

Preliminary design and analysis of a sodium-cooled, graphite-moderated TRISO-matrix reactor
(SGTR) using OpenMC modeling

by

Killian Burns

B.S., United States Military Academy, 2013

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Alan Levin Department of Mechanical and Nuclear Engineering
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2025

Approved by:

Major Professor
Jeremy Roberts, PhD

Copyright

© Killian Burns 2025.

Abstract

A pin cell of a sodium-cooled, graphite-moderated, thermal TRISO reactor (SGTR) is simulated and analyzed in OpenMC, an open-source Monte Carlo modeling program. The thermal properties, depletion characteristics, and power production capacity for a single pin are detailed and studied in this report, with extrapolation to a reactor comprised of 700 fuel rods. The assembled reactor core is designed to produce 100 MW_{th}, maintains a positive reactivity through the 60th month of operation, and operates at a burnup of 79.5 $\frac{GW_d}{MTU}$ during this period. The fuel pin is simulated using a homogenous fuel material mix to approximate varying size TRISO beads embedded in a zirconium matrix at a packing fraction of 0.55. The fuel pin measures 400 cm in length with a radius of 3.6 cm, surrounded by a coolant channel of width 1.314 cm and a 20 cm pitch of graphite moderator between hexagonally arranged pins, and thin layers of stainless-steel cladding line the coolant channel. The reactor design requires 100 fuel pins arrayed among each of its 7 hexagonal subassemblies, such that each pin generates 142.9 kW at operational power. The SGTR reactor yields a core power density of 57.32 $\frac{MW}{m^3}$ with an initial heavy metal loading of 2263.8 kg Uranium.

Table of Contents

List of Figures	v
List of Tables	vi
Acknowledgements	vii
Introduction	1
Coolant Flow	4
Temperature Calculations	7
Fuel Design	10
Thermal Analysis	13
Assumptions	20
Depletion Analysis	27
Discussion of Burnup	30
Reactor Assembly and Composition	32
Conclusion	34
References	36
Appendix A - Notebook 1: Homogenous Assumption Validation	38
Appendix B - Notebook 2: Thermal Analysis	47
Appendix C - Notebook 3: Depletion Analysis	70

List of Figures

Figure 1. Cross Sectional view and dimensions of a single channel.	4
Figure 2. Curves for specific heat (constant pressure) and dynamic viscosity of liquid sodium from 350 to 850 °C.	8
Figure 3. TRISO sphere with dimensions used to calculate volume fraction.....	10
Figure 4. Heterogeneous (top) and Homogenous (bottom) model plots for various packing fractions (P.F.)	11
Figure 5. Profile views of the top of a fuel pin and coolant channel.	13
Figure 6. Curve Fitting for the Axial Heating Rate in the Fuel Rod	15
Figure 7. Axial heating profile q'' and heat transfer coefficient H.	17
Figure 8. The temperatures along the channel for fuel centerline, cladding, and coolant.	18
Figure 9. Comparison of effective multiplication factors for homogenous and heterogenous fuel modeling in OpenMC.	21
Figure 10. OpenMC Tallies of Fissions Occurring from Fast and Thermal Neutrons	23
Figure 11. Reactivity of each fuel pin modeled against burnup.	28
Figure 12. Depletion Characteristics over 60 Month of Operation Altering Enrichment of Uranium within the Homogenized Fuel.....	29
Figure 13. Subassembly and Reactor Core configurations.....	32
Figure 14. Neutron Flux Across a Subassembly.....	33

List of Tables

Table 1. An Extension of Tables 3.4 and 3.5 from Todreas and Kazimi comparing benchmark reactor types to the SGTR described by this paper.....	3
Table 2. Dimensions, volume fractions, and thermal conductivity values used to model the fuel homogenously.	25

Acknowledgements

The author would like to first thank his wife Maria for her patience and support over the past five years. Appreciation and acknowledgement are also definitely due to Dr. Jeremy Roberts, for providing a concept to investigate, for introduction and guidance through operation of the OpenMC libraries, and for supervision and recommendations regarding this study. Finally, thanks to Dr. Igor Bolotnov of North Carolina State University and Dr. Hitesh Bindra of Kansas State University for fostering interest and guiding the author in earlier stages of his studies.

Introduction

Sodium cooling in nuclear reactors has primarily been used in the design of Sodium-Cooled Fast Reactors (SFR) or Sodium-Cooled Fast Breeder Reactors (SFBR) which make use of neutrons released in the fast spectrum (generally greater than 1 MeV per fission).^{1, 2} Tri-structural Isotropic (TRISO) fuel has been primarily used as a thermal-spectrum fuel in nuclear reactors. Generally, it is used in thermal pebble-bed reactors, high-temperature gas reactors, and in small modular reactor designs as a safety-conscious encapsulated fuel pellet³. Use of liquid sodium coolant for a TRISO-based reactor has not been well documented, though the concept has been explored for low-enriched fuels.⁴

Liquid sodium is valued as a coolant due to its thermal qualities, neutron cross section, and controllable chemical reactions.^{5, 6} Because sodium boils at a much higher temperature than water⁷, the system does not need to be kept at high-pressure during operation, which generally reduces the risk of material failure. In addition to the high boiling point, the strong thermal conductivity allows for efficient heat transfer and rapid heat removal, further strengthening its

¹Piolo, Igor L., ed. *Handbook of Generation IV Nuclear Reactors*, Woodhead Publishing Series in Energy, no. 103. Cambridge: Woodhead Publishing, 2016, 45.

² Rajan, K. K. "A Study on Sodium - The Fast Breeder Reactor Coolant," *IOP Conference Series: Materials Science and Engineering* 1045, 2021.

³ U.S. Nuclear Regulatory Commission, *Technical Assessment of the Impact of User Requirements in the Design of Advanced High-Temperature Gas-Cooled Reactors*, NUREG/CR-6844, Vol. 1, Washington, DC: U.S. Nuclear Regulatory Commission, 2004.

⁴ Weisner, Edward F., ed., *A Sodium Cooled, Graphite Moderated, Low Enrichment Uranium Reactor for the Production of Useful Power*, September 1954.

⁵ Piolo, *Handbook of Generation IV Nuclear Reactors*, 99-100.

⁶ Rajan, "A Study on Sodium - The Fast Breeder Reactor Coolant," 2021.

⁷ Bowles, Kenneth J., and Louis Rosenblum, *Vapor Pressure of Sodium from 0.5 to 120 Atmospheres*, Lewis Research Center, Cleveland, Ohio: National Aeronautics and Space Administration, May 1965.

qualities as a safety-conscious coolant. Sodium's low scattering cross section for thermal neutrons, relative to water, another commonly used coolant in thermal reactors, makes it an ideal coolant with limited moderation of neutrons produced⁸, allowing a more permissive thermal neutron flux through the core. Another measure that makes sodium a good cooling candidate is the predictability of reacting chemically.⁹ Sodium has no corrosive or chemical relationships with any components of the stainless steel, zirconium, or graphite in use for this reactor,¹⁰ and should have very limited impurities as long as the pool remains covered by an inert gas such as argon. This can reduce downtime for reactor upkeep and increase life of components.

Sodium reactors typically are designed either as a loop system or a pool system.¹¹ This report focuses on the design of a pool type reactor system in which coolant sodium exists at a consistent low pressure, covered by an inert gas to minimize impurities in the reservoir of sodium. In the design of this type of low-pressure supply of coolant, reservoirs of hot and cold sodium are present as an upper and lower plenum, respectively.

Because sodium cooling in a thermal reactor is not widely used, comparisons can be drawn between both thermal reactors and sodium fast reactors. Todreas and Kazimi¹² present parameters for common reactor types, some of which are shown in Table 1, along with representative values for the reactor design studied here. In general, the designed reactor does not closely align with any one of the six benchmark reactors presented. In terms of dimensions,

⁸ Rajan, "A Study on Sodium - The Fast Breeder Reactor Coolant," 2021.

⁹ Pioro, *Handbook of Generation IV Nuclear Reactors*, 100.

¹⁰ International Atomic Energy Agency, *Liquid Metal Coolants for Fast Reactors Cooled by Sodium, Lead and Lead-Bismuth Eutectic*, IAEA Nuclear Energy Series No. NP-T-1.6, IAEA, Vienna (2012), 9.

¹¹ Ibid., 103-106.

¹² Todreas, Neil E., and Mujid S. Kazimi. *Nuclear Systems, 2nd ed., vol. 1*, Orlando: Academic Press, 1990, 82-87.

power density, and heat flux into coolant, the SGTR most closely relates to the BWR, but this design incorporates a much smaller volume and operating power level. When compared with the only other sodium-cooled reactor on these benchmarks, the SGTR has a much lower power level, power density, and heat flux into the coolant than the SFBR.

Table 1. An Extension of Tables 3.4 and 3.5 from Todreas and Kazimi comparing benchmark reactor types to the SGTR described by this paper.

Quantity	PWR	BWR	CANDU	SFBR	HTGR	SGTR - Proposed
Equivalent Core diameter (m)	3.37	4.74	6.16	2.13	8.42	0.75
Core Length (m)	3.66	3.59	5.76	0.91	6.30	4.00
Core Volume (m ³)	32.63	63.33	171.66	3.26	350.80	1.74
Core Power Density (MW/m ³)	104.52	52.47	12.47	239.50	8.55	57.32
Average Linear Heat Generation Rate of a Fuel Rod (kW/m)	17.83	17.60	24.20	19.20	13.40	32.50
Average Heat Flux at the Interface of the Fuel Rod and Coolant (MW/m ²)	0.60	0.50	0.59	1.05	0.20	0.43
Core Power Level (MW)	3411	3323	2140	780	3000	100
% of Power Deposited into Fuel Rods	97.4	96.5	91.6	94.5	100	91
Fuel assemblies/core	193	764	4560	198	3944	7
Assembly Lateral Spacing (mm)	215	152	280	144	361	200
Fuel Rods/Assembly	264	74	37	217	72	100
Fuel Rods Length (mm)	3658	3588	480	914	787	4000
Fuel Rod Diameter (mm)	9.5	11.2	13.1	5.8	21.8	36

Coolant Flow

A characteristic quality of this pin design is the coolant flow. As opposed to some TRISO-based reactors where coolant flows around pebbles of TRISO, and unlike SFR models where sodium flows through channels of space between fuel rods, in the SGTR design, flow of sodium passes through an annular channel around the fuel rod. This relatively tight space helps to reduce the Reynolds number by reducing the hydraulic diameter,¹³ thus ensuring laminar flow through the channel at a fluid velocity of 2.2 m/s. The longer channel of coolant flow with a narrower width helps to increase the heat transfer from the fuel to the sodium. Although heat transfer generally increases in turbulent flow, the narrow channel width of this design ensures the thermal boundary layer (a function of Nusselt number)¹⁴ extends along the width, maximizing thermal transfer to the fluid.

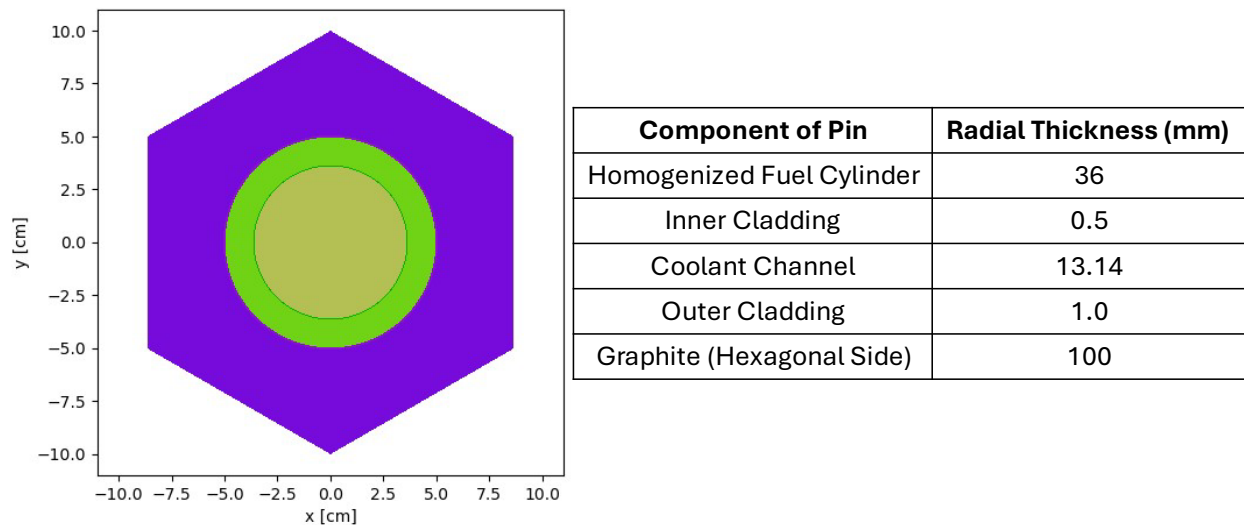


Figure 1. Cross Sectional view and dimensions of a single channel.

¹³ International Atomic Energy Agency, *Sodium Coolant Handbook: Thermal Hydraulic Correlations*, 2003, 5-11.

¹⁴ Bergman, T. L., A. S. Lavine, and F. P. Incropera. *Fundamentals of Heat and Mass Transfer*, 7th ed, 2011, 400

The primary fluid dynamics calculation for this flow is based off the work of Jaeger et al. 2015¹⁵ in which the Nusselt number for a liquid metal is based on the following formula:

$$Nu(Pe) = \Delta^{0.3} \cdot (4.75 + 175 \cdot 10^{-4} \cdot Pe^{0.8}) \quad (1)$$

where Δ is the ratio of the outer to the inner radius of the channel; Pe is the Peclet number:

$$Pe = Re \cdot Pr \quad (2)$$

$$Re = \rho \cdot u \cdot D_h / \mu \quad (3)$$

$$Pr = Cp \cdot \mu / \lambda \quad (4)$$

where ρ is the density, u is the velocity, D_h is the hydraulic diameter, μ is the dynamic viscosity, C_p is the specific heat at constant pressure, and λ is the thermal conductivity. For their study of Nusselt number in metallic flows, Jaeger et al. found that values of Δ equal to 1.36, 1.5, 1.54, and 1.86 for sodium flow produced “excellent” or “very good” fits to this curve, so this constraint was imposed in the design of sodium channel for the SGTR.

The following equation gives the hydraulic diameter¹⁶ of the annular channel:

$$D_h = \frac{4 \cdot \text{Flow Area}}{\text{Wetted Perimeter}} = 4 \cdot \frac{\pi \cdot r_{co}^2 - \pi \cdot r_{ci}^2}{2 \cdot \pi \cdot (r_{ci} + r_{co})} = 2 \cdot \frac{r_{co}^2 - r_{ci}^2}{r_{ci} + r_{co}} \quad (5)$$

where r_{co} is the outer radius of the channel and r_{ci} is the inner radius.

The primary purpose of calculating fluid properties of the fluid is to determine capacity for thermal energy removal from the fuel rod. Deposition of heat into the fluid¹⁷ is:

$$Q = (T_w - T_{coolant}) \cdot H \cdot A_s \quad (6)$$

$$H = \frac{\lambda \cdot Nu}{D_h} \quad (7)$$

¹⁵ Jaeger, W., W. Hering, and M. Lux. "On the Liquid Metal Heat Transfer in Annular Channels: Review, Proposal and Validation of Empirical Models." 2015.

¹⁶ IAEA, *Sodium Coolant Handbook: Thermal Hydraulic Correlations*, 5.

¹⁷ Bergman et al, *Fundamentals of Heat and Mass Transfer*, 115.

where Q is the thermal heat transfer, T_w is the temperature of the cladding in contact with the coolant at r_{ci} , $T_{coolant}$ is the coolant temperature (assumed to be bulk temperature along the channel), A_s is the surface area along which the heat is transferred from cladding to coolant, and H is the heat transfer coefficient for sodium.¹⁸

¹⁸ IAEA, *Sodium Coolant Handbook: Thermal Hydraulic Correlations*, 7.

Temperature Calculations

Beginning with the temperature setting of the lower plenum as 355°C, the change in temperature is calculated for incremental lengths (dz) along the channel. Enthalpy (h) is characteristic of specific temperatures of the sodium, and thus the change in enthalpy gives rise to a change in temperature¹⁹ along the channel according to the following equation:

$$h(z + dz) = \frac{h(z) \cdot \rho(z) + q''(z)/u}{\rho(z)} \quad (8)$$

in which $q''(z)$ is the local axial heating rate, derived from fitting a curve to the tallies of recoverable energy from fissions in OpenMC . The incremental length selected for this study was 1 mm, splitting the length into 4,000 increments.

The properties of the liquid sodium limited the design of the reactor through maximum temperature and laminar flow. At approximately 650°C, the curves for viscosity and specific heat of sodium are no longer able to be estimated using linear interpolation.²⁰ In order to ensure the accurate incorporation of fluid properties throughout the study, this coolant temperature of was set as an upper threshold for this design. Fluid velocity must also be set to a reasonable value that does not cause the Peclet number to exceed 270, which is the onset of turbulent behavior for metallic fluids with low Prandtl numbers flowing through an annulus.²¹ In this study, a value of 2.2 was proven to provide a reasonable range of adjustability for thermal properties and ensured a limitation of 650°C for the fluid would not be exceeded. Figure 2 demonstrates how tabular values for certain properties of liquid sodium were interpolated linearly. The blue lines indicate

¹⁹ Bergman et al, *Fundamentals of Heat and Mass Transfer*, 525.

²⁰ Todreas and Kazimi, *Nuclear Systems*, 950.

²¹ IAEA, *Sodium Coolant Handbook: Thermal Hydraulic Correlations*

the first region of interpolation (350 – 525°C) with an extended curve to 600°C to demonstrate divergence from this linear relationship beyond that value. The red lines indicate the linear relationship from 525-650°C defining the second region of interpolation. Since this loses accuracy above 650°C, this value of outlet temperature was imposed as an upper threshold for fluid temperature.

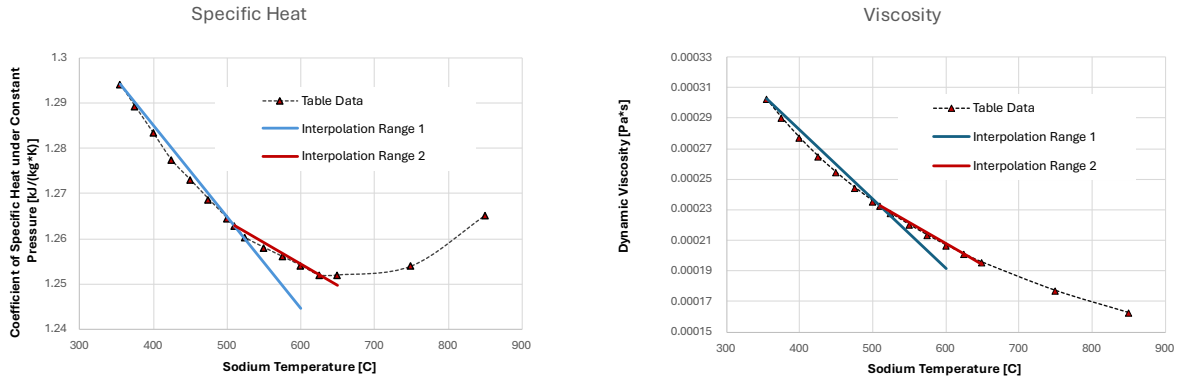


Figure 2. Curves for specific heat (constant pressure) and dynamic viscosity of liquid sodium from 350 to 850 °C.

While $T_{coolant}$ was derived through stepwise analysis as described above, each other thermal and fluid property would then arise from that one. T_w was determined by:

$$T_w = T_{coolant} + q''/u \quad (9)$$

Centerline temperature for the fuel²² was calculated assuming constant values of thermal conductivity k for the cladding and fuel, equal to that for stainless steel at 800 K, $14.4 \frac{W}{m \cdot K}$, with a volume-weighted thermal conductivity for the fuel. The composite thermal conductivity was computed to be 0.44 times that of Zirconium, at $9.5 \frac{W}{m \cdot K}$. This value appears below as k_c in the equation for centerline temperature:

²² Bergman et al, *Fundamentals of Heat and Mass Transfer*, 112-119.

$$T_{CL} = T_w + q'' \cdot \left(\frac{t_{clad}}{k_{ss}} + \frac{r_f}{8 \cdot \pi \cdot k_c} \right) \quad (10)$$

where t_{clad} is the thickness of interior cladding and r_f is the radius of the fuel element.

Fuel Design

The fuel composition itself is based off a concept described by Brown et al²³ of TRISO beads embedded in a metallic matrix. For this design, the matrix consists of pure zirconium. Though different manufacturing specifications for TRISO particles exist, the chosen parameters mirror the composition most commonly found in German manufactured fuel.^{24, 25} The general thickness of each layer is known to vary, but the figure below shows the average values.



Layer of TRISO	Atom Fraction composition	Thickness (μm)	Volume Fraction of Homogenous Mixture
Kernel (UO ₂ 40% enrichment)	66.67% Oxygen 15% U-235 18.33% U-238	212.5	2.65%
Buffer	100% Carbon	100	16.76%
Inner Pyrolytic Carbon	100% Graphite Carbon	35	11.66%
Silicon Carbide	50% Silicon 50% Carbon	35	15.58%
Outer Pyrolytic Carbon	100% Graphite Carbon	40	23.34%
Zirconium Matrix	100% Zirconium	N/A	30.00%

Figure 3. TRISO sphere with dimensions used to calculate volume fraction.

Packing fraction for TRISO particles embedded in a matrix has been investigated by Brown et al. from 0.3 to 0.74. A value of 0.55 was selected for this study, which is reasonably physically possible with single-size beads of TRISO. However, use of various particle sizes to

²³ Brown, Nicholas R., Richard Hernandez, and Andrew T. Nelson, "High Volume Packing Fraction TRISO-Based Fuel in Light Water Reactors," 2022.

²⁴ U.S. NRC, *Technical Assessment of the Impact of User Requirements in the Design of Advanced High-Temperature Gas-Cooled Reactors*, Section 2.

²⁵ Golesorkhi, S., S. Kelly, and A. Trottier, "Impact of the TRISO Fuel Homogenisation Approximation in the MHTGR," 2023.

minimize volume of the filler matrix, as seen in the Fort St. Vrain reactor,²⁶ or an injection-method compression technique as described in the NRC's *Technical Assessment* (2004)²⁷ would also assist with increasing the packing fraction.

