Design Optimization of a Porous Radiant Burner

by

Adam Philip Horsman

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Mechanical Engineering

Waterloo, Ontario, Canada, 2010

Author’s Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The design of combustion devices is very important to society today. They need to be highly efficient, while reducing emissions in order to meet strict environmental standards. These devices, however, are currently not being designed effectively. The most common method of improving them is through parametric studies, where the design parameters are altered one at a time to try and find the best operating point. While this method does work, it is not very enlightening as it neglects the non-linear interactions between the design parameters, requires a large amount of time, and does not guarantee that the best operating point is found. As the environmental standards continue to become stricter, a more robust method of optimizing combustion devices will be required.

In this work a robust design optimization algorithm is presented that is capable of mathematically accounting for all of the interactions between the parameters and can find the best operating point of a combustion device. The algorithm uses response surface modeling to model the objective function, thereby reducing computational expense and time as compared to traditional optimization algorithms.

The algorithm is tested on three case studies, with the goal of improving the radiant efficiency of a two stage porous radiant burner. The first case studied was one dimensional and involved adjusting the pore diameter of the second stage of the burner. The second case, also one dimensional, involved altering the second stage porosity. The third, and final, case study required that both of the above parameters be altered to improve the radiant efficiency. All three case studies resulted in statistically significantly changes in the efficiency of the burner.

Acknowledgements

First of all I would like to thank my supervisor, Kyle Daun, for all of his help and guidance throughout my Masters research. Whether using his NSERC grants to fund my research, helping write this thesis and other works, as well as being a friend to talk to when I needed it, he has always been there for me and supported me throughout my work. I am very grateful that he chose me as his first graduate student upon his arrival at the University of Waterloo.

I would also like to thank David Burr and Sina Haji Taheri, my co-workers and fellow graduate students, for their assistance along the way. They would listen to my questions and complaints and offer their opinions and guidance whenever it was needed, as well as provide much needed comic relief in the office (Journey breaks come to mind).

I am most grateful to my friends, my sister Leanne, my brother in-law Ryan, and my girlfriend Candice, for their patience, understanding, and support throughout my Masters research. Many fun adventures had to be put on hold for the last two years due to my research occupying my mind and time.

Lastly, I would like to thank my parents. Without their love and support I would not have been able to complete this journey, and I dedicate my thesis to them.

Table of Contents

List of Figures

Chapter 2

Chapter 3

Chapter 4

Chapter 5

List of Tables

Chapter 3

Chapter 5

Chapter 6

Nomenclature

Variables

A	Reactant
A_0	Pre-exponential Steric Factor
c	Molar Concentration [kmol/m^3]
C	Specific Heat [J/kg K]
d	Search Direction
d_p	Pore Diameter [m]
E_A	Activation Energy [kJ/kmol]
D_{ij}	Binary Diffusion Coefficient [m^2/s]
D_{im}	Diffusion Coefficient into Mixture [m^2/s]
F(**x**)	Objective Function
g	Gradient Vector
h	Enthalpy [J/kg]
h_v	Volumetric Heat Transfer Coefficient [W/m^3 K]
H	Hessian Matrix
k	Reaction Rate Constant
K_c	Equilibrium Constant
n	Number of Species
n_r	Number of Reactions
P	Pressure [atm]
q	Radiative Heat Flux [W/m^2]
R	Universal Gas Constant [m^3 atm/K kmol]
T	Temperature [K]
u	Velocity [m/s]

V	Diffusion Velocity [m/s]
W	Molecular Weight [kg/kmol]
$\overline{W}$	Mixture Molecular Weight [kg/kmol]
$\mathbf{x}$	Design Variables
X	Mole Fraction
Y	Mass Fraction
α	Step Size
β	Arrhenius Temperature Exponent
γ	RSM Point Spacing
ε	Porosity
κ	Extinction Coefficient [1/m]
λ	Thermal Conductivity [W/m K]
μ	Viscosity [kg/m s]
ρ	Density [kg/m^3]
σ	Stefan-Boltzmann Constant [W/m^2K^4]
υ	Stoichiometric Coefficient
φ	Equivalence Ratio
$\dot{\omega}$	Production Rate [$kmol/m^3$ s]
Ω	Scattering Albedo

Subscripts

g	Gas Phase
i	Chemical Species Index
j	Reaction Index
s	Solid Phase

Supersripts

k	Current Value
k+1	New Value
*	Optimal Value
+	Forward Direction
–	Backward Direction

Nondimensional

Nu_v	Volumetric Nusselt Number ($h_v d_p^2/k_g$)
Re_p	Pore Reynolds Number ($\rho_g \varepsilon u d_p/\mu$)

Chapter 1
Introduction

1.1 Motivation

In industrial combustion, the porous radiant burner is a very important development. It allows combustion of fuels with lower heating values compared to other burner types, and also has lower pollutant emissions and higher thermal efficiency. The combustion zone in these burners is located inside a porous, chemically inert, solid matrix. Combustion products heat the solid matrix downstream of the flame zone. Some of this heat is then conducted and radiated back upstream, which preheats the reactants, and enables stable operation at a higher thermal efficiency and lower temperature. Over the past forty years, a large amount of research has been dedicated to further the understanding of porous burners as well as improving the numerical models. As the technology matured, benefits were seen in the use of a two-stage porous ceramic burner. The upstream stage contains smaller pores and acts as a flame arrester, thereby anchoring the flame at the interface of the two porous sections, which further extends the stable range over that obtained from a single-stage porous burner.

To date, attempts at design optimization of these burners have been limited to univariate parametric studies that show how varying one aspect of the burner design affects its overall performance. While somewhat enlightening, a parametric study generally ignores nonlinear interactions between the parameters, and thus the optimal operating point can be missed. A more comprehensive way of looking for the best operating point is to implement design optimization

methodologies. These methods transform the design problem into a mathematical minimization problem by defining a vector of design variables and an objective function, which quantifies the design performance. The minimum of the objective function, representing the optimal design outcome, is then found using a gradient based solver, which repeatedly adjusts the design variables based on the local topography of the objective function. Despite the benefits, however, application of design optimization to industrial combustion has been quite limited to date.

This thesis presents the development of a generic optimization algorithm for use by the combustion community, with the porous radiant burner as a test case. This algorithm will be capable of dealing with the stiffness associated with the governing equations of combustion problems and is capable of finding solutions in a timely manner. The algorithm is a new approach to optimizing these kinds of problems.

The main contributions of this thesis are as follows. A comprehensive model for combustion in porous media, combining all of the most recent correlations and property data will be presented first. Then, an optimization algorithm capable of solving stiff, noisy problems accurately will be given. Finally, the algorithm is demonstrated by carrying out one and two dimensional optimizations involving a porous radiant burner.

1.2 Scope of Problem

The present study is limited to the optimization of a porous radiant burner. While modelling the porous radiant burner, the best available data and correlations available were used, and improving this model was beyond the scope of this research. Optimization is limited to problems of one and two design variables and a single objective, although extensions to a larger number of design variables and more complex objective functions are straightforward.

1.3 Outline of Thesis

This thesis is divided into five main sections; a literature review, a description of the combustion model used, a description of the optimization algorithm used, the implementation of the optimization algorithm and results, and the summary and recommendations for future work.

Chapter 2 presents a review of the literature in two fields: first the development of the porous radiant burner and its model; and second a review of design optimization algorithms found in the literature.

Chapter 3 presents the combustion model used in this research. This includes the governing equations, the boundary and initial conditions, property values and correlations, and concludes with a validation of the model.

The derivation of the optimization algorithm is presented in Chapter 4. Here the basic principles of optimization are laid out followed by a description of Newton's method and Response Surface Modelling. The chapter concludes with a validation of the optimization algorithm on a standard minimization test problem.

Chapter 5 contains three case studies of the optimization algorithm being used on a porous radiant burner. Here the results are given along with physical justifications that explain why the solutions make physical as well as numerical sense.

Finally, Chapter 6 summarises the results obtained in Chapter 5, as well as the benefits of the optimization algorithm. Recommendations for future work involving the developed optimization algorithm are also presented.

Chapter 2
Literature Review

2.1 Introduction

This chapter presents the literature relevant to the current research. The review is divided into two major sections: experimental studies and numerical modeling of porous media; and the development of design optimization for industrial combustion.

In the porous media section, the history of the porous ceramic burner will be discussed, starting with the development of the physical burner and attempts to model the physics. Next, improvements made to both the physical and model burner will be reviewed. Emission studies will be discussed third, followed by a review on research about the physical properties of the burner materials and the effect of those properties on performance.

As the practice of design optimization techniques is well established, the optimization section will be limited to the application of design optimization to industrial combustion.

2.2 Porous Media Studies

Research into porous media combustion was initially motivated by the goal of creating an excess enthalpy flame (Weinberg 1971). Weinberg used thermodynamic arguments to hypothesize that peak temperatures in excess of the adiabatic flame temperature could be achieved by recirculating heat from the combustion products to the reactants. Hardesty and Weinberg (1974) then postulated that the excess enthalpy flame would allow stable combustion of fuels having low energy content, but NO_x concentrations may increase because the peak

temperature of the burner exceeds the adiabatic flame temperature. To validate the claim they constructed a burner, shown schematically in Figure 2.1, containing a counter-flow heat exchanger to provide the incoming reactants with the heat from the products. Measurements on the burner clearly demonstrated that the peak temperature exceeded the adiabatic flame temperature.

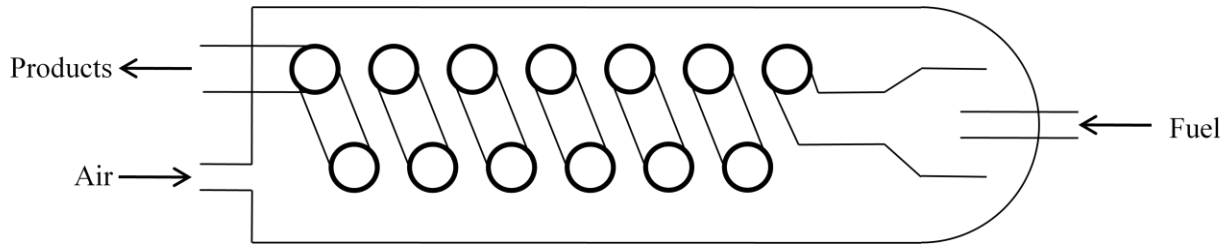


Figure 2.1 - Hardesty and Weinberg (1974) Burner

The main drawback of the Hardesty and Weinberg counter-flow burner was its complexity. Takeno and Sato (1979) hypothesized that heat could be transferred from the products to the reactants in a simpler fashion by placing a highly conductive solid into the flame to conduct the enthalpy of combustion upstream to the reactants. They modelled the problem as a laminar flame interacting with an isothermal solid. Their results demonstrated that increasing the flow rate would also increase the peak temperature as the reaction zone thinned, and unlike the Hardesty and Weinberg burner, the high peak temperature would actually have little effect on NO_x due to short residence times of the gases in the high temperature zone. This model was used for several years to study the fundamental attributes and characteristics of the burner, including the stability of the flame front (Buckmaster and Takeno 1981), the effect of the length of the solid and heat loss (Takeno and Hase 1983), and an experimental study on the stability and location of the flame front (Kotani and Takeno 1982).

A few years later, Echigo et al. (1986) investigated the idea of using a porous media in place of a solid bar for heat recirculation. They modelled the problem as one dimensional with spatially-dependant heat generation in place of detailed reaction kinetics, and scattering was excluded from the radiation equations. They were able to show this simplified model adequately matched experimental data. Experiments showed that low energy content fuels could be burned in this type of burner, and the lean limit of combustion could therefore be extended. They also noted that the porous media was far more effective at transferring the enthalpy of combustion into radiant heat compared to an open flame, leading to improvements in radiation modeling in the burner. Tong et al. (1987) modeled the radiation using absorption and anisotropic scattering, and solved the radiative transfer equation (RTE) using the P-11 spherical harmonics approximation. They showed that radiant output could be increased by increasing the optical thickness of the porous medium, and also by using a strongly backscattering medium.

The research focus then shifted to improving the combustion chemistry model. Hsu and Matthews (1993) showed that a single step chemistry mechanism over-predicted the peak flame temperature by five to twenty percent and also over predicted the burning rate; based on this observation they concluded that detailed chemical kinetics should be used whenever possible, as it provides better numerical results as well as being a more realistic representation of the system. They also showed that for equivalence ratios greater than 0.8, the porous media burner ceases to be an excess enthalpy flame.

The next major improvement to porous burner technology was the addition of a second stage of porous media (Hsu, Howell and Matthews 1993). In this design, the upstage porous section has small pores to allow for greater preheating and to also act as a flame arrester, as the

small pore diameter is comparable to the quenching distance and thus prevents the flame from travelling upstream. The second stage has larger pores to accommodate the submerged flame.

Increasingly strict emissions regulations on industrial combustion devices promoted an increased focus on this aspect of porous ceramic burners. Experimental studies were performed to determine the effect of flame speed and equivalence ratio on the emission levels of pollutants such as NO_x and CO. Khanna et al. (1994) experimented with pre-mixed methane and air and showed that the concentration of NO_x increases with flame speed, due to the increase in peak temperature, while the concentration is relatively constant for a given equivalence ratio. They also showed that CO increases with flame speed, as concentrations are dependent on flame location and at faster speeds the flame is located near the end of the burner and does not have time to oxidize the CO into CO_2.

Further improvements to the numerical model required better characterization of the porous ceramic properties. Younis and Viskanta (1993) performed experiments on alumina having several different porosities and pore sizes to determine the effect these parameters have on the volumetric heat transfer coefficient. They derived several correlations for the volumetric Nusselt number, Nu_v, of the porous media having the form

$$Nu_v = C Re_p^m \tag{2.1}$$

where C and m are constants determined by the pore diameter of the solid and Re_p is pore based Reynolds number. Research was also carried out to determine the values of the conduction and radiation properties. Hsu and Howell (1992) showed through experimentation that thermal conductivity, λ_s, of partially stabilized zirconia (PSZ) was independent of temperature and could be represented as a linear correlation with the pore size, d_p, having the form

$$\lambda_s = 0.188 - 17.5 d_p \tag{2.2}$$

They also presented two correlations for the extinction coefficient of the ceramic that were also independent of temperature. The first correlation used geometric optics and was based on the pore size and the porosity, while the second was based on experimentation and was only a function of pore size. Hendricks and Howell (1996) used experimentation and inverse analysis to determine the spectral absorption and scattering coefficients, as well as the accuracy of different scattering phase functions. It was found that scattering is far more important than absorption in these materials, and that the scattering and absorption coefficients were relatively constant with wavelength. They found the phase functions to be mostly isotropic, although at wavelengths above 2.4μm this was not the case.

The objective of the above research was to develop an efficient porous radiant burner for industrial combustion applications, and to this end several studies were carried out to determine how these burners should be designed. Barra et al. (2003) performed a parametric study to investigate the effect of equivalence ratio, solid conductivity, volumetric heat transfer coefficient, and extinction coefficient on the stable operating range of a two-stage porous ceramic burner. Increasing the equivalence ratio caused the stable operating range to shift to faster velocities as well as widen the range. The remainder of their tests were performed with an equivalence ratio of 0.65. For the solid conductivity four different cases were run with different combinations of solid conductivities in the upstream and downstream section of the burner. The best operating condition occurred when the second stage thermal conductivity was increased by a factor of ten which resulted in the stable range increasing by approximately a factor of two. For the volumetric heat transfer coefficient, three cases were carried out by altering the pore diameter in the Nusselt number correlation. The case where the pore diameter of the second stage was

decreased by a factor of two resulted in the greatest effect on the operating range. The final set of tests was for the radiation extinction coefficient, which involved five cases. Increasing the first stage extinction coefficient by a factor of six yielded the best stable operating range.

More recently, Randrianalisoa et al. (2009) attempted to find the porous radiant burner design attributes that minimized pollutant emissions, such as CO and NO_x, while maximizing radiant power. For this study, a series of experiments were carried out to determine which material would be best in what situations. Two experiments for each material were performed; one at the high and one at the low range of the operating conditions for the desired burner. While no rigorous mathematical optimization was performed, their results are still enlightening. In terms of lowering pollutants, metallic foams, such as FeCrAlY, were found to be best at both operating conditions. In terms of radiant power, on the other hand, Mullite foam was best for the low operating condition, but second worse for the high end, where FeCrAlY was the best choice. No conclusion was made about which material is best overall.

2.3 Optimization in Combustion Studies

Although the studies of Barra et al. (2003) and Randrianalisoa et al. (2009) show the general trends of the porous burner performance with material properties, these univariate parametric studies generally ignore the non-linear interactions that the properties have with each other. This is why design optimization is important; it considers all the variables and their interactions simultaneously, and identifies the combination of variables that provide the best possible solution. As previously stated the use of design optimization is established with in other disciplines and will not be the focus of this literature review. Instead this section will focus on the application of optimization to designing industrial combustion devices, which is rather limited.

One of the first studies using combustion and optimization was carried out by Smith et al. (1990), who used optimization to improve the design of coal gasification combustion while using a comprehensive model for the combustion. The objective was to maximize the cold gas efficiency of the burner by changing the pressure, oxygen/coal ratio, and steam/coal ratio. Two different injector designs were also considered: the first was a standard co-flow jet; while the second had swirl added to the coal stream. Optimization was carried out using response surface modelling (RSM), which is discussed in detail in Section 4.4. The optimal efficiencies were found to be 84.86% for the co-flow burner and 82.74% for the burner with swirl; however, experimental verification was still ongoing and not provided. Smith et al. (1990) also considered performing optimization on the coal gasification burner to maximize burnout, while keeping the NO_x in the flue gas below 200ppm. The variables were the secondary swirl number, the variance of the particle size distribution, and the primary-to-secondary momentum ratio. Using the same techniques as the previous cases a solution to the problem was found, located on the constraint due to the competing nature of NO_x concentration and burnout.

A variant of the RSM method presented in this thesis can also be used for operational optimization purposes of existing devices (Myers, Montgomery and Anderson-Cook 2009). In this method the data is collected experimentally rather than be generated numerically. This means that a physical device must exist for the experiment to be carried out on. As a result the solution obtained does not have an immediately obvious physical explanation, as the governing physics are not used to generate the surfaces. The method presented in this thesis, however, is derived from the equations, meaning that physical insight may be seen during the iterative process. The main difference between these two methods is that the one presented in this thesis

can be used during the design phase of burner construction, while the other method is used to fine-tune and improve existing systems.

Correa and Smith (1998) used design optimization to improve the operation of an ethylene furnace. Their objective was to bring the twelve coil outlet temperatures of the furnace closer to a desired temperature, thus creating a more uniform temperature field. The furnace was divided into two zones, with the coils in each zone receiving the same fuel flow rate, which were the design variables for this study. The furnace was modelled using a steady state, turbulent, incompressible reacting flow code, while the optimization was carried out using a quasi-Newton algorithm. A sixty percent improvement was made to the objective function. The burner was then divided into four zones, splitting the existing zones in half, to try and improve the objective function further, however no significant improvement was made.

Another study involving optimizing combustion devices was carried out on molten carbonate fuel cells (MCFC) (Gemmen 1998). The study involved designing a burner that could combust the excess fuel in MCFC's by injecting air into plug flow reactors (PFR). The objective function was to minimize the amount of hydrogen and carbon monoxide leaving the combustor, while the design variables were the amount of air going into each stage of the purposed combustor. The optimization, however, was carried out heuristically; the flow rate for one of the air injectors was perturbed slightly and then the other injector's flow rates were changed to ensure that the same total amount of air was injected for each iterate design. If the modified design was better than the current one then the modification becomes the new current design. This process was carried out until no significant improvement could be made. The final design reduced the amount of hydrogen and carbon monoxide to less than 1ppm at the burner exit. Although this method did lead to a design improvement, it is not true optimization since

there is no guarantee of optimality; due to the heuristic nature, different answers can be reached depending on which iterates are selected, therefore the true minimum may not be found. Unfortunately, this procedure represents a very common type of "optimization" used today in industrial combustion.

A more rigorous method for optimizing combustion devices are genetic algorithms, which are based on biological processes. Several parent designs are selected from the design space and their objective value, or fitness, is calculated. Based on the principle of survival-of-the-fittest, the designs are then "bred" to produce new designs by mixing the attributes of the parents. Mutation is also introduced during the "breeding" process by randomly perturbing a subset of variables. This process continues until the best design is found. In one study Büche et al. (2001) used a genetic algorithm to minimize the amount of NO_x produced and the amount of pulsation in the burner. The design variables were the fuel flow rates to eight different sections of the burner. Due to the competing nature of the objectives no true minimum was found, but a Pareto front did form where all designs along the front have an equal minimum value for the objective function. The design could then be chosen from this front based on the design needs of the engineer.

Finally, Catalano et al. (2006) used progressive optimization to optimize the design of duct-burners. In progressive optimization, optimization is carried out concurrently with the solution to the combustion problem, so that when the problem is in its early stages of solving the combustion problem the optimization can be quite coarse since the "exact" optimum solution is not needed, while the optimization tolerance is reduced as the solver converges, thus saving computational effort. Catalano et al. (2006) performed two different optimizations: the first was to flatten the outlet temperature profile of the burner while changing the height of the slitform

gap and the crosswise dimension of the flame-holder; and the second was to reduce the near-wall temperatures while altering the same variables as the previous study. Both tests were successful, and the combustion problem was only solved to convergence once, thereby causing a considerable time savings over regular optimization techniques.

2.4 Summary

The development of the porous radiant burner has taken place over the last forty years. The idea began as the insertion of a highly conductive solid into the flame and over time evolved into a multi-staged porous ceramic burner with a submerged flame. Attempts to improve the design of these burners have been limited to parametric studies and trial and error, which ignores the non-linear coupling effects of the equations. The performance of these burners could be improved through design optimization, which is emerging as a design technique in the industrial combustion industry. The objective of this thesis is to develop a multivariable design optimization methodology for porous ceramic burners, which could be extended to other combustion devices.

Chapter 3
The Combustion Model

3.1 Introduction

This chapter presents the combustion model used in this research. This chapter starts with an overview of the porous burner, followed by the governing equations and the necessary boundary and initial conditions. Third, properties and correlations pertaining to the gas phase and solid porous ceramic phase are defined. Forth, the numerical algorithm used to solve the governing equations will then be discussed. Finally, the combustion model will be validated against the work of other researchers.

3.2 Burner Specifications

The burner examined in this study is the same one used by Barra et al. (2003) and Khanna et al. (1994), as well as several other researchers, and is shown schematically in Figure 3.1.

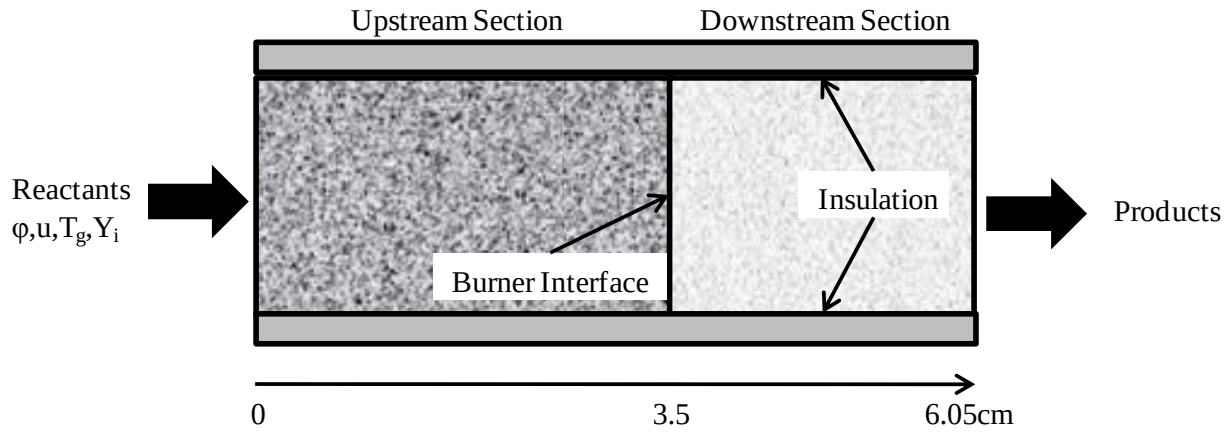


Figure 3.1 - Porous Burner Schematic

The burner is constructed of PSZ and consists of a 3.5cm upstream section and a 2.55cm downstream section; bringing the total length of the burner to 6.05cm. The upstream section contains small pores to act as a flame arrester, while the downstream stage has a large pore diameter for the reasons discussed in Section 2.2. The flame will be submerged within the solid phase and is expected to be located near the interface of the upstream and downstream porous segments.

3.3 The Governing Equations

The combustion model was based on the one presented by Barra et al. (2003), which is a one-dimensional reacting flow that interacts with the solid phase through a volumetric convection coefficient. The model also includes solid and gas phase conduction, solid radiation, species diffusion, and detailed chemical kinetics. Gas phase radiation is unimportant and is excluded from the model due to the small optical path lengths involved, while thermal diffusivity in the gas phase due to diffusion is neglected as it is considered to be negligible (Henneke 1998). Momentum conservation is also excluded from the model as the porous ceramic burner is assumed to be isobaric at one atmosphere. Conservation of mass, gas energy, solid energy, and species, as well as the ideal gas law for a multicomponent mixture give rise to five coupled partial differential equations

$$\frac{\partial(\rho_g \varepsilon)}{\partial \mathrm{t}} + \frac{\partial(\rho_g \varepsilon u)}{\partial \mathrm{x}} = 0 \tag{3.1}$$

$$\rho_{\mathrm{g}} C_{\mathrm{g}} \varepsilon \frac{\partial T_{\mathrm{g}}}{\partial \mathrm{t}} + \rho_{\mathrm{g}} C_{\mathrm{g}} \varepsilon u \frac{\partial T_{\mathrm{g}}}{\partial \mathrm{x}} + \sum_{\mathrm{i}=1}^{n} \rho_{\mathrm{g}} \varepsilon Y_{\mathrm{i}} C_{\mathrm{g,i}} \frac{\partial T_{\mathrm{g}}}{\partial \mathrm{x}} + \varepsilon \sum_{\mathrm{i}=1}^{n} \dot{\omega}_{\mathrm{i}} h_{\mathrm{i}} W_{\mathrm{i}} - \varepsilon \frac{\partial}{\partial \mathrm{x}} \left(\lambda_{\mathrm{g}} \frac{\partial T_{\mathrm{g}}}{\partial \mathrm{x}} \right) + h_v \left(T_{\mathrm{g}} - T_{\mathrm{s}} \right) = 0 \tag{3.2}$$

$$\rho_s C_s \frac{\partial T_s}{\partial \mathrm{t}} \lambda_s \frac{\partial^2 T_s}{\partial \mathrm{x}^2} h_v \left(T_{\mathrm{g}} - T_s \right) + \frac{\mathrm{d}q}{\mathrm{dx}} = 0 \tag{3.3}$$

$$\rho_g \varepsilon \frac{\partial Y_i}{\partial \mathrm{t}} + \rho_g \varepsilon u \frac{\partial Y_i}{\partial \mathrm{x}} + \frac{\partial}{\partial \mathrm{x}} \left(\rho_g \varepsilon Y_i V_i \right) - \varepsilon \dot{\omega}_i W_i = 0 \tag{3.4}$$

$$\rho_g = \frac{P\overline{W}}{\mathrm{R}T_g} \tag{3.5}$$

where ρ is the density, ε is the porosity, u is the velocity, C is the specific heat, T is the temperature, n is the number of species, Y is the mass fraction, $\dot{\omega}$ is the species production rate, h is the enthalpy, W is the molecular weight, λ is the thermal conductivity, h_v is the volumetric heat transfer coefficient, q is the radiative heat flux, V is the diffusion velocity, P is the pressure, R is the universal gas constant, and $\overline{W}$ is the mixture averaged molecular weight, calculated from

$$\overline{W} = \sum_{\mathrm{i}=1}^{\mathrm{n}} W_{\mathrm{i}} X_{\mathrm{i}} \tag{3.6}$$

where X is the mole fraction. Subscripts g, s, and i refer to the gas phase, solid phase and i^{th} species respectively.

The species production rate is calculated from

$$\dot{\omega}_{\mathrm{i}} = W_{\mathrm{i}} \sum_{\mathrm{j}=1}^{\mathrm{n_r}} \left(\upsilon_{\mathrm{ij}}^{-} - \upsilon_{\mathrm{ij}}^{+}\right) k_{\mathrm{j}} \times \left(\prod_{\text{reactants}} \mathrm{c}^{\upsilon_{\mathrm{ij}}^{+}} - \frac{1}{K_C} \prod_{\text{products}} \mathrm{c}^{\upsilon_{\mathrm{ij}}^{-}} \right) \tag{3.7}$$

where $\mathrm{n_r}$ is the number of reactions, k_{j} is the reaction rate constant, K_c is the equilibrium constant, and c, υ_{ij}^{+}, and υ_{ij}^{-} are the molar concentrations, and the stoichiometric coefficients for the forward and backward direction of the reactant $\mathrm{A_i}$ in the j^{th} chemical reaction of the form

$$\sum_{\mathrm{i}=1}^{\mathrm{n}} \upsilon_{\mathrm{ij}}^{+} \mathrm{A_k} \Leftrightarrow \sum_{\mathrm{i}=1}^{\mathrm{n}} \upsilon_{\mathrm{ij}}^{-} \mathrm{A_k} \tag{3.8}$$

The reaction rate constant for the j^{th} reaction is calculated from the modified Arrhenius expression

$$k_j = A_{0,j} T_g^{\beta_j} \exp\left(-\frac{E_{A,j}}{R T_g}\right) \tag{3.9}$$

where $A_{0,j}$ is the pre-exponential steric factor, β_j is the temperature exponent, and $E_{A,j}$ is the activation energy for the j^{th} chemical reaction.

The diffusion velocity, which represents the speed at which the species are diffusing in the gaseous mixture, is calculated from

$$V_i = -D_{im} \frac{1}{X_i} \frac{\partial X_i}{\partial x} \tag{3.10}$$

with

$$D_{im} = \frac{1 - Y_i}{\sum \frac{X_j}{D_{ij}}} \tag{3.11}$$

where D_{im} is the diffusion coefficient of the i^{th} species into the mixture and D_{ij} are the binary diffusion coefficients.

There are many methods available to estimate the radiant source term, which is the spatial derivative of the radiant heat flux found in Eqn. (3.3). Due to the highly isotropic nature of the radiant intensity in porous ceramics the Schuster-Schwarzchild (S_2) technique (Siegel and Howell 2002) was used. The S_2 method models the radiant intensity as isotropic in the forward and backward directions, and by multiplying by π we can easily convert the system of ODEs for intensity into

$$\frac{dq^+}{dx} = -\kappa(2-\Omega)q^+ + \kappa\Omega q^- + 2\kappa(1-\Omega)\sigma T_s^4 \tag{3.12}$$

$$-\frac{dq^-}{dx} = \kappa\Omega q^+ - \kappa(2-\Omega)q^- + 2\kappa(1-\Omega)\sigma T_s^4 \tag{3.13}$$

where q^+ and q^- represent the radiant heat flux in the forward and backward direction, κ is the extinction coefficient, Ω is the scattering albedo, which is the ratio of the scattering coefficient to the extinction coefficient, and σ is the Stefan-Boltzmann constant. Once the system is solved for q^+ and q^- the solutions can be used to find the radiant source term in the equation

$$\frac{dq}{dx} = 4\kappa(1-\Omega)\left(\sigma T_s^4 - \frac{q^+ + q^-}{2}\right) \tag{3.14}$$

The S_4 method (Siegel and Howell 2002), which assumes four directions of isotropic intensity, was tested as well but provided minimal improvement in accuracy compared to its increased computational expense.

The ensemble of coupled partial differential equations in Eqns. (3.1)-(3.5) is stiff due to the high degree of coupling between the equations and the fact that the variables vary over substantially different time scales. Coupling, caused by the presence of the same variables in multiple equations, can be seen in Eqns. (3.1)-(3.5) which represent the system, but can also be seen carrying over into Eqns. (3.6)-(3.14) which determine the properties. Different time scales also exist, primarily due to the chemical reaction mechanisms, where reactions occur very quickly and at different rates. As such the solver must proceed very slowly, and be spatially refined, in order to capture the details of the changing chemical species. The time scale of the temperatures is determined by diffusion, therefore the temperature profiles change much slower than the species mass fractions. If the solver were to run at the time scale of the temperatures then the solver would be inaccurate due to the missing detail of the chemical species. However, if the solver is allowed to proceed at the time scale of the chemical reactions then round-off errors can be introduced into the system, due to large number of unnecessary steps being taken

for the slower changing variables. A more detailed description of stiffness can be found in Garfinkel et al. (1977) and Section 4.3 of this thesis. Due to the stiffness a special solver must be employed to solve the governing equations which will be discussed in Section 3.6.

3.4 Boundary and Initial Conditions

The governing equations (Eqns. (3.1)-(3.4), (3.12), and (3.13)) require boundary conditions to close the system. For conservation of mass, Eqn. (3.1), a fixed inlet velocity is selected as the inlet condition, which is constant at 0.45m/s for all cases studied. Conservation of energy in the gas phase, Eqn. (3.2), requires two boundary conditions to close the problem. The first is a fixed inlet gas temperature, which is set to 300K for all calculations. The second is a zero gradient exit gas temperature, which ensures that equilibrium is reached within the burner. Conservation of energy in the solid phase, Eqn. (3.3), also requires two boundary conditions, which are selected as zero gradient solid temperature at the inlet and the outlet of the burner. A zero gradient boundary condition was selected as the solid stops at the burner inlet and outlet and cannot conduct heat any further. Although in some studies a modified Robin boundary condition is used to account for radiant emission, this effect is instead accounted for through the S_2 equations, Eqns. (3.12) and (3.13). Although the conservation of species equation, Eqn. (3.4), only requires one boundary condition per species, two were used instead. The first boundary condition is set mass fractions for each species; in this case an equivalence ratio (φ) of 0.65 was used for all calculations, where air was assumed to have a composition of 21% oxygen, 78% nitrogen, and 1% argon; all other species had an inlet mass fraction of zero. The second boundary condition was that all species mass fractions would have zero gradients at the burner exit, which is not necessary to solve the problem but ensures that equilibrium is reached within the burner. The S_2 equations, Eqns. (3.12) and (3.13), requires one boundary condition each,

which are selected as radiating to a blackbody at 300K from the inlet for Eqn. (3.12) and from the outlet for Eqn. (3.13). Table 3.1 contains a summary of the boundary conditions listed above.

Table 3.1 - Summary of Boundary Conditions

Equation	Inlet Condition	Outlet Condition
Conservation of Mass, (3.1)	u=0.45m/s	N/A
Conservation of Energy, Gas, (3.2)	T_g=300K	$\frac{dT_g}{dx}=0$
Conservation of Energy, Solid, (3.3)	$\frac{dT_s}{dx}=0$	$\frac{dT_s}{dx}=0$
Conservation of Species, (3.4)	$Y_i=Y_{i,0}$ where $Y_{i,0}$ is determined from φ	$\frac{dY_i}{dx}=0$
S_2 Equation, (3.12)	$q^+=\sigma T_s^4$	N/A
S_2 Equation, (3.13)	N/A	$q^-=\sigma T_s^4$

Due to the pseudo-transient solution scheme, explained in Section 3.6, Eqns. (3.1)-(3.4) also require initial conditions. For conservation of mass, Eqn. (3.1), a velocity profile was required. The profile, shown in Figure 3.2, linearly increases from the known inlet value to another value at the burner exit, which is determined by solving the ideal gas law, Eqn. (3.5), for the density using the adiabatic flame temperature and then using conservation of mass to determine the velocity.

Conservation of energy in the gas phase, Eqn. (3.2), requires an initial profile for the gas temperature. This profile, shown in Figure 3.3, starts at the inlet value and remains at this temperature until just upstream of the burner interface, at which point it begins to linearly increase. The profile stops increasing at a value of 2000K just downstream of the interface where it begins to linearly decrease to the adiabatic flame temperature at the burner exit. The

reason for the large spike in temperature at the burner interface is to promote formation of the flame front close to where it will likely stabilize.

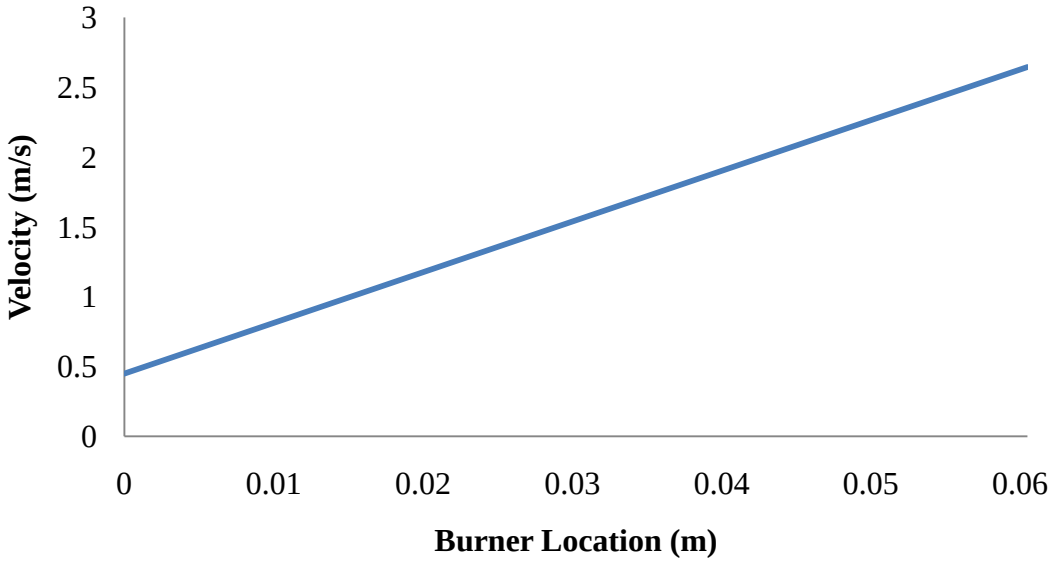


Figure 3.2 - Velocity Initial Condition

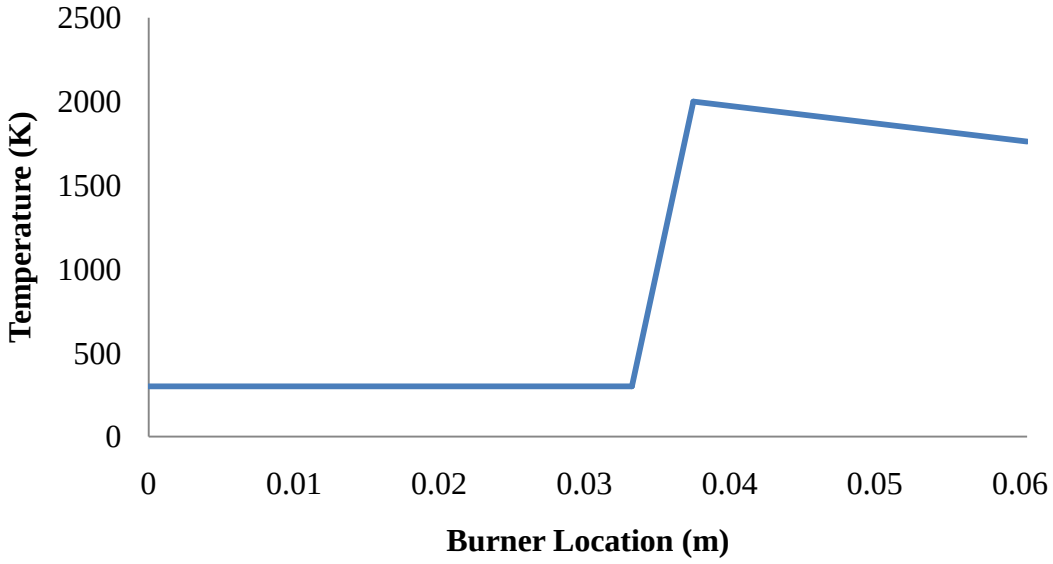


Figure 3.3 - Gas Temperature Initial Condition

The initial condition for conservation of energy in the solid phase, Eqn. (3.3), a solid temperature profile, is not set in advance but is instead calculated by the solver. Here, the radiant

source term is assumed to be zero and using the other initial conditions Eqn. (3.3) is solved to find the initial profile for the temperature of the solid. Conservation of species, Eqn. (3.4), requires initial guesses for each of the chemical species. These profiles start from their specified known inlet condition and linearly change to their equilibrium value at the burner interface, where they remain until the burner exit. The oxygen profile is provided as an example in Figure 3.4.

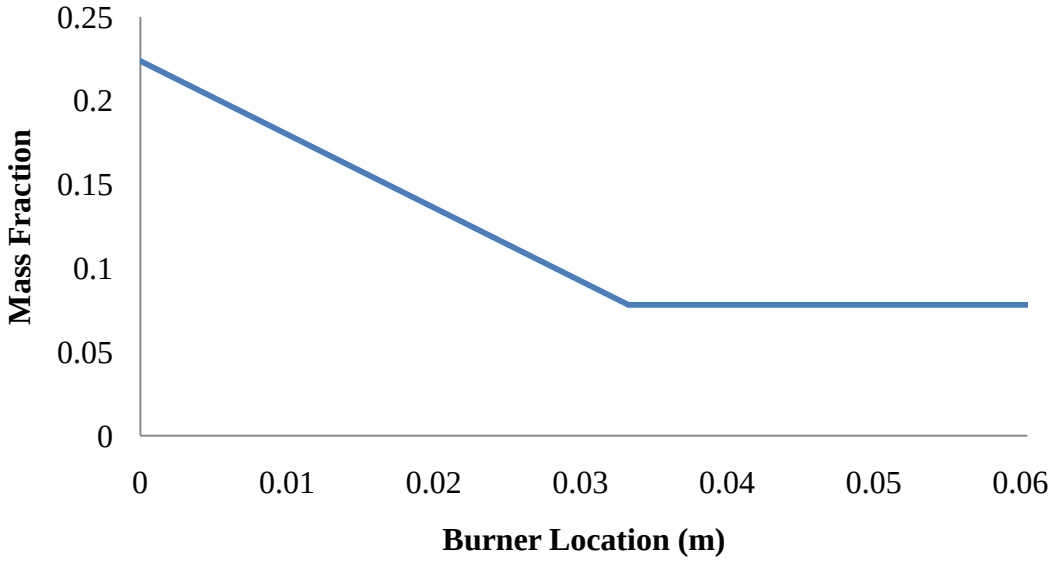


Figure 3.4 – Oxygen (O_2) Initial Condition

3.5 Properties and Correlations

This section will discuss the physical properties and correlations that pertain to the gas and solid porous ceramic phase.

3.5.1 Gas Phase Transport Properties and Reaction Kinetics

The transport properties of the gas are specified as part of the reaction mechanism and are calculated internally by the solver. The mechanism used in this research is DRM19 (Krazakov and Frenklach 1994), which is a methane combustion mechanism with 21 species participating in

84 reactions (See Appendix A for a detailed listing of this mechanism). DRM19 is a reduced mechanism based on GRI-Mech 1.2 (Frenklach, et al. 1994), and thus must use the thermodynamic and transport files associated with it as well (thermo12.dat and transport.dat respectively). A reduced mechanism reduces the computational time of the solver compared to a more detailed mechanism like GRI-Mech 3.0 (Smith, et al. 1999) but provides better accuracy compared to a global mechanism. A reduced mechanism also helps to reduce the stiffness of the system by reducing the extent of coupling between the equations and by reducing the number of different time scales, a result of having fewer variables. To verify the validity of this reaction mechanism a comparison was made to GRI-Mech 3.0 (Smith, et al. 1999), which is currently considered the most complete and accurate methane combustion mechanism. First, a solution was found using the GRI-Mech 3.0 mechanism, setting the grid refinement parameters to ensure an accurate solution and completion within a reasonable computational time. Then, a second solution was found using the DRM19 mechanism, using the same grid refinement parameters. Although this solution time was considerably faster, the solution did not match that of GRI-Mech 3.0. The refinement parameters were then increased, which increased the computational time of DRM19 but still kept it below that required by the GRI3.0 solution. This resulted in a much better agreement between the two reaction mechanisms, seen in the gas temperature profiles in Figure 3.5. The two profiles are a very close match, and deviate by a maximum of 5% at the peak temperature, where DRM19 under predicts the GRI-Mech 3.0 results by about 25K. As the peak temperatures are around 1850K and moreover the location of the peak temperature is consistent between the two models DRM19 can be considered a suitable mechanism for use in this study.

