AN ABSTRACT OF THE DISSERTATION OF

Christopher Matthews for the degree of Doctor of Philosophy in Nuclear Engineering presented on June 5, 2015.

Title: Fission Gas Bubble Behavior in Uranium Carbide

Abstract approved: _____

Andrew C. Klein

The need for cheap reliable energy, while simultaneously avoiding uranium supply constraints makes uranium carbide (UC) fueled Gas Fast Reactors offer an attractive nuclear reactor design. In order to qualify the fuel, an enhanced understanding of the behavior of uranium carbide during operation is paramount. Due to a reduced re-solution rate, uranium carbide suffers from a buildup of very large fission gas bubbles. While these bubbles serve to reduce total fission gas release through the trapping of diffusing gas atoms, they lead to high swelling and ultimately dominate the microstructure of the fuel.

The bubble size distribution is determined by the competing absorption rate and the rate of knock-out, or re-solution. As a result of the enhanced thermal dissipative properties of uranium carbide fuel, the atom-by-atom knockout process was shown to be an accurate representation of re-solution in uranium carbide. Furthermore, the Binary Collision Approximation was shown to appropriately model the re-solution event, bypassing computationally expensive Molecular Dynamics simulations. The code 3DOT was developed as an off-shoot of the code 3DTrim, both of which utilize the TRIM algorithm to calculate the kinematics of ions traveling through a material.

Benefiting from modern methods and enhanced computational power, the model created in 3DOT results in a more fundamental understanding of the re-solution process in uranium carbide. A re-solution parameter that was an order of magnitude lower than previously determined was calculated in 3DOT. A decrease in the re-solution parameter as a function of radius occurred for low bubble radii, with a nearly constant re-solution parameter for bubble radii above 50 nm. Through comparative studies on the re-solution parameter for various values of implantation energy and atomic density in the bubble, we found that while the re-solution parameter did change slightly, the overall shape did not.

A new application, BUCK, was built using the MOOSE framework to simulate the fission gas bubble concentration distribution. In order to build a bare-bones foundation, the simplistic yet historically prevalent physics that can be used to model fission gas bubble nucleation, growth, and knock-out were implemented as stepping stones until more advanced models for each physical

process can be created. As the first step towards models that are based on first-principles, the new re-solution parameter was included and tested within BUCK.

BUCK was tested using different parameters and behaved normally. However, from studies using representative simulation parameters, it is clear that the currently implemented theory does not adequately identify the growth mechanism that leads to larger bubbles. While this currently limits the applicability of BUCK in a full fuel pin calculation, it provides the baseline structure in which new physics can be implemented, and represents an important step towards understanding the complex behavior of fission gas bubbles.

©Copyright by Christopher Matthews
June 5, 2015
All Rights Reserved

Fission Gas Bubble Behavior in Uranium Carbide

by

Christopher Matthews

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 5, 2015
Commencement June 2015

Doctor of Philosophy dissertation of Christopher Matthews presented on June 5, 2015.

APPROVED:

Major Professor, representing Nuclear Engineering

Head of the Department of Nuclear Engineering and Radiation Health Physics

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Christopher Matthews, Author

ACKNOWLEDGEMENTS

Although the work described here only chronicles the last several years of research, it stands on the shoulders of all of my previous life experiences, support, and interactions.

Working from the beginning, I owe my parents for who I am today, both my Father for setting a shining example of right and wrong and an appreciation for the technical, and my Mother for showing me how to have fun at the same time.

I am grateful to the advisors I have had over the years and the opportunities they have given me, especially Dr. Klein who worked with me on the current project, and Dr. Palmer who first brought me to Oregon State University.

I am appreciative of The Fringe research group for sitting through many practice talks over the years through the up's and down's, and being as interested in bubbles as I soon became.

And finally, I am extremely grateful to have my Pack at home. Rhody has been an non-judging and unexpected companion with whom I shared many thoughtful walks. And of course, I owe my wife Denise for listening to my worries, dealing with my insecurities, and sharing in my triumphs.

Not all of us are as lucky as I am. Thank you all.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF FIGURES (Continued)

Figure Page

LIST OF TABLES

LIST OF APPENDICES

LIST OF APPENDIX FIGURES

Figure Page

LIST OF APPENDIX TABLES

CHAPTER 1: INTRODUCTION

The increase in electricity consumption is inevitably intertwined with worldwide development. At the same time, the need for clean, cheap energy is driving innovation across all energy production means. Renewed interest in advanced nuclear technology will necessitate new methods in order to perform modern safety analyses. The primary goal in nuclear safety lies in retaining the fission products produced during fission, either within the fuel or a secondary containment system. The ability to predict fuel behavior and how fuel retains these fission products is essential, and is primarily achieved through empirically-based methods, limiting the applicability of calculations to the experimental parameters that formulated the model. As a result, current simplistic fission gas models are being pushed to their limit of applicability. Many of these models are based on uranium oxide fuel experiments, resulting in poor transferability to non-oxide fuel types. Due to tremendous advances in computing power and techniques, it is now possible to build models from the ground up, relying on first-principles and fundamental understanding of phenomena, rather than integral experiments. The following work aims to advance the modeling capabilities of uranium carbide fuel by creating a physical model to predict fission gas behavior in uranium carbide, an advanced nuclear fuel with application in new Gas Fast Reactors.

The intertwining of modern society and energy has grown ever since the first man-made fires warmed our ancestors. Now it is nearly impossible to escape electrical outlets or batteries in developed countries, while the rest of the undeveloped world is racing to match first-world consumption, and with good reason. By studying factors that determine our modern quality of life, the link to energy consumption is clear: an energy consumption of 1 kW per person* ensures an adequate quality of life, such as access to drinking water, high life expectancy, and low infant mortality [1].

Energy forecasts from the US Energy Information Administration predict that worldwide electricity production will nearly double in the next four decades, from 20 trillion kWh in 2010 to 39.0 in 2050 [2]. The UN divides energy predictions into two categories: members of the OECD or Organization for Economic Cooperation and Development (North America, most of Europe, Australia, Japan and South Korea) and non-OECD members (Russia, China, India, Africa, Central and South America, and the Middle East). The increase in electrical generation is primarily in these latter countries, with China representing the overwhelming majority, more than doubling their production in the next 30 years (Figure 1.1).

Rising energy production comes with its own disadvantages; as production increases, depletion of limited fuels such as coal, oil and natural gas accelerates. Furthermore, inescapable CO_2 emission is projected to have global ramifications through climate change [3]. Due to the relatively

*Includes petroleum, natural gas, coal, hydroelectricity, and renewable energy. Energy from direct sources such as food, biomass, and solar heating are not considered.

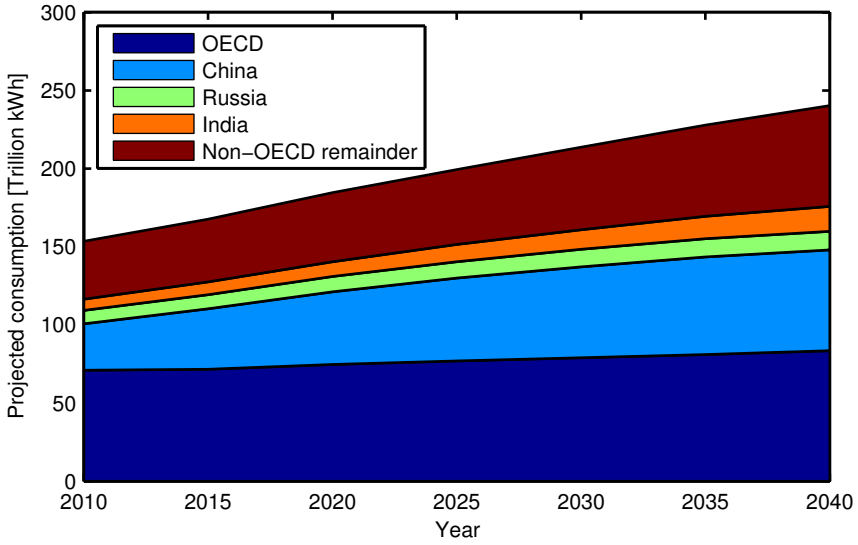

Figure 1.1: Worldwide energy consumption [2].

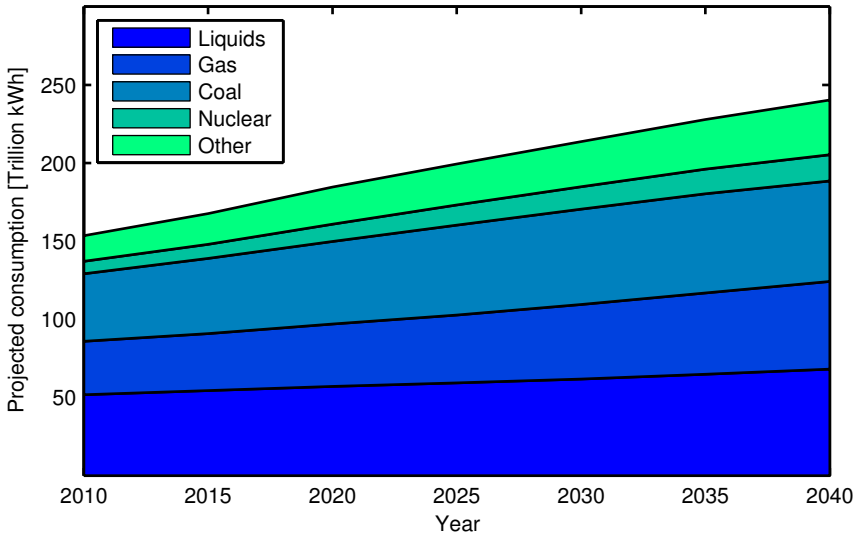

Figure 1.2: Worldwide electricity consumption by fuel type [2].

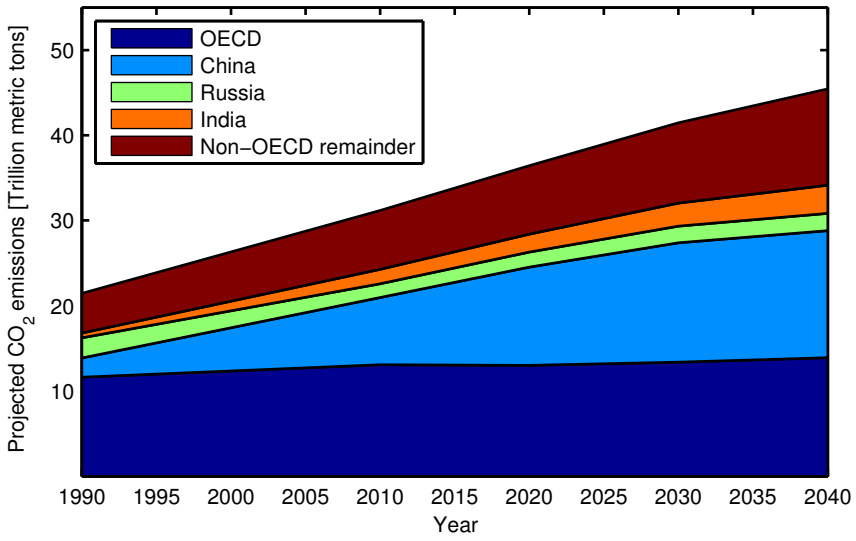

Figure 1.3: Projected CO_2 emissions by country [2].

inexpensive cost of natural gas and coal plants, they tend to be the first choices for rapidly growing countries, resulting in a sharp increase in fossil-fuel consumption (Figure 1.2) and CO_2 emissions (Figure 1.3).

Fortunately, the ever increasing need for cheap, clean, reliable energy is advancing our understanding in non-CO_2 emitting energy technologies. While hydropower is growing in non-OECD countries, maximum capacity will soon be reached as dam-worthy sites will become rare [2]. Wind and solar power are projected to grow considerably within the next several decades, however the high cost of each is projected to keep the total market share of each low, about 5% for wind and 1% for solar in 2040.

Nuclear energy is also receiving a fresh look as energy needs increase. Although suffering from a somewhat turbulent past, nuclear power is projected to see growth in the coming decades (Figure 1.2). The high energy density and lack of CO_2 emissions makes nuclear energy an attractive alternative to magange global greenhouse gas concentrations. Similar to fossil fuels, uranium depletion may soon become a problem; Between 2009 and 2011, the amount of cheap (less than \$80/kgU) identified uranium resources (reasonably assured resources and inferred resources) dropped 32.2%. Although the amount of identified uranium greater than \$80/kgU increased by 11.1%, the amount of easily recoverable uranium is likely to decrease over time [4]. In an effort to increase cost effectiveness and avoid a uranium shortage, an attractive alternative to the current fleet of thermal nuclear reactors in the US are fast breeder reactors (FBR) such as the Gas Fast Reactor (GFR), Sodium-cooled Fast Reactor (SFR), and Lead-cooled Fast Reactors (LFR).

1.1 Fast Breeder Reactors

Both thermal and fast reactor fuel consists of fissile (U-235 and Pu-239) and fertile (U-238) isotopes.[†] Energy is produced through the fissioning of fissile material. Neutrons produced during the fission event can be utilized to "breed," or produce, additional fissile isotopes through neutron capture (n) with uranium 238, and subsequent beta decay (β^-) into plutonium 239:

$$^{238}\mathrm{U} + n \rightarrow {}^{239}\mathrm{U} + \beta^- \rightarrow {}^{239}\mathrm{Np} + \beta^- \rightarrow {}^{239}\mathrm{Pu}. \tag{1.1}$$

Although this process occurs in typical Light Water Reactors (LWRs), the physics involved in fast reactors allows more fuel to be created than burned. The capacity of a reactor to breed is measured by the "breeding ratio":

$$BR = \frac{\text{fissile material produced}}{\text{fissile material consumed}} \tag{1.2}$$

By definition a breeder reactor has a breeding ratio greater than 1, meaning more fissile material is being produced than consumed.

The higher breeding ratio in fast reactors allows the more abundant fertile fuel isotopes to be utilized. Using a "seed" starter of fissile fuel, a surrounding fertile fuel "blanket" can be converted to fissile fuel over years of irradiation. The benefit of this configuration is that the blanket can be anything that contains uranium: natural, depleted, or enriched uranium, or even spent nuclear fuel. Bred fissile fuel can either remain in place to fuel the reactor or be recycled as the seed in a new plant.

Although uranium oxide fuels are overwhelmingly used in current reactors, many nuclear fuel types have been proposed and tested for use in FBRs. Metal fuel benefits from a high breeding ratio and high thermal conductivity and was used in most US fast reactors. However, metal fuels experience large swelling at only small irradiation times, requiring a high initial porosity fuel to uptake the large volume increase. Due to their familiarity, oxide fuels (mixtures of UO_2 and PuO_2) were initially favored. The high melting temperature of oxide fuel offsets its low thermal conductivity and initial studies showed high-burnup capability [5]. However, due to the presence of one metal atom to two non-metal atoms, oxide fuels have a lower heavy metal (uranium, plutonium) density, resulting in a lower breeding ratio from the softening of the neutron spectrum [6].

Carbide fuels (UC, PuC: hereby represented as just UC), and to a lesser extent, nitride fuels, received much attention in past FBR reactor programs [7]. Uranium carbide in particular is experiencing a renewed interest in the General Atomic (GA) EM^2 design [8]. By increasing the heavy metal density, a large breeding ratio is achievable. The metallic-like behavior of carbide fuels results in a high thermal conductivity, more than three times that of oxide fuels, allowing the design temperature of the coolant to be higher without melting the fuel. Carbide fuels also exhibit higher

[†]U-233/Th-232 are fissile/fertile isotopes utilized in some fuel designs as well.

power densities, requiring smaller cores for equal power output. While carbide fuel has favorable characteristics that make it an ideal candidate for FBR, there is still a lack of consistent material data and operational experience. Despite many experimental programs that have irradiated carbide fuels in both thermal and fast reactors, the bulk of the experimental work was completed 30 or more years ago; the dated methods, records, and material characterization have resulted in large uncertainties.

One of the greatest difficulties with uranium carbide was the high swelling, and eventual pellet-clad mechanical interaction. The production of fission gas inevitably leads to the complex and intertwined swelling and fission gas release phenomena which has been extensively studied in the past, but never fully captured in fuel performance simulations [9].

1.2 Fission Gas

Due to the inherent nature of heavy element fission, an accumulation of amazingly varied elements constantly keeps the fuel in chemical and mechanical imbalance. Along with several neutrons, a single uranium atom splits into two highly energetic fission fragments during fission. As these fragments bounce around within the fuel lattice, they eventually come to rest, becoming fission products and leaving a zone of displaced atoms in their wake. In general, the radioactive fission products are relatively benign and can form varied chemical compounds within the fuel. Others, such as xenon and krypton are volatile, resulting in the so-called "fission gas." Fission products are created on the order of 500 ppm per day, quickly overtaking the number of as-fabricated impurities, which are typically on the order of 1000 ppm. Table 1.1 displays a sample distribution of fission products after fissioning of 100 atoms of Pu-239.

While most atoms remain stationary after becoming fission products, volatile elements by definition will not stay in solution within the fuel lattice and will tend to thermally diffuse, forming bubbles or percolating as single gas atoms toward the grain boundaries. Furthermore, the highly energetic fission fragments constantly streaming through the fuel can knock fission gas atoms out of the bubbles, resulting in a bubble population that is dependent on both material properties (i.e. thermal conductivity, diffusivity) and state properties (i.e. temperature, fission rate). In high burn-up fuels, such as those required in fast breeder reactors, both inter-granular and grain-boundary bubbles made up of fission gases define the microstructure of the fuel: the highly temperature-dependent diffusivity of fission gas results in little bubble formation at colder temperatures and a highly porous structure near the hot interior. Since bubble interconnectivity is directly related to how much of the fission gas escapes from the fuel, an understanding of the atom and bubble behavior is essential in determining the total fission gas release.

In the simplest of terms, the fission gas release can be calculated using a spherical grain model which solves for the grain and grain boundary concentrations using empirical diffusivity values [10]. Complicating this simple model is the tendency of the insoluble fission gas to collect and form bubbles within the fuel grain before reaching the grain boundary. In addition, highly energetic fission fragments can strike the intra-granular bubbles, resulting in re-solution of gas atoms back

Table 1.1: Fission fragment distribution from the fissioning of 100 atoms of Pu-239 [9].

Atom	Yield (at. %)	Compounds
Noble gases		
Xe	22.7	
Kr	2.5	
Volatiles		
Cs	19.2	
Rb, Te, I	5.8	
Earth Alkalines		
Sr, Ba	10.1	BaC_2, SrC_2
4d metals and Ag		
Zr	19.4	Mono, di, and
Mo	22.4	sesqueicarbides
Ru	21.9	
Pd	13.0	
Y, Nb, Tc, Rh, Ag	15.0	
Lanthanides and La		
Ce	12.8	Mono, and
Nd	13.8	sesqueicarbides
La, Pr, Pm, Sm, Eu, Gd, Tb	16.3	

into the fuel lattice. Many models account for these different trapping and re-solution behaviors by empirically adjusting the diffusivity [5, 11–13]. However, estimation of fission gas in its different physical forms may result in more accurate calculation of fission gas release [14].

Due to physical differences between UC and UO_2, it is known that the re-solution rate, or rate at which gas atoms are knocked out of bubbles and back into solution, is much lower in UC fuels [15]. Since the bubble size distribution depends on the competing processes of thermal absorption of gas atoms and the knock-out of single gas atoms from bubbles, the smaller re-solution in carbide fuels allows larger bubbles to form. These large bubbles dominate the visual landscape of the fuel pin, as well as result in a sudden "breakaway" swelling rate, as discussed in Section 2.

While there are several theories as to the formation mechanism of the larger bubbles in UC, the exact cause of the sudden change in the swelling is unknown. Blank claims that bubble diffusion through the solid results in a "dumbbell" formation, causing a sharp increase in swelling due to the high surface area to volume ratio of dumbbells [9]. However, several past studies have shown the bubble mobility in uranium carbide is extremely low, limiting bubble-bubble interaction [9, 16, 17]. Another theory is that the re-solution rate decreases significantly for large bubble sizes, allowing the bubble size distribution to stabilize at high bubble sizes [9]. While the re-solution rate has been shown to be bubble size dependent, a modern calculation has yet to be completed to capture the full relationship [15].

1.3 Extending Fuel Lifetime

In order to make fast reactors more economic, the fuel is required to stay in the reactor for extended periods of time; the longer the fuel stays within the core, the more fertile material is converted into useable fissile material. Two strategies can be used to extend fuel lifetime. The first process involves removing fuel from the core after 3-7 years, reprocessing the fuel pins to separate the fissile material, refabrication of new fuel rods, and reinsertion into the core. Due to the high radioactivity of nuclear fuel after irradiation, reprocessing requires immense shielding, complicating the handling of the fuel. The shielding requirements can be reduced by implementing cool-down periods between irradiation and reprocessing, requiring a large time and money investment in fuel rods waiting in spent fuel pools [18].

The second strategy to extend fuel lifetime is to simply leave the fuel in the core for long periods of time. This can include dynamic or static fuel locations, but ultimately results in the fuel remaining in the core for the lifetime of the reactor, around 30 years, requiring robust cladding that will maintain structural integrity. One of the factors limiting fuel pin lifetime is strain on the cladding; as the fuel fissions, fission gas is continually released into the gap between the fuel and the cladding, creating a pressure differential between the pin plenum and the reactor coolant. The strain on the cladding results in creep deformation, causing the clad to bulge and ultimately leading to cladding failure through bursting or pin-to-pin fretting. The high temperature and in-pile irradiation environment accelerates the creep rate even further. The implementation of a large internal pin plenum can help extend the fuel lifetime slightly by increasing the internal pin volume.

One solution to extend the lifetime of fuel cladding has been to permit venting of the fuel pin; by allowing an escape path for the fission gas, either directly to the coolant or into a separate collection system, the differential pressure across the cladding is able to remain low. This concept was successfully implemented in the US in the now decommissioned Peach Bottom 1 reactor. Several other reactor designs have proposed to use vented fuel, such as the Gas Cooled Fast Reactor (GCFR) and GA's EM^2 modular reactor [8].

The primary safety goal in nuclear reactors is containment of both the nonvolatile and gaseous fission products. This has traditionally been achieved in the US with the "Defense in Depth" philosophy, in which the use of redundant and diverse barriers prevents release. The first of these barriers is considered to be the fuel itself: most fuels are ceramic and do a fair job of retaining solid and some gaseous fission products [5]. The second barrier is typically the cladding of the fuel. The cladding physically isolates the fuel from the coolant, and is the primary containment mechanism in the Defense in Depth concept. Other barriers include the coolant, pressure vessel, and the outer containment building. By design, vented fuels break the Defense in Depth philosophy by removing the primary fission product barrier. However, removal of activated products from the core may help reduce the emission of radioactive materials during reactor accidents. Regardless of the impact of a vented fuel system on the safety of the reactor, accurate release rates from the fuel are necessary to help quantify the source term.

1.4 EM^2

General Atomic's EM^2 design is a 500 MWth modular gas-cooled fast reactor that utilizes carbide fuel in a vented fuel pin assembly [19]. The design uses helium coolant in a direct conversion Brayton cycle configuration. The UC fuel pellet consists of many small spherical fuel kernels compacted and sintered into an annular fuel pellet. The pellets are encased in silicon carbide (SiC) cladding that contains a venting manifold at the top of the fuel pins, providing an escape path for volatile fission products that escape the fuel pellet. This side stream is diverted to a Helium Purification System (HPS) comprised of several carbon filter beds that filter radioactive isotopes. Clever utilization of the fast spectrum allows the EM^2 to breed enough fuel to allow a 30 year lifetime with no refueling. Furthermore, by utilizing the "seed and blanket" concept discussed above, the majority of the seed can be fabricated from the vast stockpiles of depleted uranium.

In an effort to be economically competitive, the innovations discussed above (Gas Fast Reactor, uranium carbide fuel, fuel pin venting) are all implemented in EM^2. However, understanding of uranium carbide behavior fuel during a 30 year residence time is limited to only a small number of past experiments. Using the EM^2 design as an example, the calculation of fission gas behavior may be approached through a collection of first-principles models into a single modular code, extending the prediction of fission gas behavior past current experimental limitations.

1.5 Research Objectives

The goal of this work is to progress the understanding and models of fission gas bubble behavior in uranium carbide in the following ways:

1. Calculate the re-solution parameter in uranium carbide using modern tools and techniques. As a fundamental parameter in determining fission gas bubble sizes, an enhanced understanding of this phenomena is necessary for bubble concentration distribution calculations that can ultimately lead to estimation of fission gas release.

2. Develop a tool for executing bubble concentration distribution calculations. By creating a modularly structured code, the phenomena describing fission gas bubble behavior in uranium carbide can be complied into a single application that can be used to simulate the bubble concentration distribution.

3. Implement the re-solution parameter in the bubble distribution code and compare calculated bubble concentration distributions to experimental observations.

Due to the differences between the re-solution parameter calculation and the bubble distribution studies, the theory, methods, and results for each are located within their own respective chapters. Before the chapters on re-solution and distribution studies, a chapter that contains general bubble behavior is included to establish some background. A Conclusions and Future Work chapter at the

end of the dissertation is utilized to contextualize the results, as well as provide recommendations for future efforts. Lastly, the appendices contain a uranium carbide material database, an example BUCK input deck, and the BUCK code,

CHAPTER 2: BACKGROUND

The behavior of fission gas in uranium carbide is a very complicated consequence of fission that depends on the interplay of different individual and coupled phenomena. The tendency of fission gas to form bubbles dominates the visual landscape of the fuel pin, and ultimately results in the release of fission gas products into the fuel plenum. This chapter identifies some of the characteristics of fission gas behavior in uranium carbide. Also included is a description of one of the past techniques used to model fission gas release in carbide-based fuels; Release over Birth, or R/B curves have been used in historical calculations of fission gas release. Through application of the R/B curves to a more recent reactor design, the ineffectiveness of such integral calculations can be highlighted.

2.1 Bubble Behavior

As described in Section 1.2, fission gases are insoluble and tend to thermally diffuse through the fuel. As a single gas atom moves through the fuel, it will combine with other gas atoms to form energetically favorable bubbles within the fuel grain. The behavior of these single atoms and bubbles within the fuel grain comprise the *intra*-granular domain. As fission gas atoms reach the grain edge, they will combine to create large, lenticular bubbles in the spacing between the grains. As these bubbles grow larger, they will eventually interconnect with each other until a pathway is established with an exterior surface fuel pellet and the fission gas is released into the fuel pellet plenum. The grain boundary bubbles and their connectivity comprise the *inter*-granular domain. Schematics of the fission gas progression is displayed in Figure 2.1.

Fission gas is continually created during the fission process, and is primarily comprised of xenon and krypton with smaller amounts of other volatiles such as cesium and iodine. The fission gas can either be created directly from the fission process, or result from transmutation of parent precursors to gaseous elements through beta-decay.

Since each fission fragment travels about 6 μm in the fuel material, if the fission event occurs near an exterior fuel surface, the fission fragment can recoil directly into the plenum. While this "athermal release" results in a geometrically-dependent release pathway, the vast majority of the fission fragments come to rest within the fuel's granular structure and can be described using the inter- and intra-granular models.

The crystal structure of uranium-carbide fuel is a NaCl type arrangement as a cubic lattice with alternating uranium and carbon atoms. Gas atom defects tend to be large, thus they exist in the combined defect of a uranium and carbon vacancy. As the fission gas diffuses through the fuel, the atoms can combine to form more energetically favorable bubbles. These bubbles will have reduced movement, and are typically assumed to be stationary within the lattice [9], effectively acting as

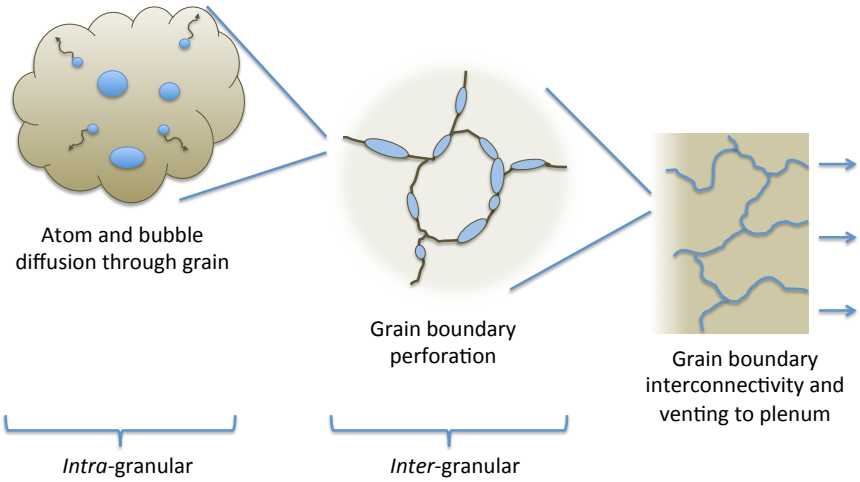

Figure 2.1: Schematic of fission gas behavior in nuclear fuel.

traps for the fission gas. During irradiation, fission fragments can strike the bubbles and knock fission gas back into the lattice, thus the number of atoms in the bubble depends on the competing processes of fission gas absorption through thermal diffusion and loss through fission re-solution. The actual size of the bubble depends on the gas atom concentration within the bubble, and requires a relationship between number of atoms and radius.

2.2 Atom Density

If an inter-granular bubble is in equilibrium with the surrounding material, the pressure p within the bubble is related to the radius of the bubble r as,

$$p = \frac{2\gamma}{r} + \sigma, \tag{2.1}$$

where γ is the surface tension of the solid and σ is the hydrostatic stress in the solid. Although noble gases can be treated as ideal gases in most cases, for bubbles below about 10 nm the van der Waals Equation of State (EOS) is more applicable [17]:

$$p\left(\frac{1}{\rho_g} - B\right) = kT, \tag{2.2}$$

where ρ_g is the atomic density of the gas at temperature T and k is Boltzmann's constant. B is a constant related to the volume occupied by the atoms, typically defined as 8.5×10^{-29} m^3/atom [17]. Combination of Equations 2.1 and 2.2 gives,

$$\frac{1}{\rho_g} = B + \left[\left(\frac{2\gamma}{kT}\right)\frac{1}{r} + \frac{\sigma}{kT}\right]^{-1}. \tag{2.3}$$

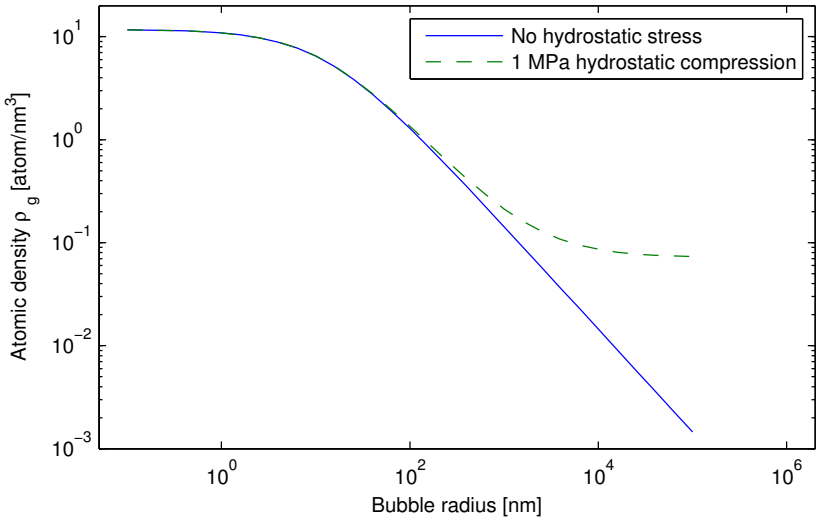

Figure 2.2: Atomic density in bubbles as a function of radius for a stress free solid and a solid with 1 MPa compressive stress.

Figure 2.2 plots the density as a function of bubble radius. In a stress free solid, the atomic density for bubbles with radius less than 1 nm goes to a constant related to the inverse of B. For large radii greater than 100 nm, the gas density in the bubble reduces to the ideal gas law:

$$\rho_g = \left(\frac{2\gamma}{kT}\right)\frac{1}{r}. \tag{2.4}$$

The atomic density is not very sensitive to temperature or surface temperature. However, in the presence of compressive hydrostatic stress the bubble is able to maintain a higher atom density for larger bubble radii (Figure 2.2).

In stress free solids, the number of atoms in a bubble m is generally calculated using the van der Waals equation of state. However, the high temperature and stress involved with very small bubbles in nuclear fuel results in an underestimation of atomic density; Ronchi computationally studied the atomic density of gas atoms in UC and showed the atom concentration was indeed higher for small bubbles [20]. More recent evaluations show that the atomic density may be very near the atom concentration for solid xenon in UO_2 [21], however the applicability of Ronchi's data to UC fuel makes it more appropriate for use in uranium carbide simulations. A smooth transition from Ronchi's data for small bubble sizes to values calculated using Equation 2.2 can be utilized to calculate the atom density in the bubble due to the large spread of bubble radii, giving the "Combined" model types, as displayed in Figure 2.3.

The number of gas atoms m within a bubble is calculated from ρ_g by,

$$m = \left(\frac{4\pi r^3}{3}\right)\rho_g. \tag{2.5}$$

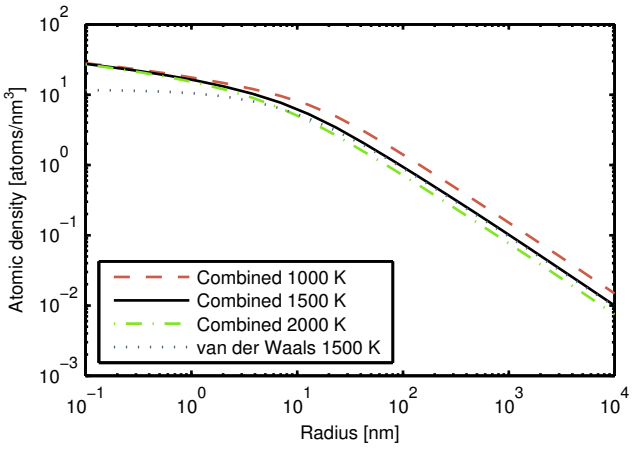

Figure 2.3: Atomic densities calculated from different EOS models.

At the lower bubble limit, the number of atoms in a bubble goes by r^3, as expected with a constant density. At the higher bubble limits, the number of atoms goes by r^2,

$$m = \left(\frac{8\gamma\pi}{3kT}\right) r^2, \quad \text{for } r > 100\text{nm}. \tag{2.6}$$

Since the volume of the bubble goes by r^3, the combination of two identical bubbles with m number of gas atoms into a single bubble with $2m$ gas atoms, results in a local swelling increase of 40%. This means as bubbles get larger and combine, the swelling will increase drastically.

2.3 Bubble Types

In general, four types of bubble concentrations are defined in UC fuels, as displayed in Table 2.1. Although bubble concentration distributions exist in all irradiated nuclear fuel, carbide based fuels contain much larger bubbles than oxide fuels due to a smaller re-solution rate in carbide fuels as a result of UC's superior thermal properties (i.e. thermal conductivity and thermal diffusivity) [9].

The P_1 bubbles are similar to the bubbles found in UO_2 fuel, while P_2 bubbles are typically found in uranium carbide or uranium nitride fuels. Although there is some recent evidence that 50-100 nm bubbles form in UO_2 at high temperatures and burnups [22], they do not result in the drastic swelling that plagues UC fuels. Although P_2 bubbles have a lower concentration, their size is highly temperature dependent and can cause problems with swelling. Both the P_1 and P_2 bubbles are included in the inter-granular model discussed above. The P_3 bubbles consist of grain-face bubbles that grow as a result of single fission gas atoms reaching the grain boundary. Once a P_3 bubble reaches an exterior fuel surface it releases its gas content to the plenum. The grain-face bubbles are typically assumed to maintain their shape for the remainder of the irradiation time and any additional gas that reaches an outside interconnected P_3 bubble is assumed to be instantaneously released to the plenum. To model release, a surface saturation point is typically calculated for each

Table 2.1: Bubble types.

Type	Size [nm]	Concentration [m^3]	Behavior
P_1	1-30	10^{21}	Nearly constant after some initial incubation time
P_2	30-400	10^{18} - 10^{20}	Dependent on temperature
P_3	50-1000	-	Increases until interconnection
P_0	10^3	-	Dependent on initial structure, decreases due to sintering
UO_2	1	10^{23}	

fuel grain, with plenum release typically occurring when 50% of the total grain boundary area is covered in P_3 bubbles [13], although some authors claim higher coverage fractions [23].

Lastly, P_0 are voids or flaws introduced into the bulk during fabrication. As the P_2 and P_3 bubbles grow, they will eventually connect with each other and P_0 bubbles.

Due to their extremely small size, few studies have been completed on the P_1 concentration. Ray and Blank performed transmission electron microscopy (TEM) to measure bubble sizes and form histogram data for both sodium and helium bonded UC fuel pins [24]. Their results showed that the bubble concentrations are relatively independent of temperature (Figure 2.4). While the P_1 swelling contribution and gas content are slightly dependent on temperature, both remain so low that the P_1 population is often ignored in simulations. Regardless, it can be inferred that the P_1 bubbles are increasing in size as a function of temperature due to the nearly constant concentration and increasing swelling.

The small size of P_1 bubbles ensures that interactions remain only between single bubbles and diffusing gas atoms in the bulk, however as P_2 bubbles grow larger, they can combine. As discussed in Section 2.1, the combination of two equal sized bubbles results in a 40% volume increase, thus a considerable amount of swelling can occur once bubbles are large enough to interact with one another. This behavior has been observed in UC fuels, and in fact causes a sharp decrease in bubble concentration as well as breakaway swelling at a characteristic temperature, T_2 (Figure 2.5).

The swelling trend due to P_2 bubbles can be divided into low and high temperature domains: at low temperatures < 1100K, the swelling is nearly linear, while at higher temperatures the swelling suddenly decreases (This behavior is only visible in Figure 2.5 as the last two data points for 11 a/o, but occurs at all burnup values [9]). The low temperature behavior can be attributed to bubble size increase from single atom absorption. As temperature crosses the threshold T_2, a sharp increase in swelling rate occurs as a result of a growth mechanism that has yet to be identified.

Blank presents several theories as to the presence of P_2 bubbles and their sudden break-away swelling growth rate [9]. The first is that the re-solution parameter decreases for larger bubble sizes, allowing larger bubbles to grow; A decrease in the re-solution parameter as a function of bubble size has been shown previously [15], but modern day analysis have yet to be completed. A second theory is that there is some sort of synergistic relationship between solid precipitates that are have been empirically shown to be nearly always associated with large P_2 bubbles [24]. A final theory

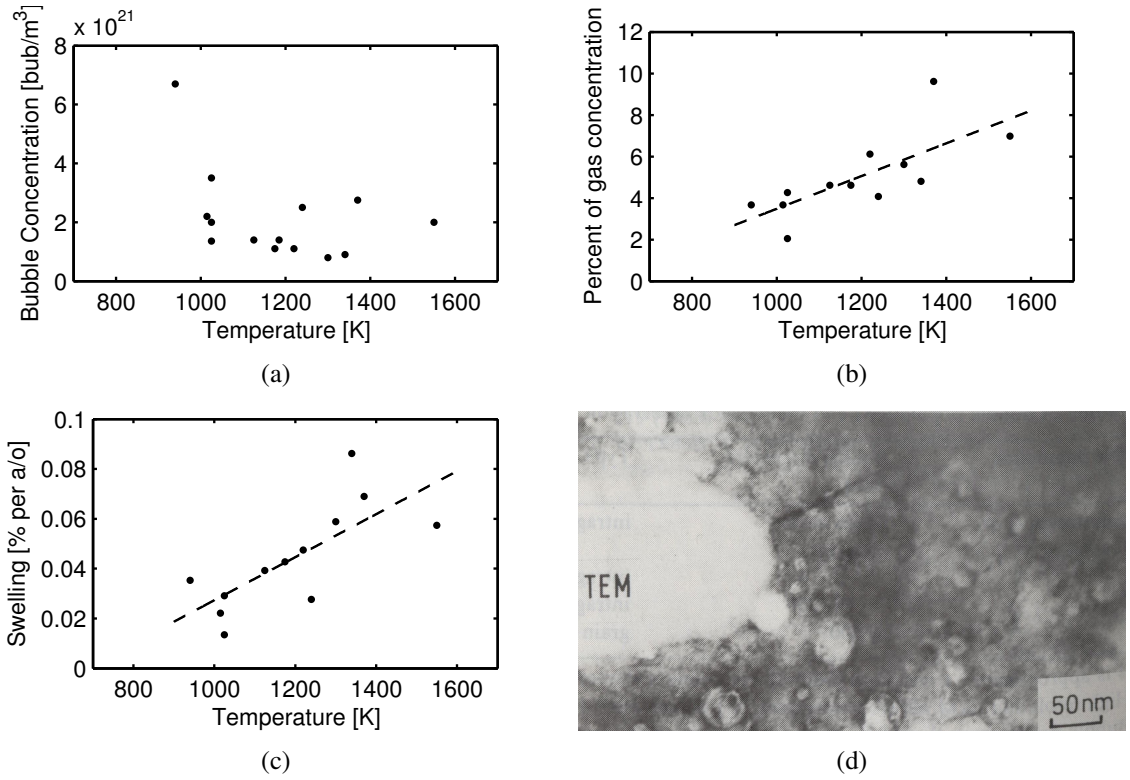

Figure 2.4: P_1 bubble behavior: (a) Concentration as a function of temperature, (b) percentage of the total gas concentration, (c) percentage of swelling per atom percent burn up, and (d) micrograph of P_1 bubbles intermixed with some larger P_2 50 nm bubbles.

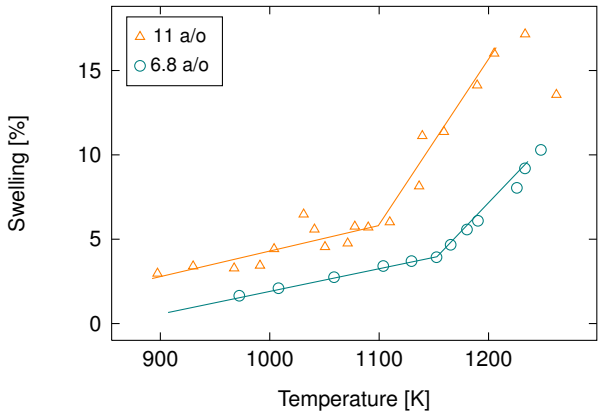

Figure 2.5: P_2 swelling behavior in UC fuel [9].

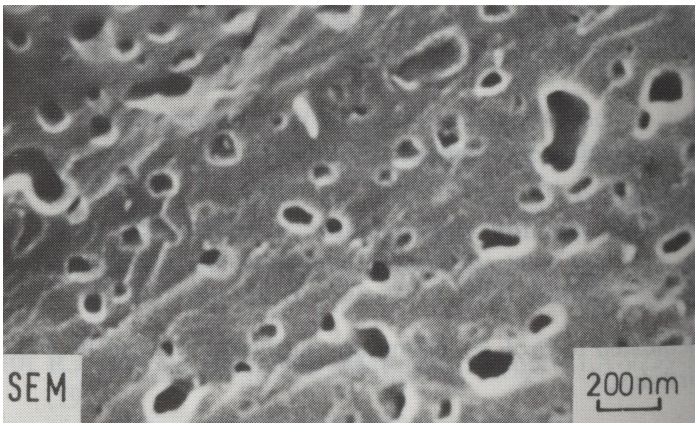

Figure 2.6: SEM micrograph of dumbbell coalescence in uranium carbide fuel [9].

is that the bubbles themselves diffuse through the solid, combining due to dumbbell coalescence, similar to behavior captured in a some UC micrographs (Figure 2.6).

As the P_2 bubbles get larger and larger, they tend to combine with both the P_3 and P_0 bubbles, resulting in a decrease in swelling contribution at high temperatures, and forming a highly porous structure in the central regions of the pin. The different regions of P_2 bubble behavior results in a 4-zone structure that has been adopted in describing irradiated carbide fuels. The different regions are described in Table 2.2. Since most of the fission gas released originates from the interior zones of the fuel pin, the larger the fraction of pellet that remains as zone IV, the lower the fission gas release and swelling.

Figure 2.5 compares the difference in P_2 swelling from two irradiation tests, showing a slight dependence on burnup: as burnup increases, T_2 decreases. By comparing many different experiments, the so-called T_2 curve can be plotted as a function of burnup (Figure 2.7). The cause of this behavior is not fully understood. One cause may be that the earlier breakaway swelling is due to a higher concentration of fission gas at higher burnups. A second cause may be due to a change in fission gas diffusivity, and is discussed below.

2.4 Diffusivity Dependence

The growth of bubbles is dependent on the ease of which fission gas atoms can diffuse through the fuel, or the diffusivity D:

$$D = D_0 \exp\left(\frac{-Q}{RT}\right) \tag{2.7}$$

where D_0 is the intrinsic diffusivity, Q is the activation energy and R is the ideal gas constant. Because of the temperature dependence of D, larger bubbles are more prevalent at higher temperatures as single atoms move through the bulk easier and combine with bubbles more often.

The diffusivity can also be modified by the chemistry of the system. The GOCAR irradiation tests studied fuel with various ratios of nitrogen, oxygen, and carbon [27]. In general, the atomic

Table 2.2: Structural zones in UC Fuel.

Zone	Properties	Required Conditions
IV	*Zone of low swelling and base release:*	$T < T$
	As-build structure	Typically below 1100-1300K
	Slight densification	
	Small P_1, P_2, P_0 bubbles	
III	*Transition from low to high:*	$T > T_2$
	Sudden change to high rate of swelling	Width of zone 200K
	Grain Growth	
	Increasing release up to 70%	
	Increasing P_3 bubbles	
	P_3 interlinkage in hotter regions	
II	*Pseudo columnar grain zone:*	High T *and* high ΔT
	Not always present	Typically only for >120 kW/m
	Fuel densification	
	Possible columnar grain growth	
	High gas release	
	Low swelling	
I	*Porous structure:*	High T
	Usually present	Central part of pin
	Highly porous	
	High gas release	
	Low swelling	
	Central hole possible	
	Pore size equals grain size	

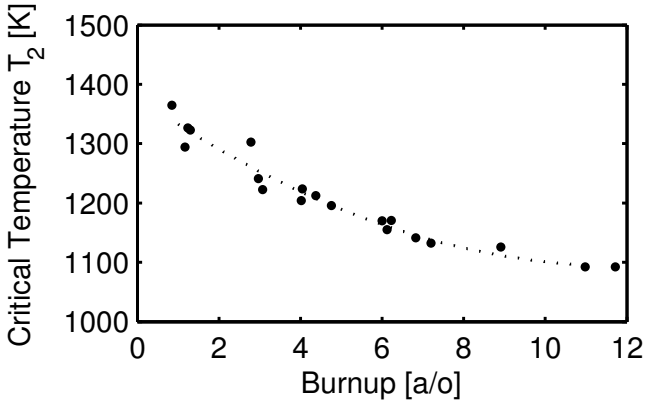

Figure 2.7: Critical temperature vs. burn up for several irradiation tests [25–28].

mobility in nitride fuel is slower than carbide fuels [5], thus as the nitrogen/carbon ratio increases, the fission gas diffusivity lowers, resulting in smaller bubbles and an increase in the critical temperature, matching the observed behavior from the GOCAR experiments; The higher the ratio of

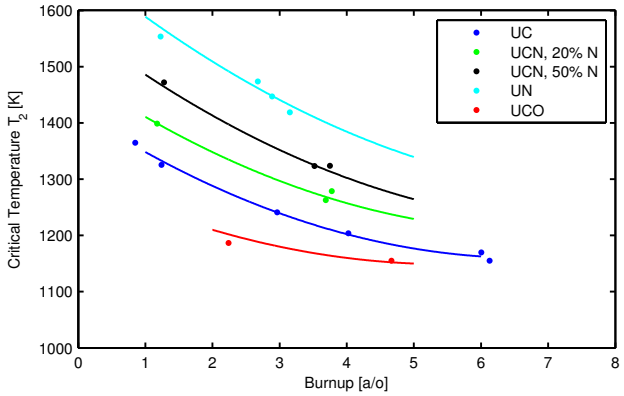

Figure 2.8: Critical Temperatures of MX fuels [27].

nitrogen in the fuel, the higher the critical temperature (Figure 2.8). Conversely, the increase of oxygen in the fuel increases diffusivity, resulting in a lower T_2; The GOCAR irradiation results repeated in Figure 2.8 compared UC fuel with a 3% oxygen impurity content to UCO fuel with 8% oxygen content. Although oxygen was present in only a small fraction, the critical temperature dropped by nearly 80 K at 2.2 a/o burnup (Figure 2.8). Extending this behavior to the concept of structural zones in UC, the higher the oxygen content or lower the nitrogen content, the smaller zone IV will be, resulting in higher swelling and fission gas release.