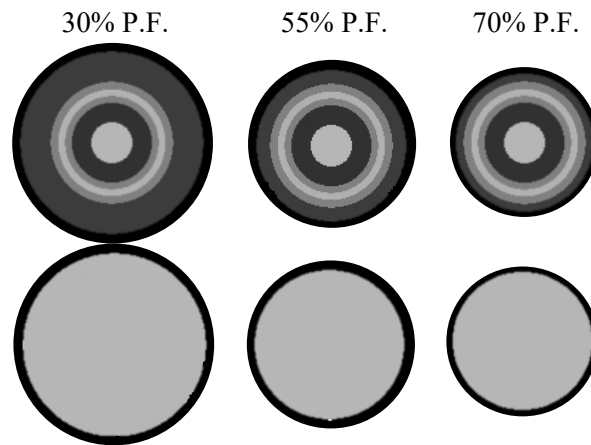


Figure 4. Heterogeneous (top) and Homogenous (bottom) model plots for various packing fractions (P.F.)

Enrichment of fuel within the TRISO kernel also is known to vary, with values from 5 to 85 percent being documented in sources published.^{28, 29, 30} For this study, a 45 percent enrichment of Uranium is selected and sufficiently produces fission energy to meet thermal design criteria. This is a lower enrichment than seen in MOX fuel in fast reactors, but higher than most manufactured TRISO or UO₂ fuel pins. Generally, this is a higher enrichment than is

²⁶ Brown et al, "High Volume Packing Fraction TRISO-Based Fuel in Light Water Reactors," 2.

²⁷ US NRC, *Technical Assessment of the Impact of User Requirements in the Design of Advanced High-Temperature Gas-Cooled Reactors*, 2-37

²⁸ Brown et al, "High Volume Packing Fraction TRISO-Based Fuel in Light Water Reactors."

²⁹ Srinivasan, G., K. V. Suresh Kumar, B. Rajendran, and P. V. Ramalingam, "The Fast Breeder Test Reactor—Design and Operating Experiences," 2005

³⁰ IAEA, *Sodium Coolant Handbook: Thermal Hydraulic Correlations*

commercially available due to several factors discussed in the Assumptions section of this study but is investigated here as a potential high-enrichment application.

Thermal Analysis

Channel analysis focuses on the capacity of the coolant to remove heat from the steel-clad fuel. The fuel is simulated as a homogenous material³¹ that models TRISO enriched to 45% embedded into a zirconium matrix at a packing fraction of 0.55. The heating of liquid sodium propels coolant up through an annular channel around the fuel rod, bordered externally by a hexagonal pitch of graphite moderator that provides the surrounding environment for this fuel pin. The simulated model includes a plenum on either end of the cylindrical channel, simulating the hot and cold liquid sodium reservoirs at the top and bottom, respectively, of the fuel cell. A thin cap of stainless steel covers each end of the fuel pin and shields it from the sodium plenum.

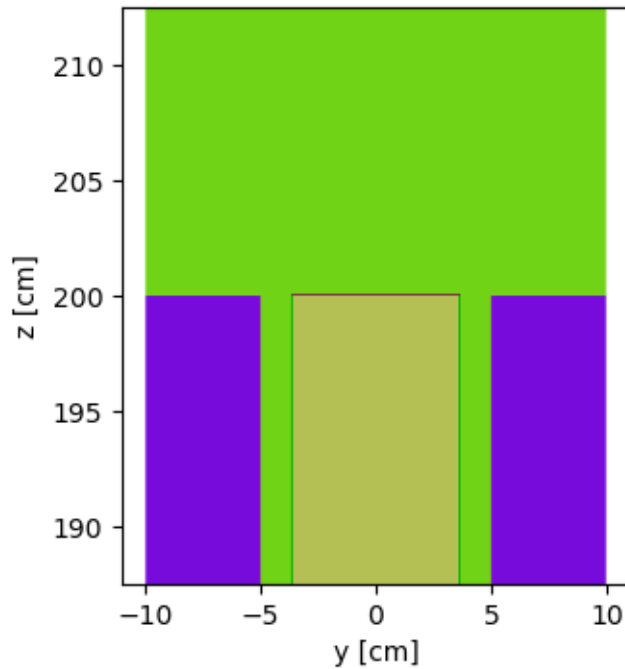


Figure 5. Profile views of the top of a fuel pin and coolant channel.

³¹ Golesorkhi et al, "Impact of the TRISO Fuel Homogenisation Approximation in the MHTGR," 2023.

Optimized values for thickness and width dimensions within the pin were iteratively evaluated in OpenMC for effective multiplication factor. The ultimate dimensions of the channel and pin are given in Figure 1.

Thermal analysis required extrapolation of the design beyond the fuel pin to instead incorporate the entire reactor. The design takes inspiration from comparison with the reactor described in Srinivasan et al 2005.³² The subassemblies of this Indian reactor arrayed fuel channels in concentric hexagonal alignment over 5 rings for 61 channels each, with 65 subassemblies. This resulted in 3965 total fuel pins over the design of the reactor. For the design of this SGTR, a 100 MW design constraint required the application of 7 subassemblies, each with 100 fuel channels in its 6-ring hexagonal design, leaving 27 channels for mechanical or reactivity control purposes. This drives the energy output requirement for each pin to 142,857 W.

Modeling the pin in OpenMC, the design was analyzed at 20 evenly spaced heights for energy deposition to the fuel (heating), fissions, and energy per fission, each calculated and weighted by number of source particles (s.p.). Because the OpenMC solver is designed to be set to various particle strengths and precisions, the outputs of these data are given irrespective of time; therefore, these outputs must be scaled to relatable parameters according to power settings of the pin.³³ Although specific results for heating were not generated at this stage, a clear profile of an axial heating rate emerged, as seen in Figure 6. Using the data for heat deposition into the fuel, two curve fits were attempted to describe the profile of axial heating. The raw data is the value of axial heating returned for the indicated length of the pin, the total nuclear heating is line that connects these points, and the dashed lines show the curve fits that were attempted. The

³² Srinivisan et al, "The Fast Breeder Test Reactor—Design and Operating Experiences," 2005.

³³ OpenMC, "8.Specifying Tallies."

cosine curve (left) did not properly account for the tapered heating at the ends of the pin. The gaussian curve (right) more precisely fitted the data recorded.

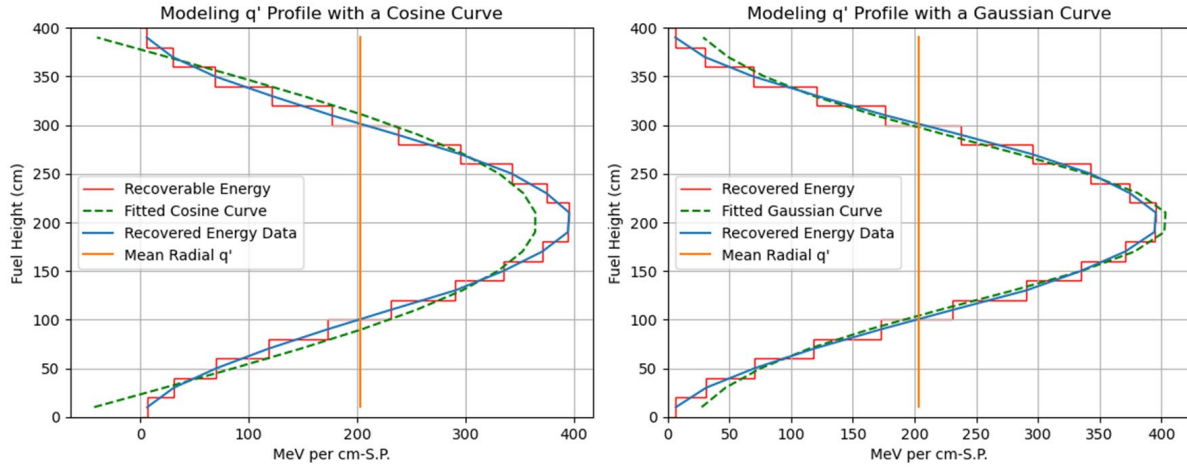


Figure 6. Curve Fitting for the Axial Heating Rate in the Fuel Rod

Initially a cosine curve was tried as fit the curve of axial heating based on the approximation provided by Todreas and Kazimi³⁴ for heating along a fuel pin. However, this curve seemed to poorly characterize the heating toward the ends of the fuel pin, which tapered off asymptotically (R^2 : 0.966). Upon noting the end behavior of the curve, a bell-shaped Gaussian curve was tried next, producing a tighter curve fit approximation (R^2 : 0.993). The parameters for describing this curve were used to produce the fit curve:

$$q''(z) = 406.056 \cdot M \cdot e^{\frac{\left(\frac{200.811 - z}{82.135}\right)^2}{-2}} \quad (11)$$

In this equation, z is channel height in centimeters and M is some constant “adjustor” with units of $\left[\frac{J}{eV} \cdot \frac{s.p.}{cm \cdot s} = \frac{W \cdot s.p.}{eV \cdot cm}\right]$ to fit the tally curve for radial recovered energy $\left[\frac{eV}{cm \cdot s.p.}\right]$ to the thermodynamic profile of axial heating $\left[\frac{W}{m^2}\right]$ desired by the pin power settings..

³⁴ Todreas and Kazimi, *Nuclear Systems*, 92.

Obtaining the ratio of maximum to average heating was also critical to the determination of true quantitative values for the axial heating profile. Because OpenMC does not output standard units for tallies, instead translating into units like “MeV per fission per source particle per centimeter” after some adjustments for dimensions, ratios of values help to provide a general shape of a relationship that then requires interpretation. Radial heating average rate was found arithmetically as the average of all 20 array values to be $203.39 \left[\frac{eV}{cm*s.p} \right]$, and the peak value of radial heating was calculated from the amplitude of the curve to be $395.66 \left[\frac{eV}{cm*s.p} \right]$. Therefore, the ratio of peak to average worked out to 1.9453. Later, as explained in equation 14, this value then helped ensure maximum volumetric heating did not exceed allowable values.

After these values defining the axial heating rate were obtained from OpenMC functionality, further calculations were conducted in Excel. As described above in the temperature calculations, the axial heating profile was fundamental to set the properties of heat absorption into the coolant and therefore thermal power generation. Defining the axial heating q'' according to equation 11, fit to the height in 1-millimeter increments, yielded the heating profile graph shown in the left of Figure 7. These curves were generated in Microsoft Excel though plotting the calculated values at each incremental height. The left graph shows the axial heating profile q'' and the right graph shows the heat transfer coefficient H .

The value of the adjustor M was determined through trial-and-error to be 753.16. This value of M yields an output coolant temperature of 644.5°C at a fluid velocity of 2.2 meters per second, deliberately below the 650°C upper threshold for coolant temperature for reasons

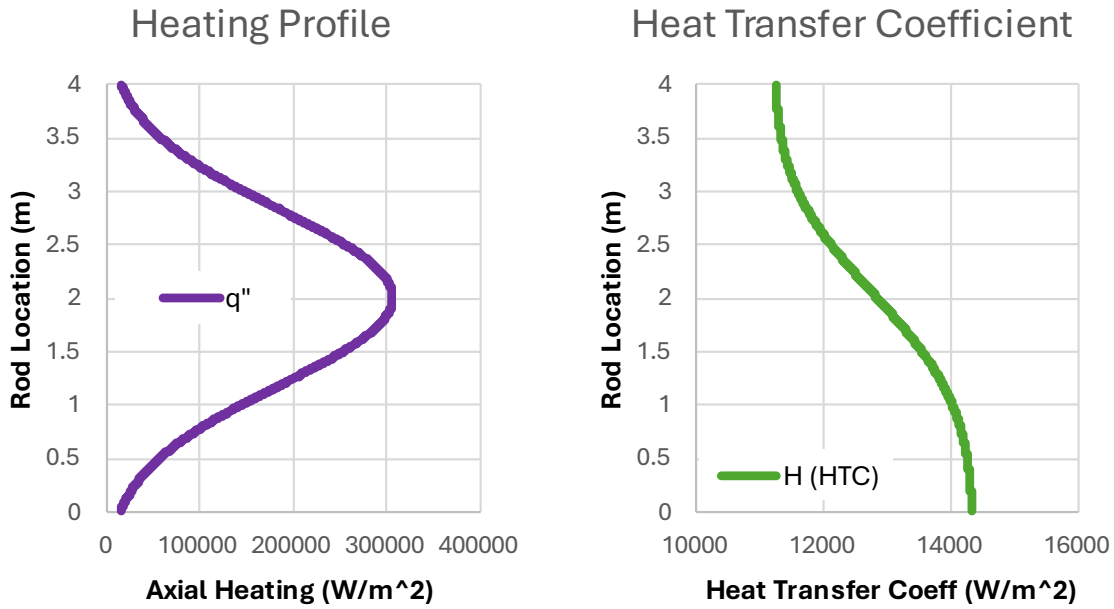


Figure 7. Axial heating profile q'' and heat transfer coefficient H .

described above. The curve for the temperatures of the coolant, cladding, and fuel centerline can be seen in the graph of Figure 8. Max wall temperature is 645.9 °C and max centerline temperature is 653.1 °C.

Since the axial heat transfer rate had been calculated at each height along the rod, upon setting a value for fluid velocity, the entire system of heat transfer was solved incrementally using equation 9. As defined above in equation 6, the total heat transfer along the pin was taken to be the difference between the wall and coolant temperature, multiplied by the average heat transfer coefficient for the coolant, multiplied by the area over which the heat transfers, the surface area of the cladding. This total heat transfer rate, Q , was calculated as 142.857 kW.

With the overall reactor design requirement to produce 100 MW_{th}, and the thermal output of each pin known, the number of rods required by the reactor remained a simple division

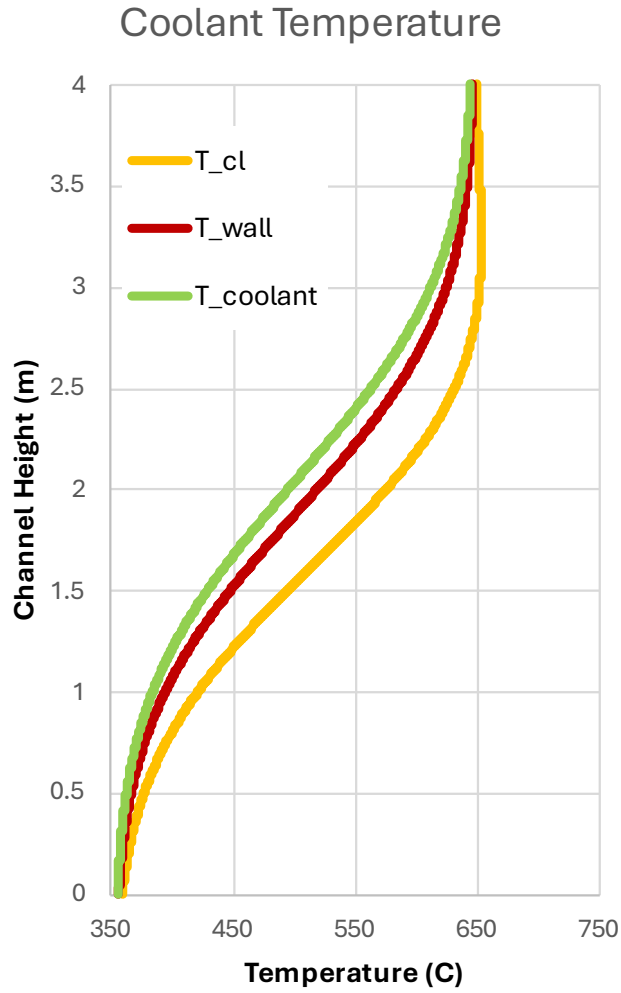


Figure 8. The temperatures along the channel for fuel centerline, cladding, and coolant.

problem to solve. 697 rods at 142.857 kW produced slightly above 100 MW_{th} for the reactor.

Considering hexagonal assemblies as a selected geometric arrangement for the reactor, the round number of 700 rods was selected as a design point, with 7 subassemblies of 100 fuel pins each.

As a measure of safety, the maximum volumetric heating rate was also calculated, to ensure that peak temperatures in the fuel never exceed the melting point for zirconium. Taking the average heat transfer rate q'' and dividing by the length of the rod yields the approximate volumetric heating rate from the fuel to the cladding. This volumetric heating rate can be

compared to the maximum allowable heating rate based off assuming maximum heat in the cladding (conservatively using the boiling point of sodium³⁵ at 1 atm, 1154°C) and setting our maximum fuel centerline temperature to the melting point of zirconium, 1855°C. Beginning with the equation for heat transfer in a cylinder³⁶:

$$T_f(r) = \frac{\dot{q} \cdot r_0^2}{4 \cdot k} \left(1 - \frac{r^2}{r_0^2}\right) + T_s \quad (12)$$

$$T_f(0) = \frac{\dot{q} \cdot r_f^2}{4 \cdot k} (1 - 0) + T_s < T_{max} = 1855^\circ\text{C}$$

$$\frac{\dot{q} \cdot (0.0360)^2}{4 \cdot k_{Zr}} < (1855 - T_s)$$

$$\dot{q} < (1855 - T_s) \cdot \frac{(4 \cdot k_c)}{(0.0360)^2}$$

$$\dot{q} < (1855 - 1154) \cdot \frac{(4 \cdot 9.5)}{(0.0360)^2}$$

$$\dot{q} < (701) \cdot \frac{(86.4)}{0.001296}$$

$$\dot{q} < 20.6 \frac{\text{MW}}{\text{m}^3} \quad (13)$$

Calculating the actual volumetric heating rate as the ratio of peak to average radial heating times the average heating rate and divided by the length of the rod:

$$\dot{q} = \frac{q''_{peak}}{q''_{avg}} \cdot \frac{\frac{Q}{A_s}}{L} = (1.9453) \cdot \frac{155.03 \frac{\text{kW}}{\text{m}^2}}{4 \text{ m}} = 36.9 \frac{\text{kW}}{\text{m}^3} \quad (14)$$

yields an actual maximum volumetric heat flux of $36.9 \frac{\text{kW}}{\text{m}^3}$. This indicates that there is a factor of safety more than 500 with regards to the fuel centerline temperature.

³⁵ Bowles and Rosenblum, *Vapor Pressure of Sodium from 0.5 to 120 Atmospheres*, 1965.

³⁶ Bergman et al, *Fundamentals of Heat and Mass Transfer*, Table C.3.

Assumptions

A key simplification made during this study, in the interest of decreasing computational time and power, was to homogenize the fuel. This was done by calculating the volumes of each layer of TRISO as shown in Figure 5, then combining the volume weight of each of those layers with Zirconium, volume weighted according to the packing fraction. This method has been previously summarized by Golesorkhi et al³⁷ and it is generally known as the Volume-Weighted Homogenization (VWH) method. Assessing the native calculation of multiplication factors allowed comparison of such a simplified fuel sample to a single simulated TRISO particle. The simulated particle was surrounded by a spherical mass of zirconium so that both this particle and a sphere of the homogenous material had the same atomic composition. The VWH method was modified for this study by adding a spherical shell of graphite to the exterior of the internal TRISO models in order to approximate the self-shielding and self-resonance qualities of TRISO³⁸ in both the homogenous and heterogenous approaches. Since the design of this fuel pin also relies heavily on the moderation of graphite in the core, this also more closely approximated the nature of the fuel as it will behave in the cell. The two models were evaluated under the same settings for multiplication factor (with a reflective exterior boundary condition) and are compared together in Figure 9. Note that the maximum packing fraction of 0.55 indicated as realistic according to the work of Brown et al.³⁹ and the maximum commercially available enrichment for HALEU TRISO (19.75%)⁴⁰ were investigated for this study but ultimately did

³⁷ Golesorkhi et al, "Impact of the TRISO Fuel Homogenisation Approximation in the MHTGR," 2023.

³⁸ Ibid.

³⁹ Brown et al, "High Volume Packing Fraction TRISO-Based Fuel in Light Water Reactors," 5

⁴⁰ Carlson, John, "HALEU: Some Safeguards and Non-Proliferation Implications," August 8, 2024.

not return a viable reactivity in the full pin design. For initial design considerations, a “viable” reactivity was considered to be an initial reactivity above 0.16, coincident with a multiplication factor of about 1.2. This required an increase of enrichment to 45%, as further discussed in the Depletion Calculations section below. To ensure feasibility of the design over 60 months of operation, a minimum value of 0.1 reactivity was set, with an acknowledgement that (as shown in the graphs of Figure 9) the homogenous fuel will over-predict the effective multiplication factor of a real design.

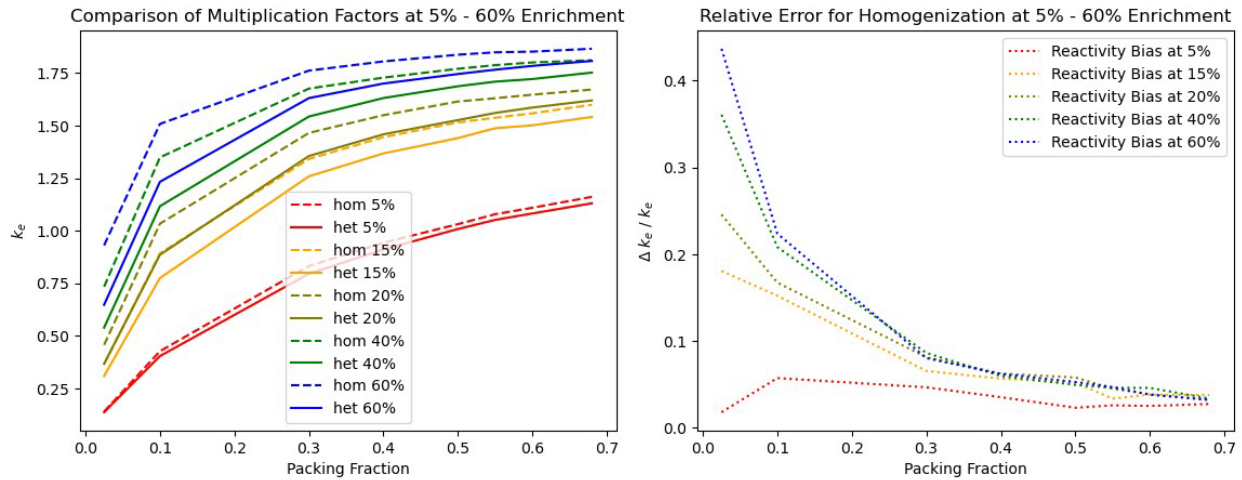


Figure 9. Comparison of effective multiplication factors for homogenous and heterogenous fuel modeling in OpenMC.