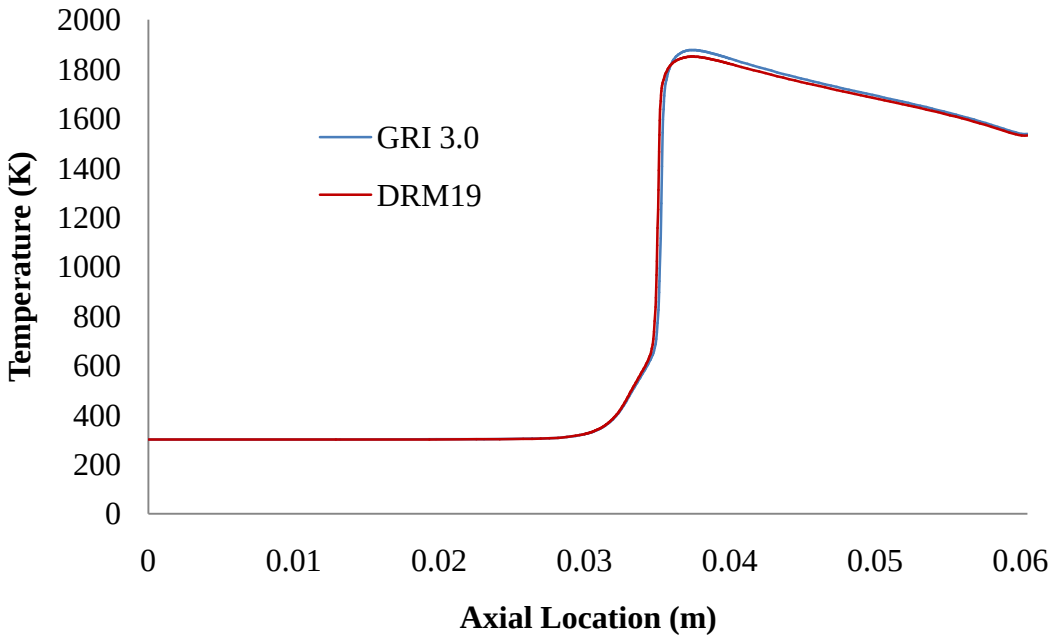


Figure 3.5 – Gas Temperature Profile Comparison using DRM19 and GRI 3.0

3.5.2 Solid Phase Properties

The nominal properties for the solid phase are given in Table 3.2 and come from the research of Khanna et al. (1994) and Barra et al. (2003). These properties are for the reference case used for validation, and also as the initial point during the optimization phase of the research.

Table 3.2 - Burner Property Data

Property	Upstream	Downstream
Pore Density	25.6ppc	3.9ppc
Pore Diameter, d_p	0.029cm	0.152cm
Porosity, ε	0.835	0.87
Scattering Albedo, Ω	0.8	0.8
Density, ρ_s	510kg/m^3	510kg/m^3
Specific Heat, C_s	824J/kgK	824J/kgK

To avoid any numerical difficulties, the pore diameter and the porosity are linearly blended over a 4mm span surrounding the burner interface, which prevents any discontinuities that could lead to failures in the numerical algorithm.

3.5.3 Solid Phase Correlations

The first solid phase correlation to be discussed is for calculating the volumetric Nusselt number, which provides the volumetric heat transfer coefficient linking the gas and solid conservation of energy equations. The correlation was proposed by Younis and Viskanta (1993) and has the form

$$Nu_v = C Re_p^m \tag{2.1}$$

This correlation was determined by carrying out experiments on alumina foams, but is used here as no data exists for PSZ foams and is used by other researchers for studies involving PSZ, e.g. Barra et al. (2003). As the Nusselt number is entirely dependent upon the gaseous properties and the pore structure, and is independent of the solid properties, we are justified in using the correlation regardless of the solid phase material. In their experiment, Younis and Viskanta (1993) placed a porous ceramic at a specified uniform temperature into a stream of gas at a different temperature. The transient variation in ceramic temperature, gas temperature and velocity was monitored; this data in turn was used to solve for the volumetric heat transfer coefficient which was then expressed in dimensionless form as the Nusselt number. A least squares fit of the data was used to form Eqn. (2.1), the results of which can be found in Table 3.3.

Table 3.3 - C and m values from Younis and Viskanta (1993)

Pore Diameter (mm)	C	m
0.29	0.638	0.42
0.42	0.485	0.55
0.76	0.456	0.70
0.94	0.139	0.92
1.52	0.146	0.96

Since the optimization study will adapt the pore diameter in a continuous manner. we must develop correlations that can be applied over a wide range of d_p. In fact, Younis and Viskanta (1993) developed their correlations with optimization studies in mind. A linear fit with the pore diameter in meters was selected, and resulted in

$$C = -400 d_p + 0.687 \tag{3.15}$$

$$m = 443.7 d_p + 0.361 \tag{3.16}$$

The inset of Figure 3.6 shows that linear fits are reasonable for the values of C and m. To validate these correlations for combustion purposes two tests were performed. We first plot the volumetric heat transfer coefficient versus the pore diameter, shown in Figure 3.6, to see if the correct trends were observed. The volumetric heat transfer coefficient decreases with increasing pore size, which makes physical sense as an increase in pore size would decrease the number of pores per millimetre, thereby decreasing the surface area per unit volume and leading to a decrease in the amount of convective heat transfer.

The second test of Eqns. (3.15) and (3.16) was to compare the results of two combustion simulations; the first simulation used the values from Table 3.3 that corresponded to each section of the porous media; while the second simulation was run using Eqns. (3.15) and (3.16). Figure 3.7 shows the gas temperature profile found by each simulation.

It can be seen that the two solutions are very similar, being within 5% at all times outside of the region contained within the horizontal bars. Inside this region the solutions differ by as much as 25%Due to the steep temperature gradient at the flame front, however, this much error is not unexpected. Overall the temperatures between these two cases agree very well, so Eqns. (3.15) and (3.16) are acceptable for the purposes of optimization.

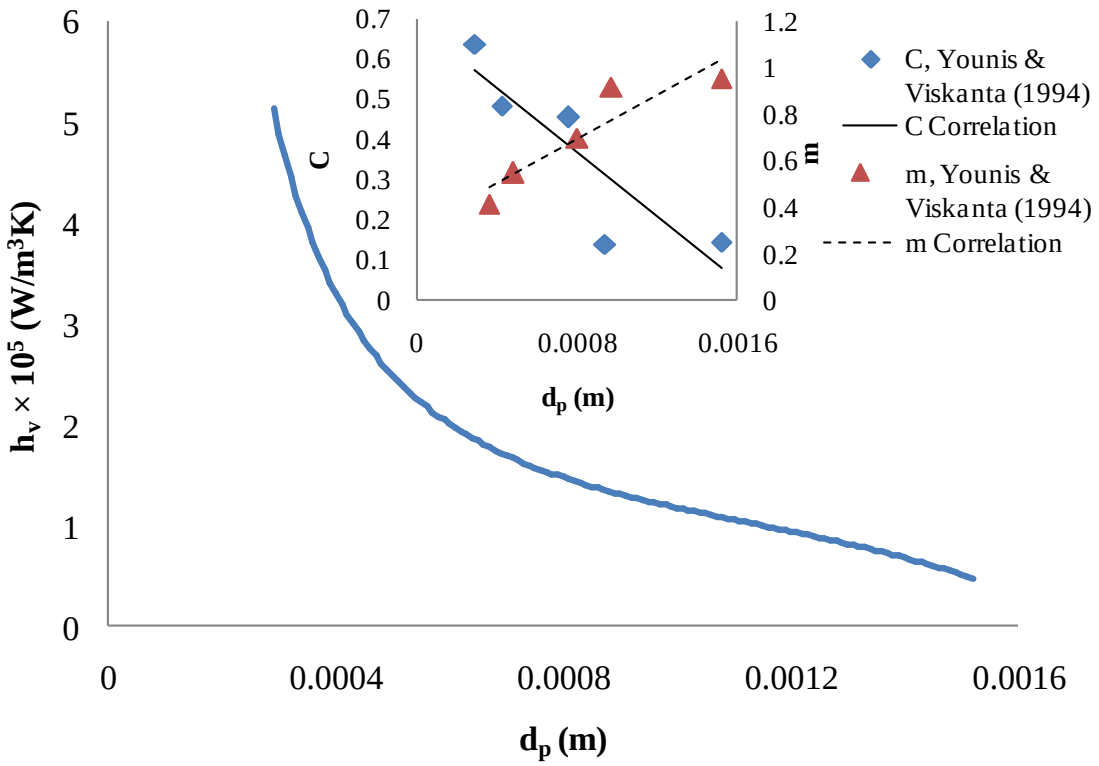


Figure 3.6 - Volumetric Heat Transfer Coefficient Using Correlated Values of C and m

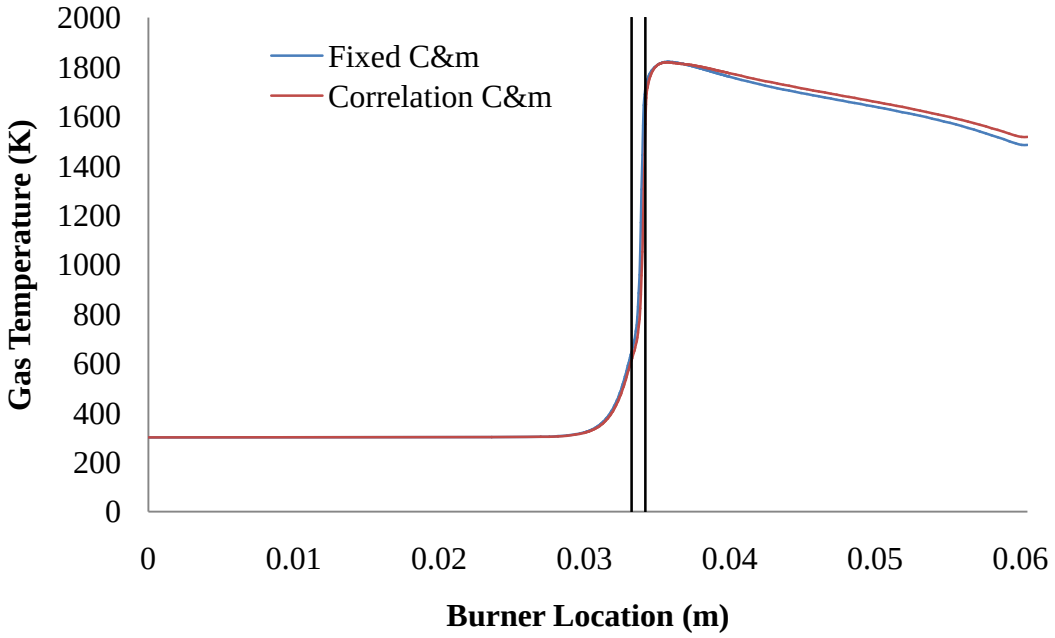


Figure 3.7 – Gas Temperature Comparison using Fixed and Correlated Values for C and m

Two other correlations were used, both proposed by Hsu and Howell (1992). The first is a correlation is for the thermal conductivity,

$$\lambda_s = 0.188 - 17.5 d_p \tag{2.2}$$

which is a linear fit to experimental data based on the pore diameter in metres. In their experiment several pieces of ceramic foams with a variety of pore diameters and a few different porosities were heated on one face using a hot plate and cooled on the other. Using Fourier's law, the thermal conductivity of the specimen could be estimated by replacing the derivative in the law with the difference in the temperatures of the faces divided by the length of the burner. A least squares regression was then used to fit a curve to the data for different pore diameters. A trend was somewhat evident with porosity as well, with the thermal conductivity decreasing with increasing porosity, which is expected since the solid phase has a higher thermal conductivity than the gas phase. Nevertheless, due to the narrow range of porosities tested this was not considered in their conclusions. A correlation of thermal conductivity in terms of the porosity by Nait-Ali et al. (2007)was also considered, but their experiments were carried out on ceramics having porosities between 50-75%, well below the porosities of the ceramic burner examined in this study. Extrapolating this correlation to the porosities of interest, the correlation of Nait-Ali et al. (2007) greatly under-predicts the value reported by Hsu and Howell (1992) as well as being very flat for porosities that are of interest to this study. Howell, Hall, and Ellzey (1996) as well as Vafai (2005) cite the correlation proposed by Hsu and Howell (1992) as the one to use when calculating the thermal conductivity of a porous ceramic, meaning the scientific community has accepted that the thermal conductivity is independent of porosity in the region of interest. To further prove the validity of the correlation used by Hsu and Howell (1992) a sensitivity analysis was carried out to ensure that the porosity had minimal effect on the objective function when

acting through the thermal conductivity. This analysis, found in Appendix B, shows that the objective function is not sensitive to the porosity, so the use of the correlation is valid. Accordingly the correlation of Hsu and Howell (1992) is used in this research as it was obtained based on material properties similar to those used in this research as well as being highly recommended by the scientific community.

The second correlation is for the radiative extinction coefficient

$$\kappa = \frac{3}{d_p}(1-\varepsilon) \tag{3.17}$$

where κ is the extinction coefficient, ε is the porosity, and d_p is the pore diameter in millimetres. Equation (3.17), which is based on geometric optics, was validated for use by Hsu and Howell (1992) for calculating the extinction coefficient for pore diameters larger than 0.6mm; however we extrapolate it to the smaller pore diameters used here, as others have done when modelling porous ceramics under the same operating conditions (Barra et al., 2003). The dominant source of extinction in a porous ceramic is scattering, hence the relatively large value of 0.8 for the scattering albedo. This scattering is caused by the change in index of refraction as the radiation propagates across grain boundaries and different pore structures.

To test the validity of the geometrics optics model we must know whether the porous ceramic solid is an independent or dependent scattering medium, which is done by examining the map of the scattering regimes, shown in Figure 3.8. If a medium is in the dependent scattering region then the participating particles are close enough together that constructive and deconstructive interference can occur to the emitted radiation from each particle, meaning that geometric optics do not apply. Independent scattering, however, means that the participating

particles are separated enough that they can be treated as isolated radiating bodies, as no interference occurs with radiation from other particles. Using the web spacing for the scattering particle diameter, which is assumed to be the same size as the pore diameter based on visual observation (Hsu and Howell 1992), the particle size parameter is on the order of magnitude of 10^3. The particle volume fraction, equal to one minus the porosity, is about 10^{-1}. Using Figure 3.8 we see that we are well within the independent scattering region, so geometric optics can be applied. We also see that any reasonable variation to pore diameter will still result in being within the independent scattering region.

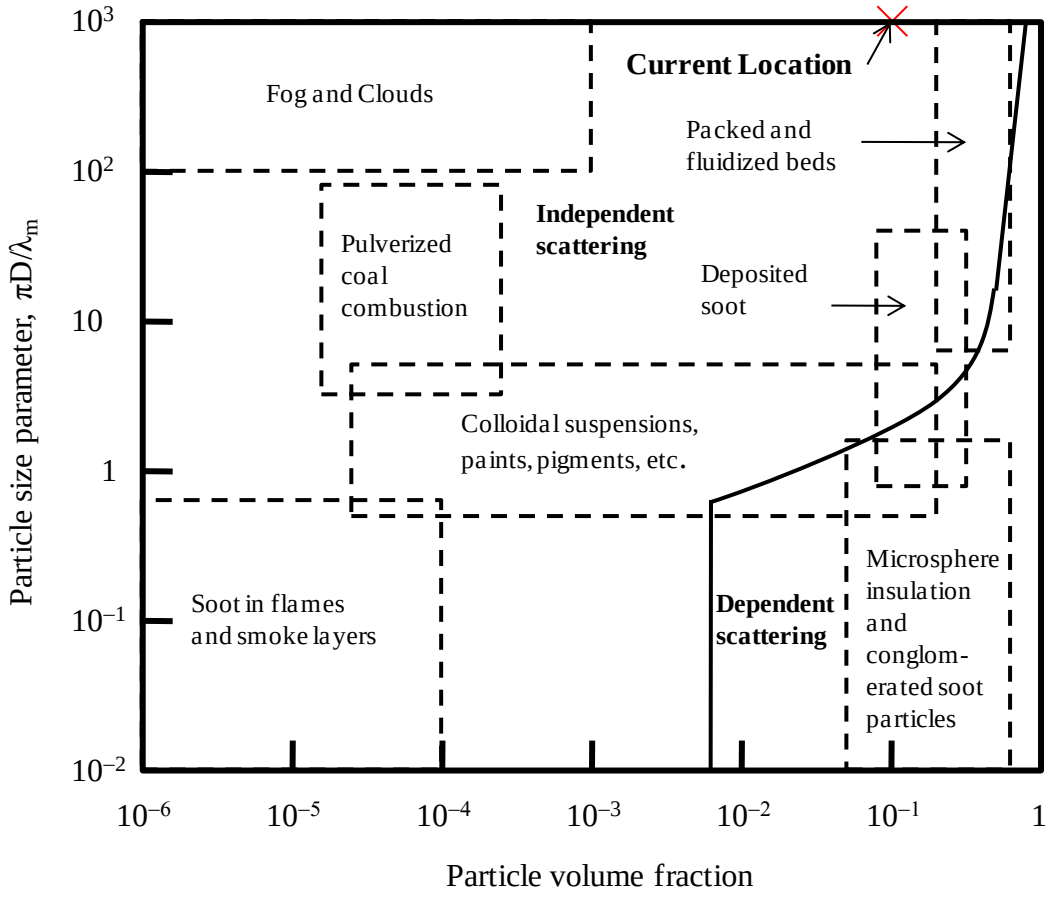


Figure 3.8 - Independent and dependent scattering regimes (Siegel and Howell 2002)

The above properties and correlations completely define the system of coupled differential equations, boundary conditions, and initial conditions, which must be solved numerically.

3.6 Solution Method

The software package Cantera (Goodwin 2003) was used to perform the combustion simulations in this research. Cantera is an open-source chemical kinetics program with a built-in one dimensional, reacting flow solver. As the solver was written for a free flame, the source code had to be adapted to account for the solid phase. The modified files can be found in Appendix C, while the code for interfacing with Cantera, as well as instruction for compiling and using Cantera, can be found in Appendix D. For these simulations the solver was used to simulate combustion at steady state, which it does in a semi-transient manner as described below and shown in Figure 3.9.

The first step in the solution procedure is creating the mesh. As noted in Section 3.3, the physical phenomena occur over a wide range of length scales: the characteristic length for the transport phenomena (heat and mass transfer) is the pore diameter; while the combustion reactions occur over a much smaller distance/time. Accordingly, for reasons of numerical stability and computational expediency, it is necessary to use an adaptive mesh that concentrates grid points where they are needed. The initial mesh contains 300 nodes, 200 of which are tightly clustered around the burner interface, as this is near where the flame is expected to be located and where the highest degree of refinement is required. The four adaption parameters: ratio, slope, curve, and prune are also set. These values were determined empirically to create an accurate yet computationally inexpensive solution. The ratio parameter, which is set to four, ensures that the ratio of the distances between two consecutive nodes and the next consecutive

nodes never exceeds this value by inserting a new node between the two existing nodes to reduce the ratio as necessary. The slope parameter, set to 0.4, ensures that value of a variable at a node and the next node never differs by more than the set percentage, again by adding a new point between the existing nodes as necessary. The curve parameter, set to 0.4, operates similar to the slope parameter, but it ensures that the slope never differs by more than the set percentage. Lastly, prune, set to 0.001, removes nodes from the mesh if the percent change in value or slope at a node goes below this value, ensuring that the mesh does not become cluttered with unneeded nodes. These adaption parameters are summarized in Table 3.4.

Table 3.4 - Grid Refinement Parameters

Parameter	Value
Ratio	4
Slope	0.4
Curve	0.4
Prune	0.001

The solution proceeds in two stages. In the initial stage grid adaption is not used and the energy equations, Eqns. (3.2) and (3.3), are not solved. Instead, the gas and solid temperatures are held at the initial guess values while the velocity and species mass fractions are allowed to change. This step creates a more realistic initial condition for velocity and species mass fractions for the chemical kinetics equations rather than relying solely on the guess profiles in Section 3.4. The solver attempts to solve the equations at steady state by minimizing the residual norm through Newton-minimizations. (A detailed description of Newton's method can be found in Section 4.3.) This process is repeated to reduce the residuals of the governing equations until one of two things happens: the solution is found where the residuals meet the tolerances of the solver; or the solution begins to diverge. If a solution is found the solver proceeds to the second step of the process.

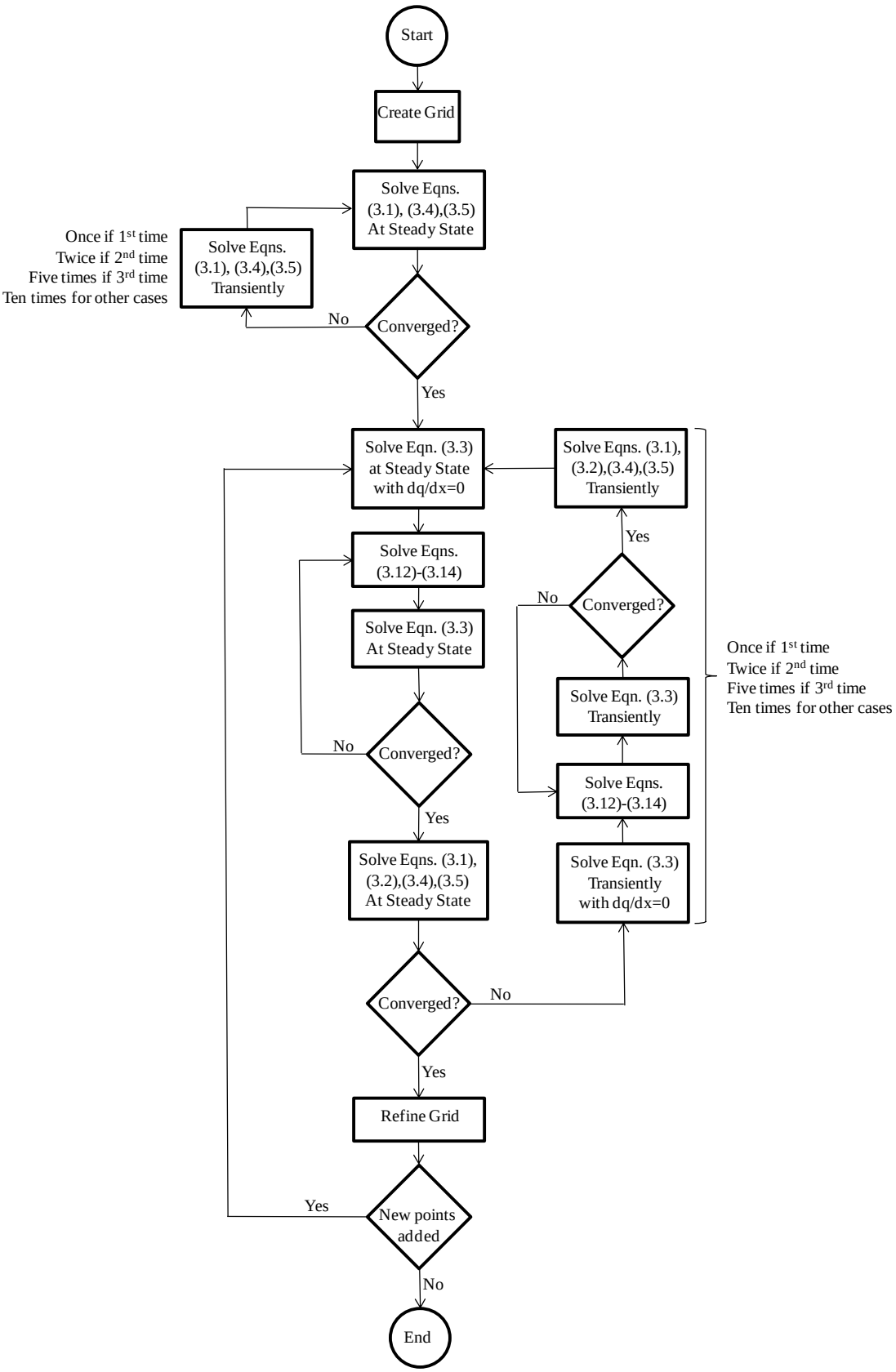


Figure 3.9 - Combustion Solver Flow Chart

On the other hand, if the solution appears to be diverging, Cantera switches from trying to solve the steady state problem to a transient problem. Physically, by switching to a transient solution the profiles will be allowed to temporally evolve, allowing for the profiles to approach the true steady state solution. Mathematically, including transient terms increases the diagonal dominance of the matrix and mitigates the stiffness of the governing equations. At first the solver only takes one step forward in time and then it reverts to solving the steady state problem, the reason being that transient calculations are comparatively slow and should be avoided as much as possible. If the steady state calculation fails again, then two transient steps are made. After a third failure, five transient steps are taken followed by ten steps after a fourth failure. Finally, if the steady state solution fails a fifth time, then ten transient steps are made, which is the maximum number of transient steps that can be taken before attempting another steady state solution. This process of attempting to solve the steady state problem, followed by ten transient steps, is repeated until the steady state solution is found, and the solver proceeds to the second stage. By taking the transient steps the solver operates on the time scale of the chemical reactions. By only taking a few steps at a time, the chances of errors being introduced into the system are reduced. By switching to steady state, the solver is operating at the time scale of the temperature profiles and thus can proceed faster. Since the solution at steady state can only be found when close to the steady state answer, the chemical reaction data is not lost by taking larger steps.

For the second stage of the solution process all of the governing equations, Eqns. (3.1)-(3.5), are solved, and grid refinement is activated. The algorithm begins by solving conservation of energy in the solid phase, Eqn. (3.3), using the previous solution to the gas temperature profile and assuming the radiant source term is zero. Next, the S_2 equations, Eqns. (3.12)-(3.14), are

solved for the radiant source term using the solid temperature profile that was just found. The algorithm then returns to Eqn. (3.3), now using the newly calculated radiant source term. This process continues until the solid temperature profile converges. The solver then tries to solve the remaining governing equations, Eqns. (3.1)-(3.2) and (3.4)-(3.5), at steady state. As is done in the first stage, if the solver fails then transient steps are taken to improve the solution. Before each transient step the solid temperature profile is updated with the new gas temperature profile. Once again, the solver alternates between trying to solve the steady state problem and taking transient steps in order to approach the steady state solution. In this stage, however, when the steady state solver is successful the grid is refined by sweeping through all of the nodes and checking the four refinement parameters. If new nodes are inserted into the domain then the second stage begins anew, the solution to the steady state problem is attempted followed by transient steps if the method fails. This process of adding nodes and resolving the problem continues until no new nodes are needed. When this occurs then the problem has been fully and accurately solved at steady state and the solver stops.

3.7 Verification of the Combustion Model

The solver is validated by comparison to the experimental work of Khanna et al. (1994) and the numerical work of Barra et al. (2003), carried out under similar conditions.

The objective of Khanna et al. (1994) was to determine the effect of flame speed and equivalence ratio on the emissions released from a porous ceramic burner and the exit temperature. Since the current research was to focus on a single equivalence ratio, 0.65, only the data pertaining to this ratio from Khanna et al. (1994) is used in the comparison. The first comparison made was the exit temperature shown in Figure 3.10.

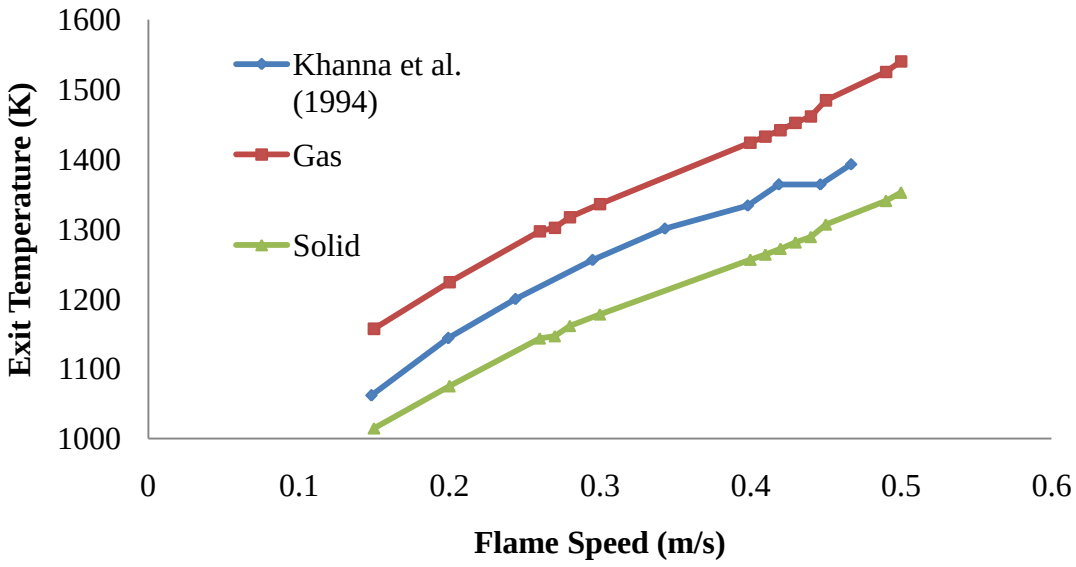


Figure 3.10 – Comparison to Experimental Data for Burner Exit Temperature

Khanna et al. (1994) do not specify whether the exit temperature shown is for the gas phase, the solid phase, or a combination of both. As the Figure 3.10 shows, the numerical temperature profiles have a very good agreement with the experimental profile, with both the solid and the gas temperatures being within 10% of the published data at all times. If the data from Khanna et al. (1994) represents an average of the gas and solid phase temperatures, then agreement between their data and that of the present study is within 3%. The next area for comparison was the CO emission from the burner, shown in Figure 3.11. Once again, a very good comparison can be seen between the two sets of data, differing by a maximum of 3ppm at the fastest flame speed. Khanna et al. (1994) believed the first data point to be the result of low temperatures, resulting in less oxidation, however, we did not observe this in the current model and therefore believe it is the result of experimental error and treat it as an outlier.

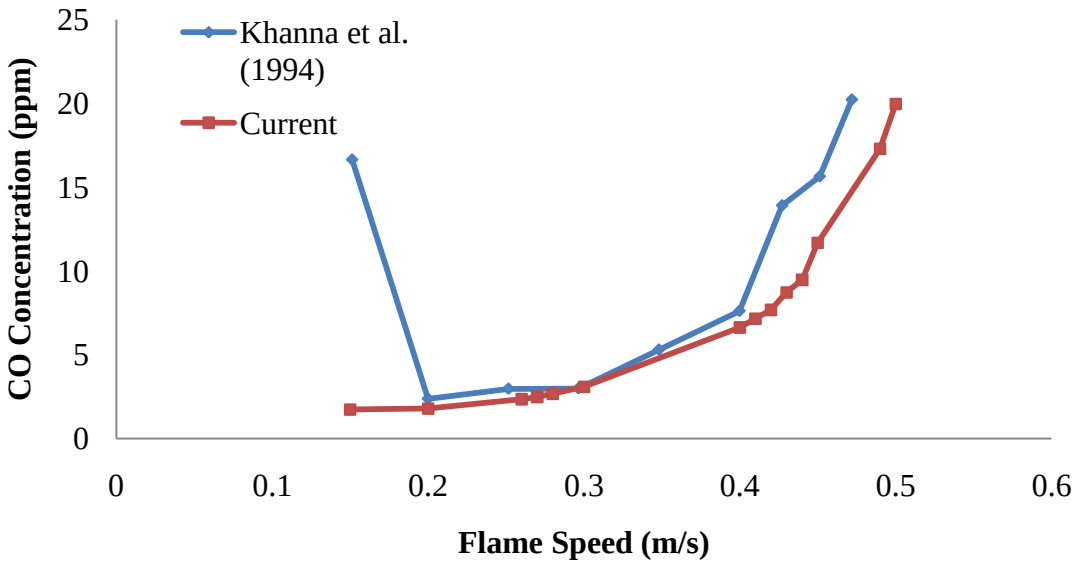


Figure 3.11 – Comparison to Experimental Data for CO Concentration at the Burner Exit

The final comparison made is between NO_x concentrations at the burner exit, shown in Figure 3.12. This figure shows a fairly good comparison between the current model and the experimental research of Khanna et al. (1994). Since predicting NO_x formation is notoriously difficult, and because a reduced chemistry model was used, we do not expect these two curves to compare as well as the previous comparisons. Here, the average difference between the two curves is around 50%, which again is quite reasonable given the current state in modelling NO_x formation. Unfortunately, this level of agreement and the current state of NO_x modelling is inadequate for conducting a multiobjective study with the goal of finding a good trade-off between NO_x formation and fuel efficiency. As all three comparisons with the work of Khanna et al. (1994) are good, the model seems to be an accurate representation of combustion in porous media.

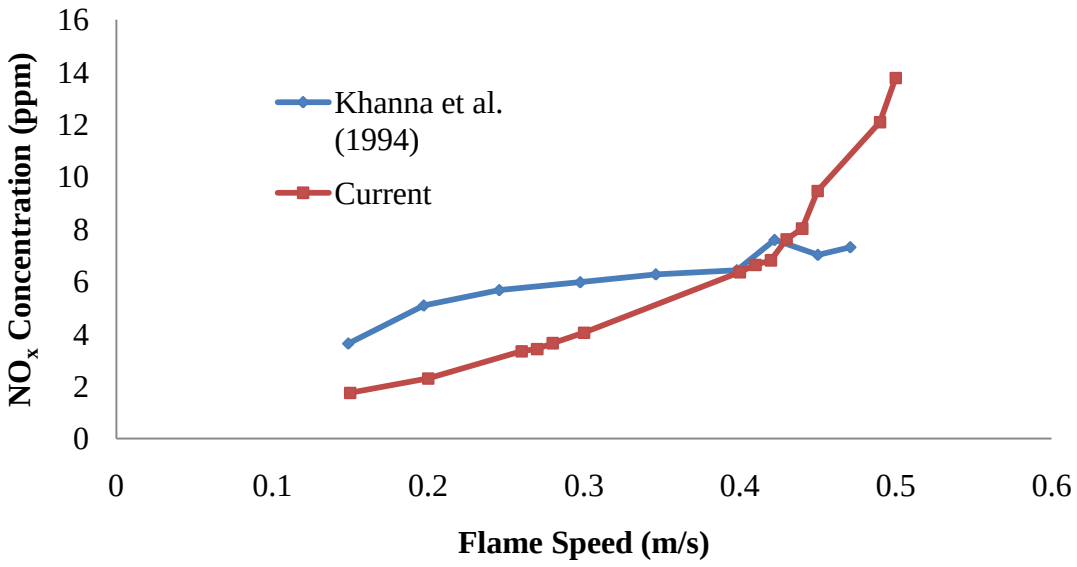


Figure 3.12 – Comparison to Experimental Data for NO_x Concentration at the Burner Exit

To further verify the model, comparisons were also made with the simulations of Barra et al. (2003). They studied the effect of changing certain parameters on the stable operating range of the burner, defined as the difference between the velocities at which blow-off and flash-back occur. The ratio of maximum and minimum velocity is called the turn-down ratio, and is considered an important attribute of the burner since it allows for operational flexibility. Flash-back is numerically difficult to simulate, so Barra et al. (2003) defined it as occurring when the peak temperature occurs just upstream of the burner interface. The stable operating range for the reference case, found numerically by Barra et al. (2003), was between 48–74cm/s, or a range of 26cm/s, while Khanna et al. (1994) found it to be 15-48cm/s or 33cm/s. For the current research, the stable operating range was found to be 43–50cm/s, or 7cm/s. The top end of the burner stable range predicted by the current model is in much better agreement with Khanna et al. (1994) compared to Barra et al. (2003). In contrast, both the present work and Barra et al. (2003) greatly overestimate the lower stabile velocity. Barra et al. (2003) attribute this disagreement to their definition of flashback. In the physical burner, the flame could stabilize upstream of the

burner face, a phenomenon that is difficult to simulate and is excluded from the numerical definition of flash-back. Therefore, our model can still be considered accurate for the parameters of interest to this study, due to agreement with the stable range upper limit result the work of Khanna et al. (1994) and the explanation for the lower limit provided by Barra et al. (2003).

Barra et al. (2003) performed twelve additional tests to investigate the effects of changing certain parameters on the stable operating range, however, only two are reported here for the sake of validation. For the first comparison, the thermal conductivity of both sections was decreased by a factor of ten. Barra et al. (2003) observed a significant decrease in the flash-back limit and only a slight decrease in the blow-off limit, resulting in an increase in the stable operating range. The same trends were observed from the current simulation; the flash-back limit dropped to 34cm/s and the blow-off limit dropped to 44cm/s resulting in a larger operating range of 10cm/s. The second comparison was made with the thermal conductivity of both sections of porous media being increased by a factor of ten. Barra et al. (2003) observed a significant increase in both the flash-back and blow-off limits; however the overall stability range decreased. The current model had the same trends, with the flash-back limit increasing to 70cm/s, the blow-off limit increasing to 73 cm/s, and the stable range decreasing to 3cm/s

As a final validation, we compare the calculated temperature and species profiles with those of Barra et al. (2003) to see if they followed similar trends and make physical sense. Although the profiles obtained from the current research are quite different in shape, peak temperature, and flame location compared to those in Barra et al. (2003) as a result of the different burning rates, the overall trends are in agreement and make physical sense. The first profiles observed were those of the gas and solid temperature, seen in Figure 3.13.

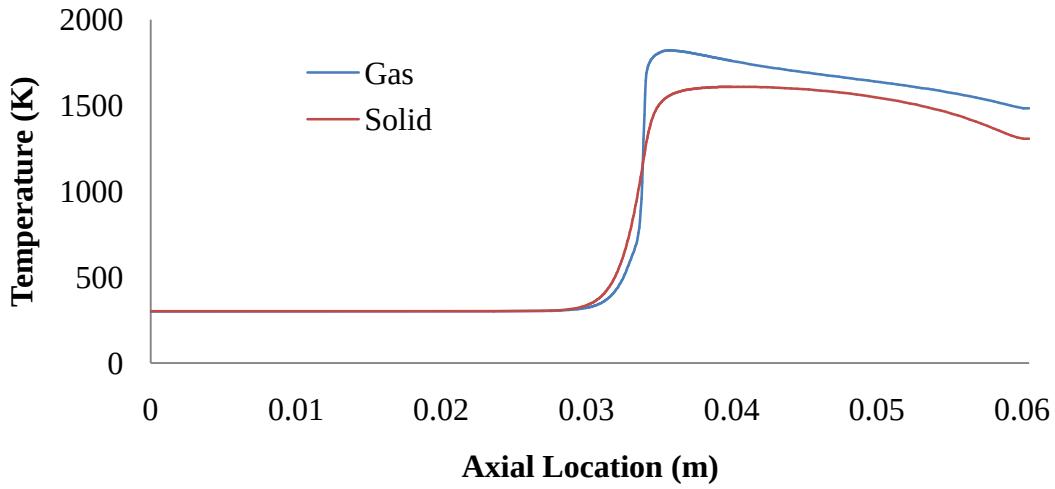


Figure 3.13 - Gas and Solid Temperature Profiles for the Reference Case

The flame front is located right at the burner interface as we would expect. Secondly, the solid temperature is higher than the gas temperature upstream of the flame front, allowing for the pre-heating of the gas. A pre-heat zone is also observed leading up to the flame front. After the flame front the general expectations are also met; the gas temperature peaks to be hotter than the solid matrix and is then cooled by it. We also examine the major species of the reaction, seen in the magnified view of the flame front in Figure 3.14.

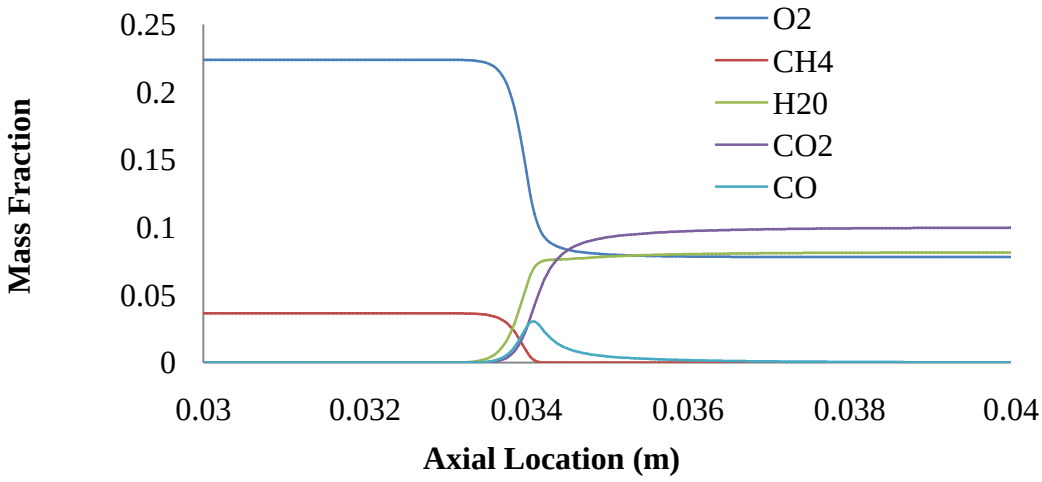


Figure 3.14 - Major Species Profiles in Flame Front for the Reference Case

All of the species profiles shown are similar to that of Barra et al. (2003) and follow the expected trends. Figure 3.14 shows the flame front occurring at the burner interface, as previously stated when discussing the temperature profiles. The reactants stay at their initial values until they are close to the flame front where they begin to change. As the reference case is for a lean flame, we see all of the methane being consumed and some oxygen being left over as expected. The products begin to form and rise to their expected values within in the flame front. The CO profile also follows the expected trend, forming quickly in the flame front and then being consumed by the excess oxygen in the reaction.

As the burner model reproduces both the experimental data gathered by Khanna et al. (1994) and the trends observed by Barra et al. (2003) the model can be considered verified and validated and can be used to accurately model the combustion of methane within a porous ceramic burner.

Chapter 4
Optimization Method

4.1 Introduction

This chapter presents the optimization algorithm used in the present research. First, some principles of optimization will be laid out, followed by a description of Newton's method. Response Surface Modelling (RSM) will be discussed next as a way of improving the functionality of Newton's method for stiff sets of equations. These two methods together will be used in the current research so a validation of the optimization algorithm is also included.

4.2 Optimization Principles

Optimization is the process that finds the minimum value of a given function, known as the objective function, which mathematically quantifies the quality of the design. The variables on which the objective function is based, known as the design variables, represent system parameters that we are able to change and control. By using mathematical algorithms to identify the minimum of the objective function we are really finding the best possible operating point for that system, and the design variables that produce that minimum are the conditions that the system should be run at to produce the optimal performance. The optimization problem is stated mathematically as

$$\mathbf{x}^* = \text{argmin}\{F(\mathbf{x})\} \tag{4.1}$$

where $\mathbf{x}$ is the vector of design variables, $\mathbf{x}^*$ are the design variables resulting in the minimum value of the objective function, $F(\mathbf{x})$. Optimization is far more rigorous than a trial-and-error

approach for several reasons. First, as the problem is represented mathematically, all of the interactions between the design variables will be accounted for in the model, unlike a parametric study where the variables are generally changed independently of each other. Second, the objective function is continuous, rather than a set of discrete points as in trial-and-error approaches, which means that sensitivities to the design variables can be found. Using these sensitivities changes to the design variables can be found that may result in improvements to the objective function. The design variables are changed by

$$\mathbf{x}^{k+1}=\mathbf{x}^{k}+\alpha^{k}\mathbf{d}^{k} \tag{4.2}$$

where $\mathbf{x}^{k+1}$ is the new set of variables, $\mathbf{x}^{k}$ is the current set of variables, α^{k} is a scalar called the step length, and $\mathbf{d}^{k}$ is the search direction. To create an improvement in the objective function, $\mathbf{d}^{k}$ must be a descent step, meaning

$$-\mathbf{g}\cdot\mathbf{d}^{k}>0 \tag{4.3}$$

where $\mathbf{g}$ is the gradient of the objective function containing the first order objective function sensitivities with respect to the unknowns in $\mathbf{x}$ such that $\mathbf{g}_{p}=\partial F\left(\mathbf{x}^{k}\right)/\partial\mathbf{x}_{p}^{k}$. There are several strategies for choosing $\mathbf{d}^{k}$ and α^{k}, but for this research, we will use a modified Newton's method.

4.3 Modified Newton's Method

Newton's method is powerful yet simple method for optimizing functions (Nocedal and Wright 2006). The method is derived by finding the vector $\mathbf{d}^{k}$ at the k^{th} iteration that, when added to $\mathbf{x}^{k}$, will produce the largest possible drop in objective function.

$$\mathbf{x}^{k+1}=\mathbf{x}^{k}+\mathbf{d}^{k} \tag{4.4}$$

Assuming the function is at least second-order differentiable, we perform a Taylor series expansion of the objective function about the new set of variables, $\mathbf{x}^{k+1}$, to obtain

$$F(\mathbf{x}^{k+1})=F(\mathbf{x}^{k}+\mathbf{d}^{k}) = F(\mathbf{x}^{k})+\mathbf{d}^{k^T}\mathbf{g}+\frac{1}{2}\mathbf{d}^{k^T}\mathbf{H}\mathbf{d}^{k}+\text{H.O.T.} \tag{4.5}$$

where $\mathbf{H}$ is the Hessian of the objective function containing the second-order sensitivities such that $\mathbf{H}_{pq}=\partial^2 F(\mathbf{x}^{k})/\partial\mathbf{x}_p^k\partial\mathbf{x}_q^k$, and H.O.T. are the higher order terms. Knowing that an extreme point of a function, which may be the minimum, occurs when the gradient is equal to zero, we now take the gradient of Eqn. (4.5) with respect to $\mathbf{x}^k$ to get

$$\nabla F(\mathbf{x}^{k}+\mathbf{d}^{k}) = \mathbf{g}+\mathbf{H}\mathbf{d}^{k}+\text{H.O.D.} \tag{4.6}$$

where H.O.D. stands for higher order derivatives. Setting the gradient equal to zero and treating the higher order derivatives as an error, followed by some rearranging, Eqn. (4.6) becomes

$$\mathbf{d}^{k} = -\mathbf{H}^{-1}\mathbf{g} \tag{4.7}$$

Equation (4.7) is known as the Newton equation and is used to find the search direction for Newton's method. At this point a few observations about Newton's method should be made. First, even though the search direction is derived to go straight to an extreme point, due to the error term it will not reach the minimum in a single step unless the objective function is quadratic, which necessitates iteration for most objective function. Fortunately, most functions can be modelled accurately as quadratic when sufficiently close to the minimum, so within the region of the minimum Newton's method guarantees rapid convergence, and is generally regarded as the most efficient minimization algorithm for problems where $F(\mathbf{x})$ is continuous and unimodal. It is important to note, however, that by setting the derivative equal to zero, Newton's method only searches for extreme points, which are not necessarily local minima. If a maximum

point or a saddle point is close to the current point, Newton's method will step towards these points, thereby increasing the objective function value. To correct for this, Newton's method needs to be improved.