The dependence of fission gas release on oxygen is also evident by examining fission gas release data from several irradiation tests, displayed in Figure 2.9 [9]. It was originally thought that fuel form, both in diameter and type, played the primary role in curbing fission gas release through direct control of centerline temperatures [7]. However, fuel irradiated in the 1980s as part of the AC-3 test experienced extremely low fission gas release despite similar irradiation conditions as previous tests, with the primary difference being low oxygen impurities [29]. Although temperature will always be the driving factor of fission gas release, the low oxygen concentration in the AC-3 tests raised the critical temperature threshold, avoiding the breakaway swelling and the bubble interconnectivity that often results in high fission gas release.

2.5 R/B curves

In an effort to highlight some of the difficulties in relying on purely experimentally formulated models, an example of a previous method to calculate fission gas release will be explored here. For many of the fission gas release calculations utilized for licensing, a Release over Birth (R/B) curve can be used to compartmentalize all of the effects relating to fission gas release into a single plot such as Figure 2.10. This plot contains an estimated release rate over birth rate as a function of isotope half-life, and is formulated using experimental data of total fission gas release from previous experiments. For carbide fuels, only a couple curves have been formulated, the first of which was used to license Peach Bottom Unit-1, one of the few gas reactors to ever operate in the United

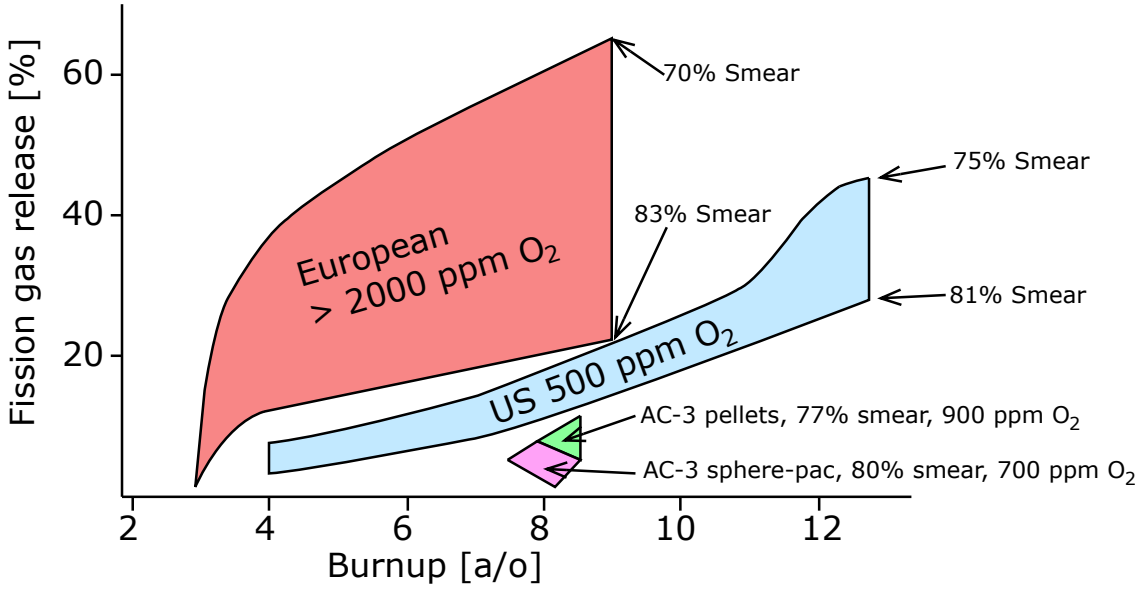

Figure 2.9: Fission gas release from several UC irradiation tests [9].

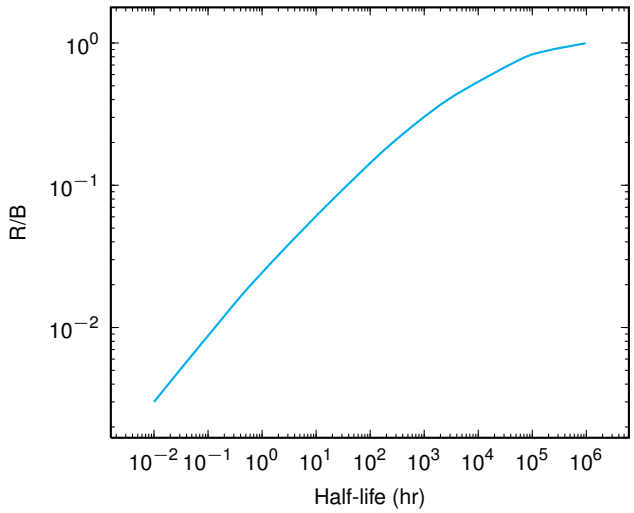

Figure 2.10: Peach Bottom's R/B curve.

States. The second model was developed for the Modular High Temperature Reactor (MHTGR), a design that has been extensively studied but never built. Finally, application of the MHTGR to the more modern EM^2 will highlight some of the difficulties in using an integral model to calculate the intricate fission gas behavior.

2.5.1 Peach Bottom

Figure 2.10 displays R/B curve used in the Peach Bottom Final Hazards Summary Report [30]. The data for the curve was taken from linear accelerator experiments on uncoated (Th,U)C_2 fuel

particles. This R/B curve was calculated at a non-specified temperature for sole application to the Peach Bottom reactor design, with no corrections for burnup or differing gas species

2.5.2 MHTGR Model

A more complicated calculation of the R/B release curve for carbide fuel can be found in the 1993 MHTGR model database compiled by Martin [31]. Coefficients for R/B calculations were provided for $UCO_{1.6}$, $UCO_{1.1}$, ThO_2, UC_2, and UO_2 fuels. In general, the diffusive behavior of gas was reduced to xenon-like and krypton-like behavior. It is assumed that iodine and tellurium release is similar to xenon, and selenium and bromine release is similar to krypton.

The general model equation for unhydrolyzed fuel is given as:

$$(R/B) = \frac{(R/B)_0 \cdot f(T) + f_s(T)}{1 + f_s(T)}, \tag{2.8a}$$

$$(R/B)_0 = 3\left(\frac{\xi_{jk}}{\lambda_i}\right)^n, \tag{2.8b}$$

$$f(T) = c + (1-c)\exp(Q_{jk}/RT)\exp(-Q_{jk}/RT_0), \tag{2.8c}$$

$$f_s(T) = \exp(a[T - T_s]), \tag{2.8d}$$

(R/B)	=	calculated release rate over birth rate as a function of isotope i,
$(R/B)_0$	=	steady-state fractional release at reference temperature T_0,
$f(T)$	=	correction factor based on temperature,
$f_s(T)$	=	correction factor based on structural changes at high temperature,
T	=	temperature [K],
T_0	=	reference temperature (T_0=1373 K),
T_s	=	reference temperature for high temperature structural correction [K],
ξ_{jk}	=	reduced diffusion coefficient for element j (i.e. Xe or Kr) [1/s],
λ_i	=	decay constant of isotope i [1/s],
n	=	constant (a=0.5),
c_j	=	constant for element j (i.e. Xe or Kr),
Q_{jk}	=	activation energy for steady state fission gas release k [J/mol],
R	=	ideal gas constant (8.314 J/mol/K),
a_k	=	constant for fuel type k.

The parameters used for Equation 2.8 are displayed in Table 2.3. According to the original documentation, $(R/B)_0$ is roughly related to the vacancy motion, $f(T)$ to the athermal release mechanism and overall temperature dependence, and $f_s(T)$ to the high temperature diffusion of gas atoms in grains and any possible bubble diffusion [31]. Missing from the above formulation is any burnup

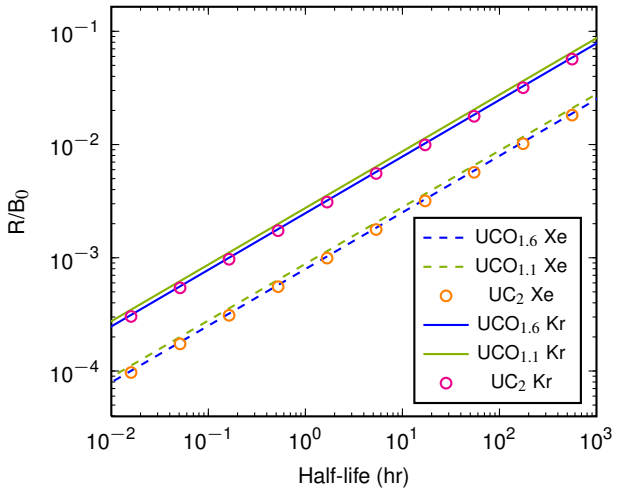

Figure 2.11: $(R/B)_0$ plotted for several fuel types and for each diffusive element.

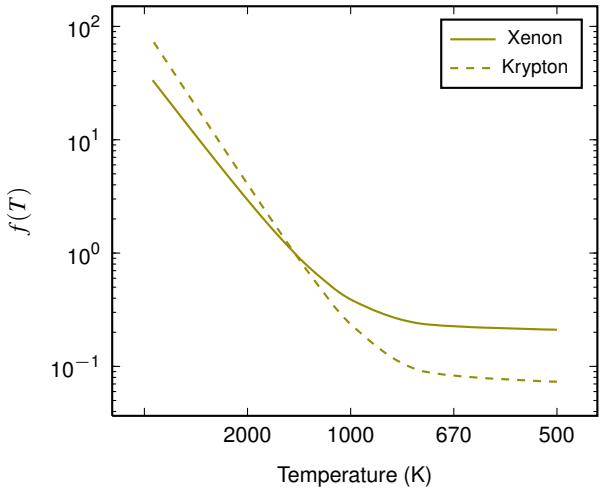

Figure 2.12: Arrhenius plot of the temperature correction $f(T)$ for Xe and Kr. The same values apply for all fuel types.

Table 2.3: Parameters for Equation 2.8 [31].

Parameter	$UCO_{1.6}$ Kr	$UCO_{1.6}$ Xe	$UCO_{1.1}$ Kr	$UCO_{1.1}$ Xe	UC_2 Kr	UC_2 Xe
ξ_{jk} [1/s]	$1.31\text{x}10^{-10}$	$1.35\text{x}10^{-11}$	$1.62\text{x}10^{-10}$	$1.67\text{x}10^{-11}$	$1.23\text{x}10^{-10}$	$1.26\text{x}10^{-11}$
c_j	0.073	0.21	0.073	0.21	0.073	0.21
Q_{jk} [J/mol]	$5.131\text{x}10^4$	$4.52\text{x}10^4$	$5.131\text{x}10^4$	$4.52\text{x}10^4$	$5.131\text{x}10^4$	$4.52\text{x}10^4$
a_k [1/K]	0.0128		0.0128		0.0133	
T_s [K]	1800		1800		1700	

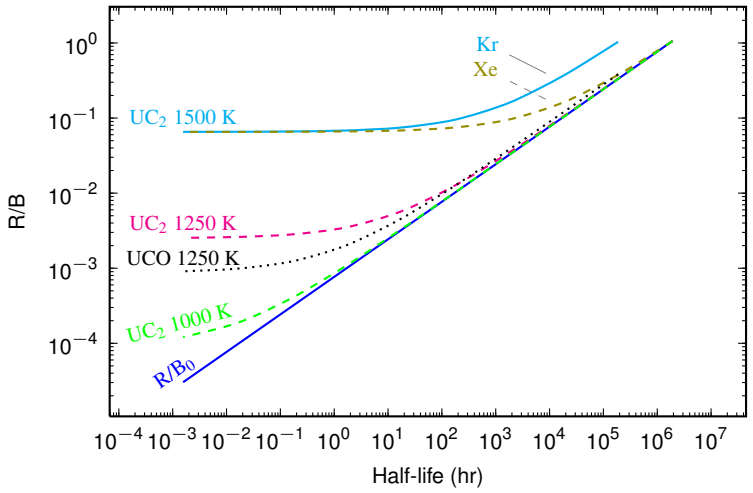

Figure 2.13: MHTGR Xe R/B values compared to the uncorrected R/B_0 curve. Here, UCO corresponds to $UCO_{1.6}$. $UCO_{1.1}$ is omitted since it is rougly same as $UCO_{1.6}$. The UC_2 Kr R/B curve at 1500 K is included for comparison.

dependence; A correction function for oxide fuel types was included in [31], however since the focus of the current work is on carbide fuel types, any R/B calculations for the oxide fuels, and thus burnup dependence has been dropped from Equation 2.8.

The $(R/B)_0$ term depends purely on the decay constant of the isotope of interest, and varies for fuel type k and element behavior j. $(R/B)_0$ is plotted in Figure 2.11 for several fuel types and for Xe and Kr diffusive properties. The UC_2 and $UCO_{1.6}$ curves are nearly identical, while the $UCO_{1.1}$ $(R/B)_0$ values are slightly higher. In general, the R/B values for Kr tend to be an order of magnitude higher than Xe.

The temperature correction function $f(T)$ is somewhat sensitive to temperature (Figure 2.12). The exponential behavior of the structure correction function $f_s(T)$ provides the greatest modification to the R/B values, especially for species with small half-lives. Figure 2.13 displays the corrected R/B calculated from Equation 2.8. Although the $f(T)$ correction is included, the $f_s(T)$ dominates the deviation from $(R/B)_0$. Due to differences between fuels in the $f_s(T)$ coefficients, the corrected R/B curve for UC_2 is higher than the corresponding UCO curves.

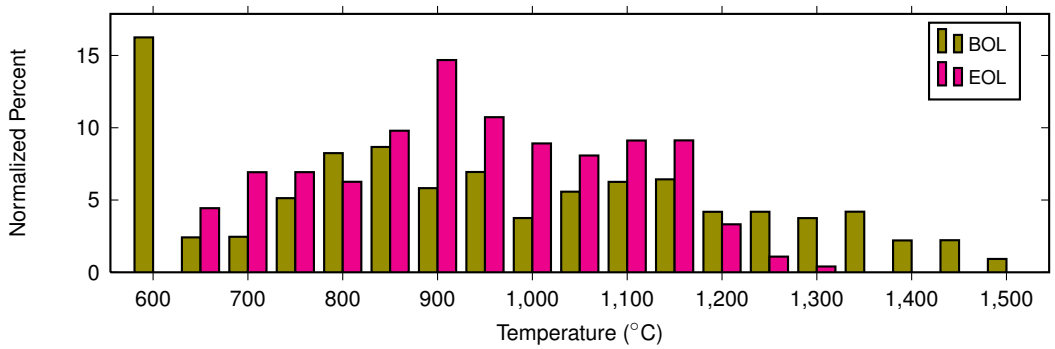

Figure 2.14: Estimated fuel temperatures in EM^2 for Beginning of Life (BOL) and End of Life (EOL) [32].

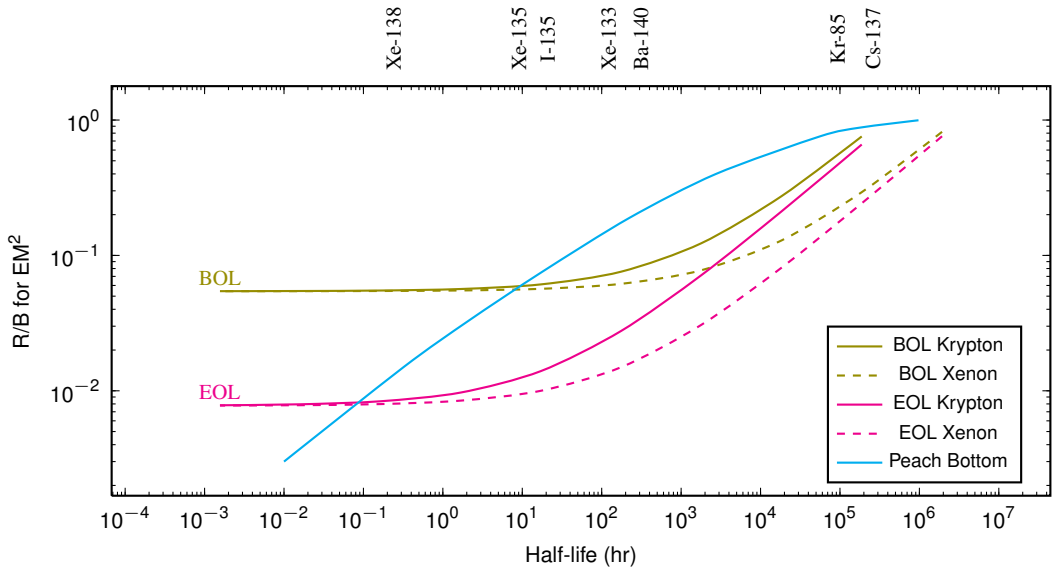

Figure 2.15: R/B values for EM^2 estimated fuel temperatures [32].

2.5.3 EM^2 R/B

In order to correctly apply the MHTGR model, knowledge of the fuel temperature distribution is necessary to accurately estimate R/B since the exponential behavior of the $f_s(T)$ makes an ordinary volume averaged temperature inappropriate. As an example, the R/B model can be applied to the EM^2 reactor design. Figure 2.14 displays some preliminary fuel temperatures for Beginning of Life (BOL) and End of Life (EOL) of a typical EM^2 fuel cycle. The peak/average fuel temperatures are 1700/990° C for BOL and 1400/1000° C for EOL.

The temperature distribution weighted MHTGR R/B curves for the estimated EM^2 BOL and EOL behavior is displayed in Figure 2.15. The parameters for UC_2 fuel were assumed for the EM^2 uranium mono-carbide fuel. Although a large number of fuel elements at BOL are relatively cold (<600° C), the R/B curve is actually higher due to the few high temperature fuel elements. The narrow EOL temperature distribution helps keep the fuel out of the high temperature region.

The R/B curves in Figure 2.15 can be represented as a two term power model. The R/B for diffusive behavior j (i.e. Kr or Xe) and isotope i can be estimated by:

$$R/B = a + \left(\frac{b}{\lambda_i}\right)^{1/2}, \tag{2.9}$$

where λ_i is the decay constant of the the isotope in [1/s], a is a constant that depends on the temperature distribution (a_{bol} = 0.0543 or a_{eol} = 0.0078), and b is a constant that depends on the diffusive behavior (b_{Xe} = 5.74e-11 [s] or b_{Kr} = 4.71e-10 [s]). In this way, the R/B can be simply calculated for each temperature distributions.

2.5.4 R/B Model Discussion

Initially, it was believed that the complexities present in the newer MHTGR model would result in lower fission gas release than the single Peach Bottom curve. While the narrow EOL temperature distribution does result in less release of gaseous fission products, the release actually increases for the BOL temperature distribution: a small percentage of hot fuel rods at BOL cause an increase in fission gas release, especially of isotopes with short half-lives. In general, the Peach Bottom curve lies between the BOL and EOL. The uncertainties involved affect both models equally (i.e. similar but different fuel types, thermal vs. fast reactor design, close packed vs. graphite dispersed particles), thus neither model is fully adequate. Regardless, the temperature dependence of the MHTGR allows for potentially more interesting calculations.

Even though there is a higher complexity level included in the MHTGR model, there are still many assumptions and uncertainties involved:

- The R/B curve was formulated from release rates of short-lived isotopes (half-life <5.3 days), although Martin claims that it is also applicable to long lived isotopes [31]. While this seems true, the curve would most likely asymptotically approach a release of 1.0, similar to the behavior of the Peach Bottom curve (Figure 2.10). Since the MHTGR R/B curve linearly approaches 1.0, it will tend to overestimate the release of long-lived isotopes.

- The MHTGR model supplies R/B curves only for Xe, Kr, I, Te, Br, and Se. Other species are assumed to follow a diffusion type models.

- The formulated R/B curves are for UC_2, not for the UC fuel present in the EM^2 core. There is some evidence that UC_2 has diffusion rates an order of magnitude slower than UC [5, p. 493].

- The MHTGR model was based on release rates from failed coated particles (diameter = 200 μm) in thermal reactor designs. There is some evidence that the release mechanism is primarily from recoil events [33]. In MHTGR fuel designs, the fuel particles are embedded in a graphite matrix in which recoiling atoms can be trapped before entering a neighboring particle. The close-packed graphite-free kernels required for fast reactor fuel designs will result

in a reduction of recoil release as knocked out atoms will have a high chance of reentering an adjoining fuel particle.

In general, the limitations of a model based purely on experimentally derived data is evident. Even when the complexities of the MHTGR model are taken into account, the Peach Bottom and MHTGR models produce similar results. Furthermore, the models are limited to the narrow band of experimental parameters utilized to make the relationships. In order to fully understand the fission gas behavior, a new approach to that builds upon first-principle models of the phenomena is required.

CHAPTER 3: RE-SOLUTION

The radiation field in a nuclear reactor results in a region of high chaos, damage, and instability in nuclear reactor fuel. One of the most consequential microscopic effects that results in an observable outcome is the behavior of fission gas; the gas atoms that are created as a product of fission move through the lattice, eventually finding one another and forming bubbles. These bubbles are then subject to other fission fragments constantly streaming through the fuel, resulting in the knock out of gas atoms back into the lattice. This loss event, or "re-solution" process, plays a large part in limiting the size of fission gas bubbles. A thorough understanding of re-solution is important for any future fission gas bubble models. While the vast majority of studies on re-solution have focused on uranium oxide fuels, the inherent material differences that uranium carbide benefits from makes it an ideal candidate for less computationally expensive modeling methods. The following section focuses on the re-solution process in uranium carbide. Through the use of a new code 3D Oregon-state Trim (3DOT), the re-solution parameter is calculate as a function of bubble radius for use in future fission gas bubble dynamics models.

3.1 Background

The re-solution event has been described as either a collisional atom-by-atom loss, or though a complete destruction process [34]. The latter theory first developed by Turnbull [35], termed "heterogenous" re-solution, assumes a bubble is destroyed if it lies within a destruction zone of approximately a nanometer around a given fission track, as portrayed in Figure 3.1a. A second "homogenous" theory originally presented by Nelson [36] assumes that individual gas atoms are knocked out of bubbles through direct collisions with fission fragments or from fuel cascades produced in the fuel as portrayed in Figure 3.1b. Many previous studies have looked at one or both of these theories in UO_2 [11, 14, 15, 37–40]. In general, the homogenous theory tends to underestimate the re-solution rate in UO_2, while success has been achieved with the heterogenous theory [14, 15, 41]. Although fewer studies have looked at re-solution in non-oxide fuels, a study by Ronchi and Elton showed the homogenous theory produces favorable results in uranium carbide fuels [15].

The ability of the heterogenous re-solution model to accurately estimate re-solution in oxide fuels, while homogenous re-solution seems more applicable to uranium carbide fuel, lies in the inherent differences between the two materials; uranium monocarbide is bonded by a mix of covalent and metallic contributions [9], resulting in different energy transport properties such as higher thermal conductivity and diffusivity when compared to UO_2. Fission fragments are well known to create displacement spikes in materials [17]. Following the lattice disorder, a local temperature increase on the order of 2000° C occurs in UO_2, along with a mechanical shock that moves through

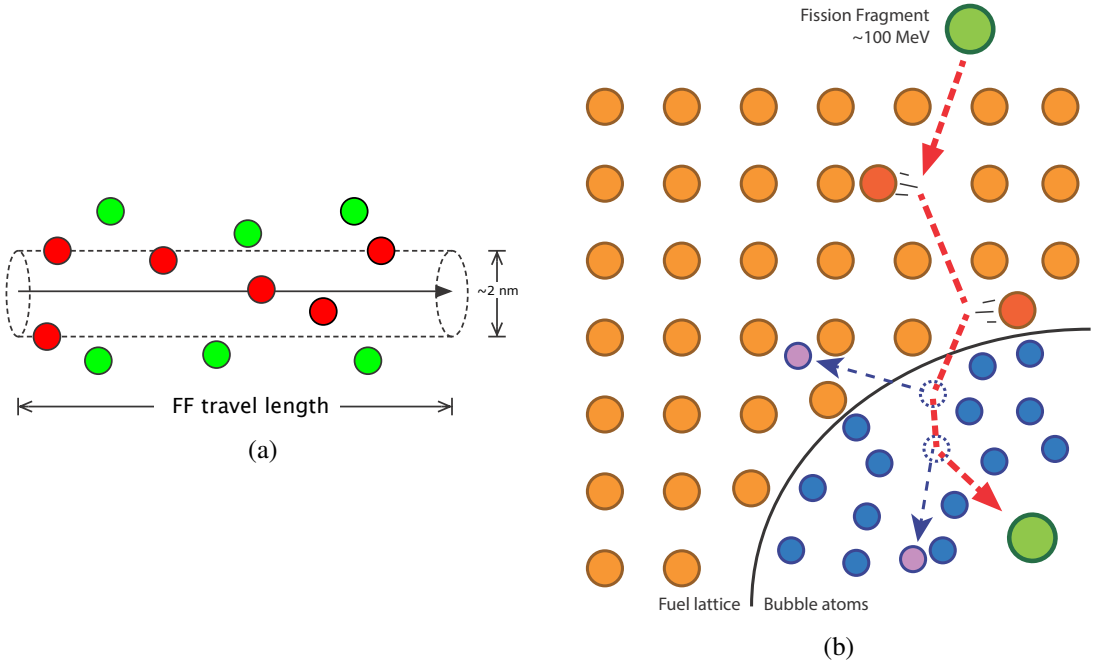

Figure 3.1: Schematics of the a) heterogenous, and b) homogeneous theories of re-solution.

the lattice as a result of rapid thermal expansion [15, 42]. However, theoretical calculations have shown that considering the higher electrical conductivity and thermal diffusivity present in uranium carbide which permits rapid dispersal of energy, the temperature increase is on the order of only 50 K [15]. As a result, the thermoelastic stress field caused by a thermal spike is nearly absent in UC [43, 44].

In light of the differences between UC and UO_2, it is understandable that the bubble destruction model has seen success in oxide fuels, while the atom-by-atom knockout microscopic model is more applicable to carbide fuels [15]. Because of this, the typically used, albeit computational expensive, Molecular Dynamics (MD) fission spike models of determining re-solution can be avoided in favor of Binary Collision Approximation (BCA) models.

In his original 1969 formulation, Nelson estimated the re-solution rate of a 5 nm xenon gas bubble in UO_2 by assuming a Rutherford potential for fission fragment collisions and a hard-sphere potential for all other collisions, with no electronic losses [36]. An atom was assumed to be implanted back into the fuel if a gas atom located within a critical distance of 1 nm from the surface of the bubble was struck by an ion and transferred the minimum implantation energy E_{min} = 300 eV [36]. The critical distance limit assumed that all centrally located fission gas atoms would suffer a large angle collision before reaching the bubble surface, while atoms close to the bubble surface result in a re-solution event if struck with sufficient energy. Ronchi and Elton extended the calculation by allowing E_{min}, as well as the radius of the bubble, to be a variable parameter of the calculation in oxide and carbide fuels [15]. Ronchi and Elton utilized the critical distance criterion for re-solution as well, ignoring the centrally located atoms in the bubble. However, due to the larger bubble sizes

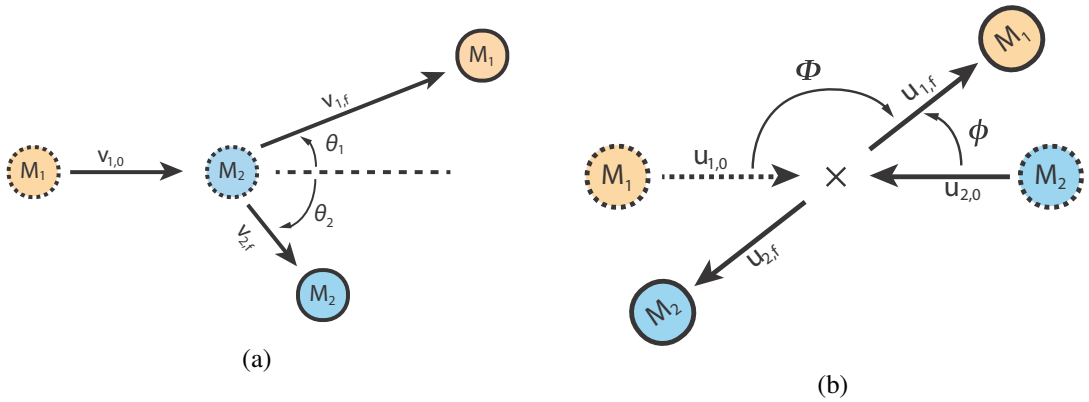

Figure 3.2: Schematics of the two-body collision in a) laboratory and b) Center of Mass coordinates.

in their study, the critical distance was not held constant, but rather allowed to increase in order to compensate for the lower atomic densities present in larger bubbles.

3.2 Theory

3.2.1 Binary Collision Dynamics

The classical description of the collision of two particles requires only the initial velocities and trajectories of the two particles to determine dynamics of the system through conservation of momentum and energy. While the charge and interaction between two subatomic particles somewhat complicates the classical view, the geometry and setup of the problems are the same.

Consider a moving atom, or "ion," with mass M_1 and velocity $\mathbf{v}_{1,0}$ colliding with an initially stationary target atom with mass M_2 (Figure 3.2a). The ion will scatter off at some angle θ_1 with a new velocity $\mathbf{v}_{1,\mathbf{f}}$, while the target atom will recoil with some velocity $\mathbf{v}_{2,\mathbf{f}}$ and angle θ_2. Applying conservation of energy and momentum yields:

$$E_0 = \frac{1}{2}M_1 v_{1,0}^2 = \frac{1}{2}M_1 v_{1,f}^2 + \frac{1}{2}M_2 v_{2,f}^2 = E_f, \tag{3.1a}$$

$$M_1 v_{1,0} = M_1 v_{1,f}\cos\theta_1 + M_2 v_{2,f}\cos\theta_2, \tag{3.1b}$$

$$0 = M_1 v_{1,f}\sin\theta_1 + M_2 v_{2,f}\sin\theta_2. \tag{3.1c}$$

The above equations can be solved in various forms to describe the behavior of the particles. In an effort to simply the problem, the equations can be recast into the Center of Mass (CM) reference frame. In this way, the same collision can be reduced to a single equation of motion due only to a central force field. This simplification holds true no matter how complex the force, so long as the force only acts along the line joining them with no transverse components. The potential energy $V(r)$, can then be described using only the absolute value of the distance between the particles, r.

To convert to the CM coordinates, the frame velocity, $\mathbf{v}_{\mathbf{cm}}$, is defined such that there is no net

momentum change in the system,

$$M_1 \mathbf{v}_{1,0} = (M_1 + M_2)\, \mathbf{v_{cm}}. \tag{3.2}$$

Solving for $\mathbf{v_{cm}}$ gives,

$$\mathbf{v_c} = \mathbf{v}_{1,0} \frac{M_c}{M_2}, \tag{3.3}$$

where M_c is the reduced mass, defined by,

$$M_c = \frac{M_1 M_2}{M_1 + M_2}. \tag{3.4}$$

If the frame of reference is changed such that the center of mass is stationary (i.e., the frame is traveling at $\mathbf{v_{cm}}$), then the velocity of the particles in the CM frame can be converted by subtracting $\mathbf{v_{cm}}$:

$$\mathbf{u}_{1,0} = \mathbf{v}_{1,0} - \mathbf{v_{cm}} = \mathbf{v}_{1,0} \frac{M_c}{M_1}, \tag{3.5a}$$

$$\mathbf{u}_{2,0} = -\mathbf{v_{cm}} = -\mathbf{v}_{1,0} \frac{M_c}{M_2}. \tag{3.5b}$$

Equation 3.5 highlights one of the major benefits of the CM coordinates; The velocities of the two particles are independent of angle, and thus remain constant throughout the collision. Furthermore, the system energy in the CM frame reduces to the constant energy of a single particle with mass M_c and velocity $v_{1,0}$:

$$E_c = \frac{1}{2} M_1 u_{1,0}^2 + \frac{1}{2} M_2 u_{2,0}^2 = \frac{1}{2} M_c v_{1,0}^2. \tag{3.6}$$

Although much of the math regarding binary collisions is presented in the CM coordinates, real world application often requires laboratory coordinates, thus relationships are necessary to convert between the two. Figure 3.3 displays the vector diagram of the both the lab and CM frames. The CM scattering angle of the target, Φ, can be determined by examining the lower triangle in Figure 3.3; since $u_{2,f} = v_{cm}$, the triangle is isosceles, therefore,

$$\Phi = 2\theta_2. \tag{3.7}$$

Furthermore, since the two particles are viewed as moving away from each other in the CM frame, $\phi + \Phi = \pi$. Equation 3.7 can be rewritten to express the CM target scattering angle in terms of the lab target scattering angle,

$$\theta_2 = \frac{\pi - \phi}{2}. \tag{3.8}$$

Of particular interest in kinematics calculations is the energy of the recoiling target atom as a

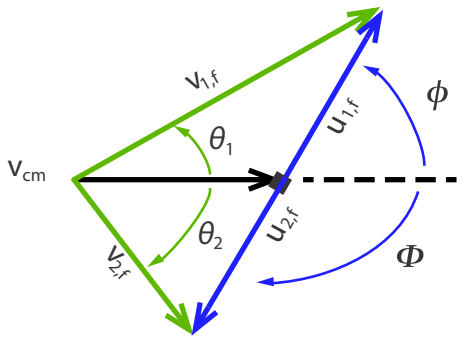

Figure 3.3: Scattering angles schematics for the lab (green) and Center of Mass frame (blue).

function of the target atom scattering angle. By using the law of cosines on the lower triangle in Figure 3.3 and the velocity relationship in Equation 3.5,

$$v_{2,f}^2 = 2v_{cm}^2(1-\cos\phi). \tag{3.9}$$

Equation 3.9 can be further simplified using Equations 3.3 and 3.7:

$$v_{2,f} = 2v_{1,0}\frac{M_c}{M_2}\cos\theta_2. \tag{3.10}$$

The recoil energy can then be calculated through the kinetic energy relationship

$$E_{2,f} = \frac{1}{2}M_2 v_{2,f}^2. \tag{3.11}$$

Combining Equations 3.10 and 3.11, the energy transferred to the recoiling target atom, defined as T, can be calculated as a function of lab recoil angle:

$$T \equiv E_{2,f} = \frac{M_2}{2}\left(2\frac{v_{1,0}M_c\cos\theta_2}{M_2}\right)^2. \tag{3.12}$$

Equation 3.12 can be transformed to give a relationship for T as a function of the ion CM recoil angle by using Equation 3.8:

$$\cos\theta_2 = \cos\left(\frac{\pi-\phi}{2}\right) = \sin\frac{\phi}{2}. \tag{3.13}$$

Combining Equations 3.12 and 3.13, along with the definition of the initial ion energy from Equation 3.1a, calculation of T can be simplified to,

$$T = \gamma E_0 \sin^2\frac{\phi}{2}, \tag{3.14}$$

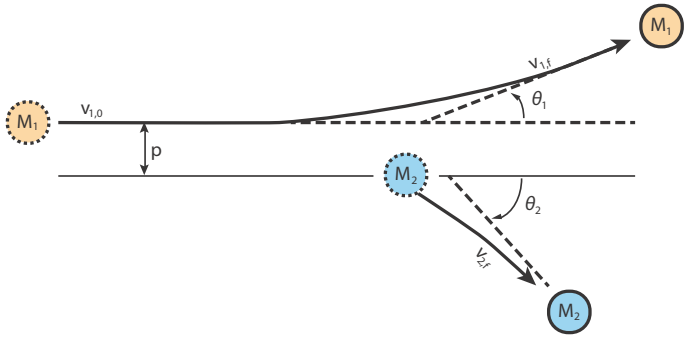

Figure 3.4: Trajectories of two-body collision in the laboratory frame.

where γ is defined as,

$$\gamma \equiv \frac{4M_1M_2}{(M_1+M_2)^2}. \tag{3.15}$$

The product γE_0 can also be defined as T_M, or the maximum energy transfer during a head-on collision.

One final relationship is important for kinematics calculations which relates the scattering angle of the ion in the lab as a function of the scattering angle in the CM frame:

$$\theta_1 = \tan^{-1}\left(\frac{M_2 \sin\phi}{M_1 + M_2\cos\phi}\right). \tag{3.16}$$

The preceding section uses only the conservation of energy and momentum to show that two particles can be represented by a single particle with mass M_c, interacting with a central force $V(r)$, with a constant energy given E_c. This was all done with the assumption that the force acts only along the line joining the two particles, with no transverse forces. In order to determine the kinematics of a collision (angle of scatter, energy of particles, etc.), the relationship $\sin^2\phi/2$ must be calculated. Equation 3.14 can then be used to determine the energy transfer, and Equation 3.16 can be used to determine the scattering angle. The following sections discuss several methods that can be used to calculate $\sin^2\phi/2$ based on a central potential.

3.2.2 Central Force Scattering

While the previous section deals the initial and final positions of the particles far away from the point of collision, knowledge of entire path of the particles is necessary to calculate scattering cross-sections, or probability of recoil energies. A more appropriate sketch of the collision is shown in Figures 3.4 and 3.5, where the dotted lines correspond to the asymptotic trajectories from Figure 3.2. Before the full trajectory can be determined, an advantage of the CM frame must be proved: angular momentum is constant.

Consider a particle with mass M_C traveling with velocity $\mathbf{v}$, at a location described by the po-

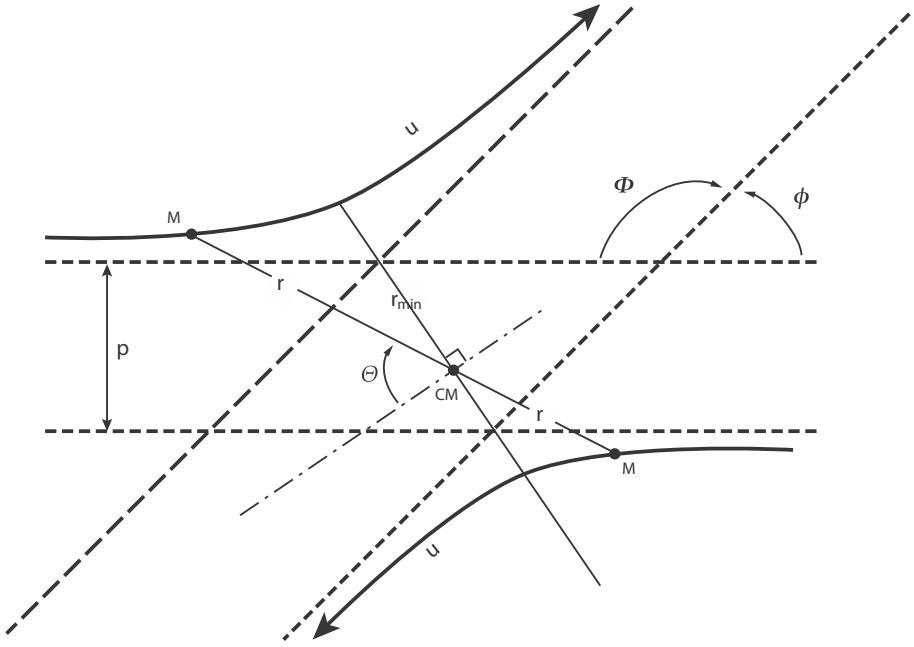

Figure 3.5: Trajectories of two-body collision in the Center of Mass frame.

sition vector $\mathbf{r}$, acted on by some force $F(\mathbf{r})$ that acts only in the radial direction. The angular momentum of this single-body problem can be expressed as,

$$l = \mathbf{r} \times \mathbf{p} = \mathbf{r} \times M_c \mathbf{v}. \tag{3.17}$$

where $\mathbf{p}$ is the momentum of the traveling ion. The time rate of change of the angular momentum is given by,

$$\dot{l} = \dot{\mathbf{r}} \times M_c \mathbf{v} + \mathbf{r} \times M_c \dot{\mathbf{v}}, \tag{3.18}$$

where the dot notation denotes the time derivative. Since $\dot{\mathbf{r}} \equiv \mathbf{v}$, the first term equals zero, while the second term is the torque due to a force,

$$\dot{l} = \mathbf{r} \times \mathbf{F}. \tag{3.19}$$

However, since $\mathbf{F}$ is defined as a central force only, $\mathbf{r} \times \mathbf{F} = 0$, and thus,

$$l = \mathbf{r} \times \mathbf{p} = \text{constant}. \tag{3.20}$$

The angular momentum can be further simplified by breaking the velocity vector into its polar components. Since the radial component points in the same direction as $\mathbf{r}$, the angular momentum can be expressed by the angular velocity only,

$$l = r M_c v_\theta = r^2 M_c \dot{\Theta}, \tag{3.21}$$

where Θ is the angle between r and r_{min}, as displayed in Figure 3.5. The distance r is defined as $r_1 + r_2$, or the sum of the distances from the particles to the center of the problem. Additionally, r_1 and r_2 are defined as,

$$r_1 = \frac{M_1}{M_1 + M_2 +} r, \tag{3.22a}$$

$$r_2 = \frac{M_2}{M_1 + M_2 +} r. \tag{3.22b}$$

Since the angular momentum stays constant throughout the entire collision, definition at any one point can define l. At very long separation distances, the particle travels in a straight line separated from the point of collision by the impact parameter p as in Figure 3.5, thus the angular momentum can be calculated as,

$$l = M_c p v_{1,0}. \tag{3.23}$$

In order to determine the scattering cross-section of the collision, manipulation of the system energy is required. The energy of the CM system can be calculated in polar coordinates as the combination of kinetic and potential energy exerted by $V(r)$:

$$E_c = \frac{1}{2} M_c (\dot{r}^2 + r^2 \dot{\Theta}^2) + V(r). \tag{3.24}$$

By equating Equations 3.21 and 3.23, $\dot{\Theta}$ can be expressed as a function of the impact parameter,

$$\dot{\Theta} = \frac{p v_{1,0}}{r^2}. \tag{3.25}$$

Inserting Equation 3.25 into Equation 3.24, and solving for $\dot{r}$ results in,

$$\dot{r} = v_{1,0} \left[1 - \frac{V(r)}{E_c} - \left(\frac{p}{r} \right)^2 \right]^{1/2}, \tag{3.26}$$

Equation 3.26 expresses the radial equation of motion, and determines the time rate of change of separation between the particle and the central force. However in order to determine the energy imparted to the recoil particle using Equation 3.14, the scattering angle is required; Utilizing the fact that $\dot{r} = (\mathrm{d}r/\mathrm{d}t)$ and $\dot{\Theta}_c = (\mathrm{d}\Theta_c/\mathrm{d}t)$, the left-hand side of Equation 3.26 can be modified by,

$$\frac{\mathrm{d}\Theta_c}{\mathrm{d}r} = \frac{\dot{\Theta}}{\dot{r}} = \frac{p}{r^2 \left[1 - \frac{V(r)}{E_c} - \left(\frac{p}{r} \right)^2 \right]^{1/2}}. \tag{3.27}$$

Equation 3.27 equates the change in angle as a function the distance between the two particles. The final scattering angle, ϕ can be determined by integrating Equation 3.27 over the orbit. Since the trajectory is symmetric about the impact parameter, the second half of the trajectory of the particle is the same as the first half and the integral can be reduced to the first half of orbit, from $r \to \infty$ to

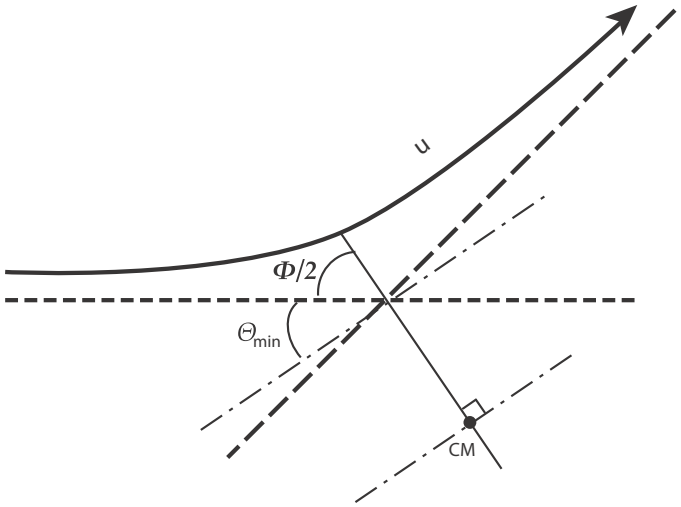

Figure 3.6: Schematic of Θ_{min}.

r_{min}, while the respective Θ limits go from Θ_{min} to $\pi/2$. Utilizing the limits for the first half of the orbit gives:

$$\int_{\Theta_{min}}^{\pi/2} d\Theta = \int_{\infty}^{r_{min}} \frac{p dr}{r^2 \left[1 - \frac{V(r)}{E_c} - \left(\frac{p}{r}\right)^2\right]^{1/2}}. \tag{3.28}$$

While the upper limit is clear from Figure 3.5, the lower limit requires further clarification; From Figure 3.6, $\Theta_{min} + \Phi/2 = \pi/2$ and from Figure 3.3, $\Phi + \phi = \pi$, thus,

$$\Theta_{min} = \frac{\phi}{2}. \tag{3.29}$$

Plugging Equation 3.29 into Equation 3.28, we can finally solve for the scattering angle ϕ,

$$\phi = \pi - 2\int_{r_{min}}^{\infty} \frac{p dr}{r^2 \left[1 - \frac{V(r)}{E_c} - \left(\frac{p}{r}\right)^2\right]^{1/2}}. \tag{3.30}$$

Equation 3.30 is known as the classical scattering integral, and can be used to determine the final scattering angle in the CM frame given some potential energy $V(r)$, energy E_c, and the impact parameter p. The last unknown, r_{min}, can be determined through clever interpretation of the inverse of Equation 3.27:

$$\frac{dr}{d\Theta_c} = \frac{r^2 \left[1 - \frac{V(r)}{E_c} - \left(\frac{p}{r}\right)^2\right]^{1/2}}{p}. \tag{3.31}$$

At the middle of the trajectories where the particles are closest, $r = r_{min}$ and $dr/d\Theta_c = 0$, thus the term in the brackets equals zero:

$$r_{min} = \frac{p}{\left[1 - \frac{V(r_{min})}{E_c}\right]^{1/2}}. \tag{3.32}$$

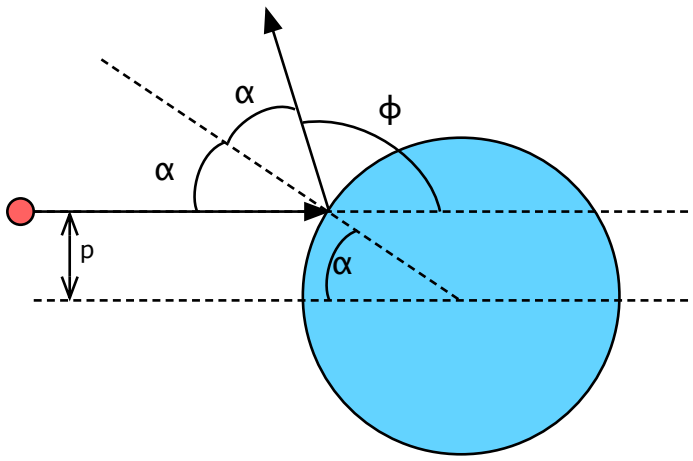

Figure 3.7: Schematic of hard-sphere scattering.