An iterative model generator used an input kernel diameter to determine volumes of each layer of the TRISO bead and the encapsulating zirconium layer. The outputs of this function, a heterogenous model and a homogenous model (as well as the required volume fractions for mixing materials homogenously), each prompted the running of the program yielding effective multiplication factors.

Due to the lack of immediate contact with graphite, this method of comparison tends to over-estimate k-effective. However, the alternative of not having any graphite shell vastly underestimates k-effective. This is due to the suppression of resonance self-shielding effects⁴¹ that is inherent to the structure of TRISO particles. Across enrichments from 5% to 60%, as shown in Figure 9, the difference measured as a percentage variation from the heterogeneous value decreases significantly to below 10% above a packing fraction of 0.3. While not an ideal loss of precision, this method of homogenizing fuel serves to evaluate trends in thermal analysis, reduce computational resources (memory), and reduce computational time.

Another key assumption made for this fuel pin was that all energy from fissions came from neutrons in the thermal spectrum. This allows use of known cross sections to determine energy output and fission rates for a given power setting. This assumption can be visually validated by comparing the OpenMC “heating” and the “fission-q-recoverable” tallies, summarized in the figure below. Fewer than 1.1% of all fissions occur due to fast neutrons. This is mathematically explained by the low fission cross section of Uranium-235 in the fast spectrum. Around 98.9% of all fissions occur due to interactions with neutrons that have been slowed to thermal velocities by scattering from other atoms in the pin. This validates the assumption that fissions occur from thermal neutrons. In the figure below, the curves generated show a relationship between the proportion of thermal- versus fast-neutron initiated fissions. Due to OpenMC calculating tallies per source particle, these values are not directly able to be translated into common units without adjusting for simulation parameters, but they are valuable in assessing proportional relationships between values within the simulation.

⁴¹ *ibid.*

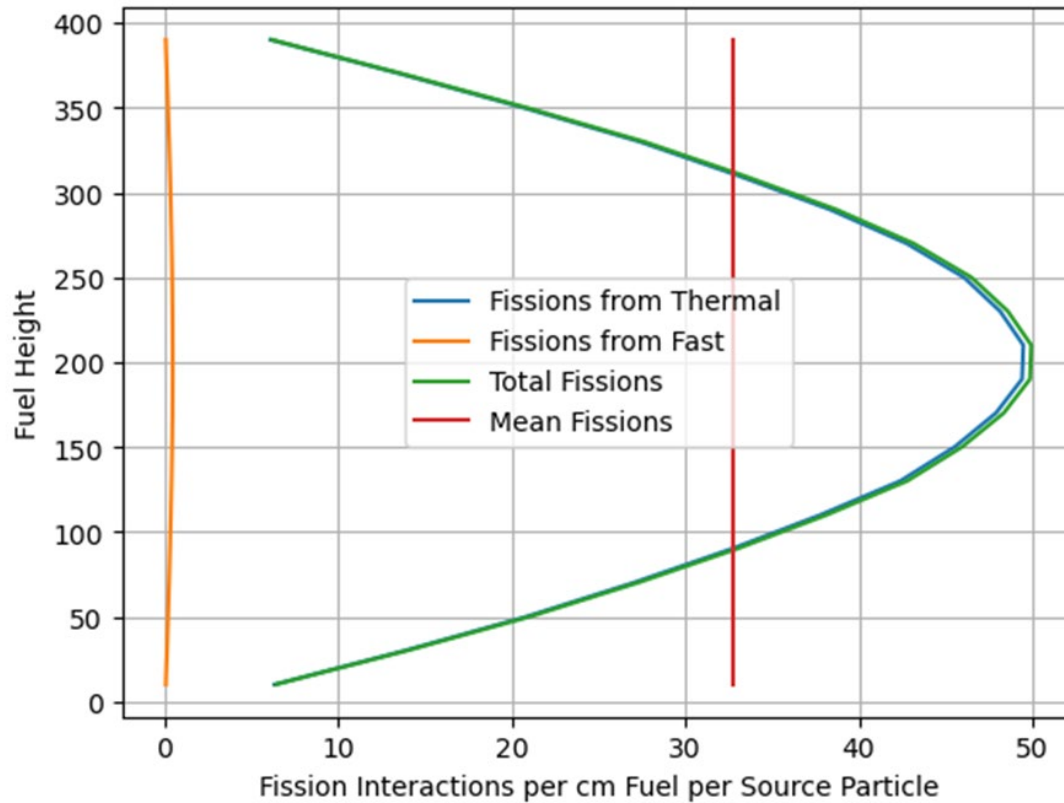


Figure 10. OpenMC Tallies of Fissions Occurring from Fast and Thermal Neutrons

During the thermal analysis of the channel, various assumptions and generalizations were accepted. First, all simulations and calculations are computed with an ideal thermal contact, neglecting gaps between materials and enabling temperature calculations based on thermal resistances of materials. This assumption naturally leads to over-estimating the amount of thermal energy able to be removed from the fuel, potentially under-estimating the fuel centerline temperature. However, the additional internal temperature would not sufficiently endanger the fuel, which is limited by the melting point of the zirconium matrix, approximately 1855°C. As shown in calculations above, culminating in equations 12 and 13, this design constraint does not pose risk to safety.

The second thermal assumption that was made was that thermal-dependent properties of sodium (density, viscosity, surface tension, specific heat, and enthalpy) could be approximated with a single value of bulk fuel temperature. Because the width of the channel is relatively small (8.82 mm), the temperature variation across that width would be minimal. This potentially leads to an over-estimation of thermal absorption into the fuel, and thus again an under-estimation of fuel centerline temperature.

A third set of thermal assumptions relates to the properties of the metals used in the design. All values of physical constants derive from information provided in *Fundamentals of Heat and Mass Transfer*.⁴² In the range of wall temperatures calculated, stainless steel (simulated to be Alloy 304) would have a thermal conductivity variation from $14.4 \frac{W}{m K}$ to $23 \frac{W}{m K}$ in this range. As a measure of safety, this lowest value of thermal conductivity is taken as constant, which generally would cause an under-estimation of heat transfer out of the fuel, contrary to the previous two assumptions. Similarly, Zirconium would have a thermal conductivity ranging from 20 to $22 \frac{W}{m K}$. Since the locations and proximity of each individual TRISO particle cannot be precisely known, a volume-weighted approach was incorporated to approximate a lumped thermal conductivity for the fuel cylinder. Using the thermal conductivities shown in Table 2 and the following equations for calculating thermal resistance⁴³, a composite resistance was calculated.

$$R_{t,cond} = \frac{1}{4 \pi k} \left(\frac{1}{r_1} - \frac{1}{r_2} \right), q' = \frac{\Delta T}{R_{t,cond}} \quad (15)$$

$$q'_{zirconium} = \frac{\Delta T}{R_{Zr}} = \Delta T * 40,500; R_{Zr} = \frac{1}{40,500}$$

⁴² Bergman et al, *Fundamentals of Heat and Mass Transfer*, Appendix A.

⁴³ Ibid, Table 3.3.

$$q'_{TRISO} = \frac{\Delta T}{R_{TRISO}} = \frac{\Delta T \cdot 4 \pi \cdot 1,000,000}{470.588 + 941.177 + 56.544 + 0.538 + 43.424} = \Delta T \cdot 8,310 ; R_{TRISO} = \frac{1}{8,310}$$

$$q'_{Composite} = \frac{\Delta T}{R_{Zr}} (0.45) + \frac{\Delta T}{R_{TRISO}} (0.55) = \Delta T \cdot 22,800 ; R_{Composite} = \frac{1}{22,800}$$

$$k_{Composite} \approx 0.44 \cdot k_{Zr} = 9.504 \frac{W}{m \cdot K} \quad (16)$$

While it would have been more conservative to take an even lower table value, it would further remove the calculated values from the true expected behavior of heat transfer in a real system. This overall low thermal conductivity constant for the fuel predicts a lower radial heat flux and thus provides a relatively high calculation of maximum temperature within the fuel at the centerline. This is itself a conservative estimation because in a heterogenous fuel cell, the heat profile would peak not precisely along the centerline of the fuel, but individually in each of the kernels. This is due to the design of the layers to stop and absorb kinetic energy from expelled fission products, localizing the heating to the kernel or buffer of the TRISO.

Table 2. Dimensions, volume fractions, and thermal conductivity values used to model the fuel homogenously.

Layer of TRISO	Atom Fraction Composition	Thickness (μm)	Volume Fraction of Homogenous Mixture	Thermal Conductivity at 800 K (W/(m·K))
UO2 Kernel (45% enrichment)	66.67% Oxygen	212.5	2.65%	10
	15% U-235			
	18.33% U-238			
Buffer	100% Carbon	100	16.76%	1.6
Inner Pyritic Carbon	100% Graphite Carbon	35	11.66%	490
Silicon Carbide	50% Silicon	35	15.58%	5.7
	50% Carbon			
Outer Pyritic Carbon	100% Graphite Carbon	40	23.34%	490
Zirconium Matrix	100% Zirconium	N/A	30.00%	22.7

The final major simplifying assumption made for thermal analysis was calculation of overall heat transfer coefficient for the rod using average values for wall temperature, coolant temperature, and heating rates. Since the cladding heats faster than the coolant, the average value of the difference will be an underestimate for the first section of the channel and an overestimate for the latter section of the channel. The shape of the curve for the rate of heat transfer follows the bell-shaped axial heating profile of Figure 7, so using that average value likely would not affect the overall heat transfer coefficient. When considered together, there is a minute numerical preference for the earlier, higher difference in temperature, which occurs prior to the heat flux profile exceeding the average rate. This will tend to slightly underestimate the heat transfer coefficient and therefore heat transfer into the coolant.

Depletion Analysis

In OpenMC, the *deplete* functionality (through the *openmc.deplete.CoupledOperator*⁴⁴ and *openmc.deplete.PredictorIntegrator*⁴⁵ classes) was used to analyze the pin based on the settings determined for the channel. The Coupled Operator class takes a model and a string that describes a path to a data library as its inputs, and it generates an operator to be used in the Predictor Integrator. The Predictor Integrator class takes this operator, an array of time steps (which in this study was assigned the units of days, with 60 steps of 30), and a power of the reactor in Watts. For this study, the power remained constant at all time steps, but it could also be iterated with this function if desired.

The integrator then sequenced through each of the 61 states (initial plus 18 steps) to generate 60 months of effective multiplication factor data. The results also had several different sets of data to pull, including masses of each isotope present at each step, which was then analyzed for burnup assessments. With reactivity from the effective multiplication factor in addition to the consumption of fissile isotopes, quantities of interest arise for assessing the design of the reactor. For the pin modeled, Figure 11 depicts the change in reactivity over the 60 months of operation. This design yields a low drop in reactivity, with a reactivity in excess of 0.10 for the entire duration.

Reactor mass of a given isotope is calculated by multiplying the pulled mass data by the number of rods in the design. Using these simple calculations demonstrates that reactor U-235 mass decreases from an initial level of 1011.83 kg and is depleted after 60 months to 776.14 kg.

⁴⁴ OpenMC. "openmc.deplete.CoupledOperator."

⁴⁵ OpenMC. "openmc.deplete.PredictorIntegrator."

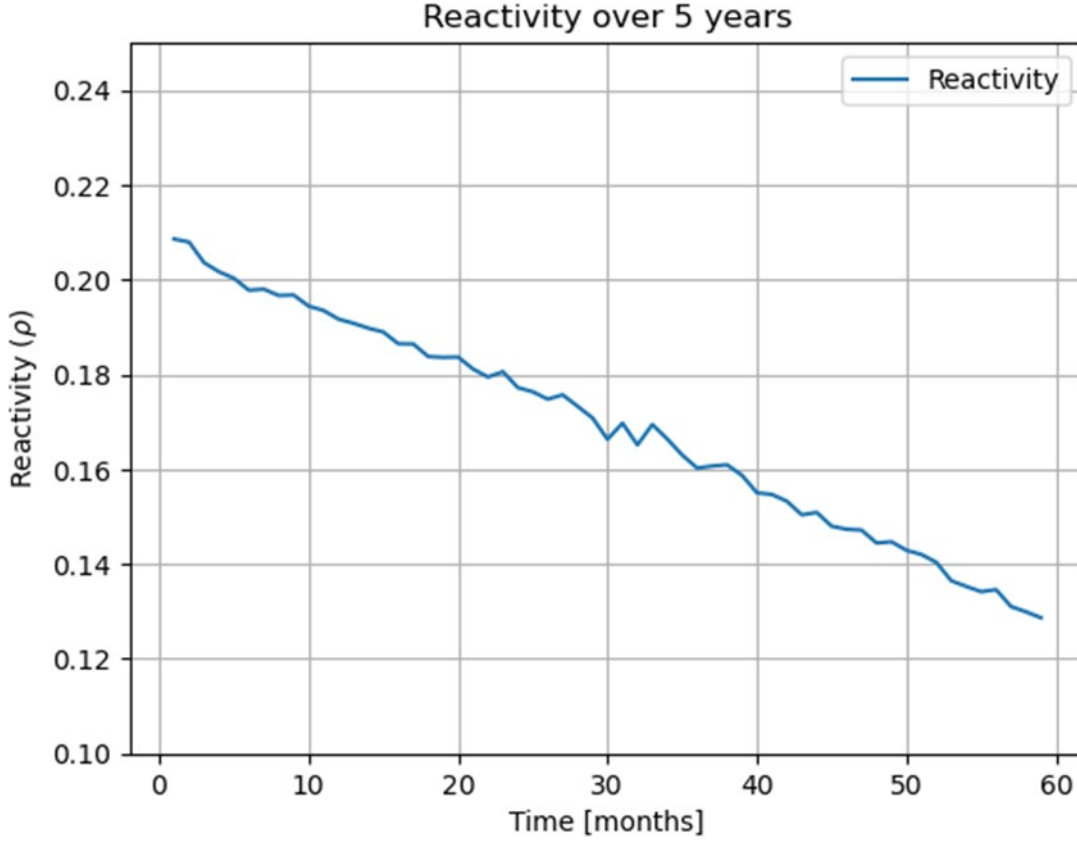


Figure 11. Reactivity of each fuel pin modeled against burnup.

Burnup is calculated^{46, 47} purely as:

$$B \left[\frac{GWd}{MTU} \right] = B \left[\frac{MWd}{kg U} \right] = \frac{P[W] \cdot t[d]}{{}^{235}U_{initial}[g] + {}^{238}U_{initial}[g]} \cdot \left[\frac{g}{1000 kg} \right] \quad (15)$$

Over the course of those 60 months, the reactor produces 180 GWd with an initial Uranium loading of 2264.34 kg, for a burnup of 79.493 GWd/MTU. During this period, multiplication factor decreases from 1.280 to 1.144 and reactivity decreases from 0.219 to 0.126.

⁴⁶ U.S. NRC. "High Burnup Spent Nuclear Fuel." 2024.

⁴⁷ Roberts, Jeremy A., Max Langston, Daniel Nichols, Eric Schlaikjer, Graham Schlaikjer, and Hitesh Bindra. "A Nonlinear Reactivity Method with Application to Accident-Tolerant Fuels." 2016.

As noted earlier in the discussion of this design, the enrichment value selected for this design is not commercially manufactured into TRISO. However, this is the lowest value of UO₂ enrichment (assessed using the dimensions described in this study) that maintains a reactivity above 0.1 for the entire 60-month life cycle. As shown in Figure 12, 25% enrichment yields a positive reactivity for 19 months, while 30%, 35%, and 40% enrichments remain above 0.1 reactivity for 5 months, 28 months, and 51 months, respectively. 20% enrichment, the maximum enrichment commercially available, does not produce a positive reactivity with this pin design. Because of the lack of account for leakage in this design, since the overall pin length is long relative to a low core volume, a reactivity threshold of 0.1 was favored for this analysis. This requirement drove the design selection for 45% enrichment.

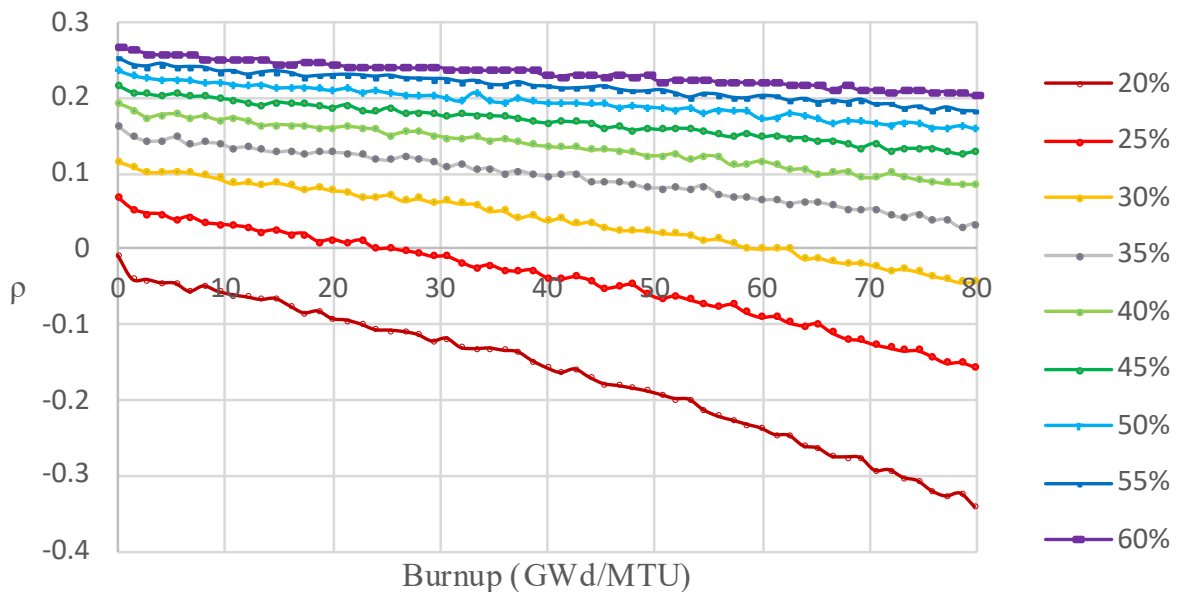


Figure 12. Depletion Characteristics over 60 Month of Operation Altering Enrichment of Uranium within the Homogenized Fuel

Discussion of Burnup

Although the rates of burnup estimated by this reactor are high, there are two considerations of practicability that must be addressed that significantly impact this output. The first consideration is the enrichment of the fuel. While higher percentages of enrichment are common for mixed-oxide fuels, the common enrichments of TRISO are more commonly seen in the range of 5^{48, 49} to 15⁵⁰% commercially, with any enrichment of uranium-based fuels uncommon above 20%.⁵¹ This is largely due to the security implications⁵² regarding higher-enrichment fuels, but also due to the higher cost associated with manufacturing these enriched fuels. Because of the high-enriched Uranium, typically only seen in military applications, this design would potentially be limited to use as a modular reactor for a military unit or on a protected military base. For example, Fort Riley, KS has an electrical power usage of approximately 40 MW_e for on-base housing,⁵³ with the remainder of this design's 100 MW_{th} easily capable of powering the non-residential buildings on post.

The second consideration is the packing factor. The packing fraction is generally limited by the method of fuel element fabrication, with a fraction of 30-40% when using the Admix process⁵⁴ or 50-60% when using the injection process. Mathematically, this value is limited

⁴⁸ Brown et al, "High Volume Packing Fraction TRISO-Based Fuel in Light Water Reactors," 2022.

⁴⁹ Golesorkhi et al, "Impact of the TRISO Fuel Homogenisation Approximation in the MHTGR," 2023

⁵⁰ Pioro, *Handbook of Generation IV Nuclear Reactors*, Table 3.2, Table 3.5.

⁵¹ U.S. Nuclear Regulatory Commission. *TRISO-Coated Particle Fuel Phenomenon Identification and Ranking Tables (PIRTs) for Fission Product Transport Due to Manufacturing, Operations, and Accidents. Main Report*. NUREG/CR-6844, Vol. 1. Washington, DC, 2004, Table 2-4 and Table 2-14.

⁵² Carlson, John, "HALEU: Some Safeguards and Non-Proliferation Implications," 2024.

⁵³ Sullivan, Dan, "U.S. Army's Fort Riley Solar Installation Reaches 16 MW, Powers 40% of Base Housing," *Microgrid Knowledge*, May 17, 2023.

⁵⁴ Ibid, 2-35.

(when using equal size spherical beads) to 68% but values above 70% are feasible when using multiple-sized beads.⁵⁵ Using a set value of 0.55 for creation of the homogenous mix in this study is well within the capability of these methods.

Because of this high packing fraction and high enrichment, this reactor achieves remarkably high burnup rates, with relatively low temperatures. However, this may not invalidate the conceptual design considering that homogenization of the fuel detracts from the self-shielding resonance within each bead and overall decreases the effective multiplication factor of the fuel.⁵⁶ A heterogenous model of this TRISO fuel embedded in a zirconium environment would have an equivalent reactivity with the homogenized fuel model used in this study at a much lower packing fraction and enrichment. Using enrichment percentage of 19.75%, the highest enrichment published⁵⁷ for high-assay low enriched Uranium (HALEU) fuels, would be a much more viable option for a commercially available design. However, the 45% enriched reactor could still be a viable option with added security and oversight of manufacturing facilities producing the TRISO particles.

⁵⁵ Brown et al, "High Volume Packing Fraction TRISO-Based Fuel in Light Water Reactors," 2022.

⁵⁶ Golesorkhi et al

⁵⁷ Carlson, John, "HALEU: Some Safeguards and Non-Proliferation Implications," 2024.