The modification begins with a test for descent. If Eqn. (4.3) does not hold, then the step determined by Newton's method is an ascent step and should not be used. Instead, we revert to the steepest descent step (Nocedal and Wright 2006), defined as

$$\mathbf{d}^k = -\mathbf{g} \tag{4.8}$$

where **g** is the gradient of the objective function, which by definition points in the direction of steepest ascent. Therefore by adding a negative sign we now point in the opposite direction. By taking the steepest descent step, we are guaranteed that the objective function will be improved. However, as no other information is accounted for, the steepest descent method converges very slowly and always requires the calculation of the step length parameter in Eqn. (4.2), which will be discussed in Section 4.4.1. Fortunately, the algorithm only uses steepest descent steps to get within the vicinity of a local minimum at which point Newton's method will be able to find descent steps with a much higher rate of convergence.

Equations (4.7) and (4.8) show that most non-linear programming algorithms require first and second order derivatives to calculate the search direction, $\mathbf{d}^k$, and occasionally the step length, α^k. For a known function these values can often be determined analytically, but for an unknown function, which is the case in this research, these values need to be determined numerically. This is usually done through finite differencing, where a first and second order derivative are approximated as

$$\nabla F(x) \approx \frac{F(x+\Delta x)-F(x)}{\Delta x} \tag{4.9}$$

$$\nabla^2 F(x) \approx \frac{F(x+\Delta x)-2F(x)+F(x-\Delta x)}{\Delta x^2} \tag{4.10}$$

respectively, where Δx is a small value usually on the order of magnitude of the square root of machine precision, which is the accuracy of a floating point system before round-off becomes significant. Since three function evaluations per variable per search direction are required, with the possibility of the function being expensive to evaluate and having many variables, this can become very time consuming. Furthermore, if the governing system of equations is stiff, as is the case in this problem, the objective function evaluations will be contaminated with noise, which is amplified when dividing by Δx and can dominate the finite difference estimates of the gradient and Hessian. As these two problems with the modified Newton's method are quite serious, and pertain to the problem at hand, a different approach to the derivative calculations must be used.

4.4 Response Surface Modelling

Response surface modelling (RSM) (Myers, Montgomery and Anderson-Cook 2009) is a method for optimizing objective functions that are expensive or difficult to evaluate. It was developed for use in optimizing physical experiments in a stochastic way. Rather than determining the true objective function, several points in the design space are selected. A low order polynomial interpolating function is then fit to these points, usually using a least squares regression, and this model function that is optimized instead of the true objective function. RSM has many benefits over a regular Newton's method approach, including: fewer function evaluations; reduced computational time; noise reduction; and finally both the gradient and Hessian are analytically tractable. This section will describe the RSM method used in this

research for selecting the points, fitting and optimizing the surface, and using these surfaces to optimize the objective function. Although we describe RSM for minimizing an objective function having two variables, it can be extended to minimize functions having many more variables.

4.4.1 Point Selection and Surface Generation

The points are selected using the Face Centered Central Composite (FCC) design method (NIST/SEMATECH 2010). In principle, only six interpolating points are required to generate a quadratic function in two dimensions. In this case, however, since the objective function contains numerical noise induced by the stiffness of the governing equations, an additional three points, for a total of nine interpolating points, are required. This mitigates the noise in $F(\mathbf{x})$ while still keeping the total number of points low to limit the computational effort required to construct the objective function. The nine points are selected in an organized fashion, shown in Figure 4.1. Of the nine points, eight of them are newly selected and centered about the current and ninth point, $\mathbf{x}^k$, in a rectangular arrangement. The points are spaced at a distance of γ_1 in the x_1 direction and a distance of γ_2 in the x_2 direction away from the current point, giving the rectangle the dimensions of $2\gamma_1 \times 2\gamma_2$. The values of γ_1 and γ_2 are selected empirically to ensure efficiency and stability; if they are too large the response surface model will poorly approximate $F(\mathbf{x})$, while if these values are too small the method will take a very long time to reach $\mathbf{x}^k$.

The function value is now calculated at each of the model points, followed by a quadratic function being fit to the nine points using a least squares regression. As the derivatives of any polynomial function can be calculated analytically, no approximations have to be used in Eqns. (4.7) and (4.8). This corrects one of the major problems with a regular Newton approach given

in Section 4.3 and is one of the ways that RSM accounts for the stiffness in the governing equations. As the derivatives are easy to calculate for any polynomial, one may intuitively conclude that a high order polynomial would be desirable to ensure a better fit to the data. However, higher order polynomials require more interpolating points. For example, a cubic fit requires fifteen points. This would substantially increase the computational time if the function evaluation was expensive to calculate, which it is for the combustion problem. Therefore, a quadratic model helps to minimize the number of required function evaluations. Lower order polynomials also act to smooth out the objective function, thus removing any noise that may be contaminating the data

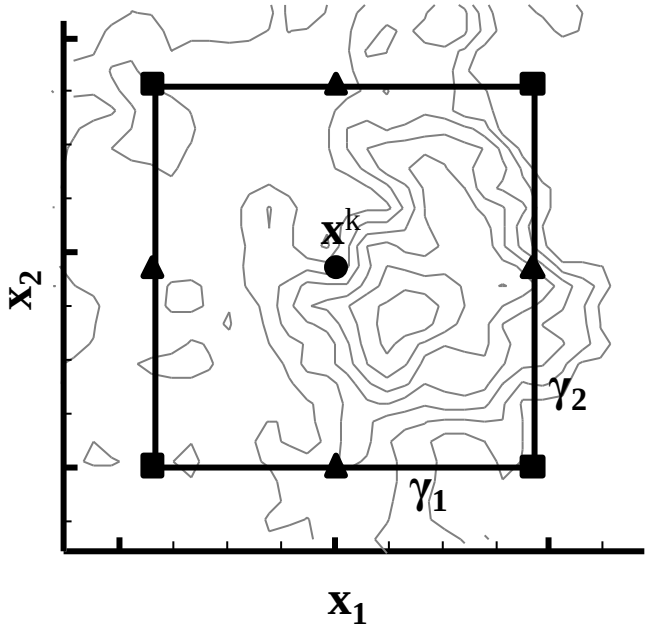


Figure 4.1 - FCC Point Selection Schematic

Now that a surface has been generated, the optimal value of the surface can be found using the modified Newton's method. As the surface is quadratic, Newton's method finds the extreme point in a single step, or takes the steepest descent step in the case of the extreme point being in an ascent direction. However, since we are now using a model of the objective function

we must constrain the problem within the model region. This means that Eqn. (4.2) must be used and a step length must be calculated. Fortunately, there are only three possible cases for determining the value of α^k:

- Case 1: If the extreme point is inside the model region and Newton's method calculates a descent direction then the step length parameter is equal to one and the full Newton step is taken to the minimum of the model region.
- Case 2: If the extreme point is outside the model region and Newton's method calculates a descent direction then the step length parameter is equal to the distance to the edge of the model region in the direction of the Newton's step.
- Case 3: If the extreme point is in an ascent direction then the step length parameter is equal to the distance to the edge of the model region in the steepest descent direction.

Once the value of α_k has been selected, Eqn. (4.2) can be used to find the new current point and its objective value. At this point, the stopping criteria, of which there are two, are checked to see if the optimal solution has been found. The first criterion is when the norm of the difference between the new and current point is less than a certain tolerance, selected as empirically as 10^{-4} for this research. This can be represented mathematically as

$$\left\|\mathbf{x}^{k+1}-\mathbf{x}^{k}\right\|<10^{-4} \tag{4.11}$$

The second stopping criterion is when the new function value is greater than the old function value, or an ascent step was taken for the actual function. As the minimization is carried out on a model function, it is possible that the descent step that it calculates is actually an ascent step for the objective function. The only way that this would occur is if the amount of numerical noise in the objective function, caused by the stiffness of the problem, overwhelms the unperturbed

objective function. In this situation further optimization would be wasted computational effort since any reductions in the objective function would be dominated by errors. Therefore, the solver does not take the calculated step and the optimization procedure terminates. If neither of the stopping criteria are met, then the solver continues by selecting new points.

When selecting new points, there are two scenarios to be considered: whether the new point is on the model region's boundary, or within the interior. If the new location is on the edge of the model region, then the values of γ_1 and γ_2 remain the same and new points are selected. This effectively shifts the model region from being centered about the old point to being centered about the new one. If the new point is inside the model region, on the other hand, then there is a good chance that the objective function minimum is located within the existing region. Therefore, to obtain a more accurate model, and therefore a more accurate answer, the model region is shrunk by decreasing γ_1 and γ_2 by a factor of two before selecting new interpolating points.

4.4.2 Constrained Optimization

If the optimization problem of interest is constrained, then several alterations need to be made to the RSM algorithm detailed above. These changes pertain to point selection, model region size, and boundary optimization.

4.4.2.1 Point Selection

During constrained optimization, if the current point is closer to the edge of the constrained region than γ_1 or γ_2 the points need to be selected in a different manner. The points that would otherwise be out of the constrained region are moved so that they are on the edge of

the region, as shown in Figure 4.2. This ensures that all function evaluations are performed at feasible locations and that any step calculated will also be within the feasible region.

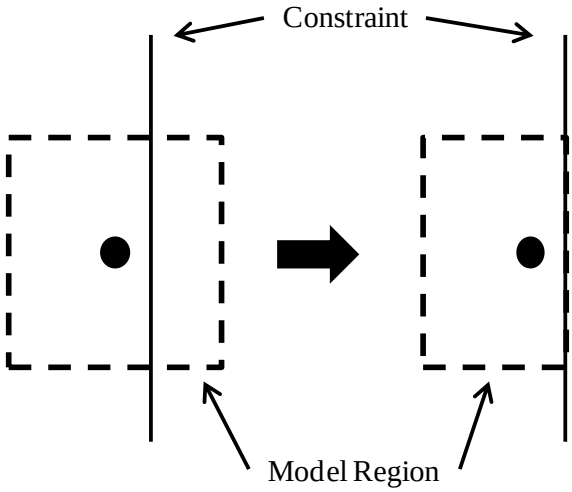


Figure 4.2 - Change to Model Region Near a Constraint

Another change to consider is if the current point is on the constraint itself. Rather than using only half of the model region, which is what the above change would recommend, the model region is shifted so that the current point is on the edge of the model region rather than being at its center, as shown in Figure 4.3. This ensures that all points are feasible as well as ensuring that the model region is not shrunk prematurely.

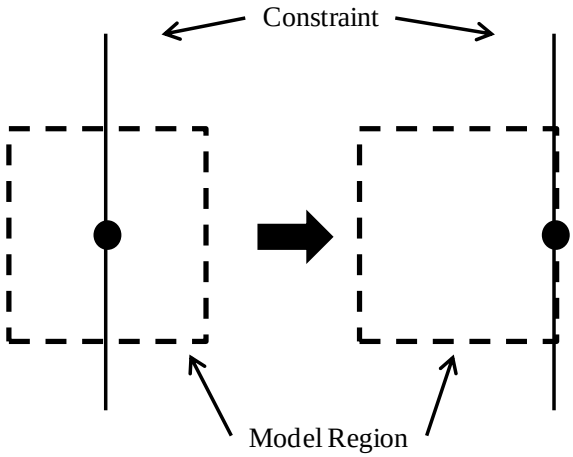


Figure 4.3 - Change to Model Region on a Constraint

4.4.2.2 Model Region Sizing

If the calculated step goes from being within the feasible region to being on the boundary of the feasible region, the size of the model region must be reconsidered. Normally, if a point is selected on the edge of the model region the region is shifted rather than being shrunk. However, if the edge of the model region and the constrained region are the same then the model region is shrunk. Being on the constraint generally means that the objective function minimum lies outside of the constrained region, however, as we are using an approximation to the objective function it is possible that the model is too large to capture the detail of the objective function near the constraint. By shrinking the model region and calculating a new search direction and step length we allow the solver to step back into the feasible region if necessary.

4.4.2.3 Boundary Optimization

If the search direction leads to an infeasible point, i.e. one that lies outside of the feasible region, the dimensionality of the problem is decreased by one by treating the inequality constraint of the boundary as an equality constraint using a method called the generalized reduced gradient method (Gill, Murray and Wright 1986). Further improvements can now be found by searching along the boundary of the feasible region. Once a minimum is found, the algorithm reverts back to its original dimensionality, shrinks the model region and calculates a new search direction. This process of optimizing along the boundary continues until either of the stopping criteria is met or the search direction points back into the feasible region.

4.4.3 Error Estimation

As RSM is a stochastic method, an estimate of the error in the function value is desirable upon the completion of the algorithm. We start by calculating the mean square error of the surface by

$$s = \sqrt{\sum_{i=1}^{n} \frac{\left(y_i - \hat{y}_i\right)^2}{n-p}} \tag{4.12}$$

where n is the number of points used to make the surface, p is the number of regressor variables, and y_i and $\hat{y}_i$ are the true objective function value and model function value of the i^{th} point respectively. This calculation also reflects why more interpolating points are used rather than the minimum required when making the response surface. If the minimum number of points were used then the amount of error in the surface would be reported as zero, as the model would be exact at the interpolating points. With the addition of more points, not only is the surface more accurately represented but the error is as well. Next, we calculate the estimated standard error of the minimum of the surface by

$$s_{\hat{y}(x)} = s\sqrt{\mathbf{x}^{(m)^T}\left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{x}^{(m)}} \tag{4.13}$$

where $\mathbf{x}^{(m)}$ is a vector, based on the minimum location of the function, of the form

$$\mathbf{x}^{(m)} = \left[1, x_1, x_2, x_1^2, x_2^2, x_1x_2\right]^T \tag{4.14}$$

and $\mathbf{X}$ is a matrix whose rows are the $\mathbf{x}^{(m)^T}$ vectors for each of the points that made up the model surface. Finally, the error can be calculated for constructing confidence intervals by

$$e = t_{\alpha/2,n-p} s_{\hat{y}(x)} \tag{4.15}$$

where $t_{\alpha/2,n-p}$ represents the value from the Student's-t distribution for a $100(1-\alpha)\%$ confidence interval. Here a 90% confidence interval was desired ($\alpha=0.1$) which led to $t_{\alpha/2,n-p}$ being equal to 2.919986. What this confidence interval means is that if the error in the data is assumed to be random noise, 90% of the time the minimum function value will be the calculated value plus or minus the error term, *e*.

4.5 Verification of RSM Algorithm

The purpose of this section is to verify the validity of the RSM algorithm for optimizing functions, the code for which can be found in Appendix D. A test case will be carried out on Rosenbrock's function, which is a common test case for optimization algorithm (Gill, Murray and Wright 1986), and can be represented mathematically as

$$F(\mathbf{x})=100\left(x_2-x_1^2\right)^2+(1-x_1)^2 \tag{4.16}$$

This function has a strong global minimum at $(1,1)^T$ with a function value of zero. Finding this minimum numerically can be quite challenging, however, since it is surrounded by a shallow valley having steep sides, as seen in Figure 4.4. The steep sides can lead to errors when calculating the gradients numerically, due to the rapidly changing function values, and the flat valley can cause the algorithm to stall, due to the fact that the Hessian becomes nearly singular. Therefore, if the RSM algorithm is capable of finding the minimum of Rosenbrock's function we can consider it a viable method for optimization and use it to optimize combustion devices. It should be noted that ascent steps were allowed in this example. To start the RSM algorithm an initial point is selected as $(0,0)^T$ while γ_1 and γ_2 are both set equal to unity. Figure 4.5 shows the

contours of the first response surface, with superimposed contours of Rosenbrock's function and the + representing the current point.

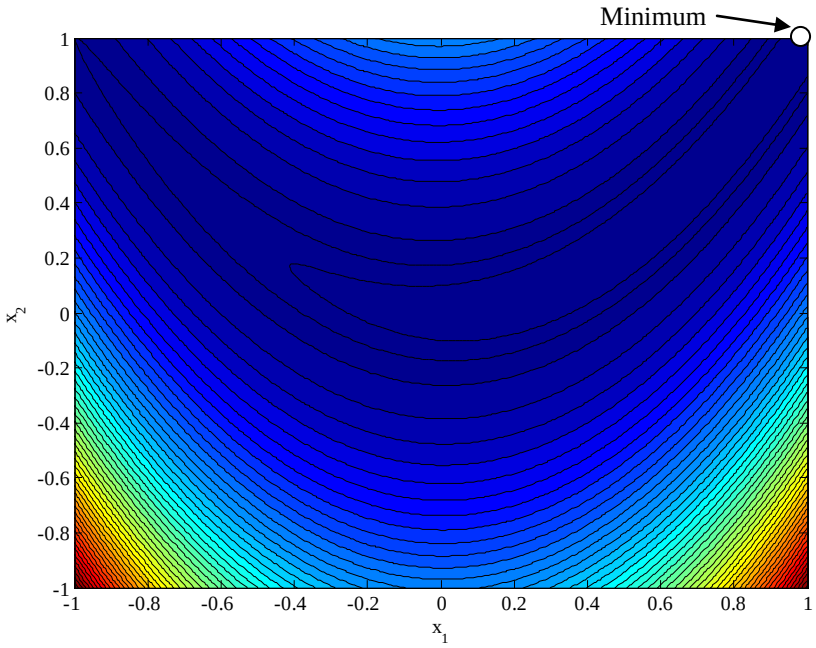


Figure 4.4 - Rosenbrock's Function

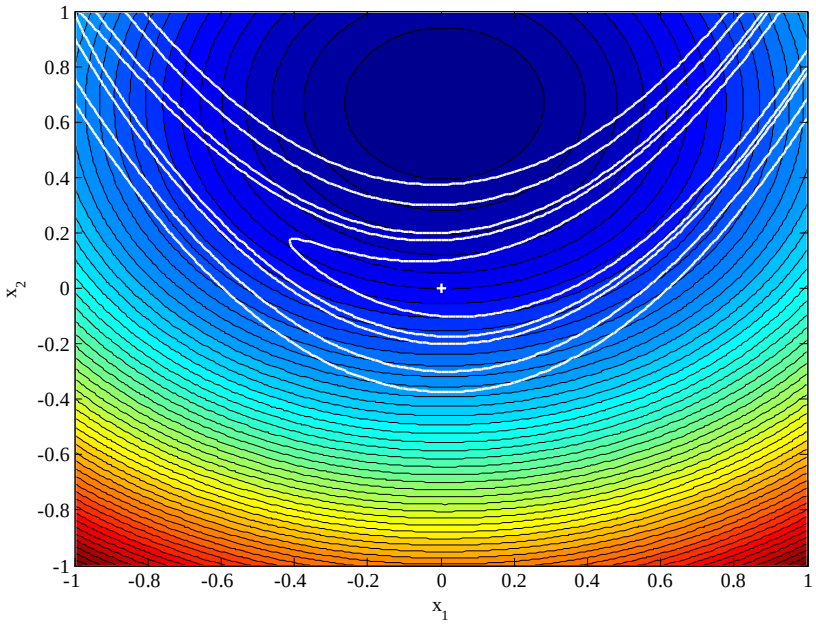


Figure 4.5 - 1st Surface of Rosenbrock's Function

As a minimum clearly exists within the model region the algorithm will take a step to this value, followed by the model region being decreased in size. New points are now selected to form a new surface to approximate the objective function, shown in Figure 4.6 with the square representing the new model region.

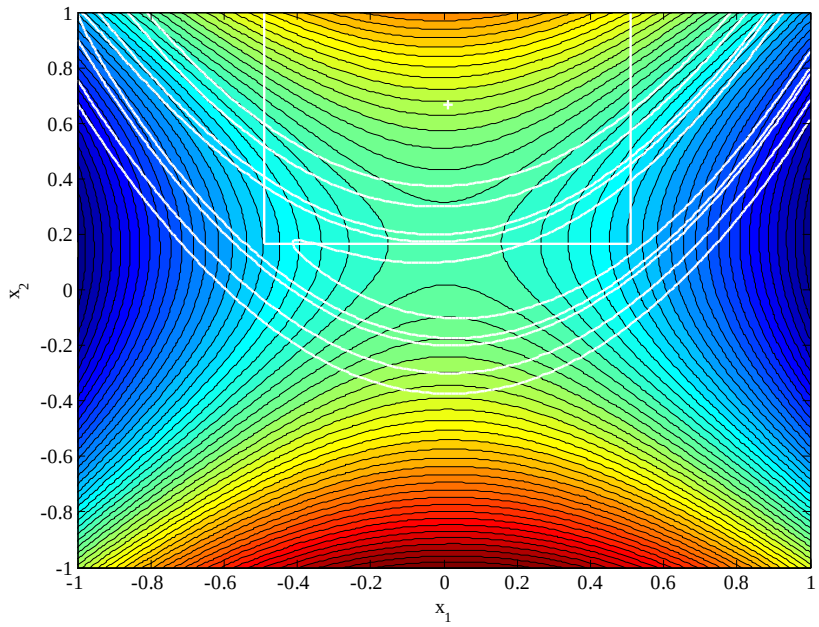


Figure 4.6 - 2nd Surface of Rosenbrock's Function

On first glance this may appear to be a bad model, but closer inspection shows that the contours of the response surface match those of Rosenbrock's function in the model region. Here the extreme point is a saddle point; it is still in a descent direction, however, so the RSM algorithm takes the Newton's step towards it. However, the extreme point is outside of the model region so the step length is set to the distance to the edge of the model region. Now the values of γ_1 and γ_2 are unchanged, causing the model region to shift, and new points are selected, leading to the new surface shown in Figure 4.7.

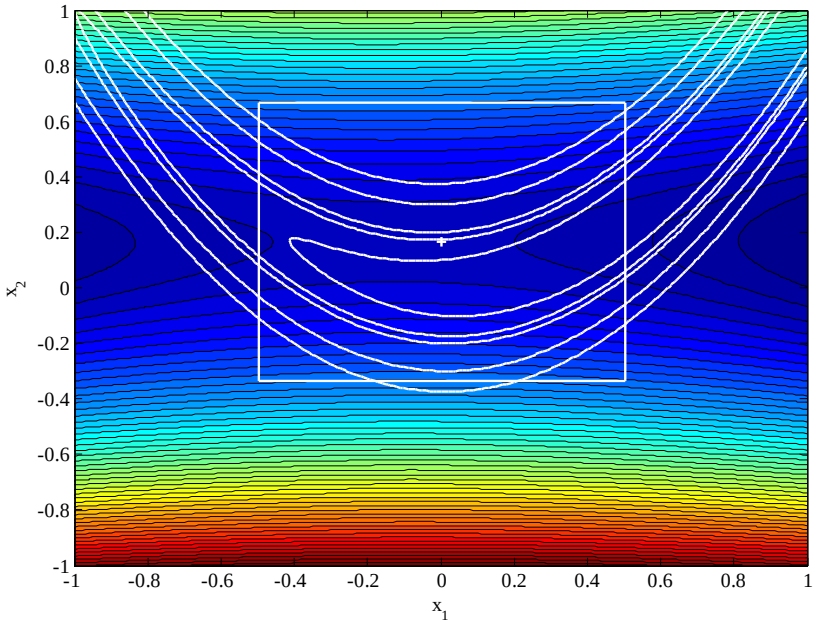
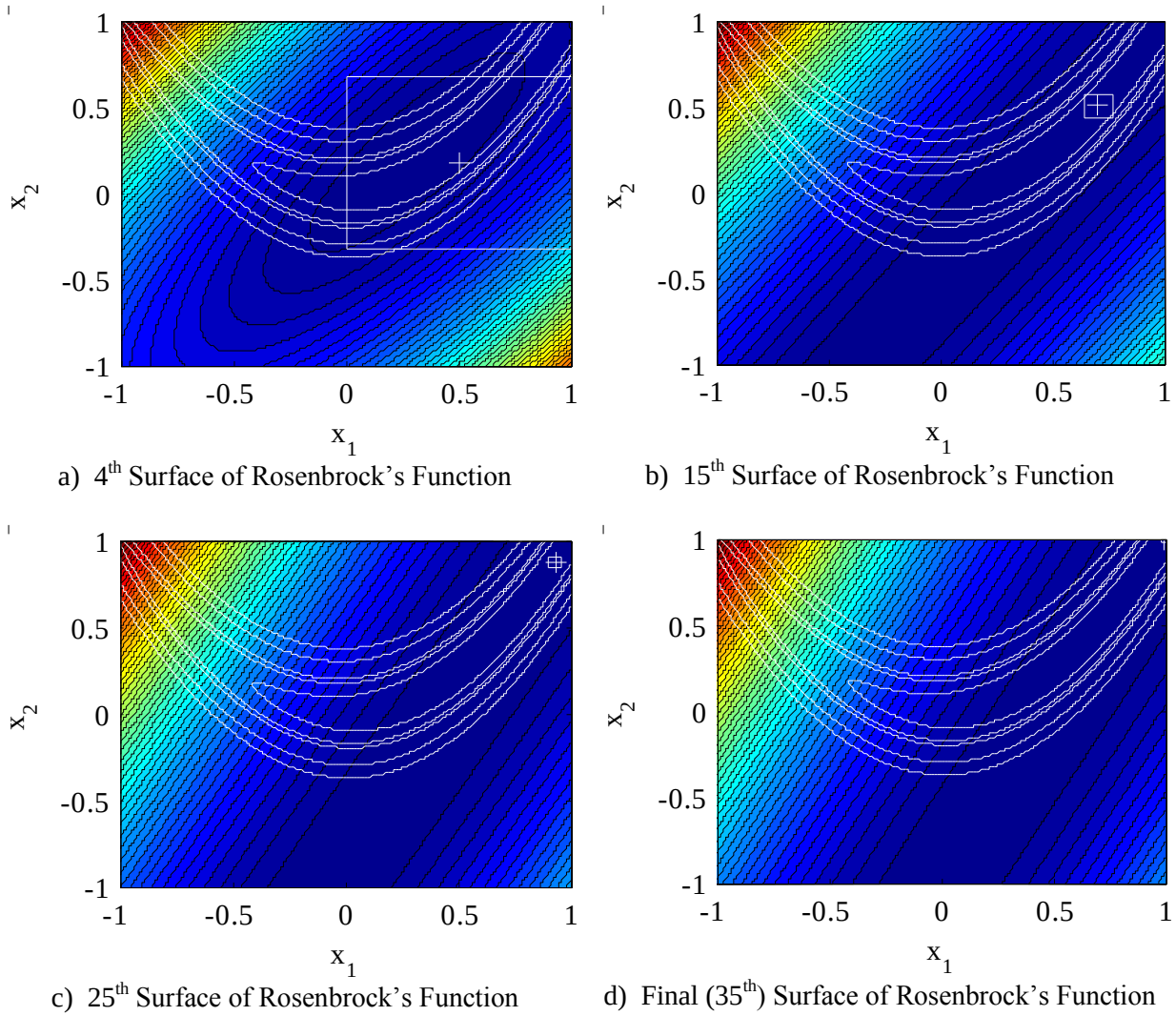


Figure 4.7 - 3rd Surface of Rosenbrock's Function

Once again, the extreme point is a saddle point; however, it is also in an ascent direction. Therefore, the RSM algorithm takes the steepest descent step to the edge of the model region. When the algorithm continues, shifting and shrinking the model region and taking the steepest descent step when necessary, it converges to the point $(0.9959,0.9920)^T$ with a function value of $1.6588\times10^{-5} \pm 0.0033$ in 35 iterations, requiring 0.035 seconds. Several of the remaining surfaces can be seen in Figure 4.8, while the path taken to the minimum is shown in Figure 4.9.

Figure 4.8 shows that as the solver progresses towards the minimum, the contours of the model function become aligned with the contours of the true objective function. The path taken to the minimum is also enlightening. At first the solver struggles, taking ascent steps and leaving the valley of the function, due to the steepness of the sides affecting the model. However, once the algorithm locates the center of the valley and the model region is sufficiently small, the solver remains at the center of the valley until the problem has converged. As the RSM algorithm was capable of finding the minimum value of Rosenbrock's function, a numerically

difficult problem to optimize, within a reasonable tolerance the RSM algorithm is considered verified and is validated for use in optimizing combustion problems.



a) 4th Surface of Rosenbrock's Function

b) 15th Surface of Rosenbrock's Function

c) 25th Surface of Rosenbrock's Function

d) Final (35th) Surface of Rosenbrock's Function

Figure 4.8 - Intermediate Surfaces of Rosenbrock's Function

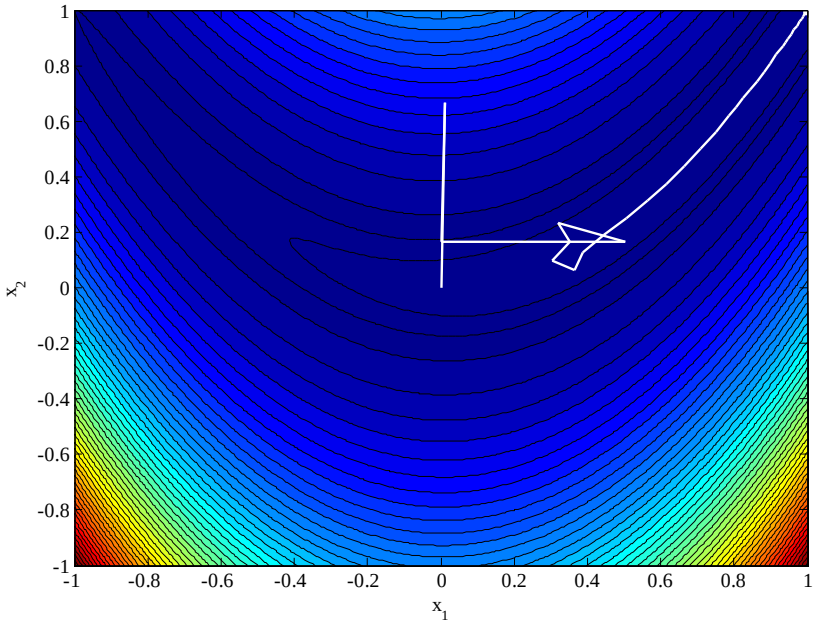


Figure 4.9 - Path to Minimum of Rosenbrock's Function

Chapter 5
Implementation and Results

5.1 Introduction

Using the combustion model and optimization algorithm presented in Chapter 3 and 4 respectively, we now consider the design optimization of the porous radiant burner. This chapter contains two one-dimensional studies and a two-dimensional study to try and improve the performance of a porous radiant burner. The objective of these studies is to maximize the radiant efficiency, expressed mathematically as

$$\mathbf{x}^* = \text{argmin}\{F(\mathbf{x})\} = \text{argmin}\left\{-\frac{T_{s,out}(\mathbf{x})^4}{T_{ad}{}^4}\right\} \tag{5.1}$$

where $\mathbf{x}$ is the vector of design variables and T_{ad} is the adiabatic flame temperature, which is the temperature achieved for complete combustion of an open flame without any additional energy being transferred into or out of the system. As the optimization algorithm is for minimizing functions, and we wish to maximize the radiant efficiency, the negative sign was added to convert the problem from maximization into minimization. The code for optimizing the efficiency can be found in Appendix D.

5.2 One Dimensional Studies

Two one-dimensional optimization studies are performed on the porous radiant burner: first on the second stage pore diameter; and then on the second stage porosity. This section details the two optimizations performed, starting with selecting the initial point and point spacing

parameter, providing the maximum function value with justification, and discussing the path to the maximum.

5.2.1 Stage Two Pore Diameter

The first optimization was carried out on the second stage pore diameter. This parameter was chosen as many of the correlations used in the combustion model rely on the pore diameter of the solid. As such, this would help to highlight the non-linear interactions of the combustion model. We focus on the pore diameter in the second stage as this is the reacting zone in which most of the heat transfer is taking place. The initial value for the pore diameter was selected as the nominal value of 1.52mm, and a grid spacing parameter of 0.0375mm was chosen. Table 5.1 provides a summary of the starting point for this optimization, as well as the function value for the reference case.

Table 5.1 - Initial Parameters for $d_{p,2}$ Optimization

Parameter	Value
$\mathbf{x}^0$	1.52mm
γ	0.0375mm
$F(\mathbf{x}^0)$	27.29%

Rather than having the initial set of RSM points surround the initial point, they were all located at smaller values, or to the left, of $\mathbf{x}^0$. The first response surface, and function values at the desired points, seen in Figure 5.1, is a concave surface, or convex when the negative is applied. Therefore, a Newton step can be taken to the edge of the model region and a new surface can be constructed about this new point. The algorithm continues building surfaces and finding the minimum, as described in Chapter 4, until it reaches the tenth surface, shown in Figure 5.2.

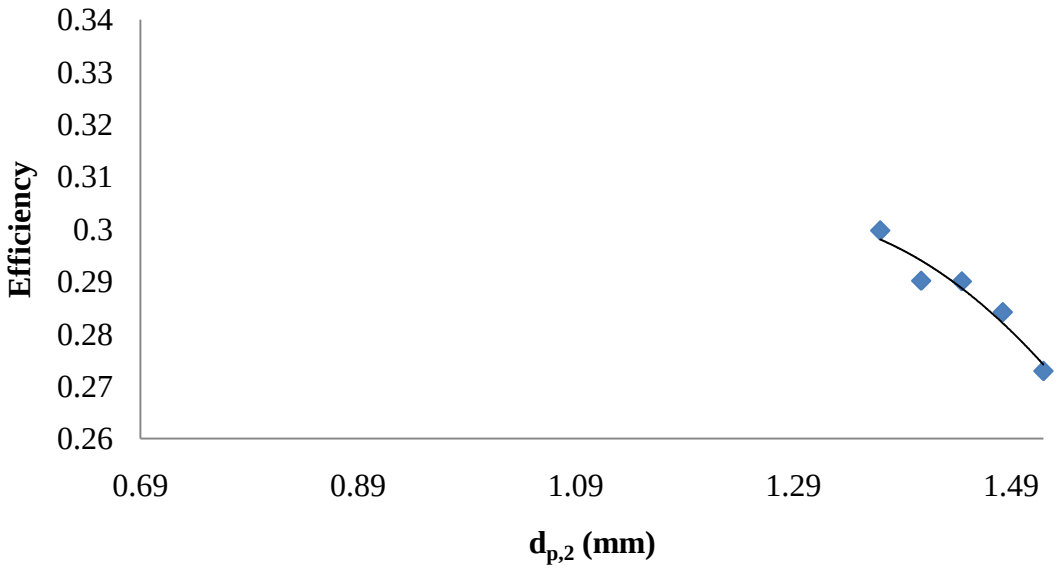


Figure 5.1 - First Response Surface for $d_{p,2}$ Optimization

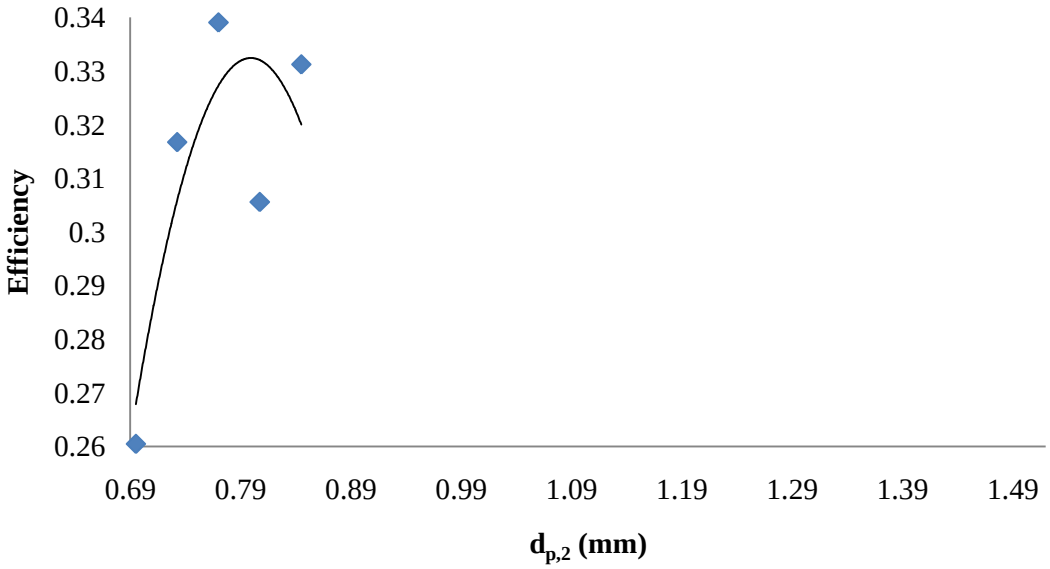


Figure 5.2 - Tenth Response Surface for $d_{p,2}$ Optimization

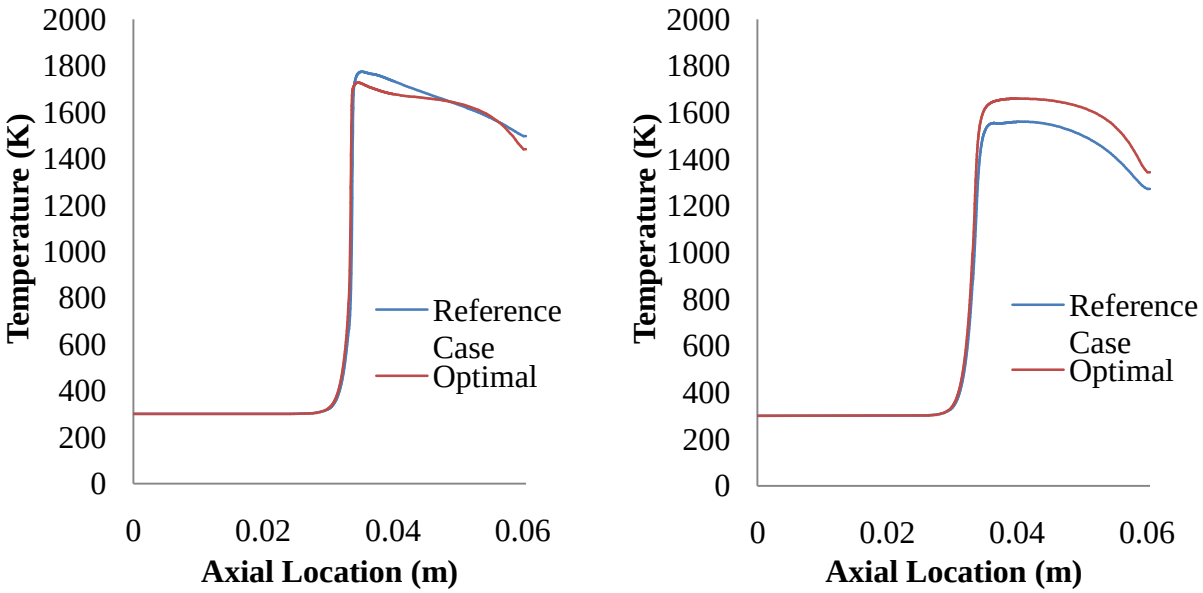
Here the algorithm has found its first maximum within the model region. The amount of noise in the model is quite large at this point, however, leading to the maximum of the model function being less than the maximum of the current point. Therefore, one of the stopping criteria has been met and the optimization terminates after 9.1 hours. The algorithm gives the

maximum efficiency as 33.91% ± 2.75% with the second stage pore diameter being equal to 0.77mm. This is a statistically significant improvement as the total change in efficiency is 6.62%, which is larger than the amount of error in the final answer. Therefore, even in the worst case scenario for the error, an improvement has still been found in the performance of the porous radiant burner.

This answer also makes physical sense. As previously shown in Figure 3.6, decreasing the pore diameter increases the volumetric heat transfer coefficient, and therefore the amount of convective heat transfer. This means that for the optimal case more heat will be transferred from the gas to the solid in the second stage of the burner, thus leading to an increase in the solid exit temperature and the radiant efficiency. This is confirmed in

Figure 5.3 where the temperature profiles for the reference case are compared to the optimal solution. For the optimal solution, the gas temperature decreases much more rapidly after the flame front before levelling off. The solid temperature on the other hand rises much higher for the optimal case and remains higher all the way to the burner exit, leading to the greater efficiency. We would also expect that decreasing the pore diameter too much would be detrimental to the efficiency. As Eqns. (2.2) and (3.17) show, a decrease in the pore diameter will lead to an increase in the thermal conductivity and extinction coefficient respectively. This means that conduction and radiation in the burner are also becoming more dominant. Eventually these two methods of heat transfer would become the dominant forms of heat transfer causing more energy to be moved upstream, resulting in a drop in exit temperature, and thus efficiency regardless of the extra energy being convected into the solid. All of the response surfaces leading to the maximum value, as well as all the actual function values, can be seen in

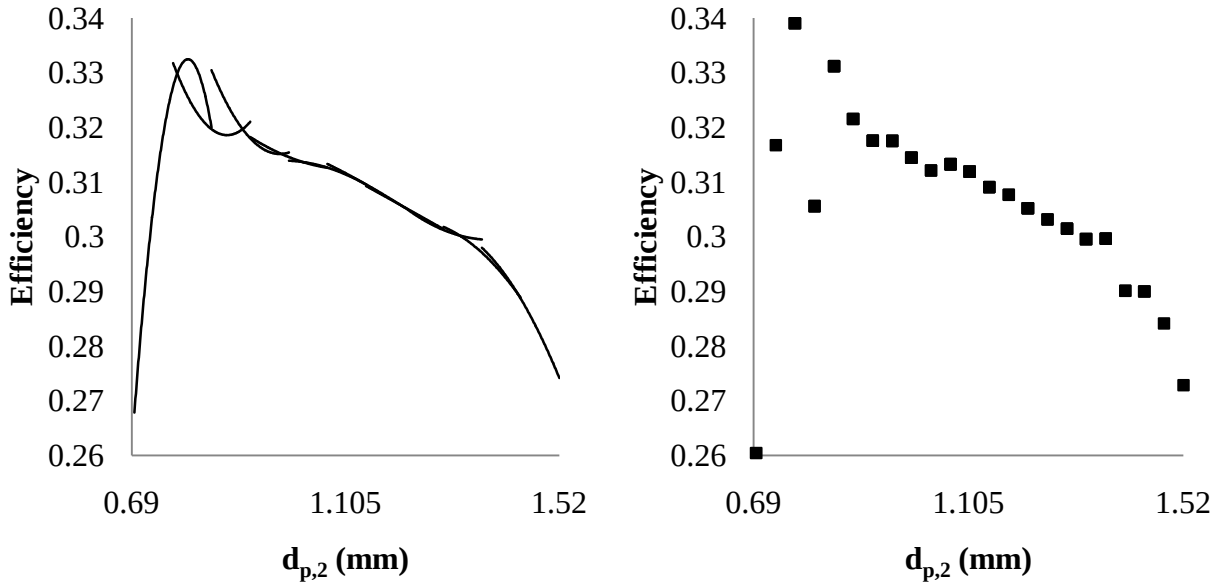
Figure 5.4.



a) Gas Temperature Profile Comparison

b) Solid Temperature Profile Comparison

Figure 5.3 - Reference vs. Optimal Temperature Profile for $d_{p,2}$ Optimization



a) Response Surface

b) Function Values

Figure 5.4 - Response Surfaces and Function Values for $d_{p,2}$ Optimization

At first the objective function is quite smooth and has a clearly defined maximum where the algorithm suggested it was located. It is interesting to note that as the solver approaches the maximum value for the radiant efficiency the amount of noise in the objective function also increases. This suggests that near the maximum value the equations have become even more stiff, leading to noise being introduced into the objective function. This is most likely caused by the production rates also increasing leading to enhanced coupling in the governing equations. Equation (3.9) shows that as the temperature of the gas increases, as

Figure 5.3 shows it does for the optimal solution, the reaction rate constants will also increase. These increases in reaction rate will in turn lead to an increase in the species production and consumption rates, as seen in Eqn. (3.7). This means that the chemical aspect of the model will become more important, thus leading to the speculated enhanced coupling as well as an increase in the stiffness through the chemical time scales becoming more important.

Figure 5.5 shows the efficiency parameter versus the iteration number. The efficiency initially improves relatively linearly with iteration number, since the RSM iterations constantly stop at the edge of the model region. A large change is observed in objective function for the first iteration as a result of the starting point being on the edge of the model region, as opposed to in the center like everywhere else, allowing for a larger step to be taken and therefore more improvement to be made. As the algorithm approaches the maximum value, the solver can be seen to make larger improvements, which is caused by the proximity to the maximum. On the tenth iteration the function value can be seen to decrease, which is why the solver stopped and the maximum of the previous iteration is reported.

As the efficiency was improved by the statistically significant amount of 6.62% and the changes in the efficiency and temperature profiles are physically justifiable, changing the second stage pore diameter to 0.77mm will improve the performance of the porous radiant burner of study.

