthalpy, entropy and Gibb's free energy of the system is simply the sum of the contribution from each species:

$$U = \sum_i u_i n_i\,, \qquad G = \sum_i n_i \mu_i$$
$$H = \sum_i h_i n_i\,, \qquad S = \sum_i s_i n_i$$

Where enthalpy 'H' and Gibb's free energy 'G' may also be defined as:

$$H = U + PV$$
$$G = H - TS$$

Note that, because the volume is fixed, the system's entire state, whether obeying the constraints of not, is well defined by the species composition and temperature, i.e.:

$$Sys\ State = Sys\ State\ (\vec{n}, T)$$

In this report, equilibrium is defined as the state $(\vec{n}, T)$ that minimizes the Gibb's free energy, while still being physical (i.e. positive temperature and mole counts) and obeying the conservation of atoms counts and internal energy.

4.1.1 Numerical Approximation of Thermo-Chemical Data

To calculate the equilibrium state of a system of chemicals, reliable and precise information is required about the thermochemical properties of each species. This is simplified by us only considering the ideal gas phase, such that each species requires only a well-defined function of the specific heat capacity:

$$c_p(T)$$

From which the remaining thermodynamic curves can be recovered:

$$s^0(T) = s_0 + \int_{T_0}^{T} \frac{c_p(T')}{T'} dT' \qquad u(T) = h(T) - \frac{PV}{N} = h(T) - RT$$

$$h(T) = h_f + \int_{T_0}^{T} c_p(T')\, dT' \qquad c_v(T) = c_p(T) - R_u$$

T_0=298.15 K, the standard temperature
h_f is the species' standard enthalpy of formation

In testing the CEC solvers presented in this report, three models were used. For simple 'figure of merit' testing, a crude low temperature approximation of constant heat capacity was used:

$$c_p(T) = c_p(T_0)$$
$$h(T) = h_f + c_p(T_0)(T - T_0)$$

Once the prototypes were completed, this model was replaced with the 'Shomate' equations, which use a five parameter fit for the specific heat, and two additional integration constants for enthalpy and entropy [18].

$$c_p(T) = A + Bt + Ct^2 + Dt^3 + \frac{E}{t^2},$$
$$t = \frac{T}{1000\ K} \qquad h(T) = h_f + 1000 \times \left[At + \frac{Bt^2}{2} + \frac{Ct^3}{3} + \frac{Dt^4}{4} - \frac{E}{t} + F - H\right]$$
$$s^0(T) = Aln|t| + Bt + \frac{Ct^2}{2} + \frac{Dt^3}{3} - \frac{E}{2t^2} + G$$

These seven parameters are held constant over a fixed range , e.g. $A = 28.98\ for\ T \in [200\ K, 500\ K]$, which the full curve for each function being a piecewise function. These 'brackets' typically extend up to ~20,000 K, and the full set of coefficients for selected species (in molar SI units) can be found in appendix C In instances where the temperature was above this range, the specific heat was approximated to be constant and held at $c_p = c_p(T = 20{,}000\ K)$.

Though most testing was carried out with the Shomate equations, the 'Glenn' coefficients were adopted for validation against CEA, as this is the model that CEA uses [1]. This model is extremely similar to the Shomate equations, but make use of an additional T^{-2} term in the description of c_p [17]:

$$\frac{c_p(T)}{R} = a_1 T^{-2} + a_2 T + a_3 + a_4 T + a_5 T^2 + a_6 T^3 + a_7 T^4$$
$$\frac{h(T)}{RT} = \frac{h_f}{RT} - \frac{a_1}{T^2} + a_2 \frac{ln(T)}{T} + a_3 + \frac{a_4 T}{2} + \frac{a_5 T^2}{3} + \frac{a_6 T^3}{4} + \frac{a_7 T^4}{5} + \frac{b_1}{T}$$
$$\frac{s^0(T)}{R} = -\frac{a_1}{2T^2} - \frac{a_2}{T} + a_3 \ln|T| + a_4 T + \frac{a_5 T^2}{2} + \frac{a_6 T^3}{3} + \frac{a_7 T^4}{4} + b_2$$

This model was only used for the dissociation of nitrogen. The coefficients and their brackets may be found for N and N_2 in appendix C.

4.2 Employed Numerical Methods

4.2.1 Method of Lagrangian Multipliers

The method of Lagrangian multipliers is a general method for converting a constrained optimization problem to an unconstrained one with more degrees of freedom.

If there is a scalar potential function $f(\vec{x})$ subject to some family of constraints $g_j(\vec{x}) = 0$, then the critical points of the constrained system occur when the gradient of 'f' is parallel to the gradient of all of the constraints:

$$\nabla f(\vec{x}) \parallel \nabla g_j \; \forall j$$

If we construct a Lagrangian potential, defined:

$$\mathcal{L}(\vec{x}, \vec{\lambda}) = f(\vec{x}) - \sum_i \lambda_j \, g_j(\vec{x})$$

Then this parallel gradient condition is achieved when:

$$\frac{\partial \mathcal{L}}{\partial x_i} = 0 \; \forall \, i, \qquad and, \qquad \frac{\partial \mathcal{L}}{\partial \lambda_j} \; \forall \, j$$

In other words, the unconstrained minimum of $\mathcal{L}(\vec{x}, \vec{\lambda})$ is the constrained minimum of $\nabla f(\vec{x})$. If a system has 'A' degrees of freedom and 'B' constraints, the critical points are defined by the roots of a system of 'A+B' equations.

This method is used to apply the physical constraints (e.g. atomic element abundance) to the free energy function in the LNR CEC solver (section 5.1)

4.2.2 Newton Raphson Optimization

The unconstrained minimum of a multivariate function $f(\vec{x})$ is defined as the point at which all of the function's partial derivatives go to zero, i.e. the mutual solution to the system of equations:

$$\frac{\partial f}{\partial x_i} = 0 \; \forall \, i$$

Though there are no ways to solve the roots of a system of non-linear equations in general, the condition of that system being the gradient of a potential function allows the use of the Newton-Raphson method [14], a multivariate extension on Newton 1 dimensional root finding method.

Expressing the system of equations as a vector $\vec{h}$, a function of some arbitrary coordinate system $\vec{y} = \vec{y}(\vec{x})$, we can take a linear approximation in the correction variables about a fixed point '$\vec{y}_0$ 'for each equation:

$$h_k(\vec{y}) \approx h_k(\vec{y}_0) + \sum_i \frac{\partial h_k(\vec{y}_0)}{\partial y_i} \Delta y_i$$

The elements of $\vec{y}$ are called 'correction variables', and, provided they account for all necessary degrees of freedom, can be defined almost arbitrarily.

Collectively, the entire system may then be approximated by a linear system:

$$\vec{h}(\vec{y}) \approx \vec{h}(\vec{y}_0) + J\Delta\vec{y}$$

Where 'J' is the Jacobian matrix:

$$J = \begin{bmatrix} \frac{\partial h_1}{\partial y_1} & \frac{\partial h_1}{\partial y_2} & \cdots \\ \frac{\partial h_2}{\partial y_1} & \ddots & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

We want to take a step $\Delta\vec{y}$ such that $h(\vec{y}_0 + \Delta\vec{y}) \approx 0$, from which we may rearrange for an iteration equation:

$$\vec{y}^{n+1} = \vec{y}^n + \Delta y$$

$$\Delta\vec{y} = -J^{-1}\vec{h}(\vec{y})$$

By iterating on the value of $\vec{y}$, the roots of the system, and thus the critical point of the Lagrangian potential, are approached.

Note that this iteration requires the inversion of the matrix J. Matrix inversion is, outside of specific cases, a particular expensive operation, scaling with $n!$ for an $n \times n$ matrix. As such, the dominant determining factor for the cost of each iteration, and for the Newton Raphson method at large, is the size of the matrix J.

This rootfinding/optimization method is used in the LNR solver to locate the minimum of the constrained energy function.

4.2.3 Steepest Descent Optimization

For a scalar potential function:

$$f(\vec{x})$$

Any small change $\Delta\vec{x}$ will produce an approximate change in 'f':

$$\Delta\vec{f}(\vec{x}) \approx \nabla f(\vec{x}_0) \cdot \Delta\vec{x}$$

If the goal is to locate the minimum of the function, we can track 'downhill', i.e. such that $\Delta\vec{x}$ is anti-parallel to the gradient vector:

$$\Delta\vec{x} = -\alpha\, \nabla f(\vec{x})$$

The step scaling, $\alpha = \frac{|\Delta\vec{x}|}{|\nabla f(\vec{x})|}$, can be selected arbitrarily. If the value of the function at the minimum is known, a reasonable estimate is to scale the step such that, under a 1D linear approximation, the iteration will take the function to that minimum value.

$$\Delta f(\vec{x}) = -\alpha|\nabla f(\vec{x})|^2 = f_{Min} - f(\vec{x})^{(n)}$$

For a step scaling:

$$\Rightarrow \alpha = \frac{f(\vec{x})^{(n)} - f_{Min}}{|\nabla f(\vec{x})|^2}$$

In this report, this 'steepest descent' process is employed in sections 5.1.3 and 5.2.3 to locate 'stoichiometric' initial states for optimization, i.e. states that satisfy the system's physical constraints.

4.2.4 Nelder-Mead Optimization

One of the solvers in this report makes use of the Nelder-Mead algorithm, a generic multivariate unconstrained optimizer that uses a successive 'guess and check' method to locate and converge towards a local optimum of some objective function. This is done by constructing a 'simplex' (an n-dimensional analogue to a triangle or tetrahedron) which is expanded until an optimum is located, and then collapsed until precise. In this way, it is analogous to an unbracketed golden search method, but generalized to 'n' dimensions.

To this end, the Nelder Mead algorithm requires only that the function it is optimizing be well defined at all points in the parameter space, and so can be used as a generic 'black box' optimizer for some potential function.

5 METHODS

This section will outline the two methods that were prototyped for use as an Eilmer equilibrium calculator. Both methods were prototyped in python (python code available in appendix A), though only the non-stoichiometric LNR method was fully developed for use in Eilmer (Dlang code available in appendix B)

5.1 LNR SOLVER

The first method that will be presented is a reduced non-stoichiometric algorithm that uses Lagrangian multipliers (see section 4.2.1) to apply the system constraints, and optimizes the resulting potential using the Newton Raphson method (see section 4.2.2). This formulation is referred to in this report as the "Lagrangian Newton Raphson" (LNR) method, and is an instance of the process that is almost universally used for general CEC solvers [3]. This section will describe the specific formulation of the iteration equations used in this project, as well as the convergence/halting conditions, and two methods for generating starting points for the iterations. This is the CEC solver programmed in both the python prototype and the final D project, the code for both of which are available in appendices A and B, respectively.

5.1.1 FORMULATION

In this project, we broadly adopt the methodology of White [11], Gordon & Mcbride [2] [7] and (Zeleznik and Gordon [1]: using the method of Lagrangian Multipliers and a Newton Raphson method with logarithmic correction variables. We specifically emulate the process of Zeleznik & Gordon in their development of the NASA CEA program. This method is distinct in that it:

- Makes all physical equations unitless in the rootfinding process
- Leaves temperature as a free (unoptimized) variable at the Lagrangian potential stage; and
- Restricts temperature only by the introduction of an additional constraint (in our case, internal energy) at the rootfinding stage.

We begin with the Gibbs free energy function for a mixture of ideal gasses, as the function to me minimized:

$$G(\vec{n},T) = \sum_i n_i \left(\mu_i^0(T) + RT \ln\left|\frac{n_i RT}{P_0}\right|\right)$$

And the elemental abundance constraints as a set of equality constraints:

$$\sum_i C_{ij} n_i - b_j = 0$$

These are then assembled into a single, unconstrained, Lagrangian potential:

$$\mathcal{L}(\vec{n},T,\vec{\lambda}) = \sum_i n_i \left(\mu_i^0(T) + RT \ln\left|\frac{n_i RT}{P_0 V}\right|\right) + \sum_j \lambda_j \left(\sum_i C_{ij} n_i - b_j\right)$$

$$\mu_i = \mu_i^0(T) + RT \ln\left|\frac{n_i RT}{P_0 V}\right|$$

As per the method of Lagrangian Multipliers (see section 4.2.1) the constrained minimum of $G(\vec{n},T)$ lies at the unconstrained minimum of $\mathcal{L}(\vec{n},T,\vec{\lambda})$. Using the CEA method, we need only optimize for $\vec{n}$ and $\vec{\lambda}$ at this stage, the temperature constraints are introduced at the later 'rootfinding' stage.

$$\nabla\mathcal{L}(\vec{n},T,\vec{\lambda}) = 0$$

For I chemical species and J atomic species, this yields a system of $I + J$ equations. We now additionally introduce the internal energy constraint to restrict the system temperature:

$$(1), \qquad \frac{\partial \mathcal{L}}{\partial n_I} = 0 = \mu_j^0 + RT\left[\ln\left|\frac{n_i RT}{P_0 V}\right| + 1\right] + \sum_j \lambda_j C_{ij}\,, \qquad \forall\, i(2), \qquad \frac{\partial \mathcal{L}}{\partial \lambda_j} = 0 = \left(\sum_i C_{ij} n_i - b_j\right),$$
$$\forall j(3), \qquad 0 = \sum_i n_i u_i(T) - U_0$$

Zeleznik & Gordon make the unexplained choice of not including the $\frac{d\mu_i}{dn_i} = RT$ term in their version of this derivation, resulting in:

$$(1), \qquad \frac{\partial \mathcal{L}}{\partial n_I} = 0 = \mu_j^0 + RT\left[\ln\left|\frac{n_i RT}{P_0 V}\right|\right] + \sum_j \lambda_j C_{ij}\,, \qquad \forall\, i$$

Functionally, this produces a small deviation in the domain $|\mu_i| \approx 0$. This has not been included in the rest of the derivation, but the choice may be introduced in both the prototype and final solver by way of a simple Boolean switch (See appendices).