Reactor Assembly and Composition

The overall design and power settings for each channel require a total of 697 fuel pins to produce the reactor design requirement of 100 MW_{th}. Since the pins are designed to be arranged hexagonally in the subassemblies, the subassemblies likewise can be arranged into a reactor assembly in a single hexagonal ring. The arrangement of pins is not within the full scope of this report, but to demonstrate the behavior and arrangement of the reactor, 100 fuel channels, 6 control rod channels, and 21 reflective nickel/mechanical channels have been arranged into a subassembly in Figure 13. A single subassembly (left) is shown, arranging 100 fuel pins, 21 reflective channels, and 6 control rod channels into a 6-ring hexagonal lattice. A full reactor assembly (right) is shown with 7 such subassemblies

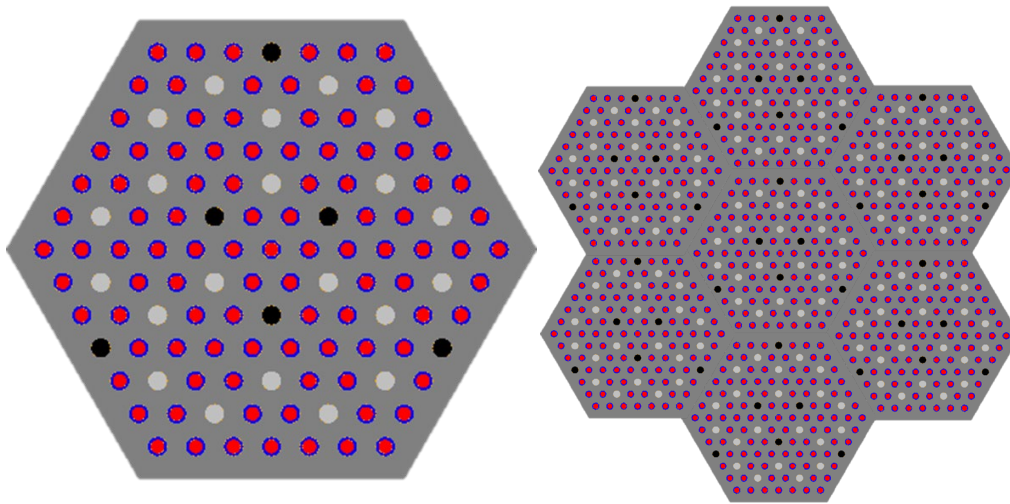


Figure 13. Subassembly and Reactor Core configurations.

Arranging the geometry of a single subassembly into OpenMC and then obtaining neutron flux for each fuel channel location, the relative effects of aggregating into a subassembly can be observed. While a numerical value is provided for flux, this number is largely influenced

by the settings on the code that is run, so the analysis to this point provides good qualitative information about the shape of the flux only.

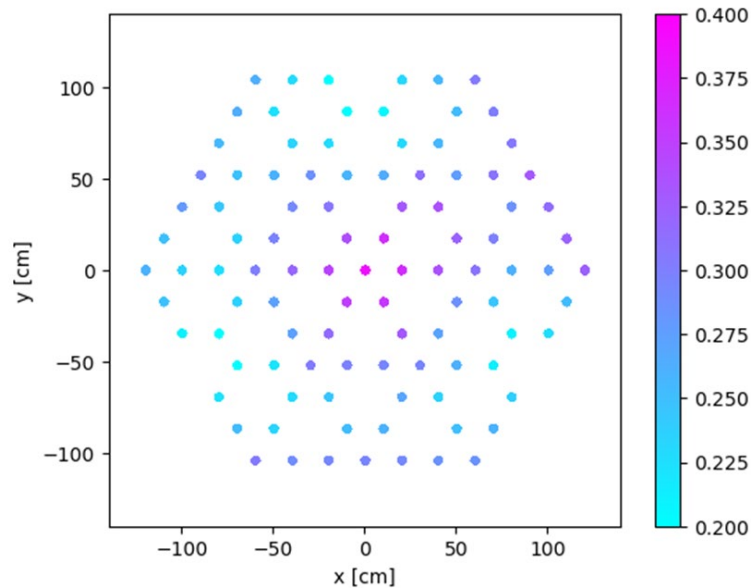


Figure 14. Neutron Flux Across a Subassembly

As shown in the figure above, the neutron flux is generally distributed across the reactor with a flux concentration towards the center. This arrangement of pins does not necessarily reflect an ideal layout, but it is provided with this study as a demonstration of how the OpenMC program can be used to investigate neutron flux. Through a balance of different control mechanisms, including control rods, burnable poisons, sensor channels, or neutron-reflective rods, the concavity of the flux distribution can be adjusted, potentially decreasing need for rotation of pins within the subassembly.

Conclusion

This design concept validates a novel design of fuel mixture with OpenMC calculations. TRISO-embedded zirconium has not been incorporated as a potential fuel source previously, according to reviews of available literature. It compares most closely to a BWR in terms of core power density but has a much higher designed average linear heat generation rate than any of the benchmark designs to which it was compared in Table 1. Largely due to its use of sodium coolant, the amount of heat able to be absorbed and utilized would likely be close to that of the HTGR and SFR designs, which see lower component wear from this chemically controllable non-corrosive fuel, lowering replacement costs and decreasing downtime for a plant.

Given the small volume of the overall reactor, with a core that fits inside an 8- by 7- by 4-meter space, the design output of 100 MW, and the incorporation of the heavily safety-conscious fuel delivery method of TRISO particles, this reactor would be an ideal candidate for deployment as a component of a small modular reactor (SMR). The sodium reservoir required for cooling could be contained in equivalent volumes above and below the core. A secondary loop of coolant, either sodium or a similar capacity fluid, could then run on a closed loop between a heat exchanger and a turbine generator system. All these components could be areas of future development for this reactor concept.

Further design testing for the fuel channel would be best aimed at increasing accuracy of the data provided, and limiting the reliance on certain simplifying assumptions. Incorporating a gap between fuel and cladding and adjusting the thermal analysis accordingly would be one such increase of accuracy. Simulation of a helical wire wrapping of the fuel and adjusting calculations for the flow of sodium helically around the pin would introduce another realistic feature that has not yet been included. A third option looking at the singular channel would be to replace the

homogenous fuel with a maximum-packed zirconium matrix filled with varying sizes of TRISO spheres (a function supported by OpenMC currently) or utilizing Reactivity-equivalent Physical Transformation⁵⁸ to adjust the fuel simulation with an external graphite layer around the homogenous core.

Future development of simulations towards the reactor design would be testing different arrangements of the 27 non-fuel channels within the subassembly to achieve the most balanced neutron flux in the core. Also valuable would be the design of reflector and control rod channels to appropriately influence the reactivity of the core. Finally, incorporating additional subassemblies into the core using batching would feasibly increase the burnup yield.

⁵⁸ Golesorkhi et al. 2023

References

- Bergman, T. L., A. S. Lavine, and F. P. Incropera. *Fundamentals of Heat and Mass Transfer*, 7th ed. Hoboken, NJ: John Wiley & Sons, 2011.
- Bowles, Kenneth J., and Louis Rosenblum. *Vapor Pressure of Sodium from 0.5 to 120 Atmospheres*. Lewis Research Center, Cleveland, Ohio: National Aeronautics and Space Administration, May 1965.
- Brown, Nicholas R., Richard Hernandez, and Andrew T. Nelson. "High Volume Packing Fraction TRISO-Based Fuel in Light Water Reactors." *Progress in Nuclear Energy* 146 (2022): 104151.
- Carlson, John. "HALEU: Some Safeguards and Non-Proliferation Implications." August 8, 2024. <https://vcdnp.org/wp-content/uploads/2024/09/HALEU-Some-Safeguards-Implications.pdf>.
- Golesorkhi, S., S. Kelly, and A. Trottier. "Impact of the TRISO Fuel Homogenisation Approximation in the MHTGR." *Annals of Nuclear Energy* 183 (2023).
- International Atomic Energy Agency. *Sodium Coolant Handbook: Thermal Hydraulic Correlations*. Vienna: IAEA, 2024. Printed by the IAEA in Austria, February 2024.
- International Atomic Energy Agency. *Liquid Metal Coolants for Fast Reactors Cooled by Sodium, Lead and Lead-Bismuth Eutectic*. IAEA Nuclear Energy Series No. NP-T-1.6. Vienna: IAEA, 2012.
- Jaeger, W., W. Hering, and M. Lux. "On the Liquid Metal Heat Transfer in Annular Channels: Review, Proposal and Validation of Empirical Models." In *Proceedings of NURETH-16*, Chicago, IL, August 30-September 4, 2015, 2073-2086.
- Pirotto, Igor L., ed. *Handbook of Generation IV Nuclear Reactors*. Woodhead Publishing Series in Energy, no. 103. Cambridge: Woodhead Publishing, 2016.
- Rajan, K. K. "A Study on Sodium - The Fast Breeder Reactor Coolant." *IOP Conference Series: Materials Science and Engineering* 1045 (2021): 012013.
- Roberts, Jeremy A., Max Langston, Daniel Nichols, Eric Schlaikjer, Graham Schlaikjer, and Hitesh Bindra. "A Nonlinear Reactivity Method with Application to Accident-Tolerant Fuels." *Annals of Nuclear Energy* 94 (2016): 581–588.
- Srinivasan, G., K. V. Suresh Kumar, B. Rajendran, and P. V. Ramalingam. "The Fast Breeder Test Reactor—Design and Operating Experiences." *Nuclear Engineering and Design*, 236 (2006): 796–811.

Sullivan, Dan. "U.S. Army's Fort Riley Solar Installation Reaches 16 MW, Powers 40% of Base Housing." *Microgrid Knowledge*, May 17, 2023.

<https://www.microgridknowledge.com/government-military/news/55270633/us-armys-fort-riley-solar-installation-reaches-16-mw-powers-40-of-base-housing>.

Todreas, Neil E., and Mujid S. Kazimi. *Nuclear Systems, 2nd ed., vol. 1*, Orlando: Academic Press, 1990.

U.S. Nuclear Regulatory Commission. "High Burnup Spent Nuclear Fuel." Updated July 2024.

<https://www.nrc.gov/reading-rm/doc-collections/fact-sheets/bg-high-burnup-spent-fuel.html>

U.S. Nuclear Regulatory Commission. *Technical Assessment of the Impact of User Requirements in the Design of Advanced High-Temperature Gas-Cooled Reactors*. NUREG/CR-6844, Vol. 1. Washington, DC: U.S. Nuclear Regulatory Commission, 2004.

<https://www.nrc.gov/reading-rm/doc-collections/nuregs/contract/cr6844/v1/index.html>

U.S. Nuclear Regulatory Commission. *TRISO-Coated Particle Fuel Phenomenon Identification and Ranking Tables (PIRTs) for Fission Product Transport Due to Manufacturing, Operations, and Accidents. Main Report*. NUREG/CR-6844, Vol. 1. Washington, DC: U.S. Nuclear Regulatory Commission, July 2004.

<https://www.nrc.gov/reading-rm/doc-collections/nuregs/contract/cr6844/v1/index.html#pub-info>

Weisner, Edward F., ed. *A Sodium Cooled, Graphite Moderated, Low Enrichment Uranium Reactor for the Production of Useful Power*. Oak Ridge, TN: Technical Information Service Extension, Atomic Energy Commission, September 15, 1954.

OpenMC. "8. Specifying Tallies." Accessed January 18, 2025.

<https://docs.openmc.org/en/stable/usersguide/tallies.html?highlight=heating>.

OpenMC. "openmc.deplete.CoupledOperator." Accessed February 8, 2025.

<https://docs.openmc.org/en/stable/pythonapi/generated/openmc.deplete.CoupledOperator.html?highlight=coupled+operator#openmc.deplete.CoupledOperator>.

OpenMC. "openmc.deplete.PredictorIntegrator." Accessed February 8, 2025.

<https://docs.openmc.org/en/stable/pythonapi/generated/openmc.deplete.PredictorIntegrator.html>.

Appendix A

Appendix B

Appendix C

Appendix A - Notebook 1: Homogenous Assumption Validation

Modeling TRISO as Homogenous Fuel inside a graphite exterior

Using Volume Weighted Homogenization (VWH) proved insufficient mathematically for approximating TRISO in the fuel, resulting in approximately 30% inaccuracy (under-estimation) with regard to k-effective. Proposed and tested in this notebook is a modified test based off Golesorkhi et al's Reactivity-equivalent Physical Transformation (RPT) in which they simulate TRISO sphere as a pin of homogenous medium encased in a shell of graphite

Load the programs that will be used, set enrichment values and packing fractions, and establish TRISO dimensions.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
import uncertainties
from uncertainties import unumpy as unp
import openmc
import openmc.mgxs as mgxs
import openmc.data
import math
```

```
In [2]: enrichment_percent = [5, 15, 20, 40, 60]
pfs = [0.025, 0.10, 0.3, 0.4, 0.5, 0.55, 0.6, 0.68]
```

```
In [3]: kd = 2125/100000
buffer_th = 100/10000
IPyC_th = 35/10000
SiC_th = 35/10000
OPyC_th = 40/10000
TRISO_r = (kd + buffer_th*2 + IPyC_th*2 + SiC_th*2 + OPyC_th*2) / 2

openmc.config['cross_sections'] = './opt/data/endf/cross_sections.xml'
```

Establish the settings for particle simulation.

```
In [4]: lower_left = [-TRISO_r, -TRISO_r, -TRISO_r]
upper_right = [+TRISO_r, +TRISO_r, +TRISO_r]
uniform_dist = openmc.stats.Box(lower_left, upper_right)

point = openmc.stats.Point((0, 0, 0))
source = openmc.IndependentSource(space=point)

settings_file = openmc.Settings()
settings_file.batches = 100
settings_file.inactive = 20
settings_file.particles = 300
settings_file.output = {'tallies': False}
settings_file.source = source
#settings_file.source = openmc.source.IndependentSource(space=uniform_dist, constraints={'fissionab
```

Establish the constant dimensional values that will be used in iteration

```
In [5]: fuel_v = (kd/2)**3 * 4/3 * math.pi
buffer_v = (kd/2 + buffer_th)**3 * 4/3 * math.pi - fuel_v
IPyC_v = (kd/2 + buffer_th + IPyC_th)**3 * 4/3 * math.pi - buffer_v - fuel_v
SiC_v = (kd/2 + buffer_th + IPyC_th + SiC_th)**3 * 4/3 * math.pi - IPyC_v - buffer_v - fuel_v
OPyC_v = (kd/2 + buffer_th + IPyC_th + SiC_th + OPyC_th)**3 * 4/3 * math.pi - SiC_v - IPyC_v - buffer_v
```

```

kernelOD = openmc.Sphere(r=kd/2)
bufferOD = openmc.Sphere(r=kd/2+buffer_th)
IPyCOD = openmc.Sphere(r=kd/2+buffer_th+IPyC_th)
SiCOD = openmc.Sphere(r=kd/2+buffer_th+IPyC_th+SiC_th)
OPyCOD = openmc.Sphere(r=kd/2+buffer_th+IPyC_th+SiC_th+OPyC_th)

```

Define an iterable function that takes enrichment percent and packing fraction as arguments, outputting a homogenous model, a heterogenous model, and an array of volume fractions for use in other notebooks.

```

In [6]: def model_generator(ep,pf):
# Generate Materials
kernel = openmc.Material()
kernel.add_nuclide('U235', ep/100)
kernel.add_nuclide('U238', (1-ep/100))
kernel.add_element('O', 2.0)
kernel.set_density('g/cm3',10.81)

buffer = openmc.Material()
buffer.add_element('carbon', 1.0)
buffer.set_density('g/cm3',1.02)

PyC = openmc.Material()
PyC.add_element('carbon', 1.0)
PyC.set_density('g/cm3',1.92)

SiC = openmc.Material()
SiC.add_element('carbon', 1.0)
SiC.add_element('Si', 1.0)
SiC.set_density('g/cm3',3.20)

SiC_het = openmc.Material()
SiC_het.add_element('carbon', 1.0)
SiC_het.add_element('Si', 1.0)
SiC_het.set_density('g/cm3',3.20)

PyC_het = openmc.Material()
PyC_het.add_element('carbon', 1.0)
PyC_het.set_density('g/cm3',1.92)
PyC_het.add_s_alpha_beta('c_Graphite')

buffer_het = openmc.Material()
buffer_het.add_element('carbon', 1.0)
buffer_het.set_density('g/cm3',1.02)
buffer_het.add_s_alpha_beta('c_Graphite')

zirconium = openmc.Material()
zirconium.add_element('Zr', 1.0)
zirconium.set_density('g/cm3', 6.6)

graphite = openmc.Material()
graphite.add_element('carbon', 1.0)
graphite.set_density('g/cm3', 2.26)
graphite.add_s_alpha_beta('c_Graphite')

# Iterate Variable Geometry
shell_th = TRISO_r * (pf**(-1/3) - 1)
zirc_v = (shell_th+TRISO_r)**3 * 4/3 *math.pi - OPyC_v - SiC_v - IPyC_v - buffer_v - fuel_v
mod_th = TRISO_r * 2
graph_v = (shell_th+TRISO_r+mod_th)**3 * 4/3 *math.pi - zirc_v - OPyC_v - SiC_v - IPyC_v - buffer_v
volumes = np.array([fuel_v, buffer_v, IPyC_v, SiC_v, OPyC_v, zirc_v])
vol_frac = volumes/sum(volumes)

```

```

shell_surface = openmc.Sphere(r=shell_th+TRISO_r)
outer_surface = openmc.Sphere(r=shell_th+TRISO_r+mod_th, boundary_type='reflective')

# Mix Materials with Volume Fraction
sphere = openmc.Material.mix_materials([kernel, buffer, PyC, SiC, PyC, zirconium],
                                       vol_frac, percent_type="vo")
graphite_volumes = IPyC_v + OPyC_v + buffer_v
sphere.add_s_alpha_beta('c_Graphite', graphite_volumes/(graphite_volumes+SiC_v))

materials_file = openmc.Materials([kernel, buffer, PyC, SiC, buffer_het,
                                   PyC_het, SiC_het, zirconium, graphite, sphere])

# Generate Geometry
kernel_cell = openmc.Cell(name='1.6% Kernel')
kernel_cell.region = -kernelOD
kernel_cell.fill = kernel
kernel_cell.volume = fuel_v

buffer_cell = openmc.Cell(name='1.6% Buffer')
buffer_cell.region = +kernelOD & -bufferOD
buffer_cell.volume = buffer_v
buffer_cell.fill = buffer_het

IPyC_cell = openmc.Cell(name='1.6% IPyC')
IPyC_cell.region = +bufferOD & -IPyCOD
IPyC_cell.volume = IPyC_v
IPyC_cell.fill = PyC_het

SiC_cell = openmc.Cell(name='1.6% SiC')
SiC_cell.region = +IPyCOD & -SiCOD
SiC_cell.volume = SiC_v
SiC_cell.fill = SiC_het

OPyC_cell = openmc.Cell(name='1.6% OPyC')
OPyC_cell.region = +SiCOD & -OPyCOD
OPyC_cell.volume = OPyC_v
OPyC_cell.fill = PyC_het

zirc_cell = openmc.Cell(name='1.6% Zirconium Matrix')
zirc_cell.region = +OPyCOD & -shell_surface
zirc_cell.volume = zirc_v
zirc_cell.fill = zirconium

mod_cell = openmc.Cell(name='1.6% Graphite Moderator')
mod_cell.region = +shell_surface & -outer_surface
mod_cell.fill = graphite

shell_cell = openmc.Cell(name='1.6% Graphite Shell')
shell_cell.region = +shell_surface & -outer_surface
shell_cell.fill = graphite

sphere_cell = openmc.Cell(region=-shell_surface)
sphere_cell.fill = sphere

# Populate Geometry
het_universe = openmc.Universe()
het_universe.add_cell(kernel_cell)
het_universe.add_cell(buffer_cell)
het_universe.add_cell(IPyC_cell)
het_universe.add_cell(SiC_cell)
het_universe.add_cell(OPyC_cell)
het_universe.add_cell(zirc_cell)

```

```

het_universe.add_cell(mod_cell)
hom_universe = openmc.Universe()
hom_universe.add_cell(sphere_cell)
hom_universe.add_cell(shell_cell)
het_geometry_file = openmc.Geometry(het_universe)
hom_geometry_file = openmc.Geometry(hom_universe)

het_model = openmc.Model(het_geometry_file, materials_file, settings_file)
hom_model = openmc.Model(hom_geometry_file, materials_file, settings_file)

return het_model, hom_model, vol_frac

```

```

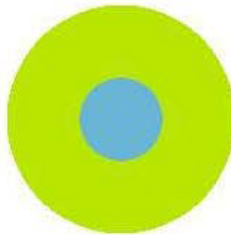
In [7]: het_models = []
hom_models = []
vol_fracs = []
het_keffs = []
hom_keffs = []

for i in range(len(enrichment_percent)):
    for j in range(len(pfs)):
        a, b, c = model_generator(enrichment_percent[i], pfs[j])
        het_models.append(a)
        hom_models.append(b)
        vol_fracs.append(c)
        sp_temp = a.run()
        sp_het = openmc.StatePoint(sp_temp)
        het_keffs.append(sp_het.keff)
        sp_het.close()
        sp_temp = b.run()
        sp_hom = openmc.StatePoint(sp_temp)
        hom_keffs.append(sp_hom.keff)
        sp_hom.close()

```

Output Omitted for Appendix.