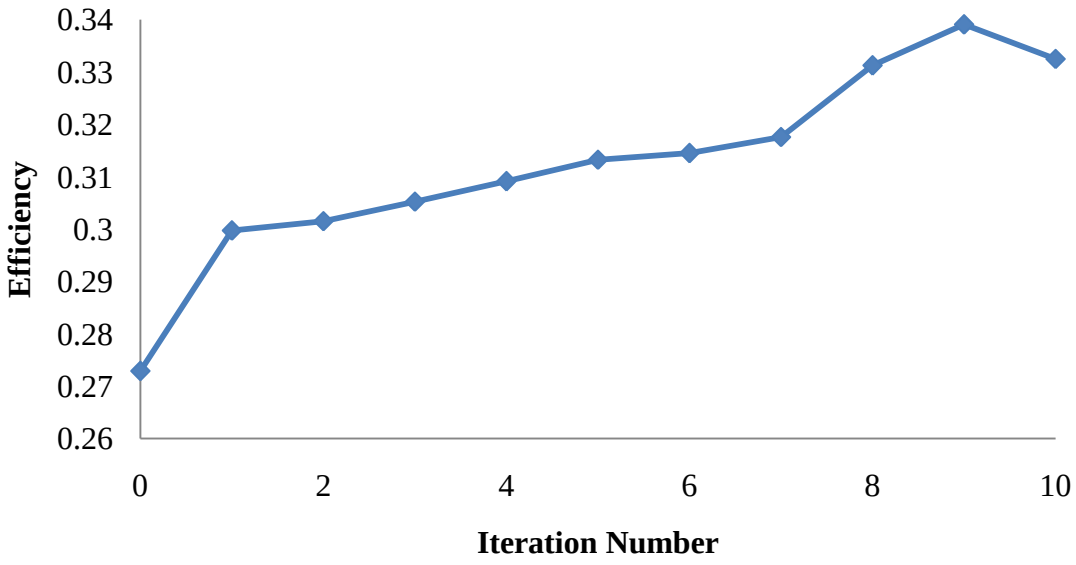


Figure 5.5 - Change in Efficiency with Iteration Number for $d_{p,2}$ Optimization

5.2.2 Stage Two Porosity

The second optimization was carried out on the second stage porosity. This parameter was chosen as porosity appears in the governing equations as well as the radiative extinction coefficient correlation. Also, this problem would be a good test case for demonstrating constrained optimization as the porosity cannot be too low or too high, so it was constrained between porosities of 0.865 and 0.95. These values were chosen because for porosities less than 0.865 the reaction zone was no longer within the burner, thus not being of interest in this study, and a porosity of 0.95 was the largest found in porous media combustion studies (Sathe, Peck and Tong 1990). The second stage was chosen for the reasons detailed in Section 5.2.1. The

initial value for the porosity was selected as the reference value given in Table 3.2, 0.87, and a grid spacing parameter was set to 0.005. Table 5.2 provides a summary of the starting point for this optimization, the function value for the reference case, and the constraints.

Table 5.2 - Initial Parameters for ε_2 Optimization

Parameter	Value
$\mathbf{x}^0$	0.87
γ	0.005
$F(\mathbf{x}^0)$	27.29%
Lower Bound	0.865
Upper Bound	0.95

As the reference value was close to one of the constraints, the initial point could not be in the center of the model region. Therefore, one point was selected to be smaller than the initial point, and therefore on the constraint, while the remaining points were selected for larger porosities. The first response surface, and function values at the desired points, seen in Figure 5.6, is a concave surface, or convex when the negative is applied. Therefore, a Newton step can be taken to the edge of the model region and then a new surface can be made centered about this new point.

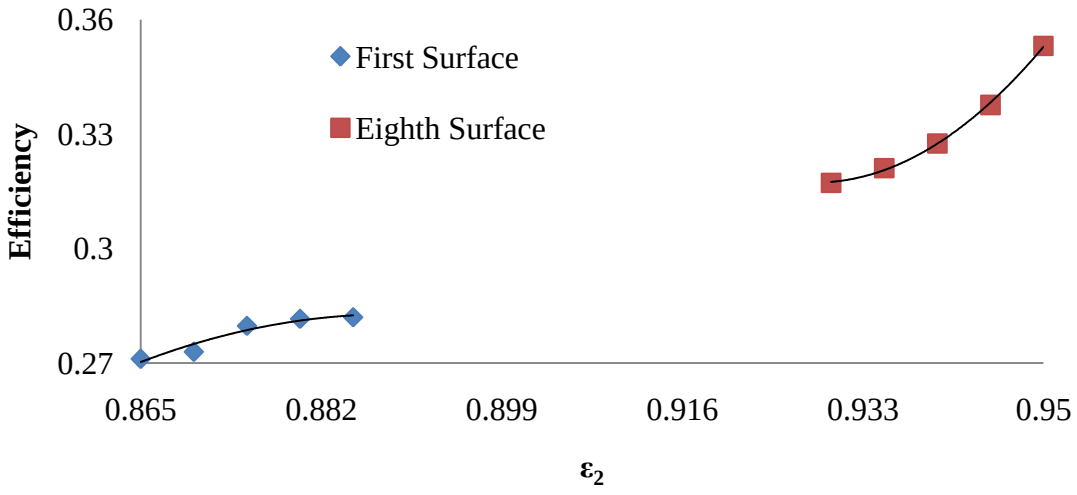


Figure 5.6 - First and Eighth Response Surface for ε_2 Optimization

The algorithm continues building surfaces and finding the minimum, as described in Chapter 4, until it reaches the eighth surface, also seen in shown in Figure 5.6. Here the maximum of the response surface is located on the boundary of the feasible region. Therefore the value of γ is decreased and a new response surface is generated, shown in Figure 5.7.

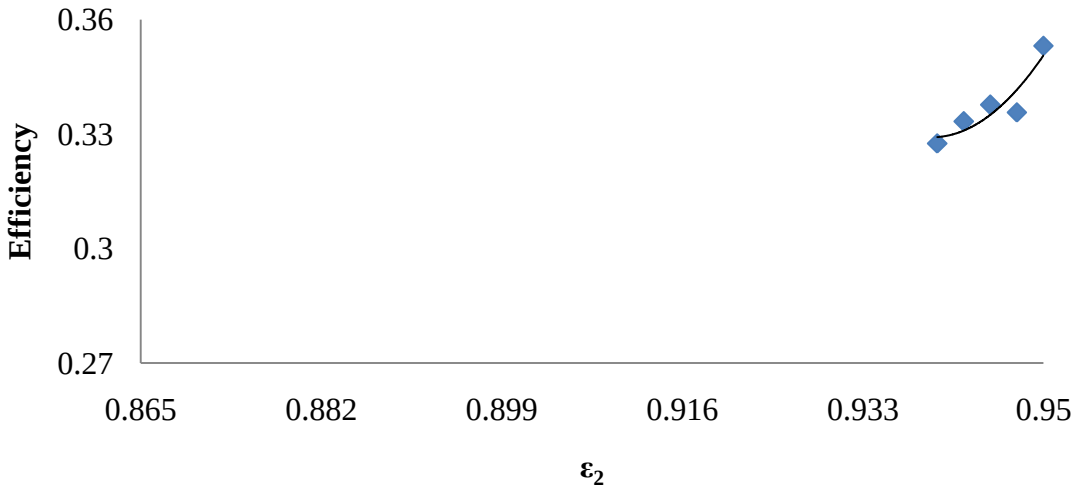


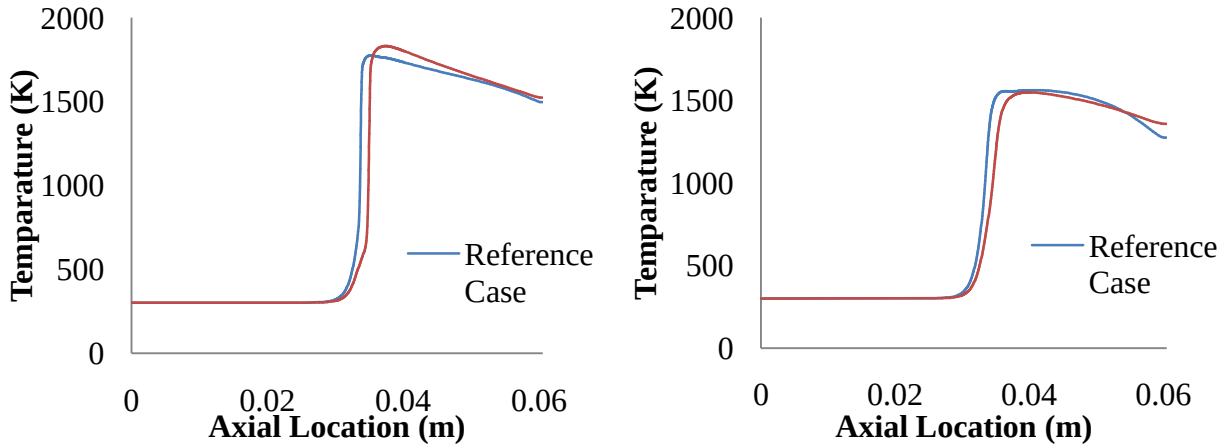
Figure 5.7 - Ninth Response Surface for ε_2 Optimization

Once again the maximum of the surface is on the constraint, so the algorithm chooses the same point as the previous iteration. Therefore, one of the stopping criteria has been met and the optimization is complete after 8 hours. The algorithm gives the maximum efficiency as 35.32% ± 0.68% with the second stage porosity being equal to 0.95. As with the pore diameter optimization, this is a statistically significant improvement as the total change in efficiency is 8.03%, which is larger than the amount of error in the final answer. Therefore, even in the worst case scenario for the error, an improvement has still been found in the performance of the porous radiant burner.

This answer also makes physical sense, as Eqn. (3.17) shows an increase in the porosity will cause a decrease in the extinction coefficient. A decrease in the extinction coefficient means

that the amount of radiative heat transfer is decreasing (if the coefficient is zero there is no scattering or absorption, hence no radiation). With a decrease in the amount of radiative heat transfer, conduction will become the dominant method of heat transfer within the solid, resulting in the temperature profile becoming more linear, and increasing the exit temperature value, resulting in an increase in efficiency. This is confirmed in

Figure 5.8 where the temperature profiles for the reference case are compared to the optimal solution. For the optimal solution the solid temperature profile linearly decreases to the exit value as one would expect for a conduction dominated problem. One would expect a true maximum to exist, even though one was not observed here due to the constraint. As Eqn. (3.2) shows, as the porosity increases the other forms of energy transfer in the gas phase become more important. Therefore, we would expect at some point the other forms of energy transfer would become more dominant than the convective heat transfer term, resulting in less energy being transferred into the solid causing the efficiency to start to drop.

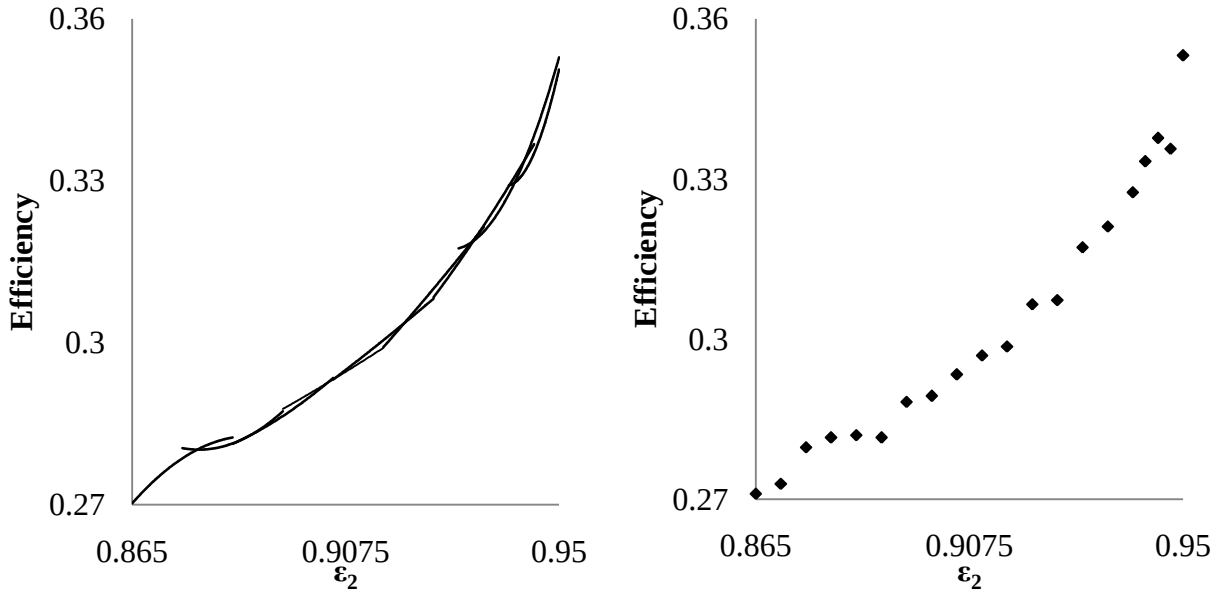


a) Gas Temperature Profile Comparison b) Solid Temperature Profile Comparison

Figure 5.8 - Reference vs. Optimal Temperature Profile for ε_2 Optimization

All of the response surfaces leading to the maximum value, as well as all the actual function values, can be seen in

Figure 5.9.



a) Response Surface b) Function Values

Figure 5.9 - Response Surfaces and Function Values for ε_2 Optimization

The objective function here is very smooth and has a quadratic shape. Once the algorithm has decreased the size of the model region it can be seen that some noise has been introduced into the objective function, resulting from the stiffness of the equation set. For this case the objective function did not become noisier as the solution was approached, however, as the solution was located on the constraint and the true maximum of the unconstrained function was not reached, we cannot make a concrete statement regarding the stiffness of the governing equations close to the optimum.

Plotting the change in objective function versus the iteration number, shown in Figure 5.10, we see that the change is relatively quadratic throughout. Once again a larger change in

objective function is seen at first due to a larger step being possible, as the initial point was not located at the center of the model region. For the final iteration, no change was observed because the solver wanted to leave the feasible region but was constrained from doing so, stopping in the same location and ending the algorithm.

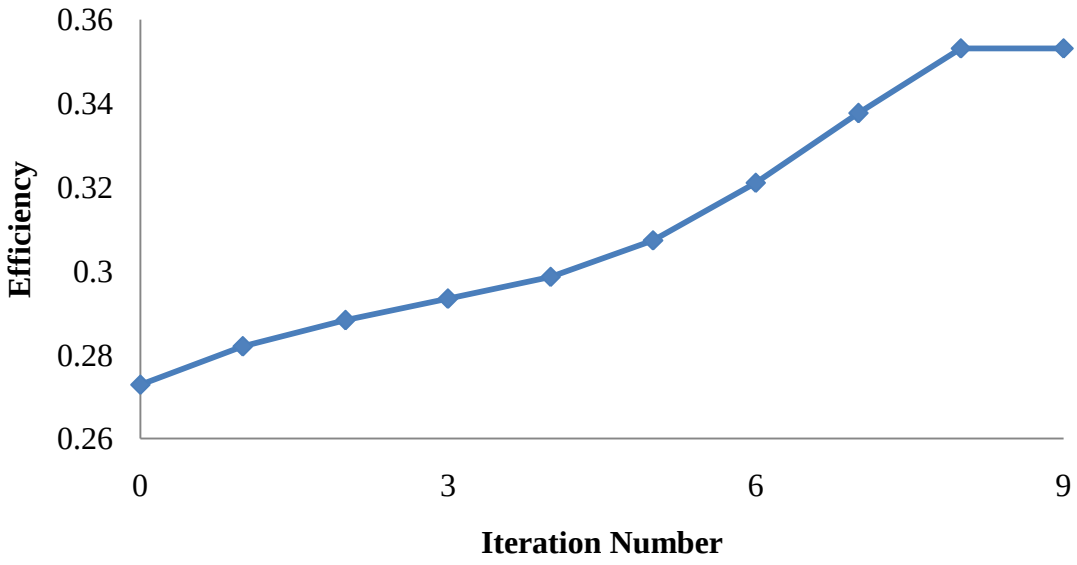


Figure 5.10 - Change in Efficiency with Iteration Number for ε_2 Optimization

As the radiant efficiency was improved by the statistically significant amount of 8.02% and the changes in the efficiency and temperature profiles are physically justifiable, changing the second stage porosity to 0.95 will improve the performance of the porous radiant burner of study.

5.3 Two Dimensional Study

After the successful completion of both one-dimensional optimizations, a two-dimensional case was studied. The design variables were the second stage pore diameter and the second stage porosity. These parameters were chosen to show that what was best for each in a one dimensional study may not be the best overall solution for a two dimensional optimization, highlighting the importance of considering non-linear interactions of the design variables and the

effect that they have on the radiant efficiency. As before, the problem is constrained between porosities of 0.865 and 0.95 but now the pore diameters are constrained between 0.69mm and 1.52mm because the flame front was found to be outside of the stable range for diameters below 0.69mm, which is beyond the scope of this study, and Eqns. (3.15) and (3.16) were derived for pore diameters less than 1.52mm, hence the top constraint. The initial values for the pore diameter and porosity were selected as 1.445mm and 0.88, and grid spacing parameters of 0.075 and 0.01 were chosen. Different values were chosen from the reference values as having the code start so close to the corner of the feasible region was undesirable. However, when considering improvements to the objective function, the reference values are used for comparison. Table 5.3 provides a summary of the starting point for this optimization, the function value for the reference case, and the constraints.

Table 5.3 - Initial Parameters for 2-D Optimization

Parameter	Value
$\mathbf{x}^0$	$(1.445\text{mm},0.88)^T$
γ_1	0.075
γ_2	0.01
$F(\mathbf{x}^0)$	27.29%
Lower Bound for $d_{p,2}$	0.69mm
Upper Bound for $d_{p,2}$	1.52mm
Lower Bound for ε_2	0.865
Upper Bound for ε_2	0.95

The first response surface, seen in Figure 5.11, contains a saddle point; since it lies in a direction that will increase the efficiency of the burner a Newton step is taken to the edge of the model region and then a new surface can be made centered about this new point. The algorithm continues building surfaces and finding the minimum, as described in Chapter 4, until it reaches the thirteenth surface, shown in Figure 5.12.

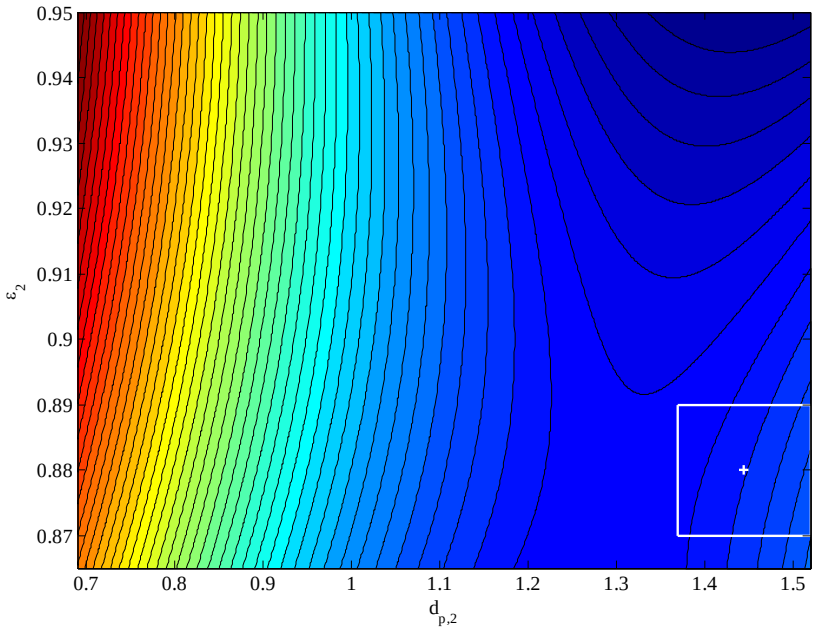


Figure 5.11 - First Response Surface for 2-D Optimization

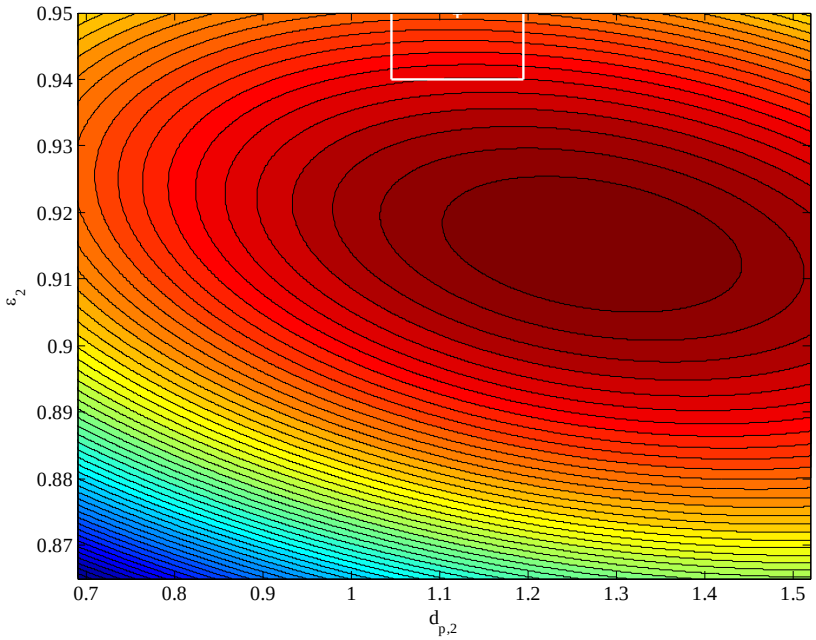


Figure 5.12 - Thirteenth Response Surface for 2-D Optimization

Here, after finding the minimum of the previous iteration to be on the edge of the feasible region and shrinking the model region size, the response surface contains a maximum value so the

steepest descent step is desired. However, this causes the solver to stop once again in the same place, meaning the generalized reduced gradient (GRG) method must now be used. Using the current point as the initial point for the one dimensional solver, and selecting the grid spacing parameter as half of the current value, the surface shown in Figure 5.13 is generated.

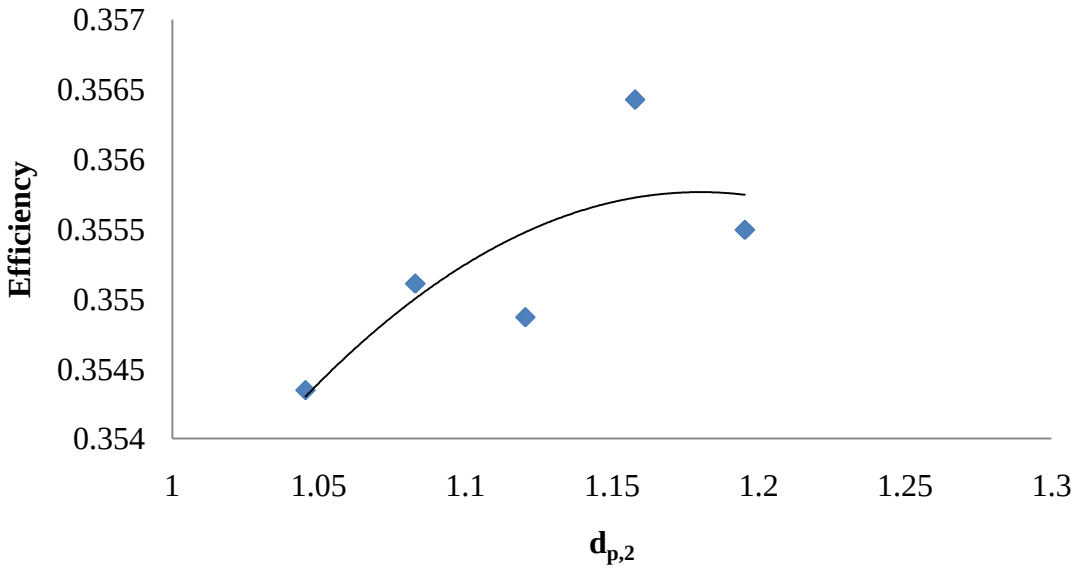


Figure 5.13 – First GRG Method Response Surface for 2-D Optimization, ε_2=0.95

This surface has a maximum value within the model region so the Newton's step is taken and the problem reverts to being a 2-D problem. The solver continues shrinking the model region, and alternating between two and one dimensions until the fourteenth surface, shown in Figure 5.14, is made. Here the minimum of the surface is within the model region; however it results in a decrease in efficiency. Therefore, one of the stopping criteria of the algorithm has been met and the solver stops after 27.5 hours of computation.

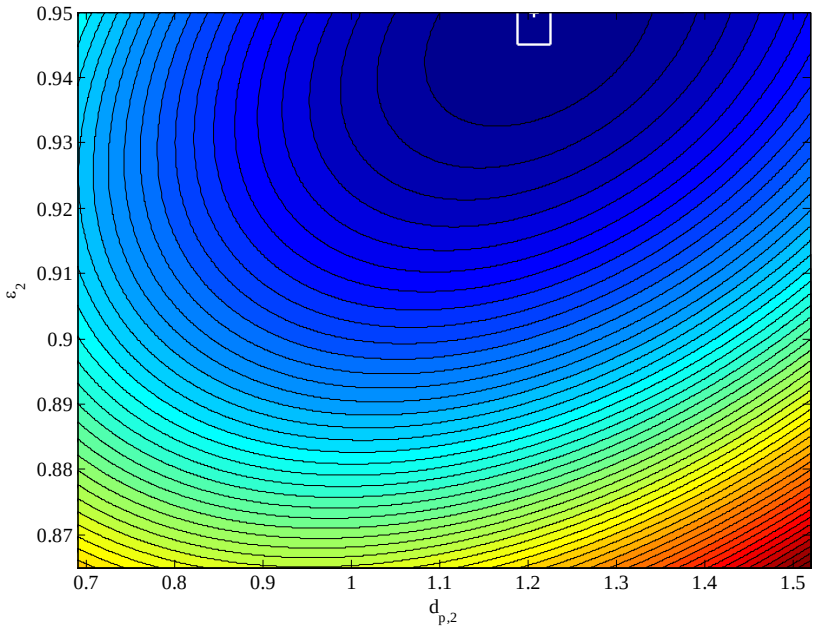
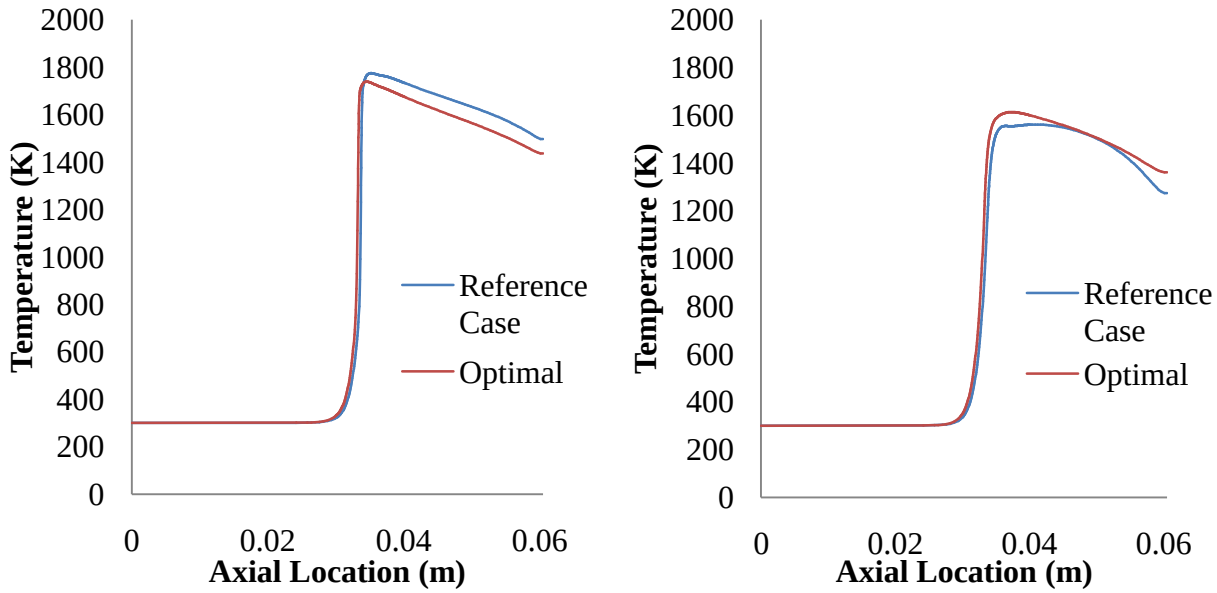


Figure 5.14 - Fifteenth Response Surface for 2-D Optimization

The algorithm gives the maximum efficiency as 35.56% ± 0.19% with the second stage pore diameter and porosity being equal to 1.21mm and 0.95 respectively. Once again this is a statistically significant improvement as the total change in efficiency is 8.27%, which is larger than the estimated amount of error in the final answer. Therefore, even in the worst case scenario for the error, an improvement has still been found in the performance of the porous radiant burner. This answer also makes physical sense for the same reasons explained in Sections 5.2.1 and 5.2.2. This is confirmed in

Figure 5.15 where the temperature profiles for the reference case are compared to the optimal solution. For the optimal solution the solid temperature profile linearly decreases to the exit value as one would expect for a conduction dominated problem and is very similar to the porosity solution.



a) Gas Temperature Profile Comparison b) Solid Temperature Profile Comparison

Figure 5.15 - Reference vs. Optimal Temperature Profile for 2-D Optimization

All of the response surfaces leading to the maximum value are included in Appendix E. Once the algorithm has decreased the size of the model region, it can be seen that some noise has been introduced into the objective function, resulting from the stiffness of the equation set. Similar to the pore diameter optimization, when the algorithm was close to the minimum value more noise was observed in the system of equations, leading to the belief that the stiffness increases when near an extreme point.

Plotting the change in objective function versus the iteration number, shown in Figure 5.16, shows that the objective function reduces quadratically throughout the optimization procedure, which we would expect from a second-order solver. This figure only contains the change in objective function between the two dimensional surfaces, ignoring the fact that a one dimensional optimization occurred between the twelfth and thirteenth iteration and two one dimensional steps were required between the thirteenth and fourteenth. Once again a larger change in objective function is seen at first due to a larger step being possible, as the reference

point was not located at the center of the model region. For the final iterations minimal changes were observed because the solver was close to the minimum value and the solution was located on the constraint and the algorithm was fine tuning along it. Figure 5.16 also shows the objective function value getting worse for the final iteration; hence the solver stopped and reported the previous iterations maximum as the answer.

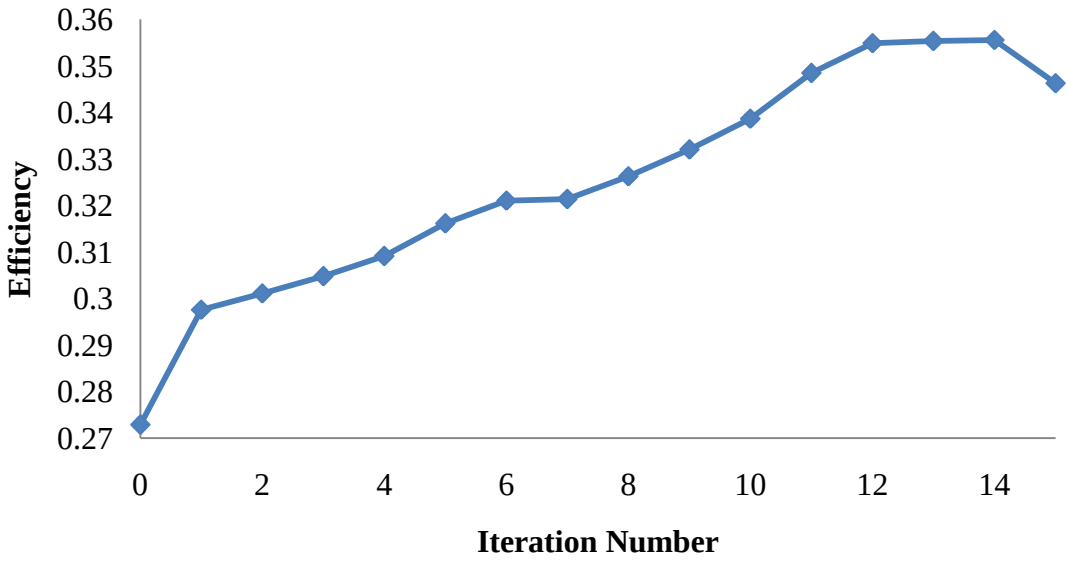


Figure 5.16 - Change in Efficiency with Iteration Number for 2-D Optimization

As the radiant efficiency was improved by the statistically significant amount of 8.27% and the changes in the efficiency and temperature profiles are physically justifiable, changing the second stage pore diameter to 1.21mm and the porosity to 0.95 will improve the performance of the porous radiant burner of study. This result also highlights the non-linear relationship between the design variables. For the one-dimensional optimization involving the pore diameter of the second stage, the algorithm recommended reducing it to a value of 0.77mm, when here it says to stop at a value of 1.21mm. Even though the algorithm has stopped on a constraint the GRG method also says to leave the pore diameter at a value of 1.21mm, as reducing it further would serve to make the objective function worse. Physically, since increasing the porosity and

decreasing the pore diameter both cause the extinction coefficient to decrease, a larger pore diameter than the one dimensional case is required to prevent the radiation term from becoming so small that the efficiency worsens. This difference in pore diameter values for the one and two-dimensional optimizations is the result of the non-linear interaction of the pore diameter with the porosity.

Chapter 6
Conclusions

6.1 Summary of Results

The goal of this thesis was to design an optimization algorithm for use on combustion devices and apply it to a porous radiant burner design. Three optimizations were carried out in attempts to improve the radiant efficiency of a porous radiant burner. From the nominal case a univariate optimization on the second stage pore diameter, $d_{p,2}$, showed an improvement of 6.62% ± 2.75% when the pore diameter was changed from 1.52mm to 0.77mm. A second univariate optimization was performed on the second stage porosity, ε_2, which showed an improvement of 8.03% ± 0.68% when the porosity was changed from 0.87 to 0.95. Lastly, a two dimensional optimization was carried out, allowing both of these parameters to change. An improvement of 8.27% ± 0.19% was observed when the pore diameter and porosity were changed to 1.21mm and 0.95 respectively. A summary of these results can be found in Table 6.1.

Table 6.1 - Summary of Optimizations

	$\mathbf{d_{p,2}}$	$\boldsymbol{\varepsilon_2}$	**2-D**
$\mathbf{x}^0$	1.52mm	0.87	$(1.52\text{mm},0.87)^T$
$F(\mathbf{x}^0)$	27.29%	27.29%	27.29%
$\mathbf{x}^*$	0.77mm	0.95	$(1.21\text{mm},0.95)^T$
$F(\mathbf{x}^*)$	33.91%	35.32%	35.56%
Error	2.75%	0.68%	0.19%

All optimizations performed led to statistically significant improvements, meaning that the improvement, $F(\mathbf{x}^0) - F(\mathbf{x}^*)$, was greater than the error estimate in $F(\mathbf{x}^*)$.

6.2 Benefits of Proposed Method

The optimization algorithm laid out in this thesis has many benefits over other optimization techniques. Trial-and-error methods, as well as parameter studies, ignore the non-linear interactions between the design variables, which were demonstrated to be important in this problem. The standard Newton's method struggles with calculation of the gradient and Hessian of the objective function, which are necessary to be able to find the minimum. The proposed method, however, is capable of correcting these problems. First, by using RSM, the difficulties with the stiffness of the governing equations are greatly reduced. RSM also presents analytically tractable estimates of the gradient vector and Hessian matrix to be solved analytically, removing the need to estimate those using finite differences, which is both problematic for stiff systems and also computationally expensive. When the stiffness of the system causes the objective function to become noisy, RSM smoothes the function, as it uses a model rather than the real function.

Using a quadratic fit to the data for generating the surfaces also has its benefits. When a modified Newton's method is applied to the function a single step can be taken towards the minimum. This avoids costly iterations, greatly reducing the computational time. A quadratic fit also requires less data than higher order polynomials, meaning that less function evaluations are required.

The use of the GRG method is another major benefit of the proposed algorithm. Here it provided minimal changes to the objective function as the solution on the boundary was already close to the minimal value. However, it is not difficult to imagine a problem where the optimal solution is far from where the iterate solution lands on the boundary. The GRG method allows

for these changes to be made, reporting the best answer for a problem rather than the solution where a constraint was met.

All of these benefits together lead to a robust algorithm for stiff systems, which is also computationally inexpensive compared to other methods. Moreover, this algorithm is generic enough that it can easily be applied to many types of combustion design problems, not just the porous radiant burner examined here.

6.3 Recommendations for Future Work

While the optimizations performed were successful and the algorithm can be used to optimize combustion devices there is still further research that can be carried out concerning the algorithm. This section will detail these extensions and describe why they are significant.

6.3.1 Relation between Pore Diameter and Porosity

In this research, it was assumed that the pore diameter and porosity were independent variables, as there was no discussion in the literature to suggest otherwise, but this seems like a counterintuitive result. Obviously, if the porosity is very large we cannot have large pores as well and if the porosity is small we cannot have small pores while maintaining a path through the solid. This means that a relationship must exist between these two variables. As it is possible for a single porosity to have a range of pore diameters however, this relationship will not be a simple function. This relationship would result in constraints for the one dimensional optimization cases, and would define the shape of the feasible region for the two dimensional case. This means that entirely different solutions could be observed due to the different constraints on the objective function. If a relationship between pore diameter and porosity were

to be researched and found, then the accuracy and feasibility of the optimization algorithm would greatly improve.

6.3.2 Other Design Variables

While good improvement was found using the second stage pore diameter and porosity, other design variables can be considered. Qiu and Hayden (2010) showed that the radiant efficiency of a porous radiant burner is a function of the equivalence ratio; therefore it could be used as a future design parameter in this algorithm. Other design parameters that could be used include the pore diameter and porosity of the first stage, the scattering albedo, the specific heat and density of the solid, and the lengths of each burner section. Each of these parameters should affect the efficiency of the burner. The pore diameter and porosity would alter the convection and radiation in the first stage of the burner for the reasons described in Section 5.2, leading to different preheating. Changing the scattering albedo, specific heat, and density will change the way that the solid conducts and radiates heat, while altering the lengths of the burner sections will lead to different flames being able to stabilize within the burner. All of these parameters are viable options that could lead to even greater improvements in the porous radiant burner design.

Another recommendation is to use more design variables. While this thesis only provided one and two dimensional optimizations, the algorithm is easily extendable to higher order problems. The non-linear interactions between the variables will be better accounted for by using more design variables. This means that the more design variables that are considered the better the system will be represented. This can lead to even greater improvements being made to the efficiency as the algorithm will have more that it can change to improve the performance.

6.3.3 Multi-Objective Optimizations

For this thesis the only objective considered was to maximize the radiant efficiency; however there are other objectives that can be considered. As emissions control is becoming of greater importance in industrial combustion, objectives such as minimizing NO_x or CO would be useful. Another objective could be to maximize the turn down ratio of the burner, which would make a single burner more desirable to the industry as it could be operated under more conditions. The algorithm could even be extended to deal with multi-objective problems. This could be done by altering the code to become a genetic algorithm, as used by Büche et al. (2001) or by using a weighted objective function. For a weighted objective function, each objective is assigned a weight based on its importance to the design engineer. This method results in a single solution, unlike the genetic algorithms in which a Pareto front is formed from many candidate solutions. Using a multi-objective approach, burners could be designed that would maximize radiant efficiency while minimizing pollutants, although with the current state of NO_x modelling this will still be difficult. Using multi-objective optimization would be very useful to the industrial combustion community as they could design the best burners for their process without compromising the environment.

6.3.4 Proximity to Optimum

In this research it was noted that close to the optimum, the noisiness of the objective function increased as a result of the stiffness. It is unknown whether this result was coincidental or a property of the stiffness, and determining so was beyond the scope of the research. Researching this would be very beneficial to the performance of the optimization algorithm. If it

is true, that when close to the optimum the stiffness increases, the algorithm could be adjusted to compensate for this stiffness, thus reducing the noise, leading to more accurate solutions.

6.3.5 Parallel Processing

Parallel processing is another recommendation that could greatly improve upon the algorithm presented in this thesis. In this study, the optimizations and calculations were carried out on a single processor. The response surface method is particularly amenable to speed up by parallelization, since each objective function evaluation used to construct the response surface can be carried out independently. This would reduce the computational time by approximately a factor of five for the one dimensional case and a factor of nine for the two dimensional case. Reducing the computational effort required to carry out the minimization means that more resources could be allocated towards more detailed chemical mechanisms and a higher degree of refinement, bringing the computational time back up to its current value. Both of these improvements would allow for more realistic and accurate representations of the system, meaning better results could be found.

6.3.6 Other Combustion Devices

The intention of this thesis was to present an optimization algorithm that would not only improve the design of a porous radiant burner, but could also be adapted for use on other combustion devices. Therefore, as a final recommendation this code should be used on other devices. Any combustion device could benefit from an optimization algorithm being used on its parameters to find its best operating point. By improving these burners we can get more useful energy output, save money, or reduce the impact combustion has on the planet, by reducing pollutants.

References

Barra, A. J., G. Diepvens, J. L. Ellzey, and M. R. Henneke. "Numerical study of the effects of material properties on flame stabilization in a porous burner." *Combustion and Flame 134*, 2003: 369-379.

Büche, D., P. Stoll, and P. Koumoutsakos. "An evolutionary algorithm for multi-objective optimization of combustion processes." *Center for Turbulence Research Annual Research Briefs*, 2001: 231-239.

Buckmaster, J., and T. Takeno. "Blow-off and flashback of an excess enthalpy flame." *Combustion and Science Technology 25*, 1981: 153-158.

Catalano, L. A., A. Dadone, D. Manodoro, and A. Saponaro. "Efficient design optimization of duct-burners for combined-cycle and cogenerative plants." *Engineering Optimization 38*, 2006: 801-820.

Correa, C. D., and P. J. Smith. "Optimization of ehtylene furnace operations." *1998 AiChe Annual General Meeting.* Miami Beach, 1998.

Echigo, R, Y. Yoshizawa, K. Hanamura, and T. Tominura. "Analytical and experimental studies on radiative propagation in porous media with internal heat generation." *Proceedings of the 8th International Heat Transfer Conference 2.* 1986. 827-832.

Frenklach, M., et al. 1994. http://www.me.berkley.edu/gri_mech/.

Garfinkel, D., C. B. Marbach, and N. Z. Shapiro. "Stiff differential equations." *Annual Review of Biophysics and Bioengineering 6*, 1977: 525-542.

Gemmen, R. S. "Oxidation of low calorific value gases - Applying optimization techniques to combustor design." *1998 International Joint ASME/EPRI Power Generation Conference.* Baltimore, 1998.

Gill, P. E., W. Murray, and M. H. Wright. *Practical Optimization.* London: Academic Press, 1986.

Goodwin, D. G. "An open-source, extensible software suite for CVD process simulatio." *Proceedings of CVD XVI and EuroCVD Fourteen*, 2003: 155-162.

Hardesty, D.R., and F. J. Weinberg. "Burners producing large excess enthalpies." *Combustion Science and Technmologies 8*, 1974: 201-214.

Hendricks, T. J., and J. R. Howell. "Absorption/Scattering coefficients and scattering phase function in reticulated porous ceramics." *Journal of Heat Transfer 118*, 1996: 79-87.

Henneke, M. R. *Simulation of transient combustion within porous inert media (PhD Thesis).* University of Texas, 1998.

Howell, J. R., M. J. Hall, and J. L. Ellzey. "Combustion of hydrocarbon fuels within porous inert media." *Progress in Energy and Combustion Science*, 1996: 121-145.

Hsu, P. F., and J. R. Howell. "Measurement of thermal conductivity and optical properties of porous partially stabilized zirconia." *Experimental Heat Transfer 5*, 1992: 293-313.

Hsu, P. F., and R. D. Matthews. "The necessity of using detailed kinetics in models for premixed combustion within porous inert media." *Combustion and Flame 93*, 1993: 457-467.

Hsu, P. F., J. R. Howell, and R. D. Matthews. "A numerical investigation of premixed combustion within porous inert media." *Journal of Heat Transfer 115(3)*, 1993: 744-750.

Khanna, V., R. Goel, and J. L. Ellzey. "Measurements of emissions and radiation for methane combustion within a porous medium burner." *Combustion Science and Technology 99*, 1994: 133-142.

Kotani, Y., and T. Takeno. "An experimental study on stability and combustion characteristics of an excess enthalpy flame." *19th Symposium (International) on Combustion.* The Combustion Institute, 1982. 1503-1509.

Krazakov, A., and M. Frenklach. 1994. http://www.me.berkeley.edu/drm/.

Myers, R. H., D. C. Montgomery, and C. M. Anderson-Cook. *Response surface methodology, 3rd ed.* Hoboken: Wiley, 2009.

Nait-Ali, B., K. Haberko, H. Vesteghem, J. Absi, and D. S. Smith. "Preparation and thermal conductivity characterisation of highly porous ceramics comparison between experimental results, analytical calculations and numerical simulations." *Journal of the European Ceramic Society 27*, 2007: 1345-1350.

NIST/SEMATECH. 2010. http://www.itl.nist.gov/div898/handbook/.

Nocedal, J., and S. J. Wright. *Numerical Optimization, 2nd ed.* New York: Springer, 2006.

Qiu, K., and S. Hayden. "Premixed gas combustion in a porous medium burner system." *CICS 2010 Spring Technical Meeting.* Ottawa, 2010. 403-408.

Randrianalisoa, J., Y. Bréchet, and D. Baillis. "Materials selection for optimal design of a porous radiant burner for environmentally driven requirements." *Advanced Engineering Materials 11*, 2009: 1049-1056.

Sathe, S. B., R. E. Peck, and T. W. Tong. "Flame stabilization and multimode heat transfer in inert porous media: A numerical study ." *Combustion Science and Technology 70*, 1990: 93-109.

Siegel, R., and J. Howell. *Thermal radiation heat transfer 4ed.* New York: Taylor & Francis, 2002.