The mutual solution to all three of these equations represents the optimum to our system. To solve them, we first, as per the CEA method, make all physical equations unitless:

$$(1), \qquad 0 = \frac{\mu_j^0}{RT} + \left[\ln\left|\frac{n_i RT}{P_0 V}\right| + 1\right] + \sum_j \frac{\lambda_j}{RT} C_{ij}\,, \qquad \forall\, i(2), \qquad 0 = \sum_i C_{ij} n_i - b_j, \qquad \forall j(3),$$
$$0 = \sum_i n_i \frac{u_i}{RT} - \frac{U_0}{RT}$$

From here on out, we define $\pi_j = \frac{\lambda_j}{RT}$, and use these as a set of J independent variables in place of the Lagrangian multipliers. We now employ a Newton Raphson iteration scheme (see section 4.2.2) with logarithmic correction variables, so that the solver cannot iterate into a negative n_i or T:

$$\frac{\partial(1)}{\partial \ln|n_i|} = 1, \qquad \forall i \qquad \Bigg| \qquad \frac{\partial(2)}{\partial \ln|n_i|} = n_i C_{ij}, \qquad \forall j \qquad \Bigg| \qquad \frac{\partial(3)}{\partial \ln|n_i|} = n_i \frac{u_i}{RT}, \qquad \forall j$$
$$\frac{\partial(1)}{\partial \pi_j} = C_{ij}, \qquad \forall j \qquad \Bigg| \qquad \frac{\partial(2)}{\partial \pi_j} = 0, \qquad \forall j \qquad \Bigg| \qquad \frac{\partial(3)}{\partial \pi_j} = 0, \qquad \forall j$$
$$\frac{\partial(1)}{\partial \ln|T|} = 1 - \frac{h_i}{RT} \qquad \Bigg| \qquad \frac{\partial(2)}{\partial \ln|T|} = 0 \qquad \Bigg| \qquad \frac{\partial(3)}{\partial \ln|T|} = \frac{U_0}{RT} + \sum_i n_i \left(\frac{c_{v_i}}{R} - \frac{u_i}{RT}\right)$$

In $\frac{\partial(1)}{\partial \ln|T|}$, we make use of:

$$\frac{d\mu_j^0}{dT} = \frac{d(h_i - s_0 T)}{dT}$$
$$= c_p - s_0 - T\frac{1}{T}c_p$$
$$= -s_0$$

Using these partial derivatives, we may assemble three sets of Newton Raphson equations:

$$(1), \qquad \Delta \ln|n_i| + \sum_j \Delta\pi_j C_{ij} + \left(1 - \frac{\mathrm{h_i}}{\mathrm{RT}}\right)\Delta \ln|T| = -\left[\frac{\mu_j^0}{RT} + \left[\ln\left|\frac{n_i RT}{P_0 V}\right| + 1\right] + \sum_j \pi_j C_{ij}\right]$$

$$(2), \qquad \sum_i n_i C_{ij}\,\Delta \ln|n_i| = -\left[\sum_i C_{ij} n_i - b_j\right]$$

$$(3), \qquad \sum_i n_i \frac{u_i}{RT}\Delta \ln|n_i| + \left(\frac{U_0}{RT} + \sum_i n_i\left(\frac{c_{v_i}}{R} - \frac{u_i}{RT}\right)\right)\Delta \ln|T| = -\left[\sum_i n_i \frac{u_i}{RT} - \frac{U_0}{RT}\right]$$

In set (3), the linearity of the π_j corrections means that we can simplify further by noting that, while iterating between steps (k) and (k+1):

$$\sum_{\mathrm{j}} \mathrm{c_{ij}}\Delta\pi_{\mathrm{j}} = \sum_{\mathrm{j}} \mathrm{c_{ij}}\left[\pi_{\mathrm{j}}^{(\mathrm{k+1})} - \pi_{\mathrm{j}}^{(\mathrm{k})}\right] \sum_{\mathrm{j}} \mathrm{c_{ij}}\pi_{\mathrm{j}} = \sum_{\mathrm{j}} \mathrm{c_{ij}}\pi_{\mathrm{j}}^{(\mathrm{k})}$$

Thus, instead of calculating a correction to π_j, it is more convenient to calculate the next value step directly. Notice that the structure of equation (1) allows us to easily rearrange for $\Delta \ln|n_i|$:

$$\Delta \ln|n_i| = -\left[\frac{\mu_j^0}{RT} + \left[\ln\left|\frac{n_i RT}{P_0 V}\right| + 1\right]\right] - \sum_j \mathrm{C_{ij}}\pi_j - \left(1 - \frac{h_i}{RT}\right)\Delta \ln|T|$$

Which may be substituted into the remaining equations and rearranged to arrive at:

$$\left[\sum_i n_i C_{ij}\left(1 - \frac{h_i}{RT}\right)\right]\Delta \ln|T| + \sum_k \left[\sum_i n_i C_{ij}\mathrm{C_{ik}}\right]\pi_k = -\sum_i n_i C_{ij}\left(\frac{\mu_i^0}{RT} + \ln\left|\frac{n_i RT}{P_0 V}\right|\right) - b_j,$$

$$\forall j \left[\frac{U_0}{RT} + \sum_i n_i\left(\frac{c_{v_i}}{R} - \frac{u_i}{RT}\frac{h_i}{RT}\right)\right]\Delta \ln|T| - \sum_k \left[\sum_i n_i \frac{u_i}{RT}\mathrm{C_{ik}}\right]\pi_k$$

$$= \sum_i n_i \frac{u_i}{RT}\left[\frac{\mu_j^0}{RT} + \ln\left|\frac{n_i RT}{P_0 V}\right|\right] + \frac{U_0}{RT}$$

We have then arrive at a linear system describing the iteration process.

$$J\Delta\vec{y} = h, \qquad \vec{y} = \begin{pmatrix} \pi_1 \\ \vdots \\ \pi_J \\ \ln|T| \end{pmatrix}$$

$$\Delta\vec{y} = J^{-1}h$$

Where we have defined the matrix and vector:

$$J = \begin{bmatrix} \sum_i n_i C_{i1}\left(1 - \frac{h_i}{RT}\right) & \sum_i n_i C_{i1}\mathrm{C_{i1}} & \sum_i n_i C_{i1}\mathrm{C_{i2}} & \cdots \\ \sum_i n_i C_{i2}\left(1 - \frac{h_i}{RT}\right) & \sum_i n_i C_{i2}\mathrm{C_{i1}} & \sum_i n_i C_{i2}\mathrm{C_{i2}} & \cdots \\ \vdots & \vdots & \vdots & \ddots \\ \frac{U_0}{RT} + \sum_i n_i\left(\frac{c_{v_i}}{R} - \frac{u_i}{RT}\left(2 - \frac{h_i}{RT}\right)\right) & -\sum_i \frac{n_i u_i}{RT}\mathrm{C_{i1}} & -\sum_i \frac{n_i u_i}{RT}\mathrm{C_{i2}} & \cdots \end{bmatrix}, \qquad h = \begin{pmatrix} -\sum_i C_{i1}\frac{n_i\mu_i}{RT} - b_1 \\ -\sum_i C_{i2}\frac{n_i\mu_i}{RT} - b_2 \\ \vdots \\ \sum_i \frac{u_i}{RT}\frac{n_i\mu_i}{RT} + \frac{U_0}{RT} \end{pmatrix}$$

This substitution stage, which Gordon & McBride refer to as 'reduction' [1], reduces the size of the Jacobian matrix to $(J+1)\times(J+1)$. As the cost of each iteration in this method is dominated by the calculation of J^{-1}, this means that the cost of the LNR method scales primarily with the number of atomic species, with little effect at all from the number of chemical species.

We can then find the corrections to $\ln|n_i|$ by:

$$\overrightarrow{\Delta \ln|n_i|} = -f - K\Delta\vec{y}, \qquad f = \begin{pmatrix} \frac{\mu_1}{RT}+1 \\ \frac{\mu_2}{RT}+1 \\ \vdots \end{pmatrix}, \qquad K = \begin{bmatrix} C_{11} & C_{12} & \dots & 1-\frac{h_1}{RT} \\ C_{21} & C_{22} & \dots & 1-\frac{h_2}{RT} \\ \vdots & \vdots & \ddots & \ddots \end{bmatrix}$$

For the sake of convenience in programming, we also express these in summation form:

$$J = J_0 + \sum_i J_i\,, \qquad h = h_0 + \sum_i F_i$$

$$J_0 = \begin{bmatrix} 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \ddots \\ \frac{U_0}{RT} & 0 & 0 & \dots \end{bmatrix} \qquad J_i = \begin{bmatrix} n_i C_{i1}\left(1-\frac{h_i}{RT}\right) & n_i C_{i1}\mathrm{C_{i1}} & n_i C_{i1}\mathrm{C_{i2}} & \dots \\ \sum_i n_i C_{i2}\left(1-\frac{h_i}{RT}\right) & n_i C_{i2}\mathrm{C_{i1}} & n_i C_{i2}\mathrm{C_{i2}} & \dots \\ \vdots & \vdots & \vdots & \ddots \\ n_i\left(\frac{c_{v_i}}{R}-\frac{u_i}{RT}\left(2-\frac{h_i}{RT}\right)\right) & -n_i\frac{u_i}{RT}\mathrm{C_{i1}} & -n_i\frac{u_i}{RT}\mathrm{C_{i2}} & \dots \end{bmatrix}$$

$$h_0 = \begin{pmatrix} -b_1 \\ -b_2 \\ \vdots \\ \frac{U_0}{RT} \end{pmatrix} \qquad h_i = \begin{pmatrix} -n_i C_{i1}\frac{\mu_i}{RT} \\ -n_i C_{i2}\frac{\mu_i}{RT} \\ \vdots \\ n_i\frac{u_i}{RT}\frac{\mu_i}{RT} \end{pmatrix}$$

We may recover the Zeleznik & Gordon formulation by replacing any '$\frac{\mu_i}{RT}$' term with '$\frac{\mu_i}{RT}-1$'.

This provides the 'direction' of the iteration, but a step-scaling factor 'α' may also be applied to improve stability and avoid iterative oscillation:

$$\begin{aligned} \vec{y}^{k+1} &= \vec{y}^{k} + \alpha\,\Delta\vec{y} \\ \overrightarrow{\ln|n|}^{k+1} &= \overrightarrow{\ln|n|}^{k} + \alpha\Delta\overrightarrow{\ln|n|} \\ \alpha &\leq 1 \end{aligned}$$

The choice of scaling factor is discussed in the section 5.1.2.

5.1.2 Convergence Test & Halting Condition

To examine convergence, we make an estimate of the solver's current deviation from the solution by the norm of the iteration's update vector:

$$E^2 = \sum_j \Delta\pi_j^2 + \sum_i \Delta \ln|n_i|^2 + \Delta \ln|T|^2$$

An optimization attempt is considered to have converged if both:

- The error is below some user-defined tolerance threshold; and
- The error has been decreasing for more than 'n' iterations.

Where n=10 was found to be a suitable value in most cases. The first condition is a precision tolerance, while the second avoids mistakenly ending the iterations at a near-stable point in the iteration scheme. For more details as to why this is necessary, see section 6.1.2.

If the simulation has failed to meet these convergence conditions after some cut-off number of iterations (~100 iterations was found to be suitable) it is considered to be non-convergent. If this is the case, repeated attempts are made by the following procedure:

1. Half the step scale 'α' and attempt to solve again from the same initial state
2. Half α once more and try again
3. If still non-convergent, try resetting the initial conditions and trying again
4. Repeat steps 1 and 2
5. If no solution has been reached, return an error

In systems where there is a physical solution to be found, the initial state was rarely found to have an effect on convergence, and step sizes of $\alpha \leq 0.5$ were found to be sufficiently stable in most cases. The remaining steps exist either to improve efficiency or robustness of the solver.

5.1.3 Starting Conditions

In some cases, an initial estimate of the optimized solution will be available; either from the result of a simplified model or from the solution of a previous solver with similar input conditions. However, such an estimate is not always readily at hand, and a reliable means of generating an initial state for iterations to begin at is required. In this project, two methods for finding this initial state were investigated.

Method 1: Averaged Atom

Gordon & McBride [1] indicate that a suitable initial state is:

$$T = 298\ K$$
$$n_i = 0.1\ mol\ \forall\ i$$

Where the chemical species abundance is decreased by a factor of 10 if the simulation fails to converge.

We employ a similar method, but with the initial guess for the chemical abundance scaling with the number of atoms:

$$n_i = \frac{\sum_j b_j}{I} = \frac{no.\,atoms}{no.\,chemical\ species}$$

$$\pi_j = 0\ \forall j$$

$$T = 298\ K$$

We refer to this as the 'averaged atom' method, and have used it as the default method in the solvers when a more specific initial state is not available. The reasons for this choice are discussed in section 6.1.2.

Method 2: Stoichiometric Initial Conditions

Many sources on CEC solvers indicate that a good starting position for a solver is one that obeys the systems physical constraints, though it may not necessarily be similar to the optimized solution. To calculate such a state, a steepest descent rootfinding method is here used.

We first define a convex potential that can only reach a minimum when the system constraints are satisfied. In keeping with the logic of Gordon & McBride, we make the energy terms of this potential unitless:

$$\begin{aligned} P &= \sum_j \left(b - b_j^0\right)^2 + \left(\frac{U}{RT} - \frac{U_0}{RT}\right)^2 \\ &= \sum_j \left(\sum_i c_{ij} n_i - b_j^0\right)^2 + \left(\frac{\sum_i u_i n_i}{RT} - \frac{U_0}{RT}\right)^2 \end{aligned}$$

We can finding the minimum of this potential by a steepest descent method with logarithm correction variables to stay within the positive T, n_i:

$$\frac{dP}{dln|n_i|} = 2n_i\left[\sum_j \Delta b_j c_{ij} + \frac{\Delta U}{RT}\cdot\frac{u_i}{RT}\right], \qquad \frac{dP}{d\ln|T|} = 2\frac{\Delta U}{RT}\left[\frac{\sum_i c_{v,i} n_i}{R} - \frac{\Delta U}{RT}\right]$$

And full gradient:

$$\nabla P = 2\begin{pmatrix} n_i\left[\sum_j \Delta b_j c_{ij} + \frac{\Delta U}{RT}\cdot\frac{u_i}{RT}\right] \\ \frac{\Delta U}{RT}\left[\frac{C_v}{R} - \frac{\Delta U}{RT}\right] \end{pmatrix}$$

By iterating the $\vec{x} = \begin{pmatrix}\ln|n| \\ \ln|T|\end{pmatrix}$ vector, starting with a state defined by method 1 (above), we can track towards a physically constrained system state via the steepest descent method, knowing that the minimum possible value of 'P' is zero:

$$\Delta\vec{x} = -\frac{P}{|\nabla P|}\frac{\nabla P}{|\nabla P|}$$

5.2 STOICHIOMETRIC SOLVER

Though the reduced non-stoichiometric LNR method is generally the most efficient means of locating solving for equilibrium conditions, a simple non-stoichiometric algorithm was also investigated to compare against, formulated to make use of Eilmer's existing Nelder Mead optimizer module. This algorithm was only prototyped in python, but was not found to be reliable enough to warrant further development, for reasons that are discussed further in section 7.1.