```
In [8]: plot = openmc.Plot()
plot.basis = 'xz'
plot.origin = (0, 0, 0)
plot.width = (0.5, 0.5)
plot.pixels = (400, 400)
plot.color_by = 'material'
plots = openmc.Plots([plot])
plots.export_to_xml()
openmc.plot_inline(plots)
```



```
In [9]: hom_keffs
```

```
Out[9]: [0.1388115478853501+/-0.0009455092481933355,  
0.42638176620399126+/-0.0026103878286800725,  
0.8319818005651748+/-0.004257778442178895,  
0.9426430793071351+/-0.004816239612239662,  
1.030876333654557+/-0.004776924975029274,  
1.0782273367103272+/-0.006191483115474371,  
1.1089102348059336+/-0.005599268237071519,  
1.1610186178323225+/-0.0062071028243414475,  
0.36422175543944496+/-0.002238004601375116,  
0.89231750375437+/-0.0052044065172110665,  
1.34090899132943+/-0.005153386003039839,  
1.4447335839795468+/-0.005666271280095305,  
1.5150958198810829+/-0.005304340403936335,  
1.5372332688375838+/-0.005811402554208229,  
1.5583781625136357+/-0.005412889484047415,  
1.5991931209488026+/-0.005945154491143365,  
0.4563892269515312+/-0.0027871720126574624,  
1.033264139717548+/-0.005686199353685125,  
1.4652274429189642+/-0.005567171363364146,  
1.549393497990689+/-0.005123673263500514,  
1.6145616764291504+/-0.005802212827069426,  
1.6301710232807671+/-0.005275154171839877,  
1.6470536159789226+/-0.006006107711482945,  
1.6715778986175307+/-0.003990103668055421,  
0.7332146020889961+/-0.0038383229379777913,  
1.3487404871753954+/-0.005765594819325692,  
1.6760459469583604+/-0.00543854706794075,  
1.728132757940156+/-0.004470273600400193,  
1.770362903038412+/-0.0047196411282237885,  
1.7875241370164936+/-0.004514812816296182,  
1.8004093659559097+/-0.004329125423931642,  
1.811053279427323+/-0.004477644618553964,  
0.930075799062541+/-0.004855414964756255,  
1.5080011221961478+/-0.005196832791726102,  
1.7615498018444815+/-0.0048426995424704295,  
1.8052874081062158+/-0.004235421902417462,  
1.8372690554219806+/-0.004078710756620521,  
1.8486828147059526+/-0.0036177208920703825,  
1.8518933714654688+/-0.00382041831456868,  
1.8656536839513167+/-0.003658396177720681]
```

```
In [10]: het_keffs
```



```
Out[10]: [0.13637534080032476+/-0.0016233546955406258,
0.4033304462752188+/-0.0030192511162787606,
0.794979434906554+/-0.004939287792744733,
0.9106144599209495+/-0.005552850557993499,
1.0077110315941933+/-0.005862014546038336,
1.0511229921449652+/-0.005308457714678442,
1.0817950729073509+/-0.004714080681113001,
1.1301441790626567+/-0.005501320610596827,
0.30847606290250895+/-0.002934404873010782,
0.7745431687739351+/-0.00588948097155334,
1.2586037260452085+/-0.005894660945249375,
1.3674183124492196+/-0.006316924904222986,
1.4402743650989716+/-0.006472055643803439,
1.4874330558007378+/-0.005978828382275499,
1.501073054857012+/-0.005087071025481561,
1.5411562323873955+/-0.0053097862634539595,
0.3663759079633048+/-0.003581236915900971,
0.8852623572903038+/-0.005772343142672888,
1.3552149074717466+/-0.005923459514716144,
1.4584736453394582+/-0.005726123088265696,
1.5261748889940843+/-0.006472789982433038,
1.5600583479846242+/-0.005640687449337056,
1.5858428405046696+/-0.005845462801675466,
1.619634477725034+/-0.006075985876796004,
0.5388229675338316+/-0.005722503101148754,
1.1163717986841237+/-0.0056547301211352175,
1.5431488164742977+/-0.006507005186501284,
1.6309578634842623+/-0.005465335549713385,
1.68704570856938+/-0.005378370875837704,
1.7090911531250532+/-0.005296062762155639,
1.7214817659260537+/-0.005362780430887832,
1.7521247064784464+/-0.004775080218134867,
0.6473307990523547+/-0.00490239270936767,
1.2324120955168463+/-0.0062287851865519,
1.630768307840968+/-0.0061233874200340805,
1.7000471524525755+/-0.005669419852270603,
1.745317896720155+/-0.005329198595658015,
1.7659394663544763+/-0.004957857576964145,
1.7840668257453423+/-0.00416356197859138,
1.8076527854255635+/-0.004392833440852059]
```

Visualization and Discussion

Because of the testing at several different enrichments, this relationship can now be viewed graphically by comparing k-eff at each enrichment percent as packing fraction increases.

```
In [12]: hom_plot = unp.nominal_values(hom_keffs)
het_plot = unp.nominal_values(het_keffs)

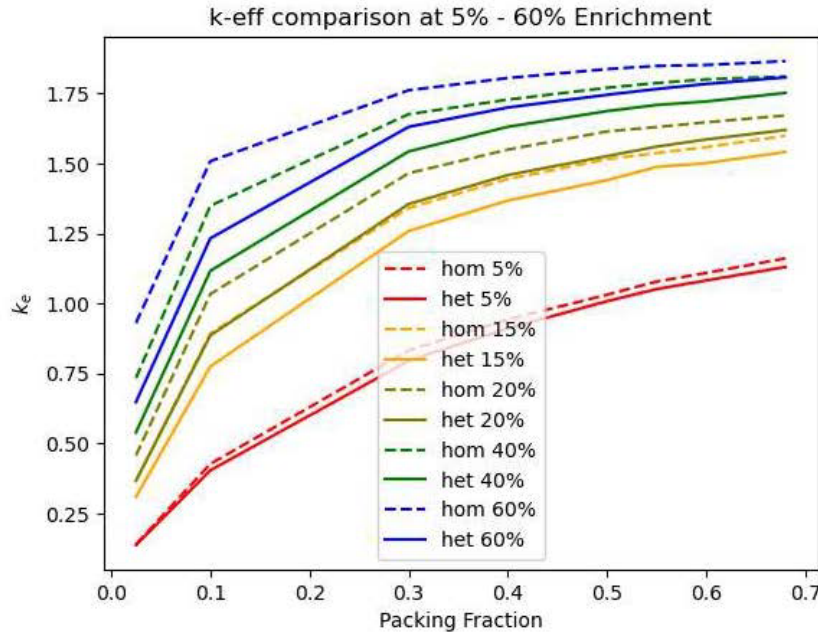
# pfs = [0.025, 0.10, 0.3, 0.4, 0.5, 0.55, 0.6, 0.68]

plt.figure()
plt.title('k-eff comparison at 5% - 60% Enrichment')
plt.plot(pfs, hom_plot[0:8], label = 'hom 5%', color='red', linestyle='--')
plt.plot(pfs, het_plot[0:8], label = 'het 5%', color='red')
plt.plot(pfs, hom_plot[8:16], label = 'hom 15%', color='orange', linestyle='--')
plt.plot(pfs, het_plot[8:16], label = 'het 15%', color='orange')
plt.plot(pfs, hom_plot[16:24], label = 'hom 20%', color='olive', linestyle='--')
plt.plot(pfs, het_plot[16:24], label = 'het 20%', color='olive')
plt.plot(pfs, hom_plot[24:32], label = 'hom 40%', color='green', linestyle='--')
plt.plot(pfs, het_plot[24:32], label = 'het 40%', color='green')
```

```

plt.plot(pfs , hom_plot[32:40], label = 'hom 60%', color='blue', linestyle='--')
plt.plot(pfs , het_plot[32:40], label = 'het 60%', color='blue')
plt.ylabel('$k_e$')
plt.xlabel('Packing Fraction')
plt.legend()
plt.show()

```



As observed in the graph above, the multiplication factor decreases exponentially as packing fraction decreases. The curve steepens with higher enrichments, such that the 5% appears comparatively linear when viewed against the higher enrichments. Additionally, as expected, multiplication factor increases with enrichment.

As for the proximity of the het. and hom. curves, each curve seems to diverge the most at the middle packing fractions (0.1 - 0.4). The hom. curves appear to be better approximates for the lower enrichments due to a lower nominal difference between k-effective values. However, as shown in the graphs below, for each one, the ratio of error to actual value decreases as packing fraction increases.

```

In [15]: dif_plot = []

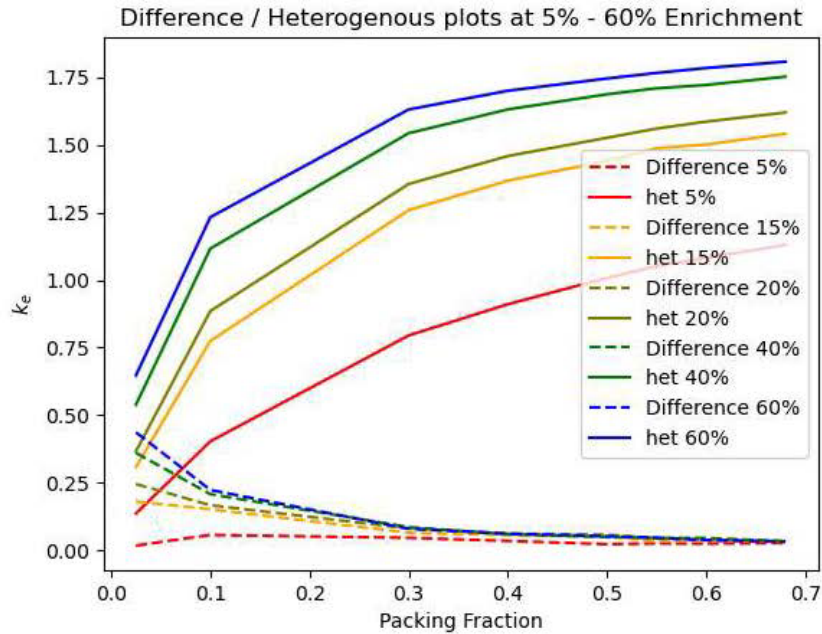
for i in range(len(hom_plot)):
    dif_plot.append((hom_plot[i] - het_plot[i]) / het_plot[i])

# pfs = [0.025, 0.10, 0.3, 0.4, 0.5, 0.55, 0.6, 0.68]

plt.figure()
plt.title('Difference / Heterogenous plots at 5% - 60% Enrichment')
plt.plot(pfs , dif_plot[0:8], label = 'Difference 5%', color='red', linestyle='--')
plt.plot(pfs , het_plot[0:8], label = 'het 5%', color='red')
plt.plot(pfs , dif_plot[8:16], label = 'Difference 15%', color='orange', linestyle='--')
plt.plot(pfs , het_plot[8:16], label = 'het 15%', color='orange')
plt.plot(pfs , dif_plot[16:24], label = 'Difference 20%', color='olive', linestyle='--')
plt.plot(pfs , het_plot[16:24], label = 'het 20%', color='olive')
plt.plot(pfs , dif_plot[24:32], label = 'Difference 40%', color='green', linestyle='--')
plt.plot(pfs , het_plot[24:32], label = 'het 40%', color='green')
plt.plot(pfs , dif_plot[32:40], label = 'Difference 60%', color='blue', linestyle='--')

```

```
plt.plot(pfs , het_plot[32:40], label = 'het 60%', color='blue')
plt.ylabel('$k_e$')
plt.xlabel('Packing Fraction')
plt.legend()
plt.show()
```



Using the Model Generation Function to get Volume Fraction for Study

```
In [23]: import pandas as pd

het_n, hom_n, vol_frac_55 = model_generator(60, 0.55)

values_table = pd.DataFrame({'Layer': ['Kernel', 'Buffer', 'Inner PyC', 'SiC', 'Outer PyC', 'Zirconium'],\
                             'Volume Fraction': vol_frac_55.T})

display(values_table)
```

	Layer	Volume Fraction
0	Kernel	0.020857
1	Buffer	0.131707
2	Inner PyC	0.091595
3	SiC	0.122429
4	Outer PyC	0.183412
5	Zirconium	0.450000

Appendix B - Notebook 2: Thermal Analysis

Determining Thermal Properties of the Channel

In the previous notebook, the volume ratios for the homogenous mixture were output after validating the the homogenous assumption validity. These values will be input during the creation of materials in this notebook.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
import uncertainties
from uncertainties import unumpy as unp
import openmc
import openmc.mgxs as mgxs
import openmc.data
import math
import openmc.lib
import pandas as pd
```

Function of Notebook

This notebook will investigate the production of recoverable energy along the length of the fuel. The next step will be to apply the curve to the equation for heat generation in a fuel pin and then applied to the heat exchange with the moderator.

Study Parameters

These are the set parameters for the design

Pitch	10.0
Fuel Radius	3.60
Inner Cladding	0.05
Outer Cladding	0.10
Coolant	1.314
Channel Length	400
Enrichment	45.0

```
In [2]: # Initial Settings from previous notebook, with minimum variables set for r_f and half_z
r_f = 3.60
th_ci = 0.05
th_c = 1.314
th_co = 0.10
r_g = 10.0
enrichment_percent = 45
half_z = 400 / 2

particles = 10000
openmc.config['cross_sections'] = "./opt/data/endl/cross_sections.xml"
```

Initializing Homogenous Matrix

Since only the geometry and settings file will be changing, the materials and tallies files can be set before the iteration.

Setting Up XML Files

```
In [3]: material_list = []
```

```
In [4]: # building the homogenous fuel
kernel = openmc.Material(name='UO2')
kernel.add_nuclide('U235', enrichment_percent/100)
kernel.add_nuclide('U238', 1-enrichment_percent/100)
kernel.add_element('O', 2.0)
kernel.set_density('g/cm3', 10.81)

PyC = openmc.Material(name='PyC')
PyC.set_density('g/cm3', 1.9)
PyC.add_element('C', 1.0)

buffer = openmc.Material(name='Buffer')
buffer.set_density('g/cm3', 1.0)
buffer.add_element('C', 1.0)

SiC = openmc.Material(name='SiC')
SiC.set_density('g/cm3', 3.2)
SiC.add_element('C', 0.5)
SiC.add_element('Si', 0.5)

zirconium = openmc.Material(name="ZrMatrix")
zirconium.add_element("Zr", 1.0)
zirconium.set_density("g/cm3", 6.505)

# Setting up the homogenous volume fractions from NB #1
vol_frac = [0.02085729, 0.13170702, 0.09159486, 0.12242857, 0.18341225, 0.45 ]
[fuel_vf, buffer_vf, IPyC_vf, SiC_vf, OPyC_vf, zirc_vf] = vol_frac

hom_fuel = openmc.Material.mix_materials([kernel, buffer, PyC, SiC, zirconium],
                                         vol_frac, percent_type="vo")
hom_fuel.add_s_alpha_beta('C_Graphite', 0.875)
material_list.append(hom_fuel)

# moderator and coolant
graphite = openmc.Material(name="Moderator")
graphite.add_element('carbon', 1.0)
graphite.set_density('g/cm3', 2.26)
graphite.add_s_alpha_beta('C_Graphite')
material_list.append(graphite)

cladding1 = openmc.Material(name="InnerCladding")
cladding1.add_element('C', 0.08, percent_type='wo')
cladding1.add_element('Si', 1.00, percent_type='wo')
cladding1.add_element('Mn', 2.00, percent_type='wo')
cladding1.add_element('P', 0.045, percent_type='wo')
cladding1.add_element('S', 0.030, percent_type='wo')
cladding1.add_element('Cr', 20.0, percent_type='wo')
cladding1.add_element('Ni', 11.0, percent_type='wo')
cladding1.add_element('Fe', 65.845, percent_type='wo')
cladding1.set_density('g/cm3', 8.0)
material_list.append(cladding1)

coolant = openmc.Material(name="Coolant")
coolant.set_density('g/cm3', 0.927)
coolant.add_element('sodium', 1.0)
material_list.append(coolant)
```

```

claddingo = openmc.Material(name="OuterCladding")
claddingo.add_element('C', 0.08, percent_type='wo')
claddingo.add_element('Si', 1.00, percent_type='wo')
claddingo.add_element('Mn', 2.00, percent_type='wo')
claddingo.add_element('P', 0.045, percent_type='wo')
claddingo.add_element('S', 0.030, percent_type='wo')
claddingo.add_element('Cr', 20.0, percent_type='wo')
claddingo.add_element('Ni', 11.0, percent_type='wo')
claddingo.add_element('Fe', 65.845, percent_type='wo')
claddingo.set_density('g/cm3', 8.0)
material_list.append(claddingo)

coolant_plenum = openmc.Material(name="Plenum")
coolant_plenum.add_element('sodium', 1.0)
coolant_plenum.set_density('g/cm3', 0.927)
material_list.append(coolant_plenum)

claddingc = openmc.Material(name="CapCladding")
claddingc.add_element('C', 0.08, percent_type='wo')
claddingc.add_element('Si', 1.00, percent_type='wo')
claddingc.add_element('Mn', 2.00, percent_type='wo')
claddingc.add_element('P', 0.045, percent_type='wo')
claddingc.add_element('S', 0.030, percent_type='wo')
claddingc.add_element('Cr', 20.0, percent_type='wo')
claddingc.add_element('Ni', 11.0, percent_type='wo')
claddingc.add_element('Fe', 65.845, percent_type='wo')
claddingc.set_density('g/cm3', 8.0)
material_list.append(claddingc)

materials_file = openmc.Materials(material_list)

```

Iterate Geometry

This iterative loop will run from the minimum values to the maximum values, filling the fuel geometry. Then all other cells are added in.

```

In [5]: z = np.arange(0, 2*half_z+0.1, 20)
z_list = [openmc.ZPlane(z0=z_i) for z_i in z]

lower_left = [-r_f, -r_f, z[0]]
upper_right = [+r_f, +r_f, z[-1]]
uniform_dist = openmc.stats.Box(lower_left, upper_right)

settings_file = openmc.Settings()
settings_file.batches = 200
settings_file.inactive = 25
settings_file.particles = particles
settings_file.temperature = {'multipole': True, 'method': 'interpolation', 'range': [290, 2500]}
settings_file.source = openmc.source.IndependentSource(space=uniform_dist, constraints={'fissionabl

channel_top = openmc.ZPlane(z0=z[-1])
channel_bot = openmc.ZPlane(z0=z[0])
plenum_top = openmc.ZPlane(z0=z[-1]+z[-1]/len(z), boundary_type = 'vacuum')
plenum_bot = openmc.ZPlane(z0=z[0] -z[-1]/len(z), boundary_type = 'vacuum')

exterior = openmc.model.HexagonalPrism(edge_length=r_g, orientation='y', origin = (0,0),
boundary_type = 'reflective', corner_radius = 0)

fuel_outer = openmc.ZCylinder(r = r_f)
clad_fuel_outer = openmc.ZCylinder(r = th_ci + r_f)
coolant_outer = openmc.ZCylinder(r = th_c + th_ci + r_f)

```

```

clad_mod_outer = openmc.ZCylinder(r = th_co + th_c + th_ci + r_f)
fuel_base = openmc.model.RightCircularCylinder(center_base = (0, 0, z[0]-th_ci), height = th_ci, radius
fuel_lid = openmc.model.RightCircularCylinder(center_base = (0, 0, z[-1]), height = th_ci, radius

root = openmc.Universe()

fuel_list = []

for i in range(0, len(z)-1):
    fuel_list.append(openmc.Cell(cell_id=i, name="HomFuel_2d" % (i)))

for j, fuels in enumerate(fuel_list):
    fuels.region = -fuel_outer & +z_list[j] & -z_list[j+1]
    fuels.fill = hom_fuel
    fuels.temperature = 900.

clad_inner = openmc.Cell()
coolant_cell = openmc.Cell()
clad_outer = openmc.Cell()
moderator_cell = openmc.Cell()
plenum_upper = openmc.Cell()
plenum_lower = openmc.Cell()
cap_upper = openmc.Cell()
cap_lower = openmc.Cell()

clad_inner.region = +fuel_outer & -clad_fuel_outer & +channel_bot & -channel_top
clad_inner.fill = cladding1
clad_inner.temperature = 700.
root.add_cell(clad_inner)

coolant_cell.region = +clad_fuel_outer & -coolant_outer & +channel_bot & -channel_top
coolant_cell.fill = coolant
coolant_cell.temperature = 450.
root.add_cell(coolant_cell)

clad_outer.region = +coolant_outer & -clad_mod_outer & +channel_bot & -channel_top
clad_outer.fill = cladding0
clad_outer.temperature = 425.
root.add_cell(clad_outer)

moderator_cell.region = +clad_mod_outer & -exterior & +channel_bot & -channel_top
moderator_cell.fill = graphite
moderator_cell.temperature = 400.
root.add_cell(moderator_cell)

root.add_cells(fuel_list)

plenum_upper = openmc.Cell()
plenum_lower = openmc.Cell()
cap_upper = openmc.Cell()
cap_lower = openmc.Cell()

cap_upper.region = -fuel_lid
cap_upper.fill = claddingc
cap_upper.temperature = 800.
root.add_cell(cap_upper)

cap_lower.region = -fuel_base
cap_lower.fill = claddingc
cap_lower.temperature = 700.
root.add_cell(cap_lower)

plenum_upper.region = -exterior & +channel_top & -plenum_top & +fuel_lid

```

```

plenun_upper.fill      = coolant_plenum
plenun_upper.temperature = 510.
root.add_cell(plenun_upper)

plenun_lower.region    = -exterior & -channel_bot & +plenun_bot & +fuel_base
plenun_lower.fill      = coolant_plenum
plenun_lower.temperature = 355.
root.add_cell(plenun_lower)

geometry_file = openmc.Geometry(root)

```

```

In [6]: # Establish a tally for counting energy from fissions as eV per source particle
fuel_filter = openmc.CellFilter(fuel_list)
recoverables = openmc.Tally(name='fiss_a')
recoverables.filters.append(fuel_filter)
recoverables.scores = ['fission-q-recoverable', 'heating', 'kappa-fission', 'flux']

tallies_file = openmc.Tallies([recoverables])

```

Plotting the Optimum Design

In order to fully analyze the neutronics at play, the simulation needs to be set up to collect tallies to tabulate the four factors (excluding leakage) that contribute to multiplication factor.

```

In [7]: # Plotting to validate the cross section of the universe.

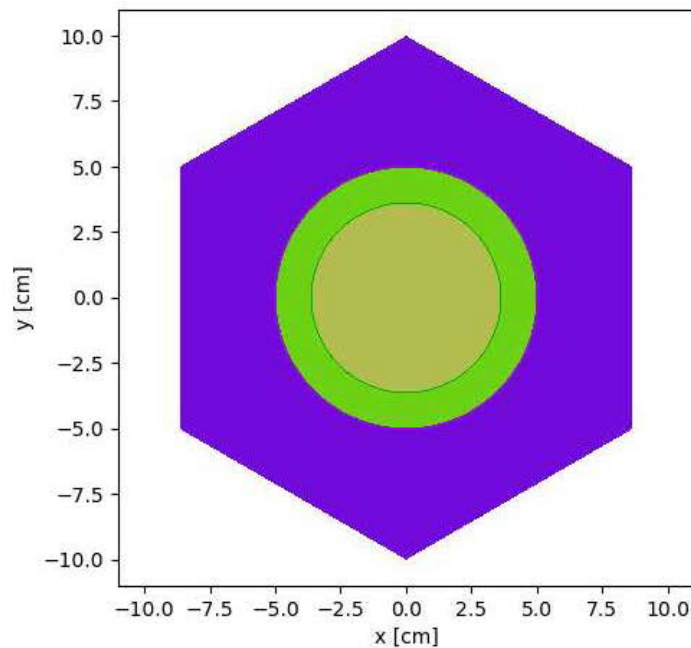
root.plot(basis='xy', width=[r_g*2.2, r_g*2.2], color_by='material', origin=[0., 0., half_z], pixels=[44