In order to determine the scattering angle in Equation 3.30, a potential energy $V(r)$ must be defined that describes the interaction between the two particles interacting through a central force to describe the binary collision dynamics. While there are many potentials that aim to describe this complicated behavior, only a few exist that can be analytically solved. These simple potentials make broad assumptions, such as the hard-sphere potential, or rely on empirically tuned parameters, such as the MAGIC potential. Several of these are discussed in the following sections.

3.2.3 Hard-Sphere Potential

The simplest model to describe the behavior between two interacting atoms is the Hard-sphere model. The potential energy is very similar to billiard ball collisions, and can be described as,

$$V(r) = \begin{cases} \infty & \text{if } r < R_1 + R_2, \\ 0 & \text{if } r \geq R_1 + R_2. \end{cases} \tag{3.33}$$

Unlike many other potentials, the Hard-sphere potential is easily solved geometrically using Figure 3.7. If the scattering angle is larger than the sum of the radii, $p > R_1 + R_2$, then no interaction will occur. Otherwise, assuming a purely elastic collision and, the scattering angle can be determined as,

$$\phi = \pi - 2\alpha = \pi - 2\sin^{-1}\left(\frac{p}{R_1 + R_2}\right). \tag{3.34}$$

The scattering angle for all cases can be simplified to,

$$\phi = \begin{cases} 2\cos^{-1}\left(\frac{p}{R_1+R_2}\right) & \text{if } p < R_1 + R_1, \\ 0 & \text{otherwise.} \end{cases} \tag{3.35}$$

3.2.4 Rutherford Potential

Along with the Hard-sphere model, the Coulombic or Rutherford potential is one of the few potentials that can be solved analytically. The formulation for the Rutherford potential follows the two-body collision of unscreened ions, or rather the Coulombic collision between two bare nuclei with no accounting of electron interaction. Although simple, Rutherford Scattering will prove to be an appropriate approximation in certain high-energy situations.

The Coulombic potential energy describes the force between two particles with atomic numbers Z_1 and Z_2 as,

$$V(r) = \frac{Z_1 Z_2 e^2}{r}, \tag{3.36}$$

where e is the charge of an electron. Plugging this into Equation 3.30,

$$\phi = \pi - 2p \int_{r_{min}}^{\infty} \frac{\mathrm{d}r}{r^2 \left[1 - \frac{p_0}{r} - \left(\frac{p}{r}\right)^2\right]^{1/2}}, \tag{3.37}$$

where p_0 is defined as,

$$p_0 = \frac{Z_1 Z_2 e^2}{E_c}. \tag{3.38}$$

Before Equation 3.37 can be integrated, the substitution $y = 1/r + p_0/2p^2$ is required:

$$\phi = \pi - 2 \int_{1/r_{min}+p_0/2p^2}^{p_0/2p^2} \frac{\mathrm{d}y}{\left[c^2 - y^2\right]^{1/2}}, \tag{3.39}$$

where c is defined as,

$$c^2 = \frac{1}{p^2} + \frac{p_0^2}{4p^4}. \tag{3.40}$$

In addition, r_{min} must be defined. Plugging Equation 3.36 into Equation 3.32,

$$r_{min} = \frac{p}{(1 - p_0/r_{min})^{1/2}}. \tag{3.41}$$

Rearranging the above equation and using the quadratic formula we can solve for r_{min}:

$$r_{min} = \frac{p_0}{2}\left[1 + \left(1 + \frac{4p^2}{p_0^2}\right)^{1/2}\right]. \tag{3.42}$$

Using this new integration limit, Equation 3.39 can finally be integrated:

$$\phi = \pi - 2\left[\sin^{-1}\left(\frac{p_0/2p^2}{c}\right) - \sin^{-1}\left(\frac{\frac{2}{p_0}\left[1 + \left(1 + \frac{4p^2}{p_0^2}\right)^{1/2}\right]^{-1} + p_0/2p^2}{c}\right)\right]. \tag{3.43}$$

Luckily the first term in the brackets goes to 0 as p_0 gets large or p gets smaller, so that Equation 3.43 reduces to,

$$\phi = \pi - 2\sin^{-1}\left(\frac{\frac{p_0}{2p^2}}{\left[\frac{1}{p^2}+\frac{p_0^2}{4p^4}\right]^{-1/2}}\right). \tag{3.44}$$

Finally, solving for ϕ gives,

$$\sin^2\frac{\phi}{2} = \frac{1}{1+\frac{4p^2}{p_0^2}}. \tag{3.45}$$

This final equation can be plugged directly into Equation 3.14 to solve for the energy imparted to the target atom based on the angle of the scattering ion.

3.2.5 MAGIC Potential

When formulating an interatomic potential energy of two interacting atoms, the assumptions utilized for the hard-sphere and Rutherford type potentials can be bypassed through the creation of a purely empirical potential formulation. This formulation forms the basis of the MAGIC potential. Through the analysis of extensive experimental results, Biersack and Haggmark [45] created a universal model for two-body interatomic potential that is utilized in the TRIM algorithm in the original SRIM/TRIM software [46], 3DTrim [47], and the 3DOT model presented here.

As discussed in Section 3.2.4, the Rutherford potential describes the interatomic potential energy function for two bare atoms. In reality, the atoms are surrounded by electrons that shield the positively charged nucleus. At high enough energies, colliding ions can pierce the outer shell of electrons, resulting in a Coulombic force that becomes a function of distance from the nucleus. Mathematically, this can be expressed through a screening function term $\Phi(R)$ multiplied against the Rutherford potential energy as,

$$V(R) = \frac{Z_1 Z_2 e^2}{aR}\Phi(R), \tag{3.46}$$

where $R \equiv r/a$ is the reduced interatomic separation and a is the universal screening length defined as,

$$a = \frac{0.8853 a_0}{Z_1^{0.23}+Z_2^{0.23}}, \tag{3.47}$$

where a_0 is the Bohr radius $= 5.29\cdot 10^{-11}$ m.

Although Biersack and Haggmark initially utilized the Molière screening function [45], Ziegler, Biersack, and Littmark later improved the TRIM algorithm through creation of an Universal Potential [48]. Through brute force calculation of 261 randomly selected pairs of target-ion element combinations, the so-called "ZBL," or Universal screening potential energy was empirically created:

$$\Phi_U = 0.1818\exp(-3.2r/a) + 0.5099\exp(-0.9423r/a) +$$

$$0.2802\exp(-0.4028r/a)+0.2817\exp(-0.2016r/a) \tag{3.48}$$

Unfortunately, the interatomic potential energy for any given target-ion combination had to be explicitly solved using Equation 3.46. In response to the high expense of specially tailored calculations, Ziegler, Biersack, and Littmark created the MAGIC formula:

$$\cos\frac{\phi}{2}=\frac{P+R_c+\Delta}{R_{min}+R_c}, \tag{3.49}$$

with the following reduced parameters:

$$P=p/a, \tag{3.50a}$$
$$R_c=\rho/a, \tag{3.50b}$$
$$\Delta=\delta/a, \tag{3.50c}$$
$$R_{min}=r_{min}/a. \tag{3.50d}$$

Here a is defined as the universal screening length that is used to reduce the impact parameter p, radii of curvature of the trajectories at the closest approach, ρ, and "correction term", δ [45].

The term ρ is calculated through the potential energy at the closest approach by,

$$\rho=\frac{-2\left[E_c-V(r_{min})\right]}{V'(r_{min})}, \tag{3.51}$$

where V' is the spatial derivative of the interatomic potential energy at point r_{min}.

Lastly, Δ is the so called "Magic Formula Parameter," a correction term that has been empirically fitted to the Universal Screening Potential, and is defined as [45],

$$\Delta=\frac{A\left(R_{min}-P\right)}{G+1}, \tag{3.52a}$$
$$A=2\alpha\varepsilon P^{\beta}, \tag{3.52b}$$
$$G=\frac{\gamma}{\sqrt{1+A^2}-A}, \tag{3.52c}$$
$$\alpha=1+C_1\varepsilon^{-1/2}, \tag{3.52d}$$
$$\beta=\frac{C_2+\varepsilon^{1/2}}{C_3+\varepsilon^{1/2}}, \tag{3.52e}$$
$$\gamma=\frac{C_4+\varepsilon}{C_5+\varepsilon}, \tag{3.52f}$$
$$\varepsilon\equiv\frac{aE_c}{Z_1Z_2e^2}. \tag{3.52g}$$

Here, the C_i terms are coefficients that are determined using a particular interatomic screening potential energy. Using Equation 3.46, the coefficients in the MAGIC formulation can be computed

as:

$$C_1 = 0.99229, \tag{3.53a}$$
$$C_2 = 0.11615, \tag{3.53b}$$
$$C_3 = 0.007122, \tag{3.53c}$$
$$C_4 = 9.3066, \tag{3.53d}$$
$$C_5 = 14.813, \tag{3.53e}$$

Although somewhat tedious, each term described above is analytically solvable, except for the distance of closest approach; r_{min} can be solved using a simple Newton method loop applied to Equation 3.32, similar to the form in Section 4.3.2. Once r_{min} is determined, Equation 3.49 can be calculated. Lastly, the Pythagorean identity,

$$\sin(\phi/2) = \sqrt{1 - \cos^2(\phi/2)}, \tag{3.54}$$

can be utilized in order to use the form required in Equation 3.14, allowing calculation of the energy imparted to the recoiling target atom.

3.2.6 Electronic Losses

Following the original TRIM assumptions, the energy losses due to nuclear-nuclear collisions are assumed to be separable from the energy losses due to target material's electrons interaction with the ion [46]. In general, the electronic loss due to the traveling ion is calculated by,

$$\Delta E_e = LNS_e(E), \tag{3.55}$$

where L is the path length between nuclear collisions, N is the atom density of the target, and S_e is the electronic stopping cross section calculated using the Brandt-Kitagawa theory [46, 49].

3.2.7 Free Flight Path

Within the TRIM algorithm, the computational expense is decreased by ignoring collisions that result in a low energy transfer, $T < T_{min}$, or small scatter angle, $\theta < \theta_{min}$. This effectively ignores many of the glancing angle collisions that have little effect on the end result.

In general, the smaller the the impact parameter, the larger the amount of energy is transferred in the collision, thus there exists a maximum impact parameter p_{max} above which a negligible amount of energy is transferred, or rather $T < T_{min}$. The volume of influence created by a passing ion in which any target atom present in the volume will cause a scatter in which $T > T_{min}$ is simply $\pi p_{max}^2 L$,

where L is the length of travel. On average, the volume per atom is just the inverse of the atomic density, or N^{-1}, thus given a particular p_{max}, L can be calculated by,

$$\pi p_{max}^2 L = N^{-1}. \tag{3.56}$$

The choice of p_{max} depends on the particular potential to be used, and is input as a parameter at run time.

3.3 Methods

The BCA code 3DOT [50] was used for all calculations in the current work. 3DOT is based on Schwen's 3DTrim [51] which utilizes the previously published TRIM algorithm [46] to track ion cascades in non-Cartesian geometry. The TRIM algorithm relies on the ZBL universal potential to model the interactions between moving ions and stationary samples, and calculates the electronic stopping power by scaling proton stopping powers using the Brandt-Kitagawa theory [46, 49]. Due to the nature of TRIM calculations, each ion interacts with an undamaged, 0 K amorphous target.

Each 3DOT simulation model consisted of a fuel cube with an implanted sphere of xenon. Owing to the small computation expense of BCA and the wide spread of bubble sizes present in uranium carbide [9], the simulated bubble radii ranged from 0.5 nm to 500 nm with periodic boundary conditions enforced on all box sides. As the current study was focused on intra-granular bubbles, grain boundary effects were not considered and the theoretical density of 13.63 g/cm^3 was assumed for uranium carbide. Fission fragments were randomly created based on the probability distribution functions of typical fission fragment mass and energies.

For these calculations, a fission gas atom was considered implanted if it exited the bubble surface with an implantation energy greater than E_{min}. The value of E_{min} = 300 eV first utilized by Nelson has been implemented in many re-solution studies [15, 36, 41]. More recently, a study aimed at determining the implantation energy in UO_2 fuels determined that while re-solution did occur for atoms with implantation energies less than the 300 eV, the original implantation energy of 300 eV resulted in a 85% probability of re-solution [52]. In light of this, the value of E_{min} = 300 eV will be utilized unless otherwise specified, although model sensitivity to this value will be explored.

The total re-solution rate at which atoms are knocked back into the fuel can be calculated by,

$$R\left[\text{atom}/(\text{s}\cdot\text{m}^3)\right] = \int \dot{F} b(r) m(r) \rho(r) dr, \tag{3.57}$$

where r is the bubble radius, $\dot{F}$ is the fission rate, b is the re-solution parameter, m is the number of atoms in a bubble, and ρ is the bubble concentration distribution function,

$$N\left[\text{bub}/\text{m}^3\right] = \int \rho(r) dr. \tag{3.58}$$

Here, N is the total concentration of bubbles in the sample.

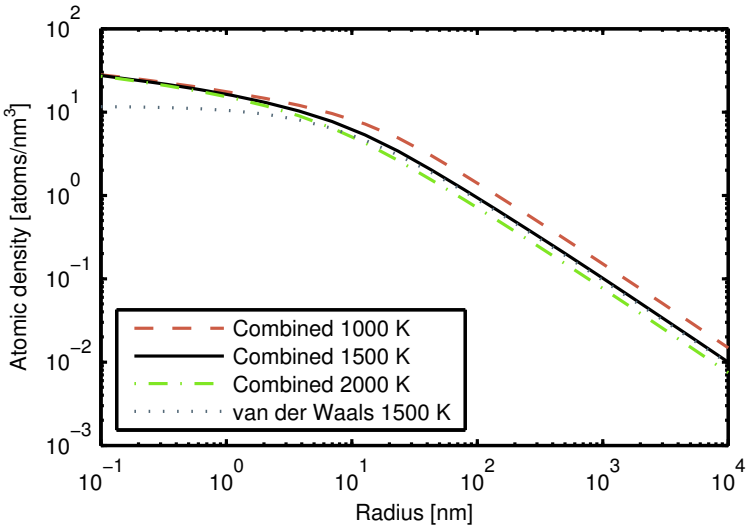

Figure 3.8: Atomic densities calculated from different EOS models.

In stress free solids, the number of atoms in a bubble m is generally calculated using the van der Waals equation of state (EOS):

$$m = \frac{4}{3}\pi r^3 (B + kTr/2\gamma)^{-1}, \tag{3.59}$$

where γ = 1 N/m is the surface energy of uranium carbide [53], k is Boltzmann's constant, T is temperature, and $B = 8.5{\cdot}10^{-29}$ m^3/atom [17]. However, the high temperature and stress involved with very small bubbles in nuclear fuel results in an underestimation of atomic density, thus the EOS values calculated by Ronchi were utilized at small bubble radii [20]. Although more recent evaluations show that the atomic density may actually be higher in very small bubbles [21], a smooth transition from Ronchi's data for small bubble sizes to values calculated using Equation 3.59 is utilized to calculate the atom density in the bubble due to the large spread of bubble radii, giving the "Combined" model types, as displayed in Figure 3.8.

Considering the many combinations of bubble radii and box sizes, any dependence on simulation volume must be removed. Due to the periodic boundary conditions, the approximately 6 μm fission fragment path will wrap around the model many times. Due to the relatively low density within the bubble, as the porosity increases, the total fission fragment path will also increase. In order to avoid a bubble concentration dependent path length, comparative studies of various bubble and box sizes showed that the artificial lengthening of the fission fragment path was minimized when the cube length was at least 10 times the bubble radius. In this way, a value for b can be calculated independent of the bubble concentration.

3.3.1 Applicability of BCA

The ability of 3DOT to quickly analyze ion trajectories lies in the basic BCA simplification in which each collision is modeled as a single two-body collision. While this approximation is physically appropriate for high ion energies (typically above 1 keV), multi-body collisions tend to occur at lower ion energies. Several publications have attempted to explicitly define the limits of BCA with limited success [54–57].

The error introduced by BCA must be quantified in order to justify its use within this study. Fortunately, the use of a threshold implantation energy allows all ions below E_{min} to be ignored. Above this limit, the error introduced by ignoring multi-body collisions can be estimated by calculating the maximum deviation that can occur from the next nearest-neighbor atom. If the angular and energy difference from the otherwise ignored atom has minimal effect on the re-solution rate, then the use of BCA is appropriate in this study for ions down to E_{min}.

In 3DOT, the value $\sin^2(\phi/2)$ for use in determining the energy transfer with Equation 3.14 is calculated using the ZBL potential with the ion and target atomic attributes, and the impact parameter p. As with all potentials, the smaller the impact parameter, the larger the angular deflection. Using Equation 3.14, large angle deflections correspond to high energy transfer. The greatest effect a second body can have on an ion, and thus the smallest impact parameter, is exactly between two atoms. The deviation caused by a target atom at this minimum impact parameter will estimate the angular deviation that is ignored by the BCA approximation in the worst case scenario.

In general, the fission gas escape process can be separated into two events. First, a fission gas atom must be hit by either a fission fragment or by a fuel atom that has been knocked out of its lattice site by a fission fragment. Due to the random nature of fission fragment creation in 3DOT, any error introduced to the fission fragment scatter angle can be ignored as long as all fission fragments are treated similarly. The same is also true for lattice atom (uranium and carbon) collisions. However, since the energy of knocked fission gases needs to be explicitly tracked, fission fragment and lattice ion energies must be correctly handled for all ion energies above E_{min}. Uranium carbide has a simple cubic NaCl crystal structure with a lattice parameter of roughly 0.5 nm, resulting in $p_{min} = 0.125$ nm as an ion travels between a uranium and carbon atom (Figure 3.9a). Due to the γ factor included in the energy transfer equation (Equation 3.14), uranium-uranium collisions result in the highest energy transfer. Using the minimum impact parameter, the maximum energy transferred in a uranium-uranium collision is 60 eV for all initial ion energies. This value quickly diminishes as the impact parameter increases, reducing to less than 3 eV when the impact parameter is 0.2 nm. In the worst possible case, a lattice atom with energy 300 eV will overestimate the ion energy after a collision by 20%. However, the probability of such an event is small enough that BCA accurately represents fission fragment and lattice collisions for the purposes of this work.

The second event that must occur to create an implanted atom is the transport of a fission gas atom from the interior to the surface of the bubble. As opposed to the fuel lattice, both fission gas ion energy transfer and angular deflection are both important to track, as a knocked fission gas atom

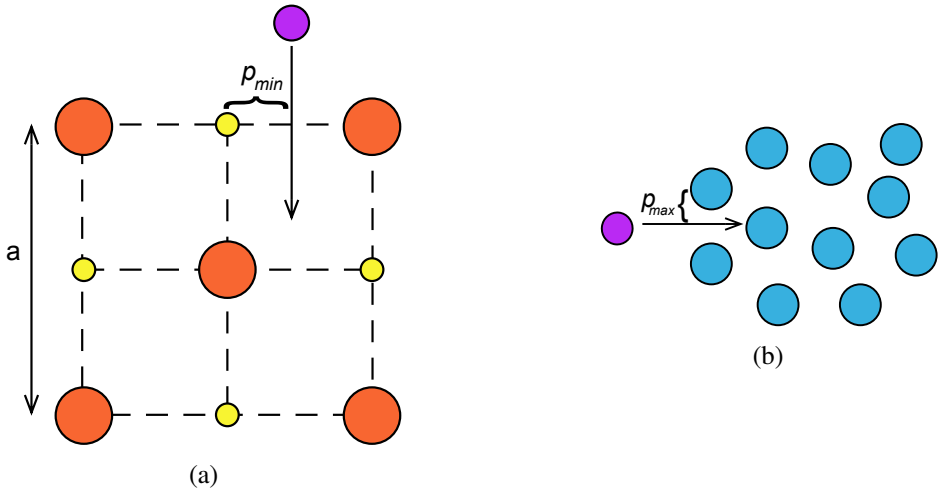

Figure 3.9: Schematics for the calculation of p_{min} a) in the fuel lattice, and b) in the bubble.

can receive a large-angle deflection very near the bubble surface. In the amorphous bubble region, the distance between fission gas atoms in the bubble d can be approximated by,

$$d = 2\,p_{min} = 2\,(3/4\pi m)^{1/3}. \tag{3.60}$$

Using the Combined EOS described previously, the density in a 1 nm bubble at 2000 K is 15.3 atoms/nm^3. Since the minimum impact parameter occurs directly between two atoms, $p_{min} = 0.25$ nm (Figure 3.9b). At this large of an impact parameter, the direction of the ion in a representative Xe-Xe collision deviates by only 2° with an initial ion energy of 300 eV. The ion energy must be reduced below 10 eV to result in a deviation of greater than 30°. In addition, such small deflections result in negligible energy transfer. Since these deviations occur during the most conservative atomic densities and smallest impact parameters, reduction of collisions within the bubble to single collisions was deemed appropriate for this work.

3.4 Results

Figure 3.10 shows the re-solution parameter b as a function of radius r for different EOS models, where b represents the number of escaped atoms per bubble atoms for a single fission and single bubble. A quick decrease initially occurred, reducing b by nearly 60% as the bubble radius increases from 1 to 50 nm, beyond which the re-solution parameter became nearly constant. Since b was calculated on a per bubble atom basis, it represents the average probability of any given bubble atom to be implanted. As the bubble radius increases, interior atoms have farther to travel before becoming implanted, resulting in the quick decrease in b as a function of radius. However, as r becomes larger than 10 nm, the atomic density exponentially decreases as a function of radius (Figure 3.8), boosting the probability of a struck interior atom reaching the surface and flattening b

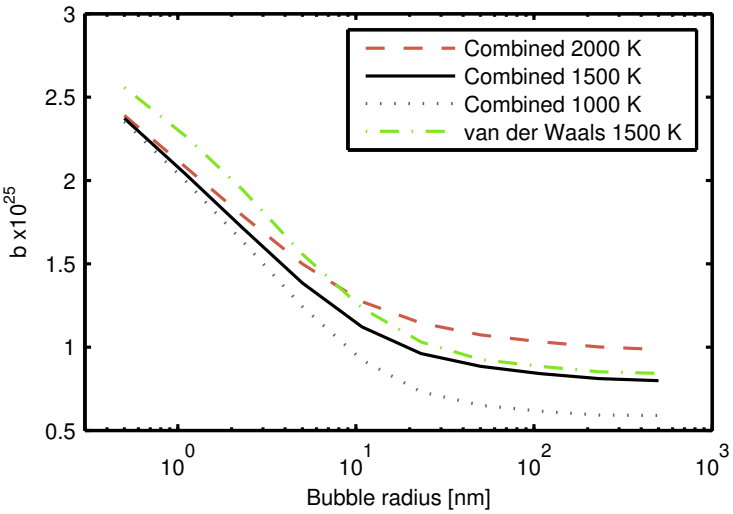

Figure 3.10: Calculated b for several atomic density calculation methods.

as a function of radius. If the bubble sizes increase past the radii of interest studied here, the low atomic density will eventually serve to create a positive slope in the re-solution parameter at very large radii.

Figure 3.10 also includes the re-solution parameter using different EOS methods: Combined at 1000 K, 1500 K, and 2000 K, and van der Waals at 1500 K. In general, b was inversely proportional to the atomic density, producing fewer implanted ions at higher atomic densities, however the re-solution parameter was not very sensitive to the particular EOS model.

As discussed previously, there is some uncertainty on the value of E_{min}. Figure 3.11 displays b for various values of E_{min} at 1500 K. Previous studies have shown that E_{min} may be less than 300 eV [52]. By reducing the assumed E_{min} to 150 eV, the re-solution parameter only increases by 30%, without much variation in the shape of the curve.

A gas atom can be knocked out of the bubble after first being struck by either a fission fragment or indirectly through a fuel cascade. Figure 3.12 displays the fraction of implanted gas atoms that were a result of either a fuel cascade or solely through a direct fission fragment hit in a 5 nm bubble at 1500 K. Roughly 90% of all implanted ions were a result of a cascade started within the fuel, showing the overall importance of fuel cascades over direct fission fragment collisions.

As ions propagate through the material, they produce additional "daughter" ions by knocking atoms from their lattice site. Figure 3.13 displays the probability of each type of "parent" as a function of bubble radius. For nearly all bubble radii, the direct fission fragment contribution remained near 10% due to the low cross-section of small bubbles and low interaction probabilities for low density, large radii bubbles. Due to their small mass and γ factor, the relative importance of carbon atoms was insignificant; for very small bubble sizes, the majority of all collisions were a result of uranium atoms directly knocking out fission gas atoms. The increase in fission gas parents indicates that as the bubble size enlarged, a knock-on effect occurred, resulting in implanted atoms different

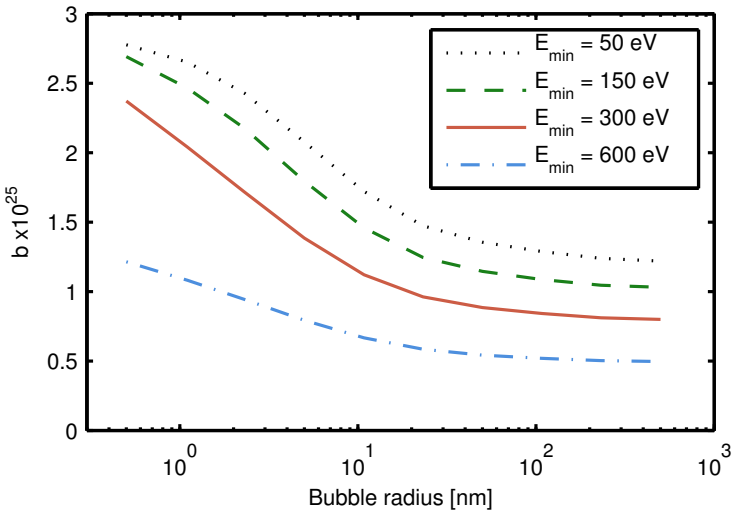

Figure 3.11: Calculated b for various E_{min} thresholds.

than the original struck gas atoms.

As shown in Figure 3.12, the importance of fission fragment collisions were relatively minor compared with the importance of fuel cascades. However, it can be expected that the highly energetic fission fragments create daughter ions with relatively high energies, thus resulting in more energetic and more deeply implanted atoms. This phenomenon can be shown by plotting the re-solution parameter in a 1500 K 5 nm bubble as a function of E_{min}, (Figure 3.14). For low E_{min} energies, fuel cascades have a much higher importance in the total re-solution parameter than fission fragments, similar to what was displayed in Figure 3.13. However at a crossover energy of about 40 keV, fission fragment interactions produced more implanted atoms than fuel cascades.

3.5 Discussion and Conclusions

The study of the re-solution parameter in this Chapter extends the understanding, methods, and calculation of the re-solution parameter well beyond previous studies. However, when comparing the current work to the historical calculations, two results seem to conflict. However, upon further inspection, the discrepancies further support the current work. The first is that for a bubble radius of 5 nm, the re-solution parameter was determined to be $1.4 \cdot 10^{-25}$ atoms/(fsn·m^3), an order of magnitude less than the previously calculated value of $b = 3 \cdot 10^{-24}$ atoms/(fsn·m^3) [15, 36]. In addition, the previous re-solution parameter vs. E_{min} results showed the cascade/fission fragment importance cross-over occurring at 5 keV, while the calculations in Section 3.4 show a cross-over at 50 keV. The reasons for these differences are rooted in differences between the models used in each study and warrant discussion.

By tallying energy losses in a 3DOT simulation, we found the amount of energy dissipated due to electronic losses accounted for about 90% of the original fission fragment energy. In light of this,

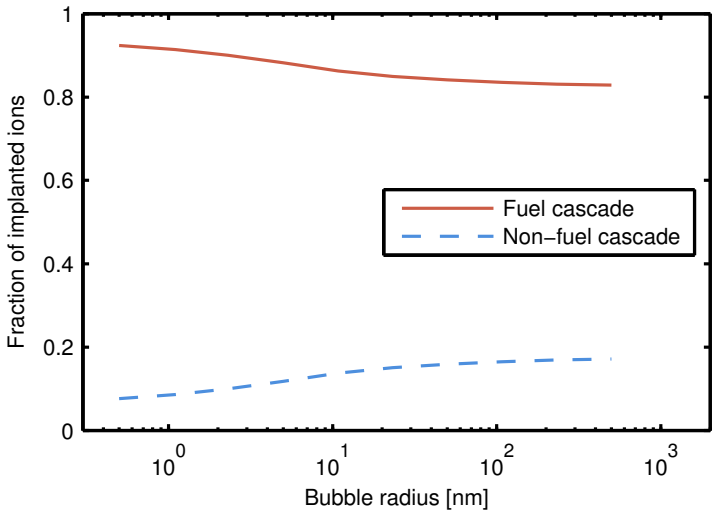

Figure 3.12: Probability of a direct fission fragment implantation or fuel cascade implantation.

it can be expected that by ignoring these losses, the fission fragment range, and thus b, will increase by an order of magnitude. This was verified by running 3DOT simulations without accounting for electronic losses in which we calculated the re-solution parameter for a 5 nm bubble at 1500 K to be $b = 4.8 \cdot 10^{-24}$ atoms/(fsn·m^3), within 50% of the previously calculated value [15].

One of the largest improvements over the previous studies is the use of the ZBL potential for all atoms and energies, as opposed to the Rutherford potential for fission fragment collisions, and the hard-sphere potential for all other collisions. The Rutherford potential energy is an adequate representation of high energy ion collisions, and in fact is utilized in 3DOT at high ion energies [46]. Conversely, the hard-sphere approximation is adequate for very low ion energies. However, at intermediate energies, neither approximation is appropriate [58].

The hard-sphere potential calculates the scattering angle for use in Equation 3.12 from the geometry of the problem, thus each scatter is independent of the ion's energy. This results in a much higher probability of high angle collisions and ultimately an underestimation of an ion's range. When applied to lattice atoms, the hard-sphere potential ensures that an entering ion will only penetrate a short distance into the bubble. Likewise, gas atoms far from the bubble surface have a low probability of re-solution due to an enhanced probability of large-angle scatter events. With the more physical ZBL potential, the scattering cross-section is inversely proportional to the ion's energy, thus uranium atoms are allowed to penetrate to more realistic depths. In addition, all atoms in the bubble become "available" for re-solution, albeit with a frequency less likely for atoms near the center of the bubble. Since the ZBL potential is well approximated by the Rutherford potential, similar behavior is not reproduced in the highly energetic fission fragments that were originally treated with a purely Coulombic potential. All of this resulted in a higher probability of the less energetic lattice cascades, shifting the fuel cascade contribution to b to higher E_{min} values in Figure 3.14, and increasing the cross-over point to 50 keV.

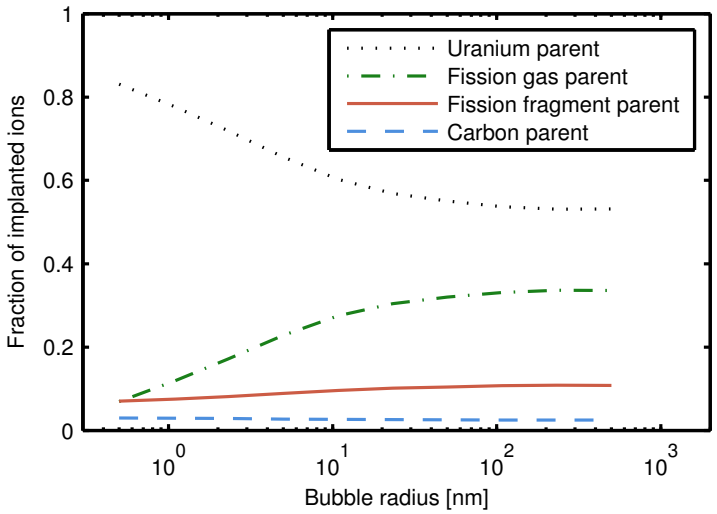

Figure 3.13: Parent fractions as a function of bubble radius.

One final note about the previous studies must be addressed; one of the most interesting results from Ronchi and Elton was a positive slope in the re-solution parameter at high radii and high E_{min} [15]. It should be noted that the value plotted is actually the combination of the product of the re-solution parameter and critical distance, thus the positive slope is heightened by an increasing critical distance as a function of radius. Although similar phenomena can be achieved with the results from 3DOT, they were not nearly as drastic without the inclusion of the critical distance.

In conclusion, due to the relatively high energy transfer material properties in uranium carbide, fission gas re-solution can be modeled using the homogeneous, atom-by-atom loss model. We found a re-solution parameter that was an order of magnitude lower than previous studies due to the inclusion of electronic ion energy losses in this study. A decrease in the re-solution parameter as a function of radius occurred for low radii, with a nearly constant re-solution parameter for radii above 50 nm. BCA allowed the computationally cheap analysis of many different types of bubbles, and it was shown that the fundamental BCA assumption is appropriate for use in the simulations presented here. Through comparative studies on the re-solution parameter for various values of implantation energy and atomic density in the bubble, we found that while the re-solution parameter did change, the overall shape did not. The re-solution parameter calculated here can now be implemented in fission gas bubble calculations to determine how the bubble size relationship impacts the concentration distribution.

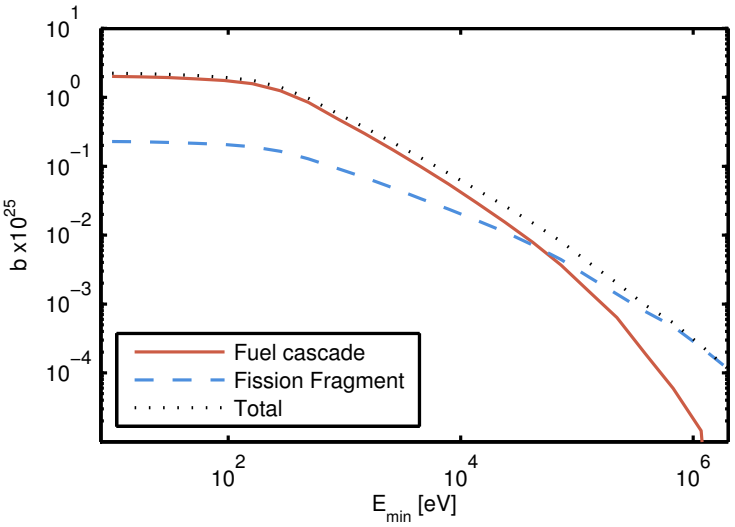

Figure 3.14: Re-solution in a 5 nm bubble due to fission fragment and fuel cascades as a function of E_{min}.

CHAPTER 4: BUBBLE AND CAVITY KINETICS

In an effort to break away from the experimentally derived models typically used for fission gas modeling in uranium carbide, a new type of code that can be developed modularly was developed. By utilizing the modularity of the MOOSE framework [59], the code Bubble and Cavity Kinetics (BUCK) was created in an effort to create a first-principles informed code. In order to build a bare-bones foundation, the simplistic yet historically prevalent models used to simulate fission gas bubble nucleation, growth, and knock-out can be implemented as stepping stones until more advanced models for each physical process are created. In this way, the fission gas bubble concentration distribution can be studied, ultimately leading to simulations of fuel swelling and fission gas release.

4.1 Background

Many past studies have focused on the bubble behavior in nuclear fuel, but as a result of the overwhelming complexity of fission gas bubble modeling, each simulation must make broad simplifications in order to achieve solutions. Many of these simulations assume that the bubble concentration distribution can be distilled into a single average bubble size [13, 23, 60–62]. These models perform adequately, especially for UO_2 studies, but rely heavily on empirically tuned parameters [22]. The inherent use of a single average bubble size precludes the applicability of these models to uranium carbide fuels in which small and large bubbles exist within the grain.

A second type of bubble population model was explored in the 1970's and relied on grouping schemes to reduce the number of bubble groups from millions of bubble sizes to dozens [63–67]. While the grouping scheme was able to reduce the computational cost of the simulation, verification of the method through the comparison to a non-grouped method at the bubble sizes of interest is absent; in both the papers that contain the grouping, as well as initial studies completed within BUCK, no grouping method solution was able to adequately match any sort of non-grouped method solution. Subsequently, a full account of each bubble type is necessary to capture the bubble concentration distribution. However, as a result of the high computational cost of a bubble model that explicitly tracks every bubble size, few models have contained bubble phenomena beyond simple nucleation studies [68]. Furthermore, no study has been able to capture the large bubble behavior that occurs at high temperature and burnup [22].

In an effort to build towards a better understanding of fission gas behavior, a step-by-step approach towards full fission gas bubble modeling is required. Models that describe bubble nucleation, growth, knock-out, and gas diffusivity are necessary, and included in the appropriate sections below.

4.2 Theory

Within BUCK, each bubble size is explicitly tracked, starting from single atom "bubbles" to a simulation dependent upper limit of bubbles with N atoms. For the current work, three primary reaction rate phenomena are considered: nucleation, growth, and re-solution. In general, the reaction rates of the bubbles are concentration driven and can be written as [17],

$$\dot{R} = (k_{ab} + k_{ba})C_a C_b, \tag{4.1}$$

where $\dot{R}$ is the reaction rate density, k_{ab} is the reaction rate constant assuming species b is immobile, k_{ba} is the reaction rate constant assuming species a is immobile, and C_a, C_b are the concentrations of the species a and b respectively. Here, the reaction rates have units of volume/seconds.

In general, there exist two types of reactions to consider for fission gas bubble modeling: rate-limited reactions and diffusion-limited reactions. In the case of single gas atoms diffusing through a medium with a relatively constant concentration, the interaction between atoms is limited by the random-walk diffusive process of the atoms moving through the bulk. If one of the sinks is much larger or has a very strong sink strength (i.e. bubbles), it is possible for a region around the sink to become starved of reactants. In this case, the reaction is limited to the ability of single gas atoms to reach the edge of the capture volume, resulting in diffusion-limited kinetics.

Between the two regimes, there exists a transition zone in which the reaction progresses as a mixture between reaction and diffusion controlled. It can be assumed that larger bubbles will exhibit spherical shapes as the pressure of gas atoms will push evenly at the surrounding lattice. However, for small clusters, it cannot be expected that the shape is truly spherical. Regardless, it is typically assumed that a hard transition occurs after gas atoms combine to form dimers [67], although some studies have assumed a transition at 5 atom clusters, using rate constants whose derivation is apparently absent [17, 68]. As discussed below, the diffusivity of bubbles is much smaller than single gas atoms, such that bubbles can be assumed to be stationary. Furthermore, at such small cluster sizes, the effective size of a dimer is twice the size of a single gas atom, supporting the assumption of rate-controlled kinetics for bubble growth. As such, the nucleation process for the formation of dimers will be modeled using rate-controlled kinetics, while absorption of single gas atoms by any size of bubble will be treated with diffusion-controlled kinetics.

In addition to the reaction rates, gas atom birth and diffusion are also important to capture the interaction of fission gas bubbles. Each physical phenomenon is discussed below.

4.2.1 Birth

The probability of a gas atom to be produced as a result of fission is about 25%, as defined in Section 1.2. Since the vast majority of gas atoms produced are xenon, it will be assumed that all

fission gas atoms behave as xenon. The corresponding rate equation for fission gas creation is,

$$\frac{\partial C_1(t)}{\partial t} = \gamma \dot{F}, \tag{4.2}$$

where C_1 is the concentration of single gas atoms, γ is the gas atom yield, and $\dot{F}$ is the fission rate density.

4.2.2 Diffusion

Diffusivity of gases in uranium carbide has been a phenomena that has received much attention in the past. Matzke's compilation of calculated diffusivities from several different studies shows a scatter of over 3 orders of magnitude [5]. This is primarily due to the poor characterization of the fuel at the time, although reproducibility also suffered from inconsistent experimental conditions such as the presence of oxide layers, unaccounted chemical reactions, and imperfect vacuum systems. Furthermore, fission dose and the corresponding presence of fission gas bubbles were rarely accounted for. Ultimately, the diffusivity of gas atoms through a pure lattice is one of the largest uncertainties involved with fission gas bubble modeling in all types of fuels [17]. Fortunately, since we are primarily concerned with the formation of fission gas bubbles, conservative models of diffusivity can be utilized.

The diffusivity of gas atoms in the solid is primarily a thermal process. In addition to thermal diffusion, the fission environment in nuclear fuel is known to boost the diffusivity of species as a function of the fission rate [17]. In general, the diffusivity of a gas through UC can be expressed by,

$$D_g = D_0 \exp\left(\frac{Q}{RT}\right) + D_i \dot{F}, \tag{4.3}$$

where D_0 is the maximum diffusion coefficient at infinite temperature, Q is the activation energy, R is Ideal Gas constant, and T is temperature, D_i is the irradiation enhanced diffusion coefficient, and $\dot{F}$ is the fission rate density. Table 4.1 displays parameters from previous simulations, while Figure 4.1 displays an Arrhenius plot of diffusivities. The dotted red line represents the model presented by Ronchi with inclusion of fission gas dependent diffusivity. At low temperatures, the fission rate dependent diffusivity dominates, while at high temperatures it can be neglected.

Table 4.1: Diffusion coefficients utilized in previous simulations.

Reference		D_0 [cm^2/s]	Q [kJ/mol]	D_i [cm^5/fission]
Matzke [5]		$3.00 \cdot 10^{-1}$	355	
Ronchi [62]	$T > 1100$ K	$4.60 \cdot 10^{-1}$	328	$1.3 \cdot 10^{-29}$
Ronchi [62]	$T < 1100$ K	$1.50 \cdot 10^{-5}$	231	$1.3 \cdot 10^{-29}$
Madrid [69]		$4.60 \cdot 10^{-3}$	326	$2 \cdot 10^{-30}$
Eyre [68]		$1.66 \cdot 10^{-9}$	221	

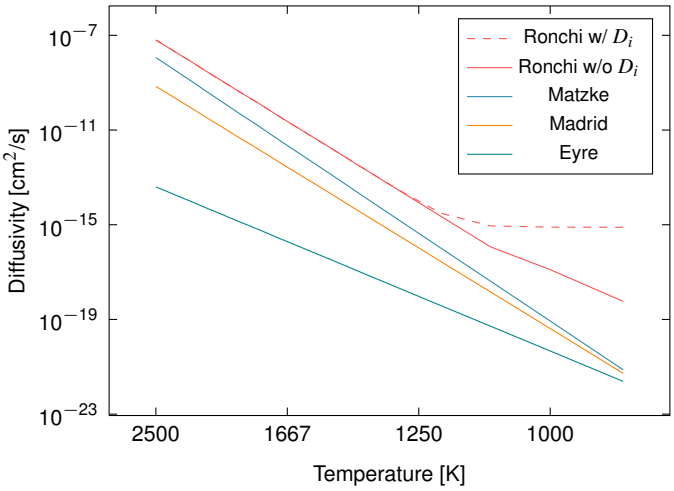

Figure 4.1: Arrhenius plot of Diffusivities.

Except for the values used by Eyre in his 1968 simulations, the diffusivities are relatively similar. In general, higher diffusivity values will result in a larger absorption rate, and thus larger bubble sizes. Since the current work is primarily concerned with the newly calculated re-solution parameter resulting in larger bubbles, the most conservative choice of diffusivity would be the largest value, thus Ronchi's value will be utilized.

Lastly, the mobility of bubbles must be addressed. Although previous studies have utilized terms that account for bubble mobility [67], Blank claims that only for temperatures above 2/3 of the melting temperature, or roughly 1850 K, does mobility impact bubble behavior. Previous studies have shown that bubble diffusivities are more than 10 orders of magnitude lower in UC than single gas atoms [70]. In light of these claims, BUCK assumes that the bubbles remain stationary, and single gas atoms are the only mobile species.

4.2.3 Nucleation

Bubble nucleation in nuclear fuel is generally assumed experimentally to be a heterogenous process, either through formation of nuclei created in the wake of fission fragments [35] or at collection points such as dislocation tangles or precipitates [9]. However, many past calculations incorporate homogenous nucleation models due to their simplicity [17, 67, 68]. Due to the focus of the current work on the re-solution rate on large bubbles, dimers are assumed to be nucleated homogeneously, providing a much more compact and simple formulation for implementation in BUCK similar to the previous studies.

The formation process for homogeneous nucleation of a dimer occurs once two single gas atoms occupy neighboring lattice positions of the form,

$$1 \text{ atom} + 1 \text{ atom} \rightarrow 1 \text{ dimer}, \tag{4.4}$$

If it is assumed that the dimer formation process is irreversible (at least in terms of thermal dissolution, not necessarily re-solution), then the rate of dimer formation can be written as:

$$\dot{R}_n = P_{11} C_1, \tag{4.5}$$

where $\dot{R}_n$ is the reaction rate density for nucleation and P_{11} is the probability per second a gas atom jumps into a nearest neighbor site to a C_1 gas atom.

The probability P_{11} depends on the crystal structure of the material at hand, thus an understanding of the UC system and how gas atom defects are incorporated is necessary; uranium monocarbide maintains a NaCl lattice structure composed of two inter-laced face centered cubic (FCC) sub-lattices of uranium and carbon (Figure 4.2). Density Functional Theory calculations for uranium monocarbide have shown that the incorporation energies of noble gases are positive in all possible sites, strengthening the assumption that thermal dissolution of dimers does not occur [71, 72]. Furthermore, the most stable of locations for noble gas inclusions is at the uranium vacancy site and is generally associated with an adjacent carbon vacancy. From this, it can be assumed that the xenon atom resides in only the uranium sites in the NaCl crystal structure. If gas atoms diffuse through the lattice by jumping between uranium vacancy sites, the NaCl crystal structure can be reduced to a single FCC sub-lattice when modeling diffusive behavior. This simplification can be visualized using Figure 4.2; assuming gas atoms only occupy uranium vacancies (green atoms), then they can only diffuse through the green atom sub-lattice. Ignoring the unused carbon (purple) sites results in a FCC structure.

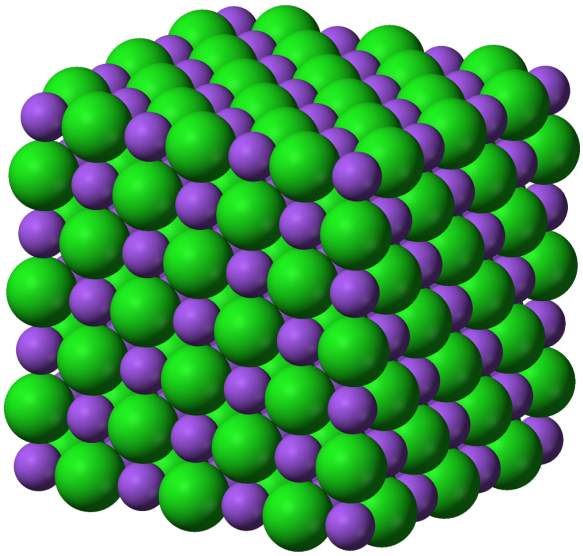

Figure 4.2: Example of NaCl lattice structure.

In order to estimate P_{11}, Figure 4.3 is utilized to break down the separate events by focusing on dimer formation at a particular gas atom (green site) in a FCC lattice. The number of nearest neighbor sites to the gas atom (pink sites) is $n_{fcc} = 12$, a value that is constant for any lattice site

in a FCC structure. If any of these sites contain a second gas atom, then a dimer is formed. The probability P_{11} can thus be broken down into,

$$P_{11} = n_{fcc} P_x, \tag{4.6}$$

where P_x is the probability per second that another gas atom jumps into one of the nearest neighbor positions.