This algorithm was developed to make use of Eilmer's Nelder Mead module as a 'black box', simply feeding in an initial state and an objective function and allowing it to optimize. As the solver was never integrated into Eilmer, an existing Nelder Mead algorithm was used in its place in the python prototype (see appendix A)

5.2.1 FORMULATION

In general, the system state can be described by its composition and temperature (see section 4.1), which, for 'I' chemical species, gives I+1 degrees of freedom. However, this state must adhere to a stated internal energy and 'J' element counts, meaning that the solution space only has 'I-J' degrees of freedom.

The element abundance constraints may summarized as a linear system:

$$C\vec{n} = \vec{b}$$

If there are 'J' elements and 'I' species, the system has 'I-J' degrees of freedom, and we may define 'I-J' species 'primary' species and use these to calculate the remaining 'J' 'secondary' species. This appears as the secondary species vector '$\vec{n}_{II}$' having a square matrix 'A' in the element balance:

$$A\overrightarrow{n_{II}} + D\overrightarrow{n_I} = \vec{b}$$
$$\overrightarrow{n_{II}} = A^{-1}\left[\vec{b} - D\overrightarrow{n_I}\right] = \overrightarrow{n_{II}}(\overrightarrow{n_I})$$

Here, 'A' and 'D' are subset 'slices' of the conservation matrix 'C'. In this way, the system composition state is fully defined by a set vector of primary species counts:

$$\vec{n} = \begin{pmatrix} \overrightarrow{n_I} \\ n_{II} \end{pmatrix} = \vec{n}(\overrightarrow{n_I})$$

We choose to divide primary and secondary species to give largest value of $\left|\frac{\det(A)}{norm(A)}\right|$, so as to provide a well conditioned system for the calculation of A^{-1}.

Additionally, the internal energy balance may be used to calculate the corresponding temperature for a set composition. As internal energy is a strictly increasing function of temperature, there is at most one temperature that satisfies the internal energy balance for a set composition:

$$U(\vec{n}, T) - U_0 = 0$$

Finding this temperature, $T(U_0, \vec{n}_I)$, is a simple matter of 1D root finding, for which many well established algorithms exist [14]. Together, these allow the systems entire state to be well defined by the abundances of the primary species only:

$$Sys\ State(\vec{n}, T) = Sys\ State\ (\vec{n}_I)$$

In this form, the problem simplifies to an instance of unconstrained optimization over a fixed domain, suitable for use with the Nelder Mead algorithm, a general purpose non-linear optimizer algorithm that requires only the ability to evaluate a function. This algorithm is already supplied by Eilmer, and so is not discussed in detail here.

To prevent the solver from venturing into the non-physical domain (the boundaries of which are discussed in detail in section 5.2.2), a 'penalty' (set as 10^{100}J) is applied if a negative species concentration or temperatures is found when evaluating the system state. The Nelder Mead algorithm is then tasked with optimizing this 'windowed' free energy function:

$$G_{window}(\vec{n_I}) = \begin{cases} Penalty & T(\vec{n}_I) < 0 \; or \; n_i < 0 \;\; in \; \vec{n}(\vec{n}_I) \\ G(\vec{n}_I) \end{cases}$$

For a more detailed description of this 'window' region, see section 5.2.2.

5.2.2 Valid Solution Space

Because of the way the constraints were applied, any given point $\vec{n}_I$ will satisfy all of the system constraint equations, but this does not guarantee that the resulting state will be a physically feasible. For example, suppose the system's energy is lower than the chemical models allows for at absolute zero, i.e.:

$$U_0 < \sum_i n_i u_i(T = 0)$$

Attempting to calculate the temperature in this instance will yield $T < 0$, which is non-physical and will lead optimization attempts astray. Similarly, separation of species doesn't necessarily prevent negative species concentrations. For example, consider some dissociating diatomic gas:

$$\begin{pmatrix} A \\ A_2 \end{pmatrix} [1 \quad 2] = (b_1), \qquad b_1 = 1$$

This elemental balance is technically satisfied by a nonphysical solution:

$$\begin{pmatrix} A \\ A_2 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

So, the physical solution space is bounded by the space between a system of 'I+1' linear equations in the 'I-J' primary species counts, defined by:

$$U_0 - \sum_i n_i u_i(T = 0) = 0, \qquad n_i = 0$$

This will always produce a contiguous solution space to search for the optimum within.

As a more complicated example, consider a 4-species, 2 element system:

$$\begin{bmatrix} 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix} \begin{pmatrix} A \\ A_2 \\ AB \\ B_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

This may be separated as:

$$\begin{pmatrix} A \\ B_2 \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \left[\begin{pmatrix} b_1 \\ b_2 \end{pmatrix} - \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} A_2 \\ AB \end{pmatrix} \right]$$

Using a simplified chemical model as shown below, and setting 1.5 mol of element 'A', 1.0 mol of element 'B' and a system internal energy of $U_0 = -8\ kJ$ in a volume of 0.1 mm^{3,} we can see the linear bounding surfaces for the solution space (figures 5.2.2.1 and 5.2.2.2). Also plotted are the contours of Free Energy for both systems.

$$u_i(T) = u_{i,0} + (T - 298)\frac{Rq}{2}$$
$$s_{T,i}(T) = s_{i,0} + \frac{Rq}{2} ln \left| \frac{T}{298} \right|$$

Species	u_0	q	s_0
A	50 kJ/mol	3	100 J/K/mol
A_2	0 J/mol	5	100 J/K/mol
AB	-40 kJ/mol	6	100 J/K/mol
B_2	0 J/mol	5	80 J/K/mol

Table 5.2.2.1-Example System Chemical Properties

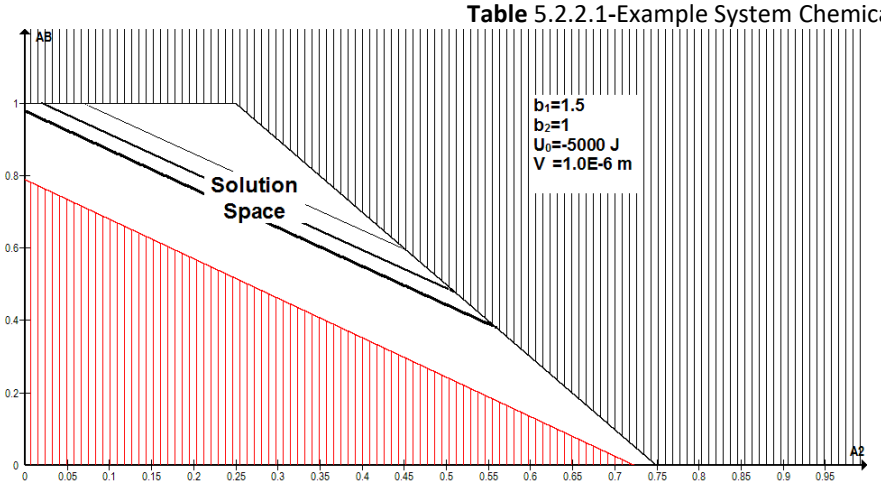

Figure 5.2.2.1- Stoichiometric Solution Space of Low Energy System

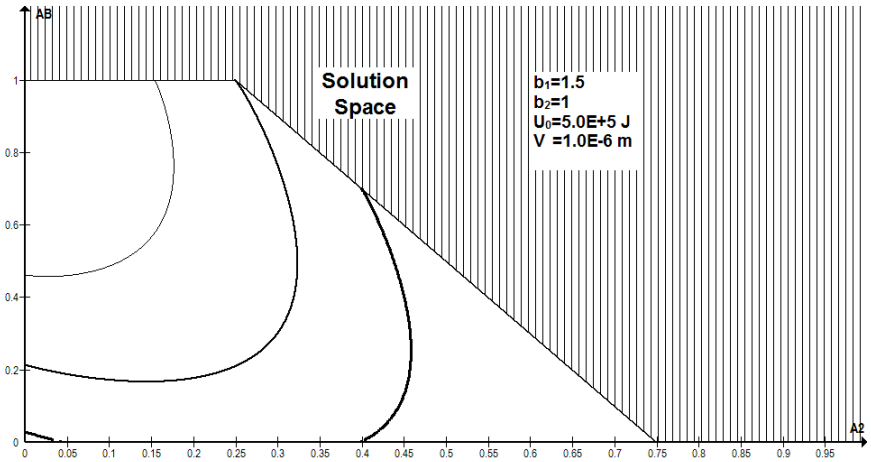

Figure 5.2.2.2- Stoichiometric Solution Space of High Energy System

5.2.3 Algorithm For Determining Starting Condition

The stoichiometric solvers require some initial state that lies within the physically valid solution space for iterations to progress from. This point may be found by any number of methods, including the minimization method outlined in section 5.1.3 or by randomly selecting points until a valid state is found. An additional method, making use of the linear boundaries on the physical solution space (discussed in section 5.2.2) is also presented here.

The bounding surfaces of the physical solution space are all linear, for 'I+1' bounding 'planar' surfaces: one for each species and one for the internal energy. As linear surfaces, each may be described by their normal vector '$\vec{\eta}_k$' and some offset constant 'c_k'.

$$f_k(\overrightarrow{n_i}) = \overrightarrow{n_i} \cdot \vec{\eta}_k + c_i = 0$$

For a particular state, $\overrightarrow{n_I}$, to be physical, it must lie inside of this surface:

$$\overrightarrow{n_i} \cdot \vec{\eta}_k + c_k \geq 0$$

For some starting non-physical guess, we can attempt to iterate towards the physical solution space based on which of these conditions are not satisfied. For each condition that is not satisfied, we identify the correction (in the direction of that surface's normal) that would be required until it is:

$$\left(\vec{n}_I + \Delta\vec{n}_{I,k}\right) \cdot \vec{\eta}_k = -c_k, \qquad \Delta\vec{n}_{I,k} = A_k\vec{\eta}_k$$

$$\Rightarrow A_k = -\frac{(c_k + \vec{\eta}_k \cdot \vec{n}_I)}{|\vec{\eta}_k|^2}$$

And repeat for all conditions, summing together to get the direction of the final step:

$$\Delta\vec{n}_I = \sum_i A_i \Delta\vec{n}_{I_k}$$

The estimate is then iterated in this direction, with some step scaling factor α. For iteration index 'L':

$$\vec{n}_I^{(L+1)} = \vec{n}_I^L + \alpha^L \Delta\vec{n}_I^L$$

The magnitude of the step is defined by a more complicated process. Firstly, it is determined if the iterations have been travelling in the same direction for more than 3 steps, where two iterations are considered to be travelling in the same direction if they meet the condition:

$$\frac{\Delta\vec{n}_I^{(L+1)} \cdot \Delta\vec{n}_I^L}{\left|\Delta\vec{n}_I^{(L+1)}\right||\Delta\vec{n}_I^L|} \leq k$$

Where k is some constant, set at $k = 0.1$. If this condition is met, it indicates that the iterations are tracking directly towards the physical solution space, and so the step size may be estimated by a pseudo Newton-Raphson scheme to minimize the function:

$$B^L = \sqrt{\sum_k f_k^2 \; \forall \; f_k < 0}$$

In which the gradient is estimated by a finite difference method based on the two most recent iterations:

$$\alpha^L = -1.1 \frac{B^L}{\left(\frac{B^L - B^{L-1}}{\alpha^L |\Delta \vec{n}_I^L|}\right)}$$

The additional factor of 1.1 is to ensure the iteration 'overshoots' to the interior of the physical solution space rather than converging exponentially close to its outside edge.

If the iterations have failed to travel in a consistent direction for more than 3 iterations, the step size is set as:

$$\alpha^L = 0.5^m$$

Where 'm' is the number of iterations since a 'straight' iteration was observed. This helps stabilize the oscillations onto a path of fixed direction, where the 'driving force' from each constraint is at a lateral balance. The step size for the first iteration is somewhat arbitrary, and $\alpha = 1$ is used here.

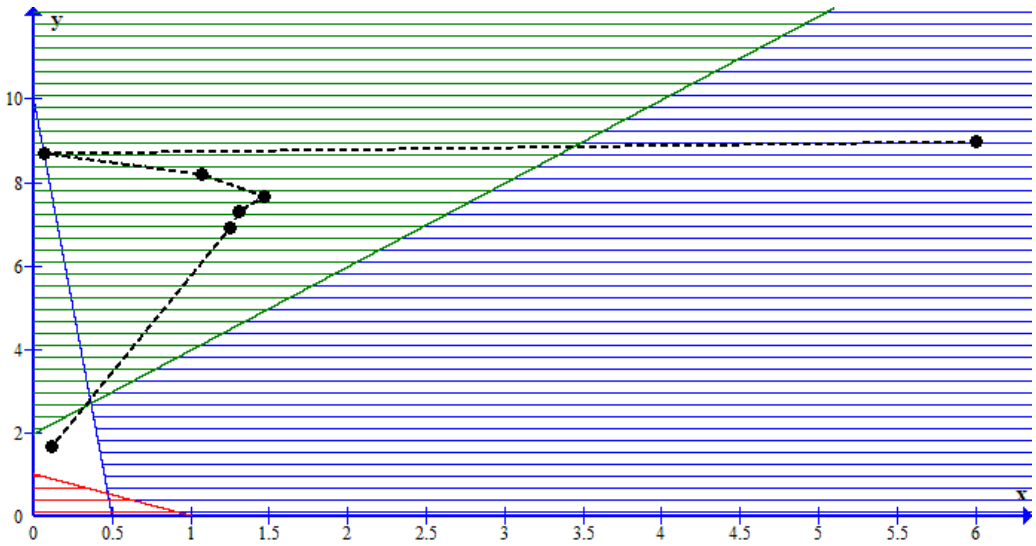

Figure 5.2.3.1-Iterations Locating the Interior of Physical Solution Space

6 RESULTS

For the two presented solver methods, there are three keys areas of interest when considering their utility:

- The accuracy of their results
- Their reliability of convergence in varying conditions
- The speed of convergence

Each of these will be examined for both the LNR and stoichiometric solvers. To validate the solvers, we choose to compare their results directly to that of CEA for the dissociation of air-pressure nitrogen over the temperature range $T \in [1 \cdot 10^3\ K, 7 \cdot 10^3 K]$. To ensure consistency between the results, the Glenn coefficients model has been use (see section 4.1.1), the specifics of which are available in appendix C. Though CEA has the facility to calculate equilibrium for prescribed pressure and temperature, the solvers in this report do not. As such, the results presented for this validation are found by numerically adjusting the solver input conditions until the desired temperature and pressure are achieved.