Smith, G. P., et al. 1999. http://www.me.berkeley.edu/gri_mech/.

Smith, Philip J., William A. Sowa, and Paul O. Hedman. "Furnace design using comprehensive combustion models." *Combustion and Flame 79*, 1990: 111-121.

Takeno, T., and K. Hase. "Effects of solid length and heat loss on an excess enthalpy flame." *Combustion and Science Technology 31*, 1983: 207-215.

Takeno, T., and K. Sato. "An excess enthalpy flame theory." *Combustion and Science Technology 20*, 1979: 73-84.

Tong., T. W., W. Q. Lin, and R. E. Peck. "Radiative heat transfer in porous media with spatially-dependant heat generation." *International Communications on Heat and Mass Transer 14*, 1987: 627-637.

Vafai, K., ed. *Handbook of porous media 2ed.* Boca Raton: Taylor & Francis Group, 2005.

Weinberg, F.J. "Combustion temperatures: The future." *Nature 233*, 1971: 239-241.

Younis, L. B., and R. Viskanta. "Experimental determination of the volumetric heat transfer coefficient between stream of air and ceramic foam." *Internation Journal of Heat and Mass Transfer 36(6)*, 1993: 1425-1434.

Appendix A: DRM19 Reaction Mechanism

The DRM19 reaction mechanism is presented below in CHEMKIN format. The reaction is presented on the left while the values for the pre-exponential steric factor, A_0, the Arhenius temperature exponent, β, and the activation energy, E_A (in cal/mol), from Eqn. (3.9) are presented on the right. For three body reactions and other special reactions, the efficiencies and other information is presented below the relevant reaction.

```
!<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><!
!  Reduced version of GRI-MECH 1.2. 19 species ( + N2, AR); 84 reactions. !
!                                 PennState Dec, 1994                      !
!<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><!
ELEMENTS
O  H  C  N  AR
END
SPECIES
H2      H       O       O2      OH      H2O     HO2
CH2     CH2(S)  CH3     CH4     CO      CO2     HCO
CH2O    CH3O    C2H4    C2H5    C2H6
N2      AR
END
REACTIONS
O+H+M<=>OH+M                             5.000E+17   -1.000      0.00
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
O+H2<=>H+OH                              5.000E+04    2.670   6290.00
O+HO2<=>OH+O2                            2.000E+13    0.000      0.00
O+CH2<=>H+HCO                            8.000E+13    0.000      0.00
O+CH2(S)<=>H+HCO                         1.500E+13    0.000      0.00
O+CH3<=>H+CH2O                           8.430E+13    0.000      0.00
O+CH4<=>OH+CH3                           1.020E+09    1.500   8600.00
O+CO+M<=>CO2+M                           6.020E+14    0.000   3000.00
H2/2.00/ O2/6.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/3.50/ C2H6/3.00/ AR/0.50/
O+HCO<=>OH+CO                            3.000E+13    0.000      0.00
O+HCO<=>H+CO2                            3.000E+13    0.000      0.00
O+CH2O<=>OH+HCO                          3.900E+13    0.000   3540.00
O+C2H4<=>CH3+HCO                         1.920E+07    1.830    220.00
O+C2H5<=>CH3+CH2O                        1.320E+14    0.000      0.00
O+C2H6<=>OH+C2H5                         8.980E+07    1.920   5690.00
O2+CO<=>O+CO2                            2.500E+12    0.000  47800.00
O2+CH2O<=>HO2+HCO                        1.000E+14    0.000  40000.00
H+O2+M<=>HO2+M                           2.800E+18   -0.860      0.00
O2/0.00/ H2O/0.00/ CO/0.75/ CO2/1.50/ C2H6/1.50/ N2/0.00/ AR/0.00/
H+2O2<=>HO2+O2                           3.000E+20   -1.720      0.00
H+O2+H2O<=>HO2+H2O                       9.380E+18   -0.760      0.00
H+O2+N2<=>HO2+N2                         3.750E+20   -1.720      0.00
H+O2+AR<=>HO2+AR                         7.000E+17   -0.800      0.00
H+O2<=>O+OH                              8.300E+13    0.000  14413.00
2H+M<=>H2+M                              1.000E+18   -1.000      0.00
H2/0.00/ H2O/0.00/ CH4/2.00/ CO2/0.00/ C2H6/3.00/ AR/0.63/
2H+H2<=>2H2                              9.000E+16   -0.600      0.00
2H+H2O<=>H2+H2O                          6.000E+19   -1.250      0.00
2H+CO2<=>H2+CO2                          5.500E+20   -2.000      0.00
```

```
H+OH+M<=>H2O+M                            2.200E+22   -2.000      0.00
H2/0.73/ H2O/3.65/ CH4/2.00/ C2H6/3.00/ AR/0.38/
H+HO2<=>O2+H2                             2.800E+13    0.000   1068.00
H+HO2<=>2OH                               1.340E+14    0.000    635.00
H+CH2(+M)<=>CH3(+M)                       2.500E+16   -0.800      0.00
     LOW  /  3.200E+27   -3.140   1230.00/
     TROE/  0.6800   78.00  1995.00  5590.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+CH3(+M)<=>CH4(+M)                       1.270E+16   -0.630    383.00
     LOW  /  2.477E+33   -4.760   2440.00/
     TROE/  0.7830   74.00  2941.00  6964.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+CH4<=>CH3+H2                            6.600E+08    1.620  10840.00
H+HCO(+M)<=>CH2O(+M)                      1.090E+12    0.480   -260.00
     LOW  /  1.350E+24   -2.570   1425.00/
     TROE/  0.7824  271.00  2755.00  6570.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+HCO<=>H2+CO                             7.340E+13    0.000      0.00
H+CH2O(+M)<=>CH3O(+M)                     5.400E+11    0.454   2600.00
     LOW  /  2.200E+30   -4.800   5560.00/
     TROE/  0.7580   94.00  1555.00  4200.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/
H+CH2O<=>HCO+H2                           2.300E+10    1.050   3275.00
H+CH3O<=>OH+CH3                           3.200E+13    0.000      0.00
H+C2H4(+M)<=>C2H5(+M)                     1.080E+12    0.454   1820.00
     LOW  /  1.200E+42   -7.620   6970.00/
     TROE/  0.9753  210.00   984.00  4374.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+C2H5(+M)<=>C2H6(+M)                     5.210E+17   -0.990   1580.00
     LOW  /  1.990E+41   -7.080   6685.00/
     TROE/  0.8422  125.00  2219.00  6882.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
H+C2H6<=>C2H5+H2                          1.150E+08    1.900   7530.00
H2+CO(+M)<=>CH2O(+M)                      4.300E+07    1.500  79600.00
     LOW  /  5.070E+27   -3.420  84350.00/
     TROE/  0.9320  197.00  1540.00 10300.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
OH+H2<=>H+H2O                             2.160E+08    1.510   3430.00
2OH<=>O+H2O                               3.570E+04    2.400  -2110.00
OH+HO2<=>O2+H2O                           2.900E+13    0.000   -500.00
OH+CH2<=>H+CH2O                           2.000E+13    0.000      0.00
OH+CH2(S)<=>H+CH2O                        3.000E+13    0.000      0.00
OH+CH3<=>CH2+H2O                          5.600E+07    1.600   5420.00
OH+CH3<=>CH2(S)+H2O                       2.501E+13    0.000      0.00
OH+CH4<=>CH3+H2O                          1.000E+08    1.600   3120.00
OH+CO<=>H+CO2                             4.760E+07    1.228     70.00
OH+HCO<=>H2O+CO                           5.000E+13    0.000      0.00
OH+CH2O<=>HCO+H2O                         3.430E+09    1.180   -447.00
OH+C2H6<=>C2H5+H2O                        3.540E+06    2.120    870.00
HO2+CH2<=>OH+CH2O                         2.000E+13    0.000      0.00
HO2+CH3<=>O2+CH4                          1.000E+12    0.000      0.00
HO2+CH3<=>OH+CH3O                         2.000E+13    0.000      0.00
HO2+CO<=>OH+CO2                           1.500E+14    0.000  23600.00
CH2+O2<=>OH+HCO                           1.320E+13    0.000   1500.00
CH2+H2<=>H+CH3                            5.000E+05    2.000   7230.00
CH2+CH3<=>H+C2H4                          4.000E+13    0.000      0.00
CH2+CH4<=>2CH3                            2.460E+06    2.000   8270.00
```

```
CH2(S)+N2<=>CH2+N2                        1.500E+13    0.000    600.00
CH2(S)+AR<=>CH2+AR                        9.000E+12    0.000    600.00
CH2(S)+O2<=>H+OH+CO                       2.800E+13    0.000      0.00
CH2(S)+O2<=>CO+H2O                        1.200E+13    0.000      0.00
CH2(S)+H2<=>CH3+H                         7.000E+13    0.000      0.00
CH2(S)+H2O<=>CH2+H2O                      3.000E+13    0.000      0.00
CH2(S)+CH3<=>H+C2H4                       1.200E+13    0.000   -570.00
CH2(S)+CH4<=>2CH3                         1.600E+13    0.000   -570.00
CH2(S)+CO<=>CH2+CO                        9.000E+12    0.000      0.00
CH2(S)+CO2<=>CH2+CO2                      7.000E+12    0.000      0.00
CH2(S)+CO2<=>CO+CH2O                      1.400E+13    0.000      0.00
CH3+O2<=>O+CH3O                           2.675E+13    0.000  28800.00
CH3+O2<=>OH+CH2O                          3.600E+10    0.000   8940.00
2CH3(+M)<=>C2H6(+M)                       2.120E+16   -0.970    620.00
     LOW  /  1.770E+50   -9.670   6220.00/
     TROE/  0.5325  151.00  1038.00  4970.00 /
H2/2.00/ H2O/6.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/ AR/0.70/
2CH3<=>H+C2H5                             4.990E+12    0.100  10600.00
CH3+HCO<=>CH4+CO                          2.648E+13    0.000      0.00
CH3+CH2O<=>HCO+CH4                        3.320E+03    2.810   5860.00
CH3+C2H6<=>C2H5+CH4                       6.140E+06    1.740  10450.00
HCO+H2O<=>H+CO+H2O                        2.244E+18   -1.000  17000.00
HCO+M<=>H+CO+M                            1.870E+17   -1.000  17000.00
H2/2.00/ H2O/0.00/ CH4/2.00/ CO/1.50/ CO2/2.00/ C2H6/3.00/
HCO+O2<=>HO2+CO                           7.600E+12    0.000    400.00
CH3O+O2<=>HO2+CH2O                        4.280E-13    7.600  -3530.00
C2H5+O2<=>HO2+C2H4                        8.400E+11    0.000   3875.00
END
```

Appendix B: Sensitivity Analysis

B.1 Introduction

A sensitivity analysis was performed on the objective function for two reasons: first, to gain an insight into the importance of each of the correlated parameters and; second to justify neglecting the effect that porosity has on the thermal conductivity. The sensitivity of a function to a variable is calculated by

$$x\left|\frac{\mathrm{df}}{\mathrm{d}x}\right| \tag{B.1}$$

where f and x are the function and variable of interest. This measurement allows us to see how important an effect the variable x has on the function f. If the sensitivity is small then x is negligible to the function output, and if x is large it dominates it.

A major problem in estimating the sensitivities is that finite-difference estimates are inaccurate when the governing equations are stiff, as discussed in Section 4.3. Instead we use a variation of the RSM method discussed in Section 4.4. Eleven equally spaced points were sampled from the nominal value of the desired parameter to the optimal value, followed by fitting a surface to the data. The slope of the surface was then used for the derivative calculation. For the case of the extinction coefficient, only five points were used as the combustion solver encountered stability issues, as a result of flash-back, for increased values of the extinction coefficient. This stability issue does not occur during the optimization as the other parameters are allowed to change along with it, stabilizing the solution.

B.2 Objective Function Sensitivity to Thermal Conductivity, Extinction Coefficient, and Volumetric Heat Transfer Coefficient

The first stage of the sensitivity analysis was to calculate the sensitivity thermal conductivity, radiative extinction coefficient, and the convection heat transfer coefficient. Figures B.1 – B.3 show the sampled points and fitted curves used for estimating the sensitivities of these three parameters.

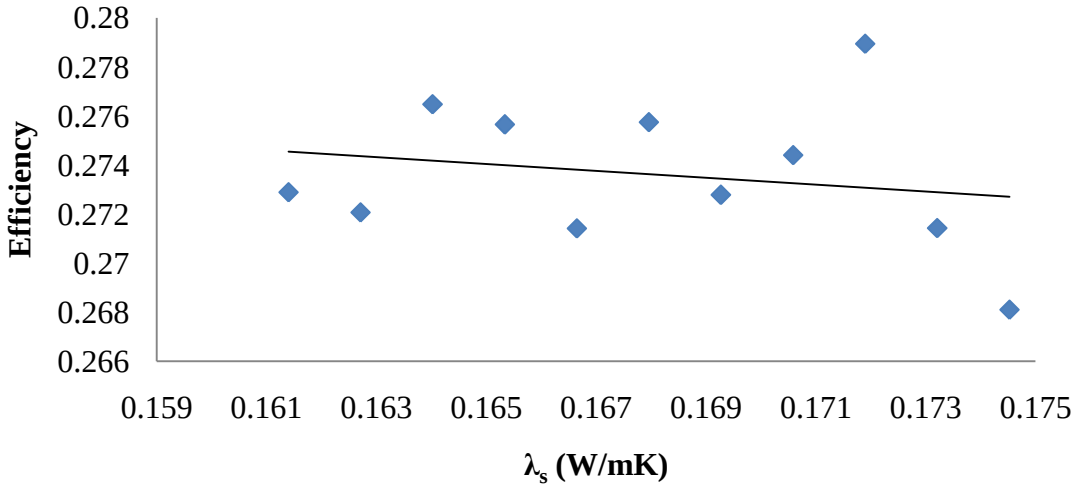


Figure B.1 – Thermal Conductivity Sensitivity Study

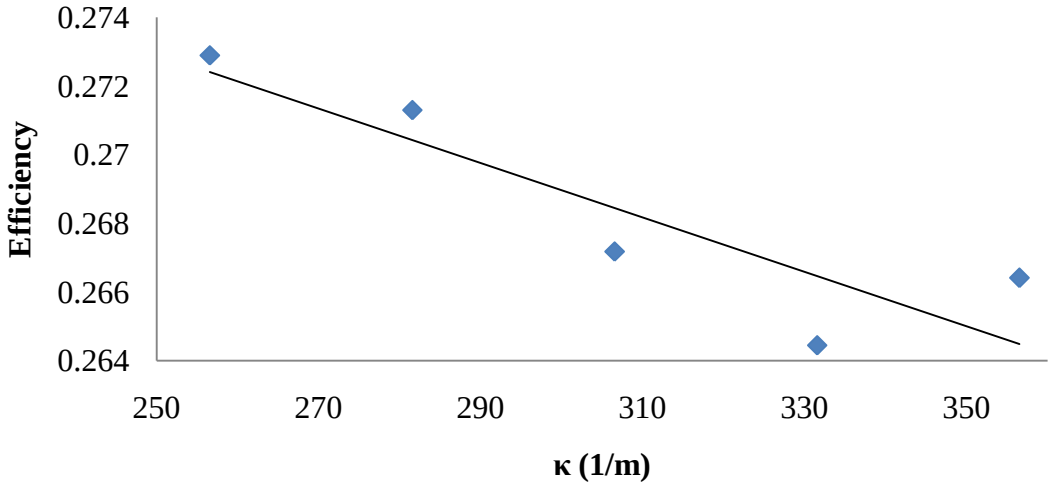


Figure B.2 – Radiative Extinction Coefficient Sensitivity Study

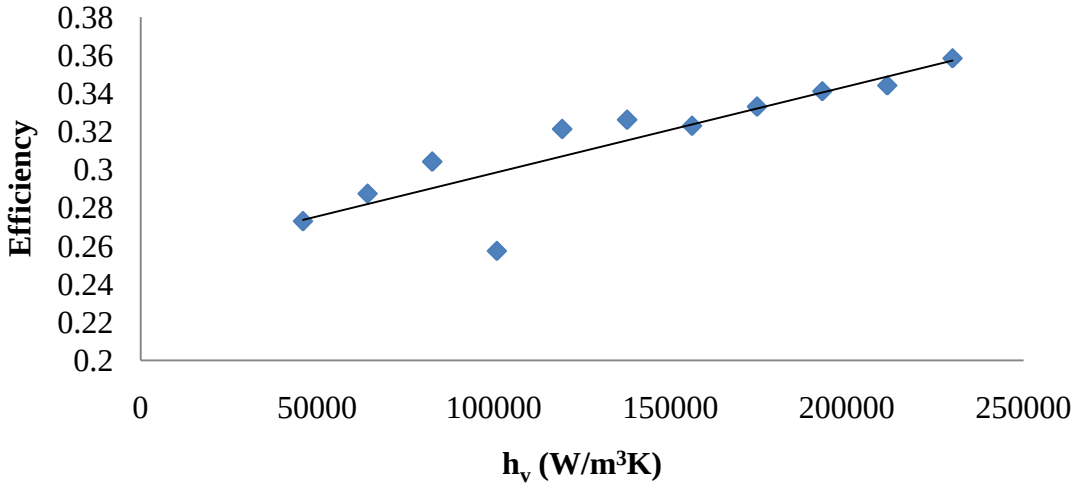


Figure B.3 – Convection Heat Transfer Coefficient Sensitivity Study

Using these curve fits and the nominal value for each parameter the sensitivity of the objective function to the thermal conductivity, the radiative extinction coefficient, and the convection heat transfer coefficient is found to be 0.0226, 0.0203, and 0.0209 respectively. Here we see that the objective function is equally as sensitive to all of the parameters.

B.3 Objective Function Sensitivity to Porosity

The next stage of the sensitivity analysis is to determine the effect that porosity has on the objective function, when acting through the thermal conductivity. If this value is small then we are justified in assuming that porosity is negligible in the thermal conductivity correlation. We start by finding the sensitivity of the porosity on the thermal conductivity. As the correlation presented by Hsu and Howell (1992) does not contain a porosity term we turn to the work of Nait-Ali et al. (2007) to estimate the slope of the curve in the region of interest, 80-100% porosity. Using this slope and the nominal value of the porosity the sensitivity of the thermal conductivity to the porosity is calculated as 0.0457W/mK.

Now that we know the sensitivity of the thermal conductivity to the porosity we can calculate the objective function's sensitivity to the porosity, when acting through the thermal conductivity. This is done using the chain rule by

$$\varepsilon \left| \frac{dF}{d\lambda_s} \frac{d\lambda_s}{d\varepsilon} \right| \tag{B.2}$$

resulting in a sensitivity of calculated as 0.0064. The objective function is not very sensitive to the porosity, evidenced by the sensitivity being an order of magnitude less than the sensitivities of the correlated parameters. As a result it is safe to use a correlation for thermal conductivity that does not contain a porosity effect, such as the one presented by Hsu and Howell (1992).

Appendix C: Alterations to Cantera Code

This section contains the Cantera files that were altered in order to add the solid phase to the existing combustion program. Each of the four following sections contains one of the four altered files. All additions and alterations are commented and bolded to stand out.

C.1 OneDim.cpp

```
#ifdef WIN32
#pragma warning(disable:4786)
#pragma warning(disable:4503)
#endif

#include "MultiJac.h"
#include "MultiNewton.h"
#include "OneDim.h"

#include "../ctml.h"
using namespace ctml;

namespace Cantera {
       int dosolid=0;  //variable added to tell algorithm when solid phase must be solved.
                                    //0 for no 1 for yes

    /**
     * Default constructor. Create an empty object.
     */
    OneDim::OneDim()
        : m_tmin(1.0e-16), m_tmax(10.0), m_tfactor(0.5),
          m_jac(0), m_newt(0),
          m_rdt(0.0), m_jac_ok(false),
          m_nd(0), m_bw(0), m_size(0),
          m_init(false),
          m_ss_jac_age(10), m_ts_jac_age(20),
          m_nevals(0), m_evaltime(0.0)
    {
      //writelog("OneDim default constructor\n");
        m_newt = new MultiNewton(1);
        //m_solve_time = 0.0;
    }


    /**
     * Construct a OneDim container for the domains pointed at by the
     * input vector of pointers.
    */
    OneDim::OneDim(vector<Domain1D*> domains) :
        m_tmin(1.0e-16), m_tmax(10.0), m_tfactor(0.5),
        m_jac(0), m_newt(0),
       m_rdt(0.0), m_jac_ok(false),
       m_nd(0), m_bw(0), m_size(0),
       m_init(false),
       m_ss_jac_age(10), m_ts_jac_age(20),
       m_nevals(0), m_evaltime(0.0)
    {
      //writelog("OneDim constructor\n");

        // create a Newton iterator, and add each domain.
        m_newt = new MultiNewton(1);
        int nd = static_cast<int>(domains.size());
        int i;
        for (i = 0; i < nd; i++) {
            addDomain(domains[i]);
        }
```

```
    init();
    resize();
}

int OneDim::domainIndex(string name) {
    for (int n = 0; n < m_nd; n++) {
        if (domain(n).id() == name) return n;
    }
    throw CanteraError("OneDim::domainIndex","no domain named >>"+name+"<<");
}


/**
 * Domains are added left-to-right.
 */
void OneDim::addDomain(Domain1D* d) {

    // if 'd' is not the first domain, link it to the last domain
    // added (the rightmost one)
    int n = static_cast<int>(m_dom.size());
    if (n > 0) m_dom.back()->append(d);

    // every other domain is a connector
    if (2*(n/2) == n)
        m_connect.push_back(d);
    else
        m_bulk.push_back(d);

    // add it also to the global domain list, and set its
    // container and position
    m_dom.push_back(d);
    d->setContainer(this, m_nd);
    m_nd++;
    resize();
}


OneDim::~OneDim() {
    delete m_jac;
    delete m_newt;
}

MultiJac& OneDim::jacobian() { return *m_jac; }
MultiNewton& OneDim::newton() { return *m_newt; }

void OneDim::writeStats() {
    saveStats();
    char buf[100];
    sprintf(buf,"\nStatistics:\n\n Grid   Functions   Time      Jacobians   Time \n");
    writelog(buf);
    int n = m_gridpts.size();
    for (int i = 0; i < n; i++) {
        sprintf(buf,"%5i   %5i    %9.4f    %5i    %9.4f \n",
            m_gridpts[i], m_funcEvals[i], m_funcElapsed[i],
            m_jacEvals[i], m_jacElapsed[i]);
        writelog(buf);
    }
}


/**
 * Save statistics on function and Jacobiab evaulation, and reset
 * the counters. Statistics are saved only if the number of
 * Jacobian evaluations is greater than zero. The statistics saved
 * are
 *
 *    - number of grid points
 *    - number of Jacobian evaluations
 *    - CPU time spent evaluating Jacobians
 *    - number of non-Jacobian function evaluations
 *    - CPU time spent evaluating functions
```

```
 */
void OneDim::saveStats() {
    if (m_jac) {
        int nev = m_jac->nEvals();
        if (nev > 0 && m_nevals > 0) {
            m_gridpts.push_back(m_pts);
            m_jacEvals.push_back(m_jac->nEvals());
            m_jacElapsed.push_back(m_jac->elapsedTime());
            m_funcEvals.push_back(m_nevals);
            m_nevals = 0;
            m_funcElapsed.push_back(m_evaltime);
            m_evaltime = 0.0;
        }
    }
}


/**
 * Call after one or more grids has been refined.
 */
void OneDim::resize() {
    int i;
    m_bw = 0;
    vector_int nvars, loc;
    int lc = 0;

    // save the statistics for the last grid
    saveStats();
    m_pts = 0;
    for (i = 0; i < m_nd; i++) {
        Domain1D* d = m_dom[i];

        int np = d->nPoints();
        int nv = d->nComponents();
        for (int n = 0; n < np; n++) {
            nvars.push_back(nv);
            loc.push_back(lc);
            lc += nv;
            m_pts++;
        }

        // update the Jacobian bandwidth
        int bw1, bw2 = 0;

        // bandwidth of the local block
        bw1 = 2*d->nComponents() - 1;

        // bandwidth of the block coupling the first point of this
        // domain to the last point of the previous domain
        if (i > 0) {
            bw2 = d->nComponents() + m_dom[i-1]->nComponents() - 1;
        }
        if (bw1 > m_bw) m_bw = bw1;
        if (bw2 > m_bw) m_bw = bw2;

        m_size = d->loc() + d->size();
    }
    m_nvars = nvars;
    m_loc = loc;

    m_newt->resize(size());
    m_mask.resize(size());

    // delete the current Jacobian evaluator and create a new one
    delete m_jac;
    m_jac = new MultiJac(*this);
    m_jac_ok = false;

    for (i = 0; i < m_nd; i++)
        m_dom[i]->setJac(m_jac);
}
```

```
int OneDim::solve(doublereal* x, doublereal* xnew, int loglevel) {
           dosolid=1;
    if (!m_jac_ok) {
        eval(-1, x, xnew, 0.0, 0);
        m_jac->eval(x, xnew, 0.0);
        m_jac->updateTransient(m_rdt, DATA_PTR(m_mask));
        m_jac_ok = true;
    }
    int m = m_newt->solve(x, xnew, *this, *m_jac, loglevel);
    return m;
}

void OneDim::evalSSJacobian(doublereal* x, doublereal* xnew) {
    doublereal rdt_save = m_rdt;
    m_jac_ok = false;
    setSteadyMode();
    eval(-1, x, xnew, 0.0, 0);
    m_jac->eval(x, xnew, 0.0);
    m_rdt = rdt_save;
}

/**
 * Return a pointer to the domain that contains component i of the
 * global solution vector. The domains are scanned right-to-left,
 * and the first one with starting location less or equal to i is
 * returned.
 *
 * 8/26/02 changed '<' to '<='  DGG
 *
 */
Domain1D* OneDim::pointDomain(int i) {
    Domain1D* d = right();
    while (d) {
        if (d->loc() <= i) return d;
        d = d->left();
    }
    return 0;
}


/**
 * Evaluate the multi-domain residual function, and return the
 * result in array r.
 */
void OneDim::eval(int j, double* x, double* r, doublereal rdt, int count) {
    clock_t t0 = clock();
    fill(r, r + m_size, 0.0);
    fill(m_mask.begin(), m_mask.end(), 0);
    if (rdt < 0.0) rdt = m_rdt;
    //        int nn;
    vector<Domain1D*>::iterator d;

    // iterate over the bulk domains first
    for (d = m_bulk.begin(); d != m_bulk.end(); ++d) {
        (*d)->eval(j, x, r, DATA_PTR(m_mask), rdt);
    }

    // then over the connector domains
    for (d = m_connect.begin(); d != m_connect.end(); ++d) {
        (*d)->eval(j, x, r, DATA_PTR(m_mask), rdt);
    }

    // increment counter and time
    if (count) {
        clock_t t1 = clock();
        m_evaltime += double(t1 - t0)/CLOCKS_PER_SEC;
        m_nevals++;
    }
}
```

```
/**
 * The 'infinity' (maximum magnitude) norm of the steady-state
 * residual. Used only for diagnostic output.
 */
doublereal OneDim::ssnorm(doublereal* x, doublereal* r) {
    eval(-1, x, r, 0.0, 0);
    doublereal ss = 0.0;
    for (int i = 0; i < m_size; i++) {
        ss = fmaxx(fabs(r[i]),ss);
    }
    return ss;
}


/**
 * Prepare for time stepping with timestep dt.
 */
void OneDim::initTimeInteg(doublereal dt, doublereal* x) {
    doublereal rdt_old = m_rdt;
    m_rdt = 1.0/dt;

    // if the stepsize has changed, then update the transient
    // part of the Jacobian
    if (fabs(rdt_old - m_rdt) > Tiny) {
        m_jac->updateTransient(m_rdt, DATA_PTR(m_mask));
    }

    // iterate over all domains, preparing each one to begin
    // time stepping
    Domain1D* d = left();
    while (d) {
        d->initTimeInteg(dt, x);
        d = d->right();
    }
}


/**
 * Prepare to solve the steady-state problem.  Set the reciprocal
 * of the time step to zero, and, if it was previously non-zero,
 * signal that a new Jacobian will be needed.
 */
void OneDim::setSteadyMode() {
    m_rdt = 0.0;
    m_jac->updateTransient(m_rdt, DATA_PTR(m_mask));
}

/**
 * Initialize all domains. On the first call, this methods calls
 * the init method of each domain, proceeding from left to right.
 * Subsequent calls do nothing.
 */
void OneDim::init() {
    if (!m_init) {
        Domain1D* d = left();
        while (d) {
            d->init();
            d = d->right();
        }
    }
    m_init = true;
}


/**
 * Signal that the current Jacobian is no longer valid.
 */
void Domain1D::needJacUpdate() {
    if (m_container) {
```

```
        m_container->jacobian().setAge(10000);
        m_container->saveStats();
    }
}

/**
 * Take time steps using Backward Euler.
 *
 *  nsteps   -- number of steps
 *  dt       -- initial step size
 *  loglevel -- controls amount of printed diagnostics
 */
doublereal OneDim::timeStep(int nsteps, doublereal dt, doublereal* x,
    doublereal* r, int loglevel) {

    // set the Jacobian age parameter to the transient value
    newton().setOptions(m_ts_jac_age);

    if (loglevel > 0) {
        //writelog("Begin time stepping.\n\n");
        writelog("\n\n step    size (s)    log10(ss) \n");
        writelog("================================\n");
    }

    int n = 0, m;
    doublereal ss;
    char str[80];
           while (n < nsteps) {
                  dosolid=1;  //solid phase must be solved before next gas phase iteration
                  if (loglevel > 0) {
            ss = ssnorm(x, r);
            sprintf(str, " %4d  %10.4g  %10.4g" , n,dt,log10(ss));
            writelog(str);
        }

        // set up for time stepping with stepsize dt
        initTimeInteg(dt,x);

        // solve the transient problem
        m = solve(x, r, loglevel-1);

        // successful time step. Copy the new solution in r to
        // the current solution in x.
        if (m >= 0) {
            n += 1;
            if (loglevel > 0) writelog("\n");
            copy(r, r + m_size, x);
            if (m == 100) {
                dt *= 1.5;
            }
            //                 else dt /= 1.5;
            if (dt > m_tmax) dt = m_tmax;
        }

        // No solution could be found with this time step.
        // Decrease the stepsize and try again.
        else {
            if (loglevel > 0) writelog("...failure.\n");
            dt *= m_tfactor;
            if (dt < m_tmin)
                throw CanteraError("OneDim::timeStep",
                    "Time integration failed.");
        }
    }

    // Prepare to solve the steady problem.
    setSteadyMode();
    newton().setOptions(m_ss_jac_age);

    // return the value of the last stepsize, which may be smaller
    // than the initial stepsize
```

```
        return dt;
    }


    void OneDim::save(string fname, string id, string desc, doublereal* sol) {

        struct tm *newtime;
        time_t aclock;
        ::time( &aclock );              /* Get time in seconds */
        newtime = localtime( &aclock ); /* Convert time to struct tm form */

        XML_Node root("doc");
        ifstream fin(fname.c_str());
        XML_Node* ct;
        if (fin) {
            root.build(fin);
            const XML_Node* same_ID = root.findID(id);
            int jid = 1;
            string idnew = id;
            while (same_ID != 0) {
                idnew = id + "_" + int2str(jid);
                jid++;
                same_ID = root.findID(idnew);
            }
            id = idnew;
            fin.close();
            ct = &root.child("ctml");
        }
        else {
            ct = &root.addChild("ctml");
        }
        XML_Node& sim = (XML_Node&)ct->addChild("simulation");
        sim.addAttribute("id",id);
        addString(sim,"timestamp",asctime(newtime));
        if (desc != "") addString(sim,"description",desc);

        Domain1D* d = left();
        while (d) {
            d->save(sim, sol);
            d = d->right();
        }
        ofstream s(fname.c_str());
        if (!s)
            throw CanteraError("save","could not open file "+fname);
        ct->write(s);
        s.close();
        writelog("Solution saved to file "+fname+" as solution "+id+".\n");
    }


    void Domain1D::setGrid(int n, const doublereal* z) {
        m_z.resize(n);
        m_points = n;
        int j;
        for (j = 0; j < m_points; j++) m_z[j] = z[j];
    }

}
```

C.2 refine.cpp

```
// turn off warnings under Windows
#ifdef WIN32
#pragma warning(disable:4786)
#pragma warning(disable:4503)
#endif


#include <map>
#include <algorithm>
#include "Domain1D.h"

#include "refine.h"
#include "Stflow.cpp"

using namespace std;

namespace Cantera {

    template<class M>
    bool has_key(const M& m, int j) {
        if (m.find(j) != m.end()) return true;
        return false;
    }

    static void r_drawline() {
        string s(78,'#');
        s += '\n';
        writelog(s.c_str());
    }

    /**
     * Return the square root of machine precision.
     */
    static doublereal eps() {
        doublereal e = 1.0;
        while (1.0 + e != 1.0) e *= 0.5;
        return sqrt(e);
    }


    Refiner::Refiner(Domain1D& domain) :
        m_ratio(10.0), m_slope(0.8), m_curve(0.8), m_prune(-0.001),
        m_min_range(0.01), m_domain(&domain), m_npmax(3000)
    {
        m_nv = m_domain->nComponents();
        m_active.resize(m_nv, true);
        m_thresh = eps();
    }


    int Refiner::analyze(int n, const doublereal* z,
        const doublereal* x) {

        if (n >= m_npmax) {
            writelog("max number of grid points reached ("+int2str(m_npmax)+".\n");
            return -2;
        }

        if (m_domain->nPoints() <= 1) {
            //writelog("can't refine a domain with 1 point: "+m_domain->id()+"\n");
            return 0;
        }

        m_loc.clear();
        m_c.clear();
        m_keep.clear();
```

```
m_keep[0] = 1;
m_keep[n-1] = 1;

m_nv = m_domain->nComponents();

// check consistency
if (n != m_domain->nPoints())
    throw CanteraError("analyze","inconsistent");


/**
 * find locations where cell size ratio is too large.
 */
int j;
vector_fp dz(n-1, 0.0);
string name;
doublereal vmin, vmax, smin, smax, aa, ss;
doublereal dmax, r;
vector_fp v(n), s(n-1);

for (int i = 0; i < m_nv; i++) {
    if (m_active[i]) {
        name = m_domain->componentName(i);
        //writelog("refine: examining "+name+"\n");
        // get component i at all points
        for (j = 0; j < n; j++) v[j] = value(x, i, j);

        // slope of component i
        for (j = 0; j < n-1; j++)
            s[j] = (value(x, i, j+1) - value(x, i, j))/
                   (z[j+1] - z[j]);

        // find the range of values and slopes

        vmin = *min_element(v.begin(), v.end());
        vmax = *max_element(v.begin(), v.end());
        smin = *min_element(s.begin(), s.end());
        smax = *max_element(s.begin(), s.end());

        // max absolute values of v and s
        aa = fmaxx(fabs(vmax), fabs(vmin));
        ss = fmaxx(fabs(smax), fabs(smin));

        // refine based on component i only if the range of v is
        // greater than a fraction 'min_range' of max |v|. This
        // eliminates components that consist of small fluctuations
        // on a constant background.

        if ((vmax - vmin) > m_min_range*aa) {

            // maximum allowable difference in value between
            // adjacent points.

            dmax = m_slope*(vmax - vmin) + m_thresh;
            for (j = 0; j < n-1; j++) {
                r = fabs(v[j+1] - v[j])/dmax;
                if (r > 1.0) {
                    m_loc[j] = 1;
                    m_c[name] = 1;
                    //if (int(m_loc.size()) + n > m_npmax) goto done;
                }
                if (r >= m_prune) {
                    m_keep[j] = 1;
                    m_keep[j+1] = 1;
                }
                else {
                    //writelog(string("r = ")+fp2str(r)+"\n");
                    if (m_keep[j] == 0) {
                        //if (m_keep[j-1] > -1 && m_keep[j+1] > -1)
                        m_keep[j] = -1;
                    }
```

```
                    //if (m_keep[j+1] == 0) m_keep[j+1] = -1;
                }
            }
        }


        // refine based on the slope of component i only if the
        // range of s is greater than a fraction 'min_range' of max
        // |s|. This eliminates components that consist of small
        // fluctuations on a constant slope background.

        if ((smax - smin) > m_min_range*ss) {

            // maximum allowable difference in slope between
            // adjacent points.
            dmax = m_curve*(smax - smin) + m_thresh; // + 0.5*m_curve*(smax + smin);
            for (j = 0; j < n-2; j++) {
                r = fabs(s[j+1] - s[j]) /dmax;
                if (r > 1.0) {
                    m_c[name] = 1;
                    m_loc[j] = 1;
                    m_loc[j+1] = 1;
                    //if (int(m_loc.size()) + n > m_npmax) goto done;
                }
                if (r >= m_prune) {
                    m_keep[j+1] = 1;
                }
                else {
                    //writelog(string("r slope = ")+fp2str(r)+"\n");
                    if (m_keep[j+1] == 0) {
                        //if (m_keep[j] > -1 && m_keep[j+2] > -1)
                        m_keep[j+1] = -1;
                    }
                }
            }
        }
    }
}
        //Section added so that Ts effects refinement process.
        name='Ts';
        for (j = 0; j < n; j++) v[j] = Tw[j];
        for (j = 0; j < n-1; j++)
                s[j] = (Tw[j+1] - Tw[j])/
                        (z[j+1] - z[j]);
        // find the range of values and slopes

        vmin = *min_element(v.begin(), v.end());
        vmax = *max_element(v.begin(), v.end());
        smin = *min_element(s.begin(), s.end());
        smax = *max_element(s.begin(), s.end());

        // max absolute values of v and s
        aa = fmaxx(fabs(vmax), fabs(vmin));
        ss = fmaxx(fabs(smax), fabs(smin));
        // refine based on component i only if the range of v is
        // greater than a fraction 'min_range' of max |v|. This
        // eliminates components that consist of small fluctuations
        // on a constant background.

        if ((vmax - vmin) > m_min_range*aa) {

                // maximum allowable difference in value between
                // adjacent points.

                dmax = m_slope*(vmax - vmin) + m_thresh;
                for (j = 0; j < n-1; j++) {
                        r = fabs(v[j+1] - v[j])/dmax;
                        if (r > 1.0) {
                                m_loc[j] = 1;
                                m_c[name] = 1;
                                //if (int(m_loc.size()) + n > m_npmax) goto done;
```

```
                        }
                        if (r >= m_prune) {
                                m_keep[j] = 1;
                                m_keep[j+1] = 1;
                        }
                        else {
                                //writelog(string("r = ")+fp2str(r)+"\n");
                                if (m_keep[j] == 0) {
                                        //if (m_keep[j-1] > -1 && m_keep[j+1] > -1)
                                        m_keep[j] = -1;
                                }
                                //if (m_keep[j+1] == 0) m_keep[j+1] = -1;
                        }
                }
        }


        // refine based on the slope of component i only if the
        // range of s is greater than a fraction 'min_range' of max
        // |s|. This eliminates components that consist of small
        // fluctuations on a constant slope background.

        if ((smax - smin) > m_min_range*ss) {

                // maximum allowable difference in slope between
                // adjacent points.
                dmax = m_curve*(smax - smin) + m_thresh; // + 0.5*m_curve*(smax + smin);
                for (j = 0; j < n-2; j++) {
                        r = fabs(s[j+1] - s[j]) /dmax;
                        if (r > 1.0) {
                                m_c[name] = 1;
                                m_loc[j] = 1;
                                m_loc[j+1] = 1;
                                //if (int(m_loc.size()) + n > m_npmax) goto done;
                        }
                        if (r >= m_prune) {
             m_keep[j+1] = 1;
                        }
                        else {
                                //writelog(string("r slope = ")+fp2str(r)+"\n");
                                if (m_keep[j+1] == 0) {
                                        //if (m_keep[j] > -1 && m_keep[j+2] > -1)
                                        m_keep[j+1] = -1;
                                }
                        }
                }
        }
        //End of new section

    dz[0] = z[1] - z[0];
    for (j = 1; j < n-1; j++) {
        dz[j] = z[j+1] - z[j];
        if (dz[j] > m_ratio*dz[j-1]) {
            m_loc[j] = 1;
            m_c["point "+int2str(j)] = 1;
        }
        if (dz[j] < dz[j-1]/m_ratio) {
            m_loc[j-1] = 1;
            m_c["point "+int2str(j-1)] = 1;
        }
        //if (m_loc.size() + n > m_npmax) goto done;
    }

    //done:
    //m_did_analysis = true;
    return static_cast<int>(m_loc.size());
}

double Refiner::value(const double* x, int i, int j) {
    return x[m_domain->index(i,j)];
}
```

```
    void Refiner::show() {
        int nnew = static_cast<int>(m_loc.size());
        if (nnew > 0) {
            r_drawline();
            writelog(string("Refining grid in ") +
                m_domain->id()+".\n"
                +"    New points inserted after grid points ");
            map<int, int>::const_iterator b = m_loc.begin();
            for (; b != m_loc.end(); ++b) {
                writelog(int2str(b->first)+" ");
            }
            writelog("\n");
            writelog("    to resolve ");
            map<string, int>::const_iterator bb = m_c.begin();
            for (; bb != m_c.end(); ++bb) {
                writelog(string(bb->first)+" ");
            }
            writelog("\n");
        }
        else if (m_domain->nPoints() > 1) {
            writelog("no new points needed in "+m_domain->id()+"\n");
            //writelog("curve = "+fp2str(m_curve)+"\n");
            //writelog("slope = "+fp2str(m_slope)+"\n");
            //writelog("prune = "+fp2str(m_prune)+"\n");
        }
    }


    int Refiner::getNewGrid(int n, const doublereal* z,
        int nn, doublereal* zn) {
        int j;
        int nnew = static_cast<int>(m_loc.size());
        if (nnew + n > nn) {
            throw CanteraError("Refine::getNewGrid",
                "array size too small.");
            return -1;
        }

        int jn = 0;
        if (m_loc.size() == 0) {
            copy(z, z + n,  zn);
            return 0;
        }

        for (j = 0; j < n - 1; j++) {
            zn[jn] = z[j];
            jn++;
            if (has_key(m_loc, j)) {
                zn[jn] = 0.5*(z[j] + z[j+1]);
                jn++;
            }
        }
        zn[jn] = z[n-1];
        return 0;
    }
}
```