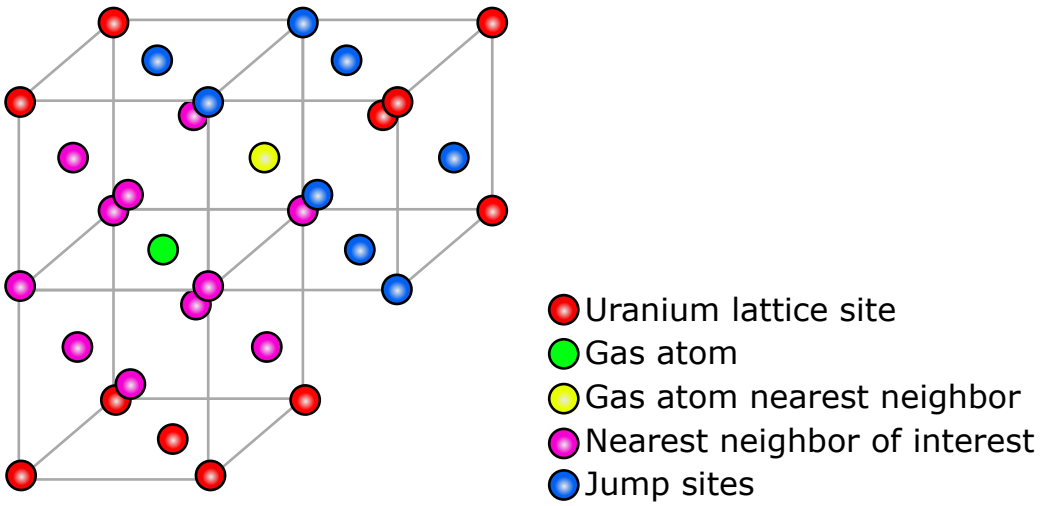

Figure 4.3: Diagram to determine dimer formation in a FCC lattice.

Since the behavior for each nearest neighbor site is equivalent, P_x calculated at one site is applicable to all other sites. Focusing on the specified yellow site in Figure 4.3, the probability P_x depends on the number of sites surrounding the yellow site, η_{fcc}, the probability that one of the adjacent sites is occupied by a gas atom, x_g, and the jump frequency of the gas atom in a particular direction, w, as described by,

$$P_x = \eta_{fcc} x_g w. \tag{4.7}$$

The number of surrounding sites to the yellow atom total $\eta_{fcc} = 7$, as designated by the blue sites in Figure 4.3. Although the number of nearest neighbors in a FCC crystal is 12, the original gas atom itself nor the sites that are already nearest neighbors are not counted since they would result in a dimer formation instantaneously.

The probability that any one of the seven sites contains a gas atom is assumed to be equal to the probability that any given site in the entire lattice contains a gas atom, defined as the gas site fraction, x_g. The site fraction can be written in terms of the concentration as,

$$x_g = C_1 \Omega, \tag{4.8}$$

where Ω is the lattice site volume, or more clearly, $1/\Omega$ is the number of lattice sites per unit volume.

Lastly, the jump frequency can be related to the gas diffusion coefficient, D_g, by [17],

$$D_g = a_0^2 w, \tag{4.9}$$

where a_0 is the lattice parameter.

Combining Equation 4.6 through 4.9 into Equation 4.5 results in a nucleation rate given by,

$$\dot{R}_n = \frac{84\Omega D_g C_1^2}{a_0^2}. \tag{4.10}$$

Since the above rate is essentially the reaction rate between a stationary gas atom and the concentration of diffusing gas atoms, it must be multiplied by 2 following Equation 4.1. Comparing Equation 4.10 to Equation 4.1, the nucleation rate can be defined as,

$$\dot{R}_n = k_{11} C_1^2 = f_n \frac{168\Omega D_g}{a_0^2} C_1^2, \tag{4.11}$$

where f_n has been introduced as a multiplication factor for parametric studies. Unless otherwise stated, $f_n = 1$.

Following the reaction definition of nucleation as,

$$C_1 + C_1 \xrightarrow{k_{11}} C_2. \tag{4.12}$$

The differential equations that model nucleation in C_1 and C_2 can then be defined as,

$$\frac{\partial C_1(t)}{\partial t} = -2k_{11} C_1(t)^2, \tag{4.13a}$$

$$\frac{\partial C_2(t)}{\partial t} = k_{11} C_1(t)^2, \tag{4.13b}$$

where the factor of -2 in the first equation is due to the nucleation process consuming two single gas atoms.

4.2.4 Growth

In order to cast the growth of bubbles into the form of Equation 4.1, we must first consider the case of an equally distributed concentration of spherical sinks C_i, with radius R, that are accumulating gas atoms from the surrounding medium. By subdividing the entire medium into unit cells centered around each sink, a capture volume with radius $\mathscr{R}$ can be defined such that,

$$\frac{4\pi\mathscr{R}^3}{3} C_i = 1. \tag{4.14}$$

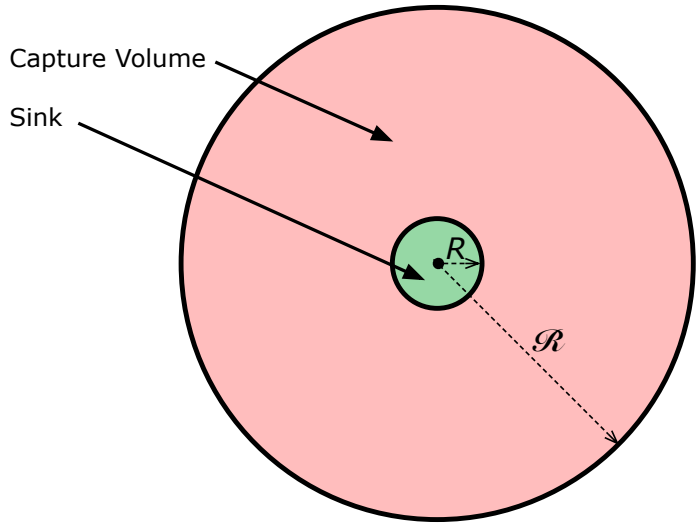

Figure 4.4: Unit cell for calculating sink behavior.

If the sinks are spaced far enough apart, then the following boundary condition applies,

$$\left(\frac{\mathrm{d}C}{\mathrm{d}t}\right)_{\mathscr{R}} = 0, \tag{4.15}$$

and the behavior of the system can be estimated by examining one particular sink, as displayed in Figure 4.4. The behavior of gas atoms within the capture volume can be solved using the diffusion equation in the annular spherical shell, $R \leq r \leq \mathscr{R}$, resulting in a concentration of gas atoms $C(r,t)$ at some radial position r and at some time t. The gas atom concentration at the surface of the sink can be specified as,

$$C(R,t) = C_R. \tag{4.16}$$

As discussed above, gas atoms are assumed to have a thermally stable concentration of 0. As a result, any gas atom immediately adjacent to the sink will be absorbed, resulting in $C_R = 0$.

Due to the fission process, gas atoms are assumed to be created uniformly within the capture volume. Furthermore, we will assume that there are no other sinks other than the sphere in the center of the unit cell. The resulting diffusion equation can then be expressed as,

$$\frac{\partial C}{\partial t} = \frac{D_g}{r^2}\frac{\partial C}{\partial r}\left(r^2\frac{\partial C}{\partial r}\right) + \gamma\dot{F} \tag{4.17}$$

At irradiation temperatures where gas atom mobility is sufficiently high, loss of gas atoms to the sink is balanced by production within the capture volume. In this case, the concentration at any point within the capture volume changes slowly and we can apply the quasi-stationary approximation, $dC/dt = 0$. Using the boundary conditions from Equations 4.15 and 4.16, the above diffusion

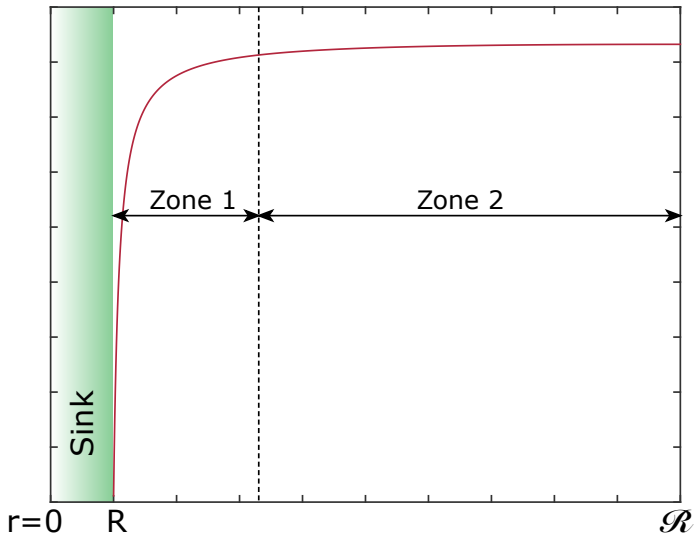

Figure 4.5: Example solution of diffusion equation for sink.

equation can be analytically solved:

$$C(r) = \frac{\gamma \dot{F}}{6D}\left(\frac{2\mathscr{R}(r-R)}{rR} - (r^2 - R^2)\right). \tag{4.18}$$

If the sinks are assumed to be widely spaced, then the capture volume is much larger than that of the sink, leading to a form similar to Figure 4.5. In this case, the solution naturally splits into two zones: Zone 1 where the concentration is rapidly changing as $r \to R$, and Zone 2 where the concentration is relatively constant. If Zone 1 is very close to the bubble, then r remains small, and Equation 4.17 for only Zone 1 can be reduced to,

$$\frac{\mathrm{d}}{\mathrm{d}r}\left(r^2 \frac{\mathrm{d}C}{\mathrm{d}r}\right) = -r^2 \frac{\gamma \dot{F}}{D} = 0. \tag{4.19}$$

The boundary condition from Equation 4.16 can be applied to the modified diffusion equation, however the second boundary condition is different:

$$C(\infty) = C(\mathscr{R}), \tag{4.20}$$

where $C(\mathscr{R})$ is the concentration determined by matching the solutions in Zones 1 and 2. Since $\mathscr{R}$ is much greater than the width of Zone 1, it effectively acts as an infinite medium when dealing with diffusion in Zone 1. Solving Equation 4.19 with the above boundary conditions results in:

$$C(r) = C(\mathscr{R})\left(1 - \frac{R}{r}\right). \tag{4.21}$$

The flux of the particles of the sink is calculated using the diffusivity of the gas atoms and the

concentration at the surface of the sink as [73],

$$J = -D_g \left(\frac{\mathrm{d}C}{\mathrm{d}r} \right)_R = -\frac{D_g C(\mathscr{R})}{R}. \tag{4.22}$$

Since the sink in the center of the unit cell is assumed to be spherical, the rate of absorption by a single sphere is given as [17],

$$\dot{g} = -4\pi R^2 J = 4\pi R D_g C(\mathscr{R}). \tag{4.23}$$

Finally, Equation 4.23 can be cast into the form of Equation 4.1 through the following observations: If the sinks are widely spaced enough, the concentration $C(\mathscr{R})$, which is defined as the concentration of atoms at the edge of the capture volume, approximates the entirety of Zone 2 as a result of the form in Figure 4.5. As such, $C(\mathscr{R})$ can be set to the concentration of single gas atoms in the solid, C_1.

In addition, Equation 4.23 applies to only one sink. To extend it to all sinks of type i in the solid, then Equation 4.23 must be multiplied by the concentration of sinks C_i. This leads to the final growth rate form of:

$$\dot{G}_i = k_i C_i C_1 = 4\pi R_i D_g C_i C_1. \tag{4.24}$$

The basic description of bubble growth follows,

$$C_1 + C_i \xrightarrow{k_i} C_{i+1}. \tag{4.25}$$

The corresponding differential equations are,

$$\frac{\partial C_1(t)}{\partial t} = -\sum_{i=2}^{N} k_i C_i(t) C_1(t), \tag{4.26a}$$

$$\frac{\partial C_i(t)}{\partial t} = k_{i-1} C_{i-1}(t) C_1(t) - k_i C_i(t) C_1(t), \quad 3 \leq i \leq N, \tag{4.26b}$$

where N is the total number of bubble groups. The first equation represents the loss of single gas atoms to each separate bubble size, requiring the negative sign. The limits on the second equation follows the assumption that nucleation only occurs between single gas atoms to form dimers, while growth occurs for everything else. As such, there can be no growth gain term for C_2. Furthermore, although the largest tracked bubble concentration in a BUCK simulation is C_N, the largest bubble group C_N still suffers losses into the C_{N+1} group. In the case that no losses are tracked for the largest bubble, as is the case with the verification calculations performed in Section 4.4.2, then $k_N = 0$, and the second portion of Equation 4.26b is removed.

4.2.5 Bubble Radius

Before discussing the re-solution rate, we must first clarify the relationship between the number of gas atoms in a particular bubble and its corresponding size within the lattice. As discussed in Section 4.2.3, single gas atoms within the lattice tend to take up one to two vacancies within the lattice. As atoms combine, the space occupied by small clusters is matched by the vacancies associated with each gas atom. Eventually, large enough groupings of atoms will start to take on a spherical shape as the interior bubble pressure pushes evenly against the lattice. However, as bubbles grow larger, the atoms require more space (Figure 2.2), leading to an increased absorption of vacancies and a decrease in the gas atom density in order to maintain equilibrium between the surface tension of the bubble and the interior pressure. This behavior is described in the Section 2.2.

Within BUCK, it is assumed that bubbles always maintain their equilibrium size. Relationships of this form assume that the vacancy availability, either through high mobility or high concentration, is large enough to always provide the bubble with the exact number of vacancies it requires to maintain equilibrium. This assumption has been made in many of the past bubble models [23, 65, 68, 74]. Some past models have incorporated explicit tracking of the vacancy and interstitial flux on the bubble surface, and have shown that the bubble sizes tend to initially be smaller than equilibrium sizes, and eventually will grow up to 30% of their equilibrium size once the interstitial and vacancy flux reaches steady-state conditions [67, 75]. This tends to support recent investigations on atom density in small UO_2 bubbles that have shown the atom densities are near that of solid xenon [21,22]. A second study focusing on the relaxation behavior of non-equilibrium bubbles showed that the only during overpower or undercooling transients does the equilibrium bubble assumption become an issue [76].

Regardless, the current work aims to isolate the behavior due to the newly calculated re-solution parameter. As such, the bubbles will be assumed to maintain their equilibrium radii following similar formulation to Equation 2.3,

$$\frac{1}{\rho_i} = B + \left[\left(\frac{2\gamma}{kT} \right) \frac{1}{R_i} + \frac{\sigma}{kT} \right]^{-1}, \tag{4.27}$$

where ρ_i is the atomic density for bubble type i, B is the van der Waals constant for xenon, γ is the surface tension, k is the Boltzmann constant, T is the temperature, and R is the bubble radius. Other equations of state exist for xenon gas within a solid, as was discussed in Section 3.3. However, comparisons of different types of models in Figure 3.8 shows little deviation between models, thus the van der Waal formulation will be utilized in BUCK.

Within BUCK, the bubbles are grouped based on the number of atoms they contain. Expanding ρ_i into its representation of atoms per volume, Equation 4.27 gives

$$m_i = \frac{4}{3\pi kTB} \left(\frac{2\gamma R_i^3 + \sigma R_i^4}{\frac{2\gamma}{kT} + R_i \left[\frac{\sigma}{kT} + \frac{1}{B} \right]} \right). \tag{4.28}$$

Equation 4.28 relates the number of gas atoms for a given bubble, m_i, to the radius of the equilibrium bubble. Due to the higher order R_i terms, no closed analytical solution exists for Equation 4.28, thus a simple Newton algorithm, similar to the method discussed in Section 4.3.2, was utilized to determine the bubble radius for a given number of atoms.

4.2.6 Re-solution

The process of re-solution is thoroughly discussed in Chapter 3. What remains is an appropriate model of the form in Equation 4.1 for implementation into BUCK.

Although it may be possible for several atoms to be knocked out of the bubble simultaneously, the formulation of the re-solution rate in Chapter 3 is essentially a time averaged phenomena. For simplicity, the re-solution knockout process is assumed to occur on an atom-by-atom basis based on the re-solution rate. Following the formulation in Equation 3.57, the re-solution rate for bubble type i is,

$$\dot{K}_i = l_i C_i = \dot{F} b_i f_b m_i C_i, \tag{4.29}$$

where $\dot{F}$ is the fission rate density, b_i is the re-solution parameter for bubble type i, f_b is a multiplier utilized for parametric studies ($f_b = 1$ unless otherwise stated), m_i is the number of atoms in bubble type i, and C_i is the concentration of bubble type i. Here, l_i represents the re-solution rate constant for the re-solution process, which is essentially a "loss" term. Since the rate constant is not multiplied by two concentrations as in Equation 4.1, the units correspond to s^{-1}.

The re-solution process can be described as a bubble of size i losing a single atom and becoming a bubble of size $i-1$, or:

$$C_i \xrightarrow{l_i} C_{i-1} + C_1, \tag{4.30}$$

the differential equations that describe the re-solution process are,

$$\frac{\partial C_1(t)}{\partial t} = 2 l_i C_2(t) + \sum_{i=3}^{N} l_i C_i(t), \tag{4.31a}$$

$$\frac{\partial C_i(t)}{\partial t} = l_{i+1} C_{i+1}(t) - l_i C_i(t), \quad 2 \leq i \leq N-1, \tag{4.31b}$$

$$\frac{\partial C_N(t)}{\partial t} = -l_N C_N(t). \tag{4.31c}$$

The first equation requires the separate inclusion for the re-solution of C_2 since the splitting of a dimer results in two single gas atoms. The summation in Equation 4.31a includes the re-solution event for all other bubble sizes, following the description of the re-solution process in Equation 4.30. Equation 4.31b includes the re-solution loss term, as well as a gain term due to the larger bubble size losing an atom. The last equation is similar to Equation 4.31b except the gain term is not included since no larger bubbles exist by definition.

4.2.7 Burnup

For many of the results, the burnup is calculated in order to provide an easy comparison tool between simulations with different fission rate densities. The burnup quantifies the consumption of uranium atoms through fission, and is calculated by,

$$Bu = \frac{t\dot{F}M_{UC}}{\rho_{th}\chi N_a}, \tag{4.32}$$

where Bu is the burnup in atoms fissioned per initial heavy metal atom, or (a/o), t is the time, $\dot{F}$ is the fission rate density, M is the molar mass of uranium monocarbide, ρ_{th} is the theoretical density of UC, χ is the fractional density of the fuel, and N_a is Avogadro's number.

4.2.8 Total Number of Gas Atoms

During each simulation, it is important to ensure the conservation of gas atoms. The total concentration of gas atoms in the simulation can be calculated as,

$$C_0 = \sum_{i=1}^{N} C_i m_i, \tag{4.33}$$

where C_i is the concentration of bubble size i, and m_i is the number of gas atoms in bubble size i.

4.2.9 Swelling

For the BUCK simulations, the overall swelling due to single gas atoms and their subsequent bubbles is the most important parameter to try and match with previous experimental studies. Each bubble occupies space within the lattice, and the swelling due to the bubbles can be calculated from their radius and concentration by,

$$S = \sum_{i=1}^{N} \frac{4}{3}\pi R_i^3 C_i, \tag{4.34}$$

where S is the sum over all the bubble groups N, each with radius R_i and concentration C_i. As defined in Equation 4.34, the swelling has units of fractional growth, or rather,

$$S = \frac{V_{\text{final}}}{V_{\text{initial}}}. \tag{4.35}$$

4.2.10 Assumptions

The assumptions discussed above are compiled into one list below. The reader is referenced to the appropriate sections for details.

- All gas atoms behave as xenon atoms,

- Gas atoms are created uniformly based on the fission rate density,
- Clusters and bubbles of gas atoms are stationary,
- Single gas atoms reside in the lattice only at uranium vacancies or Schottky defects,
- The nucleation process only results in the formation of dimers from two single gas atoms,
- The nucleation process occurs homogeneously,
- There is no thermal loss of atoms from clusters and bubbles,
- Diffusion-limited kinetics can be applied to all sizes of atom clusters,
- The gas bubbles are widely spaced such that the two zone assumption holds,
- The bubbles maintain their equilibrium size following the van der Waals equation of state (Equation 2.3).

4.3 Methods

The theory described above was implemented into the MOOSE framework [59]. MOOSE is a relatively new software framework developed by Idaho National Laboratory (INL) that builds on previous C++ finite element analysis libraries [77] to provide scientists with a Finite Element Analysis package in which to easily implement their own physics. MOOSE is developed as a framework rather than a standalone code, essentially providing the tools necessary to solve fully coupled multiphysics through integration of implicit solvers, native mesh and time-step adaptivity, and parallelization. Although MOOSE contains several physics modules developed within the framework such as solid mechanics, thermo-mechanical physics, and phase-field dynamics, separate "animals" make up the specific application within the MOOSE framework. Examples of specific applications are INL's own BISON, a nuclear fuel performance code [78], and MARMOT, a phase-field mesoscale code [79].

By sharing the same code architecture, each application can leverage the MOOSE framework to natively provide coupling [80, 81]. Since the BUCK models focus on mesoscale calculations that hope to inform full scale simulations of fuel rods, the choice to use the MOOSE framework was clear. A quick background of the details behind the MOOSE framework as it pertains to the current work is described in the following sections.

4.3.1 Coupled Ordinary Differential Equations

Since the only independent variable that drives the physics defined in Section 4.2 is time, the equations that can be used to describe the behavior of bubble size can be reduced to ordinary differential

equations. As a simple example, consider the steady state behavior of single gas atoms, dimers, and trimers:

$$\begin{bmatrix} -k_{11}C_1 & -k_{12}C_1 & -k_{13}C_1 \\ k_{11}C_1 & -k_{12}C_1 & 0 \\ 0 & k_{12}C_1 & -k_{13}C_1 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} -\gamma\dot{F} \\ 0 \\ 0 \end{bmatrix}, \tag{4.36}$$

where $\gamma\dot{F}$ represents the source term of single gas atoms, k_{11} is the rate constant for nucleation, k_{12} is the rate constant for absorption of single gas atoms by dimers, and k_{13} is the rate constant for absorption of single gas atoms by trimers. Equation 4.36 can be represented in vector notation as,

$$\mathbf{Au} = \mathbf{b}. \tag{4.37}$$

The solution is contained within the term **u**, while the physical relationships are contained in the operator **A** and the forcing term **b**. Since Equation 4.36 is non-linear, numerical methods must be used to determine the solution vector **u**. This is achieved by iteratively reducing the error of the calculated numerical approximation, or residual, given as,

$$\mathbf{r} = -\mathbf{Au} + \mathbf{b}. \tag{4.38}$$

Within MOOSE, Newton's method is the primary way to determine the roots, or the solution of Equation 4.38 where **r** is approximately 0.

4.3.2 Newton's Method

Newton's method is a popular technique for finding the roots of nonlinear equations, and is essentially based on a Taylor series expansion of a function [82]. In one dimension, the Taylor series is defined as,

$$f(x^{n+1}) = f(x^n) + f'(x^n)(x^{n+1} - x^n) + \mathrm{O}(x^n). \tag{4.39}$$

Since we are trying to solve for $f(x) = 0$ the above equation can be set to 0 and rearranged (ignoring the higher order terms $\mathrm{O}(x)$) to create an iterative step to approach $f(x) = 0$:

$$x^{n+1} = x^n - \frac{f(x^n)}{f'x^n}. \tag{4.40}$$

Figure 4.6 shows the basic iteration process using the Newton method. Starting at some initial guess x^0, the value of $f(x^0)$ and $f'(x^0)$ is determined. x^1 is then calculated by Equation 4.40, and the process is then repeated using the new value. Recall that Newton's method is trying to determine $f(x) = 0$, thus each iteration gets closer and closer to $f(x) = 0$. The total number of iterations depends is determined by some convergence criteria, where the iterations are terminated as the function $f(x^n)$ reaches some preset minimum.

As discussed in the above section, we are seeking the minimization of the residual through the

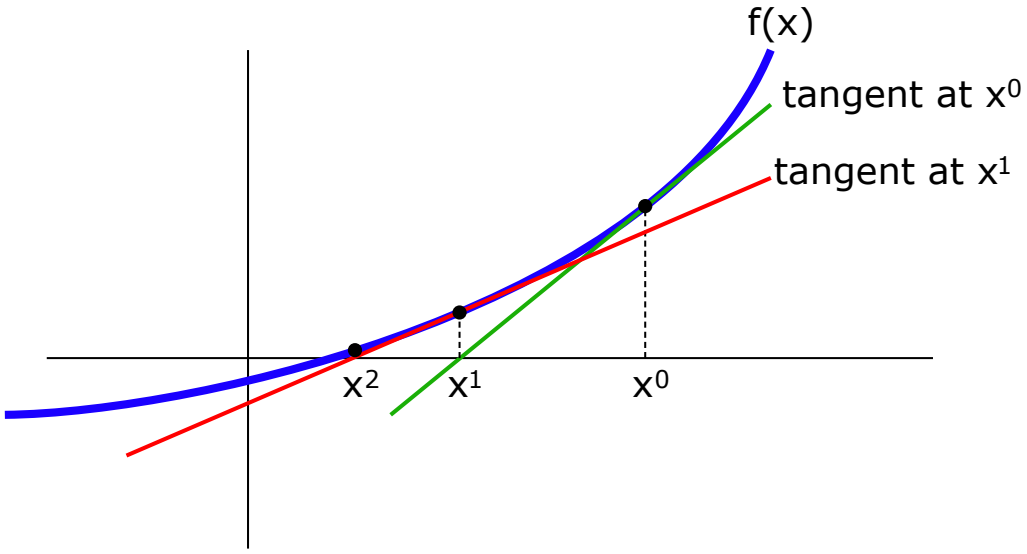

Figure 4.6: Schematic of Newton's method.

manipulation of the solution vector $\mathbf{u}$. Applying Newton's method to Equation 4.38 results in,

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \delta\mathbf{u}^n, \tag{4.41a}$$

$$\delta\mathbf{u}^n = -\frac{\mathbf{r}(\mathbf{u}^n)}{\mathbf{J}(\mathbf{u}^n)}, \tag{4.41b}$$

where n is the iteration index, $\mathbf{r}(\mathbf{u}^n)$ is the residual vector, $\mathbf{u} = [u_0, ..., u_j]$ is the set of variables and $\mathbf{J}(\mathbf{u}^n)$ is the Jacobian defined as,

$$\mathbf{J}_{ij}(\mathbf{u}^n) = \frac{\partial \mathbf{r}_i(\mathbf{u}^n)}{\partial u_j}. \tag{4.42}$$

In general, the Jacobian for two coupled nonlinear equations $F_1(u_1, u_2)$ and $F_2(u_1, u_2)$ is defined as [83],

$$\mathbf{J} = \begin{bmatrix} \frac{\partial F_1}{\partial u_1} & \frac{\partial F_1}{\partial u_2} \\ \frac{\partial F_2}{\partial u_1} & \frac{\partial F_2}{\partial u_2} \end{bmatrix}. \tag{4.43}$$

4.3.3 GMRES

If both $\mathbf{r}$ and $\mathbf{J}$ are known, then the $\delta\mathbf{u}$ terms can be determined, and $\mathbf{u}$ can be updated using Equation 4.41a. However, each term of the Jacobian can potentially be very complicated and difficult to solve, even for simple problems. To avoid these complications, the Generalized Minimal Residual (GMRES) algorithm can be utilized, avoiding calculation of the full Jacobian [84]. By rearranging Equation 4.41a, we can introduce a new residual defined as,

$$\mathbf{r}_0 = -\mathbf{r}(\mathbf{u}) - \mathbf{J}(\mathbf{u})\delta\mathbf{u}. \tag{4.44}$$

Each GMRES "linear" iteration is denoted by k. Since we are seeking the residual within one Newton, or "non-linear" step, the index n has been dropped. In GMRES, $\delta\mathbf{u}$ is defined as a linear combination of Krylov vectors up to the iteration number $k-1$, $\{\mathbf{r}_0, \mathbf{J}\mathbf{r}_0, \mathbf{J}^2\mathbf{r}_0, ..., \mathbf{J}^{k-1}\mathbf{r}_0\}$:

$$\delta\mathbf{u} = \delta\mathbf{u}_0 + \sum_{l=0}^{k-1} \beta_l (\mathbf{J})^l \mathbf{r}_0, \tag{4.45}$$

where $\mathbf{u}_0$ is some initial guess (usually 0). By recasting $\delta\mathbf{u}$ in the above way, explicit definition of J is not necessary, rather the action of J on r_0 is all that is needed. Furthermore, that action can be approximated similar to the definition of the derivative:

$$\mathbf{J}\mathbf{r}_0 \approx \frac{\mathbf{r}(\mathbf{u}+\varepsilon\mathbf{r}_0) - \mathbf{r}(\mathbf{u})}{\varepsilon}, \tag{4.46}$$

where ε is some small perturbation. We can show the validity of the Equation 4.46 by working backwards. If we plug in the basic Jacobian from Equation 4.43 into the above Equation 4.46, we find [82],

$$\frac{\mathbf{F}(\mathbf{u}+\varepsilon\mathbf{v}) - \mathbf{F}(\mathbf{u})}{\varepsilon} = \begin{pmatrix} \frac{F_1(u_1+\varepsilon v_1, u_2+\varepsilon v_2) - F_1(u_1,u_2)}{\varepsilon} \\ \frac{F_2(u_1+\varepsilon v_1, u_2+\varepsilon v_2) - F_2(u_1,u_2)}{\varepsilon} \end{pmatrix}. \tag{4.47}$$

Now using a first-order Taylor series expansion of $F(u+\varepsilon v)$ about u in the above equation:

$$\frac{\mathbf{F}(\mathbf{u}+\varepsilon\mathbf{v}) - \mathbf{F}(\mathbf{u})}{\varepsilon} \approx \begin{pmatrix} \frac{F_1(u_1,u_2) + \varepsilon v_1 \frac{\partial F_1}{\partial u_1} + \varepsilon v_2 \frac{\partial F_1}{\partial u_2} - F_1(u_1,u_2)}{\varepsilon} \\ \frac{F_2(u_1,u_2) + \varepsilon v_1 \frac{\partial F_2}{\partial u_1} + \varepsilon v_2 \frac{\partial F_2}{\partial u_2} - F_2(u_1,u_2)}{\varepsilon} \end{pmatrix}. \tag{4.48}$$

Simplification shows:

$$\frac{\mathbf{F}(\mathbf{u}+\varepsilon\mathbf{v}) - \mathbf{F}(\mathbf{u})}{\varepsilon} \approx \begin{pmatrix} v_1 \frac{\partial F_1}{\partial u_1} + v_2 \frac{\partial F_1}{\partial u_2} \\ v_1 \frac{\partial F_2}{\partial u_1} + v_2 \frac{\partial F_2}{\partial u_2} \end{pmatrix} = \mathbf{J}\mathbf{v}. \tag{4.49}$$

The GMRES is a powerful tool to allow the use of Newton's method without the full analytical form of the Jacobian.

4.3.4 JFNK

Putting all of the above together, we form the Jacobian Free Newton-Krylov algorithm [82]. The basic program flow consists of two loops. The inner loop is the GMRES algorithm, and determines the $\delta\mathbf{u}$ terms. After convergence, a Newton non-linear step, or outer loop, moves the values of u by $\delta\mathbf{u}$ in an attempt at minimizing the residual. GMRES is then utilized again to find the $\delta\mathbf{u}$ terms, and so on.

4.3.5 Pre-conditioning

Although the Jacobian is never fully formed within JFNK methods, some sort of estimation of the Jacobian is necessary to ensure convergence [82]. This process is known as preconditioning, and can be described as,

$$\mathbf{r}'(\mathbf{u}_i)\mathbf{M}^{-1}(\mathbf{M}\delta\mathbf{u}_{i+1}) = -\mathbf{r}(\mathbf{u}_i), \tag{4.50}$$

where $\mathbf{M}$ is the preconditioning matrix or process. Within GMRES, only the action $\mathbf{M}^{-1}$ is applied to the residual; Applying Equation 4.50 to Equation 4.46 results in,

$$\mathbf{J}\mathbf{M}^{-1}\mathbf{r}_0 \approx \frac{\mathbf{r}(\mathbf{u}+\varepsilon\mathbf{M}^{-1}\mathbf{r}_0)-\mathbf{r}(\mathbf{u})}{\varepsilon}. \tag{4.51}$$

Although there are several ways to create $\mathbf{M}$, the default process in MOOSE consists of basic Block Diagonal Preconditioning and consists of the main diagonal of the Jacobian. Using Equation 4.43 as an example, the Block Diagonal preconditioner is simply,

$$\mathbf{J} = \begin{bmatrix} \frac{\partial F_1}{\partial u_1} & 0 \\ 0 & \frac{\partial F_2}{\partial u_2} \end{bmatrix}. \tag{4.52}$$

Following Equation 4.52, the preconditioner is only the derivative of the partial differential equation with respects to the variable it acts on. This, along with the residual definition, are the only two pieces that need to be implemented in MOOSE to add the desired physics behavior.

4.3.6 Dampers

At the start of the simulation, all the concentrations have a value of 0. As time increases, the gas atoms will eventually filter up to larger and larger bubbles. At very low concentrations, the bubble concentration may drift to negative concentrations due, an unphysical value that occurs if the time steps are too large. In order to prevent this, a dampening coefficient is applied to variables if the change in the variable leads to a negative number. These coefficients essentially limit the change in a given variable during each non-linear iteration by modifying Equation 4.41a through the introduction of the dampening coefficient α:

$$\mathbf{u}_j^{n+1} = \mathbf{u}_j^n + \alpha\delta\mathbf{u}^{n+1}. \tag{4.53}$$

A coefficient of $\alpha = 0.01$ applied only to variables in which $\mathbf{u}_{n+1}$ is calculated to be negative was determined to be sufficient to ensure that all concentrations remained positive.

4.3.7 Time Discretization

Although the simple example in Equation 4.36 assumes steady state conditions, the time dependence of each variable is important in order to capture the swelling behavior as a function of burnup. The Implicit Euler method can be utilized to discretize the time derivative, and is described as,

$$\dot{\mathbf{u}}(t+\Delta t) = \frac{\mathbf{u}(t+\Delta t) - \mathbf{u}(t)}{\Delta t}, \tag{4.54}$$

where $\mathbf{u}$ is the solution vector with time deriviative $\dot{\mathbf{u}}$. Rearranging Equation 4.54 and casting it into a residual form similar to Equation 4.38 results in,

$$\mathbf{r}(\mathbf{u}(t+\Delta t)) = \mathbf{u}(t+\Delta t) - \mathbf{u}(t) - \Delta t\dot{\mathbf{u}}(t+\Delta t), \tag{4.55}$$

with an associated Jacobian given as,

$$\mathbf{J}(\mathbf{u}(t+\Delta t)) = \mathbf{I} - \Delta t\dot{\mathbf{u}}(t+\Delta t). \tag{4.56}$$

Armed with the definition of the residual in Equation 4.55 and the definition of the Jacobian in Equation 4.56, the time dependence can be estimated using Newton's method in Equation 4.41.

In the BUCK calculations, backward differentiation formula with s=2 (BDF2) was utilized with variable time-stepping to take advantage of the superior convergent properties of BDF2 [85]. This is essentially an extension of the Implicit Euler form described previously, and is given as [85],

$$\dot{\mathbf{u}}(t+\Delta t_1+\Delta t_2) = \frac{1}{\Delta t_2}\left[\left(\frac{1+2\tau}{1+\tau}\right)\mathbf{u}(t+\Delta t_1+\Delta t_2) - (1+\tau)\mathbf{u}(t+\Delta t_1) + \left(\frac{\tau^2}{1+\tau}\right)\mathbf{u}(t)\right], \tag{4.57}$$

where $\tau = \Delta t_2/\Delta t_1$. Equation 4.57 gives the time derivative at some time step $t+\Delta t_1+\Delta t_2$ based on the values of $\mathbf{u}$ at time step $t+\Delta t_1+\Delta t_2$, as well as $\mathbf{u}$ at two previous time steps. Equation 4.57 can be implemented into the MOOSE structure following the same procedure as Implicit Euler described above. The time-stepper settings described in Section 4.4.3.2 ensured that the upper limits on τ, given by,

$$\tau < 1+\sqrt{2}, \tag{4.58}$$

were appropriately enforced.

4.3.8 MOOSE

The basic input file for BUCK is displayed in Appendix B, Since the variables are essentially mean field values, the results are agnostic of the problem geometry. As a result, the mesh is set to a one-dimensional line of length 1 with only one segment. Initial conditions for all values are set to 0, unless otherwise specified. Moose "Actions" are utilized with the BUCK specific "Bubbles" block, and automatically create the variables, as well as the implementation of the time, nucleation,

Table 4.2: List of parameters and their values.

Symbol	Parameter	Value	Units	Ref.
γ	Fission gas yield	0.25	gas atoms/fission	[5]
Ω	Lattice site volume	$1.53{\cdot}10^{-11}$	μm^3	[69]
a_0	Lattice parameter	$4.96{\cdot}10^{-4}$	μm^3	[5]
γ	Surface tension	0.626	J/m^2	[]
B	van der Waals constant	$8.469{\cdot}10^{-29}$	m^3	[]
k	Boltzmann Constant	$1.3806{\cdot}10^{-23}$	J/K	
N_a	Avogadro's number	atoms/mol		
M_{UC}	Molar mass of UC	240	g/mol	
ρ_{th}	Theoretical density of UC	13.63	g/cm^3	[5]
χ	Fractional density of fuel	0.80		
D_0	Diffusion constant	$4.6{\cdot}10^{7}$	$\mu m^2/s$	[62]
Q	Activation energy	328.4	kJ/mol	[62]
$D_{0,f}$	Irradiation enhanced diffusion coefficient	$1.3{\cdot}10^{-9}$	μm^5/fission	[62]
f_b	Re-solution multiplication factor	1		
$\dot{F}$	Fission rate density	$6{\cdot}10^{13}$	fission/second/cm^3	[62, 69]

growth, and re-solution kernels. MOOSE-type "Postprocessors" are utilized for the total and by bubble calculations of atoms, concentration, and swelling. The executioner utilized is the Backward Differentiation Formulation with $S = 2$ (BDF2).

Studies on the maximum time-step and number of variables needed to capture adequate solution fidelity are included in Section 4.4.3.

4.4 Results

The theory described in Section 4.2 was implemented into the MOOSE framework application BUCK using techniques described in Section 4.3. The results from several simulations are described below. Before full simulations were attempted, several verification tests were completed to ensure the physics was implemented correctly. The tests followed the guidelines formulated by the MOOSE standards [86]. Some were as simple as ensuring consistent user input, while others matched simple calculations to exact solutions. Several of the latter will be discussed in Section 4.4.2. In addition, several parameters had to be optimized to ensure minimal computational expense, and are discussed in Section 4.4.3

4.4.1 Simulation Parameters

The simulations that follow utilize the same basic simulation parameters from Table 4.2 unless otherwise specified.

4.4.2 Verification

The following sections address verification of the physics implemented in BUCK through comparisons of several artificial simulations to analytical solutions.

4.4.2.1 Nucleation

The first basic test is on the nucleation process described in Section 4.2.3. The process describes the formation of a dimer from the combination of two singles atoms of the form,

$$C_1 + C_1 \xrightarrow{k_{11}} C_2, \tag{4.59}$$

where the rate constant is defined in Equation 4.11 as,

$$k_{11} = \frac{168\Omega D_g}{a_0^2}. \tag{4.60}$$

The concentration of single gas atoms and dimers as a function of time is given in Equation 4.13. Combined with the initial conditions,

$$C_1(0) = C_{1,0}, \tag{4.61a}$$
$$C_2(0) = C_{2,0}, \tag{4.61b}$$

Equations 4.13a and 4.13b can be solved for $C_1(t)$ and $C_2(t)$ as,

$$C_1(t) = \frac{C_{1,0}}{2C_{1,0}k_{11}t + 1}, \tag{4.62a}$$
$$C_2(t) = \frac{C_{1,0}}{2}\left(1 - \frac{1}{2C_{1,0}kt + 1}\right). \tag{4.62b}$$

Equation 4.62 was compared to a hypothetical BUCK simulation between two species, C_1 and C_2, where the initial conditions are given as $C_1(0) = 1{\cdot}10^5$ and $C_2(0) = 0$. A transient simulation was run for $5{\cdot}10^7$ seconds with timesteps of $1{\cdot}10^5$ at a constant temperature of 1000 K. The parameters from Table 4.2 were utilized. The results of the simulation are displayed in Table 4.3. Even with such large timesteps, BUCK matched the analytical solution with a deviation well below 0.1% for C_1 and C_2.

One other important test for each kernel is conservation of gas atoms. Following Equation 4.33, the conservation statement,

$$C_1(t) + 2C_2(t) = C_1(0) + 2C_2(0) \tag{4.63}$$

must always hold true, a behavior that occurs for the above example.

Table 4.3: Comparison between the nucleation analytical solution and BUCK.

Time	C_1 Analytical	C_1 BUCK	C_1 % diff	C_2 Analytical	C_2 BUCK	C_2 % diff
0	$1.0000 \cdot 10^2$	$1.0000 \cdot 10^2$	–	–	–	–
$1 \cdot 10^7$	$9.3700 \cdot 10^2$	$9.3700 \cdot 10^2$	0.004	$3.1500 \cdot 10^1$	$3.1500 \cdot 10^1$	0.061
$2 \cdot 10^7$	$8.8100 \cdot 10^2$	$8.8100 \cdot 10^2$	0.007	$5.9300 \cdot 10^1$	$5.9300 \cdot 10^1$	0.056
$3 \cdot 10^7$	$8.3200 \cdot 10^2$	$8.3200 \cdot 10^2$	0.010	$8.4000 \cdot 10^1$	$8.4000 \cdot 10^1$	0.051
$4 \cdot 10^7$	$7.8800 \cdot 10^2$	$7.8800 \cdot 10^2$	0.013	$1.0600 \cdot 10^2$	$1.0600 \cdot 10^2$	0.047
$5 \cdot 10^7$	$7.4800 \cdot 10^2$	$7.4800 \cdot 10^2$	0.015	$1.2600 \cdot 10^2$	$1.2600 \cdot 10^2$	0.043

4.4.2.2 Growth

The next fundamental rate equation that determines bubble size is growth due to absorption of single gas atoms, as described in Section 4.2.4. For verification purposes, we can look at the growth behavior between four species,

$$C_1 + C_2 \xrightarrow{k_2} C_3 + C_1 \xrightarrow{k_3} C_4. \tag{4.64}$$

Unlike nucleation, the rate equations that describe bubble growth are non-linear due to the inclusion of the C_1 concentration in every term, thus an analytical solution of the partial differential equations defined in Equation 4.26 is only available if C_1 is kept constant. Enforcing this restriction,

$$C_1(t) = C_{1,0}, \tag{4.65}$$

along with the initial conditions,

$$C_2(0) = C_{2,0}, \tag{4.66a}$$
$$C_3(0) = 0, \tag{4.66b}$$
$$C_4(0) = 0, \tag{4.66c}$$

the analytical solutions can be solved as,

$$C_2(t) = C_{2,0}\exp(-k_2 C_1 t), \tag{4.67a}$$
$$C_3(t) = \frac{k2}{k3-k2} C_{2,0}(\exp[-k_2 C_1 t] - \exp[-k_3 C_1 t]), \tag{4.67b}$$
$$C_4(t) = \frac{1}{k_3 - k_2} C_{2,0}[k_3(1-\exp[-k_2 C_1 t]) - k_2(1-\exp[-k_3 C_1 t])]. \tag{4.67c}$$

Equation 4.67 was compared to a hypothetical BUCK simulation between four species with the initial conditions $C_1(t) = 1{\cdot}10^5$ and $C_2(0) = 1{\cdot}10^5$. A transient simulation was run for $5{\cdot}10^7$ seconds with timesteps of $1{\cdot}10^5$ at a constant temperature of 1000 K. The parameters from Table 4.2 were utilized. The results of the simulation are displayed in Table 4.4. Even with such large timesteps, BUCK matched the analytical solution with a deviation well below 0.1% for C_2 and C_3, with a

Table 4.4: Comparison between the growth analytical solution and BUCK.

Time	C_2 BUCK	C_2 diff [%]	C_3 BUCK	C_3 diff [%]	C_4 BUCK	C_4 diff [%]
0	$1.000 \cdot 10^5$	–	–	–	–	–
$1 \cdot 10^5$	$9.222 \cdot 10^4$	0.003	$7.312 \cdot 10^3$	0.096	$4.639 \cdot 10^2$	0.868
$2 \cdot 10^5$	$8.505 \cdot 10^4$	0.007	$1.322 \cdot 10^4$	0.091	$1.728 \cdot 10^3$	0.372
$3 \cdot 10^5$	$7.844 \cdot 10^4$	0.010	$1.793 \cdot 10^4$	0.085	$3.633 \cdot 10^3$	0.209
$4 \cdot 10^5$	$7.234 \cdot 10^4$	0.013	$2.161 \cdot 10^4$	0.080	$6.046 \cdot 10^3$	0.130
$5 \cdot 10^5$	$6.672 \cdot 10^4$	0.016	$2.443 \cdot 10^4$	0.075	$8.853 \cdot 10^3$	0.083

maximum difference of 0.9% for C_4 for early timesteps.

Unlike the nucleation process, the verification example described above does not conserve the total number of gas atoms as C_1 is not allowed to change from its initial value. However, in the case of a population being produced at the expense of another (i.e. C_2 forming C_3), then the species conservation statement,

$$C_2(t) + C_3(t) + C_4(t) = C_2(0), \tag{4.68}$$

should always hold, a behavior that indeed occurred in the above example.

If a similar example is calculated in which C_1 is allowed to vary, the conservation statement,

$$C_1(t) + 2C_2(t) + 3C_3(t) + 4C_4(t) = C_1(0) + 2C_2(0), \tag{4.69}$$

was enforced.

4.4.2.3 Re-solution

Finally, we can compare an example BUCK problem to an analytical solution of three species of the form,

$$C_3 \xrightarrow{l_3} C_1 + C_2 \xrightarrow{l_2} 2C_1. \tag{4.70}$$

Utilizing the initial conditions,

$$C_3(0) = C_{3,0}, \tag{4.71a}$$
$$C_2(0) = 0, \tag{4.71b}$$
$$C_1(0) = 0, \tag{4.71c}$$

we can solve the partial differential equations defined in Equation 4.31 for the species involved:

$$C_3(t) = C_{3,0} \exp(-l_3 t), \tag{4.72a}$$
$$C_2(t) = \frac{l_3}{l_2 - l_3} C_{3,0} [\exp(-l_3 t) - \exp(-l_2 t)], \tag{4.72b}$$

Table 4.5: Comparison between the re-solution analytical solution and BUCK.

Time	C_1 BUCK	C_1 % diff	C_2 BUCK	C_2 % diff	C_3 BUCK	C_3 % diff
$0 \cdot 10^0$	–	–	–	–	$1.000 \cdot 10^3$	–
$1 \cdot 10^4$	$5.197 \cdot 10^1$	0.010	$4.914 \cdot 10^1$	0.043	$9.499 \cdot 10^2$	0.001
$2 \cdot 10^4$	$1.050 \cdot 10^2$	0.008	$9.400 \cdot 10^1$	0.042	$9.023 \cdot 10^2$	0.003
$3 \cdot 10^4$	$1.588 \cdot 10^2$	0.006	$1.349 \cdot 10^2$	0.041	$8.572 \cdot 10^2$	0.004
$4 \cdot 10^4$	$2.133 \cdot 10^2$	0.005	$1.720 \cdot 10^2$	0.040	$8.142 \cdot 10^2$	0.005
$5 \cdot 10^4$	$2.683 \cdot 10^2$	0.003	$2.057 \cdot 10^2$	0.039	$7.734 \cdot 10^2$	0.007

$$C_1(t) = 3C_{3,0} - C_{3,0}\exp(-l_3 t) - \frac{2}{l_2 - l_3} C_{3,0}[l_2 \exp(-l_3 t) - l_3 \exp(-l_2 t)]. \tag{4.72c}$$

Equation 4.72 was compared to a hypothetical BUCK simulation between three species with the initial condition $C_3(0) = 1 \cdot 10^3$. A transient simulation was run for $5 \cdot 10^4$ seconds with timesteps of $1 \cdot 10^4$ at a constant fission rate density of $\dot{F} = 1 \cdot 10^{13}$ fsn/(s·m^3). The parameters from Table 4.2 were utilized. The results of the simulation are displayed in Table 4.5. Even with such large timesteps, BUCK matched the analytical solution with a deviation well below 0.01% for all species.

Lastly, the conservation of gas atoms can be checked through the conservation statement,

$$C_1(t) + 2C_2(t) + 3C_3(t) = 3C_3(0), \tag{4.73}$$

which remained true at all time steps.