The robustness of the solvers is measured heuristically, simply testing a wide variety of systems to see if the solver converges, diverges or is forced to halt due to a computational error, and the speed of convergence is examined for a simple system to demonstrate the characteristic behaviour of each solver.

6.1 LNR ALGORITHM

6.1.1 ACCURACY

To validate the LNR results, we choose to simulate to a tolerance of 1 in 10^{-15} on correction variables and a tolerance of 1 in 10^{-10} for the pressure. As can be seen in figure 6.1.1.1, the data matches almost perfectly with that of CEA when the formulation of Gordon and McBride is adopted:

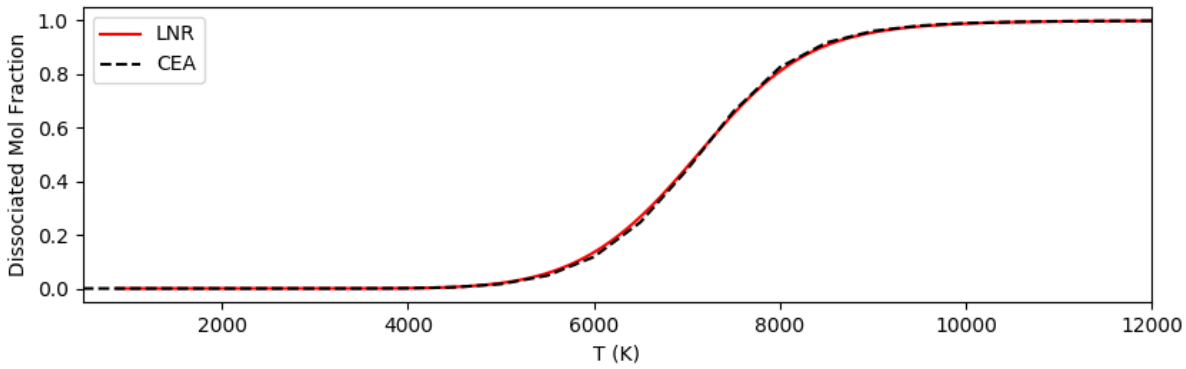

Figure 6.1.1.1-Validation of Results for LNR Method

In section 5.1.1, it is noted that Gordon and McBride make an unexplained adjustment to the calculation of the lagrangian minimum conditions, equivalent to making the transformation '$\frac{\mu}{RT} \rightarrow \frac{\mu}{RT} - 1$'. To more closely emulate their method, we have chosen to also make this adjustment in the validation. When this is not done, there is a small discrepancy between the results, with dissociation occurring ~100 K earlier (see figure 6.1.1.2).

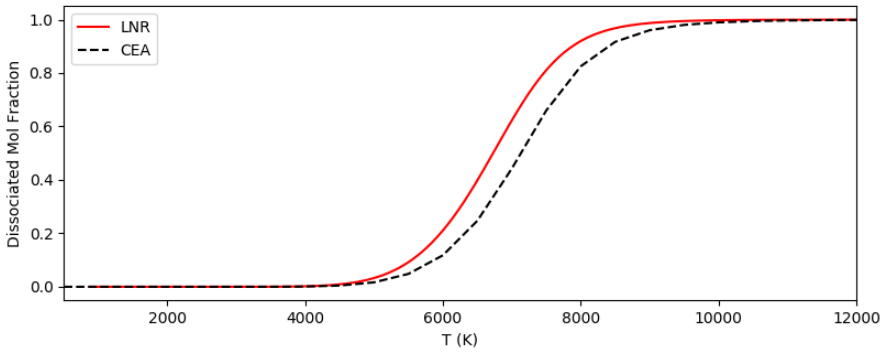

Figure 6.1.1.2-Discrepenacy Induced by Potential Adjustment

Interestingly, this discrepancy is similar in size and 'shape' to the one that is observed when the thermodynamic curves for the species are restricted to the low temperature case of constant specific heat:

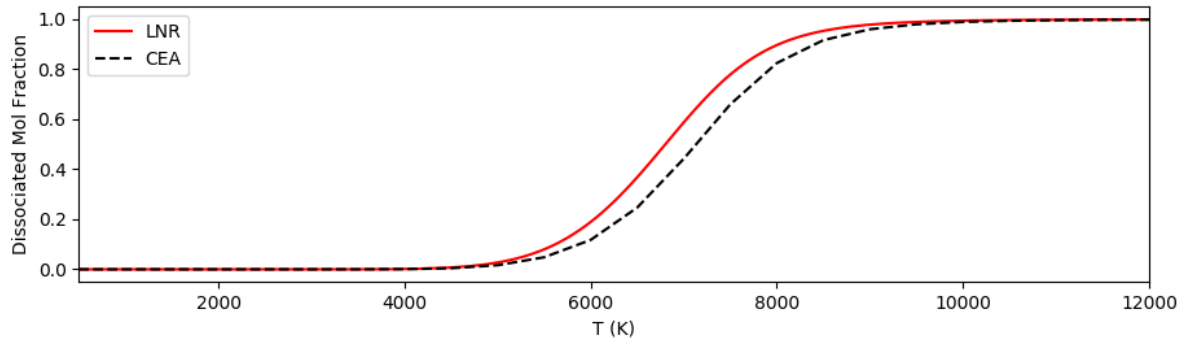

Figure 6.1.1.3-Discrepancy Introduced by Simple Thermochemical Model

It is of practical interest that simple thermodynamic models to not induce a catastrophic error, as this indicates that these low temperature models may act as a 'cheap' initial guess in systems with exceptionally complex or costly models. For this specific validation model, no significant deviation was observed between results based on the Glenn and Shomate models.

6.1.2 Convergence

As stated in section 5.1.2, we take an estimate of the error in the solver by the norm of the step size prior to applying a step-scaling factor α:

$$E^2 = \sum_j \Delta\pi_j^2 + \sum_i \Delta \ln|n_i|^2 + \Delta \ln|T|^2$$

When examining this for a typical system, we can see that, for a convergent system, the error follows a pattern of 'startup', in which the error does not progress downwards, and 'final approach', in which it decreases exponentially as the final result is refined. We can see these two branches for an example system in figure 6.1.2.1.

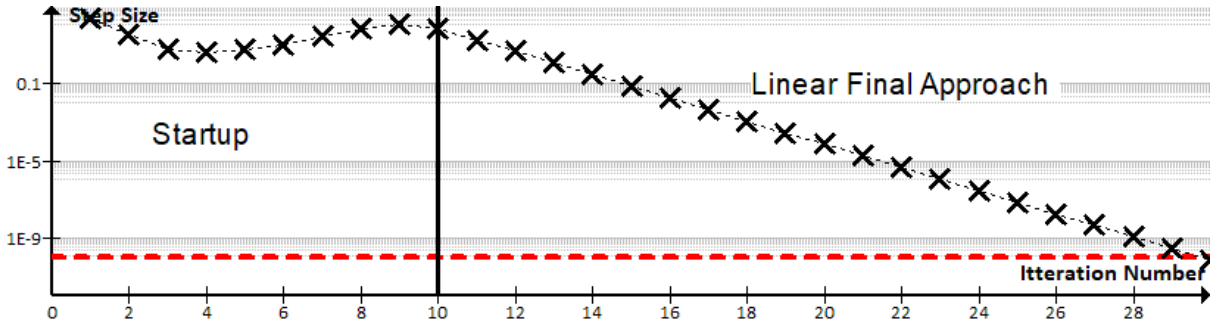

Figure 6.1.2.1-Typical Convergence Behaviour for LNR Solver

During final approach, the error decreases exponentially as iterations progress. As such, barring limits on machine precision and accuracy of the thermodynamic data, an arbitrarily tight tolerance may be achieved in linear time for any system.

This behaviour is in keeping with the convergence curves observed by Zeleznik & Gordon and Gordon & McBride in their development of similar CEC solvers [1] [2]. Though their definition of error and the conditions being solved for is slightly different, the same general shape may be observed.

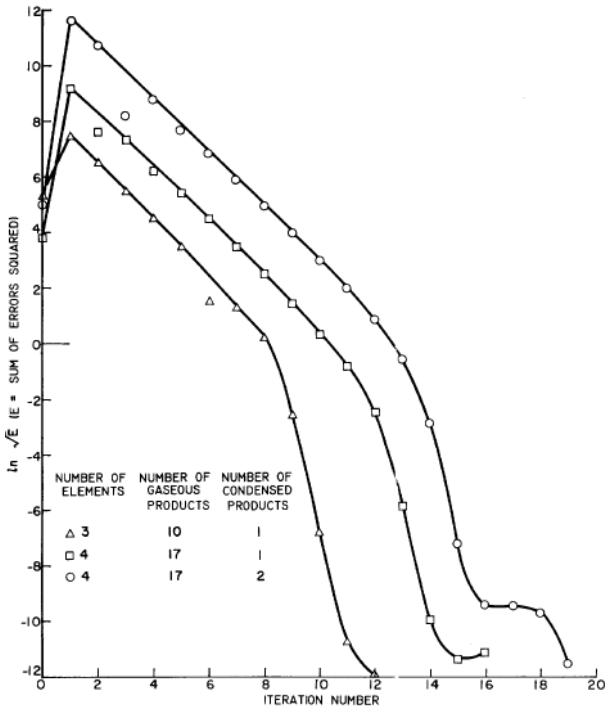

Figure 6.1.2.2-CEA Convergence Curves

6.1.2.1 Reliability of Convergence

By testing many systems at varying conditions, it was found that the LNR solver could be counted to reliably converge in all systems that did not involve excessive temperatures or densities. The limiting factor for convergence, provided a solution existed at all, is the occurring of overflows in the exponential and log natural functions used in the iteration process. In a physical sense, these conditions correspond to excessively high temperatures (e.g. $10^7\ K$) or densities (e.g. $1\frac{g}{mm^3}$). In cases where this overflow condition was borderline, the loss of precision in the iterations prevents the solver from converging to within acceptable tolerances, but does not produce wild divergence.

This reliability is, however, contingent on a sufficiently small step size. Setting $\alpha \approx 1$ can, if beginning too far from the solution, cause the solver to 'overshoot' into a non-convergent region of the parameter space, or to oscillate about the solution when nearby. The second of these is of particular interest, as it commonly occurs when using a 'nearby' solution as a starting condition, such as when generating data for a range of internal energies.

As the programmed algorithm was set to automatically scale back the step size as a first resort when encountering non-convergence, this does not affect the reliability of the LNR solver as a whole. Forcing the solver to recognise non-convergence and restart with a lower step size does, however, produce a significant impact on overall speed.

6.1.2.2 Effect of Step Size

Decreasing the step scaling improves the stability of solver, particularly in avoiding oscillatory behaviour about the optimum. This oscillation can contribute to the 'startup' iterations, and so a small step size can advance the onset of the final approach stage. However, these benefits come at the cost of decreasing the 'speed' of the final approach.

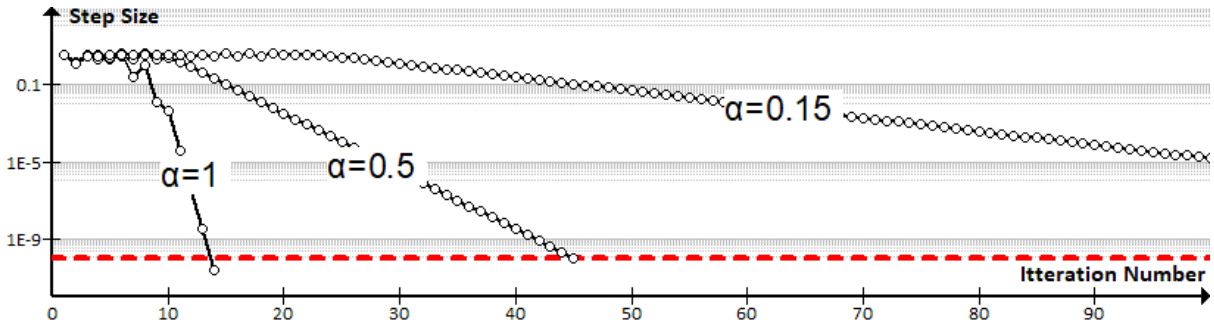

Figure 6.1.2.2.1-Effect of Stepsize on Convergence

Starting with a stepsize of $\alpha = 0.9$, and only decreasing after encountering non-convergence, was found to strike a good balance between these positives and negatives, though only when starting with no 'nearby' initial estimate. When a near-physical solution is available, it is more reasonable to begin at $\alpha \approx 0.5$ to prevent small-scale oscillation, as a smaller number of iterations are required overall.

6.1.2.3 Effect of Initial State

Many sources in the literature state that there is a benefit to using a stoichiometric state as an initial seed for iterations [3] [11]. However, this was not seen in the actual performance of the algorithm: these physically constrained starting conditions were found to perform worse in all instances when compared to the crude 'averaged atom' starting state.