C.3 Stflow.h

```
/**
 * @file StFlow.h
 *
 */

/*
 * $Author: hkmoffa $
 * $Revision: 1.13 $
 * $Date: 2006/03/07 20:52:16 $
 */

// Copyright 2001  California Institute of Technology

#ifndef CT_STFLOW_H
#define CT_STFLOW_H

#include "../transport/TransportBase.h"
#include "Domain1D.h"
#include "../Array.h"
#include "../IdealGasPhase.h"
#include "../Kinetics.h"
#include "../funcs.h"
//#include "../flowBoundaries.h"

namespace Cantera {

    typedef IdealGasPhase igthermo_t;

    class MultiJac;


    //-------------------------------------------
    //   constants
    //-------------------------------------------

    // Offsets of solution components in the solution array.
    const unsigned int c_offset_U = 0;    // axial velocity
    const unsigned int c_offset_V = 1;    // strain rate
    const unsigned int c_offset_T = 2;    // temperature
    const unsigned int c_offset_L = 3;    // (1/r)dP/dr
    const unsigned int c_offset_Y = 4;    // mass fractions

    // Transport option flags
    const int c_Mixav_Transport = 0;
    const int c_Multi_Transport = 1;
    const int c_Soret = 2;



    //-------------------------------------------------------------
    //  Class StFlow
    //-------------------------------------------------------------


    /**
     *  This class represents 1D flow domains that satisfy the
     *  one-dimensional similarity solution for chemically-reacting,
     *  axisymmetric, flows.
     */
    class StFlow : public Domain1D {

    public:

        //--------------------------------
        // construction and destruction
        //--------------------------------
```

```
        /// Constructor. Create a new flow domain.
        /// @param gas Object representing the gas phase. This object
        /// will be used to evaluate all thermodynamic, kinetic, and transport
        /// properties.
        /// @param nsp Number of species.
        StFlow(igthermo_t* ph = 0, int nsp = 1, int points = 1);

        /// Destructor.
        virtual ~StFlow(){}

        /**
         * @name Problem Specification
         */
        //@{

        virtual void setupGrid(int n, const doublereal* z);

        thermo_t& phase() { return *m_thermo; }
        kinetics_t& kinetics() { return *m_kin; }

        virtual void init(){
               }

               /**
         * Set the thermo manager. Note that the flow equations assume
         * the ideal gas equation.
         */
        void setThermo(igthermo_t& th) { m_thermo = &th; }

               //initialize the solid solver  as well as the radiant flux vector
               virtual void solid(doublereal* x, vector<double>& hconv, vector<double>& scond,
vector<double>& RK, vector<double>& Omega, double & srho, double & sCp, double rdt);
               vector<double> dq;
               //initialize the solid properties
               double pore1;
               double pore2;
               double diam1;
               double diam2;
               double Omega1;
               double Omega2;
               double srho;
               double sCp;

               /// Set the kinetics manager. The kinetics manager must
        void setKinetics(kinetics_t& kin) { m_kin = &kin; }

        /// set the transport manager
        void setTransport(Transport& trans, bool withSoret = false);

        /// Set the pressure. Since the flow equations are for the limit of
        /// small Mach number, the pressure is very nearly constant
        /// throughout the flow.
        void setPressure(doublereal p) { m_press = p; }


        /// @todo remove? may be unused
        virtual void setState(int point, const doublereal* state,
                              doublereal *x) {
            setTemperature(point, state[2]);
            int k;
            for (k = 0; k < m_nsp; k++) {
                setMassFraction(point, k, state[4+k]);
            }
        }


        /// Write the initial solution estimate into
        /// array x.
        virtual void _getInitialSoln(doublereal* x) {
            int k, j;
            for (j = 0; j < m_points; j++) {
```

```
            x[index(2,j)] = T_fixed(j);
            for (k = 0; k < m_nsp; k++) {
                x[index(4+k,j)] = Y_fixed(k,j);
            }
        }
    }

    virtual void _finalize(const doublereal* x);


    /// Sometimes it is desired to carry out the simulation
    /// using a specified temperature profile, rather than
    /// computing it by solving the energy equation. This
    /// method specifies this profile.
    void setFixedTempProfile(vector_fp& zfixed, vector_fp& tfixed) {
        m_zfix = zfixed;
        m_tfix = tfixed;
    }

    /**
     * Set the temperature fixed point at grid point j, and
     * disable the energy equation so that the solution will be
     * held to this value.
     */
    void setTemperature(int j, doublereal t) {
        m_fixedtemp[j] = t;
        m_do_energy[j] = false;
    }

    /**
     * Set the mass fraction fixed point for species k at grid
     * point j, and disable the species equation so that the
     * solution will be held to this value.
     * note: in practice, the species are hardly ever held fixed.
     */
    void setMassFraction(int j, int k, doublereal y) {
        m_fixedy(k,j) = y;
        m_do_species[k] = true; // false;
    }


     /// The fixed temperature value at point j.
    doublereal T_fixed(int j) const {return m_fixedtemp[j];}


    /// The fixed mass fraction value of species k at point j.
    doublereal Y_fixed(int k, int j) const {return m_fixedy(k,j);}


    virtual string componentName(int n) const;

    //added by Karl Meredith
    int componentIndex(string name) const;


    virtual void showSolution(const doublereal* x);

    virtual void save(XML_Node& o, doublereal* sol);

    virtual void restore(const XML_Node& dom, doublereal* soln);

    // overloaded in subclasses
    virtual string flowType() { return "<none>"; }

    void solveEnergyEqn(int j=-1) {
        if (j < 0)
            for (int i = 0; i < m_points; i++)
                m_do_energy[i] = true;
        else
            m_do_energy[j] = true;
        m_refiner->setActive(0, true);
```

```
        m_refiner->setActive(1, true);
        m_refiner->setActive(2, true);
        needJacUpdate();
    }

    void fixTemperature(int j=-1) {
        if (j < 0)
            for (int i = 0; i < m_points; i++) {
                m_do_energy[i] = false;
            }
        else m_do_energy[j] = false;
        m_refiner->setActive(0, false);
        m_refiner->setActive(1, false);
        m_refiner->setActive(2, false);
        needJacUpdate();
    }

    bool doSpecies(int k) { return m_do_species[k]; }
    bool doEnergy(int j) { return m_do_energy[j]; }

    void solveSpecies(int k=-1) {
        if (k == -1) {
            for (int i = 0; i < m_nsp; i++)
                m_do_species[i] = true;
        }
        else m_do_species[k] = true;
        needJacUpdate();
    }

    void fixSpecies(int k=-1) {
        if (k == -1) {
            for (int i = 0; i < m_nsp; i++)
                m_do_species[i] = false;
        }
        else m_do_species[k] = false;
        needJacUpdate();
    }

    void integrateChem(doublereal* x,doublereal dt);

    void resize(int components, int points);

    virtual void setFixedPoint(int j0, doublereal t0){}


    void setJac(MultiJac* jac);
    void setGas(const doublereal* x,int j);
    void setGasAtMidpoint(const doublereal* x,int j);

    //Karl Meredith
    //        doublereal density_unprotected(int j) const {
    //    return m_rho[j];
    // }
    doublereal density(int j) const {
        return m_rho[j];
    }

    virtual bool fixed_mdot() { return true; }
    void setViscosityFlag(bool dovisc) { m_dovisc = dovisc; }

protected:

    doublereal component(const doublereal* x, int i, int j) const {
        doublereal xx = x[index(i,j)];
        return xx;
    }

    doublereal conc(const doublereal* x,int k,int j) const {
        return Y(x,k,j)*density(j)/m_wt[k];
    }
```

```
doublereal cbar(const doublereal* x,int k, int j) const {
    return sqrt(8.0*GasConstant * T(x,j) / (Pi * m_wt[k]));
}

doublereal wdot(int k, int j) const {return m_wdot(k,j);}

/// write the net production rates at point j into array m_wdot
void getWdot(doublereal* x,int j) {
    setGas(x,j);
    m_kin->getNetProductionRates(&m_wdot(0,j));
}

/**
 * update the thermodynamic properties from point
 * j0 to point j1 (inclusive), based on solution x.
 */
void updateThermo(const doublereal* x, int j0, int j1) {
    int j;
    for (j = j0; j <= j1; j++) {
        setGas(x,j);
        m_rho[j] = m_thermo->density();
        m_wtm[j] = m_thermo->meanMolecularWeight();
        m_cp[j]  = m_thermo->cp_mass();
    }
}


//---------------------------------
// central-differenced derivatives
//---------------------------------

doublereal cdif2(const doublereal* x, int n, int j,
    const doublereal* f) const {
    doublereal c1 = (f[j] + f[j-1])*(x[index(n,j)] - x[index(n,j-1)]);
    doublereal c2 = (f[j+1] + f[j])*(x[index(n,j+1)] - x[index(n,j)]);
    return (c2/(z(j+1) - z(j)) - c1/(z(j) - z(j-1)))/(z(j+1) - z(j-1));
}


//---------------------------------
//      solution components
//---------------------------------


doublereal T(const doublereal* x,int j) const {
    return x[index(c_offset_T, j)];
}
doublereal& T(doublereal* x,int j) {return x[index(c_offset_T, j)];}
doublereal T_prev(int j) const {return prevSoln(c_offset_T, j);}

doublereal rho_u(const doublereal* x,int j) const {
    return m_rho[j]*x[index(c_offset_U, j)];}

doublereal u(const doublereal* x,int j) const {
    return x[index(c_offset_U, j)];}

doublereal V(const doublereal* x,int j) const {
    return x[index(c_offset_V, j)];}
doublereal V_prev(int j) const {
    return prevSoln(c_offset_V, j);}

doublereal lambda(const doublereal* x,int j) const {
    return x[index(c_offset_L, j)];
}

doublereal Y(const doublereal* x,int k, int j) const {
    return x[index(c_offset_Y + k, j)];
}

doublereal& Y(doublereal* x,int k, int j) {
    return x[index(c_offset_Y + k, j)];
```

```
}

doublereal Y_prev(int k, int j) const {
    return prevSoln(c_offset_Y + k, j);
}

doublereal X(const doublereal* x,int k, int j) const {
    return m_wtm[j]*Y(x,k,j)/m_wt[k];
}

doublereal flux(int k, int j) const {
    return m_flux(k, j);
}


// convective spatial derivatives. These use upwind
// differencing, assuming u(z) is negative

doublereal dVdz(const doublereal* x,int j) const {
    int jloc = (u(x,j) > 0.0 ? j : j + 1);
    return (V(x,jloc) - V(x,jloc-1))/m_dz[jloc-1];
}

doublereal dYdz(const doublereal* x,int k, int j) const {
    int jloc = (u(x,j) > 0.0 ? j : j + 1);
    return (Y(x,k,jloc) - Y(x,k,jloc-1))/m_dz[jloc-1];
}

doublereal dTdz(const doublereal* x,int j) const {
    int jloc = (u(x,j) > 0.0 ? j : j + 1);
    return (T(x,jloc) - T(x,jloc-1))/m_dz[jloc-1];
}

doublereal shear(const doublereal* x,int j) const {
    doublereal c1 = m_visc[j-1]*(V(x,j) - V(x,j-1));
    doublereal c2 = m_visc[j]*(V(x,j+1) - V(x,j));
    return 2.0*(c2/(z(j+1) - z(j)) - c1/(z(j) - z(j-1)))/(z(j+1) - z(j-1));
}

doublereal divHeatFlux(const doublereal* x, int j) const {
    doublereal c1 = m_tcon[j-1]*(T(x,j) - T(x,j-1));
    doublereal c2 = m_tcon[j]*(T(x,j+1) - T(x,j));
    return -2.0*(c2/(z(j+1) - z(j)) - c1/(z(j) - z(j-1)))/(z(j+1) - z(j-1));
}

int mindex(int k, int j, int m) {
    return m*m_nsp*m_nsp + m_nsp*j + k;
}

void updateDiffFluxes(const doublereal* x, int j0, int j1);


//-------------------------------------------------------------
//
//             member data
//
//-------------------------------------------------------------

// inlet
doublereal m_inlet_u;
doublereal m_inlet_V;
doublereal m_inlet_T;
doublereal m_rho_inlet;
vector_fp m_yin;


// surface
doublereal m_surface_T;

doublereal m_press;        // pressure
```

```
    // grid parameters
    vector_fp m_dz;
    //vector_fp m_z;

    // mixture thermo properties
    vector_fp m_rho;
    vector_fp m_wtm;

    // species thermo properties
    vector_fp m_wt;
    vector_fp m_cp;
    vector_fp m_enth;

    // transport properties
    vector_fp m_visc;
    vector_fp m_tcon;
    vector_fp m_diff;
    vector_fp m_multidiff;
    Array2D m_dthermal;
    Array2D m_flux;

    // production rates
    Array2D m_wdot;
    vector_fp m_surfdot;

    int m_nsp;

    igthermo_t*     m_thermo;
    kinetics_t*     m_kin;
    Transport*      m_trans;

    MultiJac*       m_jac;

    bool m_ok;

    // flags
    vector<bool> m_do_energy;
    bool m_do_soret;
    vector<bool> m_do_species;
    int m_transport_option;

    // solution estimate
    //vector_fp m_zest;
    //Array2D   m_yest;

    // fixed T and Y values
    Array2D   m_fixedy;
    vector_fp m_fixedtemp;
    vector_fp m_zfix;
    vector_fp m_tfix;

    doublereal m_efctr;
    bool m_dovisc;
    void updateTransport(doublereal* x,int j0, int j1);

private:
    vector_fp m_ybar;

};


/**
 * A class for axisymmetric stagnation flows.
 */
class AxiStagnFlow : public StFlow {
   friend class OneDim;
public:
    AxiStagnFlow(igthermo_t* ph = 0, int nsp = 1, int points = 1) :
        StFlow(ph, nsp, points) { m_dovisc = true; }
    virtual ~AxiStagnFlow() {}
    virtual void eval(int j, doublereal* x, doublereal* r,
```

```
            integer* mask, doublereal rdt);
               virtual string flowType() { return "Axisymmetric Stagnation"; }


    };

    /**
     * A class for freely-propagating premixed flames.
     */
    class FreeFlame : public StFlow {
    public:
        FreeFlame(igthermo_t* ph = 0, int nsp = 1, int points = 1) :
            StFlow(ph, nsp, points) { m_dovisc = false; }
        virtual ~FreeFlame() {}
        virtual void eval(int j, doublereal* x, doublereal* r,
            integer* mask, doublereal rdt);
        virtual string flowType() { return "Free Flame"; }
        virtual bool fixed_mdot() { return false; }
    };


    /*
    class OneDFlow : public StFlow {
    public:
        OneDFlow(igthermo_t* ph = 0, int nsp = 1, int points = 1) :
            StFlow(ph, nsp, points) {
        }
        virtual ~OneDFlow() {}
        virtual void eval(int j, doublereal* x, doublereal* r,
            integer* mask, doublereal rdt);
        virtual string flowType() { return "OneDFlow"; }
        doublereal mdot(doublereal* x, int j) {
            return x[index(c_offset_L,j)];
        }

    private:
        void updateTransport(doublereal* x,int j0, int j1);
    };
    */

    void importSolution(doublereal* oldSoln, igthermo_t& oldmech,
        doublereal* newSoln, igthermo_t& newmech);

}

#endif
```

C.4 Stflow.cpp

```
/**
 * @file StFlow.cpp
 */

/*
 * $Author: dggoodwin $
 * $Revision: 1.29 $
 * $Date: 2006/04/28 17:22:23 $
 */

// Copyright 2002  California Institute of Technology


// turn off warnings under Windows
#ifdef WIN32
#pragma warning(disable:4786)
#pragma warning(disable:4503)
#pragma warning(disable:4267)
#endif

#include <stdlib.h>
#include <time.h>
#include <vector>
#include <fstream>

#include "StFlow.h"
#include "../ArrayViewer.h"
#include "../ctml.h"
#include "MultiJac.h"
#include "OneDim.cpp"

using namespace ctml;
using namespace std;

namespace Cantera {
        //initialize solid temperature vector, the previous temperature profile, the previous
mesh,
        //the heat transfer coefficient, and no adaption
        vector<double> Tw;
        vector<double> Twprev;
        vector<double> Twprev1;
        vector<double> zprev;
        vector<double> hconv;
        int adapt=0;


    //-------------------  importSolution ------------------------

    /**
     * Import a previous solution to use as an initial estimate. The
     * previous solution may have been computed using a different
     * reaction mechanism. Species in the old and new mechanisms are
     * matched by name, and any species in the new mechanism that were
     * not in the old one are set to zero. The new solution is created
     * with the same number of grid points as in the old solution.
     */
    void importSolution(int points,
        doublereal* oldSoln, igthermo_t& oldmech,
        int size_new, doublereal* newSoln, igthermo_t& newmech) {

        // Number of components in old and new solutions
        int nv_old = oldmech.nSpecies() + 4;
        int nv_new = newmech.nSpecies() + 4;

        if (size_new < nv_new*points) {
            throw CanteraError("importSolution",
                "new solution array must have length "+
```

```
            int2str(nv_new*points));
    }

    int n, j, knew;
    string nm;

    // copy u,V,T,lambda
    for (j = 0; j < points; j++)
        for (n = 0; n < 4; n++)
            newSoln[nv_new*j + n] = oldSoln[nv_old*j + n];

    // copy mass fractions
    int nsp0 = oldmech.nSpecies();
    //int nsp1 = newmech.nSpecies();

    // loop over the species in the old mechanism
    for (int k = 0; k < nsp0; k++) {
        nm = oldmech.speciesName(k);      // name

        // location of this species in the new mechanism.
        // If < 0, then the species is not in the new mechanism.
        knew = newmech.speciesIndex(nm);

        // copy this species from the old to the new solution vectors
        if (knew >= 0) {
            for (j = 0; j < points; j++) {
                newSoln[nv_new*j + 4 + knew] = oldSoln[nv_old*j + 4 + k];
            }
        }
    }


    // normalize mass fractions
    for (j = 0; j < points; j++) {
        newmech.setMassFractions(&newSoln[nv_new*j + 4]);
        newmech.getMassFractions(&newSoln[nv_new*j + 4]);
    }
}


    static void st_drawline() {
    writelog("\n----------------------------------------"
        "----------------------------------------");
}

StFlow::StFlow(igthermo_t* ph, int nsp, int points) :
    Domain1D(nsp+4, points),
    m_inlet_u(0.0),
    m_inlet_V(0.0),
    m_inlet_T(-1.0),
    m_surface_T(-1.0),
    m_press(-1.0),
    m_nsp(nsp),
    m_thermo(0),
    m_kin(0),
    m_trans(0),
    m_jac(0),
    m_ok(false),
    m_do_soret(false),
    m_transport_option(-1),
    m_efctr(0.0)
{
    m_type = cFlowType;

    m_points = points;
    m_thermo = ph;

    if (ph == 0) return; // used to create a dummy object

    int nsp2 = m_thermo->nSpecies();
    if (nsp2 != m_nsp) {
```

```
    m_nsp = nsp2;
    Domain1D::resize(m_nsp+4, points);
}


// make a local copy of the species molecular weight vector
m_wt = m_thermo->molecularWeights();

// the species mass fractions are the last components in the solution
// vector, so the total number of components is the number of species
// plus the offset of the first mass fraction.
m_nv = c_offset_Y + m_nsp;

// enable all species equations by default
m_do_species.resize(m_nsp, true);

// but turn off the energy equation at all points
m_do_energy.resize(m_points,false);

m_diff.resize(m_nsp*m_points);
m_multidiff.resize(m_nsp*m_nsp*m_points);
m_flux.resize(m_nsp,m_points);
m_wdot.resize(m_nsp,m_points, 0.0);
m_surfdot.resize(m_nsp, 0.0);
m_ybar.resize(m_nsp);


//-------------- default solution bounds --------------------

vector_fp vmin(m_nv), vmax(m_nv);

// no bounds on u
vmin[0] = -1.e20;
vmax[0] = 1.e20;

// V
vmin[1] = -1.e20;
vmax[1] = 1.e20;

// temperature bounds
vmin[2] = 200.0;
vmax[2]= 1.e9;

// lamda should be negative
vmin[3] = -1.e20;
vmax[3] = 1.e20;

// mass fraction bounds
int k;
for (k = 0; k < m_nsp; k++) {
    vmin[4+k] = -1.0e-5;
    vmax[4+k] = 1.0e5;
}
setBounds(vmin.size(), DATA_PTR(vmin), vmax.size(), DATA_PTR(vmax));


//-------------------- default error tolerances ----------------
vector_fp rtol(m_nv, 1.0e-8);
vector_fp atol(m_nv, 1.0e-15);
setTolerances(rtol.size(), DATA_PTR(rtol), atol.size(), DATA_PTR(atol),false);
setTolerances(rtol.size(), DATA_PTR(rtol), atol.size(), DATA_PTR(atol),true);

//-------------------- grid refinement -------------------------
m_refiner->setActive(0, false);
m_refiner->setActive(1, false);
m_refiner->setActive(2, false);
m_refiner->setActive(3, false);

vector_fp gr;
for (int ng = 0; ng < m_points; ng++) gr.push_back(1.0*ng/m_points);
setupGrid(m_points, DATA_PTR(gr));
```

```
    setID("stagnation flow");

            ifstream in("Properties2.txt"); //Read in the solid properties
            double proper;
            in>>proper;
            pore1=proper;
            in>>proper;
            pore2=proper;
            in>>proper;
            diam1=proper;
            in>>proper;
            diam2=proper;
            in>>proper;
            Omega1=proper;
            in>>proper;
            Omega2=proper;
            in>>proper;
            srho=proper;
            in>>proper;
            sCp=proper;
            in.close();

}


/**
 * Change the grid size. Called after grid refinement.
 */
void StFlow::resize(int ncomponents, int points) {
    Domain1D::resize(ncomponents, points);
    m_rho.resize(m_points, 0.0);
    m_wtm.resize(m_points, 0.0);
    m_cp.resize(m_points, 0.0);
    m_enth.resize(m_points, 0.0);
    m_visc.resize(m_points, 0.0);
    m_tcon.resize(m_points, 0.0);

    if (m_transport_option ==  c_Mixav_Transport) {
        m_diff.resize(m_nsp*m_points);
    }
    else {
        m_multidiff.resize(m_nsp*m_nsp*m_points);
        m_diff.resize(m_nsp*m_points);
    }
    m_flux.resize(m_nsp,m_points);
    m_wdot.resize(m_nsp,m_points, 0.0);
    m_do_energy.resize(m_points,false);

    m_fixedy.resize(m_nsp, m_points);
    m_fixedtemp.resize(m_points);

    m_dz.resize(m_points-1);
    m_z.resize(m_points);
}


void StFlow::setupGrid(int n, const doublereal* z) {
    resize(m_nv, n);
    int j;

    m_z[0] = z[0];
    for (j = 1; j < m_points; j++) {
        m_z[j] = z[j];
        m_dz[j-1] = m_z[j] - m_z[j-1];
    }
}


/**
 * Install a transport manager.
 */
```

```
void StFlow::setTransport(Transport& trans, bool withSoret) {
    m_trans = &trans;
    m_do_soret = withSoret;

    if (m_trans->model() == cMulticomponent) {
        m_transport_option = c_Multi_Transport;
        m_multidiff.resize(m_nsp*m_nsp*m_points);
        m_diff.resize(m_nsp*m_points);
        m_dthermal.resize(m_nsp, m_points, 0.0);
    }
    else if (m_trans->model() == cMixtureAveraged) {
        m_transport_option = c_Mixav_Transport;
        m_diff.resize(m_nsp*m_points);
        if (withSoret)
            throw CanteraError("setTransport",
                "Thermal diffusion (the Soret effect) "
                "requires using a multicomponent transport model.");
    }
    else
        throw CanteraError("setTransport","unknown transport model.");
}


/**
 * Set the gas object state to be consistent with the solution at
 * point j.
 */
void StFlow::setGas(const doublereal* x,int j) {
    m_thermo->setTemperature(T(x,j));
    const doublereal* yy = x + m_nv*j + c_offset_Y;
    m_thermo->setMassFractions_NoNorm(yy);
    m_thermo->setPressure(m_press);
}


/**
 * Set the gas state to be consistent with the solution at the
 * midpoint between j and j + 1.
 */
void StFlow::setGasAtMidpoint(const doublereal* x,int j) {
    m_thermo->setTemperature(0.5*(T(x,j)+T(x,j+1)));
    const doublereal* yyj = x + m_nv*j + c_offset_Y;
    const doublereal* yyjp = x + m_nv*(j+1) + c_offset_Y;
    for (int k = 0; k < m_nsp; k++)
        m_ybar[k] = 0.5*(yyj[k] + yyjp[k]);
    m_thermo->setMassFractions_NoNorm(DATA_PTR(m_ybar));
    m_thermo->setPressure(m_press);
}


void StFlow::_finalize(const doublereal* x) {
    int k, j;
    doublereal zz, tt;
    int nz = m_zfix.size();
    bool e = m_do_energy[0];
    for (j = 0; j < m_points; j++) {
        if (e || nz == 0)
            setTemperature(j, T(x, j));
        else {
            zz = (z(j) - z(0))/(z(m_points - 1) - z(0));
            tt = linearInterp(zz, m_zfix, m_tfix);
            setTemperature(j, tt);
        }
        for (k = 0; k < m_nsp; k++) {
            setMassFraction(j, k, Y(x, k, j));
        }
    }
    if (e) solveEnergyEqn();
}
```

```
//-----------------------------------------------------

/**
 *  Evaluate the residual function for axisymmetric stagnation
 *  flow. If jpt is less than zero, the residual function is
 *  evaluated at all grid points. If jpt >= 0, then the residual
 *  function is only evaluated at grid points jpt-1, jpt, and
 *  jpt+1. This option is used to efficiently evaluate the
 *  Jacobian numerically.
 *
 */

void AxiStagnFlow::eval(int jg, doublereal* xg,
    doublereal* rg, integer* diagg, doublereal rdt) {

    // if evaluating a Jacobian, and the global point is outside
    // the domain of influence for this domain, then skip
    // evaluating the residual
    if (jg >=0 && (jg < firstPoint() - 1 || jg > lastPoint() + 1)) return;

    // if evaluating a Jacobian, compute the steady-state residual
    if (jg >= 0) rdt = 0.0;

    // start of local part of global arrays
    doublereal* x = xg + loc();
    doublereal* rsd = rg + loc();
    integer* diag = diagg + loc();

    int jmin, jmax, jpt;
    jpt = jg - firstPoint();

    if (jg < 0) {      // evaluate all points
        jmin = 0;
        jmax = m_points - 1;
    }
    else {             // evaluate points for Jacobian
        jmin = max(jpt-1, 0);
        jmax = min(jpt+1,m_points-1);
    }

    // properties are computed for grid points from j0 to j1
    int j0 = max(jmin-1,0);
    int j1 = min(jmax+1,m_points-1);


    int j, k;


    //-------------------------------------------------
    //              update properties
    //-------------------------------------------------

    // update thermodynamic properties only if a Jacobian is not
    // being evaluated
    if (jpt < 0) { //if (jpt < 0 || (m_transport_option == c_Multi_Transport)) {
        updateThermo(x, j0, j1);

        // update transport properties only if a Jacobian is not being
        // evaluated
        updateTransport(x, j0, j1);
    }

    // update the species diffusive mass fluxes whether or not a
    // Jacobian is being evaluated
    updateDiffFluxes(x, j0, j1);


    //-------------------------------------------------
    // evaluate the residual equations at all required
    // grid points
    //-------------------------------------------------
```

```
doublereal sum, sum2, dtdzj;

        doublereal lam, visc, Re;   //Defining new variables.
        double length=m_points;                         //
        hconv.resize(length);  //

        //initialize property vectors
        vector<double> pore(length);
        vector<double> diam(length);
        vector<double> scond(length);
        vector<double> Omega(length);
        vector<double> Cmult(length);
        vector<double> mpow(length);
        vector<double> RK(length);

        //populate property vectors
        for (int i=0; i<=length-1;i++)
        {
                if (z(i)<0.033)
                {
                        pore[i]=pore1;
                        diam[i]=diam1;
                }
                else if (z(i)>0.037)
                {
                        pore[i]=pore2;
                        diam[i]=diam2;
                }
                else
                {
                        pore[i]=(((pore2-pore1)/(.037-.033))*(z(i)-0.033))+pore1;
                        diam[i]=(((diam2-diam1)/(.037-.033))*(z(i)-0.033))+diam1;
                }
                RK[i]=(3*(1-pore[i])/diam[i]);
                Cmult[i]=-400*diam[i]+0.687;
                mpow[i]=443.7*diam[i]+0.361;
                scond[i]=0.188-17.5*diam[i];
        }
        for (int i=0; i<=length-1;i++)
        {
                if (z(i)<0.035)
                {
                        Omega[i]=Omega1;
                }
                else
                {
                        Omega[i]=Omega2;
                }
        }

        int solidenergy=0;
        //loop over gas energy vecotr.  If it is going to be solved then find hv
        for(j=jmin;j<=jmax;j++)
        {
                solidenergy+=m_do_energy[j];
        }
        solidenergy=1;

        if (solidenergy!=0)
        {
        for (j = jmin; j <= jmax; j++)
        {
                lam=m_tcon[j];
                visc=m_visc[j];
                Re=(rho_u(x,j)*pore[j]*diam[j])/visc;
                hconv[j]=((lam*Cmult[j]*pow(Re,mpow[j]))/pow(diam[j],2));
        }
        //Solve for the solid profile if required
        if (dosolid==1)
        {
```

```
                solid(x,hconv,scond,RK,Omega,srho,sCp,rdt);
                dosolid=0;
        }
        }

for (j = jmin; j <= jmax; j++) {

    //-----------------------------------------------
    //         left boundary
    //-----------------------------------------------

    if (j == 0) {

        // these may be modified by a boundary object


        // Continuity. This propagates information right-to-left,
        // since rho_u at point 0 is dependent on rho_u at point 1,
        // but not on mdot from the inlet.
         rsd[index(c_offset_U,0)] =
             -(rho_u(x,1) - rho_u(x,0))/m_dz[0]
             -(density(1)*V(x,1) + density(0)*V(x,0));

         // the inlet (or other) object connected to this one
         // will modify these equations by subtracting its values
         // for V, T, and mdot. As a result, these residual equations
         // will force the solution variables to the values for
         // the boundary object
         rsd[index(c_offset_V,0)] = V(x,0);
         rsd[index(c_offset_T,0)] = T(x,0);
         rsd[index(c_offset_L,0)] = -rho_u(x,0);

         // The default boundary condition for species is zero
         // flux. However, the boundary object may modify
         // this.
         sum = 0.0;
         for (k = 0; k < m_nsp; k++) {
             sum += Y(x,k,0);
             rsd[index(c_offset_Y + k, 0)] =
                 -(m_flux(k,0) + rho_u(x,0)* Y(x,k,0));
         }
         rsd[index(c_offset_Y, 0)] = 1.0 - sum;
    }


    //-----------------------------------------------
    //
    //         right boundary
    //
    //-----------------------------------------------

    else if (j == m_points - 1) {

         // the boundary object connected to the right of this
         // one may modify or replace these equations. The
         // default boundary conditions are zero u, V, and T,
         // and zero diffusive flux for all species.

        rsd[index(0,j)] = rho_u(x,j);
        rsd[index(1,j)] = V(x,j);
        rsd[index(2,j)] = T(x,j);
        rsd[index(c_offset_L, j)] = lambda(x,j) - lambda(x,j-1);
        diag[index(c_offset_L, j)] = 0;
        doublereal sum = 0.0;
        for (k = 0; k < m_nsp; k++) {
            sum += Y(x,k,j);
            rsd[index(k+4,j)] = m_flux(k,j-1) + rho_u(x,j)*Y(x,k,j);
        }
        rsd[index(4,j)] = 1.0 - sum;
        diag[index(4,j)] = 0;
```

```
        }


        //-----------------------------------------
        //     interior points
        //-----------------------------------------

        else {

            //-----------------------------------------------
            //    Continuity equation
            //
            //    Note that this propagates the mass flow rate
            //    information to the left (j+1 -> j) from the
            //    value specified at the right boundary. The
            //    lambda information propagates in the opposite
            //    direction.
            //
            //    d(\rho u)/dz + 2\rho V = 0
            //
            //-----------------------------------------------

            rsd[index(c_offset_U,j)] =
                -(rho_u(x,j+1)*pore[j+1] - rho_u(x,j)*pore[j])/m_dz[j]  //added porosity
                -(density(j+1)*V(x,j+1) + density(j)*V(x,j));

            //algebraic constraint
            diag[index(c_offset_U, j)] = 0;


            //-----------------------------------------------
            //    Radial momentum equation
            //
            //    \rho u dV/dz + \rho V^2 = d(\mu dV/dz)/dz - lambda
            //
            //-----------------------------------------------
            rsd[index(c_offset_V,j)]
                = (shear(x,j) - lambda(x,j) - rho_u(x,j)*dVdz(x,j)
                    - m_rho[j]*V(x,j)*V(x,j))/m_rho[j]
                - rdt*(V(x,j) - V_prev(j));
            diag[index(c_offset_V, j)] = 1;


            //-----------------------------------------------
            //    Species equations
            //
            //   \rho u dY_k/dz + dJ_k/dz + M_k\omega_k
            //
            //-----------------------------------------------
            getWdot(x,j);

            doublereal convec, diffus;
            for (k = 0; k < m_nsp; k++) {
                convec = rho_u(x,j)*dYdz(x,k,j)*pore[j];       //added porosity
                diffus = 2.0*(m_flux(k,j)*pore[j] - m_flux(k,j-1)*pore[j-1])   //added
porosity
                         /(z(j+1) - z(j-1));
                rsd[index(c_offset_Y + k, j)]
                    = (m_wt[k]*(wdot(k,j)*pore[j] )             //added porosity
                        - convec - diffus)/(m_rho[j]*pore[j])          //added porosity
                    - rdt*(Y(x,k,j) - Y_prev(k,j));
                diag[index(c_offset_Y + k, j)] = 1;
            }


            //-----------------------------------------------
            //    energy equation
            //-----------------------------------------------

            if (m_do_energy[j]) {
```

```
                    setGas(x,j);

                    // heat release term
                    const vector_fp& h_RT = m_thermo->enthalpy_RT_ref();
                    const vector_fp& cp_R = m_thermo->cp_R_ref();

                    sum = 0.0;
                    sum2 = 0.0;
                    doublereal flxk;
                    for (k = 0; k < m_nsp; k++) {
                        flxk = 0.5*(m_flux(k,j-1) + m_flux(k,j));
                        sum += wdot(k,j)*h_RT[k];
                        sum2 += flxk*cp_R[k]/m_wt[k];
                    }
                    sum *= GasConstant * T(x,j);
                    dtdzj = dTdz(x,j);
                    sum2 *= GasConstant * dtdzj;

                    rsd[index(c_offset_T, j)]   =
                        - m_cp[j]*rho_u(x,j)*dtdzj
                        - divHeatFlux(x,j) - sum - sum2;

                                rsd[index(c_offset_T, j)]   =
                                        //adding of convective term
                                        rsd[index(c_offset_T, j)] - (hconv[j]*(T(x,j)-
Tw[j]))/pore[j];       //

                    rsd[index(c_offset_T, j)] /= (m_rho[j]*m_cp[j]);

                    rsd[index(c_offset_T, j)] =
                        rsd[index(c_offset_T, j)] + m_efctr*(T_fixed(j) - T(x,j));

                    rsd[index(c_offset_T, j)] -= rdt*(T(x,j) - T_prev(j));
                    diag[index(c_offset_T, j)] = 1;
                }

                // residual equations if the energy equation is disabled

                if (!m_do_energy[j]) {
                    rsd[index(c_offset_T, j)] = T(x,j) - T_fixed(j);
                    diag[index(c_offset_T, j)] = 0;
                }

                rsd[index(c_offset_L, j)] = lambda(x,j) - lambda(x,j-1);
                diag[index(c_offset_L, j)] = 0;
            }
        }
    }

        //Solid solver
        void StFlow::solid(doublereal* x, vector<double> &hconv, vector<double>& scond,
vector<double>& RK, vector<double>&Omega,double & srho,double & sCp, double rdt) {
        writelog("Computing Solid Temperature Field...");

                double length=m_points;                         //
                Tw.resize(length);
                Twprev.resize(length);
                dq.resize(length);

                if (adapt==1)
                {
                int j=0;
                //Ensures solid profile has the correct number of points after an adaption
                for (int i=0;i<=length-1;i++)
                {
                        if (z(i)==zprev[j]) //same point
                        {
                                Twprev[i]=Twprev1[j];
                                j++;
                        }
                        else if (z(i)>zprev[j]) //deleted point
```

```
                        {
                                i--;
                                j++;
                        }
                        else if (z(i)<zprev[j])  //added point
                        {
                                if (i==1)
                                {
                                        Twprev[i]=Twprev[i-1];
                                }
                                else
                                {
                                        Twprev[i]=Twprev[i-1]+(Twprev[i-1]-Twprev[i-2])*((z(i)-z(i-
1))/(z(i-1)-z(i-2)));
                                }
                        }
                }
                }
                adapt=1;
                zprev.resize(length);
                Twprev1.resize(length);

                //Start of Conduction Radiation Stuff
                //
                //Vector Iinitialization
                vector<double> edia(length);
                vector<double> fdia(length);
                vector<double> gdia(length);
                vector<double> rhs(length);
                vector<double> dqnew(length);
                double sigma=0.0000000567;
                double change1=1;

                //Vector Population
                for(int i=0;i<=length-1;i++)
                {
                        dq[i]=0;
                }
                double T0=300;
                double T1=300;
                int count1=0;
                int fail1=0;
                while (change1>0.000001)
                {
                        count1=count1+1;
                        for(int i=0;i<=length-1;i++)
                        {
                                if (i==0)
                                {
                                        edia[i]=0;
                                        fdia[i]=1;
                                        gdia[i]=-1;
                                        rhs[i]=0;
                                }
                                else if (i==length-1)
                                {
                                        edia[i]=-1;
                                        fdia[i]=1;
                                        gdia[i]=0;
                                        rhs[i]=0;
                                }
                                else
                                {
                                        edia[i]=(2*scond[i])/((z(i)-z(i-1))*(z(i+1)-z(i-1)));
                                        fdia[i]=-(2*scond[i])/((z(i+1)-z(i))*(z(i+1)-z(i-1)))-
(2*scond[i])/((z(i)-z(i-1))*(z(i+1)-z(i-1)))-hconv[i]-srho*sCp*rdt;
                                        gdia[i]=(2*scond[i])/((z(i+1)-z(i))*(z(i+1)-z(i-1)));
                                        rhs[i]=-hconv[i]*T(x,i)+dq[i]-srho*sCp*rdt*Twprev[i];
                                }
                        }
```

```
//Decomposition
for(int i=1;i<=length-1;i++)
{
        edia[i]=edia[i]/fdia[i-1];
        fdia[i]=fdia[i]-edia[i]*gdia[i-1];
}

//Forward Substitution
for(int i=1;i<=length-1;i++)
{
        rhs[i]=rhs[i]-edia[i]*rhs[i-1];
}

//Back Substitution
Tw[length-1]=rhs[length-1]/fdia[length-1];
for(int i=length-2;i>=0;i--)
{
        Tw[i]=(rhs[i]-gdia[i]*Tw[i+1])/fdia[i];
}
T0=Tw[0];
T1=Tw[length-1];
//Radiation Time
//Vector Initialization
vector<double> qplus(length);
vector<double> qpnew(length);
vector<double> qminus(length);
vector<double> qmnew(length);
double change2=1;

//Vector Population
for(int i=0;i<=length-1;i++)
{
        double temp=T(x,i);
        double temp2;
        if (i==0)
        {
                qplus[i]=sigma*pow(temp,4);
                qpnew[i]=sigma*pow(temp,4);
                qminus[i]=0;
                qmnew[i]=0;
                temp2=temp;
        }
        else if (i==length-1)
        {
                qplus[i]=0;
                qpnew[i]=0;
                qminus[i]=sigma*pow(temp2,4);
                qmnew[i]=sigma*pow(temp2,4);
        }
        else
        {
                qplus[i]=0;
                qpnew[i]=0;
                qminus[i]=0;
                qmnew[i]=0;
        }
}
int count=0;
int fail=0;
S2 method
while (change2>0.000001)
{
        count=count+1;
        for(int i=1;i<=length-1;i++)
        {
                double temp=Tw[i];
                qpnew[i]=(qpnew[i-1]+RK[i]*(z(i)-z(i-
1))*Omega[i]*qminus[i]+2*RK[i]*(z(i)-z(i-1))*(1-Omega[i])*sigma*pow(temp,4))/(1+(z(i)-z(i-
1))*RK[i]*(2-Omega[i]));
        }
        for(int i=length-2;i>=0;i--)
```

```
                                {
                                        double temp=Tw[i];
                                        qmnew[i]=(qmnew[i+1]+RK[i]*(z(i+1)-
z(i))*Omega[i]*qpnew[i]+2*RK[i]*(z(i+1)-z(i))*(1-Omega[i])*sigma*pow(temp,4))/(1+(z(i+1)-
z(i))*RK[i]*(2-Omega[i]));
                                }
                                double norm1=0;
                                double norm2=0;
                                for(int i=0;i<=length-1;i++)
                                {
                                        norm1+=(qpnew[i]-qplus[i])*(qpnew[i]-qplus[i]);
                                        norm2+=(qmnew[i]-qminus[i])*(qmnew[i]-qminus[i]);
                                        qplus[i]=qpnew[i];
                                        qminus[i]=qmnew[i];
                                }
                                norm1=sqrt(norm1);
                                norm2=sqrt(norm2);
                                if (count>100)
                                {
                                        change2=0;
                                        fail=1;
                                }
                                else
                                {
                                        change2=max(norm1,norm2);
                                }
                        }
                        if (fail==1)
                        {
                                for(int i=0;i<=length-1;i++)
                                {
                                        dqnew[i]=dq[i];
                                }
                                writelog("Rad Stall");
                        }
                        else
                        {
                                for(int i=0;i<=length-1;i++)
                                {
                                        double temp=Tw[i];
                                        dqnew[i]=4*RK[i]*(1-Omega[i])*(sigma*pow(temp,4)-
0.5*qplus[i]-0.5*qminus[i]);
                                }
                        }
                        double norm=0;
                        double a=0.1;
                        for (int i=0;i<=length-1;i++)
                        {
                                norm+=(dqnew[i]-dq[i])*(dqnew[i]-dq[i]);
                                dq[i]=a*dqnew[i]+(1-a)*dq[i];
                        }
                        if (count1>400)
                        {
                                fail1=1;
                                change1=0;
                        }
                        else
                        {
                                change1=sqrt(norm);
                        }
                }
                if (fail1==1)
                {
                        for (int i=0;i<=length-1;i++)
                        {
                                Tw[i]=Twprev1[i];
                        }
                        writelog("Rad not Converged");
                }
                for (int i=0;i<=length-1;i++)
                {
```

```
                Twprev1[i]=Tw[i];
                zprev[i]=z(i);
            }
            writelog("Success\n");
            //
            //End of Newly Added Conduction Radiation Stuff
    }


/**
 * Update the transport properties at grid points in the range
 * from j0 to j1, based on solution x.
 */
void StFlow::updateTransport(doublereal* x,int j0, int j1) {
    int j,k,m;

    if (m_transport_option == c_Mixav_Transport) {
        for (j = j0; j < j1; j++) {
            setGasAtMidpoint(x,j);
            m_visc[j] = m_trans->viscosity();
            m_trans->getMixDiffCoeffs(DATA_PTR(m_diff) + j*m_nsp);
            m_tcon[j] = m_trans->thermalConductivity();
        }
    }
    else if (m_transport_option == c_Multi_Transport) {
        doublereal sum, sumx, wtm, dz;
        doublereal eps = 1.0e-12;
        for (m = j0; m < j1; m++) {
            setGasAtMidpoint(x,m);
            dz = m_z[m+1] - m_z[m];
            wtm = m_thermo->meanMolecularWeight();

            m_visc[j] = m_trans->viscosity();

            m_trans->getMultiDiffCoeffs(m_nsp,
                DATA_PTR(m_multidiff) + mindex(0,0,m));

            for (k = 0; k < m_nsp; k++) {
                sum = 0.0;
                sumx = 0.0;
                for (j = 0; j < m_nsp; j++) {
                    if (j != k) {
                        sum += m_wt[j]*m_multidiff[mindex(k,j,m)]*
                               ((X(x,j,m+1) - X(x,j,m))/dz + eps);
                        sumx += (X(x,j,m+1) - X(x,j,m))/dz;
                    }
                }
                m_diff[k + m*m_nsp] = sum/(wtm*(sumx+eps));
            }

            m_tcon[m] = m_trans->thermalConductivity();
            if (m_do_soret) {
                m_trans->getThermalDiffCoeffs(m_dthermal.ptrColumn(0) + m*m_nsp);
            }
        }
    }
}


//-------------------------------------------------------

/**
 *  Evaluate the residual function for axisymmetric stagnation
 *  flow. If jpt is less than zero, the residual function is
 *  evaluated at all grid points. If jpt >= 0, then the residual
 *  function is only evaluated at grid points jpt-1, jpt, and
 *  jpt+1. This option is used to efficiently evaluate the
 *  Jacobian numerically.
 *
 */
```

```
void FreeFlame::eval(int jg, doublereal* xg,
    doublereal* rg, integer* diagg, doublereal rdt) {

    // if evaluating a Jacobian, and the global point is outside
    // the domain of influence for this domain, then skip
    // evaluating the residual
    if (jg >=0 && (jg < firstPoint() - 1 || jg > lastPoint() + 1)) return;

    // if evaluating a Jacobian, compute the steady-state residual
    if (jg >= 0) rdt = 0.0;

    // start of local part of global arrays
    doublereal* x = xg + loc();
    doublereal* rsd = rg + loc();
    integer* diag = diagg + loc();

    int jmin, jmax, jpt;
    jpt = jg - firstPoint();

    if (jg < 0) {      // evaluate all points
        jmin = 0;
        jmax = m_points - 1;
    }
    else {             // evaluate points for Jacobian
        jmin = max(jpt-1, 0);
        jmax = min(jpt+1,m_points-1);
    }

    // properties are computed for grid points from j0 to j1
    int j0 = max(jmin-1,0);
    int j1 = min(jmax+1,m_points-1);


    int j, k;


    //------------------------------------------------------
    //              update properties
    //------------------------------------------------------

    // update thermodynamic properties only if a Jacobian is not
    // being evaluated
    if (jpt < 0) {
        updateThermo(x, j0, j1);
        updateTransport(x, j0, j1);
    }

    // update the species diffusive mass fluxes whether or not a
    // Jacobian is being evaluated
    updateDiffFluxes(x, j0, j1);


    //------------------------------------------------------
    // evaluate the residual equations at all required
    // grid points
    //------------------------------------------------------

    doublereal sum, sum2, dtdzj;

    for (j = jmin; j <= jmax; j++) {


        //-----------------------------------------------
        //         left boundary
        //-----------------------------------------------

        if (j == 0) {

            // these may be modified by a boundary object

            // Continuity. This propagates information right-to-left,
```

```
    // since rho_u at point 0 is dependent on rho_u at point 1,
    // but not on mdot from the inlet.
     rsd[index(c_offset_U,0)] =
         -(rho_u(x,1) - rho_u(x,0))/m_dz[0]
         -(density(1)*V(x,1) + density(0)*V(x,0));

     // the inlet (or other) object connected to this one
     // will modify these equations by subtracting its values
     // for V, T, and mdot. As a result, these residual equations
     // will force the solution variables to the values for
     // the boundary object
     rsd[index(c_offset_V,0)] = V(x,0);
     rsd[index(c_offset_T,0)] = T(x,0);
     rsd[index(c_offset_L,0)] = -rho_u(x,0);

     // The default boundary condition for species is zero
     // flux
     sum = 0.0;
     for (k = 0; k < m_nsp; k++) {
         sum += Y(x,k,0);
         rsd[index(c_offset_Y + k, 0)] =
             -(m_flux(k,0) + rho_u(x,0)* Y(x,k,0));
     }
     rsd[index(c_offset_Y, 0)] = 1.0 - sum;
}


//-----------------------------------------------
//
//         right boundary
//
//-----------------------------------------------

else if (j == m_points - 1) {

     // the boundary object connected to the right of this
     // one may modify or replace these equations. The
     // default boundary conditions are zero u, V, and T,
     // and zero diffusive flux for all species.

    // zero gradient
    rsd[index(0,j)] = rho_u(x,j) - rho_u(x,j-1);
    rsd[index(1,j)] = V(x,j);
    rsd[index(2,j)] = T(x,j) - T(x,j-1);
    doublereal sum = 0.0;
    rsd[index(c_offset_L, j)] = lambda(x,j) - lambda(x,j-1);
    diag[index(c_offset_L, j)] = 0;
    for (k = 0; k < m_nsp; k++) {
        sum += Y(x,k,j);
        rsd[index(k+4,j)] = m_flux(k,j-1) + rho_u(x,j)*Y(x,k,j);
    }
    rsd[index(4,j)] = 1.0 - sum;
    diag[index(4,j)] = 0;
}

//-----------------------------------------
//     interior points
//-----------------------------------------

else {

    //-----------------------------------------------
    //    Continuity equation
    //-----------------------------------------------

    if (grid(j) > m_zfixed){
        rsd[index(c_offset_U,j)] =
            - (rho_u(x,j) - rho_u(x,j-1))/m_dz[j-1]
            - (density(j-1)*V(x,j-1) + density(j)*V(x,j));
    }
```

```
else if (grid(j) == m_zfixed){
    if (m_do_energy[j]) {
        rsd[index(c_offset_U,j)] = (T(x,j) - m_tfixed);
    }
    else {
        rsd[index(c_offset_U,j)] = (rho_u(x,j)
            - m_rho[0]*0.3);
    }
}
else if(grid(j) < m_zfixed){
    rsd[index(c_offset_U,j)] =
        - (rho_u(x,j+1) - rho_u(x,j))/m_dz[j]
        - (density(j+1)*V(x,j+1) + density(j)*V(x,j));
}
//algebraic constraint
diag[index(c_offset_U, j)] = 0;

//-----------------------------------------------
//    Radial momentum equation
//
//    \rho u dV/dz + \rho V^2 = d(\mu dV/dz)/dz - lambda
//
//-----------------------------------------------
rsd[index(c_offset_V,j)]
    = (shear(x,j) - lambda(x,j) - rho_u(x,j)*dVdz(x,j)
        - m_rho[j]*V(x,j)*V(x,j))/m_rho[j]
    - rdt*(V(x,j) - V_prev(j));
diag[index(c_offset_V, j)] = 1;


//-----------------------------------------------
//    Species equations
//
//   \rho u dY_k/dz + dJ_k/dz + M_k\omega_k
//
//-----------------------------------------------
getWdot(x,j);

doublereal convec, diffus;
for (k = 0; k < m_nsp; k++) {
    convec = rho_u(x,j)*dYdz(x,k,j);
    diffus = 2.0*(m_flux(k,j) - m_flux(k,j-1))
             /(z(j+1) - z(j-1));
    rsd[index(c_offset_Y + k, j)]
        = (m_wt[k]*(wdot(k,j) )
            - convec - diffus)/m_rho[j]
        - rdt*(Y(x,k,j) - Y_prev(k,j));
    diag[index(c_offset_Y + k, j)] = 1;
}


//-----------------------------------------------
//    energy equation
//-----------------------------------------------

if (m_do_energy[j]) {

    setGas(x,j);

    // heat release term
    const vector_fp& h_RT = m_thermo->enthalpy_RT_ref();
    const vector_fp& cp_R = m_thermo->cp_R_ref();

    sum = 0.0;
    sum2 = 0.0;
    doublereal flxk;
    for (k = 0; k < m_nsp; k++) {
        flxk = 0.5*(m_flux(k,j-1) + m_flux(k,j));
        sum += wdot(k,j)*h_RT[k];
        sum2 += flxk*cp_R[k]/m_wt[k];
    }
```

```
                sum *= GasConstant * T(x,j);
                dtdzj = dTdz(x,j);
                sum2 *= GasConstant * dtdzj;

                rsd[index(c_offset_T, j)]   =
                    - m_cp[j]*rho_u(x,j)*dtdzj
                    - divHeatFlux(x,j) - sum - sum2;
                rsd[index(c_offset_T, j)] /= (m_rho[j]*m_cp[j]);

                rsd[index(c_offset_T, j)] =
                    rsd[index(c_offset_T, j)] + m_efctr*(T_fixed(j) - T(x,j));

                rsd[index(c_offset_T, j)] -= rdt*(T(x,j) - T_prev(j));
                diag[index(c_offset_T, j)] = 1;
            }
            // residual equations if the energy equation is disabled
            else {
                rsd[index(c_offset_T, j)] = T(x,j) - T_fixed(j);
                diag[index(c_offset_T, j)] = 0;
            }

            rsd[index(c_offset_L, j)] = lambda(x,j) - lambda(x,j-1);
            diag[index(c_offset_L, j)] = 0;
        }
    }
}


/**
 * Print the solution.
 */
void StFlow::showSolution(const doublereal* x) {
    int nn = m_nv/5;
    int i, j, n;
    //char* buf = new char[100];
    char buf[100];

    // The mean molecular weight is needed to convert
    updateThermo(x, 0, m_points-1);

    sprintf(buf, "    Pressure:  %10.4g Pa \n", m_press);
    writelog(buf);
    for (i = 0; i < nn; i++) {
        st_drawline();
        sprintf(buf, "\n        z   ");
        writelog(buf);
        for (n = 0; n < 5; n++) {
            sprintf(buf, " %10s ",componentName(i*5 + n).c_str());
            writelog(buf);
        }
        st_drawline();
        for (j = 0; j < m_points; j++) {
            sprintf(buf, "\n %10.4g ",m_z[j]);
            writelog(buf);
            for (n = 0; n < 5; n++) {
                sprintf(buf, " %10.4g ",component(x, i*5+n,j));
                writelog(buf);
            }
        }
        writelog("\n");
    }
    int nrem = m_nv - 5*nn;
    st_drawline();
    sprintf(buf, "\n        z   ");
    writelog(buf);
    for (n = 0; n < nrem; n++) {
        sprintf(buf, " %10s ", componentName(nn*5 + n).c_str());
        writelog(buf);
    }
    st_drawline();
    for (j = 0; j < m_points; j++) {
```

```
            sprintf(buf, "\n %10.4g ",m_z[j]);
            writelog(buf);
            for (n = 0; n < nrem; n++) {
                sprintf(buf, " %10.4g ",component(x, nn*5+n,j));
                writelog(buf);
            }
        }
        writelog("\n");
    }


    /**
     * Update the diffusive mass fluxes.
     */
    void StFlow::updateDiffFluxes(const doublereal* x, int j0, int j1) {
        int j, k, m;
        doublereal sum, wtm, rho, dz, gradlogT;

        switch (m_transport_option) {

        case c_Mixav_Transport:
        case c_Multi_Transport:
            for (j = j0; j < j1; j++) {
                sum = 0.0;
                wtm = m_wtm[j];
                rho = density(j);
                dz = z(j+1) - z(j);

                for (k = 0; k < m_nsp; k++) {
                    m_flux(k,j) = m_wt[k]*(rho*m_diff[k+m_nsp*j]/wtm);
                    m_flux(k,j) *= (X(x,k,j) - X(x,k,j+1))/dz;
                    sum -= m_flux(k,j);
                }
                // correction flux to insure that \sum_k Y_k V_k = 0.
                for (k = 0; k < m_nsp; k++) m_flux(k,j) += sum*Y(x,k,j);
            }
            break;

        default:
            throw CanteraError("updateDiffFluxes","unknown transport model");
        }

        if (m_do_soret) {
            for (m = j0; m < j1; m++) {
                gradlogT = 2.0*(T(x,m+1) - T(x,m))/(T(x,m+1) + T(x,m));
                for (k = 0; k < m_nsp; k++) {
                    m_flux(k,m) -= m_dthermal(k,m)*gradlogT;
                }
            }
        }
    }


    string StFlow::componentName(int n) const {
        switch(n) {
        case 0: return "u";
        case 1: return "V";
        case 2: return "T";
        case 3: return "lambda";
        default:
            if (n >= (int) c_offset_Y && n < (int) (c_offset_Y + m_nsp)) {
                    return m_thermo->speciesName(n - c_offset_Y);
            }
            else
                return "<unknown>";
        }
    }


       //added by Karl Meredith
    int StFlow::componentIndex(string name) const {
```

```
    if(name=="u") {return 0;}
    else if (name=="V") {return 1;}
    else if (name=="T") {return 2;}
    else if (name=="lambda") {return 3;}
    else {
        for (int n=4;n<m_nsp+4;n++){
            if(componentName(n)==name){
                return n;
            }
        }
    }

    return -1;
}


void StFlow::restore(const XML_Node& dom, doublereal* soln) {

    vector<string> ignored;
    int nsp = m_thermo->nSpecies();
    vector_int did_species(nsp, 0);

    vector<XML_Node*> str;
    dom.getChildren("string",str);
    int nstr = static_cast<int>(str.size());
    for (int istr = 0; istr < nstr; istr++) {
        const XML_Node& nd = *str[istr];
        writelog(nd["title"]+": "+nd.value()+"\n");
    }

    //map<string, double> params;
    double pp = -1.0;
    pp = getFloat(dom, "pressure", "pressure");
    setPressure(pp);


    vector<XML_Node*> d;
    dom.child("grid_data").getChildren("floatArray",d);
    int nd = static_cast<int>(d.size());

    vector_fp x;
    int n, np = 0, j, ks, k;
    string nm;
    bool readgrid = false, wrote_header = false;
    for (n = 0; n < nd; n++) {
        const XML_Node& fa = *d[n];
        nm = fa["title"];
        if (nm == "z") {
            getFloatArray(fa,x,false);
            np = x.size();
            writelog("Grid contains "+int2str(np)+
                " points.\n");
            readgrid = true;
            setupGrid(np, DATA_PTR(x));
        }
    }
    if (!readgrid) {
        throw CanteraError("StFlow::restore",
            "domain contains no grid points.");
    }

    writelog("Importing datasets:\n");
    for (n = 0; n < nd; n++) {
        const XML_Node& fa = *d[n];
        nm = fa["title"];
        getFloatArray(fa,x,false);
        if (nm == "u") {
            writelog("axial velocity   ");
            if ((int) x.size() == np) {
```

```
            for (j = 0; j < np; j++) {
                soln[index(0,j)] = x[j];
            }
        }
        else {
            goto error;
        }
    }
    else if (nm == "z") {
        ;   // already read grid
    }
    else if (nm == "V") {
        writelog("radial velocity   ");
        if ((int) x.size() == np) {
            for (j = 0; j < np; j++)
                soln[index(1,j)] = x[j];
        }
        else goto error;
    }
    else if (nm == "T") {
        writelog("temperature   ");
        if ((int) x.size() == np) {
            for (j = 0; j < np; j++)
                soln[index(2,j)] = x[j];

            // For fixed-temperature simulations, use the
            // imported temperature profile by default.  If
            // this is not desired, call setFixedTempProfile
            // *after* restoring the solution.

            vector_fp zz(np);
            for (int jj = 0; jj < np; jj++)
                zz[jj] = (grid(jj) - zmin())/(zmax() - zmin());
            setFixedTempProfile(zz, x);
        }
        else goto error;
    }
    else if (nm == "L") {
        writelog("lambda   ");
        if ((int) x.size() == np) {
            for (j = 0; j < np; j++)
                soln[index(3,j)] = x[j];
        }
        else goto error;
    }
    else if (m_thermo->speciesIndex(nm) >= 0) {
        writelog(nm+"   ");
        if ((int) x.size() == np) {
            k = m_thermo->speciesIndex(nm);
            did_species[k] = 1;
            for (j = 0; j < np; j++)
                soln[index(k+4,j)] = x[j];
        }
    }
    else
        ignored.push_back(nm);
}

if (ignored.size() != 0) {
    writelog("\n\n");
    writelog("Ignoring datasets:\n");
    int nn = static_cast<int>(ignored.size());
    for (int n = 0; n < nn; n++) {
        writelog(ignored[n]+"   ");
    }
}

for (ks = 0; ks < nsp; ks++) {
    if (did_species[ks] == 0) {
        if (!wrote_header) {
            writelog("Missing data for species:\n");
```

```
                    wrote_header = true;
                }
                writelog(m_thermo->speciesName(ks)+" ");
            }
        }

        return;
 error:
        throw CanteraError("StFlow::restore","Data size error");
    }



    void StFlow::save(XML_Node& o, doublereal* sol) {
        int k;

        ArrayViewer soln(m_nv, m_points, sol + loc());

                //Added to provide Tw as an output
                ofstream f("Solid.txt");
                ofstream f2("Rad.txt");
                ofstream f3("Tout.txt");
                double length=Tw.size();
                for (int i=0;i<=length-1;i++)
                {
                        f<<Tw[i]<<endl;
                        f2<<dq[i]<<endl;
                }
                f3<<Tw[length-1]<<endl;
                f.close();
                f2.close();
                f3.close();
                //

        XML_Node& flow = (XML_Node&)o.addChild("domain");
        flow.addAttribute("type",flowType());
        flow.addAttribute("id",m_id);
        flow.addAttribute("points",m_points);
        flow.addAttribute("components",m_nv);

        if (m_desc != "") addString(flow,"description",m_desc);
        XML_Node& gv = flow.addChild("grid_data");
        addFloat(flow, "pressure", m_press, "Pa", "pressure");
        addFloatArray(gv,"z",m_z.size(),DATA_PTR(m_z),
            "m","length");
               vector_fp x(static_cast<size_t>(soln.nColumns()));

                soln.getRow(0,DATA_PTR(x));
                addFloatArray(gv,"u",x.size(),DATA_PTR(x),"m/s","velocity");

        soln.getRow(1,DATA_PTR(x));
        addFloatArray(gv,"V",
            x.size(),DATA_PTR(x),"1/s","rate");

        soln.getRow(2,DATA_PTR(x));
        addFloatArray(gv,"T",x.size(),DATA_PTR(x),"K","temperature",0.0);

        soln.getRow(3,DATA_PTR(x));
        addFloatArray(gv,"L",x.size(),DATA_PTR(x),"N/m^4");

        for (k = 0; k < m_nsp; k++) {
            soln.getRow(4+k,DATA_PTR(x));
            addFloatArray(gv,m_thermo->speciesName(k),
                x.size(),DATA_PTR(x),"","massFraction",0.0,1.0);
        }
    }
    void StFlow::setJac(MultiJac* jac) {
        m_jac = jac;
    }

}  // namespace
```