4.4.3 Optimization

4.4.3.1 Number of Variables

The potentially large number of variables needed to represent the full bubble distribution is the largest source of computational expense. Since each bubble size is tracked explicitly, a scheme to determine the number of bubble groups is required to ensure that variables are not unnecessarily tracked in the simulation.

All of the rate processes described in Section 4.2 act on some concentration C_i through multiplication of the reaction rate by either C_i or the surrounding concentrations, C_{i-1}, C_{i+1}. As a result, it can be expected that if the concentration of C_i or its neighbors is sufficiently small, then explicit tracking of those concentrations will have minimal effect on the overall solution.

In determining the number of bubble groups necessary for a given calculation, the impact of any given group on the total solution is necessary. Within BUCK, the primary tracked value is the swelling defined in Section 4.2.9. In addition, all bubble concentrations are coupled through the single gas atom concentration C_1. As such, the ignoring variables above a given number of atoms N must not have a significant impact on the total swelling value or C_1 concentration.

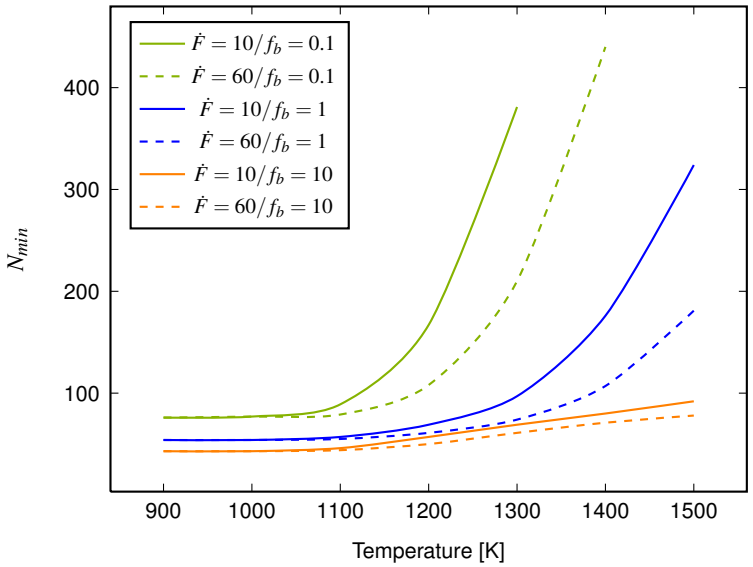

Figure 4.7: N_{min} for several different simulation parameters.

Following these guidelines, criteria for the minimum bubble size necessary for a given simulation can be defined: All bubble concentrations that are within 10^{-10} of the single gas atom concentration at any time must be explicitly tracked, or,

$$\frac{C_i(t)}{C_1(t)} \geq 10^{-10}. \tag{4.74}$$

The minimum bubble size that must be tracked, N_{min}, can be calculated for several different scoping studies; in order to determine N_{min}, simulations with overestimated values of N were run for different fission rate densities, $\dot{F} = \{10, 60\}$, as well as different re-solution multipliers, $f_b = \{0.1, 1, 10\}$. From the results presented in Figure 4.7, it is clear that f_b determines the baseline values of N_{min}, with lower values of f_b resulting in larger bubbles. As temperatures increase, $\dot{F}$ determines the slope, with lower values of $\dot{F}$ requiring more variables to capture the bubble distribution.

Due to radiation enhanced diffusion of gas atoms (Section 4.2.2), it was initially expected that higher values of $\dot{F}$ would result in an increase in bubble absorption, and thus larger bubbles. However, the re-solution rate also depends on $\dot{F}$, and ends up being the dominant parameter, ensuring that a high re-solution rate prevents large bubbles at both low and high temperatures. This behavior will be discussed later in Section 4.4.4

In order to ensure that information is not being lost by reducing the number of variables down to the values predicted by the curves in Figure 4.7, the simulation with $\dot{F} = 60$ and $f_b = 1$ was compared to a corresponding simulation that enforces N_{min}. Two temperatures, $T = \{900, 1500\}$ were compared for differences in swelling and C_1 concentration as presented in Table 4.6. The largest difference due to the total variable reduction is 0.5%, thus limiting simulations to N_{min} has a minimal impact on the result. Consequently, all of the simulations that follow utilize N values guided

Table 4.6: Difference between simulations with large N and N_{min}.

Time	900 K swelling [%]	900 K C_1 [%]	1500 K swelling [%]	1500 K C_1 [%]
$1 \cdot 10^1$	–	–	$3.73 \cdot 10^{-11}$	$3.34 \cdot 10^{-11}$
$1 \cdot 10^2$	–	$6.67 \cdot 10^{-12}$	$4.36 \cdot 10^{-9}$	$4.73 \cdot 10^{-9}$
$1 \cdot 10^3$	$4.58 \cdot 10^{-10}$	$5.04 \cdot 10^{-10}$	$2.08 \cdot 10^{-8}$	$3.72 \cdot 10^{-9}$
$1 \cdot 10^4$	$3.06 \cdot 10^{-8}$	$1.63 \cdot 10^{-8}$	$2.05 \cdot 10^{-9}$	$1.68 \cdot 10^{-10}$
$1 \cdot 10^5$	$5.24 \cdot 10^{-9}$	$6.07 \cdot 10^{-10}$	$4.83 \cdot 10^{-1}$	$2.54 \cdot 10^{-2}$
$1 \cdot 10^6$	$1.03 \cdot 10^{-1}$	$1.31 \cdot 10^{-2}$	$8.14 \cdot 10^{-2}$	$1.69 \cdot 10^{-3}$
$1 \cdot 10^7$	$1.69 \cdot 10^{-1}$	$1.78 \cdot 10^{-2}$	$8.08 \cdot 10^{-3}$	$1.45 \cdot 10^{-5}$
$7 \cdot 10^7$	$2.3 \cdot 10^{-2}$	$1.9 \cdot 10^{-3}$	$1.62 \cdot 10^{-3}$	$4.31 \cdot 10^{-7}$

by Figure 4.7, while simultaneously ensuring that the the criteria in Equation 4.74 is maintained.

4.4.3.2 Time stepping

In a single BUCK simulation, there are several different processes that occur at different time scales; nucleation from a field of single gas atoms occurs relatively early in the simulation, while growth of larger bubbles occurs much more slowly. Within the MOOSE framework, the variables are fully coupled and are explicitly solved, thus the time-step intimately determines the computational expense. This expense manifests itself through the number of non-linear iterations required for convergence.

An iteration adaptive time step routine was utilized to determine each time-step size. An initial time step of 0.1 seconds was utilized to ensure that the early nucleation behavior was adequately captured. After the first time-step, the MOOSE "TimeStepper" *IterationAdaptiveDT* raised the time step by a factor of 1.1 if the number of non-linear iterations was below 10. Conversely, the time step was reduced by a factor of 2 if the number of iterations was above 10. In this way, the total number of iterations is kept low enough through the direct control of the time-step, while simultaneously allowing the time-step to increase if the solution easily converged. This helped to reduce the computational expense for each simulation.

4.4.4 Bubble Distribution

The bubble concentration distribution at various burnups for a single temperature is shown in Figure 4.8. In general, higher temperatures increase the diffusivity of the gas atoms, resulting in more absorption and pushing the peak of the concentration distribution to larger bubble sizes. The bubble size distribution remains similar for temperatures below 1100 K, however as the temperature increases, the distribution shifts to larger bubble sizes. Since larger bubbles accommodate more gas atoms, the total concentration of bubbles must decrease due to the conservation of gas atoms, resulting in a lowering of the peak concentration as a function of temperature.

Figures 4.9 and 4.10 show bubble distributions utilizing different simulation parameters, the

first at 1100 K, representing low temperature behavior, and the other at 1400 K, representing high temperature behavior. Further analysis of the parametric study of the fission rate density and resolution multiplier are continued in Section 4.5.

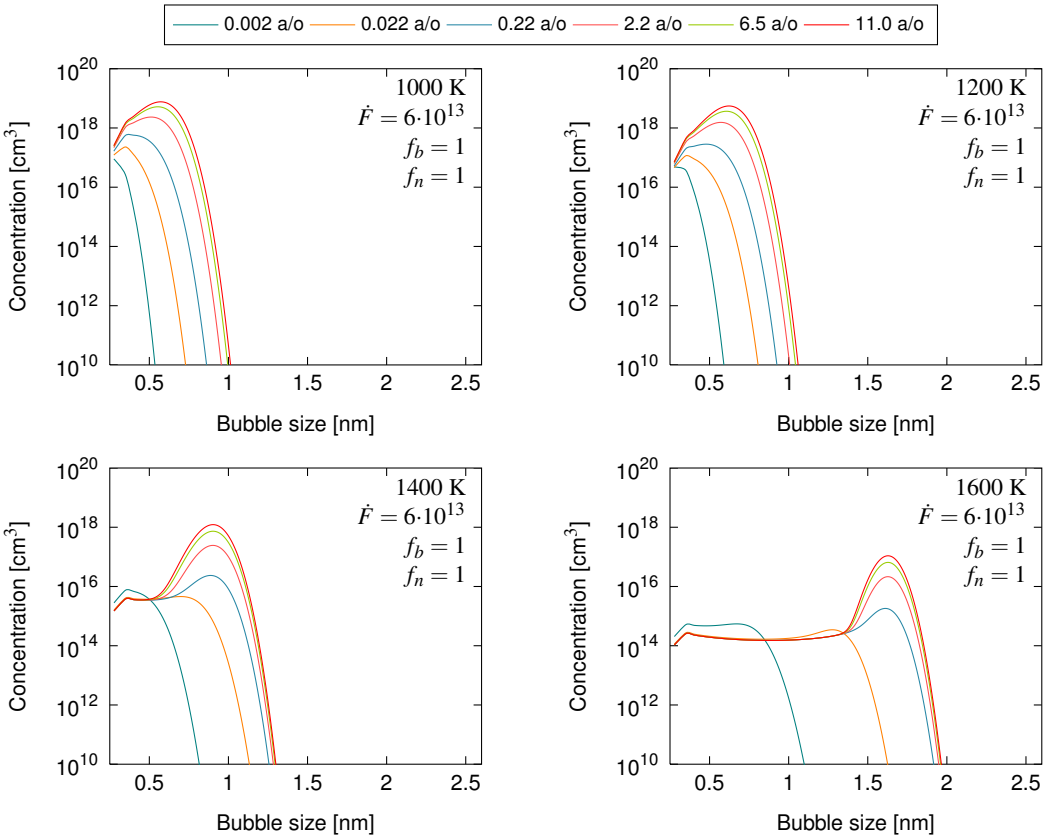

Figure 4.8: Bubble distribution at different temperatures for $\dot{F} = 6{\cdot}10^{13}$ fsn/(s·m^3) and $f_b = 1$.

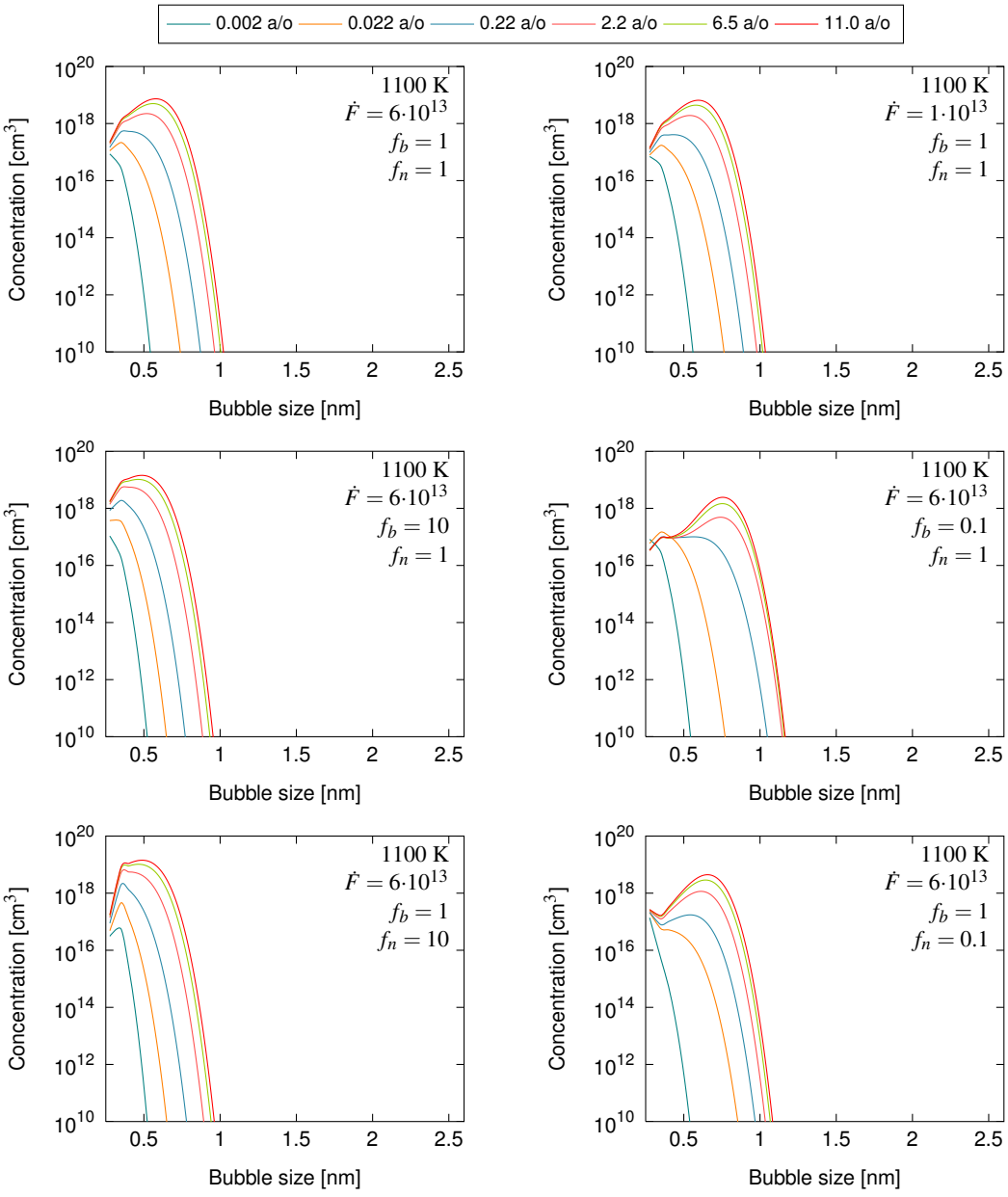

Figure 4.9: Bubble distribution at 1100 K for different irradiation parameters.

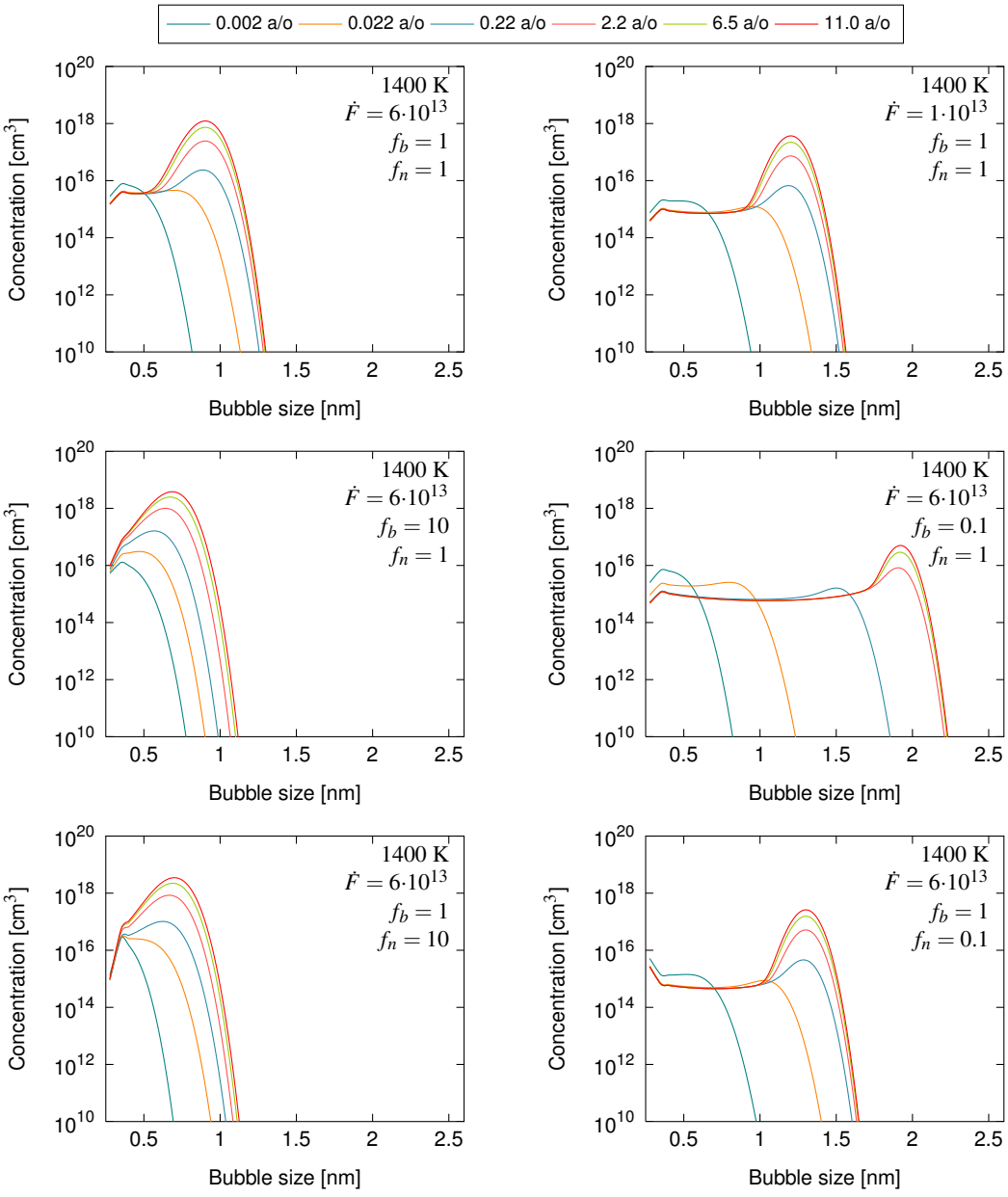

Figure 4.10: Bubble distribution at 1400 K for different parameters.

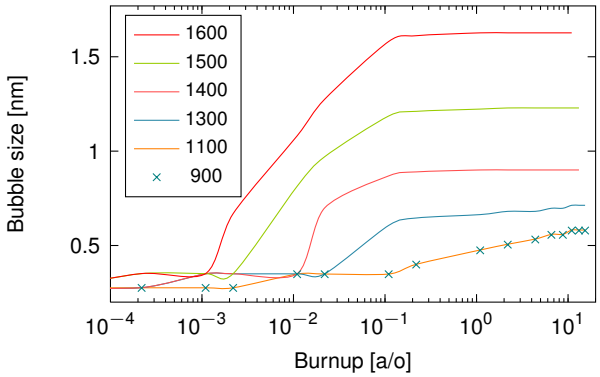

Figure 4.11: Bubble size at the concentration peak for $\dot{F} = 6{\cdot}10^{13}$ fsn/(s·m^3) and $f_b = 1$.

4.5 Discussion

4.5.1 Bubble Distribution Peak

The bubble size corresponding to the peak concentration of the bubble distribution is a convenient tool to describe the trends in Figure 4.8, and is plotted in Figure 4.11 as a function of burnup for several different temperatures. From both Figures 4.8 and 4.11, it is clear that the general shape of the distribution forms very early within the simulation, especially for the high temperature irradiations; For temperatures above 1400 K, the concentration peak reaches a plateau by burnup 0.1 a/o, or 5 days into the simulation. For these higher temperature simulations, a smooth distribution around the peak concentration forms, with the right side tail dropping quickly towards 0 and a left shoulder 40-50 nanometers wide that leads into a nearly flat concentration to the smallest bubble sizes. As time continues, the newly produced fission gas atoms accumulate around the bubble size at the peak concentration, boosting the peak to higher values, while the flat, small bubble concentration profile remains nearly constant. Unlike the higher temperature irradiations, a well defined peak does not develop at lower temperatures; The peak concentrations at lower temperatures occurs at bubble sizes smaller than the shoulder visible in the higher temperature irradiations. As a result, the peak distribution slowly pushes towards slightly larger bubble sizes. Due to the fission rate dependent diffusivity term defined in Equation 4.3 resulting in a constant diffusivity value at low temperatures (Figure 4.1), all bubble distributions are nearly identical below 1100 K.

Figure 4.12 displays the values of the bubble size at the peak concentration at the highest burnup, or rather the farthest right-hand value of each line in Figure 4.11, as a function of temperature. Figure 4.12 captures the tendency of higher temperature irradiations to create larger bubbles, as well as showing that lower temperature irradiations form similar distributions. Linearly extrapolating from the last two points to the melting temperature of uranium carbide (2600 K), the population will center around a 5 nanometer bubble, much less than the 100-400 nanometer bubble determined experimentally [9].

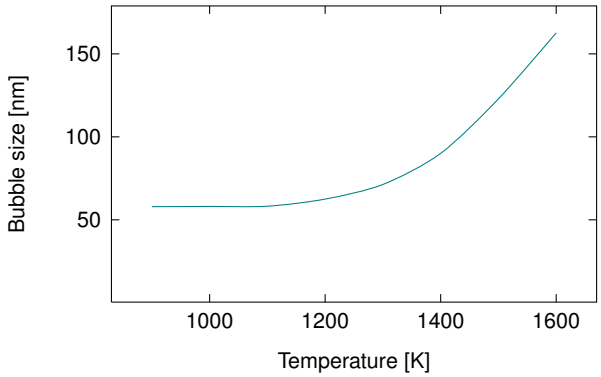

Figure 4.12: Concentration peak as a function of temperature for $\dot{F} = 6{\cdot}10^{13}$ fsn/(s·m^3) and $f_b = 1$.

4.5.2 Parametric Studies

The bubble distributions for differing values of fission rate density and the resolution multiplier are displayed in Figure 4.10 and 4.10. As expected, the re-solution rate has a strong effect on the bubble distribution. If the re-solution rate is increased by a factor of 10, the bubble distribution maintains the low temperature structure at all studied temperatures, with little change in the bubble size at the concentration peak as a function of temperature. Conversely, decreasing the re-solution rate by a factor of 10 results in a concentration peak at larger bubble sizes. The bubble distribution is the result of the competing thermal absorption and re-solution loss phenomena. As such, tuning the re-solution parameter produces bubble distributions similar to changing the temperature; Lowering the re-solution rate has a similar effect as raising the temperature, with the converse being true as well. At 1400 K, re-solution multipliers of $f_b = \{10, 1, 0.1\}$ leads to the bubble sizes at the concentration peak of $\{0.68, 0.90, 1.92\}$ nanometers. If the general trend given by these sparse points is followed, the re-solution parameter multiplier f_b must be between 10^{-3} and 10^{-4} to achieve bubble sizes on the order of 10^2 nanometers, or there must be some other physical phenomena driving the larger bubble sizes that is not captured in these simulations.

By changing the fission rate density, the impact of diffusivity vs. re-solution rate can be studied. When $\dot{F}$ is lowered from $6{\cdot}10^{13}$ to $1{\cdot}10^{13}$ fission/(sec·m^3), the bubble distribution shifts to larger bubble sizes. Following Section 4.2.2, the diffusivity includes a $\dot{F}$ dependent term which should result in smaller bubble sizes for lower values of $\dot{F}$. Simultaneously, the re-solution rate linearly scales by $\dot{F}$ (Equation 4.29), resulting increased bubble sizes. Between the two competing effects, the drop in re-solution rate is greater such that the bubbles produced at lower fission rate density values are larger, although not as large as the bubbles produced with $f_b = 0.1$.

The last parameter to be explored was the homogenous nucleation rate, which was scaled by $f_n = \{10, 1, 0.1\}$. The nucleation multiplier has a very similar contribution as the re-solution multiplier, with a smaller nucleation rate resulting in a higher concentration of larger bubbles, and a lower nucleation rate resulting in a bubble concentration distribution that is similar to low temperature irradiation distributions. In all cases, the nucleation multiplier has less of an impact on the overall

distribution deviation when compared to the re-solution multiplier. The cause of this deviation lies in the consumption of single gas atoms and the population of small bubbles; As the nucleation rate increases, more small bubbles remain in the distribution. Since all of the bubbles act as sinks of single gas atoms, fewer single gas atoms remain to be quickly absorbed. Since bubble size is determined by the competing growth and re-solution rate, a slower growth rate results in smaller bubbles. Ultimately, by limiting the total number of bubbles, slow bubble nucleation increases the chances of bubbles to grow.

One final parameter that was not explicitly tested was the diffusivity of the gas atoms. However, due to the temperature dependence of the diffusivity, some comparisons can be qualitatively made between different diffusivity models. Recalling Figure 4.1, the diffusivity model used in BUCK taken from Ronchi can be related to the diffusivities calculated by Matzke [5] by modifying the temperature. For example, utilizing the Ronchi formulation (the default BUCK model), a temperature of 1275 K matches the diffusivity of the Matzke model at 1400 K. Since the diffusivity is the only temperature dependent parameter in the simulations, artificially tuning the temperature reproduces results from different diffusivity models. Applying this knowledge to the bubble concentration distributions in Figure 4.8, it can be expected that the slower diffusivity models would produce distributions centered around smaller bubbles.

The isolation of individual parameters in the preceding parametric study highlights the impact of each parameter to the bubble concentration distribution. Focusing on the parameters above, the bubble concentration distribution can be shifted towards larger bubble sizes, in decreasing importance:

- Increasing temperature: Higher temperatures result in a higher absorption rate;
- Increasing diffusivity: Changes in the diffusivity model leads to similar results as changing temperature.
- Decreasing the re-solution parameter: Lowering the re-solution rate leads to more absorption, and larger bubbles;
- Decreasing the nucleation rate: Lowering nucleation results in fewer bubbles from which to consume single gas atoms. This boosts absorption in the few bubbles present, and larger bubble sizes;

4.5.3 Swelling

The swelling for several temperatures for $\dot{F} = 6{\cdot}10^{13}$ fsn/(s·m^3) and $f_d = 1$ is displayed in Figure 4.13. The swelling increases linearly for all temperatures, and at higher rates for higher temperatures. By examining the swelling rate as a function of time in Figure 4.14, the swelling rate dependence on temperature is clearly visible. Following a transition period very early in the simulation history, the swelling rate remains constant. This is the same behavior as the bubble size at

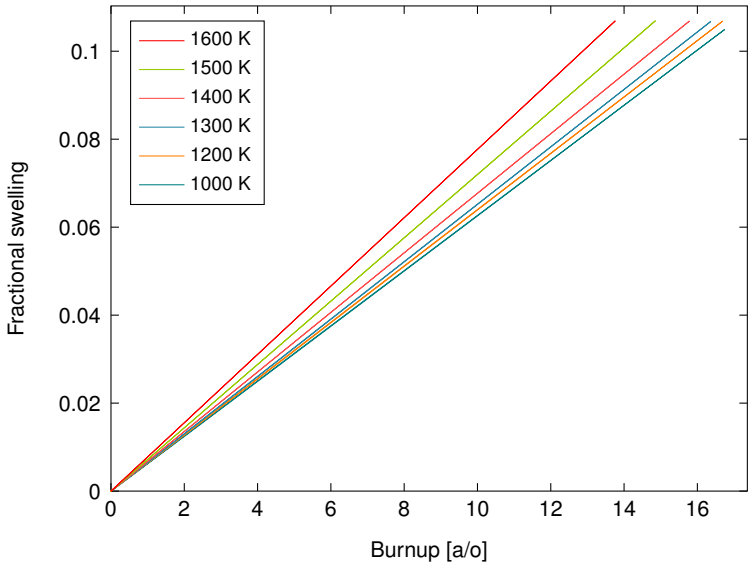

Figure 4.13: Swelling for different temperatures for $\dot{F} = 6{\cdot}10^{13}$ fsn/(s·m^3) and $f_b = 1$.

the peak concentration, as shown in Figure 4.11, linking the swelling behavior to the growth of the bubble size at the concentration peak.

By looking at the swelling for a given burnup slice from Figure 4.13, the relationship between temperature and swelling can be plotted in Figure 4.15. Although the swelling rate increases as a function of temperature, the swelling rate is so small that the plots remain nearly linear. By comparing the swelling to the experimentally derived swelling from Figure 2.5 (noting the change in scale), it is clear that the swelling calculated from the BUCK model does not capture the sharp "knee" transition seen in the data [9]

4.5.4 Comparison to Experimental Results

From the previous sections, it is clear that the currently implemented BUCK model does not capture the growth phenomena that leads to the large 100-400 nanometer P_2 bubbles, but rather may capture the behavior of the smaller 1-10 nanometer P_1 bubble population. P_2 bubbles have been extensively studied experimentally using scanning electron microscopes, while the smaller P_1 bubbles have had little attention due to requirement of the more difficult tunneling electron microscopy required for the nanometer sized bubbles. As such, comparison to experimental data is limited to a single study from 1984 [24].

The data in the study was limited to only a handful of temperature and burnup values, thus only qualitative comparison is achievable. Regardless, the overall trend is that the BUCK simulations here underestimate the size of the bubbles by nearly an order of magnitude. Similar to the behavior described in Section 4.4.4 in which the concentration lowered as the bubble distribution shifted to larger bubble sizes to accommodate the same number of gas atoms, so do these simulations over-

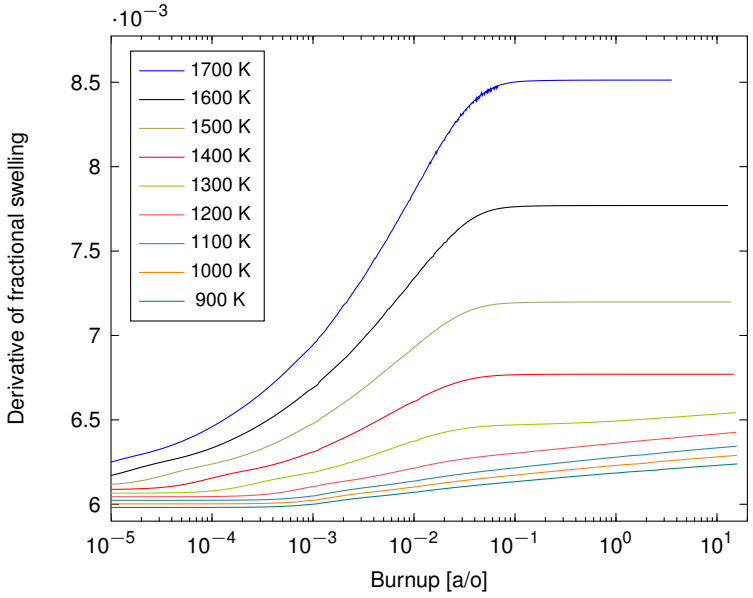

Figure 4.14: Derivative of swelling for different temperatures for $\dot{F} = 6{\cdot}10^{13}$ fsn/(s·m^3) and $f_b = 1$.

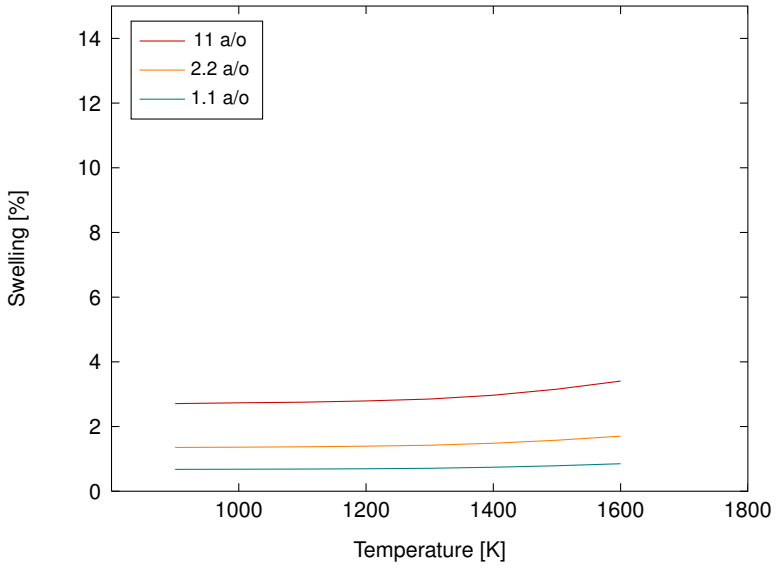

Figure 4.15: Total swelling as a function of temperature for $\dot{F} = 6{\cdot}10^{13}$ fsn/(s·m^3) and $f_b = 1$.

predict the bubble concentrations in comparison to the previous work.

4.5.5 Bubble Coalescence

One of the primary assumptions in BUCK is that the bubbles do not interact. As discussed in Section 4.2.2, the bubbles are assumed to remain stationary [9]. Although the bubbles can be expected to move slightly due to Brownian motion [17], directed diffusion is assumed to be non-existent, primarily due the high thermal conductivity leading to a minimal temperature gradient in uranium carbide [70].

Even if the bubbles do not move, they may interact if their concentration and/or size increase such that they can physically come in contact. The potential for bubble interaction can be estimated using some simple assumptions; Since bubbles starve the region around them of gas atoms, it can be expected that they will form roughly the same distance apart from one another. As a first approximation, it can be assumed that the bubbles nucleate in a cubic structure with with the volume occupied by each atom is equal to the inverse of the concentration C. The distance between the centers of adjoining bubbles can then be determined as,

$$a = C^{-1/3}. \tag{4.75}$$

If the bubbles have some radius R, then the space between the faces of the two bubbles can be calculated by,

$$d = a - 2R = C^{-1/3} - 2R. \tag{4.76}$$

Assuming that two bubbles combine as $d \to 0$, a plot of the concentration vs. radius given $d = 0$ can be plotted, as in Figure 4.16. Also included is the bubble concentration/size points at the peak of the concentration distributions from BUCK results plotted in Figure 4.8. It is clear that the bubble concentration is well below the threshold value that results in coalescence. For the 1600 K case, the distance between the faces of the bubbles is 43 nm, 27 times greater than the associated bubble radius.

While it can be expected that bubbles will coalesce as they grow, the comparison between the threshold line and the example points in Figure 4.8 show that any sort of interaction is absent in the BUCK simulations.

4.6 Conclusions

A new application, BUCK, was built using the MOOSE framework to simulate the fission gas bubble concentration distribution. In order to build a bare-bones foundation, the simplistic yet historically prevalent physics that can be used model fission gas bubble nucleation, growth, and knock-out were implemented as stepping stones until more advanced models for each physical process can be created. As the first step towards models that are based on first-principles, the re-solution parameter

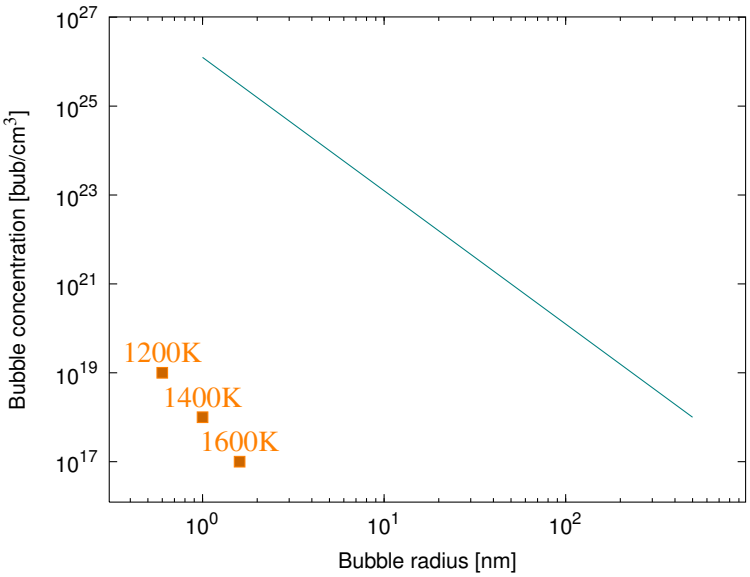

Figure 4.16: Threshold for bubble interaction with results from the bubble distribution at $\dot{F} = 6{\cdot}10^{13}$ fsn/(s·m^3) and $f_b = 1$.

calculated in Chapter 3 was included and tested within BUCK. In general, the currently implemented physics does not capture the rapid bubble growth that leads to the large fission gas bubbles experimentally measured [9]. The modular structure of BUCK will allow for easy implementation of more advanced models, either as refinements to the currently implemented physics, or new phenomena that is necessary to capture the large bubble growth. Further discussion of potential reasons for the larger bubbles and future work that may be implemented will be continued in Chapter 5.

CHAPTER 5: CONCLUSIONS AND FUTURE WORK

The need for cheap reliable energy, while simultaneously avoiding uranium supply constraints makes Gas Fast Reactors an attractive nuclear reactor design. Building on past experience, uranium carbide fuel may further enhance safety through superior energy transfer characteristics, and increase profitability through high uranium concentration. In order to qualify the fuel, an enhanced understanding of UC's behavior during operation is paramount. This requires fundamental understanding of basic material properties of the fuel. In particular, fission gas release and swelling are two complex and intertwined consequences of fission gas behavior that may lead to pellet-clad mechanical interaction and internal pressure buildup, both of which are prominent safety concerns.

Due to a reduced re-solution rate, uranium carbide suffers from a buildup of very large fission gas bubbles. While these bubbles serve to reduce total fission gas release through the trapping of diffusing gas atoms, they lead to high swelling and ultimately dominate the microstructure of the fuel.

In order to calculate the re-solution parameter in uranium carbide, the code 3DOT was created. As a result of the enhanced thermal dissipative properties of uranium carbide fuel, the atom-by-atom knockout process was shown to be an accurate representation of re-solution in uranium carbide. Furthermore, the Binary Collision Approximation was shown to appropriately model the re-solution event, bypassing computationally expensive Molecular Dynamics simulations. The code 3DOT was developed as an off-shoot of the code 3DTrim, both of which utilize the TRIM algorithm to calculate the kinematics of ions traveling through a material.

Benefiting from modern methods and enhanced computational power, the model created in 3DOT surpassed the complexity of previous historical calculations, resulting in a more fundamental understanding of the re-solution process in uranium carbide. A re-solution parameter was calculated that was an order of magnitude lower than previous studies suggested due to the inclusion of electronic losses in the current study. A decrease in the re-solution parameter as a function of radius occurred for low radii, with a nearly constant re-solution parameter for bubble radii above 50 nm. Through comparative studies on the re-solution parameter for various values of implantation energy and atomic density in the bubble, we found that while the re-solution parameter did change slightly, the overall shape did not.

A new application, BUCK, was built using the MOOSE framework to simulate the fission gas bubble concentration distribution. In order to build a foundation, the simplistic yet historically prevalent models that have been used to describe fission gas bubble nucleation, diffusion, and growth were implemented in BUCK as stepping stones until more advanced models for each physical process can be created. The re-solution parameter calculated from 3DOT simulations was included and tested within BUCK.

BUCK was tested against hand-calculations and various parameters, and behaved as expected. However, from studies using representative simulation parameters, it is clear that the currently implemented theory does not adequately identify the growth mechanism that leads to larger bubbles. While this currently limits the applicability of BUCK in a full fuel pin calculation, it provides the baseline structure in which new physics can be implemented, and represents an important step towards understanding the complex behavior of fission gas bubbles.

5.1 Future Work

The BUCK code presented here provides a baseline code in which newly developed models used to describe individual fission gas phenomena can be implemented. We incorporated a new model for re-solution in uranium carbide, which is only one piece of the complicated coupled phenomena, albeit the property that fundamentally separates uranium carbide from oxide fuels. Each model implemented in Section 4.2 relies on historic models and data for the physics that describes fission gas release that are ripe for modern analysis:

- Diffusion: Fission gas diffusion in all types of fuel has been notoriously difficult to measure, while simultaneously remaining a fundamental input to any gas model. Fortunately, recent advances in ab-initio studies have led to first-principles calculation of diffusivity in uranium oxide fuels at different stoichiometric compositions and irradiation conditions [87]. Similar techniques can be applied to uranium carbide fuels to determine gas diffusivity in UC. As discussed in Section 2.4, the diffusivity as a function of carbon/nitrogen radio and oxygen impurities leads to vastly different fission gas release measurements. Large-scale ab-initio studies that can capture this behavior may highlight the effect of nitrogen and oxygen on the general diffusivity of fission gas through an imperfect uranium carbide lattice.

- Nucleation: The nucleation process in uranium carbide has been largely uninvestigated, and is assumed to be a homogeneous process in a system where many heterogeneous mechanisms likely exist. Turnbull's theory of nucleation in the wake of fission gas bubbles seems a likely mechanism, but requires molecular dynamics simulations to determine the exact parameters, such as number and size of the bubbles produced, as well as gas saturation required before nucleation occurs [35]. Furthermore, dislocation tangles may act as nucleation sites, yet quantitative models that describe the interaction do not yet exist [17, 24].

- Growth: The absorption of single gas atoms in distributed sinks is reasonably vetted for large bubbles, however the transition between small clusters to bubbles would benefit from further investigation, as discussed in Section 4.2.4. Kinetic Monte Carlo (KMC) simulations may be more beneficial than typically used Molecular Dynamics simulations to model the diffusion dominated growth process [88].

- Point defects: As discussed in Section 4.2.5, the bubble is assumed always to be in equilibrium with the surrounding lattice. In actuality, the interaction of vacancies and interstitials with the bubble determines the size. By including a model that captures point defect behavior, BUCK may be able to capture the dynamic behavior of the bubble size [67]. Such a model will also be necessary to capture the potential symbiotic relationship between bubbles and other flaws.

- Defect symbiotic relationship: Micrographs captured by Ray and Blank showed that almost invariably, small P_1 bubbles interact with dislocation tangles, while the P_2 bubbles interact with larger precipitates [24]. Similar behavior has been capture in UO_2 studies [89]. The sharp growth rate that occurs in the swelling of uranium carbide may be described by some sort of symbiotic relationship between the defects and the bubbles. A 3-dimensional model that incorporates a point defect model may be able to quantify the relationship.

Beyond incorporation of more advanced models, several factors incorporated in BUCK can be improved. Perhaps most obvious, the re-solution parameter calculated in Chapter 3 may benefit from supporting studies using Molecular Dynamics. Although 3DOT simulations are less computationally expensive and therefore essential in order to calculate the re-solution parameter for a wide swath of simulation parameters, comparison to Molecular Dynamics simulations could support the re-solution parameter calculations.

Secondly, the BUCK framework could be improved to more easily support the large number of bubble sizes that need to simulated. As discussed briefly in Section 4.1, a handful of previous studies utilized a grouping scheme to reduce the total number of tracked variables. These studies failed to validate their simulations against non-grouped simulations, and preliminary studies that incorporated their grouping method within BUCK lead to vastly inaccurate time scales. Regardless, implementation of some sort of grouping method is necessary to reduce the computational expense associated with tracking the thousands of bubble sizes necessary to capture the large P_2 bubbles. This might be achieved through adaptive variable tracking, or some other sort of estimation of the bubble concentration distribution.

Ultimately, formulation of a complete fission gas model will allow calculation of the swelling and fission gas release for a uranium carbide fuel pin. The BUCK simulation can then be coupled to the thermo-mechanical fuel performance code BISON in which the material properties presented in Appendix A are implemented, to calculate the behavior of an entire pin during irradiation. These simulations can be verified using past experimental data [26, 27, 90, 91], and ultimately inform new simulations.

References

[1] C. Pasten, J. C. Santamarina, Energy and quality of life, Energy Policy 49 (2012) 468–476.

[2] EIA, International Energy Outlook 2013, Tech. Rep. DOE/EIA-0484, US Energy Information Administration (2013).

[3] X. Liu, A. Vedlitz, J. W. Stoutenborough, S. Robinson, Scientists' views and positions on global warming and climate change: A content analysis of congressional testimonies, Climatic Change.

[4] NEA, IAEA, Uranium 2011: Resources, Production and Demand, Tech. Rep. NEA No. 7059, NEA/IAEA (2007).

[5] H. Matzke, Science of Advanced LMFBR Fuels, North-Holland, 1986.

[6] A. E. Waltar, A. B. Reynolds, Fast Breeder Reactors, 1st Edition, Pergamon Press, 1981.

[7] R. B. Matthews, R. J. Herbst, Uranium-plutonium carbide fuel for fast breeder reactors, Nuclear Technology 63 (1) (1983) 9–22.

[8] R. W. Schleicher, T. C. Bertch, A. S. Shenoy, P. K. Gupta, K. R. Schultz, EM^2: An Innovative Approach to U.S. Energy Security and Nuclear Waste Disposition, in: Nuclear Power International 2010 Conference, no. May, 2010.

[9] H. Blank, Nonoxide ceramic nuclear fuels, in: Materials science and technology: a comprehensive treatment Vol 10a, Wiley-VCH, 1994.

[10] A. Booth, A method of calculating fission gas diffusion from UO_2 fuel and its application to the X-2-f loop test, Tech. Rep. CRDC-721, AECL (1957).

[11] R. White, M. Tucker, A new fission-gas release model, Journal of Nuclear Materials 118 (1983) 1–38.

[12] M. Speight, A calculation on the migration of fission gas in material exhibiting precipitation and re-solution of gas atoms under irradiation, Nuclear Science and Engineering 37 (1969) 180–185.

[13] G. Pastore, L. Luzzi, V. Di Marcello, P. Van Uffelen, Physics-based modelling of fission gas swelling and release in UO_2 applied to integral fuel rod analysis, Nuclear Engineering and Design 256 (2013) 75–86.

[14] K. Govers, C. Bishop, D. Parfitt, S. Lemehov, M. Verwerft, R. Grimes, Molecular dynamics study of Xe bubble re-solution in UO_2, Journal of Nuclear Materials 420 (2012) 282–290.

[15] C. Ronchi, P. Elton, Radiation re-solution of fission gas in uranium dioxide and carbide, Journal of Nuclear Materials 140 (1986) 228–244.

[16] M. Coquerelle, Survey of post-irradiation examinations made of mixed carbide fuels, Tech. rep. (1997).

[17] D. R. Olander, Fundamental Aspects of Nuclear Reactor Fuel Elements, Tech. Rep. TID-26711-P1, DOE (1976).

[18] IAEA, Thermophysical Properties of Materials for Nuclear Engineering: A Tutorial and Collection of Data, Vienna, Austria, 2008.

[19] H. Choi, R. Schleicher, Design Characteristics of the Energy Multiplier Module: Presentation from 2011 ANS Annual Meeting (2011).

[20] C. Ronchi, Extrapolated equation of state for rare gases at high temperatures and densities, Journal of Nuclear Materials 96 (1981) 314–328.

[21] K. Nogita, K. Une, High resolution TEM observation and density estimation of Xe bubbles in high burnup UO_2 fuels, Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms 141 (1-4) (1998) 481–486.

[22] P. Garcia, G. Martin, C. Sabathier, G. Carlot, A. Michel, P. Martin, B. Dorado, M. Freyss, M. Bertolus, R. Skorek, J. Noirot, L. Noirot, O. Kaitasov, S. Maillard, Nucleation and growth of intragranular defect and insoluble atom clusters in nuclear oxide fuels, Nuclear Instruments and Methods in Physics Research, Section B: Beam Interactions with Materials and Atoms 277 (2012) 98–108.

[23] P. Prajoto, A. Wazzan, D. Okrent, Computer Modeling of Steady State Fission Gas behavior in Carbide Fuels, Nuclear Engineering and Design 48 (1978) 461–495.

[24] I. Ray, H. Blank, Microstructure and fission gas bubbles in irradiated mixed carbide fuels at 2 to 11 a/o burnup, Journal of Nuclear Materials 124 (1984) 159–174.

[25] H. Blank, I. Ray, C. Walker, A coherent description of fission gas swelling in the structural zones III and IV of advanced fuels Part I: Properties of Bubble Populations P_2 and the Critical Temperature Burn-up Relation in Carbide Fuel Under the Conditions of Free Swelling, European Applied Research, Nuclear Science and Technology 4 (5) (1983) 1223–1274.