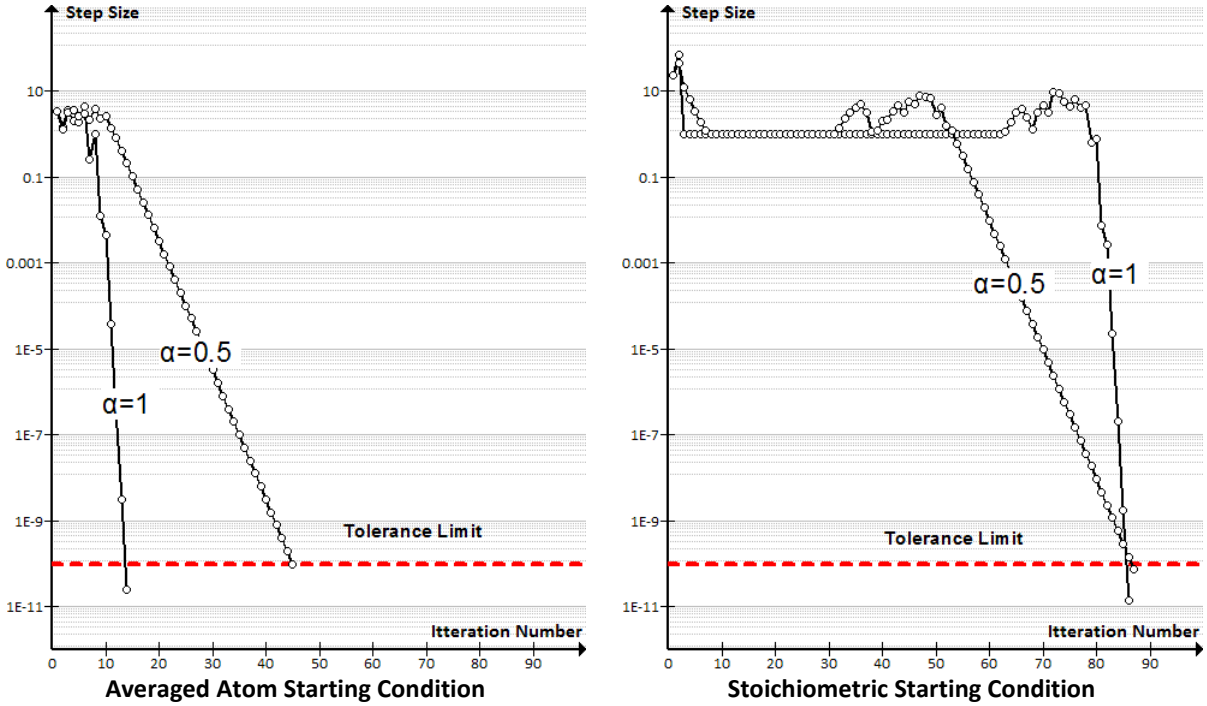

Figure 6.1.2.3.1-Convergence Curves for LNR Initial States

Though the final approach is similar for both starting conditions, non-nearby stoichiometric starting conditions lead to a much longer startup time in the solver, with no apparent benefits to balance this out.

6.2 Stoichiometric Algorithm

6.2.1 Validity

Unlike the LNR solver, the stoichiometric solver, at least when optimized via the Nelder-Mead method, performed extremely poorly when validated against CEA results. As shown in figure 6.2.1.1, the solver has a tendency to adhere to the boundaries of the solution space, failing to predict any dissociation at all. This failure to converge is discussed further in section 6.2.2.

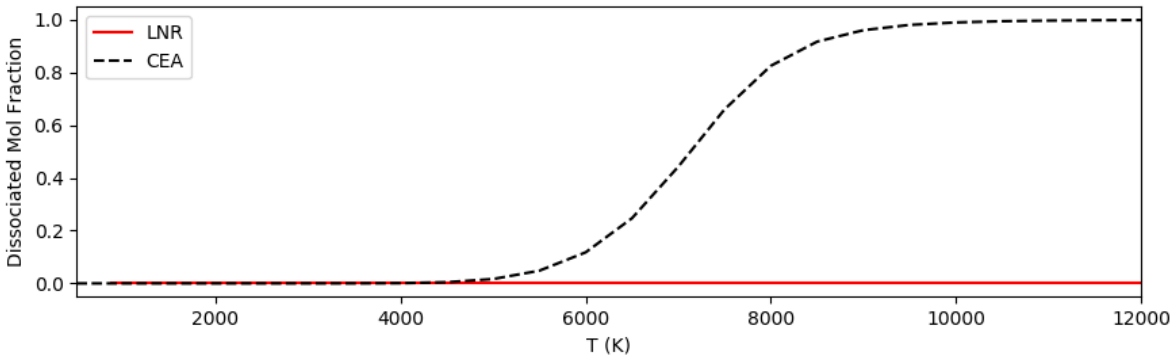

Figure 6.2.1.1- Validation of Results for Stoichiometric Method

6.2.2 Convergence

Testing the stoichiometric Nelder-Mead algorithm on many systems showed that it did correctly reproduce the results of the reliable LNR algorithm in some cases, but could not be relied on to do so in general. Take, for example, a system of the species:

$$H_2O, O_2, O_3 \; and \; H_2$$

At a high prescribed internal energy. Selecting ozone and water as the primary species, both solvers converge to the same point somewhere in the middle of the physical solution space:

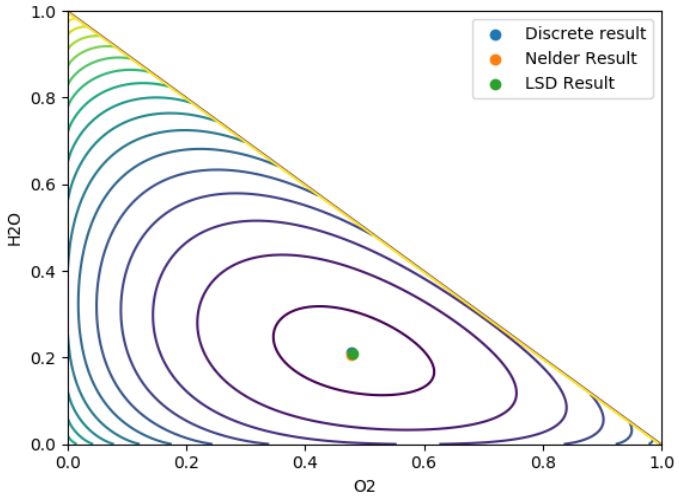

Figure 6.2.2.1-Free Energy Contours for Successfully Converging Stoichiometric System

However, in a system where the solution lies near the boundary of the physical solution space, the stoichiometric algorithm will almost always fail to converge accurately. Consider the reaction of:

$$O_2, O_3 \; and \; O$$

At a sufficiently high energy for ozone to form. In this system, the solution lies on the boundary of the solution space, which causes the Nelder Mead algorithm to interact poorly with the discontinuous windowing function applied to avoid nonphysical solutions:

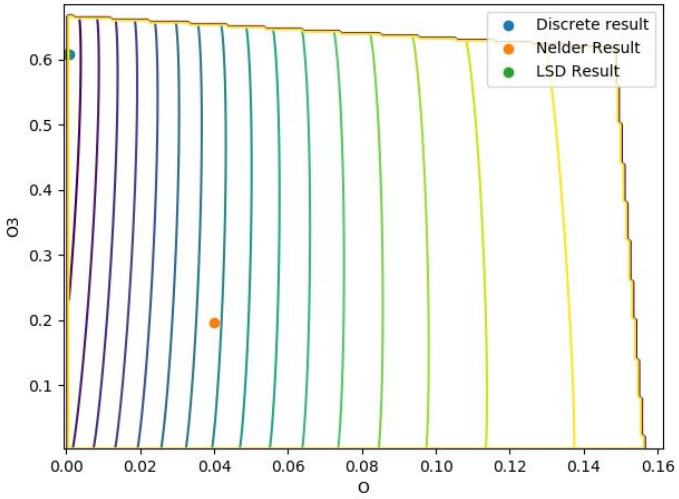

Figure 6.2.2.1-Free Energy Contours for Unsuccessfully Converging Stoichiometric System

Even in instances where the solution was not near-boundary, the stoichiometric algorithm could still not be relied upon to converge accurately, as can be seen in the nitrogen dissociation validation in section 6.2.1. This issue with near-boundary solutions is not a trivial one: most complex systems should be expected to contain one or more species that are at or near zero concentration (see figure 6.2.2.3 for an example), making this algorithm wholly unsuited to this project.

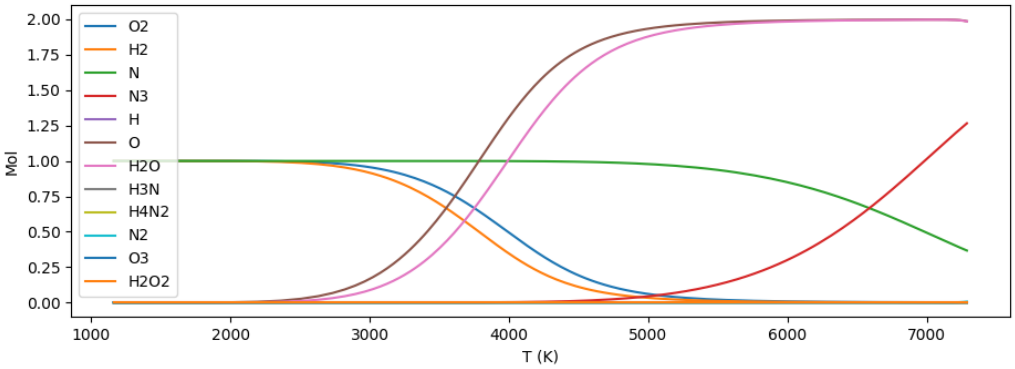

Figure 6.2.2.3-Equilibrium State of Many-Species System

It is likely that these issues are a direct result of the crude implementation of the Nelder Mead algorithm, which performs best in a completely open parameter space. A more complex approach, specifically formulated to deal with near-boundary solutions, may be more reliable, but is beyond the scope of this report.

7 Discussion

7.1 Comparison of Methods

Towards this project's underlying goal of establishing a chemical equilibrium solver for use with Eilmer, the LNR method outperforms the stoichiometric solver in virtually every front. Summarizing the results of section six, it was found that the LNR solver is, reliably convergent except in systems of physical conditions beyond the scope of Eilmer's application, rapidly converging, and is consistent in its results as compared to those of the existing CEA program. Additionally, the LNR method's cost scaled with the number of atomic species only, allowing a large set of reacting chemicals to be simulated for little additional cost.

Conversely, the stoichiometric approach, at least when paired with a 'black box' Nelder Mead module, is neither reliable nor accurate, tending to diverge almost completely in many practical systems. It is also much slower than the rapidly converging Newton Raphson scheme of the LNR method, taking several times longer to arrive at a stable (though often incorrect) solution for even simple systems. It is possible that a more complex stoichiometric scheme may be more reliable, but the poor cost scaling of stoichiometric algorithms make this of little practical interest.

Solver Scheme	LNR	Stoichiometric
Accuracy Against CEA	Valid	Invalid
Speed	Fast	Slow
Scaling	Good	Poor
Robustness	Good	Poor

Table 7.1-Summary of Method Comparison

In regards to the initial iteration, the simple averaged atom method has appeared to be the most effective in simple systems, but there are no guarantees that this will hold for complex or high energy systems. In these instances, it may prove to be more effective to use a stoichiometric initial state, recovered either from the steepest descent method in section 5.1.3 or the iterative scheme recovered from the stoichiometric formulation in section 5.2.3.

Reduced non-stoichiometric algorithms make up virtually all equilibrium solvers in the literature, and so it is not surprising that they outperform other competing methods. Similarly, the averaged atom method is similar to the method used by [1] in their CEA code, and so its performance is also in keeping with the literature.

7.2 Further Improvements To The Equilibrium Solver

Though the presented LNR algorithm is effective, there is still headway to be made in terms of improving its overall efficiency. Firstly, there exists some freedom in how the iteration equations of the LNR solver are formulated, namely in what correction variables are used and how the Newton Rapshon equations are defined before attempting to solve. It may be of interest to vary these and see under which conditions different formulations perform more effectively, perhaps even using a scheme that changes iteration scheme based on the system being solved or how close to the optimum the solver is. Two possible alterations to consider are:

1. Not making the energy terms unitless in the Newton Rapshon equations
2. Using linear correction variables on the temperature and/or species abundance

Another area of possible improvement is in selecting the step scaling factor and identifying when a solution attempt is failing to converge. At present, a value near unity is chosen, and then decreased for each successive attempt until convergence is achieved or global non-convergence is established. During each optimization attempt, the scaling factor is held constant, and a single attempt is only abandoned when a fixed iteration limit (usually 100 iterations) is exceeded. Setting this iteration limit too high results in wasting iterations on oscillating iterations, and too low may cause slowly converging systems to be falsely labeled as non-convergent. Selection of the step scaling factor presents a similar problem: too high and stability is harmed, too low and convergence is slowed to a crawl.

To alleviate this issue, we propose a number of possible avenues of future interest:

1. Refinement of the default values for maximum iteration and step scaling, and an automated process for making the best selection for a particular system and initial guess;
2. Replacement of the maximum iteration limit with a more robust check that determines if the system is failing to converge; and
3. A varied step scaling that ensures stability before increasing during the final approach, much as is done for the stoichiometric state algorithm in section 5.2.3

Through one or more of these, the speed of the solver may be improved in general without requiring additional effort from those tasked with making use of the solver. Ideally, the solver should be as efficient as possible without requiring any ‘tuning’ from the user, relying entirely on internal processes, something that is not entirely achieved at present.

It may also be of use to investigate a means of estimating a state nearby the optimum, possibly by a simplified thermodynamic model. Though it was found that stoichiometric initial states were not more efficient in general, beginning ‘near’ the optimum prevents divergence and decreases the number of iterations required to converge. In the context of CFD, the bulk of calculations will be performed with such a state provided by a previous optimized solution from a time or space adjacent cell, and the similar input conditions will allow the results of these cells to act as a good starting point for their neighbours.

Another means of improving the existing LNR solver may be to introduce means of avoiding the ‘overflow’ error that produces divergence in high temperature/density systems. Though high density systems are well beyond the scope of the ideal gas assumptions that the the solver is contingent on, extremely high temperature schemes may still be of a practical interest. This will be particularly true if high temperature thermodynamic data is made available in the future.