Appendix D: Cantera Interface and Optimization Code

This section will detail how to setup Cantera, starting from downloading the files up to compiling the code. It will also include all of the code the author wrote, for interfacing with Cantera, the RSM algorithm, the GRG method algorithm, and any other functions that were useful to the problem.

D.1 Using Cantera

Here we will layout the process involved to setup Cantera. First, Python version 2.4 or greater needs to be installed on your computer. If you do not have it, it can be found at http://www.python.org/ftp/python/2.5/python-2.5.msi. Once it is installed, you must download and add the nummarray add on, which can be found at http://sourceforge.net/projects/numpy. Be sure to download the installer for numarray, not the newer numpy package, as Cantera does not support this new version. Next, some environmental variables need to be set on your PC. From the control panel select system, press the advanced tab, and press the environmental variables button. Here we will create two new user variables called PYTHON_CMD and MATLAB_CMD with the value being set to the respective programs directory. These variables allow Cantera to access the programs while compiling. The last setup step is to configure MATLab. To do this, open MATLab and type "mex –setep" at the prompt. A window will pop-up where you need to select Visual C++ from your list of compilers.

Now that your system is properly configured you should download the latest version of Cantera from http://sourceforge.net/projects/cantera, which was 1.7.0 at the time of publication, and extract the files into a temporary location of your choosing. You will also need to download the SUNDIALS package, which Cantera uses to solve equations, from www.llnl.gov/casc/sundials/. Extract these files into the same temporary folder as Cantera, as

the Cantera compiler is setup to build SUNDIALS and itself simultaneously. Open Visual C++ and open cantera.sln found in the cantera\win32\vc7 directory. You may be prompted to convert the solution to a newer format, click finish to do so. Open the file arith.h found under the f2c_libs->Header Files directory on the left side and change the line ending style to Unix when prompted. Finally, change the configuration to Release and select Build Selection from the Build menu. If your version of Python was not built with the same version of Visual C++ you will receive an error upon completion of the build. Cantera will still work, however the Python component will not. To ensure that MATLab can access Cantera the files clib.dll, from the directory cantera\build\lib\i686-pc-win32, and the file msvcr80.dll, found on your PC, must be copied into the directory cantera\Cantera\matlab\cantera. For the final step change the name of the src file to kernel in the directory cantera\Cantera. You are now ready to use Cantera.

D.2 Interface and Optimization Code

The author's code to solve the optimization problem of maximizing the radiant efficiency of a two stage porous radiant burner is given below.

```
#include <iostream>
#include <vector>
#include <fstream>
#include <time.h>
#include <Cantera.h>
#include <onedim.h>
#include <IdealGasMix.h>
#include <equilibrium.h>
#include <transport.h>

using namespace std;
using namespace Cantera;

double Combust(vector<double> vars);
int RSM();
int RSM2D();
double GRGM(vector<double> &x0, double f0, double alpha, double lcon, double rcon, int dim);
double norm(vector<double> x);
double inprod(vector<double>x, vector<double>y);
vector<double> Mv(double **M,vector<double> v,double row, double col);
double **MM(double **M1, double **M2,double a, double b, double c);
vector<double> LUSolve(double **M,vector<double> v);
double sum(vector<double> x);
double newfun(vector<double> x, vector<double> b, double &g, double &H);
double newfun2D(vector<double> x, vector<double> b, vector<double> &grad, double **&Hess);
```

```
double fun2(double x);
double fun2D(double x,double y);
double **MT(double **M,double n, double p);
double max(double x, double y);
double min(double x, double y);
int sign(double x);
double f;
double fnew;
vector<double> g;
vector<double> gnew;
vector<double> xnew;



int main()
{
        //Setup Timer
        clock_t start;
        clock_t end;
        double duration;
        start=clock();

        //int dude=RSM();
        int dude=RSM2D();


        //End Timer
        end=clock();
        duration=((double) (end - start)) / CLOCKS_PER_SEC;
        cout<<duration<<endl;

        system("PAUSE");
        return(0);
}

double Combust(vector<double> vars)
{
        //New Parameters to allow for correlations
        //Define Material Properties
        double pore1=0.835;
        double pore2=vars[1];
        double dpore1=.00029;
        double dpore2=vars[0];
        double Omega1=0.8;
        double Omega2=0.8;
        double srho=510;
        double sCp=824;

        //Export Proerties to fill
        ofstream fid("Properties2.txt");
        fid<<pore1<<endl;
        fid<<pore2<<endl;
        fid<<dpore1<<endl;
        fid<<dpore2<<endl;
        fid<<Omega1<<endl;
        fid<<Omega2<<endl;
        fid<<srho<<endl;
        fid<<sCp<<endl;
        fid.close();

        //Define Fluid Properties
        double P=OneAtm;
        double Tburner=300;
        double u0=0.45;
        double phi=.65;
        IdealGasMix gas("drm19.cti","drm19");
        int nsp=gas.nSpecies();
        vector_fp x;
    x.resize(nsp);
        for(int k=0;k<nsp;k++)
```

```
    {
            if(k==gas.speciesIndex("CH4"))
            {
                    x[k]=1.0;
            }
            else if(k==gas.speciesIndex("O2"))
            {
                    x[k]=0.21/phi/.105;
            }
            else if(k==gas.speciesIndex("N2"))
            {
                    x[k]=0.78/phi/.105;
            }
            else if(k==gas.speciesIndex("AR"))
            {
                    x[k]=0.01/phi/.105;
            }
            else
            {
                    x[k]=0.0;
            }
}
    gas.setState_TPX(Tburner,P,DATA_PTR(x));
    double rhoin=gas.density();
    double *yin=new double[nsp];
    gas.getMassFractions(yin);
    equilibrate(gas,"HP");
    double rhoout=gas.density();
    double Tad=gas.temperature();
    double *yeq=new double[nsp];
    gas.getMassFractions(yeq);

    //Create the grid
    double *grid=new double[301];
    double dz1=0.0006;
    double dz2=0.00005;
    double dz3=0.00041;
    for (int i=0;i<301;i++)
    {
            if (i<=50)
            {
                    grid[i]=((double)i)*dz1;
            }
            else if (i<=250)
            {
                    grid[i]=.03+((double)(i-50))*dz2;
            }
            else
            {
                    grid[i]=.04+((double)(i-250))*dz3;
            }
    }

    //Create the flow object
    AxiStagnFlow flow(&gas);
    flow.setupGrid(301,grid);
    Transport* tr=newTransportMgr("Mix",&gas);
    flow.setTransport(*tr);
    flow.setKinetics(gas);
    flow.setPressure(P);

    //Create the inlet
    Inlet1D inlet;
    inlet.setMoleFractions(DATA_PTR(x));
    double mdot=u0*rhoin;
    inlet.setMdot(mdot);
    inlet.setTemperature(Tburner);

    //Create the outlet
    Outlet1D outlet;
```

```
//Create the flame object
vector<Domain1D*> domains;
domains.push_back(&inlet);
domains.push_back(&flow);
domains.push_back(&outlet);
Sim1D flame(domains);

//Build initial guess
vector_fp locs;
vector_fp value;
double z1=0.55;
double z2=0.62;
double uout=inlet.mdot()/rhoout;
//Velocity Profile
locs.resize(2);
value.resize(2);
locs[0]=0;
locs[1]=1;
value[0]=u0;
value[1]=uout;
flame.setInitialGuess("u",locs,value);
//Species Profiles
locs.resize(3);
value.resize(3);
locs[0]=0;
locs[1]=z1;
locs[2]=1;
for (int i=0;i<nsp;i++)
{
        value[0]=yin[i];
        value[1]=yeq[i];
        value[2]=yeq[i];
        flame.setInitialGuess(gas.speciesName(i),locs,value);
}
//Temperature Profile
locs.resize(4);
value.resize(4);
locs[0]=0;
locs[1]=z1;
locs[2]=z2;
locs[3]=1;
value[0]=Tburner;
value[1]=Tburner;
value[2]=2000;
value[3]=Tad;
flame.setInitialGuess("T",locs,value);
//Reset Inlet
inlet.setMoleFractions(DATA_PTR(x));
inlet.setMdot(mdot);
inlet.setTemperature(Tburner);

//Set solver parameters
int loglevel=1;
bool refine_grid=false;
double rtolSS=1.0e-4;
double atolSS=1.0e-9;
double rtolTS=1.0e-4;
double atolTS=1.0e-9;
flow.setTolerancesSS(rtolSS,atolSS);
flow.setTolerancesTS(rtolTS,atolTS);
double SSJacAge=5;
double TSJacAge=10;
flame.setJacAge(SSJacAge,TSJacAge);

//Solve and Save
flame.solve(loglevel,refine_grid);
refine_grid=true;
int flowdomain=1;
double ratio=4;
double slope=0.4;
double curve=0.4;
```

```
        double prune=0.001;
        flame.setRefineCriteria(flowdomain,ratio,slope,curve,prune);
        flow.solveEnergyEqn();
        flame.solve(loglevel,refine_grid);
        flame.save("gradienttest.xml","run","solution with energy equation");
        flame.writeStats();

        ifstream in("Tout.txt");
        double input;
        in>>input;
        double Tout=input;
        double eff=pow(Tout,4)/pow(Tad,4);
        return(eff);
}

int RSM()
{
        vector<double> x0(1);
        vector<double> xprev(1);
        x0[0]=1.52;
        xprev[0]=x0[0];
        //Selecting starting points
        vector<double> points(5);
        points[0]=1.37;
        points[1]=1.4075;
        points[2]=1.445;
        points[3]=1.4825;
        points[4]=1.52;
        double alpha=0.0375;
        //Initialize quantities
        double length=points.size();
        vector<double> f(length);
        double diff1=10000000000000000000;
        double diff2=10000000000000000000;
        int minloc=0;
        int maxloc=4;
        double lcon=-100000000000000;  //Left Constraint
        double rcon=10000000000000000;   //Right Constraint
        int count=0;        //Added to control number of shrinks
        double error;
        f[0]=fun2(points[0]);
        cout<<endl<<endl<<points[0]<<"   "<<f[0]<<endl<<endl;
        //Loop until the model and function have same value or the model points solution is the
same
        while ((diff1>0.0001) & (diff2>0.0001))
        {
                for (int i=1;i<length;i++)
                {
                        f[i]=fun2(points[i]);
                        cout<<endl<<endl<<points[i]<<"   "<<f[i]<<endl<<endl;
                }
                //Perform Least Squares fit
                double n=length;
                int k=1; //This is equal to the number of variables;
                double p=2*k+1;  //Number of regressor variable.  Need to change for higher order.

                double **X;
                X=new double* [n];
                for (int i=0;i<n;i++)
                {
                        *(X+i)=new double[p];
                }

                for (int i=0;i<n;i++)
                {
                        for (int c=0;c<k;c++)
                        {
                                X[i][c]=1;
                                X[i][c+1]=points[i];
                                X[i][c+k+1]=pow(points[i],2);
                        }
```

```
}
double **Xt;
Xt=new double* [p];
for (int i=0;i<p;i++)
{
        *(Xt+i)=new double[n];
}
Xt=MT(X,n,p);

double **A;
A=new double* [p];
for (int i=0;i<p;i++)
{
        *(A+i)=new double[p];
}
A=MM(Xt,X,p,n,p);
vector<double> c(p);
c=Mv(Xt,f,p,n);

vector<double> b(p);
b=LUSolve(A,c);
//Perform Newtons Method
double g;
double H;
double fmin=newfun(x0,b,g,H);
if (H<0)
{
        double g1;
        double H1;
        vector<double> p(1);
        p[0]=points[minloc];
        double b1=newfun(p,b,g1,H1);
        double g2;
        double H2;
        p[0]=points[maxloc];
        double b2=newfun(p,b,g2,H2);
        if (b1<=b2)
        {
                fmin=b1;
                x0[0]=points[minloc];
        }
        else
        {
                fmin=b2;
                x0[0]=points[maxloc];
        }
}
else
{
        double d=-g/H;
        double temp;  //Added to stop solver from going beyond model range.
        temp=x0[0]+d;
        if (temp<points[minloc])
        {
                x0[0]=points[minloc];
        }
        else if (temp>points[maxloc])
        {
                x0[0]=points[maxloc];
        }
        else
        {
                x0[0]=x0[0]+d;
                if (count<3)
                {
                        alpha=alpha/2;//Shrinks if convex and inside box
                        count=count+1;
                }
                else
                {
                        xprev[0]=x0[0];
```

```
                        }
                }
                fmin=newfun(x0,b,g,H);
        }
        cout<<x0[0]<<endl<<endl;
        double fnew=fun2(x0[0]);

        int ignore=0;
        if (fnew>f[0])
        {
                x0[0]=points[0];
                fnew=f[0];
                xprev[0]=x0[0];
                ignore=1;
        }


        //Error Stuff
        if (ignore==0)
        {
                vector<double> xm(p);
                xm[0]=1;
                xm[1]=x0[0];
                xm[2]=pow(x0[0],2);
                vector<double> yhat(n);
                yhat=Mv(X,b,n,p);
                vector<double> fsurf(n);
                for (int i=0;i<n;i++)
                {
                        vector<double> node(1);
                        node[0]=points[0];
                        fsurf[i]=pow((f[i]-yhat[i]),2)/(n-p);
                }
                double temp=sum(fsurf);
                double s=sqrt(temp);
                vector<double> error1(p);
                error1=LUSolve(A,xm);
                error=inprod(xm,error1);
                error=s*2.919986*sqrt(error);  //The number is for 90% confidence from
students t
        }

        cout<<endl<<endl<<x0[0]<<"   "<<fnew<<endl<<endl;
        //Update loop ending parameters
        diff1=sqrt(pow((fnew-fmin)/fnew,2));
        vector<double> xtemp(1);
        xtemp[0]=x0[0]-xprev[0];
        xprev[0]=x0[0];
        diff2=norm(xtemp);
        cout<<diff1<<"   "<<diff2<<endl;

        f[0]=fnew;
        //Update points
        points[0]=x0[0];
        if (x0[0]<(lcon+2*alpha))
        {
                if (x0[0]<=(lcon+alpha))
                {
                        if (x0[0]<=lcon)
                        {
                                if (count<3)
                                {
                                        alpha=alpha/2;//Shrink if on min edge
                                        count=count+1;
                                }
                                points[0]=lcon;
                                points[1]=points[0]+alpha;
                                points[2]=points[1]+alpha;
                                points[3]=points[2]+alpha;
                                points[4]=points[3]+alpha;
                                minloc=0;
```

```
                        maxloc=4;
                }
                else
                {
                        points[1]=lcon;
                        points[2]=points[0]+alpha;
                        points[3]=points[2]+alpha;
                        points[4]=points[3]+alpha;
                        minloc=1;
                        maxloc=4;
                }
        }
        else
        {
                points[1]=lcon;
                points[2]=points[0]-alpha;
                points[3]=points[0]+alpha;
                points[4]=points[3]+alpha;
                minloc=1;
                maxloc=4;
        }
}
else if (x0[0]>(rcon-2*alpha))
{
        if (x0[0]>=(rcon-alpha))
        {
                if (x0[0]>=rcon)
                {
                        if (count<3)
                        {
                                alpha=alpha/2;//Shrink if on max edge
                                count=count+1;
                        }
                        points[0]=rcon;
                        points[1]=points[0]-alpha;
                        points[2]=points[1]-alpha;
                        points[3]=points[2]-alpha;
                        points[4]=points[3]-alpha;
                        minloc=4;
                        maxloc=0;
                }
                else
                {
                        points[1]=rcon;
                        points[2]=points[0]-alpha;
                        points[3]=points[2]-alpha;
                        points[4]=points[3]-alpha;
                        minloc=4;
                        maxloc=1;
                }
        }
        else
        {
                points[1]=rcon;
                points[2]=points[0]+alpha;
                points[3]=points[0]-alpha;
                points[4]=points[3]-alpha;
                minloc=4;
                maxloc=1;
        }
}
else
{
        points[1]=points[0]-alpha;
        points[2]=points[1]-alpha;
        points[3]=points[0]+alpha;
        points[4]=points[3]+alpha;
        minloc=2;
        maxloc=4;
}
}
```

```
        cout<<"x*="<<x0[0]<<" with fmax="<<f[0]<<"+/-"<<error<<endl;
        return(0);
}

int RSM2D()
{
        vector<double> x0(2);
        vector<double> xprev(2);
        x0[0]=1.445;
        x0[1]=0.88;
        xprev[0]=x0[0];
        xprev[1]-x0[1];
        //Selecting starting points
        double **points;
        points=new double* [9];
        for (int i=0;i<9;i++)
        {
                *(points+i)=new double[2];
        }
        points[0][0]=1.37;
        points[1][0]=1.37;
        points[2][0]=1.37;
        points[3][0]=1.445;
        points[4][0]=1.445;
        points[5][0]=1.445;
        points[6][0]=1.52;
        points[7][0]=1.52;
        points[8][0]=1.52;

        points[0][1]=.87;
        points[1][1]=.88;
        points[2][1]=.89;
        points[3][1]=.87;
        points[4][1]=.88;
        points[5][1]=.89;
        points[6][1]=.87;
        points[7][1]=.88;
        points[8][1]=.89;

        double alpha1=0.075;
        double alpha2=0.01;

        //Initialize quantities
        double length=9;
        vector<double> f(length);
        double diff1=10000000000000000000;
        double diff2=10000000000000000000;
        double lcon=0.69;
        double rcon=1.52;
        double bcon=0.865;
        double tcon=0.95;
        double error;
        int flagl=0;
        int flagr=0;
        int flagb=0;
        int flagt=0;
        int count1=0;
        int count2=0;
        int nshrinks=4;
        int noshrinkh=0;
        int noshrinkv=0;
        int start=0;
        f[0]=fun2D(points[0][0],points[0][1]);
        //Loop until the model points solution is the same
        while (diff2>0.0001)
        {
                for (int i=1;i<length;i++)
                {
                        f[i]=fun2D(points[i][0],points[i][1]);
                }
                //Perform Least Squares fit
```

```
double n=length;
int k=2;
double p=2*k+2;

double **X;
X=new double* [n];
for (int i=0;i<n;i++)
{
        *(X+i)=new double[p];
}

for (int i=0;i<n;i++)
{
        X[i][0]=1;
        for (int c=0;c<k;c++)
        {
                X[i][c+1]=points[i][c];
                X[i][c+k+1]=pow(points[i][c],2);
        }
        for (int c=0;c<k-1;c++)
        {
                for (int j=c;j<k;j++)
                {
                        X[i][c+2*k+1]=points[i][c]*points[i][j];
                }
        }
}

double **Xt;
Xt=new double* [p];
for (int i=0;i<p;i++)
{
        *(Xt+i)=new double[n];
}
Xt=MT(X,n,p);

double **A;
A=new double* [p];
for (int i=0;i<p;i++)
{
        *(A+i)=new double[p];
}
A=MM(Xt,X,p,n,p);
vector<double> c(p);
c=Mv(Xt,f,p,n);

vector<double> b(p);
b=LUSolve(A,c);
//Perform Newtons Method
vector<double> grad(2);
double **Hess;
Hess=new double* [2];
for (int i=0;i<2;i++)
{
        *(Hess+i)=new double[2];
}
double fmin=newfun2D(x0,b,grad,Hess);
vector<double> d(2);
vector<double> gneg(2);
gneg[0]=-grad[0];
gneg[1]=-grad[1];
double **Hes;
Hes=new double* [2];
for (int i=0;i<2;i++)
{
        *(Hes+i)=new double[2];
}
Hes[0][0]=Hess[0][0];
Hes[0][1]=Hess[0][1];
Hes[1][0]=Hess[1][0];
Hes[1][1]=Hess[1][1];
```

```
d=LUSolve(Hes,gneg);

//New code added to deal with indefinite Hessians.  Gaurentees descent.
double LS=inprod(gneg,d);
if (LS<0)
{
        vector<double> dprime(2);
        dprime[0]=sqrt(pow(gneg[0],2));
        dprime[1]=sqrt(pow(gneg[1],2));
        double alpha1prime=min(alpha1,min(rcon-x0[0],x0[0]-lcon));
        double alpha2prime=min(alpha2,min(tcon-x0[1],x0[1]-bcon));

        if ((alpha1prime==rcon-x0[0])&(sign(gneg[0])==-1))
        {
                alpha1prime=alpha1;
        }
        else if ((alpha1prime==x0[0]-lcon)&(sign(gneg[0])==1))
        {
                alpha1prime=alpha1;
        }
        if ((alpha2prime==tcon-x0[1])&(sign(gneg[1])==-1))
        {
                alpha2prime=alpha2;
        }
        else if ((alpha2prime==x0[1]-bcon)&(sign(gneg[1])==1))
        {
                alpha2prime=alpha2;
        }

        vector<double> dnew(2);
        double gamma1=atan(alpha2prime/alpha1prime);
        double gamma2=atan(dprime[1]/dprime[0]);
        if (gamma2>=gamma1)
        {
                dnew[0]=dprime[0]*alpha2prime/dprime[1];
                dnew[1]=alpha2prime;
                d[0]=sign(gneg[0])*dnew[0];
                d[1]=sign(gneg[1])*dnew[1];
        }
        else
        {
                dnew[0]=alpha1prime;
                dnew[1]=dprime[1]*alpha1prime/dprime[0];
                d[0]=sign(gneg[0])*dnew[0];
                d[1]=sign(gneg[1])*dnew[1];
        }
}
else
{
        vector<double> dprime(2);
        dprime[0]=sqrt(pow(d[0],2));
        dprime[1]=sqrt(pow(d[1],2));
        double alpha1prime=min(alpha1,min(rcon-x0[0],x0[0]-lcon));
        double alpha2prime=min(alpha2,min(tcon-x0[1],x0[1]-bcon));

        if ((alpha1prime==rcon-x0[0])&(sign(d[0])==-1))
        {
                alpha1prime=alpha1;
        }
        else if ((alpha1prime==x0[0]-lcon)&(sign(d[0])==1))
        {
                alpha1prime=alpha1;
        }
        if ((alpha2prime==tcon-x0[1])&(sign(d[1])==-1))
        {
                alpha2prime=alpha2;
        }
        else if ((alpha2prime==x0[1]-bcon)&(sign(d[1])==1))
        {
                alpha2prime=alpha2;
        }
```

```
        vector<double> dnew(2);
        if ((dprime[0]>alpha1prime)|(dprime[1]>alpha2prime))
        {
                double gamma1=atan(alpha2prime/alpha1prime);
                double gamma2=atan(dprime[1]/dprime[0]);
                if (gamma2>=gamma1)
                {
                        dnew[0]=dprime[0]*alpha2prime/dprime[1];
                        dnew[1]=alpha2prime;
                        d[0]=sign(d[0])*dnew[0];
                        d[1]=sign(d[1])*dnew[1];
                }
                else
                {
                        dnew[0]=alpha1prime;
                        dnew[1]=dprime[1]*alpha1prime/dprime[0];
                        d[0]=sign(d[0])*dnew[0];
                        d[1]=sign(d[1])*dnew[1];
                }
        }
        else
        {
                if (count1<nshrinks)
                {
                        if (count2<nshrinks)
                        {
                                alpha1=alpha1/2;
                                alpha2=alpha2/2;
                                count1=count1+1;
                                count2=count2+1;
                        }
                        else
                        {
                                alpha1=alpha1/2;
                                count1=count1+1;
                        }
                }
                else if (count2<nshrinks)
                {
                        alpha2/alpha2/2;
                        count2=count2+1;
                }
                else
                {
                        xprev[0]=x0[0];
                        xprev[1]=x0[1];
                }
        }
}
//Update point
x0[0]=x0[0]+d[0];
x0[1]=x0[1]+d[1];
fmin=newfun2D(x0,b,grad,Hess);
double fnew=fun2D(x0[0],x0[1]);
cout<<endl<<endl<<x0[0]<<"   "<<x0[1]<<"   "<<fnew<<endl<<endl;

//If new point is worse than previous stop
int ignore=0;
if (fnew>f[0])
{
        if (start==1)
        {
                x0[0]=points[0][0];
                x0[1]=points[0][1];
                fnew=f[0];
                flagt=0;
                flagb=0;
                flagl=0;
                flagr=0;
                xprev[0]=x0[0];
```

```
                        xprev[1]=x0[1];
                        diff2=0;
                        ignore=1;
                }
                else
                {
                        start=1;
                }
        }

        //Error Calc
        if (ignore==0)
        {
                vector<double> xm(p);
                xm[0]=1;
                xm[1]=x0[0];
                xm[2]=x0[1];
                xm[3]=pow(x0[0],2);
                xm[4]=pow(x0[1],2);
                xm[5]=x0[0]*x0[1];
                vector<double> yhat(n);
                yhat=Mv(X,b,n,p);
                vector<double> fsurf(n);
                for (int i=0;i<n;i++)
                {
                        fsurf[i]=pow((f[i]-yhat[i]),2)/(n-p);
                }
                double temp=sum(fsurf);
                double s=sqrt(temp);
                vector<double> error1(p);
                error1=LUSolve(A,xm);
                error=inprod(xm,error1);
                error=s*2.919986*sqrt(error);  //The number is for 90% confidence from
students t
        }

        //Update loop ending parameters
        diff1=sqrt(pow((fnew-fmin)/fnew,2));
        vector<double> xtemp(2);
        xtemp[0]=x0[0]-xprev[0];
        xtemp[1]=x0[1]-xprev[1];
        xprev[0]=x0[0];
        xprev[1]=x0[1];
        diff2=norm(xtemp);
        cout<<diff1<<"   "<<diff2<<endl;

        //Check if GRGM needs to be used
        if (diff2<0.0001)
        {
                if ((flagl==1)|(flagr==1))
                {
                        fnew=GRGM(x0,fnew,alpha2,bcon,tcon,1);
                        if (count2<nshrinks)
                        {
                                count2=count2+1;
                                alpha2=alpha2/2;
                                if (noshrinkh==0)
                                {
                                        alpha1=alpha1*2;
                                        noshrinkh=1;
                                }
                        }
                        else
                        {
                                xprev[0]=x0[0];
                                xprev[1]=x0[1];
                        }
                }
                else if ((flagb==1)|(flagt==1))
                {
                        fnew=GRGM(x0,fnew,alpha1,lcon,rcon,0);
```

```
                if (count1<nshrinks)
                {
                        count1=count1+1;
                        alpha1=alpha1/2;
                        if (noshrinkv==0)
                        {
                                alpha2=alpha2*2;
                                noshrinkv=1;
                        }
                }
                else
                {
                        xprev[0]=x0[0];
                        xprev[1]=x0[1];
                }
        }
        xtemp[0]=x0[0]-xprev[0];
        xtemp[1]=x0[1]-xprev[1];
        xprev[0]=x0[0];
        xprev[1]=x0[1];
        diff2=norm(xtemp);
        cout<<diff1<<"   "<<diff2<<endl;
        //system("PAUSE");
}

f[0]=fnew;
//Update points
points[0][0]=x0[0];
points[0][1]=x0[1];

if (x0[0]<(lcon+alpha1))
{
        if (x0[0]<=lcon)
        {
                if (count1<nshrinks)
                {
                        alpha1=alpha1/2;//Shrink if on left edge.
                        count1=count1+1;
                        flagl=1;
                }
                else
                {
                        flagl=0;
                }
                points[0][0]=lcon;
                points[1][0]=lcon;
                points[2][0]=lcon;
                points[3][0]=points[0][0]+alpha1;
                points[4][0]=points[0][0]+alpha1;
                points[5][0]=points[0][0]+alpha1;
                points[6][0]=points[3][0]+alpha1;
                points[7][0]=points[3][0]+alpha1;
                points[8][0]=points[3][0]+alpha1;
                flagr=0;
                flagb=0;
                flagt=0;
        }
        else
        {
                points[1][0]=points[0][0];
                points[2][0]=points[0][0];
                points[3][0]=lcon;
                points[4][0]=lcon;
                points[5][0]=lcon;
                points[6][0]=points[0][0]+alpha1;
                points[7][0]=points[0][0]+alpha1;
                points[8][0]=points[0][0]+alpha1;
                flagl=0;
                flagr=0;
                flagb=0;
                flagt=0;
```

```
                noshrinkh=0;
        }
}
else if (x0[0]>(rcon-alpha1))
{
        if (x0[0]>=rcon)
        {
                if (count1<nshrinks)
                {
                        alpha1=alpha1/2;//Shrink if on right edge.
                        count1=count1+1;
                        flagr=1;
                }
                else
                {
                        flagr=0;
                }
                points[0][0]=rcon;
                points[1][0]=rcon;
                points[2][0]=rcon;
                points[3][0]=points[0][0]-alpha1;
                points[4][0]=points[0][0]-alpha1;
                points[5][0]=points[0][0]-alpha1;
                points[6][0]=points[3][0]-alpha1;
                points[7][0]=points[3][0]-alpha1;
                points[8][0]=points[3][0]-alpha1;
                flagl=0;
                flagb=0;
                flagt=0;
        }
        else
        {
                points[1][0]=points[0][0];
                points[2][0]=points[0][0];
                points[3][0]=rcon;
                points[4][0]=rcon;
                points[5][0]=rcon;
                points[6][0]=points[0][0]-alpha1;
                points[7][0]=points[0][0]-alpha1;
                points[8][0]=points[0][0]-alpha1;
                flagl=0;
                flagr=0;
                flagb=0;
                flagt=0;
                noshrinkh=0;
        }
}
else
{
        points[1][0]=points[0][0];
        points[2][0]=points[0][0];
        points[3][0]=points[0][0]+alpha1;
        points[4][0]=points[0][0]+alpha1;
        points[5][0]=points[0][0]+alpha1;
        points[6][0]=points[0][0]-alpha1;
        points[7][0]=points[0][0]-alpha1;
        points[8][0]=points[0][0]-alpha1;
        flagl=0;
        flagr=0;
        flagb=0;
        flagt=0;
        noshrinkh=0;
}

if (x0[1]<(bcon+alpha2))
{
        if (x0[1]<=bcon)
        {
                if (count2<nshrinks)
                {
                        alpha2=alpha2/2;//Shrink if on bottom edge.
```

```
                        count2=count2+1;
                        flagb=1;
                }
                else
                {
                        flagb=0;
                }
                points[0][1]=bcon;
                points[1][1]=points[0][1]+alpha2;
                points[2][1]=points[1][1]+alpha2;
                points[3][1]=bcon;
                points[4][1]=points[0][1]+alpha2;
                points[5][1]=points[1][1]+alpha2;
                points[6][1]=bcon;
                points[7][1]=points[0][1]+alpha2;
                points[8][1]=points[1][1]+alpha2;
                flagl=0;
                flagr=0;
                flagt=0;
        }
        else
        {
                points[1][1]=points[0][1]+alpha2;
                points[2][1]=bcon;
                points[3][1]=points[0][1];
                points[4][1]=points[0][1]+alpha2;
                points[5][1]=bcon;
                points[6][1]=points[0][1];
                points[7][1]=points[0][1]+alpha2;
                points[8][1]=bcon;
                flagl=0;
                flagr=0;
                flagb=0;
                flagt=0;
                noshrinkv=0;
        }
}
else if (x0[1]>(tcon-alpha2))
{
        if (x0[1]>=tcon)
        {
                if (count2<nshrinks)
                {
                        alpha2=alpha2/2;//Shrink if on top edge.
                        count2=count2+1;
                        flagt=1;
                }
                else
                {
                        flagt=0;
                }
                points[0][1]=tcon;
                points[1][1]=points[0][1]-alpha2;
                points[2][1]=points[1][1]-alpha2;
                points[3][1]=tcon;
                points[4][1]=points[0][1]-alpha2;
                points[5][1]=points[1][1]-alpha2;
                points[6][1]=tcon;
                points[7][1]=points[0][1]-alpha2;
                points[8][1]=points[1][1]-alpha2;
                flagl=0;
                flagr=0;
                flagb=0;
        }
        else
        {
                points[1][1]=points[0][1]-alpha2;
                points[2][1]=tcon;
                points[3][1]=points[0][1];
                points[4][1]=points[0][1]-alpha2;
                points[5][1]=tcon;
```

```
                                points[6][1]=points[0][1];
                                points[7][1]=points[0][1]-alpha2;
                                points[8][1]=tcon;
                                flagl=0;
                                flagr=0;
                                flagb=0;
                                flagt=0;
                                noshrinkv=0;
                        }
                }
                else
                {
                        points[1][1]=points[0][1]+alpha2;
                        points[2][1]=points[0][1]-alpha2;
                        points[3][1]=points[0][1];
                        points[4][1]=points[0][1]+alpha2;
                        points[5][1]=points[0][1]-alpha2;
                        points[6][1]=points[0][1];
                        points[7][1]=points[0][1]+alpha2;
                        points[8][1]=points[0][1]-alpha2;
                        flagl=0;
                        flagr=0;
                        flagb=0;
                        flagt=0;
                        noshrinkv=0;
                }
        }
        cout<<"x*=("<<x0[0]<<","<<x0[1]<<") with fmax="<<f[0]<<"+/-"<<error<<endl;
        return(0);
}

double GRGM(vector<double> &x0, double f0, double alpha, double lcon, double rcon, int dim)
{
        //Selecting starting points
        alpha=alpha/2;
        int stop=0;
        double minloc;
        double maxloc;
        int adim;
        if (dim==1)
        {
                adim=0;
        }
        else
        {
                adim=1;
        }
        vector<double> points(5);
        points[0]=x0[dim];

        //checks which piece of the x vector we are altering and then chooses points
        if (x0[dim]<(lcon+2*alpha))
        {
                if (x0[dim]<=(lcon+alpha))
                {
                        if (x0[dim]<=lcon)
                        {
                                stop=1;
                        }
                        else
                        {
                                points[1]=lcon;
                                points[2]=points[0]+alpha;
                                points[3]=points[2]+alpha;
                                points[4]=points[3]+alpha;
                                minloc=1;
                                maxloc=4;
                        }
                }
                else
                {
```

```
                        points[1]=lcon;
                        points[2]=points[0]-alpha;
                        points[3]=points[0]+alpha;
                        points[4]=points[3]+alpha;
                        minloc=1;
                        maxloc=4;
                }
        }
        else if (x0[dim]>(rcon-2*alpha))
        {
                if (x0[dim]>=(rcon-alpha))
                {
                        if (x0[dim]>=rcon)
                        {
                                stop=1;
                        }
                        else
                        {
                                points[1]=rcon;
                                points[2]=points[0]-alpha;
                                points[3]=points[2]-alpha;
                                points[4]=points[3]-alpha;
                                minloc=4;
                                maxloc=1;
                        }
                }
                else
                {
                        points[1]=rcon;
                        points[2]=points[0]+alpha;
                        points[3]=points[0]-alpha;
                        points[4]=points[3]-alpha;
                        minloc=4;
                        maxloc=1;
                }
        }
        else
        {
                points[1]=points[0]-alpha;
                points[2]=points[1]-alpha;
                points[3]=points[0]+alpha;
                points[4]=points[3]+alpha;
                minloc=2;
                maxloc=4;
        }

        //Initialize quantities
        double length=points.size();
        double fnew;
        vector<double> f(length);
        f[0]=f0;
        //Loop until the model and function have same value or the model points solution is the
same
        while (stop==0)
        {
                for (int i=1;i<length;i++)
                {
                        if (adim==0)
                        {
                                f[i]=fun2D(x0[adim],points[i]);
                        }
                        else
                        {
                                f[i]=fun2D(points[i],x0[adim]);
                        }
                }
                //Perform Least Squares fit
                double n=length;
                int k=1; //This is equal to the number of variables;
                double p=2*k+1;  //Number of regressor variable.  Need to change for higher order.
```

```
double **X;
X=new double* [n];
for (int i=0;i<n;i++)
{
        *(X+i)=new double[p];
}

for (int i=0;i<n;i++)
{
        //Need to add here for higher dimensions
        for (int c=0;c<k;c++)
        {
                X[i][c]=1;
                X[i][c+1]=points[i];
                X[i][c+k+1]=pow(points[i],2);
        }
}
double **Xt;
Xt=new double* [p];
for (int i=0;i<p;i++)
{
        *(Xt+i)=new double[n];
}
Xt=MT(X,n,p);

double **A;
A=new double* [p];
for (int i=0;i<p;i++)
{
        *(A+i)=new double[p];
}
A=MM(Xt,X,p,n,p);
vector<double> c(p);
c=Mv(Xt,f,p,n);

vector<double> b(p);
b=LUSolve(A,c);
//Perform Newtons Method
double g;
double H;
double fmin=newfun(x0,b,g,H);
if (H<0)
{
        double g1;
        double H1;
        vector<double> p(1);
        p[0]=points[minloc];
        double b1=newfun(p,b,g1,H1);
        double g2;
        double H2;
        p[0]=points[maxloc];
        double b2=newfun(p,b,g2,H2);
        if (b1<=b2)
        {
                fmin=b1;
                x0[dim]=points[minloc];
        }
        else
        {
                fmin=b2;
                x0[dim]=points[maxloc];
        }
}
else
{
        double d=-g/H;
        double temp;  //Added to stop solver from going beyond model range.
        temp=x0[dim]+d;
        if (temp<points[minloc])
        {
                x0[dim]=points[minloc];
```

```
        }
        else if (temp>points[maxloc])
        {
                x0[dim]=points[maxloc];
        }
        else
        {
                x0[dim]=x0[dim]+d;
                stop=1;
        }
}
if (adim==0)
{
        fnew=fun2D(x0[adim],x0[dim]);
}
else
{
        fnew=fun2D(x0[dim],x0[adim]);
}
cout<<endl<<endl<<x0[dim]<<"   "<<fnew<<endl<<endl;
//system("PAUSE");
if (fnew>f[0])
{
        x0[dim]=points[0];
        fnew=f[0];
        stop=1;
}

f[0]=fnew;
//Update points
points[0]=x0[dim];
//Added consraints back in and generalized April 14, 2010
if (x0[dim]<(lcon+2*alpha))
{
        if (x0[dim]<=(lcon+alpha))
        {
                if (x0[dim]<=lcon)
                {
                        stop=1;
                }
                else
                {
                        points[1]=lcon;
                        points[2]=points[0]+alpha;
                        points[3]=points[2]+alpha;
                        points[4]=points[3]+alpha;
                        minloc=1;
                        maxloc=4;
                }
        }
        else
        {
                points[1]=lcon;
                points[2]=points[0]-alpha;
                points[3]=points[0]+alpha;
                points[4]=points[3]+alpha;
                minloc=1;
                maxloc=4;
        }
}
else if (x0[dim]>(rcon-2*alpha))
{
        if (x0[dim]>=(rcon-alpha))
        {
                if (x0[dim]>=rcon)
                {
                        stop=1;
                }
                else
                {
                        points[1]=rcon;
```

```
                                        points[2]=points[0]-alpha;
                                        points[3]=points[2]-alpha;
                                        points[4]=points[3]-alpha;
                                        minloc=4;
                                        maxloc=1;
                                }
                        }
                        else
                        {
                                points[1]=rcon;
                                points[2]=points[0]+alpha;
                                points[3]=points[0]-alpha;
                                points[4]=points[3]-alpha;
                                minloc=4;
                                maxloc=1;
                        }
                }
                else
                {
                        points[1]=points[0]-alpha;
                        points[2]=points[1]-alpha;
                        points[3]=points[0]+alpha;
                        points[4]=points[3]+alpha;
                        minloc=2;
                        maxloc=4;
                }
        }
        return(fnew);
}
double norm(vector<double> x)
//finds norms of vectors
{
        double length=x.size();
        double sum=0;
        for (int i=0;i<length;i++)
        {
                sum+=x[i]*x[i];
        }
        double norm=sqrt(sum);
        return(norm);
}

double inprod(vector<double>x, vector<double>y)
//inner product of vectors
{
        double length=x.size();
        double ans=0;
        for(int i=0;i<length;i++)
        {
                ans+=x[i]*y[i];
        }
        return(ans);
}

vector<double> Mv(double **M,vector<double> v,double row, double col)
//matrix times a vector
{
        vector<double> ans(row,0);
        for(int i=0;i<row;i++)
        {
                for(int j=0;j<col;j++)
                {
                        ans[i]=ans[i]+M[i][j]*v[j];
                }
        }
        return(ans);
}

double **MM(double **M1, double **M2,double a, double b, double c)
//matrix times matrix
{
```

```
        double **ans;
        ans=new double* [a];
        for(int i=0;i<a;i++)
        {
                *(ans+i)=new double[c];
        }
        for(int i=0;i<a;i++)
        {
                for(int j=0;j<c;j++)
                {
                        ans[i][j]=0;
                }
        }
        for(int k=0;k<a;k++)
        {
                for(int i=0;i<c;i++)
                {
                        for(int j=0;j<b;j++)
                        {
                                ans[k][i]=ans[k][i]+M1[k][j]*M2[j][i];
                        }
                }
        }
        return(ans);
}

vector<double> LUSolve(double **M,vector<double> v)
//LU decomposion solver
{
        int length=v.size();
        for(int i=0;i<length-1;i++)
        {
                for(int j=i+1;j<length;j++)
                {
                        double m=M[j][i]/M[i][i];
                        M[j][i]=0;
                        for(int k=i+1;k<length;k++)
                        {
                                M[j][k]=M[j][k]-m*M[i][k];
                        }
                        M[j][i]=m;
                }
        }
        for(int i=1;i<length;i++)
        {
                for (int j=0;j<i;j++)
                {
                        v[i]=v[i]-M[i][j]*v[j];
                }
        }
        v[length-1]=v[length-1]/M[length-1][length-1];
        for (int i=0;i<length-1;i++)
        {
                for (int j=length-1-i;j<length;j++)
                {
                        v[length-2-i]=v[length-2-i]-M[length-2-i][j]*v[j];
                }
                v[length-2-i]=v[length-2-i]/M[length-2-i][length-2-i];
        }
        return(v);
}
double sum(vector<double> x)
//add vector components
{
        double add=0;
        double length=x.size();
        for (int i=0;i<length;i++)
        {
                add+=x[i];
        }
        return (add);
```

```
}
double newfun(vector<double> x, vector<double> b, double &g, double &H)
//1-D model function
{
        double val=b[0]+b[1]*x[0]+b[2]*pow(x[0],2);
        g=b[1]+2*b[2]*x[0];
        H=2*b[2];
        return(val);
}
double newfun2D(vector<double> x, vector<double> b, vector<double> &grad, double **&Hess)
//2-D model function
{
        double val=b[0]+b[1]*x[0]+b[2]*x[1]+b[3]*pow(x[0],2)+b[4]*pow(x[1],2)+b[5]*x[0]*x[1];
        grad[0]=b[1]+2*b[3]*x[0]+b[5]*x[1];
        grad[1]=b[2]+2*b[4]*x[1]+b[5]*x[0];
        Hess[0][0]=2*b[3];
        Hess[0][1]=b[5];
        Hess[1][0]=b[5];
        Hess[1][1]=2*b[4];
        return(val);
}
double fun2(double x)
//combustion function evaluation for 1-D case
{
        vector<double> vars(1);
        vars[0]=x/1000;          //Divide by 1000 for pore diameter
        double eff=-Combust(vars); //Added negative for maximization.
        return(eff);
}
double fun2D(double x, double y)
//combustion function evaluation for 2-D case
{
        //double eff=100*pow((y-pow(x,2)),2)+pow((1-x),2);  //Rosenbrock
        vector<double> vars(2);
        vars[0]=x/1000;
        vars[1]=y;
        double eff=-Combust(vars);
        return(eff);
}
double **MT(double **M,double n, double p)
//matrix times its transpose
{
        double **ans;
        ans=new double* [p];
        for(int i=0;i<p;i++)
        {
                *(ans+i)=new double[n];
        }
        for (int i=0;i<p;i++)
        {
                for (int j=0;j<n;j++)
                {
                        ans[i][j]=M[j][i];
                }
        }
        return(ans);
}
double max(double x, double y)
//max of two numbers
{
        double temp1=sqrt(pow(x,2));
        double temp2=sqrt(pow(y,2));
        double ans;
        if (temp1>=temp2)
        {
                ans=temp1;
        }
        else
        {
                ans=temp2;
        }
```

```
        return(ans);
}
double min(double x, double y)
//min of two numbers
{
        double temp1=sqrt(pow(x,2));
        double temp2=sqrt(pow(y,2));
        double ans;
        if (temp1<=temp2)
        {
                ans=temp1;
        }
        else
        {
                ans=temp2;
        }
        return(ans);
}
int sign(double x)
//sign of a number
{
        int ans;
        if (x>=0)
        {
                ans=1;
        }
        else
        {
                ans=-1;
        }
        return(ans);
}
```