[26] M. Colin, M. Coquerelle, I. Ray, C. Ronchi, C. T. Walker, H. Blank, The Sodium-bonding pin concept for advanced fuels Part I: Swelling of carbide fuel up to 12% burnup, Nuclear Technology 63 (1983) 442–460.

[27] C. Ronchi, M. Campana, M. Coquerelle, J. V. de Laar, Reactor Performance of MC, MN, MCN and MCO: Results of the comparative irradiation experiments GOCAR, Tech. Rep. EUR 9186 EN (1984).

[28] C. Ronchi, I. Ray, H. Thiele, J. van de Laar, Measurements and observations on microscopic swelling in MX-type fuels, Tech. Rep. EUR 5907 EN, Institue for Transuranium elements (1978).

[29] G. Bart, F. Botta, C. Hoth, G. Ledergerber, R. Mason, R. Stratton, AC-3-irradiation test of sphere-pac and pellet (U, Pu)C fuel in the US Fast Flux Test Facility, Journal of Nuclear Materials 376 (1) (2008) 47–59.

[30] Philadelphia Electric Company, Final Hazards Summary Report Peach Bottom Atomic Power Station: Volume V - Plant Description and Safeguards Analysis (Annexes), Tech. rep. (1961).

[31] R. Martin, Compilation of Fuel Performance and Fission Product Transport Models and database for MHTGR Design, Tech. Rep. ORNL/NPR-91/6, ORNL (1993).

[32] J. Saurwein, Fission Product Vent System Presentation to OSU (Apr. 2014).

[33] B. Myers, N. Baldwin, W. Bell, R. Burnette, The behavior of fission product gases in HTGR Fuel Material, Tech. Rep. GA-A13723, GA (1977).

[34] D. R. Olander, D. Wongsawaeng, Re-solution of fission gas - A review: Part I. Intragranular bubbles, Journal of Nuclear Materials 354 (2006) 94–109.

[35] J. Turnbull, The distribution of intragranular fission gas bubbles in UO^2 during irradiation, Journal of Nuclear Materials 38 (2) (1971) 203–212.

[36] R. Nelson, The stability of gas bubbles in an irradiation environment, Journal of Nuclear Materials 31 (2) (1969) 153–161.

[37] G. Pastore, Modelling of Fission Gas Swelling and Release in Oxide Nuclear Fuel and Application to the TRANSURANUS Code, Ph.D. thesis (2012).

[38] H. Blank, H. Matzke, The effect of fission spikes on fission gas re-solution, Radiation Effects 17 (1973) 57–64.

[39] J. Turnbull, A review of irradiation induced re-solution in oxide fuels, Radiation Effects 53 (1980) 243–250.

[40] K. Govers, S. Lemehov, M. Verwerft, In-pile Xe diffusion coefficient in UO_2 determined from the modeling of intragranular bubble growth and destruction under irradiation, Journal of Nuclear Materials 374 (2008) 461–472.

[41] D. Schwen, M. Huang, P. Bellon, R. Averback, Molecular dynamics simulation of intragranular Xe bubble re-solution in UO_2, Journal of Nuclear Materials 392 (2009) 35–39.

[42] C. Ronchi, T. Wiss, Fission-fragment spikes in uranium dioxide, Journal of Applied Physics 92 (10) (2002) 5837.

[43] H. Blank, Properties of Fission Spikes in UO_2 and UC Due to Electronic Stopping Power, Physica Status Solidi (a) 10 (1972) 465.

[44] C. Ronchi, H. Matzke, J. van de Laar, H. Blank, Fission Gas Behaviour in Nuclear Fuels, European Applied Research Reports - Nuclear Science and Technology 1 (1) (1979) 1–350.

[45] J. Biersack, L. Haggmark, A Monte Carlo computer program for the transport of energetic ions in amorphous targets, Nuclear Instruments and Methods 174 (1-2) (1980) 257–269.

[46] J. F. Ziegler, J. Biersack, U. Littmark, The stopping and range of ions in solids: Volume 1, Pergamon, New York, 1985.

[47] D. Schwen, R. Averback, Intragranular Xe bubble population evolution in UO_2: A first passage Monte Carlo simulation approach, Journal of Nuclear Materials 402 (2010) 116–123.

[48] J. Ziegler, J. Biersack, M. Ziegler, SRIM: The stopping and Range of Ions in Matter, SRIM Company, 2008.

[49] W. Brandt, M. Kitagawa, Effective stopping-power charges of swift ions in condensed matter, Physical Review B 25 (9).

[50] C. Matthews, 3DOT Source.
URL: `https://github.com/tophmatthews/3DOT`

[51] D. Schwen, 3DTrim Source.
URL: `http://groups.mrl.illinois.edu/averback/3dtrim/`

[52] D. Parfitt, R. Grimes, Predicting the probability for fission gas resolution into uranium dioxide, Journal of Nuclear Materials 392 (2009) 28–34.

[53] H. Blank, Fabrication of carbide and nitride pellets and the nitride irradiations of Niloc 1 and Niloc 2, Tech. Rep. EUR 13220 EN, Institute for Transuranium Elements (1991).

[54] M. Posselt, Comparison of BC and MD simulations of low-energy ion implantation 102 (1995) 236–241.

[55] V. Ferleger, I. Wojciechowski, On non-binary nature of the collisions of heavy hyperthermal particles with solid surfaces, Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms 164-165 (2000) 641–644.

[56] G. Hobler, G. Betz, On the useful range of application of molecular dynamics simulations in the recoil interaction approximation, Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms 180 (2001) 203–208.

[57] I. Chakarov, R. Webb, An investigation of collision propagation in energetic ion initiated cascades in copper, Nuclear Instruments and Methods in Physics Research B 102 (1995) 145–150.

[58] G. S. Was, Fundamentals of radiation materials science: Metals and alloys, Springer Berlin Heidelberg, 2007.

[59] D. Gaston, C. Newman, G. Hansen, D. Lebrun-Grandié, MOOSE: A parallel computational framework for coupled systems of nonlinear equations, Nuclear Engineering and Design 239 (10) (2009) 1768–1778.

[60] L. Noirot, MARGARET: A comprehensive code for the description of fission gas behavior, Nuclear Engineering and Design 241 (6) (2011) 2099–2118.

[61] M. Wood, J. Matthews, A simple operational gas release and swelling model, Journal of Nuclear Materials (1980) 35–40.

[62] C. Ronchi, J. van de Laar, H. Blank, The sodium-bonding pin concept for advanced fuels. Part III: Calculation of the swelling performance, Nuclear Technology 68 (1984) 48–65.

[63] M. Hayns, On the group method for the approximate solution of a hierarchy of rate equations describing nucleation and growth kinetics, Journal of Nuclear Materials 59 (1976) 175–182.

[64] M. Hayns, R. Bullough, The Nucleation and Growth of Fission Gas Bubbles, Tech. Rep. IAEA-SM-190/15 (1975).

[65] M. Hayns, M. Wood, On the rate theory model for fission-gas behaviour in nuclear fuels, Journal of Nuclear Materials 59 (3) (1976) 293–302.

[66] M. Hayns, M. Wood, Models of fission gas behaviour in fast reactor fuels under steady state and transient conditions, Journal of Nuclear Materials 67 (1977) 155–170.

[67] J. Griesmeyer, N. Ghoniem, D. Okrent, A dynamic intragranular fission gas behavior model, Nuclear Engineering and Design 55 (1979) 69–95.

[68] B. Eyre, R. Bullough, The formation and behaviour of gas bubbles in a non-uniform temperature environment, Journal of Nuclear Materials 26 (3) (1968) 249–266.

[69] A. Madrid, An Intermediate Model on Intragranular Fission Gas Behavior During Steady State Irradiation of LMFBR Uranium Carbide Nuclear Fuel, Ph.D. thesis, Unicersity of California, Los Angeles (1980).

[70] R. Weeks, R. Scattergood, S. Pati, Migration velocities of bubble-defect configurations in nuclear fuels, Journal of Nuclear Materials 36 (2) (1970) 223–229.

[71] M. Freyss, First-principles study of uranium carbide: Accommodation of point defects and of helium, xenon, and oxygen impurities, Physical Review B 81 (1) (2010) 014101.

[72] R. Ducher, R. Dubourg, M. Barrachin, A. Pasturel, First-principles study of defect behavior in irradiated uranium monocarbide, Physical Review B 83 (10) (2011) 1–12.

[73] D. A. Porter, K. E. Easterling, M. Y. Sherif, Phase Transformations in Metals and Alloys, CRC Press, 2009.

[74] J. Rest, An improved model for fission product behavior in nuclear fuel under normal and accident conditions, Journal of Nuclear Materials 120 (2-3) (1984) 195–212.

[75] J. Griesmeyer, N. Ghoniem, The response of fission gas bubbles to the dynamic behavior of point defects, Journal of Nuclear Materials 80 (1979) 88–101.

[76] A. Wazzan, J. Tatsumi, D. Okrent, M. Billone, Effect of Non-equilibrium fission gas and fuel creep on swelling and release in irradiated carbide fuels, Nuclear Engineering and Design 88 (1985) 93–101.

[77] B. S. Kirk, J. W. Peterson, R. H. Stogner, G. F. Carey, libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations, Engineering with Computers 22 (3-4) (2006) 237–254.

[78] J. Hales, S. Novascone, B. Spencer, R. Williamson, G. Pastore, D. Perez, Verification of the BISON fuel performance code, Annals of Nuclear Energy 71 (2014) 81–90.

[79] M. R. Tonks, D. Gaston, P. C. Millett, D. Andrs, P. Talbot, An object-oriented finite element framework for multiphysics phase field simulations, Computational Materials Science 51 (1) (2012) 20–29.

[80] M. Tonks, D. Gaston, C. Permann, P. Millett, G. Hansen, D. Wolf, A coupling methodology for mesoscale-informed nuclear fuel performance codes, Nuclear Engineering and Design 240 (10) (2010) 2877–2883.

[81] M. R. Tonks, P. C. Millett, P. Nerikar, S. Du, D. Andersson, C. R. Stanek, D. Gaston, D. Andrs, R. Williamson, Multiscale development of a fission gas thermal conductivity model: Coupling atomic, meso and continuum level simulations, Journal of Nuclear Materials 440 (1-3) (2013) 193–200.

[82] D. A. Knoll, D. E. Keyes, Jacobian-free Newton-Krylov methods: A survey of approaches and applications (2004).

[83] J. Farlow, J. E. Hall, J. M. McDill, B. H. West, Differential Equations and Linear Algebra, Prentice-Hall, Upper Saddle River, New Jersey, 2002.

[84] Y. Saad, M. H. Schultz, GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems, SIAM Journal on Scientific and Statistical Computing 7 (3) (1986) 856–869.

[85] S. Eckert, H. Baaser, D. Gross, O. Scherf, A BDF2 integration method with step size control for elasto-plasticity, Computational Mechanics 34 (5) (2004) 377–386.

[86] Idaho National Laboratory, MOOSE Workshop (2015).

[87] D. a. Andersson, P. Garcia, X. Y. Liu, G. Pastore, M. Tonks, P. Millett, B. Dorado, D. R. Gaston, D. Andrs, R. L. Williamson, R. C. Martineau, B. P. Uberuaga, C. R. Stanek, Atomistic modeling of intrinsic and radiation-enhanced fission gas (Xe) diffusion in $UO_{2\pm x}$: Implications for nuclear fuel performance modeling, Journal of Nuclear Materials 451 (1-3) (2014) 225–242.

[88] M. Stan, Multi-Scale Models and Simulations of Nuclear Fuels, Nuclear Engineering and Technology 41 (1) (2009) 39–52.

[89] S. Kashibe, K. Une, K. Nogita, Formation and growth of intragranular fission gas bubbles in UO_2 fuels with burnup of 6-83 GWd/t, Journal of Nuclear Materials 206 (1) (1993) 22–34.

[90] C. Ronchi, I. Ray, H. Thiele, J. V. de Laar, Swelling analysis of highly-rated MX-type LMFBR fuels: II. Microsopic swelling behaviour, Journal of Nuclear Materials 74 (1978) 193–211.

[91] C. Ronchi, C. Sari, Swelling analysis of highly rated MX-type LMFBR fuels. I. Restructuring and Porosity Behaviour, Journal of Nuclear Materials 58 (2) (1975) 140–152.

[92] T. Preusser, Modeling of Carbide Fuel Rods, Nuclear Technology 57 (1982) 343–371.

[93] P. Petkevich, Development and Application of an Advanced Fuel Model for the Safety Analysis of the Generation IV Gas-cooled Fast Reactor, Ph.D. thesis, Ecole Polytechnique Federale de Lausanne (2008).

[94] IAEA, Status and Trends of Nuclear Fuels Technology for Sodium Cooled Fast Reactors, Tech. rep., IAEA, Vienna, Austria (2009).

[95] M. Tetenbaum, A. Sheth, W. Olson, A Review of the Thermodynamics of the U-C, Pu-C, and U-Pu-C Systems, Tech. Rep. ANL-AFP-8, ANL (1975).

[96] C. Holley, Jr., M. Rand, E. Storms, The Actinide Carbides, in: Chemical Thermodynamics, IAEA, 1984, Ch. 6.

[97] S. Govindarajan, P. Puthiyavinayagam, S. Clement Ravi Chandar, S. Chetal, S. Bhoje, Performance of FBTR Mixed Carbide Fuel, in: Influence of high dose irradiation on core structural and fuel materials in advanced reactors, 1997, pp. 47–56.

[98] E. Storms, An Analytical Representation of the Thermal Conductivity and Electrial Resistivity of UC, PuC, and (UPu)C, Tech. Rep. LA-9524, LANL (1982).

[99] P. Petkevich, K. Mikityuk, P. Coddington, R. Chawla, Development and benchmarking of a 2D transient thermal model for GFR plate-type fuel, Annals of Nuclear Energy 34 (9) (2007) 707–718.

[100] J. Matthews, Mechanical Properties and Diffusion Data for Carbide and Oxide Fuels, Tech. rep., UK Atomic Energy Authority (1974).

[101] W. Dienst, Swelling, Densification and Creep of (U, Pu)C Fuel under irradiation, Journal of Nuclear Materials 124 (1984) 153–158.

APPENDICES

APPENDIX A: MATERIAL PROPERTIES

The following attempts to collect material data for uranium monocarbide fuels. However, data information will be included for $(U_w,Pu_x)_yC_z$ for various values of w, x, y, and z, when it is available. All combinations will now be labeled generically as MC fuels.

The primary resources utilized for this Database are as follows:

- H. Matzke, Science of Advanced LMFBR Fuels. (1986) [5]: This extensive monograph touches on all topics of MC fuels up to the publishing date. Matzke himself contributes his own data to most topics. The book emphasizes the spread of sparse material data, and tries to give recommendations on the best practice, although it does not make explicit recommendations;

- H. Blank, "Nonoxide ceramic nuclear fuels," in Materials science and technology: a comprehensive treatment Vol 10a, (1994) [9]: This summary is extensive in the material data up to the publishing date. It contains much of the same data as Matzke;

- T. Preusser, "Modeling of carbide fuel rods," Nuclear Technology, vol. 57, pp. 343-371, 1982 [92]: This article contains the relations used in the fuel performance code URANUS. A review of thermo-mechanical properties of UC and (U,Pu)C is used to backup the choices for the program. Also included is the physical model used for the fission gas release module in URANUS. Due to the complexity of the material data, much of the URANUS relations are simplistic;

- P. Petkevich, "Development and Application of an Advanced Fuel Model for the Safety Analysis of the Generation IV Gas-cooled Fast Reactor," (2008) [93]: This dissertation contains material properties compiled from several sources. The thermomechanical properties are taken from generic material property sources rather than MC specific sources, however hard to find swelling and creep relations are included.

A.1 Density

The density of MC from Matzke are displayed in Table A.1. One interesting note is that ($U_{0.8}$, $Pu_{0.2}$)C has a lower density than both UC and PuC.

A.2 Melting Temperature

The melting points for several MC compounds are listed in Table A.2.

Table A.1: Density for MC from several sources.

Compound	Density kg/m^3	
	From Matzke [5]	*Other values*
UC	13,630	13,630 [92, 93]
UC_2	11,680	
U_2C_3	12,880	
PuC	13,600	
Pu_2C_3	12,700	
Pu_3C_2	15,300	
$(U_{0.8}, Pu_{0.2})C$	13,580	13,600 [92]

Table A.2: Melting points for several MC species [94].

Compound	Temperature [K]
UC	2780 ± 25
PuC	1875 ± 25
U_2C_3	2100
Pu_2C_3	2285
$(U_{0.8}Pu_{0.2})C$	2750 ± 30
$(U_{0.8}Pu_{0.2})_2C_3$	2480 ± 50

A.3 Specific Heat Capacity

Although steady state calculations are relatively insensitive to the heat capacity, accurate relations for C_p may be necessary for transient models. The heat capacity is dependent primarily on temperature [92]. Blank claims the first studies of MC heat capacity did not take into account a strong upward trend in C_p at temperatures above 60% of the melting temperature [9]. This incorrect data was reproduced in the ANL collection on thermodynamic data for MC fuels [95]. Blank believes the most accurate assessment of the heat capacity was completed by Holley in 1984 [96]. The heat capacity for MC is typically cast into a form as,

$$c_p(T) = a + bT + cT^2 + dT^3 + eT^{-2}, \tag{A.1}$$

C_p = heat capacity [J/mol/K],
T = temperature [K].

Since simple relations are adequate for steady-state calculations, Preusser uses a linear trend for the heat capacity of UC given as,

$$C_p(T) = 54.4 + (0.00962)T. \tag{A.2}$$

The heat capacities for uranium carbides are plotted in Figure A.1, with plutonium carbides plotted in Figure A.2.

Table A.3: Coefficients for the heat capacities for several MC species.

Fuel	a	b	c	d	e	Trange (K)	Ref.
UC	50.98	2.57E-02	-1.87E-05	5.72E-09	-6.19E+05	298-2780	[2]
$UC_{1.5}$	5.354	-2.39E-02	2.07E-05	0	-1.45E+06	298-1670	[2]
UC_2	49	8.25E-02	-7.82E-05	3.03E-08	-5.93E+05	298-2000	[1]
U_2C_3	150.7	-4.79E-02	4.13E-05	0	-2.91E+05	360-1700	[1]
$PuC_{0.84}$	57.876	-1.45E-02	7.71E-06	8.62E-09	-6.55E+05	298-1875	[2]
$PuC_{1.5}$	78.0375	-4.00E-02	3.52E-05	0	-1.09E+06	298-2285	[2]

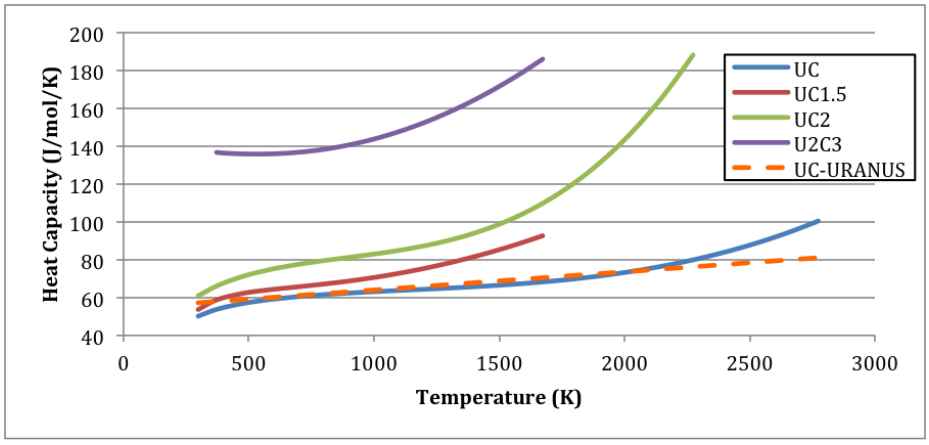

Figure A.1: Heat capacities for several uranium carbide compounds.

As the M/C ratio decreases, the heat capacity tends to increase. The largest deviation occurs in the U_2C_3 composition, resulting in nearly a factor of 2 increase. At 1100 K, the transition occurs for hyperstoichiometric fuels as (UC+C) transitions into (U_2C_3 + C) this phase change is visible in the heat capacity curves $UC_{1.5}$ and especially UC_2 transitions from following the UC heat capacity curve, to following the slope of U_2C_3.

The heat capacities given by Blank will be utilized as the standard. Deviation in stoichiometry and plutonium content will be ignored for the present. A relationship based on M/C or U/Pu ratio may be possible, perhaps even as simple as a linear relation.

A.4 Thermal Conductivity

The thermal conductivity is heavily dependent on temperature, porosity, and stoichiometry of MC. Any imperfections, such as Pu content, fission product, and deviations in stoichiometry serve to decrease the thermal conductivity [92]. Preusser uses the simple relation [92],

$$\lambda = \begin{cases} 20, & \text{for } T \leq 773 \text{ K}, \\ 20, + (1.3 \cdot 10^{-3}) T & \text{for } T > 773 \text{ K}, \end{cases} \tag{A.3}$$

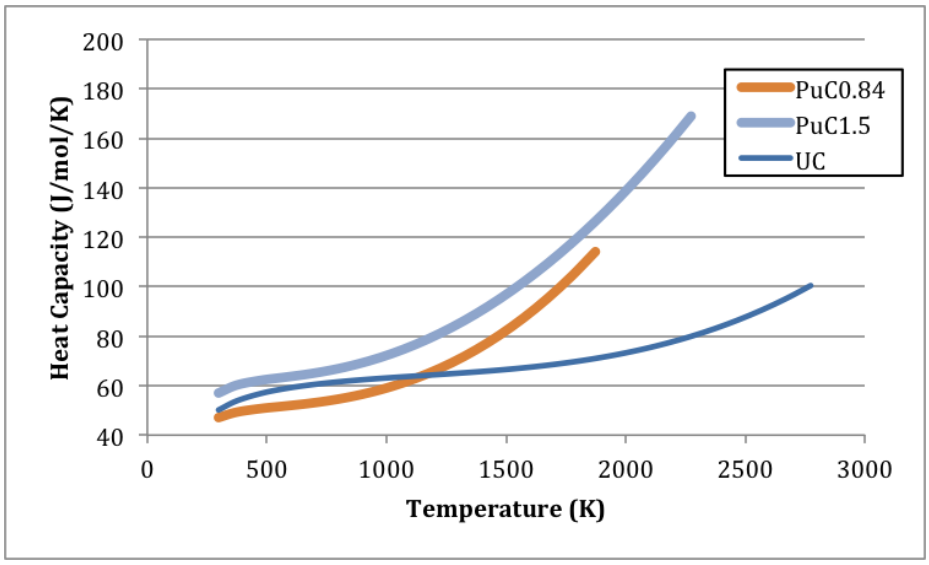

Figure A.2: Heat capacities for several plutonium carbide compounds.

λ = thermal conductivity [W/m/K],
T = temperature [K].

Porosity corrections are typically given in the form:

$$\phi = \phi'(1-P)^n, \tag{A.4}$$

ϕ = Property of sub-dense material,
ϕ' = Property of fully-dense material,
P = Fractional porosity,
n = correction factor.

The most basic approximation is for n=1, and usually applies to material in which the microstructural information such as the shape and orientation of the pores is unknown. Both $n = 3/2$ and a second type of correction,

$$\phi = \phi'\left(\frac{1-P}{1+P}\right) \tag{A.5}$$

have been used to for (U, Pu)C specifically [97].

Thermal conductivity is heavily dependent on C/U ratio, especially when there is excess uranium. If the fuel is slightly hypo-stoichiometric, the extra uranium atoms will form a metallic phase along the grain boundaries, increasing the thermal conductivity. However, the extra carbon atoms incorporated in the crystal in hyper-stoichiometric fuels will result in a decreased thermal conductivity. Note, UC fuels are typically kept at hyper-stoichiometric levels to avoid the swelling issues caused by free uranium metal.

By taking into account the C/M ratio, Storms [98] was able to isolate the effect of the U/Pu ratio on the thermal conductivity. For hyper-stoichiometric fuel, a plutonium ratio of 20% results in a very slight decrease in λ. Increasing the Pu ratio to 40% markedly decreases the conductivity. However, hypo-stochiometric fuel experiences decrease in thermal conductivity of about 20% up to 1000° C, at which point the UC and $(U_{0.8},Pu_{0.2})C$ have similar thermal conductivities.

Blank includes a summary on studies on the thermal conductivity of $(U_{0.8}Pu_{0.2})C_{1+x}$ with artificially mixed fission products Ce, Zr, and Mo. Table A.4 displays the ratio of the thermal conductivities between the pure and FP mixed samples, λ/λ_{mod}.

Table A.4: Thermal conductivity ratios due to artificial additions of fission products. The change in the ratio is nearly constant for 5-10% molar introduction of MoC. Reproduced from [9].

Species	Atomic radius [μm]	Molar addition	λ/λ_{mod} at 700 K	λ/λ_{mod} at 1200 K
U	175	-	-	-
Ce	185	10%	1.15	1.087
Zr	155	10%	1.21	1.16
Mo	145	5%	1.25	1.186

Since the decrease in thermal conductivity is higher at lower temperatures, it can be expected that the outer ring of the fuel pellet will experience a greater change in thermal conductivity as a function of fission product addition.

From Table A.4, it is clear that the addition of the Ce results in the lowest modification in the thermal conductivity, while Mo results in the highest. By plotting λ/λ_{mod} by the percentage difference in atomic radius of each FP with uranium, a nearly linear trend is obvious (Figure A.3).

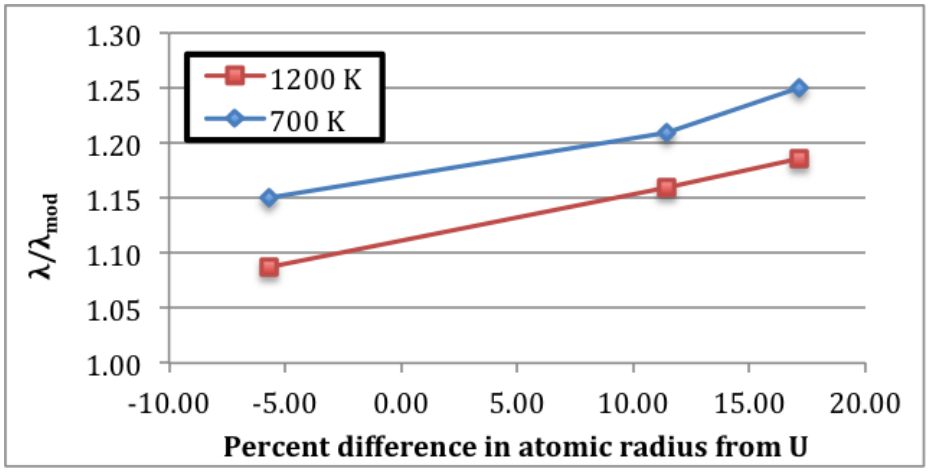

Figure A.3: λ/λ_{mod} vs. percent difference between the atomic radius of a given FP and U.

Although a trend might be drawn from the above, Matzke claims that even though the thermal conductivity has a slight dependence on the FP concentration, it is a smaller effect when compared to porosity, C/M ratio, and restructuring [5].

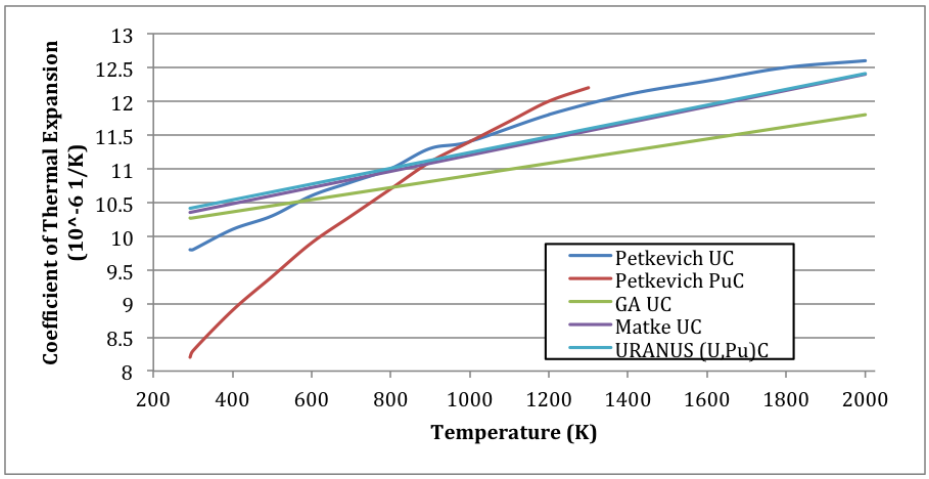

Figure A.4: Thermal expansion [5, 92, 99].

From the previous discussion, we can see that the thermal conductivity varies as a function of C/M and U/Pu ratios. What was not discussed was the even wider deviations between thermal conductivity reports. Matzke attributes the difficulty in determining an accurate value of λ (and many other physical properties) on the poor characterization of the material before measurement.

A.5 Thermal Expansion

Most thermal expansion relations are given in the linear form:

$$\lambda_m = a + bT, \tag{A.6}$$

λ_m = mean thermal expansion coefficient, [1/K],
T = temperature, [K].
Values for a and b are displayed in Table A.5.

Table A.5: Coefficients for thermal expansion.

Reference	a	b
General Atomics	1E-5	0.9E-9
Matzke [5]	1E-5	1.2E-9
Preusser [92]	1.007E-5	1.17E-9

Preusser claims that the above relation utilized in URANUS is highly representative of the values in literature, as well as conforming well to experimental data [92].

The coefficient of thermal expansion is independent of the stoichiometry up to 800° C [5]. Above this temperature, hypo-stochiometric UC behaves differently than stoichiometric UC due to the phase changes of the free uranium present at the grain boundaries [92]. The decomposi-

tion temperature $U_2C_3 \rightarrow$ UC + UC_2 occurs at 1780° C, and is accompanied with a large volume change [5]. Differences in the thermal expansion between UC and U_2C_3 should lead to noticeable internal stresses in two-phase structures [9].

Matzke claims that the U/Pu mixture does not exhibit great deviations, and the URANUS code utilizes the same coefficient of thermal expansion for both UC and PuC.

In a thorough explanation of thermal expansion data, Blank claims that in principle, the porosity does not affect the thermal expansion, with data from dense arc-melted UC producing nearly the same results as 86% sub-dense hot-pressed UC.

A.6 Young's Modulus

Table A.6 displays the elastic moduli at room-temperature.

Table A.6: Elastic moduli at room-temperature [5].

Material	Young's [10^6 Pa]	Shear [10^6 Pa]	Bulk [10^6 Pa]	Poisson's
UC, $(U_{0.85}Pu_{0.15})C$	225	87.3	176.8	0.228
$UC_1$0.96	211	82.9	157.5	0.28
UC	210	81.3	167.8	0.291
$(U_{0.8}Pu_{0.2})C$	202	78.5	160.3	0.29

Preusser implements a temperature and porosity dependent thermal conductivity of the form,

$$E(T,p) = 2.15 \cdot 10^{11} \left(1 - a \cdot P\right) \left(1 - 0.92x10^{-4} \left[T - 273\right]\right), \tag{A.7}$$

E = Young's modulus, [Pa],
P = fractional porosity, $P \leq 0.3$,
T = temperature, [K],
a = porosity correction coefficient.

A porosity correction coefficient of a = 2.3 is typically used [5, 92], with a value of a = 1.54 for fuel with 20% Pu. The porosity correction does not consider the shape of the pores which may be important for highly porous material.

Matthews [100] claims differences in stoichiometry and grain radius does not significantly modify Young's Modulus. Matzke references several studies of Young's Modulus after the introduction of several fission products, and claims they have little affect [5].

A.7 Poisson's Ratio

Preusser [92] utilizes a porosity corrected υ:

$$\upsilon = 0.288 - 0.286P, \quad \text{for } 0.05 \leq P \leq 0.27. \tag{A.8}$$

Preusser acknowledges other papers use fixed values, namely the first value given in Table A.6. However, since the URANUS code utilizes values for Young's modulus and Poisson's ratio from the same resource, Preusser recommends the above formulation. Matzke claims that Poisson's ratio does not change as a function of temperature due to similar temperature dependencies for the moduli [5].

A.8 Thermal Creep

For fuel rod analysis, only primary and secondary creep is applicable as tertiary creep regions lie out of the time span of the rod lifetime. However, due to the complexity of primary creep, typically only secondary creep is modeled. In general, thermal creep begins at 1300 [K], and dominates radiation creep, although both are important to reducing stress in the fuel [92].

Preusser claims that the following relationship is the most applicable for thermal creep:

$$\dot{\varepsilon}_{cr}^{th} = c_{th} \cdot \sigma^{2.44} \exp(-Q/RT) \tag{A.9}$$

$\dot{\varepsilon}_{cr}^{th}$ = thermal creep rate, [1/s],
c_{th} = coefficient, $c_{th} = 9.48 \cdot 10^{-9}$
σ = effective stress, [Pa],
Q = activation energy, $Q = 5.255 \cdot 10^5$ [J/mol],
R = ideal gas constant, 8.314 [J/mol· K],
T = temperature, [K].

The above equation only accounts for temperature and pressure dependence. The grain size will also affect creep since the speed of grain boundary sliding increase as grain size decreases. This results in different creep rates between coarse-grained arc-melted fuel and fine-grained sintered fuel.

The effect of impurities can be considerable to the creep rate. This is especially true with the addition of the sintering aid nickel, with an Ni addition to 0.05% resulting in 2-3 orders of magnitude increase in the creep rate. Similar behavior is observable with higher concentrations of carbon, however, additions of Pu, Zr, or O have little effect. Free mixed metal (U,Pu) in hypostoichiometric fuel can result in phases with low melting point eutectics that can lead to low creep resistance [92].

A.9 Irradiation Creep

As a result of the relatively low irradiation temperatures of carbide fuel, irradiation creep is more significant than similar behavior in oxide fuels where thermal creep dominates. Regardless, irradiation creep behavior is similar in most nuclear fuels, most likely due to the linear relationship between creep and the number of point defects resulting from neutron dose. Preusser utilizes the

following relationship for irradiation creep:

$$\dot{\varepsilon}_{cr}^{irr} = c_{irr}\dot{F}\sigma \tag{A.10}$$

$\dot{\varepsilon}_{cr}^{irr}$ = irradiation creep rate, [1/s],
c_{irr} = coefficient, $1 \cdot 10^{-37}$ [1/Pa·s·$\dot{F}$],
σ = effective stress, [Pa],
$\dot{F}$ = fission rate density, [fsn/m^3·s].

A.10 Irradiation Induced Swelling and Densification

Since the swelling of uranium carbide fuel is based on the total number of fissions, it can be approximated using a stepwise approach with the burnup, B:

$$V(B+\Delta B) = (1+V(B))\left[\frac{\partial(\Delta V/V)}{\partial B}\right]\Delta B. \tag{A.11}$$

In this way, the swelling rate can be defined for given timestep. Although an approximation, the above equation is valid if the volume increase is small relative the the volume, which can be controlled by the timestep size.

A.10.1 Solid Fission Product Swelling

Most authors quote a swelling rate of around 0.5% per atom percent burnup. This is derived from a very simplistic model: Given 100 fission events, 200 fission products are created. 50 of these fission products will be volatile and will not contribute to the solid swelling factor. 100 of the remaining 150 atoms can be assumed to occupy the original actinide location, thus 50 atoms create a new site in the lattice. If these atoms are assumed to take the same space as the actinide atoms, then each fission will cause the bulk to swell by the 0.5 per fission per initial metal (U, Pu) atom (FIMA).

Preusser uses the following relationship for swelling due to solid fission products [92]:

$$\frac{\partial(\Delta V/V)_{ss}}{\partial B} = c_{ss}, \tag{A.12}$$

$\Delta(\Delta V/V)_{ss}$ = fractional volume change,
c_{ss} = solid swelling factor, 0.417,
ΔB = change in burnup over timestep, [FIMA].

Although the above solid swelling factor is slightly less than the typical 0.5, the lower value is included with the same gaseous swelling formulation presented in [92].

A.10.2 Gaseous Fission Product Swelling

The swelling due to gaseous fission products is one of the most complex behaviors in mixed carbide fuels. As discussed in Section 2.1, the bubble behavior in UC fuels is very complex and highly dependent on temperature, burnup, impurities, and microstructure.

The swelling due to gaseous fission products can be divided into two contributions. The first is the inclusion of single gas atoms which mimics the behavior of solid fission product swelling. At low temperatures when atomic diffusion is nearly absent, the gas atoms will remain isolated as point defects. However, at high temperatures, the gas atoms will tend to group together and form bubbles. These bubbles themselves can interconnect and form larger bubbles, resulting in a less efficient use of space (Section 2.1). As a result, the bulk fuel will swell as the concentration and sizes of bubbles increase.

Ideally, the swelling caused by gaseous fission products would be calculated using a bubble concentration distribution function. However, the complexity of such a model has typically forced empirical relationships to be used to describe the swelling behavior.

Preusser uses a volume swelling model that accounts for porosity, temperature, burnup, and contact pressure:

$$\frac{\partial(\Delta V/V)_{gs}}{\partial B} = \begin{cases} c_{gs} f_{por} f_{cont}, & \text{for } T \leq 973 \text{ K}, \\ (c_{gs} + f_{temp} f_{Bu}) f_{por} f_{cont}, & \text{for } T > 973 \text{ K}, \end{cases} \tag{A.13}$$

$$\frac{\partial(\Delta V/V)_{gs}}{\partial B} \leq 3.653, \qquad \text{for all } T, \tag{A.14a}$$

$$f_{por} = \exp(0.04 - P), \qquad \text{for } P > 0.04, \tag{A.14b}$$

$$f_{cont} = \exp\left(-\frac{pc}{pc_0} b\right), \qquad \text{for } f_{cont} \leq 1, \tag{A.14c}$$

$$f_{temp} = 12.95 - (0.0281)T + (1.520 \cdot 10^{-5})T^2, \tag{A.14d}$$

$$f_{Bu} = \frac{B}{B_0} - a, \qquad \text{for } f_{Bu} \geq 0, \tag{A.14e}$$

$\Delta(\Delta V/V)_{gs}$ = fractional volume change,
c_{ss} = gaseous swelling factor, $c_{ss} = 1.528$,
f_{por} = porosity correction,
f_{cont} = contact pressure correction,
f_{temp} = temperature correction,
f_{Bu} = burn correction,
P = fractional porosity,
pc = cladding contact pressure, [Pa]
pc_0 = constant, $pc_0 = 1 \cdot 10^6$ [Pa],
T = temperature, [K],
B = burnup, [FIMA],
B_0 = threshold burnup, $B_0 = 0.0112$ [FIMA],
a = model fitting parameter, $a = 2.0$,
b = model fitting parameter, $b = 0.1$.

The model is Equation A.13 combines the complexity of the gas bubble distribution into one experimentally derived equation. At low temperatures, the swelling rate dependent only on the porosity of the fuel and contact pressure with the cladding. At high temperatures, the swelling rate is much higher, simulating the run-away swelling above the threshold temperature, similar to the behavior discussed in section 2.1. The high temperature growth is also limited by the burnup correction, and is constrained only above the threshold burnup value B_0. The model parameters a and b were used to tune the model the the experimental data, resulting in a correlation that is somewhat limited.

A.10.3 Densification

Densification of the fuel occurs as a result of the high temperatures and irradiation environment in fuel rods. Carbide fuel pins have been observed to have an approximate volume fractional densification rate of $\Delta V/V = 0.67[\text{FIMA}]^{-1}$, with a maximum densification of about 90% of the theoretical density (TD). Creep tests on 85% TD $(U_{0.85}Pu_{0.15})C$ pellets showed a densification behavior of [101],

$$\Delta P = \Delta P_{max}[1 - \exp(-B/B_c)] \tag{A.15}$$

ΔP = porosity decrease,
ΔP_{max} = maximum porosity decrease, $\Delta P_{max} = -0.034$,
B = burnup, [FIMA],
B_c = burnup constant, $B_c = 0.006$ [FIMA].

APPENDIX B: INPUT FILE

Listing B.1: example.i

```
[GlobalParams]
  temp = temp
  fission_rate = fission_rate
  N = 100
  s = 100
  block = 0
  allow_loss = true
[]


[Mesh]
  type = GeneratedMesh
  dim = 1
[]

[Bubbles]
  [./Conc]
    c1_initial_condition = 0
    c2_initial_condition = 0
    initial_condition = 0
  [../]
  [./Knockout]
    factor = 1
  [../]
  [./Growth]
  [../]
  [./Rad]
    [./Eq]
    [../]
  [../]
  [./PPs]
    fission_rate = fission_rate
    concentrations = 'csv'
    total_atoms = 'console'
    total_concentrations = 'none'
    swelling = 'swelling_csv'
    total_swelling = 'console csv'
    c1_loss = 'c1_csv'
    gain_rate = 'gain_csv'
    knockout_rate = 'knockout_csv'
  [../]
  [./Nucleation]
  [../]
  [./Dampers]
    damping = 0.01
    starting_index = 10
  [../]
[]


[Kernels]
  [./fg_source]
    type = VariableScaledSource
    variable = c001
    scaling_variable = fission_rate
    factor = 0.25
```

```
  [../]
[]

[AuxVariables]
  [./temp]
  [../]
  [./fission_rate]
  [../]
[]


[AuxKernels]
  [./temp_aux]
    type = ConstantAux
    variable = temp
    value = 1000
  [../]
  [./fsn_rate_aux]
    type = ConstantAux
    variable = fission_rate
    value = 60
  [../]
[]


[Materials]
  [./Dg]
    type = GasAtomDiffusivity
    model = 4
    factor = 1
    block = 0
  [../]
[]



[Executioner]
  type = Transient

  solve_type = PJFNK

  line_search = none
  trans_ss_check = true
  scheme = bdf2

  l_max_its = 1000
  nl_max_its = 20

  end_time = 1e10

  [./TimeStepper]
    type = IterationAdaptiveDT
    dt = 1e-1
    growth_factor = 1.1
    optimal_iterations = 10
    iteration_window = 2
    linear_iteration_ratio = 100
  [../]
[]


[Postprocessors]
  [./dt]
    type = TimestepSize
  [../]
  [./swelling_terminator]
```

```
    type = PostprocessorTerminator
    postprocessor = gas_swelling
    threshold = 0.1
    outputs = 'none'
  [../]
[]


[Outputs]
  print_perf_log = true
  interval = 10
  output_final = true
  output_initial = true
  [./csv]
    file_base = '1000'
    interval = 1
    type = CSV
  [../]
  [./swelling_csv]
    file_base = '1000-sw'
    interval = 1
    type = CSV
  [../]
  [./c1_csv]
    file_base = '1000-c1'
    interval = 1
    type = CSV
  [../]
  [./gain_csv]
    type = CSV
    interval = 1
    file_base = '1000-g'
  [../]
  [./knockout_csv]
    type = CSV
    interval = 1
    file_base = '1000-k'
  [../]
  sync_times = '1e1 1e2 1e3 5e3 1e4 5e4 1e5 5e5 1e6 5e6 1e7 5e7 1e8 5e8 1e9'
[]
```

APPENDIX C: BUCK SOURCE

LISTINGS

The BUCK source requires the backbone framework of MOOSE, which can be downloaded using instructions from www.MOOSEframework.com. The BUCK source code can also be accessed from github.com/tophmatthews/buck.

What follows is the necessary header and source files needed to run the example input file in Appendix B, and should reside in the base directory ./buck, with the MOOSE framework in the adjacent directory ./moose.