7.3 Additional Areas Of Interest

Though section 7.2 examines potential improvements specific to the LNR solver and the scope of this project, there is still room for future work, pertaining to Eilmer that lies beyond this scope. We identify two possible areas of interest:

1. The development of a robust and efficient stoichiometric solver
2. The extension of the LNR solver to account for non-ideal behavior, such as condensed phases and chemical interaction

Though the stoichiometric solver was found to be completely ineffective in the implementation presented here, there is good reason to believe that CEC solvers of this family are not ineffective in general, as they are present to some degree in the literature [3]. Given the issues with the presented non-stoichiometric algorithm outlined in section 6.2, we suggest three possible points of improvement:

1. Replacing the 'black box' Nelder Mead algorithm with a more efficient approach, e.g. steepest descent optimization
2. Altering the formulation to use linearly independent parameter space vectors in place of the cumbersome separation of species method presented here
3. Optimizing over a further restricted parameter space in cases of near-boundary solutions, so as to avoid the issues associated with the discontinuous windowing function

Though the stoichiometric solvers scale less efficiently than the LNR method in general, they do provide an advantage in how broadly they may be applied. If formulated correctly, a stoichiometric solver will be able to handle chemical interaction, non-ideal behavior and any number of other complicating factors, provided the system state is adequately defined. Should the scope of the CEC solver's applications be expanded, such an algorithm, if formulated to be sufficiently reliable, may provide a powerful tool to validate against or fall back on.

Towards the goal of handling non-ideal behavior, there exist a number of well established approaches in the literature. The approach of Gordon & McBride in their CEA analysis indicate that condensed phases (liquid or solid phases within the system mixture) may be accounted for by using linear correction variables instead of logarithmic for these species, thus allowing them to go to be iterated to zero concentration. Though not discussed in detail here, there is also a host of approaches for non-ideal chemical interaction in the system's state function, something that may need to be accounted for if, for example, ions were considered in addition to molecules and neutral atoms.

7.4 Integration & Implementation

This project has focused on the problem of fixed energy/volume equilibrium solving as an isolated pursuit, entirely separated from its practical implementation in a finite volume CFD code. This section contains a brief discussion of how the methods outlined in this report may be applied in a practical and efficient way. Only the LNR method is considered for implementation, for the reasons outlined in section 7.1.

As described previously, Eilmer's equilibrium flow calculations model each finite volume as a closed system of set internal energy and volume, and some predetermined elemental composition. To model equilibrium flow requires equilibrium solutions to be found for many cells at each iteration or timestep in the simulation. We then require an implementation that minimizes the cost of finding these many solutions.

In section 6, it is shown that beginning at a state 'near' the optimum solution decreases the number of iterations to converge. The simplest implementation, then, is to start each cell optimization with:

- A prior result of a cell that is space-adjacent or from the current cell's prior iteration; or
- Some non-equilibrium or low temperature chemistry approximation

Starting nearby, and using a small step scale to avoid oscillation, this will drastically decrease the cost as compared to starting with an averaged atom or 'blind' stoichiometric state for each cell. This has an advantage of working in any flow, even one in which the relative elemental composition of the flow is not constant.

In cases where the relative elemental composition is constant, and only the density varies from cell to cell, there exists a second option. In these cases, changing density is analogous to changing the volume for fixed elemental abundance, and so the equilibrium state really only has two degrees of freedom:

$$\vec{N} = \vec{N}(\rho, U)$$

As such, we may choose to calculate the equilibrium composition for many densities and energies prior to any flow calculations, and simply refer back to this when an equilibrium state is needed, by table lookup and/or interpolation, allowing many cells and iterations to be handled without needing to repeatedly call the solver.

8 Conclusion

Two algorithms have been proposed for solving the fixed-internal energy and fixed volume chemical equilibrium condition of an arbitrary chemical system of well mixed ideal gasses, one following the reduced nonstoichiometric methodology established in the literature, and another stoichiometric method intended to make use of a Nelder Mead optimizer. Both have been prototyped, tested, validated against externally established results and compared to one another. To summarize: the “Lagrangian Newton Raphson” approach was found to be fast, robust and accurate, and to have a computational cost that scales well with both system complexity and required precision. By contrast, the stoichiometric approach was found to be slow and imprecise, in addition to being unwieldy and scaling poorly with the sort of complexity expected in Eilmer’s equilibrium flow calculations.

The LNR algorithm consistently and continuously outperforms the stoichiometric algorithm by every meaningful measure, and so is recommended for integration into Eilmer, and for further refinement to improve its utility and efficiency. This result is in keeping with the literature; the LNR algorithm presented here is based entirely on the work of Zeleznik & Gordon [7] and Gordon & McBride [1], whose methods have been continuously refined and in consistent, uncontested use for the better part of a century. As such, it is not surprising to see that this approach outperforms any competitors.

9 REFERENCES

[1] S. Gordon and B. J. McBride, "Computer Program for Calculation of Complex Chemical Equilibrium Compositions & Applications (NASA RP-1311)," National Aeronautics & Space Administration, Cleveland, 1994.

[2] S. Gordon, F. J. Zeleznik and V. N. Huff, "A General Method for Automatic Computation of Equilibrium Compositions and Theoretical Rocket Performance of Propellants (NASA TN D-132)," National Aeronautic and Space Administration, Washington, 1959.

[3] W. R. Smith, "The Computation of Chemical Equilibria in Complex Systems," *Industrial Engineering Fundamentals,* vol. 19, pp. 1-10, 1980.

[4] J. Blecic, J. Harrington and M. O. Bowman, "TEA: A Code For Calculationg Thermochemical Equilibrium Abundances (arXiv:1505.06392v1)," University of Central Florida Department of Physics, Orlando, 2010.

[5] S. B. Pope, "Gibbs Function Continuation for the Stable Computation of Chemical Equilibrium," *Combustion and Flame,* vol. 139, pp. 222-226, 2004.

[6] S. Patil, R. C. Aiyer and K. C. Sharma, "Globally Convergent Computation of Chemical Equilibrium Composition," Department of Physics, University of Pune, Pune, 2007.

[7] F. J. Zeleznik and S. Gordon, "An Analytical Investigation of Three General Methods of Calculating Chemical Equilibrium (NASA-TN D-473)," National Aeronautics And Space Administration, Washington, 1960.

[8] H. J. Kandiner and R. S. Brinkley, "Calcualtion of Complex Equilibrium Relations," *Industrial And Engineering Chemistry,* vol. 42, no. 5, 1950.

[9] K. Denbigh, Principles of Chemical Equilibrium 4th Ed, Cambridge: Cambridge University Press, 181.

[10] C. De Capitani and T. H. Brown, "The Computation of Chemical Equilibrium in Complex Systems," *Geochimica el Cosmochimica Acta,* vol. 51, pp. 2639-2652, 1987.

[11] W. B. White, S. M. Johnson and G. B. Dantzig, "Chemical Equilibrium in Complex Mixtures (P-1059)," Rand Corporation, Santa Monica, 1957.

[12] D. R. Cruise, "Notes on the Rapid Computation of Chemical Equilibrium," *Rapid Computation of Chemical Equilibria,* vol. 68, no. 12, pp. 3797-3802, 1964.

[13] V. N. Huff, G. Sanford and V. E. Morrel, "General Method and Thermodynamic Relations for Computation of Equilibrium Composition and Temperature of Chemical Reactions," in *Report 1037*, National Advisory Committee for Aeronautics, 1951, pp. 831-885.

[14] W. H. Press, S. A. Teukolsky, W. T. Vetterling and F. P. Flannery, Numerical Recipes in C, Cambridge: Cambridge University Press, 1992, pp. Pages 379-383.

[15] R. J. Clasen, "The Numerical Solution of the Chemical Equilibrium Problem (C&EE49)," Rand Corporation,

Santa Monica, 1965.

[16] H. Greiner, "Computing Complex Chemical Equilibria by Generalized Linear Programming," *Mathematical Computational Modelling,* vol. 10, pp. 529-550, 1988.

[17] B. J. McBride, M. J. Zehe and S. Gordon, "NASA Glenn Coefficients for Calculating Thermodynamic Properties of Individual Species (NASA/TP—2002-211556)," National Aeronautics and Space Administration, Glenn Research Center, Cleveland, 2002.

[18] National Institute of Standards and Technology, "NIST Chemistry WebBook," U.S. Secretary of Commerce , 2017. [Online]. Available: https://webbook.nist.gov/chemistry/.