Appendix E: Two Dimensional Response Surfaces

This section contains all of the response surfaces for the two dimensional case study. They are presented here to allow for a better understanding of the solution path of the algorithm.

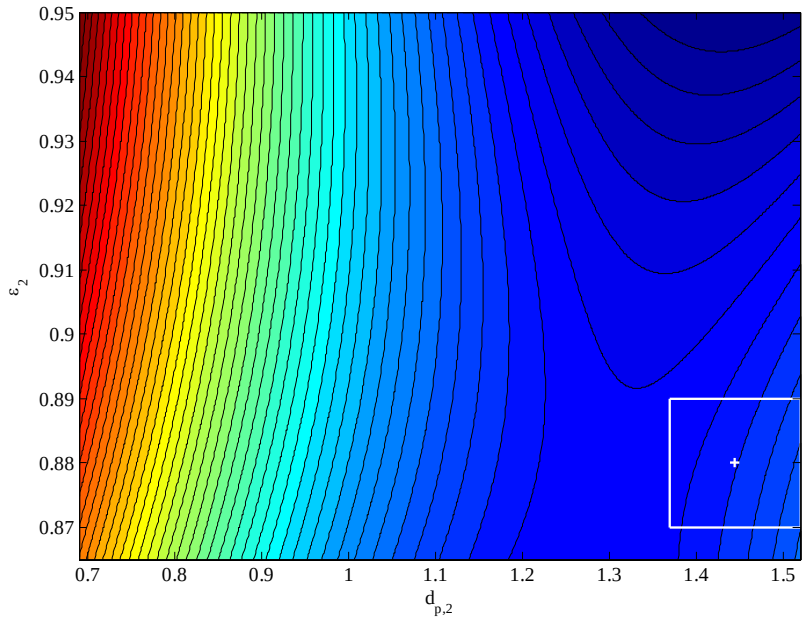


Figure E.1 – First Response Surface for the 2-D Case

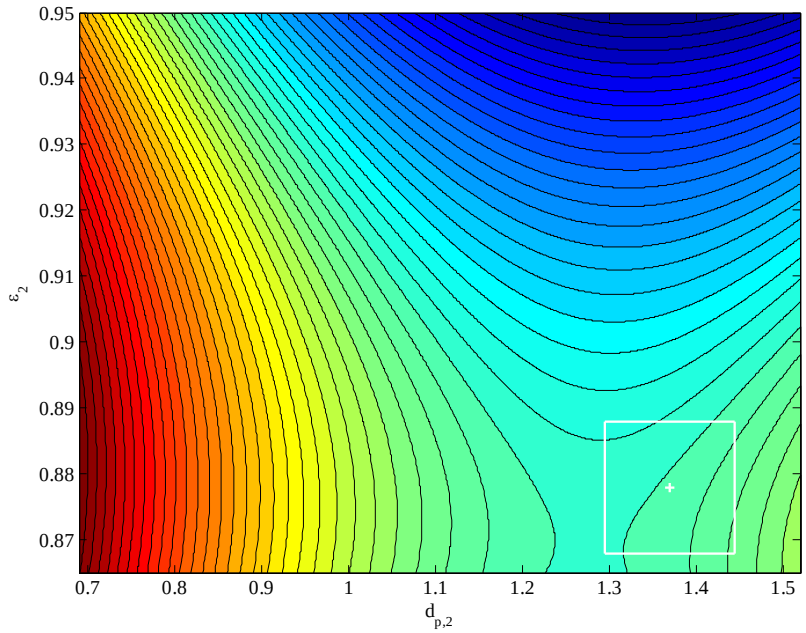


Figure E.2 – Second Response Surface for the 2-D Case

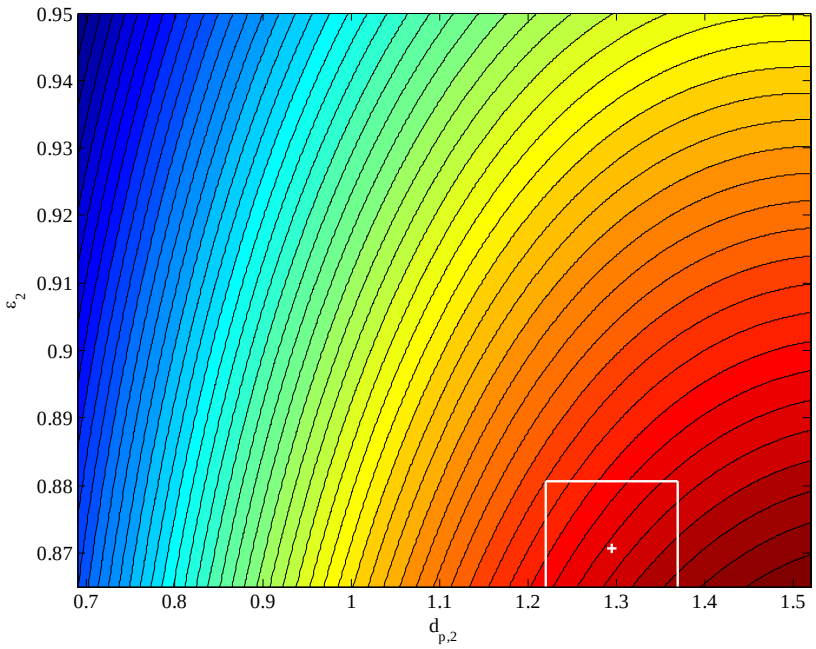


Figure E.3 – Third Response Surface for the 2-D Case

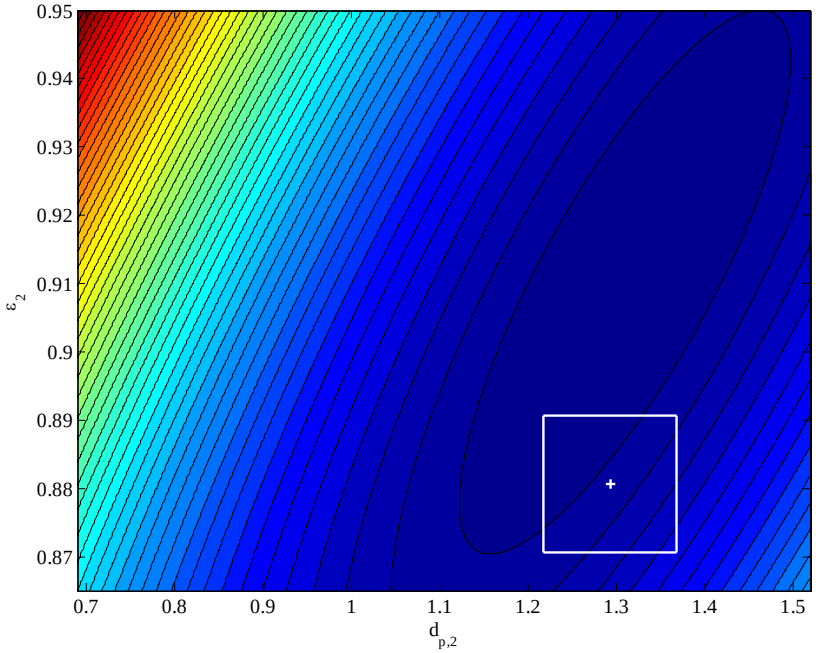


Figure E.4 – Fourth Response Surface for the 2-D Case

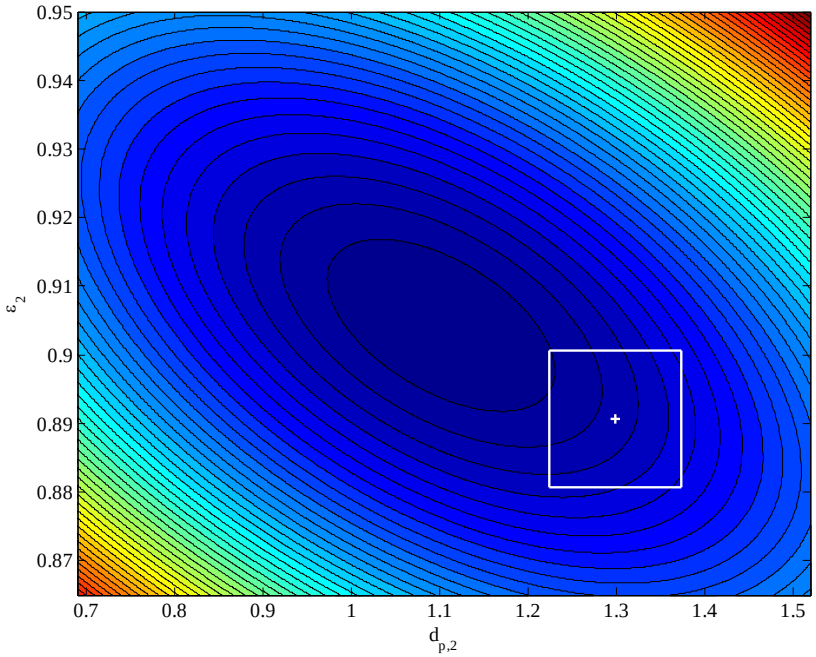


Figure E.5 – Fifth Response Surface for the 2-D Case

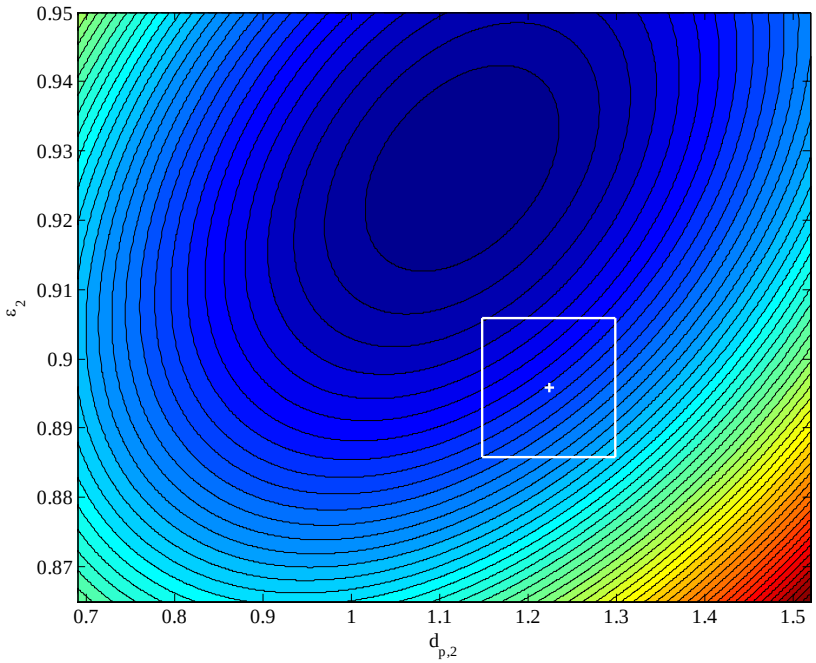


Figure E.6 – Sixth Response Surface for the 2-D Case

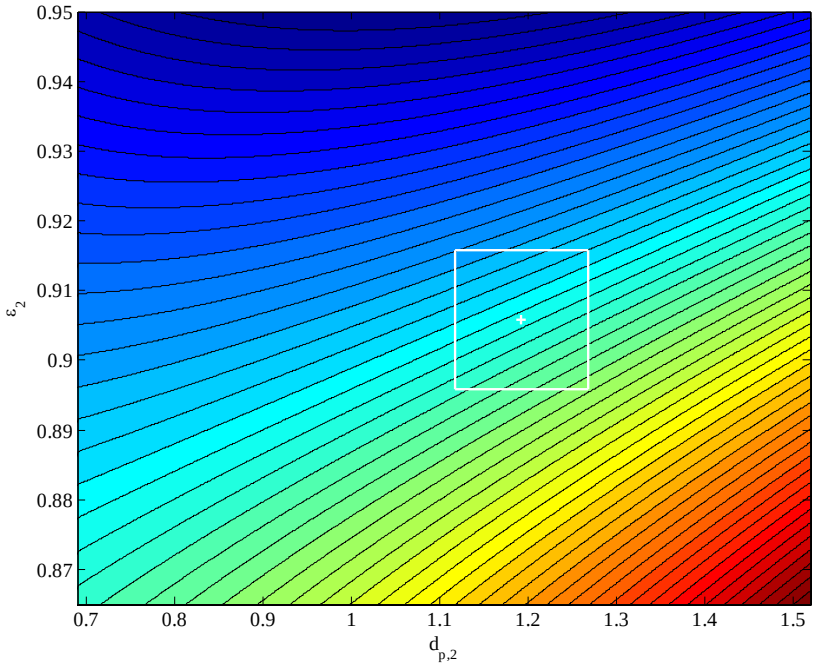


Figure E.7 – Seventh Response Surface for the 2-D Case

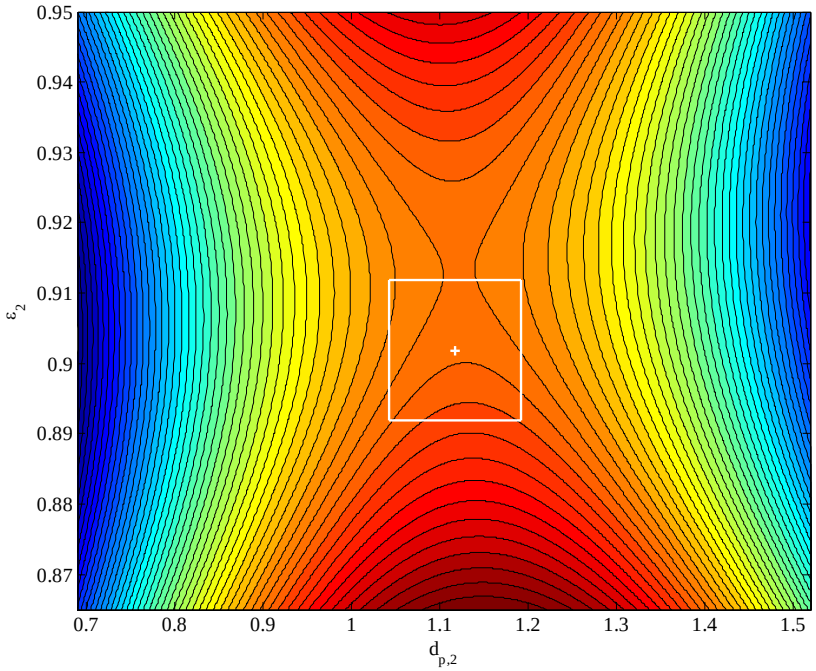


Figure E.8 – Eighth Response Surface for the 2-D Case

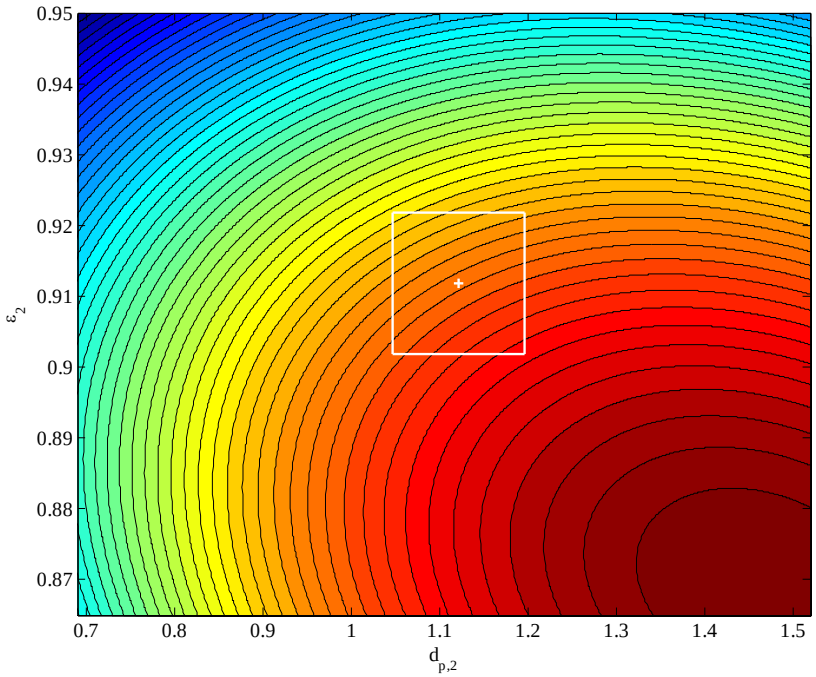


Figure E.9 – Ninth Response Surface for the 2-D Case

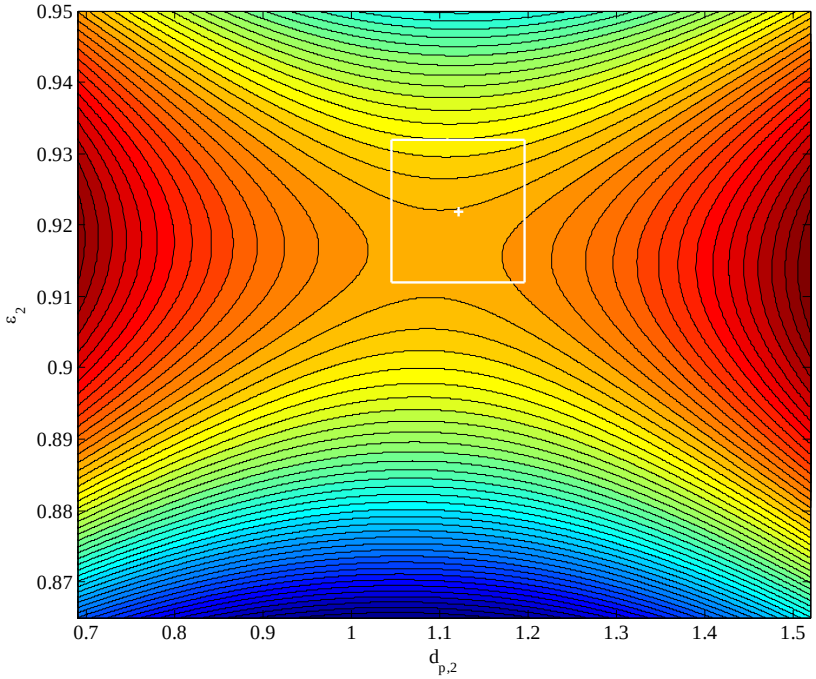


Figure E.10 – Tenth Response Surface for the 2-D Case

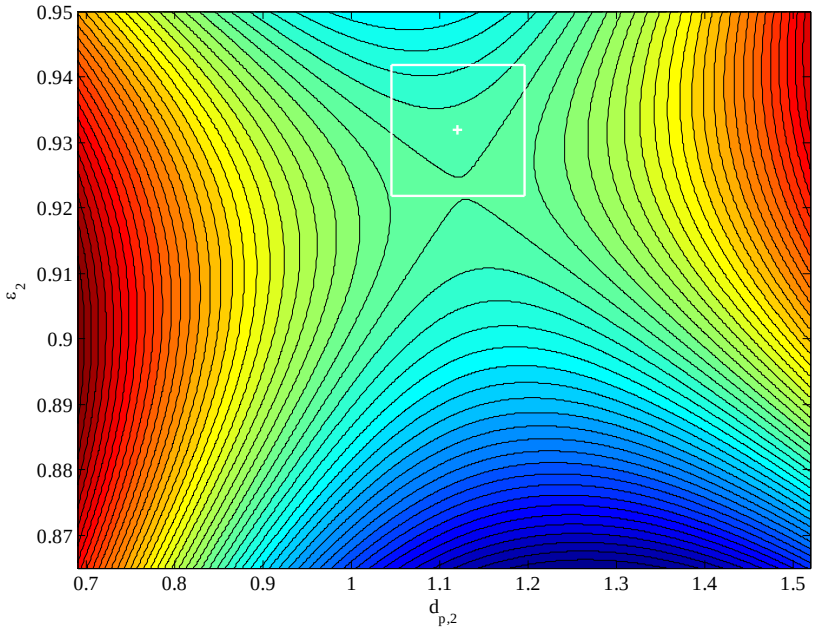


Figure E.11 – Eleventh Response Surface for the 2-D Case

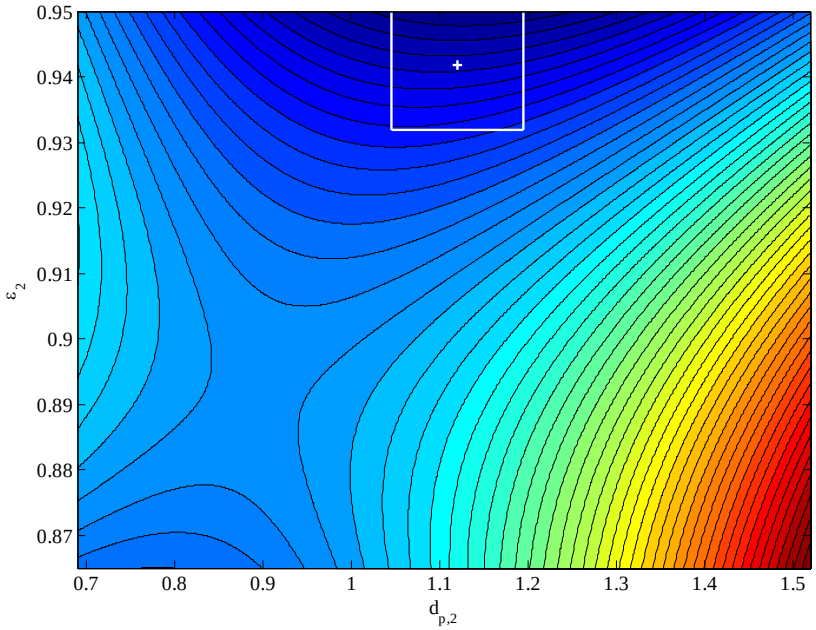


Figure E.12 – Twelfth Response Surface for the 2-D Case

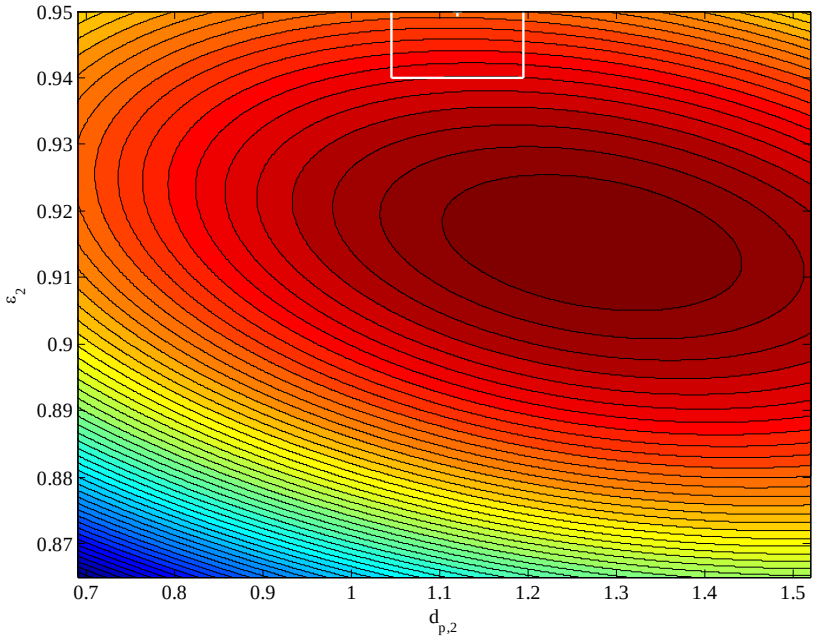


Figure E.13 – Thirteenth Response Surface for the 2-D Case

At this point the algorithm has stopped in the same location twice on a boundary, so the GRG method is used until a maximum is reached.

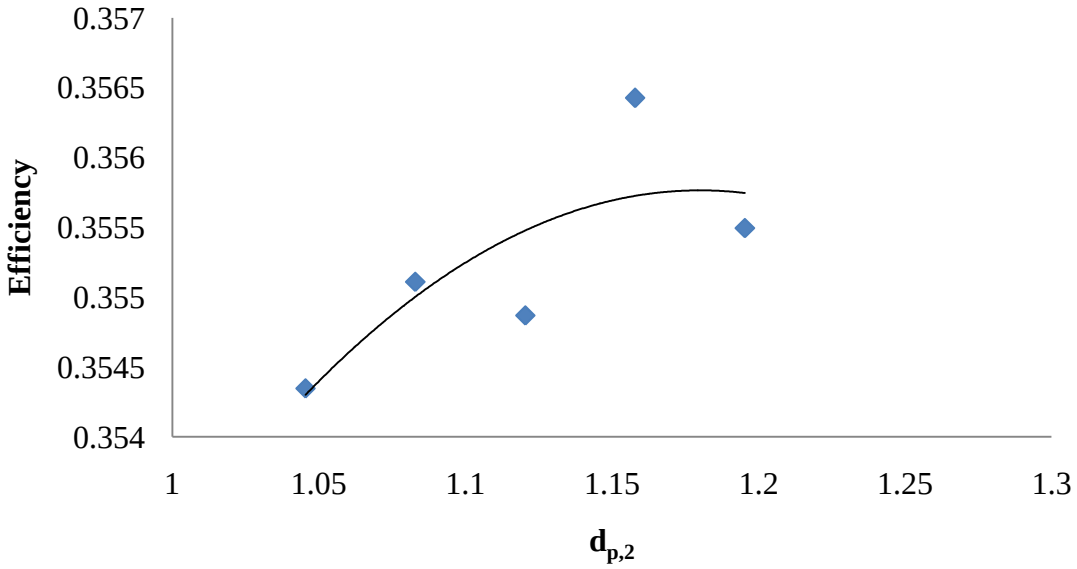


Figure E.14 – First GRG Method Surface for the 2-D Case, ε_2=0.95

The first surface generated contains a maximum so only one iteration of the GRG method is needed. The solver then reverts back to a 2-D method and continues to try and optimize the burner further. Once again, however, the algorithm has lands in the same spot on the boundary so the GRG method is needed, resulting in the following response surfaces.

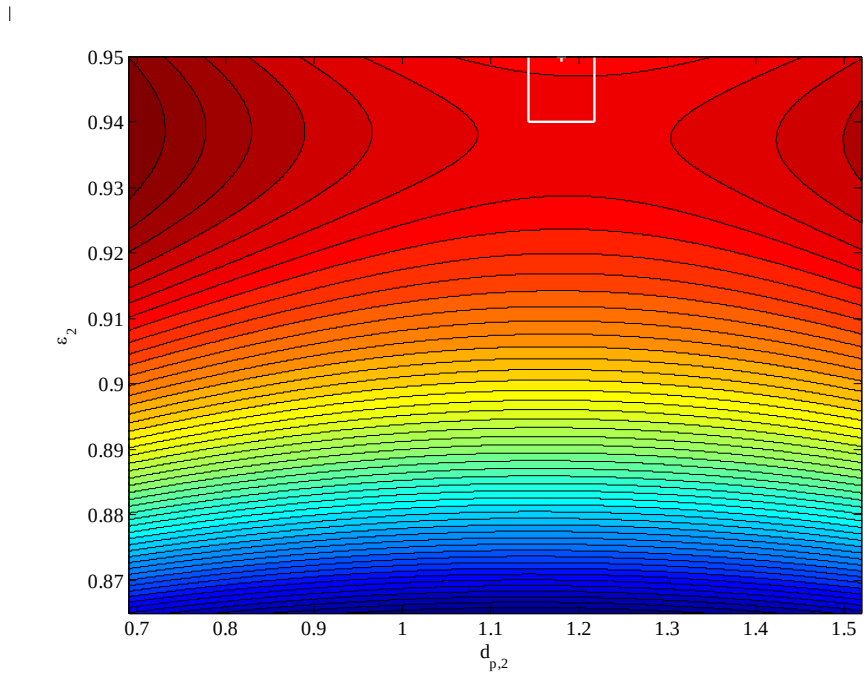


Figure E.15 – Fourteenth Response Surface for the 2-D Case

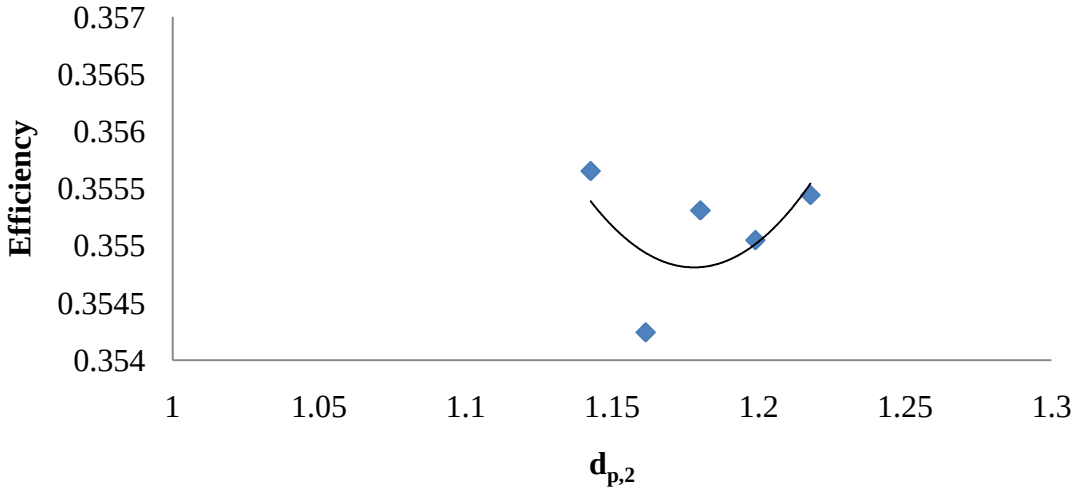


Figure E.16 – Second GRG Method Surface for the 2-D Case, ε_2=0.95

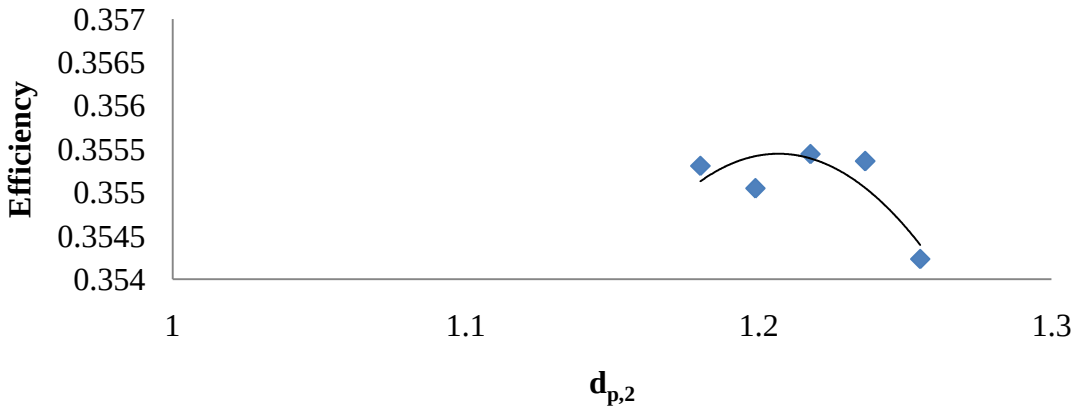


Figure E.17 – Third GRG Method Surface for the 2-D Case, ε_2=0.95

The third GRG method surface contains a maximum value so the algorithm reverts back to a two dimensional problem. Here, however, the maximum of the response surface results in the objective function worsening, as a result of noise being introduced due to the small size of the surface. Therefore the solver stops and reports the maximum of the previous surface, the third GRG method surface, as the optimal value.

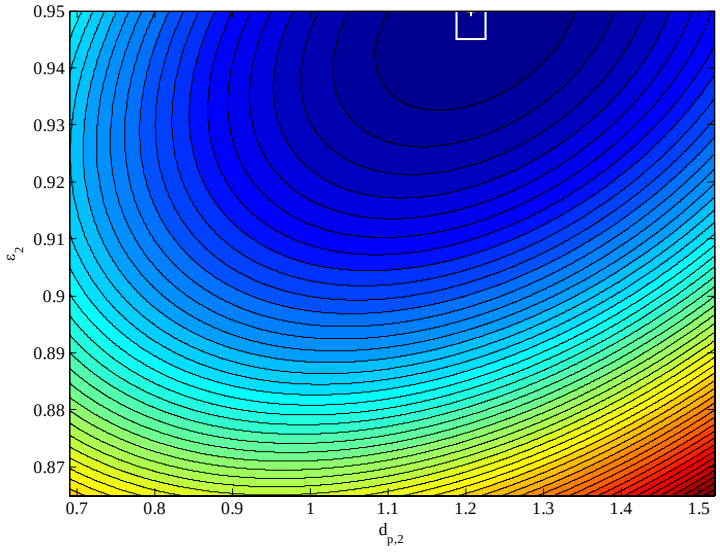


Figure E.18 – Fifteenth Response Surface for the 2-D Case