```

```

Out[7]: <Axes: xlabel='x [cm]', ylabel='y [cm]'>

```



```

In [8]: # Plotting to validate the cross section of the universe.

root.plot(basis='xz', width=[r_g*2.2, half_z*2.2], color_by='material', origin=[0., 0., (z[0]+z[-1])/2],

```


Establish Tallies

A generic set of multigroup characteristics (used to analyze results during design) as well as heating rate, fission rate, and energy from fission tallies are generated in this section.

```
In [10]: groups = mgxs.EnergyGroups(group_edges=np.array([0., 0.1e6, 10.0e6]))

#Instantiation of the cross sections
total_fuel = mgxs.TotalXS(      domain=hom_fuel, energy_groups=groups)
abs_fuel   = mgxs.AbsorptionXS(  domain=hom_fuel, energy_groups=groups)
scatter_fuel = mgxs.ScatterMatrixXS(domain=hom_fuel, energy_groups=groups)
nusigf_fuel = mgxs.FissionXS(    domain=hom_fuel, energy_groups=groups, nu=True)

total_clad1 = mgxs.TotalXS(      domain=cladding1, energy_groups=groups)
abs_clad1   = mgxs.AbsorptionXS(  domain=cladding1, energy_groups=groups)
scatter_clad1 = mgxs.ScatterMatrixXS(domain=cladding1, energy_groups=groups)
nusigf_clad1 = mgxs.FissionXS(    domain=cladding1, energy_groups=groups, nu=True)

total_coolant = mgxs.TotalXS(      domain=coolant, energy_groups=groups)
abs_coolant   = mgxs.AbsorptionXS(  domain=coolant, energy_groups=groups)
scatter_coolant = mgxs.ScatterMatrixXS(domain=coolant, energy_groups=groups)
nusigf_coolant = mgxs.FissionXS(    domain=coolant, energy_groups=groups, nu=True)

total_clado = mgxs.TotalXS(      domain=claddingo, energy_groups=groups)
abs_clado    = mgxs.AbsorptionXS(  domain=claddingo, energy_groups=groups)
scatter_clado = mgxs.ScatterMatrixXS(domain=claddingo, energy_groups=groups)
nusigf_clado = mgxs.FissionXS(    domain=claddingo, energy_groups=groups, nu=True)

total_mod = mgxs.TotalXS(      domain=graphite, energy_groups=groups)
abs_mod    = mgxs.AbsorptionXS(  domain=graphite, energy_groups=groups)
scatter_mod = mgxs.ScatterMatrixXS(domain=graphite, energy_groups=groups)
nusigf_mod = mgxs.FissionXS(    domain=graphite, energy_groups=groups, nu=True)

total_cladc = mgxs.TotalXS(      domain=claddingc, energy_groups=groups)
abs_cladc    = mgxs.AbsorptionXS(  domain=claddingc, energy_groups=groups)
scatter_cladc = mgxs.ScatterMatrixXS(domain=claddingc, energy_groups=groups)
nusigf_cladc = mgxs.FissionXS(    domain=claddingc, energy_groups=groups, nu=True)

total_plenum = mgxs.TotalXS(      domain=coolant_plenum, energy_groups=groups)
abs_plenum    = mgxs.AbsorptionXS(  domain=coolant_plenum, energy_groups=groups)
scatter_plenum = mgxs.ScatterMatrixXS(domain=coolant_plenum, energy_groups=groups)
nusigf_plenum = mgxs.FissionXS(    domain=coolant_plenum, energy_groups=groups, nu=True)

# Fuel Fission Cross Section Broken down by cell for thermal analysis
nusigf_fuelA = mgxs.FissionXS(    domain=openmc.Cell(cell_id=0), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelA.tallies.values()
nusigf_fuelB = mgxs.FissionXS(    domain=openmc.Cell(cell_id=1), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelB.tallies.values()
nusigf_fuelC = mgxs.FissionXS(    domain=openmc.Cell(cell_id=2), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelC.tallies.values()
nusigf_fuelD = mgxs.FissionXS(    domain=openmc.Cell(cell_id=3), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelD.tallies.values()
nusigf_fuelE = mgxs.FissionXS(    domain=openmc.Cell(cell_id=4), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelE.tallies.values()
nusigf_fuelF = mgxs.FissionXS(    domain=openmc.Cell(cell_id=5), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelF.tallies.values()
nusigf_fuelG = mgxs.FissionXS(    domain=openmc.Cell(cell_id=6), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelG.tallies.values()
nusigf_fuelH = mgxs.FissionXS(    domain=openmc.Cell(cell_id=7), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelH.tallies.values()
nusigf_fuelI = mgxs.FissionXS(    domain=openmc.Cell(cell_id=8), energy_groups=groups, nu=True)
```

```

tallies_file += nusigf_fuelI.tallies.values()
nusigf_fuelJ = mgxs.FissionXS( domain=openmc.Cell(cell_id=9), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelJ.tallies.values()
nusigf_fuelK = mgxs.FissionXS( domain=openmc.Cell(cell_id=10), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelK.tallies.values()
nusigf_fuelL = mgxs.FissionXS( domain=openmc.Cell(cell_id=11), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelL.tallies.values()
nusigf_fuelM = mgxs.FissionXS( domain=openmc.Cell(cell_id=12), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelM.tallies.values()
nusigf_fuelN = mgxs.FissionXS( domain=openmc.Cell(cell_id=13), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelN.tallies.values()
nusigf_fuelO = mgxs.FissionXS( domain=openmc.Cell(cell_id=14), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelO.tallies.values()
nusigf_fuelP = mgxs.FissionXS( domain=openmc.Cell(cell_id=15), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelP.tallies.values()
nusigf_fuelQ = mgxs.FissionXS( domain=openmc.Cell(cell_id=16), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelQ.tallies.values()
nusigf_fuelR = mgxs.FissionXS( domain=openmc.Cell(cell_id=17), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelR.tallies.values()
nusigf_fuelS = mgxs.FissionXS( domain=openmc.Cell(cell_id=18), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelS.tallies.values()
nusigf_fuelT = mgxs.FissionXS( domain=openmc.Cell(cell_id=19), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelT.tallies.values()
nusigf_fuelU = mgxs.FissionXS( domain=openmc.Cell(cell_id=20), energy_groups=groups, nu=True)
tallies_file += nusigf_fuelU.tallies.values()

#Add cross section tallies to the tallies file
tallies_file += total_fuel.tallies.values()
tallies_file += abs_fuel.tallies.values()
tallies_file += scatter_fuel.tallies.values()
tallies_file += nusigf_fuel.tallies.values()

tallies_file += total_cladi.tallies.values()
tallies_file += abs_cladi.tallies.values()
tallies_file += scatter_cladi.tallies.values()
tallies_file += nusigf_cladi.tallies.values()

tallies_file += total_coolant.tallies.values()
tallies_file += abs_coolant.tallies.values()
tallies_file += scatter_coolant.tallies.values()
tallies_file += nusigf_coolant.tallies.values()

tallies_file += total_clado.tallies.values()
tallies_file += abs_clado.tallies.values()
tallies_file += scatter_clado.tallies.values()
tallies_file += nusigf_clado.tallies.values()

tallies_file += total_mod.tallies.values()
tallies_file += abs_mod.tallies.values()
tallies_file += scatter_mod.tallies.values()
tallies_file += nusigf_mod.tallies.values()

tallies_file += total_cladc.tallies.values()
tallies_file += abs_cladc.tallies.values()
tallies_file += scatter_cladc.tallies.values()
tallies_file += nusigf_cladc.tallies.values()

tallies_file += total_plenum.tallies.values()
tallies_file += abs_plenum.tallies.values()
tallies_file += scatter_plenum.tallies.values()
tallies_file += nusigf_plenum.tallies.values()

```

Run the Model and Pull Data from the Statepoint File

The model is only run a single time, but each of the vertical sections generates its own set of tallies, which will be computationally used below.

```
In [11]: model = openmc.Model(geometry=geometry_file, materials=materials_file, settings=settings_file, tallies=tallies)
statepoint_R6 = model.run()
```

Omitted Output for Appendix

```

In [12]: #Loading the statepoint file
sp = openmc.StatePoint(statepoint_R6)

nusigf_fuelA.load_from_statepoint(sp)
nusigf_fuelB.load_from_statepoint(sp)
nusigf_fuelC.load_from_statepoint(sp)
nusigf_fuelD.load_from_statepoint(sp)
nusigf_fuelE.load_from_statepoint(sp)
nusigf_fuelF.load_from_statepoint(sp)
nusigf_fuelG.load_from_statepoint(sp)
nusigf_fuelH.load_from_statepoint(sp)
nusigf_fuelI.load_from_statepoint(sp)
nusigf_fuelJ.load_from_statepoint(sp)
nusigf_fuelK.load_from_statepoint(sp)
nusigf_fuelL.load_from_statepoint(sp)
nusigf_fuelM.load_from_statepoint(sp)
nusigf_fuelN.load_from_statepoint(sp)
nusigf_fuelO.load_from_statepoint(sp)
nusigf_fuelP.load_from_statepoint(sp)
nusigf_fuelQ.load_from_statepoint(sp)
nusigf_fuelR.load_from_statepoint(sp)
nusigf_fuelS.load_from_statepoint(sp)
nusigf_fuelT.load_from_statepoint(sp)
nusigf_fuelU.load_from_statepoint(sp)

# Tallies within the fuel
heating_tally = sp.get_tally(name='fiss_a')
flux_tally = heating_tally.get_slice(scores = ['flux'])
flux_tally = flux_tally.mean.ravel()
ev_tally = heating_tally.get_slice(scores = ['fission-q-recoverable'])
ev_tally = ev_tally.mean.ravel()
k_tally = heating_tally.get_slice(scores = ['kappa-fission'])
k_tally = k_tally.mean.ravel()
h_tally = heating_tally.get_slice(scores = ['heating'])
h_tally = h_tally.mean.ravel()

```



```

total_fuel.load_from_statepoint(sp)
abs_fuel.load_from_statepoint(sp)
scatter_fuel.load_from_statepoint(sp)
nusigf_fuel.load_from_statepoint(sp)

total_cladi.load_from_statepoint(sp)
abs_cladi.load_from_statepoint(sp)
scatter_cladi.load_from_statepoint(sp)
nusigf_cladi.load_from_statepoint(sp)

total_coolant.load_from_statepoint(sp)
abs_coolant.load_from_statepoint(sp)
scatter_coolant.load_from_statepoint(sp)
nusigf_coolant.load_from_statepoint(sp)

total_clado.load_from_statepoint(sp)
abs_clado.load_from_statepoint(sp)
scatter_clado.load_from_statepoint(sp)
nusigf_clado.load_from_statepoint(sp)

total_mod.load_from_statepoint(sp)
abs_mod.load_from_statepoint(sp)
scatter_mod.load_from_statepoint(sp)
nusigf_mod.load_from_statepoint(sp)

total_cladc.load_from_statepoint(sp)
abs_cladc.load_from_statepoint(sp)
scatter_cladc.load_from_statepoint(sp)
nusigf_cladc.load_from_statepoint(sp)

total_plenum.load_from_statepoint(sp)
abs_plenum.load_from_statepoint(sp)
scatter_plenum.load_from_statepoint(sp)
nusigf_plenum.load_from_statepoint(sp)

k_eff = sp.keff

#closing the summary file
sp.close()

```

```

In [13]: nusigfflux_t_A, nusigfflux_f_A = nusigf_fuelA.get_flux(order_groups='decreasing')
          nusigfflux_t_B, nusigfflux_f_B = nusigf_fuelB.get_flux(order_groups='decreasing')
          nusigfflux_t_C, nusigfflux_f_C = nusigf_fuelC.get_flux(order_groups='decreasing')
          nusigfflux_t_D, nusigfflux_f_D = nusigf_fuelD.get_flux(order_groups='decreasing')
          nusigfflux_t_E, nusigfflux_f_E = nusigf_fuelE.get_flux(order_groups='decreasing')
          nusigfflux_t_F, nusigfflux_f_F = nusigf_fuelF.get_flux(order_groups='decreasing')
          nusigfflux_t_G, nusigfflux_f_G = nusigf_fuelG.get_flux(order_groups='decreasing')
          nusigfflux_t_H, nusigfflux_f_H = nusigf_fuelH.get_flux(order_groups='decreasing')
          nusigfflux_t_I, nusigfflux_f_I = nusigf_fuelI.get_flux(order_groups='decreasing')
          nusigfflux_t_J, nusigfflux_f_J = nusigf_fuelJ.get_flux(order_groups='decreasing')
          nusigfflux_t_K, nusigfflux_f_K = nusigf_fuelK.get_flux(order_groups='decreasing')
          nusigfflux_t_L, nusigfflux_f_L = nusigf_fuelL.get_flux(order_groups='decreasing')
          nusigfflux_t_M, nusigfflux_f_M = nusigf_fuelM.get_flux(order_groups='decreasing')
          nusigfflux_t_N, nusigfflux_f_N = nusigf_fuelN.get_flux(order_groups='decreasing')
          nusigfflux_t_O, nusigfflux_f_O = nusigf_fuelO.get_flux(order_groups='decreasing')
          nusigfflux_t_P, nusigfflux_f_P = nusigf_fuelP.get_flux(order_groups='decreasing')
          nusigfflux_t_Q, nusigfflux_f_Q = nusigf_fuelQ.get_flux(order_groups='decreasing')
          nusigfflux_t_R, nusigfflux_f_R = nusigf_fuelR.get_flux(order_groups='decreasing')
          nusigfflux_t_S, nusigfflux_f_S = nusigf_fuelS.get_flux(order_groups='decreasing')
          nusigfflux_t_T, nusigfflux_f_T = nusigf_fuelT.get_flux(order_groups='decreasing')
          nusigfflux_t_U, nusigfflux_f_U = nusigf_fuelU.get_flux(order_groups='decreasing')

          nusigf_t_A, nusigf_f_A = nusigf_fuelA.get_xs(order_groups='decreasing')

```

```

nusigf_t_B, nusigf_f_B = nusigf_fuelB.get_xs(order_groups='decreasing')
nusigf_t_C, nusigf_f_C = nusigf_fuelC.get_xs(order_groups='decreasing')
nusigf_t_D, nusigf_f_D = nusigf_fuelD.get_xs(order_groups='decreasing')
nusigf_t_E, nusigf_f_E = nusigf_fuelE.get_xs(order_groups='decreasing')
nusigf_t_F, nusigf_f_F = nusigf_fuelF.get_xs(order_groups='decreasing')
nusigf_t_G, nusigf_f_G = nusigf_fuelG.get_xs(order_groups='decreasing')
nusigf_t_H, nusigf_f_H = nusigf_fuelH.get_xs(order_groups='decreasing')
nusigf_t_I, nusigf_f_I = nusigf_fuelI.get_xs(order_groups='decreasing')
nusigf_t_J, nusigf_f_J = nusigf_fuelJ.get_xs(order_groups='decreasing')
nusigf_t_K, nusigf_f_K = nusigf_fuelK.get_xs(order_groups='decreasing')
nusigf_t_L, nusigf_f_L = nusigf_fuelL.get_xs(order_groups='decreasing')
nusigf_t_M, nusigf_f_M = nusigf_fuelM.get_xs(order_groups='decreasing')
nusigf_t_N, nusigf_f_N = nusigf_fuelN.get_xs(order_groups='decreasing')
nusigf_t_O, nusigf_f_O = nusigf_fuelO.get_xs(order_groups='decreasing')
nusigf_t_P, nusigf_f_P = nusigf_fuelP.get_xs(order_groups='decreasing')
nusigf_t_Q, nusigf_f_Q = nusigf_fuelQ.get_xs(order_groups='decreasing')
nusigf_t_R, nusigf_f_R = nusigf_fuelR.get_xs(order_groups='decreasing')
nusigf_t_S, nusigf_f_S = nusigf_fuelS.get_xs(order_groups='decreasing')
nusigf_t_T, nusigf_f_T = nusigf_fuelT.get_xs(order_groups='decreasing')
nusigf_t_U, nusigf_f_U = nusigf_fuelU.get_xs(order_groups='decreasing')

axial_thermal_flux = []
axial_thermal_flux.append( nusigfflux_t_A)
axial_thermal_flux.append( nusigfflux_t_B)
axial_thermal_flux.append( nusigfflux_t_C)
axial_thermal_flux.append( nusigfflux_t_D)
axial_thermal_flux.append( nusigfflux_t_E)
axial_thermal_flux.append( nusigfflux_t_F)
axial_thermal_flux.append( nusigfflux_t_G)
axial_thermal_flux.append( nusigfflux_t_H)
axial_thermal_flux.append( nusigfflux_t_I)
axial_thermal_flux.append( nusigfflux_t_J)
axial_thermal_flux.append( nusigfflux_t_K)
axial_thermal_flux.append( nusigfflux_t_L)
axial_thermal_flux.append( nusigfflux_t_M)
axial_thermal_flux.append( nusigfflux_t_N)
axial_thermal_flux.append( nusigfflux_t_O)
axial_thermal_flux.append( nusigfflux_t_P)
axial_thermal_flux.append( nusigfflux_t_Q)
axial_thermal_flux.append( nusigfflux_t_R)
axial_thermal_flux.append( nusigfflux_t_S)
axial_thermal_flux.append( nusigfflux_t_T)
axial_thermal_flux.append( nusigfflux_t_U)

axial_thermal_x = []
axial_thermal_x.append( nusigf_t_A)
axial_thermal_x.append( nusigf_t_B)
axial_thermal_x.append( nusigf_t_C)
axial_thermal_x.append( nusigf_t_D)
axial_thermal_x.append( nusigf_t_E)
axial_thermal_x.append( nusigf_t_F)
axial_thermal_x.append( nusigf_t_G)
axial_thermal_x.append( nusigf_t_H)
axial_thermal_x.append( nusigf_t_I)
axial_thermal_x.append( nusigf_t_J)
axial_thermal_x.append( nusigf_t_K)
axial_thermal_x.append( nusigf_t_L)
axial_thermal_x.append( nusigf_t_M)
axial_thermal_x.append( nusigf_t_N)
axial_thermal_x.append( nusigf_t_O)
axial_thermal_x.append( nusigf_t_P)
axial_thermal_x.append( nusigf_t_Q)
axial_thermal_x.append( nusigf_t_R)

```

```

axial_thermal_x.append( nusigf_t_S)
axial_thermal_x.append( nusigf_t_T)
axial_thermal_x.append( nusigf_t_U)

axial_fast_flux = []
axial_fast_flux.append( nusigfflux_f_A)
axial_fast_flux.append( nusigfflux_f_B)
axial_fast_flux.append( nusigfflux_f_C)
axial_fast_flux.append( nusigfflux_f_D)
axial_fast_flux.append( nusigfflux_f_E)
axial_fast_flux.append( nusigfflux_f_F)
axial_fast_flux.append( nusigfflux_f_G)
axial_fast_flux.append( nusigfflux_f_H)
axial_fast_flux.append( nusigfflux_f_I)
axial_fast_flux.append( nusigfflux_f_J)
axial_fast_flux.append( nusigfflux_f_K)
axial_fast_flux.append( nusigfflux_f_L)
axial_fast_flux.append( nusigfflux_f_M)
axial_fast_flux.append( nusigfflux_f_N)
axial_fast_flux.append( nusigfflux_f_O)
axial_fast_flux.append( nusigfflux_f_P)
axial_fast_flux.append( nusigfflux_f_Q)
axial_fast_flux.append( nusigfflux_f_R)
axial_fast_flux.append( nusigfflux_f_S)
axial_fast_flux.append( nusigfflux_f_T)
axial_fast_flux.append( nusigfflux_f_U)

axial_fast_x = []
axial_fast_x.append( nusigf_f_A)
axial_fast_x.append( nusigf_f_B)
axial_fast_x.append( nusigf_f_C)
axial_fast_x.append( nusigf_f_D)
axial_fast_x.append( nusigf_f_E)
axial_fast_x.append( nusigf_f_F)
axial_fast_x.append( nusigf_f_G)
axial_fast_x.append( nusigf_f_H)
axial_fast_x.append( nusigf_f_I)
axial_fast_x.append( nusigf_f_J)
axial_fast_x.append( nusigf_f_K)
axial_fast_x.append( nusigf_f_L)
axial_fast_x.append( nusigf_f_M)
axial_fast_x.append( nusigf_f_N)
axial_fast_x.append( nusigf_f_O)
axial_fast_x.append( nusigf_f_P)
axial_fast_x.append( nusigf_f_Q)
axial_fast_x.append( nusigf_f_R)
axial_fast_x.append( nusigf_f_S)
axial_fast_x.append( nusigf_f_T)
axial_fast_x.append( nusigf_f_U)

total_t_fuel, total_f_fuel = total_fuel.get_xs(order_groups='decreasing')
abs_t_fuel, abs_f_fuel = abs_fuel.get_xs(order_groups='decreasing')
scatter_t_fuel, scatter_f_fuel = scatter_fuel.get_xs(order_groups='decreasing')
nusigf_t_fuel, nusigf_f_fuel = nusigf_fuel.get_xs(order_groups='decreasing')

totalflux_t_fuel, totalflux_f_fuel = total_fuel.get_flux(order_groups='decreasing')
absflux_t_fuel, absflux_f_fuel = abs_fuel.get_flux(order_groups='decreasing')
scatterflux_t_fuel, scatterflux_f_fuel = scatter_fuel.get_flux(order_groups='decreasing')
nusigfflux_t_fuel, nusigfflux_f_fuel = nusigf_fuel.get_flux(order_groups='decreasing')

total_t_cladi, total_f_cladi = total_cladi.get_xs(order_groups='decreasing')
abs_t_cladi, abs_f_cladi = abs_cladi.get_xs(order_groups='decreasing')
scatter_t_cladi, scatter_f_cladi = scatter_cladi.get_xs(order_groups='decreasing')