Listing C.1: buck/Makefile

```
################################################################################
################## MOOSE Application Standard Makefile ######################
################################################################################
#
# Optional Environment variables
# MOOSE_DIR        - Root directory of the MOOSE project
#
################################################################################
# Use the MOOSE submodule if it exists and MOOSE_DIR is not set
MOOSE_SUBMODULE    := $(CURDIR)/moose
ifneq ($(wildcard $(MOOSE_SUBMODULE)/framework/Makefile),)
  MOOSE_DIR        ?= $(MOOSE_SUBMODULE)
else
  MOOSE_DIR        ?= $(shell dirname `pwd`)/moose
endif

# framework
FRAMEWORK_DIR      := $(MOOSE_DIR)/framework
include $(FRAMEWORK_DIR)/build.mk
include $(FRAMEWORK_DIR)/moose.mk

################################################################################

APPLICATION_DIR    := $(CURDIR)
APPLICATION_NAME   := buck
APP_REV_HEADER     := $(CURDIR)/include/base/BuckRevision.h
BUILD_EXEC         := yes
DEP_APPS           := $(shell $(FRAMEWORK_DIR)/scripts/find_dep_apps.py $(APPLICATION_NAME
    ))
include            $(FRAMEWORK_DIR)/app.mk

################################################################################
```

Listing C.2: buck/include/actions/BubblesActionBase.h

```
#ifndef BUBBLESACTIONBASE_H
#define BUBBLESACTIONBASE_H

#include "Action.h"

class BubblesActionBase: public Action
{
public:
  BubblesActionBase(const std::string & name, InputParameters params);
  virtual void act(){}

protected:
  const std::string _conc_name_base;
  const std::string _conc_1stM_name_base;
  const std::string _rad_name_base;
  const bool _exp;

  unsigned int _N;
  unsigned int _s;
  std::vector<VariableName> _c;
  std::vector<VariableName> _m;
  std::vector<VariableName> _r;
  std::vector<Real> _atoms;
  std::vector<Real> _widths;
  unsigned int _G;

private:
  void varNamesFromG(std::vector<VariableName> & vars, const std::string prefix, const int
     G, const int start=1);
};

template<>
InputParameters validParams<BubblesActionBase>();

#endif //BUBBLESACTIONBASE_H
```

buck/src/actions/BubblesActionBase.C

```
#include "BubblesActionBase.h"

#include "BuckUtils.h"
#include <iomanip>

template<>
InputParameters validParams<BubblesActionBase>()
{
  InputParameters params = validParams<Action>();

  params.addParam<std::string>("conc_name_base", "c", "Specifies the base name of the
    variables");
  params.addParam<std::string>("conc_1stM_name_base", "m", "Specifies the base name of the
     variables");
  params.addParam<std::string>("rad_name_base", "r", "Specifies the base name of the
    variables");
  params.addParam<int>("N", "Largest group size");
  params.addParam<int>("logN", "Log10 of largest group size");
  params.addParam<int>("s", "Total number of ungrouped equations");
  params.addParam<bool>("experimental", false, "Flag to use experimental kernel");

  return params;
}

BubblesActionBase::BubblesActionBase(const std::string & name, InputParameters params) :
  Action(name, params),
```

```
  _conc_name_base(getParam<std::string>("conc_name_base")),
  _conc_1stM_name_base(getParam<std::string>("conc_1stM_name_base")),
  _rad_name_base(getParam<std::string>("rad_name_base")),
  _exp(getParam<bool>("experimental"))
{
  if ( !isParamValid("N") && !isParamValid("logN") )
    mooseError("From BubblesActionBase: N or logN must be specified");
  else if ( isParamValid("N") && isParamValid("logN") )
    mooseError("From BubblesActionBase: Either N or logN must be specified");

  if ( isParamValid("N") )
  {
    if ( isParamValid("logN") )
      mooseError("From BubblesActionBase: Either N or logN must be specified, not both.");
    _N = getParam<int>("N");
  }
  else if ( !isParamValid("logN"))
    mooseError("From BubblesActionBase: N or log N must be specified");
  else
    _N = std::pow(10.0, getParam<int>("logN"));

  if ( isParamValid("s") )
    _s = getParam<int>("s");
  else
    _s = _N;

  for ( int j=0; j<_s; ++j )
    _atoms.push_back(j+1);

  for ( int j=_s; j<_N; ++j )
  {
    Real x = (_atoms.back() + 1) / _s + _atoms.back();

    if ( x <= _N )
      _atoms.push_back( x );
    else
    {
      _atoms.push_back( _N );
      break;
    }
  }

  _G = _atoms.size();

  for ( unsigned int i=0; i<_atoms.size()-1; ++i)
    _widths.push_back(_atoms[i+1] - _atoms[i]);
  _widths.push_back(1.0);

  varNamesFromG( _c, _conc_name_base, _G );
  varNamesFromG( _m, _conc_1stM_name_base, _G );
  varNamesFromG( _r, _rad_name_base, _G );
}


void
BubblesActionBase::varNamesFromG(std::vector<VariableName> & vars, const std::string
    prefix, const int G, const int start)
{
  int digits = Buck::numDigits(G);

  for ( int i=start; i<G+1; ++i)
  {
    VariableName var_name = prefix;
    std::stringstream out;
    out << std::setw(digits) << std::setfill('0') << i;
    var_name.append(out.str());
```

```
        vars.push_back(var_name);
90    }
}
```

Listing C.3: buck/include/actions/BubblesCoalescenceKernelsAction.h

```
#ifndef BUBBLESCOALESCENCEKERNERLSACTION_H
#define BUBBLESCOALESCENCEKERNERLSACTION_H

#include "BubblesActionBase.h"

class BubblesCoalescenceKernelsAction: public BubblesActionBase
{
public:
  BubblesCoalescenceKernelsAction(const std::string & name, InputParameters params);
  virtual void act();
};

template<>
InputParameters validParams<BubblesCoalescenceKernelsAction>();

#endif //BUBBLESCOALESCENCEKERNERLSACTION_H
```

buck/src/actions/BubblesCoalescenceKernelsAction.C

```
#include "BubblesCoalescenceKernelsAction.h"

#include "Factory.h"
#include "FEProblem.h"

template<>
InputParameters validParams<BubblesCoalescenceKernelsAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addRequiredParam<NonlinearVariableName>("temp", "The temperature variable name");
  params.addParam<bool>("use_displaced_mesh", false, "Whether to use displaced mesh in the
     kernels");

  return params;
}

BubblesCoalescenceKernelsAction::BubblesCoalescenceKernelsAction(const std::string & name,
     InputParameters params) :
  BubblesActionBase(name, params)
{
}


void
BubblesCoalescenceKernelsAction::act()
{
  for (int g=0; g<_G; ++g)
  {
    std::string var_name = _c[g];

    InputParameters p = _factory.getValidParams("BubbleCoalescence");
    p.set<NonlinearVariableName>("variable") = var_name;
    p.set<std::vector<VariableName> >("coupled_conc") = _c;
    p.set<std::vector<VariableName> >("coupled_rad") = _r;
    p.set<std::vector<Real> >("coupled_atoms") = _atoms;

    p.addCoupledVar("temp", "");
    p.set<std::vector<VariableName> >("temp") = std::vector<VariableName>(1, getParam<
    NonlinearVariableName>("temp"));

    std::string kernel_name = var_name;
    kernel_name.append("_growth");

    _problem->addKernel("BubbleCoalescence", kernel_name, p);
```

```
    }
}
```

Listing C.4: buck/include/actions/BubblesConcTimeKernelAction.h

```
#ifndef BUBBLESCONCTIMEKERNERLACTION_H
#define BUBBLESCONCTIMEKERNERLACTION_H

#include "BubblesActionBase.h"

class BubblesConcTimeKernelAction: public BubblesActionBase
{
public:
  BubblesConcTimeKernelAction(const std::string & name, InputParameters params);

  virtual void act();
};

template<>
InputParameters validParams<BubblesConcTimeKernelAction>();

#endif //BUBBLESCONCTIMEKERNERLACTION_H
```

buck/src/actions/BubblesConcTimeKernelAction.C

```
#include "BubblesConcTimeKernelAction.h"

#include "Factory.h"
#include "FEProblem.h"

template<>
InputParameters validParams<BubblesConcTimeKernelAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addParam<bool>("use_displaced_mesh", false, "Whether to use displaced mesh in the
     kernels");
  params.addParam<bool>("transient", true, "Flag to determine if TimeDerivative kernels
    should be made for nucleation concentration variables");

  return params;
}


BubblesConcTimeKernelAction::BubblesConcTimeKernelAction(const std::string & name,
                                                         InputParameters params) :
BubblesActionBase(name, params)
{
}


void
BubblesConcTimeKernelAction::act()
{
  if ( getParam<bool>("transient") )
  {
    for (unsigned int g = 0; g < _G; ++g)
    {
      InputParameters poly_params = _factory.getValidParams("TimeDerivative");

      poly_params.set<NonlinearVariableName>("variable") = _c[g];
      poly_params.set<bool>("use_displaced_mesh") = getParam<bool>("use_displaced_mesh");

      std::string kernel_name = _c[g];
      kernel_name.append("_time");

      _problem->addKernel("TimeDerivative", kernel_name, poly_params);
    }
  }
```

```
}
```

Listing C.5: buck/include/actions/BubblesConcVarsAction.h

```
#ifndef BUBBLESCONCVARSACTION_H
#define BUBBLESCONCVARSACTION_H

#include "BubblesActionBase.h"

class BubblesConcVarsAction : public BubblesActionBase
{
public:
  BubblesConcVarsAction(const std::string & name, InputParameters params);
  virtual ~BubblesConcVarsAction(){}
  virtual void act();

private:
  const std::string _order;
  const std::string _family;

  const Real _ic;
};

template<>
InputParameters validParams<BubblesConcVarsAction>();

#endif //BUBBLESCONCVARSACTION_H
```

buck/src/actions/BubblesConcVarsAction.C

```
#include "BubblesConcVarsAction.h"

#include "FEProblem.h"
#include "libmesh/string_to_enum.h"

template<>
InputParameters validParams<BubblesConcVarsAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addParam<std::string>("order", "FIRST",  "Specifies the order of the FE shape
    function to use for this variable");
  params.addParam<std::string>("family", "LAGRANGE", "Specifies the family of FE shape
    functions to use for this variable");
  params.addParam<Real>("scaling", 1.0, "Specifies a scaling factor to apply to the
    variables");
  params.addParam<Real>("c1_initial_condition", "Specifies a initial condtion to apply to
    c1");
  params.addParam<Real>("c2_initial_condition", "Specifies a initial condtion to apply to
    c2");
  params.addParam<Real>("initial_condition", "Specifies a initial condtion apply the rest
    of the variables");

  return params;
}

BubblesConcVarsAction::BubblesConcVarsAction(const std::string & name,
                                             InputParameters params) :
  BubblesActionBase(name, params),
  _order(getParam<std::string>("order")),
  _family(getParam<std::string>("family")),
  _ic(getParam<Real>("initial_condition"))
{
}

void
BubblesConcVarsAction::act()
{
```

```
  if (_current_task == "add_variable")
  {
    for (unsigned int i = 0; i < _G; ++i)
    {
      Real scale = 1.0;
      _problem->addVariable(_c[i],
                            FEType(Utility::string_to_enum<Order>(_order),
                                   Utility::string_to_enum<FEFamily>(_family)),
                            scale);
    }
  }
  else if (_current_task == "add_ic" && isParamValid("initial_condition"))
  {
    for (unsigned int i = 0; i < _G; ++i)
    {
      InputParameters poly_params = _factory.getValidParams("ConstantIC");
      poly_params.set<VariableName>("variable") = _c[i];
      if ( i == 0 )
        poly_params.set<Real>("value") = getParam<Real>("c1_initial_condition");
      else if ( i == 1 )
        poly_params.set<Real>("value") = getParam<Real>("c2_initial_condition");
      else
      {
        poly_params.set<Real>("value") = _ic / _widths[i];
      }
      _problem->addInitialCondition("ConstantIC", "Initialize_" + 1+i, poly_params);
    }
  }
}
```

Listing C.6: buck/include/actions/BubblesDampersAction.h

```
#ifndef BUBBLESDAMPERSACTION_H
#define BUBBLESDAMPERSACTION_H

#include "BubblesActionBase.h"

class BubblesDampersAction : public BubblesActionBase
{
public:
  BubblesDampersAction(const std::string & name, InputParameters params);
  virtual void act();

private:
  const int _index;
};

template<>
InputParameters validParams<BubblesDampersAction>();

#endif
```

buck/src/actions/BubblesDampersAction.C

```
#include "BubblesDampersAction.h"

#include "FEProblem.h"

template<>
InputParameters validParams<BubblesDampersAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addParam<Real>("damping", 0.1, "The maximum newton increment.");
  params.addParam<int>("starting_index", 1, "Variable list index at which to start
    applying damper.");

  return params;
}

BubblesDampersAction::BubblesDampersAction(const std::string & name, InputParameters
    params) :
  BubblesActionBase(name, params),
  _index(getParam<int>("starting_index"))
{
  if (_index > _G)
    mooseError("In BubblesDamperAction: starting_index must be less than then the total
    number of groups, G");
}

void
BubblesDampersAction::act()
{
  for (int g=_index; g<_G; ++g)
  {
    std::string base_kernel = "PositiveDamper";

    InputParameters p = _factory.getValidParams(base_kernel);
    std::string var_name = _c[g];
    std::string kernel_name = var_name;
    kernel_name.append(base_kernel);

    // BubbleBase variables
    p.set<NonlinearVariableName>("variable") = var_name;
    p.set<std::vector<VariableName> >("coupled_conc") = _c;
    p.set<std::vector<VariableName> >("coupled_rad") = _r;
```

```
    p.set<std::vector<Real> >("coupled_atoms") = _atoms;
    p.set<std::vector<Real> >("coupled_widths") = _widths;

    // BubbleDamping variables
    p.set<Real>("damping") = getParam<Real>("damping");

    _problem->addDamper(base_kernel, kernel_name, p);
  }
}
```

Listing C.7: buck/include/actions/BubblesFFNucleationKernelsAction.h

```
#ifndef BUBBLESFFNUCLEATIONKERNELSACTION_H
#define BUBBLESFFNUCLEATIONKERNELSACTION_H

#include "BubblesActionBase.h"

class BubblesFFNucleationKernelsAction: public BubblesActionBase
{
public:
  BubblesFFNucleationKernelsAction(const std::string & name, InputParameters params);
  virtual void act();
};

template<>
InputParameters validParams<BubblesFFNucleationKernelsAction>();

#endif //BubblesFFNucleationKernelsAction_H
```

buck/src/actions/BubblesFFNucleationKernelsAction.C

```
#include "BubblesFFNucleationKernelsAction.h"

#include "Factory.h"
#include "FEProblem.h"

template<>
InputParameters validParams<BubblesFFNucleationKernelsAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addRequiredParam<VariableName>("fission_rate", "The fission rate density variable
     name");
  params.addParam<bool>("use_displaced_mesh", false, "Whether to use displaced mesh in the
     kernels");
  params.addParam<int>("number", 5, "Number of bubbles created per fission");
  params.addParam<int>("size", 4, "Size of bubbles created");
  params.addParam<Real>("factor", 1.0, "User supplied multiplier.");
  params.addParam<Real>("upper", 1e7, "Upper dead-band limit");
  params.addParam<Real>("lower", 1e6, "Lower dead-band limit");

  return params;
}

BubblesFFNucleationKernelsAction::BubblesFFNucleationKernelsAction(const std::string &
    name,
                                                                   InputParameters params) :
  BubblesActionBase(name, params)
{
}

void
BubblesFFNucleationKernelsAction::act()
{
  int n=0;
  {
    std::string var_name = _c[n];

    InputParameters p = _factory.getValidParams("BubbleFFNucleation");

    p.set<bool>("use_displaced_mesh") = getParam<bool>("use_displaced_mesh");

    p.set<Real>("factor") = getParam<Real>("factor");
    p.set<int>("number") = getParam<int>("number");
    p.set<int>("size") = getParam<int>("size");
    p.set<Real>("upper") = getParam<Real>("upper");
```

```
    p.set<Real>("lower") = getParam<Real>("lower");

    p.set<NonlinearVariableName>("variable") = var_name;
    p.set<std::vector<VariableName> >("coupled_conc") = _c;
    p.set<std::vector<VariableName> >("coupled_rad") = _r;
    p.set<std::vector<Real> >("coupled_atoms") = _atoms;
    p.set<std::vector<Real> >("coupled_widths") = _widths;

    p.addCoupledVar("fission_rate", "");
    p.set<std::vector<VariableName> >("fission_rate") = std::vector<VariableName>(1,
    getParam<VariableName>("fission_rate"));

    std::string kernel_name = var_name;
    kernel_name.append("_nucleation");

    _problem->addKernel("BubbleFFNucleation", kernel_name, p);
  }

  n = getParam<int>("size")-1;
  {
    std::string var_name = _c[n];

    InputParameters p = _factory.getValidParams("BubbleFFNucleation");

    p.set<bool>("use_displaced_mesh") = getParam<bool>("use_displaced_mesh");

    p.set<Real>("factor") = getParam<Real>("factor");
    p.set<int>("number") = getParam<int>("number");
    p.set<int>("size") = getParam<int>("size");

    p.set<NonlinearVariableName>("variable") = var_name;
    p.set<std::vector<VariableName> >("coupled_conc") = _c;
    p.set<std::vector<VariableName> >("coupled_rad") = _r;
    p.set<std::vector<Real> >("coupled_atoms") = _atoms;
    p.set<std::vector<Real> >("coupled_widths") = _widths;

    p.addCoupledVar("fission_rate", "");
    p.set<std::vector<VariableName> >("fission_rate") = std::vector<VariableName>(1,
    getParam<VariableName>("fission_rate"));

    std::string kernel_name = var_name;
    kernel_name.append("_nucleation");

    _problem->addKernel("BubbleFFNucleation", kernel_name, p);
  }
}
```

Listing C.8: buck/include/actions/BubblesGrowthKernelsAction.h

```
#ifndef BUBBLESGROWTHKERNERLSACTION_H
#define BUBBLESGROWTHKERNERLSACTION_H

#include "BubblesActionBase.h"

class BubblesGrowthKernelsAction: public BubblesActionBase
{
public:
  BubblesGrowthKernelsAction(const std::string & name, InputParameters params);
  virtual void act();

private:
  const bool _include_c1;
  const bool _include_c2;
};

template<>
InputParameters validParams<BubblesGrowthKernelsAction>();

#endif //BUBBLESGROWTHKERNERLSACTION_H
```

buck/src/actions/BubblesGrowthKernelsAction.C

```
#include "BubblesGrowthKernelsAction.h"

#include "FEProblem.h"

template<>
InputParameters validParams<BubblesGrowthKernelsAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addRequiredParam<VariableName>("temp", "The temperature variable name");

  params.addParam<bool>("allow_loss", false, "Flag to allow losses from the largest bubble
     group.");
  params.addParam<bool>("include_c1", true, "Flag to create growth kernel for c1");
  params.addParam<bool>("include_c2", true, "Flag to create growth kernel for c2");

  return params;
}

BubblesGrowthKernelsAction::BubblesGrowthKernelsAction(const std::string & name,
    InputParameters params) :
  BubblesActionBase(name, params),
  _include_c1(getParam<bool>("include_c1")),
  _include_c2(getParam<bool>("include_c2"))
{
}


void
BubblesGrowthKernelsAction::act()
{
  for (int g=0; g<_G; ++g)
  {
    if ( g==0 && !_include_c1 )
      continue;

    if ( g==1 && !_include_c2 )
      continue;

    std::string base_kernel = "BubbleGrowth";
```

```
40      InputParameters p = _factory.getValidParams(base_kernel);
        std::string var_name = _c[g];
        std::string kernel_name = var_name;
        kernel_name.append(base_kernel);

45      // BubbleBase variables
        p.set<NonlinearVariableName>("variable") = var_name;
        p.set<std::vector<VariableName> >("coupled_conc") = _c;
        p.set<std::vector<VariableName> >("coupled_rad") = _r;
        p.set<std::vector<Real> >("coupled_atoms") = _atoms;
50      p.set<std::vector<Real> >("coupled_widths") = _widths;

        // BubbleGrowth variables
        p.set<bool>("allow_loss") = getParam<bool>("allow_loss");
        p.set<bool>("experimental") = _exp;
55      p.addCoupledVar("temp", "");
        p.set<std::vector<VariableName> >("temp") = std::vector<VariableName>(1, getParam<
        VariableName>("temp"));

        _problem->addKernel(base_kernel, kernel_name, p);
    }
60  }
```

Listing C.9: buck/include/actions/BubblesKnockoutKernelsAction.h

```
#ifndef BUBBLESKNOCKOUTKERNERLSACTION_H
#define BUBBLESKNOCKOUTKERNERLSACTION_H

#include "BubblesActionBase.h"

class BubblesKnockoutKernelsAction: public BubblesActionBase
{
public:
  BubblesKnockoutKernelsAction(const std::string & name, InputParameters params);
  virtual void act();
};

template<>
InputParameters validParams<BubblesKnockoutKernelsAction>();

#endif //BUBBLESKNOCKOUTKERNERLSACTION_H
```

buck/src/actions/BubblesKnockoutKernelsAction.C

```
#include "BubblesKnockoutKernelsAction.h"

#include "Factory.h"
#include "FEProblem.h"
#include "BuckUtils.h"

template<>
InputParameters validParams<BubblesKnockoutKernelsAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addParam<bool>("use_displaced_mesh", false, "Whether to use displaced mesh in the
     kernels");

  params.addRequiredParam<VariableName>("fission_rate", "The fission_rate variable name");
  params.addParam<Real>("factor", 1, "Scaling factor");
  params.addParam<Real>("b", -1, "Value to set constant knockout parameter. B is
    automatically calculated if not given.");

  return params;
}

BubblesKnockoutKernelsAction::BubblesKnockoutKernelsAction(const std::string & name,
    InputParameters params) :
  BubblesActionBase(name, params)
{
}

void
BubblesKnockoutKernelsAction::act()
{
  for (int g=0; g<_G; ++g)
  {
    std::string base_kernel = "BubbleKnockout";

    InputParameters p = _factory.getValidParams(base_kernel);
    std::string var_name = _c[g];
    std::string kernel_name = var_name;
    kernel_name.append(base_kernel);

    // BubbleBase variables
    p.set<NonlinearVariableName>("variable") = var_name;
    p.set<std::vector<VariableName> >("coupled_conc") = _c;
    p.set<std::vector<VariableName> >("coupled_rad") = _r;
    p.set<std::vector<Real> >("coupled_atoms") = _atoms;
```

```
    p.set<std::vector<Real> >("coupled_widths") = _widths;

    // BubbleKnockout variables
    p.set<bool>("experimental") = _exp;
    p.set<Real>("factor") = getParam<Real>("factor");
    p.set<Real>("b") = getParam<Real>("b");
    p.addCoupledVar("fission_rate", "");
    p.set<std::vector<VariableName> >("fission_rate") = std::vector<VariableName>(1,
    getParam<VariableName>("fission_rate"));


    _problem->addKernel(base_kernel, kernel_name, p);
  }
}
```

Listing C.10: buck/include/actions/BubblesNucleationKernelsAction.h

```
#ifndef BUBBLESNUCLEATIONKERNELSACTION_H
#define BUBBLESNUCLEATIONKERNELSACTION_H

#include "BubblesActionBase.h"

class BubblesNucleationKernelsAction: public BubblesActionBase
{
public:
  BubblesNucleationKernelsAction(const std::string & name, InputParameters params);
  virtual void act();
};

template<>
InputParameters validParams<BubblesNucleationKernelsAction>();

#endif //BUBBLESNUCLEATIONKERNELSACTION_H
```

buck/src/actions/BubblesNucleationKernelsAction.C

```
#include "BubblesNucleationKernelsAction.h"

#include "Factory.h"
#include "FEProblem.h"

template<>
InputParameters validParams<BubblesNucleationKernelsAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addRequiredParam<VariableName>("temp", "The temperature variable name");
  params.addParam<bool>("use_displaced_mesh", false, "Whether to use displaced mesh in the
     kernels");
  params.addParam<Real>("a", 4.96e-4, "Lattice Parameter [um]");
  params.addParam<Real>("omega", 1.53e-11, " Atomic volume [um^3]");
  params.addParam<Real>("factor", 1.0, "User supplied multiplier.");

  return params;
}

BubblesNucleationKernelsAction::BubblesNucleationKernelsAction(const std::string & name,
                                                               InputParameters params) :
  BubblesActionBase(name, params)
{
}

void
BubblesNucleationKernelsAction::act()
{
  for (unsigned int n = 0; n < 2; ++n)
  {
    std::string var_name = _c[n];

    InputParameters p = _factory.getValidParams("BubbleNucleation");

    p.set<bool>("use_displaced_mesh") = getParam<bool>("use_displaced_mesh");
    p.set<Real>("a") = getParam<Real>("a");
    p.set<Real>("omega") = getParam<Real>("omega");
    p.set<Real>("factor") = getParam<Real>("factor");

    p.set<NonlinearVariableName>("variable") = var_name;
    p.set<std::vector<VariableName> >("coupled_conc") = _c;
    p.set<std::vector<VariableName> >("coupled_rad") = _r;
    p.set<std::vector<Real> >("coupled_atoms") = _atoms;
    p.set<std::vector<Real> >("coupled_widths") = _widths;
```

```
45
    p.addCoupledVar("temp", "");
    p.set<std::vector<VariableName> >("temp") = std::vector<VariableName>(1, getParam<
    VariableName>("temp"));

    std::string kernel_name = var_name;
50  kernel_name.append("_nucleation");

    _problem->addKernel("BubbleNucleation", kernel_name, p);
  }
}
```

Listing C.11: buck/include/actions/BubblesPostprocessorsAction.h

```
#ifndef BUBBLESPOSTPROCESSORSACTION_H
#define BUBBLESPOSTPROCESSORSACTION_H

#include "BubblesActionBase.h"

class BubblesPostprocessorsAction: public BubblesActionBase
{
public:
  BubblesPostprocessorsAction(const std::string & name, InputParameters params);
  virtual void act();

private:
  const bool _conc;
  const bool _total_conc;
  const bool _total_atoms;
  const bool _swelling;
  const bool _total_swelling;
  const bool _c1_loss;
  const bool _gain_rate;
  const bool _knockout_rate;
};

template<>
InputParameters validParams<BubblesPostprocessorsAction>();

#endif
```

buck/src/actions/BubblesPostprocessorsAction.C

```
#include "BubblesPostprocessorsAction.h"

#include "Factory.h"
#include "FEProblem.h"

template<>
InputParameters validParams<BubblesPostprocessorsAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addRequiredParam<VariableName>("fission_rate", "fission rate density");

  params.addParam<std::vector<OutputName> >("concentrations", "Where to output
    concentration postprocessors");
  params.addParam<std::vector<OutputName> >("total_concentrations", "Where to output
    concentration postprocessor. Not calculated if empty.");
  params.addParam<std::vector<OutputName> >("total_atoms", "Where to output concentration
    postprocessor. Not calculated if empty.");
  params.addParam<std::vector<OutputName> >("swelling", "Where to output swelling
    postprocessor. Not calculated if empty.");
  params.addParam<std::vector<OutputName> >("total_swelling", "Where to output total
    swelling postprocessor. Not calculated if empty.");
  params.addParam<std::vector<OutputName> >("c1_loss", "Where to output
    C1LossPostprocessor. Not calculated if empty.");
  params.addParam<std::vector<OutputName> >("gain_rate", "Where to output
    GainRatePostprocessor. Not calculated if empty.");
  params.addParam<std::vector<OutputName> >("knockout_rate", "Where to output
    KnockoutRatePostprocessor. Not calculated if empty.");

  return params;
}

BubblesPostprocessorsAction::BubblesPostprocessorsAction(const std::string & name,
    InputParameters params) :
  BubblesActionBase(name, params),
```

```
  _conc(isParamValid("concentrations") ? true : false),
  _total_conc(isParamValid("total_concentrations") ? true : false),
  _total_atoms(isParamValid("total_atoms") ? true : false),
  _swelling(isParamValid("swelling") ? true : false),
  _total_swelling(isParamValid("total_swelling") ? true : false),
  _c1_loss(isParamValid("c1_loss") ? true : false),
  _gain_rate(isParamValid("gain_rate") ? true : false),
  _knockout_rate(isParamValid("knockout_rate") ? true : false)
{
}

void
BubblesPostprocessorsAction::act()
{
  std::vector<PostprocessorName> pp_names;
  std::vector<PostprocessorName> swelling_pp_names;

  if ( _conc || _total_conc || _total_atoms )
  {
    for ( int i=0; i<_G; ++i )
    {
      std::string pp_to_use = "BoundedElementAverage";
      pp_names.push_back(_c[i] + "_conc");

      InputParameters params = _factory.getValidParams(pp_to_use);
      params.set<MultiMooseEnum>("execute_on") = "timestep_end";
      params.set<VariableName>("variable") = _c[i];
      params.set<Real>("lower") = 0;

      std::vector<OutputName> outs;
      if ( _conc )
        outs = getParam<std::vector<OutputName> >("concentrations");
      else
        outs.push_back("none");

      params.set<std::vector<OutputName> >("outputs") = outs;

      _problem->addPostprocessor(pp_to_use, pp_names[i], params);
    }
  }

  if (_total_conc)
  {
    std::string pp_to_use = "SumOfPostprocessors";
    std::string this_pp_name = "total_conc";

    InputParameters params = _factory.getValidParams(pp_to_use);
    params.set<MultiMooseEnum>("execute_on") = "timestep_end";
    params.set<std::vector<PostprocessorName> >("postprocessors") = pp_names;

    std::vector<OutputName> outs(getParam<std::vector<OutputName> >("total_concentrations"
    ));
    params.set<std::vector<OutputName> >("outputs") = outs;

    _problem->addPostprocessor(pp_to_use, this_pp_name, params);
  }

  if (_total_atoms)
  {
    std::string pp_to_use = "SumOfPostprocessors";
    std::string this_pp_name = "total_atoms";

    InputParameters params = _factory.getValidParams(pp_to_use);
    params.set<MultiMooseEnum>("execute_on") = "timestep_end";
    params.set<std::vector<PostprocessorName> >("postprocessors") = pp_names;
    params.set<std::vector<Real> >("factors") = _atoms;
```

```
    std::vector<OutputName> outs(getParam<std::vector<OutputName> >("total_atoms"));
    params.set<std::vector<OutputName> >("outputs") = outs;

    _problem->addPostprocessor(pp_to_use, this_pp_name, params);
  }

  if (_swelling || _total_swelling)
  {
    for ( int i=0; i<_G; ++i )
    {
      std::string pp_to_use = "SwellingPostprocessor";
      swelling_pp_names.push_back(_c[i] + "_swelling");

      InputParameters params = _factory.getValidParams(pp_to_use);
      params.set<MultiMooseEnum>("execute_on") = "timestep_end";
      params.set<VariableName>("variable") = _c[i];

      params.addCoupledVar("r", "");
      params.set<std::vector<VariableName> >("r") = std::vector<VariableName>(1, _r[i]);

      params.set<Real>("width") = _widths[i];

      std::vector<OutputName> outs;
      if ( _swelling )
        outs = getParam<std::vector<OutputName> >("swelling");
      else
        outs.push_back("none");

      params.set<std::vector<OutputName> >("outputs") = outs;

      _problem->addPostprocessor(pp_to_use, swelling_pp_names[i], params);
    }
  }

  if (_total_swelling)
  {
    std::string pp_to_use = "SumOfPostprocessors";
    std::string this_pp_name = "gas_swelling";

    InputParameters params = _factory.getValidParams(pp_to_use);
    params.set<MultiMooseEnum>("execute_on") = "timestep_end";
    params.set<std::vector<PostprocessorName> >("postprocessors") = swelling_pp_names;

    std::vector<OutputName> outs(getParam<std::vector<OutputName> >("total_swelling"));
    params.set<std::vector<OutputName> >("outputs") = outs;

    _problem->addPostprocessor(pp_to_use, this_pp_name, params);
  }

  if (_c1_loss)
  {
    std::vector<PostprocessorName> these_pp_names;
    for ( int i=0; i<_G; ++i )
    {
      std::string pp_to_use = "C1LossPostprocessor";
      these_pp_names.push_back(_c[i] + "_c1_loss");

      InputParameters params = _factory.getValidParams(pp_to_use);
      params.set<MultiMooseEnum>("execute_on") = "timestep_end";
      params.set<VariableName>("variable") = _c[i];

      params.addCoupledVar("r", "");
      params.set<std::vector<VariableName> >("r") = std::vector<VariableName>(1, _r[i]);

      params.addCoupledVar("c1", "");
```

```
      params.set<std::vector<VariableName> >("c1") = std::vector<VariableName>(1, _c[0]);

      params.addCoupledVar("fission_rate", "");
      params.set<std::vector<VariableName> >("fission_rate") = std::vector<VariableName
    >(1, getParam<VariableName>("fission_rate"));

      params.set<Real>("width") = _widths[i];
      params.set<Real>("atoms") = _atoms[i];

      params.set<std::vector<OutputName> >("outputs") = getParam<std::vector<OutputName>
    >("c1_loss");

      _problem->addPostprocessor(pp_to_use, these_pp_names[i], params);
    }
  }

  if (_gain_rate)
  {
    std::vector<PostprocessorName> these_pp_names;
    for ( int i=0; i<_G; ++i )
    {
      std::string pp_to_use = "GainRatePostprocessor";
      these_pp_names.push_back(_c[i] + "_gain_rate");

      InputParameters params = _factory.getValidParams(pp_to_use);
      params.set<MultiMooseEnum>("execute_on") = "timestep_end";
      params.set<VariableName>("variable") = _c[i];

      params.addCoupledVar("r", "");
      params.set<std::vector<VariableName> >("r") = std::vector<VariableName>(1, _r[i]);

      params.addCoupledVar("c1", "");
      params.set<std::vector<VariableName> >("c1") = std::vector<VariableName>(1, _c[0]);

      params.set<Real>("width") = _widths[i];

      params.set<std::vector<OutputName> >("outputs") = getParam<std::vector<OutputName>
    >("gain_rate");

      _problem->addPostprocessor(pp_to_use, these_pp_names[i], params);
    }
  }

  if (_knockout_rate)
  {
    std::vector<PostprocessorName> these_pp_names;
    for ( int i=0; i<_G; ++i )
    {
      std::string pp_to_use = "KnockoutRatePostprocessor";
      these_pp_names.push_back(_c[i] + "_knockout_rate");

      InputParameters params = _factory.getValidParams(pp_to_use);
      params.set<MultiMooseEnum>("execute_on") = "timestep_end";
      params.set<VariableName>("variable") = _c[i];

      params.addCoupledVar("r", "");
      params.set<std::vector<VariableName> >("r") = std::vector<VariableName>(1, _r[i]);

      params.addCoupledVar("c1", "");
      params.set<std::vector<VariableName> >("c1") = std::vector<VariableName>(1, _c[0]);

      params.addCoupledVar("fission_rate", "");
      params.set<std::vector<VariableName> >("fission_rate") = std::vector<VariableName
    >(1, getParam<VariableName>("fission_rate"));

      params.set<Real>("width") = _widths[i];
```

```
220         params.set<Real>("atoms") = _atoms[i];

            params.set<std::vector<OutputName> >("outputs") = getParam<std::vector<OutputName>
        >("knockout_rate");

            _problem->addPostprocessor(pp_to_use, these_pp_names[i], params);
225       }
    }
}
```

Listing C.12: buck/include/actions/BubblesRadAuxKernelAction.h

```
#ifndef BubblesRadAuxKernelAction_H
#define BubblesRadAuxKernelAction_H

#include "BubblesActionBase.h"

class BubblesRadAuxKernelAction;

template<>
InputParameters validParams<BubblesRadAuxKernelAction>();

class BubblesRadAuxKernelAction : public BubblesActionBase
{
public:
  BubblesRadAuxKernelAction(const std::string & name, InputParameters params);

  virtual void act();

private:
  std::vector<SubdomainName> _blocks;
};

#endif // BubblesRadAuxKernelAction_H
```

buck/src/actions/BubblesRadAuxKernelAction.C

```
#include "BubblesRadAuxKernelAction.h"

#include "FEProblem.h"
#include "Factory.h"

template<>
InputParameters validParams<BubblesRadAuxKernelAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addRequiredParam<NonlinearVariableName>("temp", "The temperature variable name");
  params.addRequiredParam<std::vector<SubdomainName> >("block", "The blocks where bounds
    should be applied.");

  return params;
}


BubblesRadAuxKernelAction::BubblesRadAuxKernelAction(const std::string & name,
    InputParameters params) :
  BubblesActionBase(name, params),
  _blocks(getParam<std::vector<SubdomainName> >("block"))
{
}


void
BubblesRadAuxKernelAction::act()
{
  for ( unsigned int i=0; i<_G; ++i)
  {
    InputParameters params = _factory.getValidParams("EquilibriumRadiusAux");

    params.set<std::vector<SubdomainName> >("block") = _blocks;
    params.set<AuxVariableName>("variable") = _r[i];
    params.set<Real>("m") = _atoms[i];

    params.addCoupledVar("temp", "");
```

```
    params.set<std::vector<VariableName> >("temp") = std::vector<VariableName>(1, getParam
    <NonlinearVariableName>("temp"));

    std::string kernel_name = _r[i];
    kernel_name.append("_eq");
    _problem->addAuxKernel("EquilibriumRadiusAux", kernel_name, params);
  }
}
```

Listing C.13: buck/include/actions/BubblesRadAuxVarsAction.h

```
#ifndef BUBBLESRADAUXVARSACTION_H
#define BUBBLESRADAUXVARSACTION_H

#include "BubblesActionBase.h"

class BubblesRadAuxVarsAction : public BubblesActionBase
{
public:
  BubblesRadAuxVarsAction(const std::string & name, InputParameters params);
  virtual ~BubblesRadAuxVarsAction(){}
  virtual void act();

private:
  const std::string _order;
  const std::string _family;
};

template<>
InputParameters validParams<BubblesRadAuxVarsAction>();


#endif //BUBBLESRADAUXVARSACTION_H
```

buck/src/actions/BubblesRadAuxVarsAction.C

```
#include "BubblesRadAuxVarsAction.h"

#include "FEProblem.h"
#include "libmesh/string_to_enum.h"

template<>
InputParameters validParams<BubblesRadAuxVarsAction>()
{
  InputParameters params = validParams<BubblesActionBase>();

  params.addParam<std::string>("order", "FIRST",  "Specifies the order of the FE shape
    function to use for this variable");
  params.addParam<std::string>("family", "LAGRANGE", "Specifies the family of FE shape
    functions to use for this variable");
  params.addRequiredParam<std::vector<SubdomainName> >("block", "The block id where this
    variable lives");

  return params;
}

BubblesRadAuxVarsAction::BubblesRadAuxVarsAction(const std::string & name,
                                                 InputParameters params) :
  BubblesActionBase(name, params),
  _order(getParam<std::string>("order")),
  _family(getParam<std::string>("family"))
{
  mooseAssert(!getParam<std::vector<SubdomainName> >("block").empty(), "Blocks must be
    specified in BubblesRadAuxVarsAction");
}

void
BubblesRadAuxVarsAction::act()
{
  std::set<SubdomainID> blocks;

  std::vector<SubdomainName> block_param = getParam<std::vector<SubdomainName> >("block");
  for (std::vector<SubdomainName>::iterator it = block_param.begin(); it != block_param.
    end(); ++it)
    blocks.insert(_problem->mesh().getSubdomainID(*it));
```

```
35
  for ( unsigned int i=0; i<_G; ++i )
  {
    _problem->addAuxVariable(_r[i],
                             FEType(Utility::string_to_enum<Order>(_order),
40                                    Utility::string_to_enum<FEFamily>(_family)),
                              &blocks);
  }
}
```

Listing C.14: buck/include/auxkernels/EquilibriumRadiusAux.h

```
#ifndef EQUILIBRIUMRADIUSAUX_H
#define EQUILIBRIUMRADIUSAUX_H

#include "AuxKernel.h"

class EquilibriumRadiusAux : public AuxKernel
{
public:
  EquilibriumRadiusAux(const std::string & name, InputParameters parameters);
  virtual ~EquilibriumRadiusAux() {}

protected:
  virtual Real computeValue();

private:
  VariableValue & _temp;
  VariableValue & _sigma;
  const Real _m;
  const Real _gamma;
  const Real _B;
};

template<>
InputParameters validParams<EquilibriumRadiusAux>();

#endif //EQUILIBRIUMRADIUSAUX_H
```

buck/src/auxkernels/EquilibriumRadiusAux.C

```
#include "EquilibriumRadiusAux.h"

#include "Material.h"
#include "MaterialXeBubble.h"

template<>
InputParameters validParams<EquilibriumRadiusAux>()
{
  InputParameters params = validParams<AuxKernel>();

  params.addRequiredCoupledVar("temp", "Coupled temperature");
  params.addCoupledVar("sigma", 0, "Coupled hydrostatic stress");
  params.addRequiredParam<Real>("m", "Number of atoms");
  params.addParam<Real>("gamma", 0.626, "Surface tension [J/m^2]");
  params.addParam<Real>("B", 8.469e-29, " Atomic volume [m^3]");
  return params;
}

EquilibriumRadiusAux::EquilibriumRadiusAux(const std::string & name, InputParameters
    parameters)
  :AuxKernel(name, parameters),
  _temp(coupledValue("temp")),
  _sigma(coupledValue("sigma")),
  _m(getParam<Real>("m")),
  _gamma(getParam<Real>("gamma")),
  _B(getParam<Real>("B"))
{
}

Real
EquilibriumRadiusAux::computeValue()
{
  Real gamma_in_m = _gamma;
  Real B_in_m = _B;
```

```
  Real rad_in_m = MaterialXeBubble::VDW_MtoR(_m, _temp[_qp], _sigma[_qp], gamma_in_m,
    B_in_m, false);
  return rad_in_m * 1.0e6;
}
```

Listing C.15: buck/include/base/BuckApp.h

```
#ifndef BUCKAPP_H
#define BUCKAPP_H

#include "MooseApp.h"

class BuckApp;

template<>
InputParameters validParams<BuckApp>();

class BuckApp : public MooseApp
{
public:
  BuckApp(const std::string & name, InputParameters parameters);
  virtual ~BuckApp();

  virtual void runInputFile();

  static void registerApps();
  static void registerObjects(Factory & factory);
  // static void associateSyntax(Syntax & syntax, ActionFactory & action_factory);

protected:
  void printHeader();

};

#endif /* BUCKAPP_H */
```

buck/src/base/BuckApp.C

```
#include "BuckApp.h"
#include "Moose.h"
#include "AppFactory.h"

// Buck
#include "BuckSyntax.h"
#include "BuckRevision.h"

// AuxKernels
#include "EquilibriumRadiusAux.h"

// dampers
#include "PositiveDamper.h"

// Kernels
#include "VariableScaledSource.h"
#include "BasicDiffusion.h"
#include "BubbleBase.h"
#include "BubbleNucleation.h"
#include "BubbleFFNucleation.h"
#include "BubbleGrowth.h"
#include "BubbleKnockout.h"

// Materials
#include "GasAtomDiffusivity.h"

// Postprocessors
#include "GrainBoundaryGasFlux.h"
#include "SumOfPostprocessors.h"
#include "MaterialXeBubbleTester.h"
#include "BoundedElementAverage.h"
#include "SwellingPostprocessor.h"
#include "PostprocessorTerminator.h"
```

```
#include "C1LossPostprocessor.h"
#include "GainRatePostprocessor.h"
#include "KnockoutRatePostprocessor.h"

template<>
InputParameters validParams<BuckApp>()
{
  InputParameters params = validParams<MooseApp>();
  return params;
}

BuckApp::BuckApp(const std::string & name, InputParameters parameters) :
    MooseApp(name, parameters)
{
  srand(processor_id());

  Moose::registerObjects(_factory);
  BuckApp::registerObjects(_factory);

  Moose::associateSyntax(_syntax, _action_factory);
  Buck::associateSyntax(_syntax, _action_factory);
}

BuckApp::~BuckApp()
{
}

extern "C" void BuckApp__registerApps() { BuckApp::registerApps(); }
void
BuckApp::registerApps()
{
  registerApp(BuckApp);
}

void
BuckApp::registerObjects(Factory & factory)
{
  registerAux(EquilibriumRadiusAux);

  registerDamper(PositiveDamper);

  registerKernel(VariableScaledSource);
  registerKernel(BasicDiffusion);
  registerKernel(BubbleBase);
  registerKernel(BubbleNucleation);
  registerKernel(BubbleFFNucleation);
  registerKernel(BubbleGrowth);
  registerKernel(BubbleKnockout);

  registerMaterial(GasAtomDiffusivity);

  registerPostprocessor(GrainBoundaryGasFlux);
  registerPostprocessor(SumOfPostprocessors);
  registerPostprocessor(MaterialXeBubbleTester);
  registerPostprocessor(BoundedElementAverage);
  registerPostprocessor(SwellingPostprocessor);
  registerPostprocessor(PostprocessorTerminator);
  registerPostprocessor(C1LossPostprocessor);
  registerPostprocessor(GainRatePostprocessor);
  registerPostprocessor(KnockoutRatePostprocessor);
}

void
BuckApp::runInputFile()
{
  printHeader();
```

```
100   MooseApp::runInputFile();
}

void
BuckApp::printHeader()
105 {
  Moose::out  << "\n"
              << "\n"
              << "      ______  _      _  ______ _     _       \n"
              << "     (_____  \\| |    | |/ _____) |   / )     \n"
              << "      ____)   | |    | | /      | | / /      \n"
110           << "     |  __   (| |    | | |      | |< <       \n"
              << "     | |__)   | |___| | \\_____| | \\ \\    \n"
              << "     |______/ \\______|\\______)_|  \\_)   \n"

              << "\n"
115           << "\n"
              << "       Bubble and Cluster Kinetics \n"
              << "         Oregon State University   \n"
              << "              Corvallis, OR        \n"
              << "\n"
120           << "\n";

  Moose::out << "Input file:   " << _input_filename << "\n"
             << "Input units:  micrometer, gram, second, kelvin, mole\n"
             << "\n"
125          << "BUCK version: " << BUCK_REVISION << std::endl
             << std::endl;
}
```

Listing C.16: buck/include/dampers/PositiveDamper.h

```
#ifndef POSITIVEDAMPER_H
#define POSITIVEDAMPER_H

#include "Damper.h"

class PositiveDamper : public Damper
{
public:
  PositiveDamper(std::string name, InputParameters parameters);
  virtual ~PositiveDamper() {}

protected:
  virtual Real computeQpDamping();

  Real _damping;
};

template<>
InputParameters validParams<PositiveDamper>();

#endif //POSITIVEDAMPER_H
```

buck/src/dampers/PositiveDamper.C

```
#include "PositiveDamper.h"

template<>
InputParameters validParams<PositiveDamper>()
{
  InputParameters params = validParams<Damper>();
  params.addRequiredParam<Real>("damping", "Damping coefficient to use if increment
    results in a negative value");
  return params;
}

PositiveDamper::PositiveDamper(std::string name, InputParameters parameters) :
    Damper(name, parameters),
    _damping(parameters.get<Real>("damping"))
{
}

Real
PositiveDamper::computeQpDamping()
{

  Real delu = _u[_qp] - _u_increment[_qp];
  if ( delu < 0)
    return _damping;

  return 1.0;
}
```

Listing C.17: buck/include/kernels/BasicDiffusion.h

```
#ifndef BASICDIFFUSION_H
#define BASICDIFFUSION_H

#include "Diffusion.h"

class BasicDiffusion : public Diffusion
{
public:
  BasicDiffusion( const std::string & name, InputParameters parameters );

protected:
  virtual Real computeQpResidual();
  virtual Real computeQpJacobian();

private:
  MaterialProperty<Real> & _diffusivity;
};

template<>
InputParameters validParams<BasicDiffusion>();

#endif // BASICDIFFUSION_H
```

buck/src/kernels/BasicDiffusion.C

```
#include "BasicDiffusion.h"

template<>
InputParameters validParams<BasicDiffusion>()
{
  InputParameters params = validParams<Diffusion>();

  params.addParam<std::string>("diffusivity", "diffusivity", "Diffusion coefficient (
    Default: diffusivity");

  return params;
}


BasicDiffusion::BasicDiffusion( const std::string & name, InputParameters parameters ) :
    Diffusion( name, parameters ),
    _diffusivity(getMaterialProperty<Real>(getParam<std::string>("diffusivity")))
{
}


Real
BasicDiffusion::computeQpResidual()
{
  return _diffusivity[_qp] * Diffusion::computeQpResidual();
}


Real
BasicDiffusion::computeQpJacobian()
{
  return _diffusivity[_qp] * Diffusion::computeQpJacobian();
}
```

Listing C.18: buck/include/kernels/BubbleBase.h

```
#ifndef BUBBLEBASE_H
#define BUBBLEBASE_H

#include "Kernel.h"

#include "BuckUtils.h"

class BubbleBase;

template<>
InputParameters validParams<BubbleBase>();

class BubbleBase : public Kernel
{
public:
  BubbleBase(const std::string & name, InputParameters parameters);

protected:
  virtual Real computeQpResidual();
  virtual Real computeQpJacobian();

  virtual void calcLosses(Real & losses, const bool jac){}
  virtual void calcGains(Real & gains, const bool jac){}
  virtual void displayBubbleInfo();

  std::vector<VariableName> _names;
  const NonlinearVariableName _this_var;

  std::vector<VariableValue *> _c;
  std::vector<VariableValue *> _r;
  std::vector<Real> _atoms;
  std::vector<Real> _widths;

  unsigned int _G;
  int _g;
};

#endif //BUBBLEBASE_H
```

buck/src/kernels/BubbleBase.C

```
#include "BubbleBase.h"

template<>
InputParameters validParams<BubbleBase>()
{
  InputParameters params = validParams<Kernel>();

  params.addRequiredCoupledVar("coupled_conc", "List of coupled concentration variables.")
    ;
  params.addRequiredCoupledVar("coupled_rad", "List of coupled radius variables.");
  params.addRequiredParam<std::vector<Real> >("coupled_atoms", "List of atom sizes for
    coupled variables.");
  params.addRequiredParam<std::vector<Real> >("coupled_widths", "List of group sizes");

  return params;
}


BubbleBase::BubbleBase(const std::string & name, InputParameters parameters)
  :Kernel(name,parameters),
  _names(getParam<std::vector<VariableName> >("coupled_conc")),
  _this_var(getParam<NonlinearVariableName>("variable")),
  _atoms(getParam<std::vector<Real> >("coupled_atoms")),
```

```
  _widths(getParam<std::vector<Real> >("coupled_widths"))
{
  _G = coupledComponents("coupled_conc");
  if ( _G != coupledComponents("coupled_rad") )
    mooseError("From BubbleBase: The number of coupled concentrations does not match
    coupled radii.");
  if ( _G != _atoms.size() )
    mooseError("From BubbleBase: The number of coupled concentrations does not match atom
    sizes list.");

  for ( unsigned int i=0; i<_G; ++i )
  {
    _c.push_back( &coupledValue("coupled_conc", i) );
    _r.push_back( &coupledValue("coupled_rad", i) );
  }

  // Determine which group current kernel acts on
  _g = -1;
  for ( unsigned int i=0; i<_G; ++i )
  {
    if ( _names[i].compare(_this_var) == 0 )
    {
      _g = i;
      break;
    }
  }
  if (_g == -1)
    mooseError("From BubbleBase: Variable not found in coupled_conc list. Check the list."
    );

  mooseDoOnce( displayBubbleInfo() );
}


Real
BubbleBase::computeQpResidual()
{
  Real losses(0);
  Real gains(0);
  calcLosses(losses, false);
  calcGains(gains, false);

  return -( gains - losses ) * _test[_i][_qp];
}


Real
BubbleBase::computeQpJacobian()
{
  Real losses(0);
  Real gains(0);
  calcLosses(losses, true);
  calcGains(gains, true);

  return -( gains - losses ) * _phi[_j][_qp] * _test[_i][_qp];
}


void
BubbleBase::displayBubbleInfo()
{
  std::cout.precision(6);
  std::cout << std::scientific;
  std::cout << "=====================================\n";
  std::cout << "    --< BUCK Bubble Information >--     \n";
  std::cout << "=====================================\n";
```

```
85    std::cout << " group\t| avg atoms\t| width\n";
      std::cout << "--------+---------------+--------------\n";
      for (int i=0; i<_G; ++i)
        std::cout << " " << i+1 << "\t| " << _atoms[i] << "\t| " << _widths[i] << "\n";
      std::cout << "=========================================\n" << std::endl;
90  }
```