APPENDIX A: EQUILIBRIUM SOLVER PROTOTYPE PYTHON CODE

```
from __future__ import division
from Database_Schomer import species,R_u
from Simclass import *
import numpy as np
from squareup import *
from scipy.optimize import root,    minimize,   bisect
from math import log as mathlog

'''
This is the prototype solver for the eilmer chemical equilibrium
module. This file contains the solver for both the stoichiometric
and LNR methods, each of which is called by the 'solve' method.

'Simclass.py' defines the species properties.

Hugh McDougall, 1/6 2018
'''

ZG_SWITCH=True      #A switch to apply the mu/RT+=1 switch of the Z&G formulation

class solver:
    def __init__(self,sim):
        '''
        A solver is created with a 'simulation' as a template
        to read its physical properties from. These can be changed
        at runtime. A simulation has set U, V and species abundances.
        '''

        self.debug=False

        self.U0 =sim.u          #Prescribed internal energy
        self.V  =sim.V          #Prescribed volume

        self.I=sim.N_spec       #The number of chemical species
        self.J=sim.N_atom       #The number of atomic species

        self.b      =np.zeros([sim.N_atom])     #Elemental abundance atom
        self.C      =np.zeros([self.J,self.I])  #Elemental conservation matrix

        #Seed conditions for solver
        self.T0 =298
        self.N0 =np.zeros([self.I])+(sum(self.b)/self.I)
        self.n0 =np.zeros(self.I-self.J)+(sum(self.b)/self.I)
        self.PI0=np.zeros([self.J])

        #Make Conservation matrices
        for i,entry in zip( range(self.I),  sim.speclist):
            for j,name in zip(range(self.J),sim.atomlist):
                for atomname in entry[0].atom.keys():
                    if atomname==name:
                        self.b[j]       +=entry[0].atom[atomname]*entry[1]
                        self.C[j,i]      =entry[0].atom[atomname]

        #Sort into primary and secondary species using squareup()
        """
        This is the separation of species step.
        From this point forward, all ordering of chemical species
        is of the form N=[n_primary,n_secondary]

        This step is only reversed in the output of solve()
        """
        self.A, self.D, self.cutcols=squareup(self.C)
        self.Ainv=np.linalg.inv(self.A)
        self.sorter,self.unsorter=assembly_vectors(self.I,self.cutcols)

        #split species list
        self.specs    =(np.array(sim.speclist)[:,0])[self.sorter]

        #Establish primary species boundaries
        self.nmax=self.getnmax()
```

```
    #Sys State Evaluation Functions
    #---------------------------------
    def getnmax(self):
        '''
        Makes a rough estimate of the 'square' region that
        the physical solution space lies within.

        Takes no arguments.
        Returns an array of primary species
        '''
        nmax=np.zeros([self.I-self.J])
        for l in range(self.I-self.J):
            for j in range(self.J):

                nmax[l]=min([(self.b[j]/self.D[j,l])    for    j    in    range(self.J)    if
self.D[j,l]!=0])

        return(nmax)

    def resetseed(self):
        '''
        Resets the solver's initial estimate for the solvers
        by way of the averaged atom method.
        Takes no arguments.
        Acts directly on the solver variables.
        '''
        self.N0=np.zeros([self.I])+sum(self.b)/self.I
        self.T0=298
        self.PI0=np.zeros([self.J])
        self.n0=self.N0[self.sorter][:self.I-self.J]  #Read  the  primary  species  from  current
seed


    #Sys State Evaluation Functions
    #---------------------------------
    def sys_internalenergy(self,N,T):
        '''
        Calculates the internal enegry for a set temperature
        and species count. For use in temp_solve

        Takes (sorted) complete species list and temperature.
        Returns internal energy.
        '''
        assert len(N)==len(self.specs), "Wrong count length in sys_internalenergy"

        outU=0
        for entry,n in zip(self.specs,N):
            outU+=entry.u(T)*n
        return(outU)

    def sys_enthalpy(self,N,T):
        '''
        Calculates the enthalpy for a set temperature and species count
        For use in getting gibbs

        Takes (sorted) complete species list and temperature.
        Returns total enthalpy.
        '''
        assert len(N)==len(self.specs), "Wrong count length in sys_enthalpy"

        outH=0
        for entry,n in zip(self.specs,N):
            outH+=entry.h(T)*n
        return(outH)

    def sys_entropy(self,N,T):
        '''
        Calculates the entropy for a set temperature and species count
        For use in getting gibbs

        Takes (sorted) complete species list and temperature.
        Returns total entropy.
        '''
        assert(min(N))>0,"min(N)<0 in entropy eval"
        assert(T>0), "Negative temp in entropy eval"
        outS=0
```

```
        #Perform Summation
        for entry,n in zip(self.specs,N):
            if n>0:
                si  =   entry.s(T)
                si +=   -R_u*mathlog(n*R_u*T/(self.V*101.3*10**3))
                outS+=n*si

        return(outS)

    def sys_gibbs(self,N,T):
        '''
        Calculates the entropy for a set temperature and species count
        For use in getting gibbs

        Takes (sorted) complete species list and temperature.
        Returns total gibb's free energy.
        '''

        outG=self.U0+(T*sum(N)*R_u)-T*self.sys_entropy(N=N,T=T)
        return(outG)

    #Gibbs Solver Functions
    #-----------------------------------
    def temp_solve(self,N):
        '''
        Uses a 1D rootfinder to get the temp for a known
        species count and set internal energy.
        'root' imported from scipy

        Takes a sorted complete species list
        Returns the temperature.
        '''

        def f(T):
            return(self.sys_internalenergy(N=N,T=T)-self.U0)
        out=root(f   , x0=self.T0)

        if float(out.x)<0:
            print("WARNING! No valid temp solution found for N=",N)
        return(float(out.x))

    def n_to_N(self,n):
        '''
        Takes a primary species vector 'n' and sorts into secondary
        '''
        ndash=np.dot(   self.Ainv,   (self.b-np.dot(self.D,n)     ))
        N=np.hstack([n,ndash])
        return(N)

    def check_physical(self,n=None,N=None):
        '''
        Checks to see if a system state is physical or not

        Takes either primary 'n' or complete species 'N' array
        Returns as a boolean.
        '''

        if type(N)==type(None):
            N=self.n_to_N(n)

        if min(N)<=0:
            return(False)

        if self.sys_internalenergy(N=N,T=0)>self.U0:
            return(False)

        return(True)

    def genseed(self):
        '''
        Randomly pings points in the in the n<nmax domain
        until a physical solution is found

        Takes no arguments
        Returns an array of primary species
        '''
```

```
        maxits=1E5
        noits=0
        ntest=np.zeros([self.I-self.J])

        while self.check_physical(n=ntest)==False:
                if noits<maxits:
                    ntest=np.random.rand( len(ntest) ) * self.nmax
                    noits+=1
                else:
                    print("WARNING!: Failed to find physical seed")
                    return(None)

        return(ntest)

    def windowgibbs(self,n=None,N=None):
        '''
        The potential that is fed into the nelder solver
        Accepts a primary species vector, ensures a physical solution

        Takes either primary 'n' or complete species 'N' array
        Returns the gibb's function with a window applied.
        '''
        if type(N)==type(None):
            N=self.n_to_N(n)

        if self.check_physical(N=N)==False:
            return(1E100)
        else:
            T=self.temp_solve(N)
            G=self.sys_gibbs(N=N,T=T)
            return(G)

    def LSD_step(self,Y,logN,alpha=1):
        '''
        Performs the newton rapshon step to locate approach the laplace potential
        minimum.

        Takes array inputs in the form:
            -logN=log|n|
            -Y=[log|T|,pi]
            -(optional) alpha=alpha

        Outputs as Y and logN arrays.
        '''

        #Read matrices to extract physical values
        T   =np.exp(Y[0])
        N   =np.exp(logN)+1E-16

        if self.debug:
            print("logN:    ",logN)
            print("N:   ",np.exp(logN))

        if self.debug:
            print(N,T)

        J=np.zeros([self.J+1,self.J+1])
        F=np.zeros([self.J+1])
        A=self.C[:,self.sorter]

        B=np.dot(A,N) #Current  nodes

        #J0 and F0
        J[0,0]  =self.U0/R_u/T
        F[0]    =self.U0/R_u/T
        F[1:]   -=self.b

        MU_RT   =np.zeros([self.I])
        K       =np.zeros([self.I,self.J+1])
        K[:,1:] +=A.T

        #Summate over i
        for i in range(self.I):

            ui  =self.specs[i].u(T)
            hi  =self.specs[i].h(T)
```

```
            ui_RT   =ui/R_u/T
            hi_RT   =hi/R_u/T
            mui_RT  =hi_RT-self.specs[i].s(T)/R_u+(logN[i]+mathlog(R_u*T/self.V/101.3/10**3))

            if ZG_SWITCH==True:
                mui_RT+=1

            cv_R    =self.specs[i].cp(T)/R_u-1

            J[0,0]  +=N[i]*(cv_R-ui_RT*(2-hi_RT))
            F[0]    +=N[i]*ui_RT*(mui_RT-1)

            MU_RT[i]=mui_RT
            K[i,0]  =1-hi_RT

            for row in range(1,self.J+1):
                    for col in range(1,self.J+1):
                        J[row,col]+=A[col-1,i]*A[row-1,i]*N[i]

            for row in range(1,self.J+1):
                J[row,0]+=A[row-1,i]*N[i]*(1-hi_RT)
                F[row]  +=A[row-1,i]*N[i]*(1-mui_RT)

            for col in range(1,self.J+1):
                J[0,col]+=-N[i]*ui_RT*A[col-1,i]

        #Calculate Update
        JINV=np.linalg.inv(J)
        dY  =np.dot(JINV,F)
        dlogN=-MU_RT-np.dot(K,dY)

        #Get outputs
        Yout    =np.zeros([self.J+1])
        Yout[0] =Y[0]+dY[0]*alpha
        Yout[1:]+=dY[1:]

        logNout=logN+dlogN*alpha

        return(Yout,logNout)

    #Equilibrium Optimizers
    #---------------------------------
    def nelder_optimizer(self,n0,tol=1E-10):
        '''
        Solves the equilibrium concentrations by minimizing the
        windowgibbs function. Called by solve().

        Takes arguments
            -Initial primary species vector 'n0'
            -Tolerance 'tol', passed to existing Eilmer solver.

        Outputs are in the form of a primary species array
        '''

        out=minimize(self.windowgibbs,x0=n0,method="nelder-mead",tol=tol)
        return(out.x,out.fun)

    def laplace_optimizer(self,Y0,logN0,tol=1E-10,maxits=1E5,alpha=0.9):
        '''
        Solves the minimum of the laplace potential by itterating
        the newton rapshon equation. Called by solve().

        Takes array inputs in the form:
            -logN0=log|n|
            -Y0=[log|T|,pi]
        And optional inputs
            -(optional) alpha=0.9
            -(optional) maxits=1E5
            -(optional) tol=1E-10

        Outputs as Y and logN arrays.
        '''

        stepsize=tol*2

        #Set up initial conditions
        Y   =Y0
```

```
        logN=logN0

        itno=0
        while stepsize>=tol:
            assert itno<maxits, "Itteration Number exceeded in laplace solver"
            itno+=1
            Ystep,logNstep=self.LSD_step(Y=Y,logN=logN,alpha=alpha)
            stepsize=min(min(abs(Ystep-Y)),min(abs(logNstep-logN)))

            Y   =Ystep
            logN=logNstep

        return(Y,logN)

    #Solver Wrapper
    #---------------------------------
    def solve(self,method='nelder-mead',tol=1E-10,maxits=1E5,alpha=0.9):
        '''
        The actual solver that uses a given method
        to calculate the equilibrium conditions.

        Called with solver method as a string:
            method='nelder-mead'
        or
            method='LSD'

        Takes optional inputs
            -(optional) alpha=0.9
            -(optional) maxits=1E5
            -(optional) tol=1E-10

        Returns a tuple of the form:
            (N,T,P)
        Where:
            N    is an array of solution species abundances sorted in the order the species
were added to the solver's parent sim
            T is the solution temperature
            P is the corresponding pressure.

        '''

        assert method in ['nelder-mead','steepest-descent','LSD'], method+" is not a valid
solver method"

        #Check if using constrained method
        constrained = method in ['nelder-mead','steepest-descent']

        #Get initial conds
        if not constrained:
            if min(self.N0)<=0:
                self.N0=self.N0*0+sum(self.b)/self.I

        self.n0=self.N0[self.sorter][:self.I-self.J]  #Read the primary species from current
seed

        if constrained:
            if self.check_physical(self.n0)==False: #If non physical, make a new one
                print("Bad initial seed, making new")
                self.n0=self.genseed()
                self.N0=self.n_to_N(self.n0)
                print("New Seed:",self.N0)

        #Actual solver runtime
        #----------------------
        if method=='nelder-mead':
            #Get results from nelder mead method
            outx,outf=self.nelder_optimizer(self.n0,tol=tol)

            #Sort into meaningful information
            outputN=self.n_to_N(outx)
            outputT=self.temp_solve(outputN)
            outputN=outputN[self.unsorter]

        elif method=='steepest-descent':
            print("Steepest Descent not yet implemented")
            return(None)
```

```
        elif method=='LSD':
            #Get initial conditions
            Y0      =np.zeros([self.J+1])
            Y0[0]   =mathlog(self.T0)
            Y0[1:]   +=self.PI0
            logN0   =np.log(self.N0[self.sorter])

            try:
                try:
                    Y,logN
=self.laplace_optimizer(Y0,logN0,tol=tol,maxits=maxits,alpha=alpha)
                except:
                    alpha=alpha
                    while alpha>=0.01:
                        try:
                            print("Itterations failed to converge. Decreasing Step Scaling")
                            alpha*=0.5
                            Y,logN
=self.laplace_optimizer(Y0,logN0,tol=tol,maxits=maxits,alpha=alpha)
                            break
                        except:
                            continue
                    assert alpha>=0.01, "Simulation failed to converge"
            except:
                print("Itterations failed to converge. Resetting Seed")
                self.resetseed()
                outputN,outputT,outputP=solve(self,method='LSD',tol=tol,maxits=maxits)

            #Sort into meaningful information
            self.PI0=np.zeros([self.J])+Y[1:]
            outputN=np.exp(logN)
            outputT=np.exp(Y[0])
            outputN=outputN[self.unsorter]


        outputP=sum(outputN*R_u*outputT/self.V)

        self.N0=outputN
        self.T0=outputT

        return(outputN,outputT,outputP)
```