```



```

nusigf_t_cladi, nusigf_f_cladi = nusigf_cladi.get_xs(order_groups='decreasing')

totalflux_t_cladi, totalflux_f_cladi = total_cladi.get_flux(order_groups='decreasing')
absflux_t_cladi, absflux_f_cladi = abs_cladi.get_flux(order_groups='decreasing')
scatterflux_t_cladi, scatterflux_f_cladi = scatter_cladi.get_flux(order_groups='decreasing')
nusigfflux_t_cladi, nusigfflux_f_cladi = nusigf_cladi.get_flux(order_groups='decreasing')

total_t_coolant, total_f_coolant = total_coolant.get_xs(order_groups='decreasing')
abs_t_coolant, abs_f_coolant = abs_coolant.get_xs(order_groups='decreasing')
scatter_t_coolant, scatter_f_coolant = scatter_coolant.get_xs(order_groups='decreasing')
nusigf_t_coolant, nusigf_f_coolant = nusigf_coolant.get_xs(order_groups='decreasing')

totalflux_t_coolant, totalflux_f_coolant = total_coolant.get_flux(order_groups='decreasing')
absflux_t_coolant, absflux_f_coolant = abs_coolant.get_flux(order_groups='decreasing')
scatterflux_t_coolant, scatterflux_f_coolant = scatter_coolant.get_flux(order_groups='decreasing')
nusigfflux_t_coolant, nusigfflux_f_coolant = nusigf_coolant.get_flux(order_groups='decreasing')

total_t_clado, total_f_clado = total_clado.get_xs(order_groups='decreasing')
abs_t_clado, abs_f_clado = abs_clado.get_xs(order_groups='decreasing')
scatter_t_clado, scatter_f_clado = scatter_clado.get_xs(order_groups='decreasing')
nusigf_t_clado, nusigf_f_clado = nusigf_clado.get_xs(order_groups='decreasing')

totalflux_t_clado, totalflux_f_clado = total_clado.get_flux(order_groups='decreasing')
absflux_t_clado, absflux_f_clado = abs_clado.get_flux(order_groups='decreasing')
scatterflux_t_clado, scatterflux_f_clado = scatter_clado.get_flux(order_groups='decreasing')
nusigfflux_t_clado, nusigfflux_f_clado = nusigf_clado.get_flux(order_groups='decreasing')

total_t_mod, total_f_mod = total_mod.get_xs(order_groups='decreasing')
abs_t_mod, abs_f_mod = abs_mod.get_xs(order_groups='decreasing')
scatter_t_mod, scatter_f_mod = scatter_mod.get_xs(order_groups='decreasing')
nusigf_t_mod, nusigf_f_mod = nusigf_mod.get_xs(order_groups='decreasing')

totalflux_t_mod, totalflux_f_mod = total_mod.get_flux(order_groups='decreasing')
absflux_t_mod, absflux_f_mod = abs_mod.get_flux(order_groups='decreasing')
scatterflux_t_mod, scatterflux_f_mod = scatter_mod.get_flux(order_groups='decreasing')
nusigfflux_t_mod, nusigfflux_f_mod = nusigf_mod.get_flux(order_groups='decreasing')

total_t_cladc, total_f_cladc = total_cladc.get_xs(order_groups='decreasing')
abs_t_cladc, abs_f_cladc = abs_cladc.get_xs(order_groups='decreasing')
scatter_t_cladc, scatter_f_cladc = scatter_cladc.get_xs(order_groups='decreasing')
nusigf_t_cladc, nusigf_f_cladc = nusigf_cladc.get_xs(order_groups='decreasing')

totalflux_t_cladc, totalflux_f_cladc = total_cladc.get_flux(order_groups='decreasing')
absflux_t_cladc, absflux_f_cladc = abs_cladc.get_flux(order_groups='decreasing')
scatterflux_t_cladc, scatterflux_f_cladc = scatter_cladc.get_flux(order_groups='decreasing')
nusigfflux_t_cladc, nusigfflux_f_cladc = nusigf_cladc.get_flux(order_groups='decreasing')

total_t_plenum, total_f_plenum = total_plenum.get_xs(order_groups='decreasing')
abs_t_plenum, abs_f_plenum = abs_plenum.get_xs(order_groups='decreasing')
scatter_t_plenum, scatter_f_plenum = scatter_plenum.get_xs(order_groups='decreasing')
nusigf_t_plenum, nusigf_f_plenum = nusigf_plenum.get_xs(order_groups='decreasing')

totalflux_t_plenum, totalflux_f_plenum = total_plenum.get_flux(order_groups='decreasing')
absflux_t_plenum, absflux_f_plenum = abs_plenum.get_flux(order_groups='decreasing')
scatterflux_t_plenum, scatterflux_f_plenum = scatter_plenum.get_flux(order_groups='decreasing')
nusigfflux_t_plenum, nusigfflux_f_plenum = nusigf_plenum.get_flux(order_groups='decreasing')

```


Utilizing Output Tally Data

The section above categorized the output data into variables, which is now capable of being simply called in the computations below. First, volumetric data will be used to generate four-factor analysis of the data (unused in study, but used to alter during the design process), then the heating and energy rates will be analyzed.

```
In [29]: V_fuel      = r_f**2 * math.pi * z[-1]
V_cladi   = (th_ci + r_f)**2 * math.pi * z[-1] - V_fuel
V_coolant = (th_c + th_ci + r_f)**2 * math.pi * z[-1] - V_fuel - V_cladi
V_clado   = (th_co + th_c + th_ci + r_f)**2 * math.pi * z[-1] - V_fuel - V_cladi - V_coolant
V_mod     = r_g**2 * math.pi * z[-1] - V_fuel - V_cladi - V_coolant - V_clado
V_cladc   = (th_ci + r_f)**2 * math.pi * th_ci
V_coolantp = 3 * math.sqrt(3) * r_g**2 / 2 * z[-1] * 2
```

```

print(V_fuel, V_cladi + V_clado + V_cladc, V_coolant + V_coolantp, V_mod)
16286.016316209489 1717.7792733848244 222069.7154526979 93438.38470297118

In [17]: z[-1]/len(z)

Out[17]: np.float64(19.047619047619047)

In [18]: fissionprod_th = nusigf_t_fuel * nusigfflux_t_fuel * V_fuel
fissionprod_fast = nusigf_f_fuel * nusigfflux_f_fuel * V_fuel
epsilon = (fissionprod_th + fissionprod_fast) / (fissionprod_th)

absorption_th_fuel = V_fuel * abs_t_fuel * absflux_t_fuel
absorption_th_coolant = V_coolant * abs_t_coolant * absflux_t_coolant + \
V_coolantp * abs_t_plenum * absflux_t_plenum
absorption_th_clad = V_cladi * abs_t_cladi * absflux_t_cladi + \
V_clado * abs_t_clado * absflux_t_clado + \
V_cladc * abs_t_cladc * absflux_t_cladc
absorption_th_mod = V_mod * abs_t_mod * absflux_t_mod
f = absorption_th_fuel / (absorption_th_fuel + (absorption_th_coolant + absorption_th_clad + absorption_

absorption_fast_fuel = V_fuel * abs_f_fuel * absflux_f_fuel
total_th_fuel = V_fuel * total_t_fuel * totalflux_t_fuel
total_th_coolant = V_coolant * total_t_coolant * totalflux_t_coolant + \
V_coolantp * total_t_plenum * totalflux_t_plenum
total_th_clad = V_cladi * total_t_cladi * totalflux_t_cladi + \
V_clado * total_t_clado * totalflux_t_clado + \
V_cladc * total_t_cladc * totalflux_t_cladc
p = 1 - absorption_fast_fuel / (total_th_fuel + total_th_coolant + total_th_clad)

eta_T = fissionprod_th / absorption_th_fuel

k_inf = epsilon * p * f * eta_T

values_table = pd.DataFrame({'Variable': [r'$\epsilon$', 'p', 'f', r'$\eta_T$', r'$k_{inf}$'], \
'Value': [epsilon, p, f, eta_T, k_inf]})
display(values_table)

```

	Variable	Value
0	ϵ	1.009264
1	p	0.999379
2	f	0.818261
3	η_T	1.752644
4	k_{inf}	1.446508

```

In [19]: values_table2 = pd.DataFrame({'Parameter': ['Pitch', 'Fuel Radius', 'Cladding', 'Coolant', 'Channel Length', \
'Value': [r_g, r_f, th_ci, th_c, half_z*2, enrichment_percent]})
display(values_table2)

```

	Parameter	Value
0	Pitch	10.000
1	Fuel Radius	3.600
2	Cladding	0.050
3	Coolant	1.314
4	Channel Length	400.000
5	Enrichment	45.000

```
In [20]: # Since the most troublesome of the four factors was f, each of its components are broken down here
# and were used to assess the validity of the calculations for k-effective.

denom = (absorption_th_fuel + (absorption_th_coolant + absorption_th_clad + absorption_th_mod))
variable_names = ['absorption_th_fuel', 'denominator', 'absorption_th_mod', 'absorption_th_coolant', 'absorption_th_clad']
xs_values = [absorption_th_fuel, denom, absorption_th_mod, absorption_th_coolant, absorption_th_clad]
values_table3 = pd.DataFrame({'Variable': variable_names, 'Cross Section': xs_values})
display(values_table3)
```

	Variable	Cross Section
0	absorption_th_fuel	12064.381066
1	denominator	14743.922556
2	absorption_th_mod	1270.085118
3	absorption_th_coolant	1244.901106
4	absorption_th_clad	164.555265

K-effective, Heating Tally, Recoverable Energy Tally, and the Kappa-Fission Tally (unused)

```
In [21]: k_eff

Out[21]: 1.311407809387036+/-0.0006531419223093708

In [22]: h_tally

Out[22]: array([ 17729.93608972,  37337.41028943,  55852.89931937,  72588.2339598 ,
                87656.96538015, 101259.10523936, 112985.32842733, 122140.58783552,
                128467.84094903, 132266.13550386, 132050.66630052, 128860.0664882 ,
                122908.05938531, 114411.46479892, 102678.01219266,  89152.33973253,
                73140.74408064,  54592.63876326,  35982.00336596, 16827.75526045])

In [23]: ev_tally

Out[23]: array([1013236.03450974, 2217058.97952549, 3338080.43621011,
                4326937.211922 , 5234723.63685091, 6053537.05301467,
                6789430.94756363, 7289445.15487958, 7671238.18267863,
                7911517.90863918, 7922989.96515314, 7708683.68168453,
                7374224.91021636, 6845291.46414298, 6137449.72123661,
                5289035.99713456, 4378945.71139079, 3307559.1495664 ,
                2168672.19893747,  977062.56283479])

In [24]: k_tally
```

```
Out[24]: array([1013305.90247997, 2217207.73213662, 3338301.18372172,
4327226.579297 , 5235073.04859677, 6053939.1107907 ,
6789880.16722664, 7289930.08519705, 7671749.82930104,
7912043.21511772, 7923514.35277479, 7709193.89941724,
7374714.21273144, 6845748.69784143, 6137857.80036525,
5289389.09378294, 4379235.95831074, 3307775.96814716,
2168816.23632809, 977128.8223304 ])
```

Midpoint setting for graphical representation and flux plotting

```
In [25]: midpoints = (z[1:-1]+z[0:-2])/2
```

```
In [26]: axial_thermal_flux
```

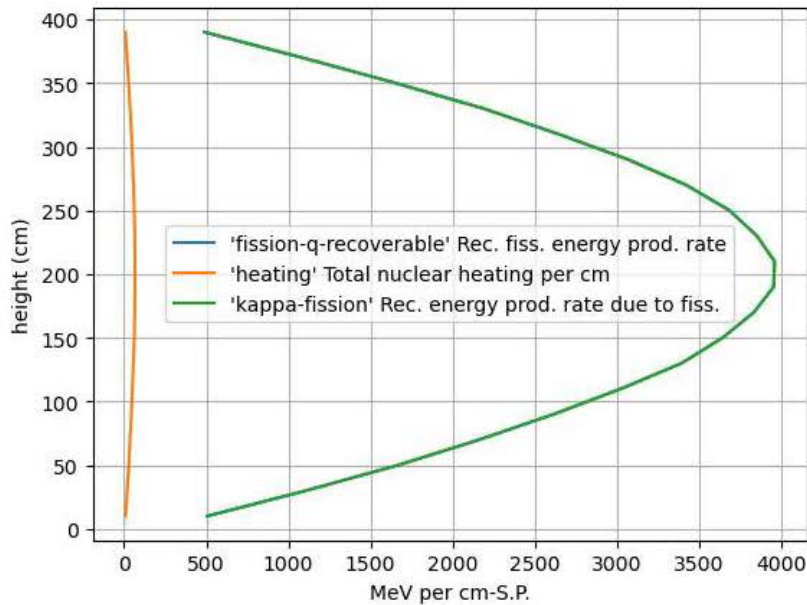
```
Out[26]: [np.float64(0.3289644905940131),
np.float64(0.6964951759388457),
np.float64(1.042267177036758),
np.float64(1.3567160883581224),
np.float64(1.6336112574259334),
np.float64(1.890806191235331),
np.float64(2.109030134907094),
np.float64(2.2781871992325837),
np.float64(2.397220685098163),
np.float64(2.4653446741916207),
np.float64(2.4721659326541308),
np.float64(2.403462513064009),
np.float64(2.2988333495303723),
np.float64(2.1350692722166422),
np.float64(1.9156423563660965),
np.float64(1.663692558026024),
np.float64(1.3671730564193998),
np.float64(1.0253366058317124),
np.float64(0.677240882982027),
np.float64(0.31457250026249867),
np.float64(0.9488094983617306)]
```

```
In [27]: from operator import add
axial_flux = list(map(add, axial_thermal_flux, axial_fast_flux))

heightsMid = []
for height in range(len(z)-1):
    heightsMid.append((z[height]+z[height+1])/2)

fig, ax = plt.subplots()
ax.plot(ev_tally/20*particles/10**6, heightsMid, label="'fission-q-recoverable' Rec. fiss. energy prod.")
ax.plot(h_tally/20*particles/10**6, heightsMid, label="'heating' Total nuclear heating per cm")
ax.plot(k_tally/20*particles/10**6, heightsMid, label="'kappa-fission' Rec. energy prod. rate due to fis")
ax.set_xlabel('MeV per cm-S.P.')
ax.set_ylabel('height (cm)')
ax.grid()
ax.legend(loc = 'center')
```

```
Out[27]: <matplotlib.legend.Legend at 0x7f6e6aa5d7f0>
```



Comparison of Fast and Thermal Neutron production in the Fuel

```
In [50]: fast_interactions = []
therm_interactions = []
total_fissions = []
for entry in range(len(z)-1):
    fast_interactions.append(axial_fast_flux[entry]*axial_fast_x[entry]*particles/(heightsMid[1]-heights
    therm_interactions.append(axial_thermal_flux[entry]*axial_thermal_x[entry]*particles/(heightsMid[1]-
    total_fissions.append(fast_interactions[entry]+therm_interactions[entry])

mean_fr = np.mean(total_fissions)
mean_fr_array = np.full((len(z)-1), mean_fr)

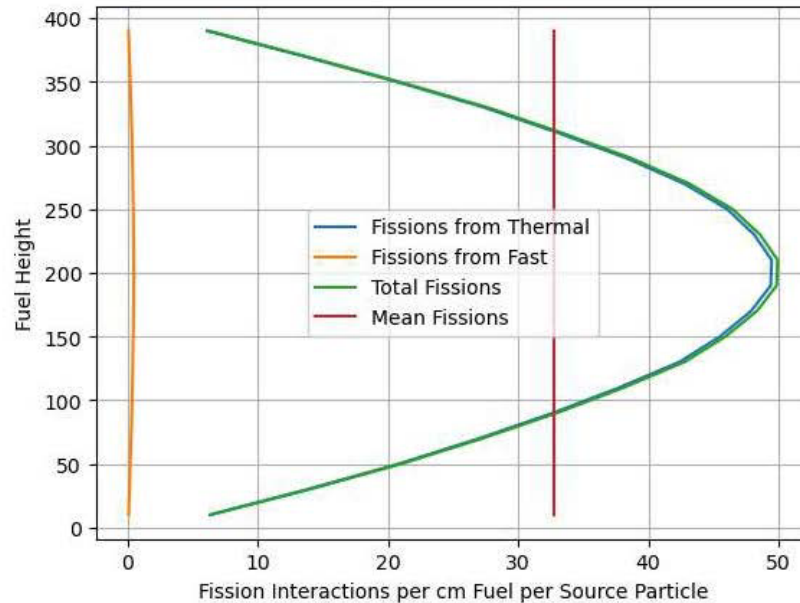
print(f'Mean fission rate is approximately {mean_fr} fissions/cm')
max_fr = np.max(total_fissions)
sum_th = np.sum(therm_interactions)
sum_fast = np.sum(fast_interactions)
tf_ratio = sum_th / sum_fast
print(f'Max fission rate is approximately {max_fr} fissions/cm')
print(f'The ratio between these two values is {max_fr/mean_fr}')
print(f'The fission rate for the entire rod is {mean_fr*2*half_z} fissions/rod')
print(f'The ratio of thermal to fast interactions is {tf_ratio} and {100*sum_fast/(sum_th+sum_fast)} per

fig, ax = plt.subplots()
ax.plot(therm_interactions, heightsMid, label='Fissions from Thermal') # part-cm/source part * 1/cm * so
ax.plot(fast_interactions, heightsMid, label='Fissions from Fast')
ax.plot(total_fissions, heightsMid, label='Total Fissions')
ax.plot(mean_fr_array, heightsMid, label='Mean Fissions')

ax.set_xlabel('Fission Interactions per cm Fuel per Source Particle')
ax.set_ylabel('Fuel Height')
ax.grid()
ax.legend(loc='center')
```


Mean fission rate is approximately 32.75885452213278 fissions/cm
 Max fission rate is approximately 49.934448612489454 fissions/cm
 The ratio between these two values is 1.524303866569949
 The fission rate for the entire rod is 13103.541808853111 fissions/rod
 The ratio of thermal to fast interactions is 107.94533606329779 and 0.9178915189347755 percent of fission
 s are produced by fast neutron absorption.

Out[50]: <matplotlib.legend.Legend at 0x7f6e637715b0>



Neutrons Absorbed versus Neutrons Produced in the Fuel (unused)

```
In [29]: fig, ax = plt.subplots()

energy_production = []
rec_production_fuel = []
heat_production_fuel = []

for cell in range(len(z)-1):
    energy_production.append(total_fissions[cell]*ev_tally[cell]/10**6)
    rec_production_fuel.append(total_fissions[cell]*k_tally[cell]/10**6)
    heat_production_fuel.append(total_fissions[cell]*h_tally[cell]/10**6)

sum_e_prod = np.sum(energy_production)
sum_h_fuel = np.sum(heat_production_fuel)
sum_k_fuel = np.sum(rec_production_fuel)
lambda_ah_f = sum_h_fuel / sum_e_prod

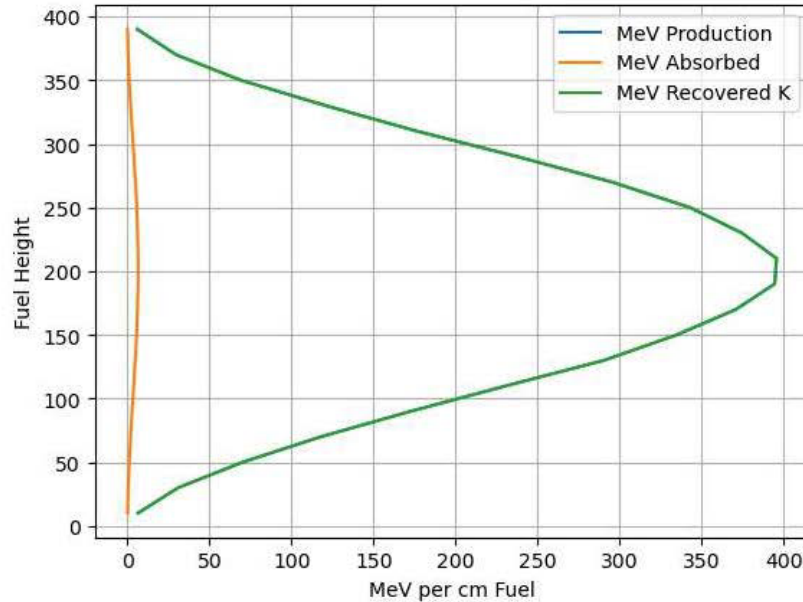
print(f'{lambda_ah_f*100}% of energy produced in the fuel is absorbed in the fuel as heat.')

ax.plot(energy_production, heightsMid, label='MeV Production') # part/cm *eV/part/10^6 = MeV/cm
ax.plot(heat_production_fuel, heightsMid, label='MeV Absorbed')
ax.plot(rec_production_fuel, heightsMid, label='MeV Recovered K')
ax.set_label('Axial MeV Production and Absorption by Fuel')
ax.set_xlabel('MeV per cm Fuel')
ax.set_ylabel('Fuel Height')
```

```
ax.grid()
ax.legend()
```

1.6716777788651358% of energy produced in the fuel is absorbed in the fuel as heat.