Listing C.19: buck/include/kernels/BubbleFFNucleation.h

```
#ifndef BUBBLEFFNUCLEATION_H
#define BUBBLEFFNUCLEATION_H

#include "BubbleBase.h"

class BubbleFFNucleation;

template<>
InputParameters validParams<BubbleFFNucleation>();

class BubbleFFNucleation : public BubbleBase
{
public:
  BubbleFFNucleation(const std::string & name, InputParameters parameters);

protected:
  virtual void calcLosses(Real & losses, const bool jac);
  virtual void calcGains(Real & gains, const bool jac);

private:
  VariableValue & _frd;
  const Real _factor;
  const int _num;
  const int _size;
  const Real _upper;
  const Real _lower;
};

#endif
```

buck/src/kernels/BubbleFFNucleation.C

```
#include "BubbleFFNucleation.h"

#include "BuckUtils.h"

template<>
InputParameters validParams<BubbleFFNucleation>()
{
  InputParameters params = validParams<BubbleBase>();

  params.addRequiredCoupledVar("fission_rate", "Fission Rate Density");
  params.addParam<Real>("factor", 1.0, "User supplied multiplier.");
  params.addParam<int>("number", 5, "Number of bubbles created per fission");
  params.addParam<int>("size", 4, "Size of bubbles created");
  params.addParam<Real>("upper", 1e7, "Upper deadband limit");
  params.addParam<Real>("lower", 1e6, "Lower deadband limit");

  return params;
}


BubbleFFNucleation::BubbleFFNucleation(const std::string & name, InputParameters
    parameters)
  :BubbleBase(name,parameters),
  _frd(coupledValue("fission_rate")),
  _factor(getParam<Real>("factor")),
  _num(getParam<int>("number")),
  _size(getParam<int>("size")),
  _upper(getParam<Real>("upper")),
  _lower(getParam<Real>("lower"))
{
}
```

```
void
BubbleFFNucleation::calcLosses(Real & losses, bool jac)
{
  if (_g != 0)
    return;
  if (jac)
    return;
  if ( (*_c[0])[_qp] < _upper )
  {
    if ( (*_c[_size-1])[_qp] == 0 )
      return;
    else if ( (*_c[0])[_qp] < _lower )
      return;
  }

  losses += _factor * _num * _size * _frd[_qp];
}


void
BubbleFFNucleation::calcGains(Real & gains, bool jac)
{
  if ( _g != _size - 1)
    return;
  if (jac)
    return;
  if ( (*_c[0])[_qp] < _upper )
  {
    if ( (*_c[_size-1])[_qp] == 0 )
      return;
    else if ( (*_c[0])[_qp] < _lower )
      return;
  }

  gains += _factor * _num * _frd[_qp];
}
```

Listing C.20: buck/include/kernels/BubbleGrowth.h

```
#ifndef BUBBLEGROWTH_H
#define BUBBLEGROWTH_H

#include "BubbleBase.h"

class BubbleGrowth;

template<>
InputParameters validParams<BubbleGrowth>();

class BubbleGrowth : public BubbleBase
{

public:
  BubbleGrowth(const std::string & name, InputParameters parameters);

protected:
  virtual void calcLosses(Real & losses, const bool jac);
  virtual void calcGains(Real & gains, const bool jac);

private:
  void calcLossesExperimental(Real & losses, const bool jac);
  void calcGainsExperimental(Real & gains, const bool jac);

  bool _allow_loss;
  MaterialProperty<Real> & _Dg;

  const bool _exp;
};

#endif //BUBBLEGROWTH_H
```

buck/src/kernels/BubbleGrowth.C

```
#include "BubbleGrowth.h"

template<>
InputParameters validParams<BubbleGrowth>()
{
  InputParameters params = validParams<BubbleBase>();

  params.addParam<bool>("allow_loss", false, "Flag to allow losses from the largest bubble
     group.");
  params.addParam<bool>("experimental", false, "Flag to use experimental formulations.");

  return params;
}

BubbleGrowth::BubbleGrowth(const std::string & name, InputParameters parameters)
  :BubbleBase(name,parameters),
  _allow_loss(getParam<bool>("allow_loss")),
  _Dg(getMaterialProperty<Real>("gas_diffusivity")),
  _exp(getParam<bool>("experimental"))
{
}

void
BubbleGrowth::calcLosses(Real & losses, bool jac)
{
  if (_exp)
  {
    calcLossesExperimental(losses, jac);
    return;
  }
```

```
30
  if ( !_allow_loss && _g==_G-1 ) return;// Don't allow losses if largest bubble size

  if (_g == 0)
  {
35  for ( unsigned int i=1; i<_G; ++i )
    {
      if ( !_allow_loss && i==_G-1 ) break;

      Real k = 4.0 * M_PI * _Dg[_qp] * (*_r[i])[_qp] * (*_c[i])[_qp] * _widths[i];
40    if (!jac)
        losses += k * _u[_qp];
      else
        losses += k;
    }
45 }
  else
  {
    Real k = 4.0 * M_PI * _Dg[_qp] * (*_r[_g])[_qp] * (*_c[0])[_qp];
    if (!jac)
50    losses += k * _u[_qp];
    else
      losses += k;
  }
}
55
void
BubbleGrowth::calcGains(Real & gains, bool jac)
{
  if (_exp)
60 {
    calcGainsExperimental(gains, jac);
    return;
  }

65 if ( _g==0 || _g==1 ) return; // Don't count gains if single atom bubble or dimer
  if (jac) return;

  gains += 4.0 * M_PI * _Dg[_qp] * (*_r[_g-1])[_qp] * (*_c[_g-1])[_qp] * (*_c[0])[_qp];
}
70
void
BubbleGrowth::calcLossesExperimental(Real & losses, bool jac)
{
}
75


void
BubbleGrowth::calcGainsExperimental(Real & gains, bool jac)
{
80 }
```

Listing C.21: buck/include/kernels/BubbleKnockout.h

```
#ifndef BUBBLEKNOCKOUT_H
#define BUBBLEKNOCKOUT_H

#include "BubbleBase.h"

class BubbleKnockout;

template<>
InputParameters validParams<BubbleKnockout>();

class BubbleKnockout : public BubbleBase
{
public:
  BubbleKnockout(const std::string & name, InputParameters parameters);

protected:
  virtual void calcLosses(Real & losses, const bool jac);
  virtual void calcGains(Real & gains, const bool jac);

private:
  Real calcKnockoutRate(const int i);
  Real calcB(const Real r);

  const Real _factor;
  VariableValue  & _fsn_rate_den;
  Real _b;
};

#endif
```

buck/src/kernels/BubbleKnockout.C

```
#include "BubbleKnockout.h"

template<>
InputParameters validParams<BubbleKnockout>()
{
  InputParameters params = validParams<BubbleBase>();

  params.addParam<Real>("factor", 1.0, "User supplied multiplier.");
  params.addCoupledVar("fission_rate", 0, "Variable for fission rate density.");
  params.addParam<Real>("b", -1, "Value to set constant knockout parameter. B is
    automatically calculated if not given");

  return params;
}


BubbleKnockout::BubbleKnockout(const std::string & name, InputParameters parameters)
  :BubbleBase(name,parameters),
  _factor(getParam<Real>("factor")),
  _fsn_rate_den(coupledValue("fission_rate")),
  _b(getParam<Real>("b"))
{
}


void
BubbleKnockout::calcLosses(Real & losses, bool jac)
{
  if (_g == 0) return;

  Real k = calcKnockoutRate(_g);
```

```
  if (!jac)
    losses += k * _u[_qp];
  else
    losses += k;
}


void
BubbleKnockout::calcGains(Real & gains, bool jac)
{
  if (_g == _G-1) return;
  if (jac) return;

  if (_g==0)
  {
    for (int i=1; i<_G; ++i)
    {
      Real k = calcKnockoutRate(i);
      if (i==1)
        k *= 2.0; // needed since split dimer results in two single atoms
      gains += k * (*_c[i])[_qp] * _widths[i-1];
    }
  }
  else
  {
    Real k = calcKnockoutRate(_g+1);
    gains += k * (*_c[_g+1])[_qp];
  }
}


Real
BubbleKnockout::calcKnockoutRate(int i)
{
  Real b = calcB( (*_r[i])[_qp] );
  Real frd = _fsn_rate_den[_qp] * 1.0e18;

  return _factor * b * frd * _atoms[i];
}


Real
BubbleKnockout::calcB(const Real r)
{
  if ( _b >= 0 )
    return _b;

  Real a = 0.02831;
  Real b =-0.0803;
  Real c =-0.149;

  Real logr = std::log10(r);
  Real right = a * std::pow( logr, 2.0 ) + b * logr + c;
  return std::pow(10.0, right) * 1e-25;
}
```

Listing C.22: buck/include/kernels/BubbleNucleation.h

```
#ifndef BUBBLENUCLEATION_H
#define BUBBLENUCLEATION_H

#include "BubbleBase.h"

class BubbleNucleation;

template<>
InputParameters validParams<BubbleNucleation>();

class BubbleNucleation : public BubbleBase
{
public:
  BubbleNucleation(const std::string & name, InputParameters parameters);

protected:

  virtual void calcLosses(Real & losses, const bool jac);
  virtual void calcGains(Real & gains, const bool jac);

private:

  VariableValue & _temp;
  const Real _a;
  const Real _omega;
  const Real _factor;

  MaterialProperty<Real> & _Dg;

  const Real _Z11;
};

#endif //BUBBLENUCLEATION_H
```

buck/src/kernels/BubbleNucleation.C

```
#include "BubbleNucleation.h"

#include "BuckUtils.h"

template<>
InputParameters validParams<BubbleNucleation>()
{
  InputParameters params = validParams<BubbleBase>();

  params.addRequiredCoupledVar("temp", "Temperature");
  params.addParam<Real>("a", 4.96e-4, "Lattice Parameter [um]");
  params.addParam<Real>("omega", 1.53e-11, " Atomic volume [um^3]");
  params.addParam<Real>("factor", 1.0, "User supplied multiplier.");

  return params;
}


BubbleNucleation::BubbleNucleation(const std::string & name, InputParameters parameters)
  :BubbleBase(name,parameters),
  _temp(coupledValue("temp")),
  _a(getParam<Real>("a")),
  _omega(getParam<Real>("omega")),
  _factor(getParam<Real>("factor")),

  _Dg(getMaterialProperty<Real>("gas_diffusivity")),

  _Z11(168.0)
```

```
{
  if ( _g > 1 )
    mooseError("In BubbleNucleation: Cannont implement on non-dimer or non-single atoms.")
    ;
}


void
BubbleNucleation::calcLosses(Real & losses, bool jac)
{
  if (_g != 0)
    return;

  Real P11 = _factor * _Z11 * _omega * _Dg[_qp] * _u[_qp] / std::pow(_a, 2.0);

  if (!jac)
    losses += 2.0 * P11 * _u[_qp];
  else
    losses += 4.0 * P11;
}


void
BubbleNucleation::calcGains(Real & gains, bool jac)
{
  if ( _g != 1 )
    return;
  if (jac)
    return;

  Real R = _factor * _Z11 * _omega * _Dg[_qp] * std::pow( (*_c[0])[_qp]/_a, 2.0 );

  gains += R;
}
```

Listing C.23: buck/include/kernels/VariableScaledSource.h

```
#ifndef VARIABLESCALEDSOURCE_H
#define VARIABLESCALEDSOURCE_H

#include "Kernel.h"

//Forward Declarations
class VariableScaledSource;

template<>
InputParameters validParams<VariableScaledSource>();

class VariableScaledSource : public Kernel
{
public:

  VariableScaledSource(const std::string & name, InputParameters parameters);

protected:
  virtual Real computeQpResidual();
  virtual Real computeQpJacobian();

  Real _factor;
  VariableValue  & _var;
};

#endif
```

buck/src/kernels/VariableScaledSource.C

```
#include "VariableScaledSource.h"

template<>
InputParameters validParams<VariableScaledSource>()
{
  InputParameters params = validParams<Kernel>();

  params.addParam<Real>("factor", 1, "Number multiplied scaling variable.");
  params.addCoupledVar("scaling_variable", 1, "Variable for Scaling");

  return params;
}


VariableScaledSource::VariableScaledSource(const std::string & name, InputParameters
    parameters) :
    Kernel(name, parameters),
    _factor(getParam<Real>("factor")),
    _var(coupledValue("scaling_variable"))
{
}


Real
VariableScaledSource::computeQpResidual()
{
  return -_test[_i][_qp] * _factor * _var[_qp];
}


Real
VariableScaledSource::computeQpJacobian()
{
  return 0.0;
}
```

Listing C.24: buck/include/materials/GasAtomDiffusivity.h

```
#ifndef GASATOMDIFFUSIVITY_H
#define GASATOMDIFFUSIVITY_H

#include "Material.h"

class GasAtomDiffusivity;

template<>
InputParameters validParams<GasAtomDiffusivity>();

class GasAtomDiffusivity : public Material
{
public:
  GasAtomDiffusivity(const std::string & name,
                     InputParameters parameters);

protected:
  virtual void computeQpProperties();

  VariableValue & _temp;
  VariableValue & _fission_rate;

  Real _D0;
  Real _Q;
  Real _D0f;
  Real _Qf;
  const Real _R;
  const Real _factor;
  const int _model;
  int _G;

  MaterialProperty<Real> & _gas_diffusivity;
};

#endif // GASATOMDIFFUSIVITY_H
```

buck/src/materials/GasAtomDiffusivity.C

```
#include "GasAtomDiffusivity.h"
#include "MooseEnum.h"

template<>
InputParameters validParams<GasAtomDiffusivity>()
{
  InputParameters params = validParams<Material>();

  params.addRequiredCoupledVar("temp", "Coupled Temperature");
  params.addCoupledVar("fission_rate", 0, "Coupled fission rate");
  params.addParam<Real>("D0", "Diffusion coefficient [um^2/s]");
  params.addParam<Real>("Q", "Activation energy [J/mol]");
  params.addParam<Real>("D0f", 0, "Fission enhanced diffusion coefficient [um^2/s]");
  params.addParam<Real>("Qf", 0, "Fission enhanced activation energy [J/mol]");
  params.addParam<Real>("R", 8.31446, "Ideal gas constant [J/(K*mo)]");
  params.addParam<Real>("factor", 1, "Scaling factor to multiply by diffusivity.");
  params.addParam<int>("model", 1, "Switch for diffusion coefficient model (0=user input,
    1=UC Matzke, 2=UC Madrid, 3=UC Eyre, 4=UC Ronchi, 5=UO2 Griesmeyer");

  return params;
}


GasAtomDiffusivity::GasAtomDiffusivity(const std::string & name, InputParameters
    parameters) :
  Material(name, parameters),
```

```
  _temp(coupledValue("temp")),
  _fission_rate(coupledValue("fission_rate")),
  _R(getParam<Real>("R")),
  _factor(getParam<Real>("factor")),
  _model(getParam<int>("model")),
  _gas_diffusivity(declareProperty<Real>("gas_diffusivity"))
{
  if ( _model == 0 )
  {
    if ( !isParamValid("D0") || !isParamValid("Q") )
      mooseError("In GasAtomDiffusivity: if model = 0 (user supplied), D0 and Q must also
    be supplied");
    _D0 = getParam<Real>("D0");
    _Q = getParam<Real>("Q");
    _D0f = getParam<Real>("D0f");
    _Qf = getParam<Real>("Qf");
  }
  else
  {
    if ( isParamValid("D0") || isParamValid("Q") )
      mooseError("In GasAtomDiffusivity: D0 and Q are supplied, model must = 0");

    if ( _model == 1 ) // UC Matzke
    {
      _D0 = 0.3e8;
      _Q = 355000.0;
    }
    else if ( _model == 2 ) // UC Madrid
    {
      _D0 = 4.6e5;
      _Q = 326360.0;
    }
    else if ( _model == 3 ) // UC Eyre
    {
      _D0 = 1.66e-1;
      _Q = 221154.0;
    }
    else if ( _model == 4 ) // UC Ronchi
    {
      _D0 = 4.6e7;
      _Q = 328421.0;
      _D0f = 1.3e-9;
    }
    else if ( _model == 5 ) // UO2 Griesmeyer
    {
      _D0 = 2.1e4;
      _Q = 381000.0;
      _D0f = 1.e-4;
      _Qf = 26.36;
    }
    else
      mooseError("In GasAtomDiffusivity: Invalid model value given.");
  }
}


void
GasAtomDiffusivity::computeQpProperties()
{
  Real diff_thermal = _D0 *  std::exp( -_Q  / _R / _temp[_qp] );
  Real diff_fission = _D0f * std::exp( -_Qf / _R / _temp[_qp] ) * _fission_rate[_qp];

  _gas_diffusivity[_qp] = (diff_thermal + diff_fission) * _factor;
}
```

Listing C.25: buck/include/materials/MaterialXeBubble.h

```
#ifndef MATERIALXEBUBBLE_H
#define MATERIALXEBUBBLE_H

namespace MaterialXeBubble {

  double VDW_RtoRho(double R, double T, double sigma);
  double VDW_RtoP(double R, double T, double m);
  double VDW_MtoR(double m, double T, double sigma, double gamma, double B, bool testing);

}

#endif // MATERIALXEBUBBLE_H
```

buck/src/materials/MaterialXeBubble.C

```
#include "MaterialXeBubble.h"

#include "MooseError.h"

#include <iostream>
#include <cmath>

namespace MaterialXeBubble{

  double VDW_RtoRho(double R, double T, double sigma)
  {
    // Calculates the atomic density in a bubble based on Van der Waal's EOS
    //
    // 1/rho = B + 1/(2*gamma/k/T/R + sigma/k/T)
    //
    // R = Bubble radius, [m]
    // T = Temperature, [K]
    // sigma = Hydrostatic stress, [Pa]
    // B = VDW constant, [m^3/atom]
    // k = boltzmann constant, [J/K]
    // gamma = surface tension, N/m

    double B = 8.5e-29;
    double k = 1.3806488e-23;
    double gamma = 1.0;

    double invrho = ( B + 1.0 / (2.0*gamma/k/T/R + sigma/k/T) );

    return 1.0/invrho;
  }

  ////////////////////////////////////////////////////////////////////////

  double VDW_RtoP(double rad, double T, double m)
  {
    // Calculates the bubble preessure in a bubble based on Van der Waal's EOS
    //
    // p = kT/(1/rho - B)
    //
    // T = Temperature, [K]
    // rad = bubble radius, [m]
    // m = number of atoms
    // k = Boltzmann constant, [J/K]
    // rho = Bubble atomic density, [atom/m3]
    // B = constant, [m3/atom]

    double k = 1.3806488e-23; // [J/K]
    double B = 8.469e-29; // [m3/atom]
    double V = 4.0/3.0 * M_PI * std::pow(rad,3);
```

```
  double rho = m/V;

  double p = k*T/(1.0/rho - B);

  if (p<0)
  {
    // p = 1e50;
    std::cout << "rad: " << rad << " T: " << T << " m: " << m << " VDW pressure: " << p
  << std::endl;
    mooseError("In MaterialXeBubble: Negative bubble pressure calculated");
  }

  return p;
}

////////////////////////////////////////////////////////////////////////

double VDW_MtoR(double m, double T, double sigma, double gamma, double B, bool testing)
{
  // Uses a simple Newton's method to determine equilbrium bubble radius as a function
  of atoms
  //
  // 1/rho = B + 1/(2*gamma/k/T/R + sigma/k/T)
  //
  // R = Bubble radius, [m]
  // T = Temperature, [K]
  // sigma = Hydrostatic stress, [Pa]
  // B = VDW constant, [m^3/atom]
  // k = boltzmann constant, [J/K]
  // gamma = surface tension, N/m

  // Physical Paramters
  // double B = 8.5e-29;
  double k = 1.3806488e-23;
  // double gamma = 1.0;

  // Calculation setup
  double A = 4.0/3.0 * M_PI;
  double C = 2.0*gamma/k/T;
  double D = sigma/k/T;

  // Iteration paramters
  int max_its = 100;      // Max iterations
  double rel_conv = 1e-5; // Relative convergence criteria.
  int it(0);              // Iteration counter
  double dR(1);           // Newton change in R

  if (testing)
  {
    std::cout << "m: " << m << "\tT: " << T << "\tsigma: " << sigma << std::endl;
    std::cout << "B: " << B << "\tk: " << k << "\tgamma: " << gamma << std::endl;
  }

  double R = 1.0e-9; // bubble radius initial guess
  while( it<max_its && std::abs(dR/R) > rel_conv )
  {
    double RR = R*R;
    double RRR = R*RR;
    double RRRR = R*RRR;
    double res = A*D*RRRR + A*C*RRR - m*(B*D+1)*R - B*C*m;
    double dres = 4.0*A*D*RRR + 3.0*A*C*RR - m*(B*D+1);

    dR = res/dres;
    double new_R = R - dR;

    R = new_R;
```

```
      ++it;

      if (R<0)
      {
        R = 1;
        dR = 1;
        if (testing)
          std::cout << "R dropped below 0, reseting calculation by R=1" << std::endl;
      }

      if (testing)
      {
        std::cout << "it: " << it << "\tR: " << R << "\tdR: " << res/dres << "\tnew_R: "
    << new_R << "\tconv: " << std::abs(dR/R) << std::endl;
      }

    } // end while iteration loop

    if (testing)
    {
      double rho = m/(4.0/3.0*M_PI*std::pow(R,3));
      double calc_rho = VDW_RtoRho(R, T, sigma);
      std::cout << std::endl << "rho: " << rho << "\tcalc_rho: " << calc_rho << "\t%% diff
    : " << std::abs(rho-calc_rho)/calc_rho*100.0 << std::endl;
    }

    return R;
  }
}
```

Listing C.26: buck/include/parser/BuckSyntax.h

```
#ifndef BUCKSYNTAX_H
#define BUCKSYNTAX_H

#include "Factory.h"
#include "Syntax.h"
#include "ActionFactory.h"

namespace Buck
{
  void associateSyntax(Syntax & syntax, ActionFactory & action_factory);
}

#endif // BUCKSYNTAX_H
```

buck/src/parser/BuckSyntax.C

```
#include "BuckSyntax.h"
#include "ActionFactory.h"

// Actions
#include "BubblesConcVarsAction.h"
#include "BubblesConcTimeKernelAction.h"

#include "BubblesRadAuxVarsAction.h"
#include "BubblesRadAuxKernelAction.h"

#include "BubblesPostprocessorsAction.h"

#include "BubblesCoalescenceKernelsAction.h"
#include "BubblesGrowthKernelsAction.h"
#include "BubblesNucleationKernelsAction.h"
#include "BubblesFFNucleationKernelsAction.h"
#include "BubblesKnockoutKernelsAction.h"

#include "BubblesDampersAction.h"

namespace Buck
{

  void
  associateSyntax(Syntax & syntax, ActionFactory & action_factory)
  {
    syntax.registerActionSyntax("BubblesConcVarsAction", "Bubbles/Conc/", "add_variable");
    syntax.registerActionSyntax("BubblesConcVarsAction", "Bubbles/Conc/", "add_ic");
    syntax.registerActionSyntax("BubblesConcTimeKernelAction", "Bubbles/Conc/", "
    add_kernel");
    registerAction(BubblesConcVarsAction, "add_variable");
    registerAction(BubblesConcVarsAction, "add_ic");
    registerAction(BubblesConcTimeKernelAction, "add_kernel");

    syntax.registerActionSyntax("BubblesPostprocessorsAction", "Bubbles/PPs/", "
    add_postprocessor");
    registerAction(BubblesPostprocessorsAction, "add_postprocessor");

    syntax.registerActionSyntax("BubblesGrowthKernelsAction", "Bubbles/Growth/", "
    add_kernel");
    registerAction(BubblesGrowthKernelsAction, "add_kernel");

    syntax.registerActionSyntax("BubblesKnockoutKernelsAction", "Bubbles/Knockout/", "
    add_kernel");
    registerAction(BubblesKnockoutKernelsAction, "add_kernel");

    syntax.registerActionSyntax("BubblesCoalescenceKernelsAction", "Bubbles/Coalescence/",
     "add_kernel");
```

```
    registerAction(BubblesCoalescenceKernelsAction, "add_kernel");

    syntax.registerActionSyntax("BubblesNucleationKernelsAction", "Bubbles/Nucleation/", "
    add_kernel");
    registerAction(BubblesNucleationKernelsAction, "add_kernel");

    syntax.registerActionSyntax("BubblesFFNucleationKernelsAction", "Bubbles/FFNucleation/
    ", "add_kernel");
    registerAction(BubblesFFNucleationKernelsAction, "add_kernel");

    syntax.registerActionSyntax("BubblesDampersAction", "Bubbles/Dampers/", "add_damper");
    registerAction(BubblesDampersAction, "add_damper");

    syntax.registerActionSyntax("BubblesRadAuxVarsAction", "Bubbles/Rad/", "
    add_aux_variable");
    syntax.registerActionSyntax("BubblesRadAuxKernelAction", "Bubbles/Rad/Eq/", "
    add_aux_kernel");
    registerAction(BubblesRadAuxVarsAction, "add_aux_variable");
    registerAction(BubblesRadAuxKernelAction, "add_aux_kernel");
  }

} //end namespace
```

Listing C.27: buck/include/postprocessors/BoundedElementAverage.h

```
#ifndef BOUNDEDELEMENTAVERAGE_H
#define BOUNDEDELEMENTAVERAGE_H

#include "ElementAverageValue.h"

class BoundedElementAverage;

template<>
InputParameters validParams<BoundedElementAverage>();

class BoundedElementAverage : public ElementAverageValue
{
public:
  BoundedElementAverage(const std::string & name, InputParameters parameters);

  virtual void execute();

};

#endif // BOUNDEDELEMENTAVERAGE_H
```

buck/src/postprocessors/BoundedElementAverage.C

```
#include "BoundedElementAverage.h"

template<>
InputParameters validParams<BoundedElementAverage>()
{
  InputParameters params = validParams<ElementAverageValue>();

  params.addParam<Real>("upper", "The upper bound for the variable");
  params.addParam<Real>("lower", "The lower bound for the variable");

  return params;
}

BoundedElementAverage::BoundedElementAverage(const std::string & name, InputParameters
    parameters) :
  ElementAverageValue(name, parameters)
{}

void
BoundedElementAverage::execute()
{
  for (_qp=0; _qp<_qrule->n_points(); _qp++)
  {
    if ( isParamValid("upper") )
    {
      if ( _u[_qp] > getParam<Real>("upper") )
        mooseError("From Postprocessor "<< _name << ": value is greater than upper limit
    of " << getParam<Real>("upper") );
    }
    if ( isParamValid("lower") )
    {
      if ( _u[_qp] < getParam<Real>("lower") )
        mooseError("From Postprocessor "<< _name << ": value is lower than lower limit of
    " << getParam<Real>("lower") );
    }
  }
  ElementAverageValue::execute();
}
```

Listing C.28: buck/include/postprocessors/C1LossPostprocessor.h

```
#ifndef C1LOSSPOSTPROCESSOR_H
#define C1LOSSPOSTPROCESSOR_H

#include "ElementAverageValue.h"

class C1LossPostprocessor: public ElementAverageValue
{

public:
  C1LossPostprocessor(const std::string & name, InputParameters parameters);

protected:
  virtual Real computeQpIntegral();

private:
  Real calcKnockoutRate();
  Real calcB(const Real r);
  VariableValue & _r;
  VariableValue & _c1;
  VariableValue & _frd;
  MaterialProperty<Real> & _Dg;
  const Real _width;
  const Real _atoms;
  Real _b;
  const Real _factor;
};

template<>
InputParameters validParams<C1LossPostprocessor>();

#endif
```

buck/src/postprocessors/C1LossPostprocessor.C

```
#include "C1LossPostprocessor.h"

template<>
InputParameters validParams<C1LossPostprocessor>()
{
  InputParameters params = validParams<ElementAverageValue>();

  params.addRequiredCoupledVar("r", "radius variable.");
  params.addRequiredCoupledVar("c1", "c1");
  params.addRequiredCoupledVar("fission_rate", "fission rate density");
  params.addRequiredParam<Real>("width", "width");
  params.addRequiredParam<Real>("atoms", "atoms");
  params.addParam<Real>("b", -1, "Value to set constant knockout parameter. B is
    automatically calculated if not given");
  params.addParam<Real>("factor", 1.0, "Number multiplied scaling variable.");

  return params;
}


C1LossPostprocessor::C1LossPostprocessor(const std::string & name, InputParameters
    parameters) :
  ElementAverageValue(name, parameters),
  _r(coupledValue("r")),
  _c1(coupledValue("c1")),
  _frd(coupledValue("fission_rate")),
  _Dg(getMaterialProperty<Real>("gas_diffusivity")),
  _width(getParam<Real>("width")),
  _atoms(getParam<Real>("atoms")),
  _b(getParam<Real>("b")),
```

```
  _factor(getParam<Real>("factor"))
{
}


Real
C1LossPostprocessor::computeQpIntegral()
{
  Real growth = 4.0 * M_PI * _Dg[_qp] * _r[_qp] * _u[_qp] * _width * _c1[_qp];

  Real k = calcKnockoutRate();
  Real knockout = k * _u[_qp] * _width;

  return std::abs(growth) + std::abs(knockout);
}


Real
C1LossPostprocessor::calcKnockoutRate()
{
  Real b = calcB( _r[_qp] );
  Real frd = _frd[_qp] * 1.0e18;

  return _factor * b * frd * _atoms;
}


Real
C1LossPostprocessor::calcB(const Real r)
{
  if ( _b >= 0 )
    return _b;

  Real a = 0.02831;
  Real b =-0.0803;
  Real c =-0.149;

  Real logr = std::log10(r);
  Real right = a * std::pow( logr, 2.0 ) + b * logr + c;
  return std::pow(10.0, right) * 1e-25;
}
```

Listing C.29: buck/include/postprocessors/GainRatePostprocessor.h

```
#ifndef GAINRATEPOSTPROCESSOR_H
#define GAINRATEPOSTPROCESSOR_H

#include "ElementAverageValue.h"

class GainRatePostprocessor : public ElementAverageValue
{

public:
  GainRatePostprocessor(const std::string & name, InputParameters parameters);

protected:
  virtual Real computeQpIntegral();

private:
  VariableValue & _r;
  VariableValue & _c1;
  MaterialProperty<Real> & _Dg;
  const Real _width;
};

template<>
InputParameters validParams<GainRatePostprocessor>();

#endif
```

buck/src/postprocessors/GainRatePostprocessor.C

```
#include "GainRatePostprocessor.h"

template<>
InputParameters validParams<GainRatePostprocessor>()
{
  InputParameters params = validParams<ElementAverageValue>();

  params.addRequiredCoupledVar("r", "radius variable.");
  params.addRequiredCoupledVar("c1", "c1");
  params.addRequiredParam<Real>("width", "width");

  return params;
}


GainRatePostprocessor::GainRatePostprocessor(const std::string & name, InputParameters
    parameters) :
  ElementAverageValue(name, parameters),
  _r(coupledValue("r")),
  _c1(coupledValue("c1")),
  _Dg(getMaterialProperty<Real>("gas_diffusivity")),
  _width(getParam<Real>("width"))
{
}


Real
GainRatePostprocessor::computeQpIntegral()
{
  Real growth = 4.0 * M_PI * _Dg[_qp] * _r[_qp] * _u[_qp] * _width * _c1[_qp];
  return growth;
}
```

Listing C.30: buck/include/postprocessors/GrainBoundaryGasFlux.h

```
#ifndef GRAINBOUNDARYGASFLUX_H
#define GRAINBOUNDARYGASFLUX_H

#include "SideIntegralVariablePostprocessor.h"

class GrainBoundaryGasFlux;

template<>
InputParameters validParams<GrainBoundaryGasFlux>();

class GrainBoundaryGasFlux : public SideIntegralVariablePostprocessor
{
public:
  GrainBoundaryGasFlux(const std::string & name, InputParameters parameters);

protected:
  virtual Real computeQpIntegral();

  MaterialProperty<Real> & _diffusivity;
};

#endif // GRAINBOUNDARYGAS_H
```

buck/src/postprocessors/GrainBoundaryGasFlux.C

```
#include "GrainBoundaryGasFlux.h"

template<>
InputParameters validParams<GrainBoundaryGasFlux>()
{
  InputParameters params = validParams<SideIntegralVariablePostprocessor>();
  return params;
}


GrainBoundaryGasFlux::GrainBoundaryGasFlux(const std::string & name, InputParameters
    parameters) :
    SideIntegralVariablePostprocessor(name, parameters),
    _diffusivity(getMaterialProperty<Real>("atomic_diffusivity"))
{
}


Real
GrainBoundaryGasFlux::computeQpIntegral()
{
  return -_diffusivity[_qp]*_grad_u[_qp]*_normals[_qp];
}
```

Listing C.31: buck/include/postprocessors/KnockoutRatePostprocessor.h

```
#ifndef KNOCKOUTRATEPOSTPROCESSOR_H
#define KNOCKOUTRATEPOSTPROCESSOR_H

#include "ElementAverageValue.h"

class KnockoutRatePostprocessor: public ElementAverageValue
{

public:
  KnockoutRatePostprocessor(const std::string & name, InputParameters parameters);

protected:
  virtual Real computeQpIntegral();

private:
  Real calcKnockoutRate();
  Real calcB(const Real r);
  VariableValue & _r;
  VariableValue & _c1;
  VariableValue & _frd;
  MaterialProperty<Real> & _Dg;
  const Real _width;
  const Real _atoms;
  Real _b;
  const Real _factor;
};

template<>
InputParameters validParams<KnockoutRatePostprocessor>();

#endif
```

buck/src/postprocessors/KnockoutRatePostprocessor.C

```
#include "KnockoutRatePostprocessor.h"

template<>
InputParameters validParams<KnockoutRatePostprocessor>()
{
  InputParameters params = validParams<ElementAverageValue>();

  params.addRequiredCoupledVar("r", "radius variable.");
  params.addRequiredCoupledVar("c1", "c1");
  params.addRequiredCoupledVar("fission_rate", "fission rate density");
  params.addRequiredParam<Real>("width", "width");
  params.addRequiredParam<Real>("atoms", "atoms");
  params.addParam<Real>("b", -1, "Value to set constant knockout parameter. B is
    automatically calculated if not given");
  params.addParam<Real>("factor", 1.0, "Number multiplied scaling variable.");

  return params;
}


KnockoutRatePostprocessor::KnockoutRatePostprocessor(const std::string & name,
    InputParameters parameters) :
  ElementAverageValue(name, parameters),
  _r(coupledValue("r")),
  _c1(coupledValue("c1")),
  _frd(coupledValue("fission_rate")),
  _Dg(getMaterialProperty<Real>("gas_diffusivity")),
  _width(getParam<Real>("width")),
  _atoms(getParam<Real>("atoms")),
  _b(getParam<Real>("b")),
```

```
  _factor(getParam<Real>("factor"))
{
}


Real
KnockoutRatePostprocessor::computeQpIntegral()
{
  Real k = calcKnockoutRate();
  Real knockout = k * _u[_qp] * _width;

  return knockout;
}


Real
KnockoutRatePostprocessor::calcKnockoutRate()
{
  Real b = calcB( _r[_qp] );
  Real frd = _frd[_qp] * 1.0e18;

  return _factor * b * frd * _atoms;
}


Real
KnockoutRatePostprocessor::calcB(const Real r)
{
  if ( _b >= 0 )
    return _b;

  Real a = 0.02831;
  Real b =-0.0803;
  Real c =-0.149;

  Real logr = std::log10(r);
  Real right = a * std::pow( logr, 2.0 ) + b * logr + c;
  return std::pow(10.0, right) * 1e-25;
}
```

Listing C.32: buck/include/postprocessors/MaterialXeBubbleTester.h

```
#ifndef MATERIALXEBUBBLETESTER_H
#define MATERIALXEBUBBLETESTER_H

#include "GeneralPostprocessor.h"

class MaterialXeBubbleTester;

template<>
InputParameters validParams<MaterialXeBubbleTester>();

class MaterialXeBubbleTester : public GeneralPostprocessor
{
public:
  MaterialXeBubbleTester(const std::string & name, InputParameters parameters);

  virtual ~MaterialXeBubbleTester() {};
  virtual void initialize() {};
  virtual void execute() {};
  virtual PostprocessorValue getValue();

protected:
  const Real _temp;
  const Real _sigma;

  const PostprocessorValue & _m_mag;
};

#endif // MATERIALXEBUBBLETESTER_H
```

buck/src/postprocessors/MaterialXeBubbleTester.C

```
#include "MaterialXeBubbleTester.h"

#include "MaterialXeBubble.h"

template<>
InputParameters validParams<MaterialXeBubbleTester>()
{
  InputParameters params = validParams<GeneralPostprocessor>();

  params.addParam<Real>("temp" , 1000, "Number of atoms.");
  params.addParam<Real>("sigma", 0, "stress");

  params.addRequiredParam<PostprocessorName>("m_mag", "The postprocessor that has m order
    of magnitude");

  return params;
}


MaterialXeBubbleTester::MaterialXeBubbleTester(const std::string & name, InputParameters
    parameters) :
    GeneralPostprocessor(name, parameters),
    _temp(getParam<Real>("temp")),
    _sigma(getParam<Real>("sigma")),
    _m_mag(getPostprocessorValueByName(getParam<PostprocessorName>("m_mag")))
{
}


Real
MaterialXeBubbleTester::getValue()
{
```

```
  return MaterialXeBubble::VDW_MtoR(std::pow(10, _m_mag), _temp, _sigma, 1.0, 8.5e-29,
    true);
}
```

Listing C.33: buck/include/postprocessors/PostprocessorTerminator.h

```
#ifndef POSTPROCESSORTERMINATOR_H
#define POSTPROCESSORTERMINATOR_H

#include "GeneralPostprocessor.h"

class PostprocessorTerminator;

template<>
InputParameters validParams<PostprocessorTerminator>();

class PostprocessorTerminator : public GeneralPostprocessor
{
public:
  PostprocessorTerminator(const std::string & name, InputParameters parameters);
  virtual void initialize() {};
  virtual void execute();
  virtual Real getValue() { return 0; }
  virtual void threadJoin(const UserObject &) {};

protected:
  const Real _threshold;
  const PostprocessorValue & _value;
};

#endif
```

buck/src/postprocessors/PostprocessorTerminator.C

```
#include "PostprocessorTerminator.h"

template<>
InputParameters validParams<PostprocessorTerminator>()
{
  InputParameters params = validParams<GeneralPostprocessor>();

  params.addRequiredParam<PostprocessorName>("postprocessor", "The postprocessor name");
  params.addRequiredParam<Real>("threshold", "Threshold above which to terminate the solve
    ");

  return params;
}


PostprocessorTerminator::PostprocessorTerminator(const std::string & name, InputParameters
     parameters) :
    GeneralPostprocessor(name, parameters),
    _threshold(getParam<Real>("threshold")),
    _value(getPostprocessorValue("postprocessor"))
{
}


void
PostprocessorTerminator::execute()
{
  if (_value > _threshold)
    _fe_problem.terminateSolve();
}
```

Listing C.34: buck/include/postprocessors/SumOfPostprocessors.h

```
#ifndef SUMOFPOSTPROCESSORS_H
#define SUMOFPOSTPROCESSORS_H

#include "GeneralPostprocessor.h"

class SumOfPostprocessors;

template<>
InputParameters validParams<SumOfPostprocessors>();

class SumOfPostprocessors : public GeneralPostprocessor
{
public:

  SumOfPostprocessors(const std::string & name, InputParameters parameters);

  virtual void initialize() {};
  virtual void execute() {};
  virtual void threadJoin(const UserObject &) {};
  virtual Real getValue();

protected:
  std::vector<Real> _factors;
  std::vector<const PostprocessorValue *> _postprocessor_values;
};

#endif
```

buck/src/postprocessors/SumOfPostprocessors.C

```
#include "SumOfPostprocessors.h"
#include "PostprocessorInterface.h"

template<>
InputParameters validParams<SumOfPostprocessors>()
{
  InputParameters params = validParams<GeneralPostprocessor>();

  params.addRequiredParam<std::vector<PostprocessorName> >("postprocessors", "The
    postprocessors whose values are to be summed");
  params.addParam<std::vector<Real> >("factors", "Factors that postprocessors are
    multiplied against.");

  return params;
}


SumOfPostprocessors::SumOfPostprocessors(const std::string & name, InputParameters
    parameters) :
    GeneralPostprocessor(name, parameters),
    _factors(getParam<std::vector<Real> >("factors"))
{
  std::vector<PostprocessorName> pps_names(getParam<std::vector<PostprocessorName> >("
    postprocessors"));

  // Create vector of pps
  unsigned int _N = pps_names.size();
  for ( unsigned int i=0; i<_N; ++i )
  {
    if (!hasPostprocessorByName(pps_names[i]))
      mooseError("In SumOfPostprocessors, postprocessor with name: "<<pps_names[i]<<" does
     not exist");
    _postprocessor_values.push_back(&getPostprocessorValueByName(pps_names[i]));
  }
```

```
30
  // Create default factors if it doesn't exist
  if ( _factors.size() == 0 )
  {
    for ( unsigned int i=0; i<_N; ++i )
35    _factors.push_back(1);
  }
  else if (_factors.size() != _N)
  {
    mooseError("In SumOfPostprocessors: Size of factors does not match number of
    postprocessors.");
40  }
}


Real
45 SumOfPostprocessors::getValue()
{
  Real val(0.0);
  for ( unsigned int i=0; i<_postprocessor_values.size(); ++i)
  {
50  val += *_postprocessor_values[i] * _factors[i];
  }
  return val;
}
```

Listing C.35: buck/include/postprocessors/SwellingPostprocessor.h

```
#ifndef SWELLINGPOSTPROCESSOR_H
#define SWELLINGPOSTPROCESSOR_H

#include "ElementAverageValue.h"

class SwellingPostprocessor: public ElementAverageValue
{
public:
  SwellingPostprocessor(const std::string & name, InputParameters parameters);

protected:
  virtual Real computeQpIntegral();

private:
  VariableValue & _r;
  const Real _width;
};

template<>
InputParameters validParams<SwellingPostprocessor>();

#endif // SWELLINGPOSTPROCESSOR_H
```

buck/src/postprocessors/SwellingPostprocessor.C

```
#include "SwellingPostprocessor.h"

template<>
InputParameters validParams<SwellingPostprocessor>()
{
  InputParameters params = validParams<ElementAverageValue>();

  params.addRequiredCoupledVar("r", "radius variable.");
  params.addRequiredParam<Real>("width", "width");

  return params;
}


SwellingPostprocessor::SwellingPostprocessor(const std::string & name, InputParameters
    parameters) :
  ElementAverageValue(name, parameters),
  _r(coupledValue("r")),
  _width(getParam<Real>("width"))
{
}


Real
SwellingPostprocessor::computeQpIntegral()
{
  Real swell(0);

  swell += _u[_qp] * 4.0/3.0 * M_PI * std::pow( _r[_qp], 3.0 );

  return swell;
}
```

Listing C.36: buck/include/utils/BuckUtils.h

```
#ifndef BUCKUTILS_H
#define BUCKUTILS_H

#include "Moose.h"
#include "MooseTypes.h"
#include "Conversion.h"
#include "MooseError.h"

namespace Buck
{
  Real linEst(const Real xLeft, const Real xRight, const Real yLeft, const Real yRight,
    const Real x);
  Real dlinEstdLeft(const Real xLeft, const Real xRight, const Real yLeft, const Real
    yRight, const Real x);
  Real dlinEstdRight(const Real xLeft, const Real xRight, const Real yLeft, const Real
    yRight, const Real x);
  Real logEst(const Real xLeft, const Real xRight, const Real yLeft, const Real yRight,
    const Real x);
  Real dlogEstdLeft(const Real xLeft, const Real xRight, const Real yLeft, const Real
    yRight, const Real x);
  Real dlogEstdRight(const Real xLeft, const Real xRight, const Real yLeft, const Real
    yRight, const Real x);

  ////////////////////////////////////////////////////////////////

  template <typename T>
  inline void iterateAndDisplay(std::string name, std::vector<T> &show, std::string banner
    ="")
  {
    // Iterates through vector of values and dispalys them
    std::cout << "==========================\n";
    if ( banner.size() > 0 )
      std::cout << banner << "\n--------------------------\n";
    for ( unsigned int i=0; i<show.size(); ++i)
      std::cout << name << "[" << i << "]: " << show[i] << std::endl;
    std::cout << "==========================\n" << std::endl;
  }

  ////////////////////////////////////////////////////////////////

  inline int numDigits(int x)
  {
    if ( x == 0 ) return 0;
    if ( x < 0 ) x *= -1;
    if ( x < 10 ) return 1;
    if ( x < 100 ) return 2;
    if ( x < 1000 ) return 3;
    if ( x < 10000 ) return 4;
    if ( x < 100000 ) return 5;
    if ( x < 1000000 ) return 6;

    mooseError("In BuckUtils: number too large for numDigits");
    return -1;
  }

  template <typename T>
  inline T log10(T x)
  {
    if ( x <= 0 )
      mooseError("In BuckUtils: Cannot take log of a negative number or 0.");
    return std::log10(x);
  }
```

```
  inline void getPartition(Real & fk1, Real & fk2, const Real Ng, const Real Nk1, const
    Real Nk2)
  {
    fk1 = Ng / Nk1 * ( Ng - Nk2) / ( Nk1 - Nk2);
    fk2 = Ng / Nk2 * ( Nk1 - Ng) / ( Nk1 - Nk2);
;
  }

  template <typename T1, typename T2>
  inline T1 pow(T1 x, T2 p)
  {
    if (x < 0)
    {
      mooseError("In BuckUtils: Trying to take a power of a negative number with a non-
    integar power");
      return 0;
    }
    return std::pow(x,p);
  }

} // end namspace



#endif //BUCKUTILS_H
```

buck/src/utils/BuckUtils.C

```
#include "BuckUtils.h"

#include "MooseError.h"

namespace Buck {

  Real linEst(const Real xLeft, const Real xRight, const Real yLeft, const Real yRight,
    const Real x)
  {
    Real slope = (yRight - yLeft) / (xRight - xLeft);
    return slope * ( x - xLeft ) + yLeft;;
  }

  Real dlinEstdLeft(const Real xLeft, const Real xRight, const Real yLeft, const Real
    yRight, const Real x)
  {
    return 1 - (x - xLeft) / (xRight - xLeft);
  }

  Real dlinEstdRight(const Real xLeft, const Real xRight, const Real yLeft, const Real
    yRight, const Real x)
  {
    return (x - xLeft) / (xRight - xLeft);
  }

  Real logEst(const Real xLeft, const Real xRight, const Real yLeft, const Real yRight,
    const Real x)
  {
    // if ( yLeft <= 0 || yRight <= 0 )
    //   mooseError("In BuckUtils: One of the input values is less than 0. Cannont compute
     logEst");
    if ( yLeft <= 0 || yRight <= 0 )
      return linEst(xLeft, xRight, yLeft, yRight, x);

    Real power = (x - xLeft) / (xRight - xLeft);
    Real frac = yRight / yLeft;
    frac = std::pow(frac, power);
```

```
    frac *= yLeft;

    return frac;
  }

  Real dlogEstdLeft(const Real xLeft, const Real xRight, const Real yLeft, const Real
    yRight, const Real x)
  {
    if ( yLeft <= 0 || yRight <= 0 )
      return dlinEstdLeft(xLeft, xRight, yLeft, yRight, x);
    Real power = (x - xLeft) / (xRight - xLeft);
    Real frac = yRight / yLeft;
    frac = std::pow(frac, power);
    frac *= (x - xRight) / (xLeft - xRight);

    return frac;
  }

  Real dlogEstdRight(const Real xLeft, const Real xRight, const Real yLeft, const Real
    yRight, const Real x)
  {
    if ( yLeft <= 0 || yRight <= 0 )
      return dlinEstdRight(xLeft, xRight, yLeft, yRight, x);
    Real power = (x - xRight) / (xRight - xLeft);
    Real frac = yRight / yLeft;
    frac = std::pow(frac, power);
    frac *= (x - xLeft) / (xRight - xLeft);

    return frac;
  }
}
```