APPENDIX B: LNR EQUILIBRIUM SOLVER

```
import std.stdio;
import nm.bbla;
import simclass;
import specclass;
import matrix exp;
import std.file;
import std.math : isNaN, pow;

double P0 = 101.3*1000;

/*
--------------------
solver.d
--------------------
The solver model that uses the LNR method to calculate
chemical equilibrium conditions for set internal energy and volume.

Note that this code refers to the lagrangian newton rapshon (LNR)
method as the lagrangian steepest descent (LSD) method.

        -Hugh McDougall, 1/6/2018

NOTES:
        All methods are stored in the "solver" class.

        At present, doesn't have a 'loading' process in the init function,
        the input conditions and species must be defined manually.

        All float variables are taken to be at the "double" precision level,
        in line with the linear algebra module that's been made available in Eilmer
*/

struct LSDstruc{
        /*
        This is a structure to help store the
        itterations in the LNR step method.
        */
        Matrix Y;
        Matrix logN;
        bool converged;
}
```

```
class solver{
	//List of variables stored within this class
	//Element properties
	double U0;	//Total energy in volume element
	double V;	//Volume of element

	bool DEBUG;

	//Chemical abundances
	int I;		//No. Species
	int J;		//No. Elements
	Matrix b;	//Atomic counts, total. I length vector
	Matrix C;	//Atomic counts per  species. IxJ matrix

	species[] specs; //A list of the chemical species

	//The Initial Seed for input to the LSD
	double T0;	//Temperature
	Matrix N0;	//Species counts
	Matrix PI0;	//Lagrange multipliers


	//__init__ function
	this(){
		/*
		Later on, this will be used to directly load a 'simclass' struc
		*/

		bool DEBUG;

		//Element properties
		this.U0 =U0;	//Total energy in volume element
		this.V  =V;		//Volume of element

		//Chemical abundances
		this.I  =I;		//No. Species
		this.J  =J;		//No. Elements
		this.b  =b;		//Atomic counts, total. I length vector
		this.C  =C;		//Atomic counts per  species. IxJ matrix

		this.specs=specs; //A list of the chemical species

		//The Initial Seed for input to the LSD
		this.T0 =T0;	//Temperature
		this.N0 =N0;	//Species counts
		this.PI0=PI0;	//Lagrange multipliers
	}


	//Gibbs Solver Functions
	//----------------------------------

	/*
	This section was mostly geared towards the nelder mead optimizer
	and generating seeds for it. It might be useful for the LSD method as well,
	but will require the complexity of the "separation of species" to also be built in

	I might add this in another module, come to think of it.
	*/

	void resetseed(int noresets, int method=2,float tol=1E-2){
			if(method==1){
				this.N0 =zeros(this.I,1)+sum(this.b)/this.I/pow(10,noresets);
				this.T0 =298;
				this.PI0=zeros(this.J,1);
			}
			else if(method==2){
				writeln(
				"resetseed() being called with method 2, loop ",noresets);
				resetseed(1, 1, tol);

				Matrix Ni		=	this.N0*1;
				Matrix logNi		=	Mlog(Ni);
				double T		=	this.T0;
				double logT		=	mathlog(T);
				Matrix deltab		=	zeros(this.J,1);
				Matrix ui_RT		=	zeros(this.I,1);
```

```
                        double cv_R             =       0;
                        double dU_RT            =       0;
                        double P                =       tol*2;
                        Matrix grad             =       zeros(this.I+1,1);
                        Matrix dx               =       zeros(this.I+1,1);

                        while(P>tol){

                                grad            =       zeros(this.I+1,1);
                                cv_R            =       0;
                                deltab          =       dot(this.C,Ni)-this.b;

                                foreach(i;0..this.I){
                                        ui_RT[i,0]      =this.specs[i].u(T)/R_u/T;
                                        cv_R            +=this.specs[i].cv(T)/R_u;
                                }

                                dU_RT   =vdot(Ni,ui_RT)-this.U0/R_u/T;
                                P               =vdot(deltab,deltab)+dU_RT*dU_RT;

                                foreach(i;0..this.I){
                                        foreach(j;0..this.J){
                                                grad[i+1,0]+=deltab[j,0]*this.C[j,i];
                                        }
                                        grad[i+1,0]+=dU_RT*ui_RT[i,0];
                                        grad[i+1,0]*=Ni[i,0];
                                }
                                grad[0,0]=dU_RT*(cv_R-dU_RT);

                                grad=grad*2;

                                dx=grad*-(P/vdot(grad,grad));

                                foreach(i;0..this.I){
                                        logNi[i,0]+=dx[i+1,0];
                                }
                                logT+=dx[0,0];

                                T=mathexp(logT);
                                Ni=Mexp(logNi);
                        }

                this.N0=Ni;
                this.T0=T;
                }
        }
    //Equilibrium Optimizers
    //-----------------------------------

        LSDstruc LSD_Step(LSDstruc prevstep){
                /*
                Takes an input Y,PI in the form of an LSDstruc
                Returns a step of a similar form.

                'Alpha' is a variable step scaling, usually controlled in the solver() function
                */

                LSDstruc outstep;

                 //Read matrices to extract physical values
                Matrix Y        =prevstep.Y;
                Matrix logN     =prevstep.logN;

                double T        =mathexp(Y[0,0]);
                Matrix PI       =zeros(this.J,1);
                foreach(j; 0..this.J){
                        PI[j,0]=Y[j+1,0];
                }

                 Matrix N       =Mexp(logN);
```

```
        //Define step matrices
        Matrix J=zeros(this.J+1,this.J+1);    //Linearizer matrix to invert step
        Matrix F=zeros(this.J+1,1);           //Function eval matrix
        Matrix K=zeros(this.I,this.J+1);      //Used to get delta-N's

        Matrix JINV;
        Matrix dY;
        Matrix dlogN;

        //Physical values
        Matrix A=this.C; //Just kept short for consistency
        Matrix B=dot(A,N); //Current  values of 'b'

        //Comp Savers
Matrix MU_RT   =zeros(this.I,1); //These values show up a lot.

//J0 and F0
J[0,0]  =this.U0/R_u/T;
F[0,0]  =this.U0/R_u/T;
        foreach(j; 0..this.J){
                F[j+1,0]-=this.b[j,0];
        }

        foreach(i;0..this.I){
                foreach(j;0..this.J){
                K[i,j+1]=A[j,i];
                }
        }

                //Relevant thermo props
        double ui;
        double hi;
        double si;
                //Unitless thermo props
        double ui_RT;
        double hi_RT;
        double mui_RT;
        double cv_R;

//Summate over i
foreach(i; 0..this.I ){

                ui      =this.specs[i].u(T);
                hi      =this.specs[i].h(T);
                si      =this.specs[i].s(T);

    ui_RT   =ui/R_u/T;
    hi_RT   =hi/R_u/T;
    mui RT  =hi RT-si/R u+(logN[i,0]+mathlog(R_u*T/this.V/P0));
    cv_R    =this.specs[i].cp(T)/R_u-1;

    J[0,0]  +=N[i,0]*(cv_R-ui_RT*(2-hi_RT));
    F[0,0]  +=N[i,0]*ui_RT*(mui_RT-1);

    MU RT[i,0] =mui RT;
    K[i,0]     =1-hi_RT;

    foreach(int row; 1..this.J+1 ){
            foreach(int col; 1..this.J+1 ){
                J[row,col]+=A[col-1,i]*A[row-1,i]*N[i,0];
                            }
                }

    foreach(row; 1..this.J+1 ){
        J[row,0]      +=A[row-1,i]*N[i,0]*(1-hi_RT);
        F[row,0]      +=A[row-1,i]*N[i,0]*(1-mui_RT);
                }

    foreach(col; 1..this.J+1 ){
        J[0,col]+=-N[i,0]*ui_RT*A[col-1,i];
                }
        }
```

```
        //Calculate Update

        JINV    =inverse(J);
        dY      =dot(JINV,F);
        dlogN   =-1*dot(K,dY);
                foreach(i;0..this.I){
                        dlogN[i,0]-=MU_RT[i,0];
                }

        //Get outputs
        Matrix Yout      =dY;
        Yout[0,0]       +=Y[0,0];

        Matrix logNout  =logN+dlogN;

                outstep.Y=Yout;
                outstep.logN=logNout;

        return(outstep);
        }

        LSDstruc LSD_Optimizer(LSDstruc inputs,
                               int maxits=100000,
                               double tol=1E-10,
                               double alpha=0.9,
                               int step_dec_goal=5){
                /*
                Solves the minimum of the laplace potential by itterating
                the newton rapshon equation.
                Outputs as LSDstruc. For internal use within the module,
                use the solve() function in main runtime

                inputs:
                        inputs: Initial inputs in the form of an LSDstruc
                        maxits: maximum number of LSD_Step itteratons before reset
                        tol:    How small the error should be for convergence
                */

                writeln("LSD Optimizer being called with alpha=",alpha);
                LSDstruc currit;
                LSDstruc nextit;
                LSDstruc deltait;

                //Arbitrarily high
                double stepsize=tol*2;
                double old_stepsize;

                //Get initial conditions
                currit.Y        =inputs.Y;
                currit.logN     =inputs.logN;
                currit.converged=false;

                int itno=0;
                writeln("Beginning Itteration loop in LSD_optimizer");
                writeln("Input conditions are:");
                writeln("T: ",mathexp(currit.Y[0,0]));
                writeln("N: ",Mexp(currit.logN));

                File debugfile = File("debugfile.txt","w+");

                bool errorraised=false;
                int step_dec=0;
                while(stepsize>=tol || step_dec<=step_dec_goal){
                        writeln(itno," ",stepsize," ",stepsize<=tol);

                        itno++;

                        //Halting condition
                        if(itno>=maxits){
                                printf("Maximum itterations exceeded. \n");
                                currit.converged=false;
                                break;
                        }

                        //Get step from the LSD Step function
                        try{nextit=this.LSD_Step(currit);}
                        catch{errorraised=true; break;}
```

```
                        deltait.Y       =nextit.Y       -       currit.Y;
                        deltait.logN    =nextit.logN    -       currit.logN;

                        //Evaluate "size" of step
                        old stepsize=stepsize;
                        stepsize=norm(abs(
                                vstack([
                                deltait.Y,
                                deltait.logN]
                                )
                        ));

                        if(isNaN(stepsize)){errorraised=true; break;}

                        if(old_stepsize>=stepsize || stepsize<tol){
                                step_dec++;
                        }
                        else{
                                step_dec=0;
                        }

                        //Update
                        currit.Y                =currit.Y       +       deltait.Y       ;
                        currit.logN             =currit.logN    +       deltait.logN*alpha;


                        if(this.DEBUG){
                                debugfile.writeln(itno,"\t",stepsize);

                        }
                }

                debugfile.close();

                //Protection against non-physical values
                if(NaNin(currit.Y) || NaNin(currit.logN)){return(currit);}

                if(itno<maxits && errorraised==false){currit.converged=true;}
                return(currit);
        }


        //Solver Wrapper
        //-----------------------------------

        Matrix solve(double tol=1E-10, int maxits=100000, double alpha=0.9, int maxresets=10,
bool updateseed=true){
                /*
                The actual solver function:
                -Updates the T0,N0,PI0 variables of the solver
                -Returns matrix of the form (T,P,n0,n1...nI)
                */

                LSDstruc solvetest;                                             //Stores the output
of LSD Optimizer, including a boolean tag to describe wether it converged or not
                solvetest.converged=false;
                Matrix phys_output = zeros(this.I+2,1); //Actual output variable
                double alpha_i;
                int noresets=0;

                //If STILL not converged, try rejiggering the seed and go from the top
                while(solvetest.converged==false && noresets<maxresets){
                        alpha_i=alpha;

                        //Run loop. If not converged, halve step size and try again
                        while(alpha_i>0.1 && solvetest.converged!=true){

                                //Process seed values into init conditions
                                solvetest.Y                             =zeros(this.J+1,1);
                                solvetest.Y[0,0]                +=mathlog(this.T0);
                                foreach(j;0..this.J){
                                        solvetest.Y[j+1,0]      +=this.PI0[j,0];
                                }

                                solvetest.logN =Mlog(this.N0);
```

```
                                solvetest=LSD_Optimizer(solvetest, maxits, tol, alpha_i);
                                writeln(solvetest.converged);
                                if(solvetest.converged!=true){
                                        alpha_i*=0.5;
                                        printf("Failed to converge, halving step size to %lf
\n",alpha i);
                                }
                        }

                        //If failed to converge, reset seed and retry
                        if(solvetest.converged!=true){
                                if(noresets<maxresets){
                                        writeln("Still failed to converge. Resetting seed");
                                        resetseed(noresets);
                                        noresets++;
                                }
                                else if(noresets==maxresets){
                                        writeln("Maximum number of resets exceeded");
                                        break;
                                }
                        }

                        //Final check to make sure converged point is physical
                        double solveT  =       mathexp(solvetest.Y[0,0]);
                        Matrix solveN0 =       Mexp(solvetest.logN);

                }

                if(solvetest.converged){
                        //Update seed conditions
                        writeln("Itterations complete \n");
                        if(updateseed){
                                this.T0 =       mathexp(solvetest.Y[0,0]);
                                this.N0 =       Mexp(solvetest.logN);
                                foreach(j;0..this.J){
                                        this.PI0[j,0]= solvetest.Y[j+1,0];
                                }
                                writeln("Seed updated, returning values\n");
                        }

                        phys_output[0,0]       =this.T0;
                        phys_output[1,0]       =this.T0*sum(this.N0)*R_u/this.V;
                        foreach(i;0..this.I){
                                phys_output[i+2,0]     =this.N0[i,0];
                        }

                        writeln("Output formatted. Solver done\n");
                }
                else{
                        writeln("Simulation failed to converge");
                }
                //Return useful information
                return(phys_output);
        }

}
```

APPENDIX C: SHOMATE AND GLENN COEFFICIENTS FOR SIMULATED SPECIES

Shomate Coefficients

The following are the Shomate coefficients for the species simulated during model validation for the solvers. These are used in the Shomate equations, in which a species' thermodynamic properties are defined as:

$$c_p(T) = A + Bt + Ct^2 + Dt^3 + \frac{E}{t^2},$$

$$t = \frac{T}{1000\ K} \qquad h(T) = h_f + 1000 \times \left[At + \frac{Bt^2}{2} + \frac{Ct^3}{3} + \frac{Dt^4}{4} - \frac{E}{t} + F - H\right]$$

$$s^0(T) = Aln|t| + Bt + \frac{Ct^2}{2} + \frac{Dt^3}{3} - \frac{E}{2t^2} + G$$

The coefficients used here are based acquired from the NIST web-book [18], a publically available resource.

Species	N
Bracket Start	**200 K**
Bracket End	**500 K**
A	21.13581
B	-0.388842
C	0.043545
D	0.024685
E	-0.025678
F	466.311
G	178.8263
H	472.6832

Species	N2		
Bracket Start	**200 K**	**500 K**	**6000 K**
Bracket End	**500 K**	**6000 K**	**20000 K**
A	28.98641	19.50583	35.51872
B	1.853978	19.88705	1.128728
C	-9.647459	-8.598535	-0.196103
D	16.63537	1.369784	0.014662
E	0.000117	0.527601	-4.55376
F	-8.671914	-4.935202	-18.97091
G	226.4168	212.39	224.981
H	0	0	0

Species	N3
Bracket Start	**200 K**
Bracket End	**500 K**
A	0
B	29.099
C	0
D	0
E	0
F	0
G	0
H	0

Species	H2
Bracket Start	**200 K**
Bracket End	**500 K**
A	29.099
B	0
C	0
D	0
E	0
F	0
G	165.9090475
H	0

Species	H
Bracket Start	**200 K**
Bracket End	**500 K**
A	20.785
B	0
C	0
D	0
E	0
F	0
G	139.8636054
H	0

Species	O
Bracket Start	**200 K**
Bracket End	**500 K**
A	20.785
B	0
C	0
D	0
E	0
F	0
G	186.2226054
H	0

Species	O2		
Bracket Start	**200 K**	**500 K**	**6000 K**
Bracket End	**500 K**	**6000 K**	**20000 K**
A	0	500	2000
B	31.32234	30.03235	20.91111
C	-20.23531	8.772972	10.72071
D	57.86644	-3.988133	-2.020498
E	-36.50624	0.788313	0.146449
F	-0.007374	-0.741599	9.245722
G	-8.903471	-11.32468	5.337651
H	246.7945	236.1663	237.6185

Species	O3	
Bracket Start	**200 K**	**1200 K**
Bracket End	**1200 K**	**20000 K**
A	21.66157	57.81409
B	79.86001	0.730941
C	-66.02603	-0.039253
D	19.58363	0.00261
E	-0.079251	-3.560367
F	132.9407	115.7717
G	243.6406	294.5607
H	142.674	142.674

Species	H2O	
Bracket Start	**200 K**	**1700 K**
Bracket End	**1700 K**	**20000 K**
A	30.092	41.96426
B	6.832514	8.622053
C	6.793435	-1.49978
D	-2.53448	0.098119
E	0.082139	-11.15764
F	-250.881	-272.1797
G	223.3967	219.7809
H	-241.8264	-241.8264

Species	H2O2	
Bracket Start	**200 K**	
Bracket End	**20000 K**	
A	34.25667	
B	55.18445	
C	-35.15443	
D	9.08744	
E	-0.422157	
F	-149.9098	
G	257.0604	
H	-136.1064	

Species	H3N	
Bracket Start	**200 K**	**1400 K**
Bracket End	**1400 K**	**20000 K**
A	19.99563	52.02427
B	49.77119	18.48801
C	-15.37599	-3.765128
D	1.921168	0.248541
E	0.189174	-12.45799
F	-53.30667	-85.53895
G	203.8591	223.8022
H	-45.89806	-45.89806

Species	H4N2	
Bracket Start	**200 K**	**2000 K**
Bracket End	**2000 K**	**20000 K**
A	35.1824	121.401
B	96.0526	4.81688
C	-40.5013	-0.763012
D	6.66807	0.043232
E	-0.874233	-40.7865
F	77.9915	-11.3811
G	249.425	305.344
H	95.3534	95.3534

Glenn Coefficients

The following are the Glenn Coefficients for atomic and diatomic nitrogen, as were used to compare the results for nitrogen dissociation against those of CEA. These are used to define a species' thermodynamic properties by the following equations:

$$\frac{c_p(T)}{R} = a_1 T^{-2} + a_2 T + a_3 + a_4 T + a_5 T^2 + a_6 T^3 + a_7 T^4$$

$$\frac{h(T)}{RT} = \frac{h_f}{RT} - \frac{a_1}{T^2} + a_2 \frac{ln(T)}{T} + a_3 + \frac{a_4 T}{2} + \frac{a_5 T^2}{3} + \frac{a_6 T^3}{4} + \frac{a_7 T^4}{5} + \frac{b_1}{T}$$

$$\frac{s^0(T)}{R} = -\frac{a_1}{2T^2} - \frac{a_2}{T} + a_3 \ln|T| + a_4 T + \frac{a_5 T^2}{2} + \frac{a_6 T^3}{3} + \frac{a_7 T^4}{4} + b_2$$

The Glenn coefficients are publically available, and here have been sourced from "NASA Glenn Coefficients for Calculating Thermodynamic Properties of Individual Species", NASA/TP—2002-211556 [17].

Species	N		
Bracket Start	**200 K**	**500 K**	**6000 K**
Bracket End	**500 K**	**6000 K**	**20000 K**
a_1	0	88765.0138	547518105
a_2	0	-107.12315	-310757.498
a_3	2.5	2.362188287	69.1678274
a_4	0	2.92E-04	-6.85E-03
a_5	0	-1.73E-07	3.83E-07
a_6	0	4.01E-11	-1.10E-11
a_7	0	-2.68E-15	1.28E-16
b_1	56104.6378	56973.5133	2550585.618
b_2	4.193905036	4.865231506	-584.8769753

Species	N		
Bracket Start	**200 K**	**500 K**	**6000 K**
Bracket End	**500 K**	**6000 K**	**20000 K**
a_1	22103.71497	587712.406	831013916
a_2	-381.846182	-2239.249073	-642073.354
a_3	6.08273836	6.06694922	202.0264635
a_4	-0.008530914	-0.000613969	-0.03065092
a_5	1.38E-05	1.49E-07	2.49E-06
a_6	-9.63E-09	-1.92E-11	-9.71E-11
a_7	2.52E-12	1.06E-15	1.44E-15
b_1	710.846086	12832.10415	4938707.04
b_2	-10.76003744	-15.86640027	-1672.09974