Out[29]: <matplotlib.legend.Legend at 0x7f6c6a98a180>



Fitting a curve to the Heating Data

In the Nuclear Systems (Todreas & Kazimi) text, page 830, $q'''(z) = q_o''' * \cos(\pi * z / L_c)$ gives the approximate profile for single heated channel or an axially heated length of coolant such as the one we simulate here. By fitting this data to a cosine curve, q_o''' and L_c can be found.

```
In [44]: from scipy.optimize import curve_fit

# Define the cosine model function
def cosina_model(x, amplitude, frequency, phase):
    return amplitude * np.cos(frequency * x + phase)

# Fit the curve to the data
initial_guess = [ev_tally[10]/20./10**6, 1/810, 0] # Initial guess for A, B, C
params, covariance = curve_fit(cosina_model, z[0:-1], ev_tally/20./10**6, p0=initial_guess)

# Extract optimized parameters
optimized_amplitude, optimized_frequency, optimized_phase = params

# Print optimized parameters
print(f'Peak q\'' is approximately {optimized_amplitude*particles/10**6} MeV/cm')
print(f'Extrapolated length: {1/optimized_frequency*np.pi}')
print(f'The peak heat generation occurs {-optimized_phase} cm above the midpoint')
print(f'The equation for the heating is {optimized_amplitude} * COS({optimized_frequency} * z + {optimiz

fitheight = z[0:-1]
fitcurve = optimized_amplitude * np.cos(optimized_frequency * fitheight + optimized_phase)

# Plot the data and the fit
```

```

plt.stairs(ev_tally/20./10**6, z, label='Recovered Energy', orientation='horizontal', color='red')
plt.plot(fitcurve, fitheight+10, label='Fitted Cosine Curve', color='green', linestyle='--')
plt.ylim(0, 2*half_z)
plt.xlabel('Recoverable MeV per S.P.-cm')
plt.ylabel('Fuel Height (cm)')
plt.title('Approximating q\' Profile')
plt.legend()
plt.grid()
plt.show()

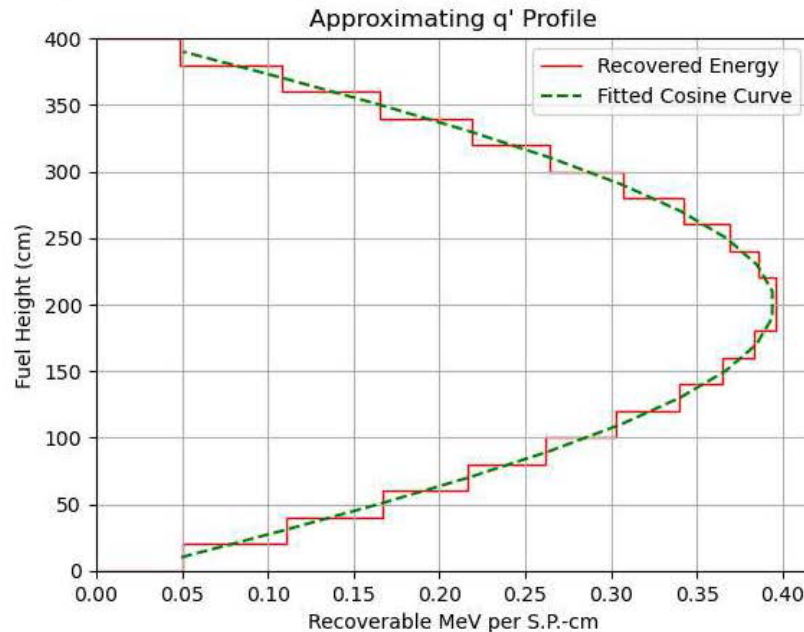
```

Peak q' is approximately 0.003950576040499124 MeV/cm

Extrapolated length: 413.47338478367936

The peak heat generation occurs 1.4448835186183573 cm above the midpoint

The equation for the heating is $0.3950576040499124 * \cos(0.007598052907887671 * z + -1.4448835186183573)$



```

In [49]: # Fit the curve to the data
initial_guess2 = [rec_production_fuel[10]/20, 1/810, 1] # Initial guess for A, B, C
params2, covariance2 = curve_fit(cosine_model, fitheight, rec_production_fuel, p0=initial_guess2)

# Extract optimized parameters
optimized_amplitude2, optimized_frequency2, optimized_phase2 = params2

heights = z[0:-1] + 10
mean_q = np.mean(rec_production_fuel)
max_q = np.max(rec_production_fuel)
mean_q_array = np.full((len(z)-1), mean_q)

# Print optimized parameters and values
print(f'Mean q\' is {mean_q} MeV/cm')
print(f'Extrapolated length: {1/optimized_frequency2*np.pi}')
print(f'The peak heat generation is {max_q} MeV/cm and occurs {-optimized_phase2} cm above the midpoint')
print(f'The ratio between peak and mean values is {max_q/mean_q}')
print(f'The thermal energy production is {mean_q*400} MeV per Rod')
print(f'The equation for the heating is {optimized_amplitude2} * COS({optimized_frequency2} * z + {optim

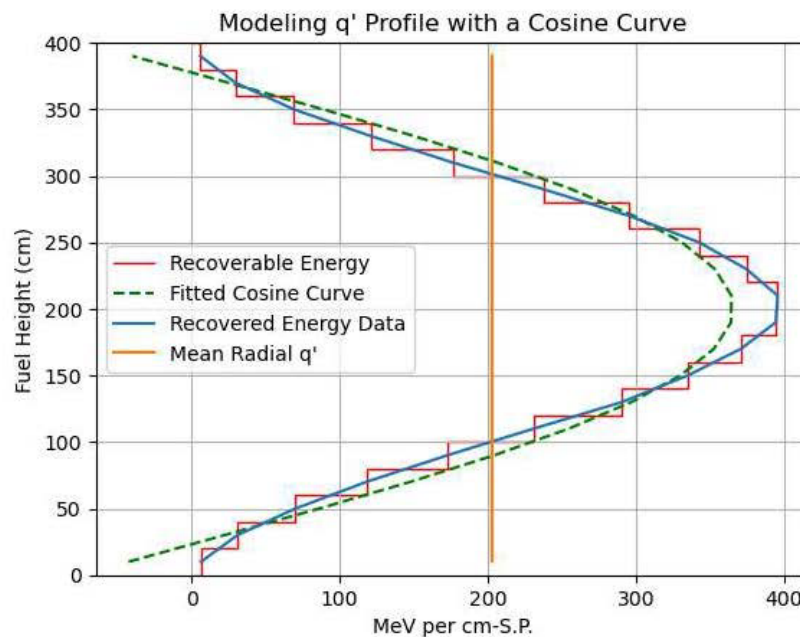
fitcurve2 = optimized_amplitude2 * np.cos(optimized_frequency2 * fitheight + optimized_phase2)

```



```
# Plot the data and the fit
plt.stairs(rec_production_fuel, z, label='Recoverable Energy', orientation='horizontal', color='red')
plt.plot(fitcurve2, fitheight+10, label='Fitted Cosine Curve', color='green', linestyle='--')
plt.plot(rec_production_fuel, heights, label='Recovered Energy Data')
plt.plot(mean_q_array, heights, label='Mean Radial q\'')
plt.ylim(0, 2*half_z)
plt.xlabel('MeV per cm-S.P.')
plt.ylabel('Fuel Height (cm)')
plt.title("Modeling q' Profile with a Cosine Curve")
plt.legend()
plt.grid()
plt.show()
```

Mean q' is 203.38790902594064 MeV/cm
 Extrapolated length: 413.47338478367936
 The peak heat generation is 395.6563202789556 MeV/cm and occurs 1.4448835186183573 cm above the midpoint
 The ratio between peak and mean values is 1.9453286194534432
 The thermal energy production is 81355.16361037626 MeV per Rod
 The equation for the heating is $365.82015876008415 * \cos(-0.008861396088144543 * z + 1.6872025124045973)$



This resultant graph seemed too far from the actual values of the graph. Visually, it seemed better represented by a bell-curve or a Gaussian Curve. This shape is tested below, with more successful results.

```
In [47]: # Define the Gaussian function
def gaussian(x, amplitude, mean, stddev):
    return amplitude * np.exp(-(x - mean) / stddev)**2 / 2)

# Fit the Gaussian curve to the data
popt, pcov = curve_fit(gaussian, heights, rec_production_fuel, p0=[8, 200, 50])

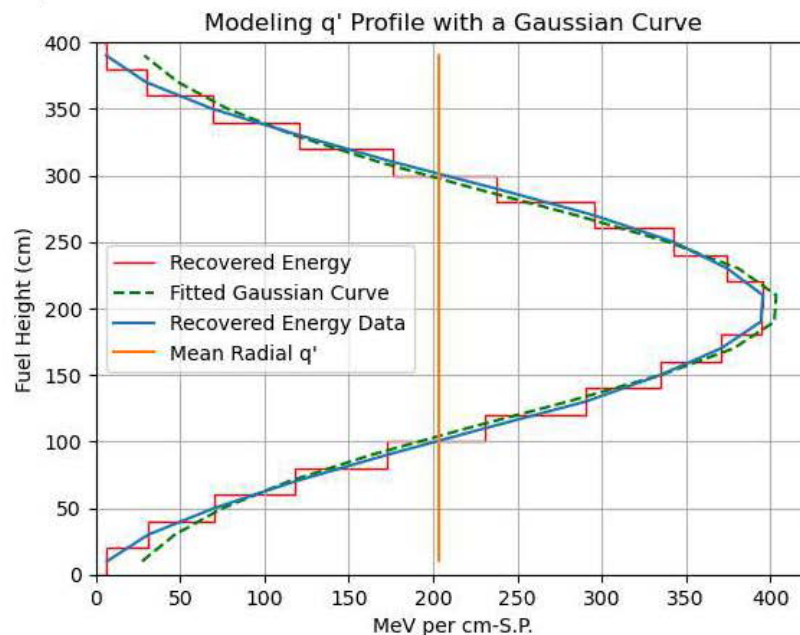
# Extract the fitted parameters
amplitude_fit, mean_fit, stddev_fit = popt

# Generate the fitted curve
y_fit = gaussian(heights, amplitude_fit, mean_fit, stddev_fit)
```

```
# Print optimized parameters and values
print(f'The peak heat generation is {max_q} MeV/cm and occurs {200 - mean_fit} cm below the midpoint')
print(f'The equation for the heating is {amplitude_fit} * e^{{{mean_fit} - z} / {stddev_fit}}^2 / 2)')

# Plot the data and the fit
plt.stairs(rec_production_fuel, z, label='Recovered Energy', orientation='horizontal', color='red')
plt.plot(y_fit, heights, label='Fitted Gaussian Curve', color='green', linestyle='--')
plt.plot(rec_production_fuel, heights, label='Recovered Energy Data')
plt.plot(mean_q_array, heights, label='Mean Radial q\'')
plt.ylim(0, 2*half_z)
plt.xlabel('MeV per cm-S.P.')
plt.ylabel('Fuel Height (cm)')
plt.title('Modeling q\' Profile with a Gaussian Curve')
plt.legend(loc='center left')
plt.grid()
plt.show()
```

The peak heat generation is 395.6563202789556 MeV/cm and occurs -0.8110944313635571 cm below the midpoint
The equation for the heating is 406.05606424042355 * e^{((200.81109443136356 - z) / -82.13512325945197)² / 2)}



```
In [34]: from sklearn.metrics import r2_score

cosine_error = ((rec_production_fuel - fitcurve2)**2)**(1/2)
gauss_error = ((rec_production_fuel - y_fit)**2)**(1/2)
mean_err_c = np.mean(cosine_error)
max_err_c = np.max(cosine_error)
mean_err_g = np.mean(gauss_error)
max_err_g = np.max(gauss_error)

r_squared_c = r2_score(rec_production_fuel, fitcurve2)
print("R-squared:", r_squared_c)
r_squared_g = r2_score(rec_production_fuel, y_fit)
print("R-squared:", r_squared_g)
```

R-squared: 0.966154684473047
R-squared: 0.9931483990692744

Appendix C - Notebook 3: Depletion Analysis

Studying the Effects of Depletion over the Lifespan of the SGTR

The efforts up to this point have primarily focused on the initial steady state of a pin cell. In the following simulations, the consequences of fuel burnup and depletion on the multiplication factor are investigated. As before, the simulation will be iterative; however, this study will instead vary the time linearly to study effects on the eigenvalue.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
import uncertainties
from uncertainties import unumpy as unp
import openmc
import openmc.mgxs as mgxs
import openmc.data
import math
import openmc.lib
import pandas as pd
```

Instantiate Materials and Register the Nuclides

The particles in this notebook adjust the accuracy of the results at the expense of calculation time and memory use. Rod Count is used here solely to influence the reactor-level statements output after running the code.

```
In [2]: # Initial Settings
r_f = 3.60
th_ci = 0.05
th_c = 1.314
th_co = 0.10
r_g = 10.0
enrichment_percent = 45
half_z = 400 / 2

r_ci = r_f + th_ci
r_c = r_ci + th_c
r_co = r_c + th_co

rod_count = 700
particles = 5000
openmc.config['cross_sections'] = './opt/data/endl/cross_sections.xml'

# Depletion Settings for ease of access
time_step_count = 60
time_steps = [30] * time_step_count
power = 142857
```

Initiate Materials, Build Geometry, and Input Settings

The geometry of this section will need the additional data of volume in order to calculate the depletion of the fissile material (for the homogenous fuel mixture, this is U-235).

```
In [3]: # homogenous materials
kernel = openmc.Material(name='UO2')
```

```

kernel.add_nuclide('U235', enrichment_percent/100)
kernel.add_nuclide('U238', 1-enrichment_percent/100)
kernel.add_element('O', 2.0)
kernel.set_density('g/cm3', 10.81)

PyC = openmc.Material(name='PyC1')
PyC.set_density('g/cm3', 1.9)
PyC.add_element('C', 1.0)

buffer = openmc.Material(name='Buffer')
buffer.set_density('g/cm3', 1.0)
buffer.add_element('C', 1.0)

SiC = openmc.Material(name='SiC')
SiC.set_density('g/cm3', 3.2)
SiC.add_element('C', 0.5)
SiC.add_element('Si', 0.5)

zirconium = openmc.Material(name="ZrMatrix")
zirconium.add_element("Zr", 1.0)
zirconium.set_density('g/cm3', 6.505)

# Setting up the homogenous volume fractions from Appendix A
vol_frac = [0.02085729, 0.13170702, 0.09159486, 0.12242857, 0.18341225, 0.45 ]
[fuel_vf, buffer_vf, IPyC_vf, SiC_vf, OPyC_vf, zirc_vf] = vol_frac
PyC_vf = IPyC_vf + OPyC_vf
vol_frac = [fuel_vf, buffer_vf, PyC_vf, SiC_vf, zirc_vf]

hom_fuel = openmc.Material.mix_materials([kernel, buffer, PyC, SiC, zirconium],
                                         vol_frac, percent_type="vo")
hom_fuel.volume = r_f ** 2 * math.pi * half_z * 2
hom_fuel.add_s_alpha_beta('c_Graphite', 0.875)

# moderator and coolant
graphite = openmc.Material(name="Moderator")
graphite.add_element('carbon', 1.0)
graphite.set_density('g/cm3', 2.26)
graphite.add_s_alpha_beta('c_Graphite')

cladding1 = openmc.Material(name="Cladding")
cladding1.add_element('C', 0.08, percent_type='wo')
cladding1.add_element('Si', 1.00, percent_type='wo')
cladding1.add_element('Mn', 2.00, percent_type='wo')
cladding1.add_element('P', 0.045, percent_type='wo')
cladding1.add_element('S', 0.030, percent_type='wo')
cladding1.add_element('Cr', 20.0, percent_type='wo')
cladding1.add_element('Ni', 11.0, percent_type='wo')
cladding1.add_element('Fe', 65.845, percent_type='wo')
cladding1.set_density('g/cm3', 8.0)

coolant = openmc.Material(name="Coolant")
coolant.add_element('sodium', 1.0)
coolant.set_density('g/cm3', 0.927)

cladding0 = openmc.Material(name="Cladding")
cladding0.add_element('C', 0.08, percent_type='wo')
cladding0.add_element('Si', 1.00, percent_type='wo')
cladding0.add_element('Mn', 2.00, percent_type='wo')
cladding0.add_element('P', 0.045, percent_type='wo')
cladding0.add_element('S', 0.030, percent_type='wo')
cladding0.add_element('Cr', 20.0, percent_type='wo')
cladding0.add_element('Ni', 11.0, percent_type='wo')
cladding0.add_element('Fe', 65.845, percent_type='wo')

```



```

claddingo.set_density('g/cm3', 8.0)

coolant_plenum = openmc.Material(name="Plenum")
coolant_plenum.add_element('sodium', 1.0)
coolant_plenum.set_density('g/cm3', 0.927)

claddingc = openmc.Material(name="Cladding")
claddingc.add_element('C', 0.08, percent_type='wo')
claddingc.add_element('Si', 1.00, percent_type='wo')
claddingc.add_element('Mn', 2.00, percent_type='wo')
claddingc.add_element('P', 0.045, percent_type='wo')
claddingc.add_element('S', 0.030, percent_type='wo')
claddingc.add_element('Cr', 20.0, percent_type='wo')
claddingc.add_element('Ni', 11.0, percent_type='wo')
claddingc.add_element('Fe', 65.845, percent_type='wo')
claddingc.set_density('g/cm3', 8.0)

materials_file = openmc.Materials([hom_fuel, claddingi, coolant, claddingo, graphite, claddingc, coolant

```

```

In [4]: z = [-half_z, half_z]
channel_top = openmc.ZPlane(z0=z[1])
channel_bot = openmc.ZPlane(z0=z[0])
plenum_top = openmc.ZPlane(z0=2*z[1], boundary_type = 'vacuum')
plenum_bot = openmc.ZPlane(z0=2*z[0], boundary_type = 'vacuum')

exterior = openmc.model.HexagonalPrism(edge_length=r_g, orientation='y', origin = (0,0),
                                         boundary_type = 'reflective', corner_radius = 0)

fuel_outer = openmc.ZCylinder(r = r_f)
clad_fuel_outer = openmc.ZCylinder(r = r_ci)
coolant_outer = openmc.ZCylinder(r = r_c)
clad_mod_outer = openmc.ZCylinder(r = r_co)

th_ci = r_ci - r_f
fuel_base = openmc.model.RightCircularCylinder(center_base = (0, 0, z[0]-th_ci), height = th_ci, radius
fuel_lid = openmc.model.RightCircularCylinder(center_base = (0, 0, z[1]), height = th_ci, radius

```

```

In [5]: root = openmc.Universe(name='root universe')
fuel_cell = openmc.Cell()
clad_inner = openmc.Cell()
coolant_cell = openmc.Cell()
clad_outer = openmc.Cell()
moderator_cell = openmc.Cell()
plenum_upper = openmc.Cell()
plenum_lower = openmc.Cell()
cap_upper = openmc.Cell()
cap_lower = openmc.Cell()

fuel_cell.region = -fuel_outer & +channel_bot & -channel_top
fuel_cell.fill = hom_fuel
fuel_cell.temperature = 600.
fuel_cell.volume = r_f ** 2 * math.pi * half_z * 2
hom_fuel.volume = fuel_cell.volume
root.add_cell(fuel_cell)

clad_inner.region = +fuel_outer & -clad_fuel_outer & +channel_bot & -channel_top
clad_inner.fill = claddingi
clad_inner.temperature = 580.
root.add_cell(clad_inner)

coolant_cell.region = +clad_fuel_outer & -coolant_outer & +channel_bot & -channel_top
coolant_cell.fill = coolant
coolant_cell.temperature = 560.
root.add_cell(coolant_cell)

```

```

clad_outer.region      = +coolant_outer & -clad_mod_outer & +channel_bot & -channel_top
clad_outer.fill        = claddingo
clad_outer.temperature = 520.
root.add_cell(clad_outer)

moderator_cell.region  = +clad_mod_outer & -exterior & +channel_bot & -channel_top
moderator_cell.fill    = graphite
moderator_cell.temperature = 500.
root.add_cell(moderator_cell)

cap_upper.region       = -fuel_lid
cap_upper.fill         = claddingc
cap_upper.temperature  = 650.
root.add_cell(cap_upper)

cap_lower.region       = -fuel_base
cap_lower.fill         = claddingc
cap_lower.temperature  = 355.
root.add_cell(cap_lower)

plenum_upper.region    = -exterior & +channel_top & -plenum_top & +fuel_lid
plenum_upper.fill      = coolant
plenum_upper.temperature = 650.
root.add_cell(plenum_upper)

plenum_lower.region    = -exterior & -channel_bot & +plenum_bot & +fuel_base
plenum_lower.fill      = coolant
plenum_lower.temperature = 355.
root.add_cell(plenum_lower)

geometry_file = openmc.Geometry(root)

```

```

In [6]: lower_left = [-r_f, -r_f, z[0]]
upper_right = [+r_f, +r_f, z[-1]]
uniform_dist = openmc.stats.Box(lower_left, upper_right)

settings_file = openmc.Settings()
settings_file.batches = 100
settings_file.inactive = 20
settings_file.particles = particles
settings_file.temperature = {'multipole': True, 'method': 'interpolation', 'range': [290, 2500]}
settings_file.source = openmc.source.IndependentSource(space=uniform_dist, constraints={'fissionabl

```

Establishing Standard Volumes

In order for the phi calculations to be accurate, standard values for volumes need to be determined. For the purposes of depletion calculations, the caps and plenums are neglected.

```

In [7]: V_fuel = r_f**2 * math.pi * z[-1]
V_cladi = r_ci**2 * math.pi * z[-1] - V_fuel
V_cool = r_c**2 * math.pi * z[-1] - V_fuel - V_cladi
V_clado = r_co**2 * math.pi * z[-1] - V_fuel - V_cladi - V_cool
V_mod = 3 * math.sqrt(3) * r_g**2 / 2 * z[-1] - V_fuel - V_cladi - V_cool - V_clado

```

Setting up Tallies over Multigroup cross-sections

While Lewis defines three energy groups in chapter four, two of his energy groups will be combined in this simulation to count all intermediate region as fast. The lower energy bound is zero, the dividing boundary is 0.1

MeV, and the upper bound is 10 MeV.

The tallies defined here establish a library that will be accessed for each statepoint to calculate each of the four factors.

```
In [8]: groups = mgxs.EnergyGroups([1e-3, 1.0, 10.0e6])

tallies_file = openmc.Tallies()

spectrum_bounds = openmc.mgxs.GROUP_STRUCTURES['CASMO-70']
energy_filter = openmc.EnergyFilter(spectrum_bounds)
cell_filter = openmc.CellFilter([fuel_cell, clad_inner, coolant_cell, clad_outer, moderator_cell])
spectrum_tally = openmc.Tally()
spectrum_tally.filters = [cell_filter, energy_filter]
spectrum_tally.scores = ['flux']
tallies_file.append(spectrum_tally)

xs_lib = openmc.mgxs.Library(geometry_file)
xs_lib.energy_groups = groups
xs_lib.mgxs_types = ['total', 'absorption', 'nu-fission', 'fission', 'scatter matrix', 'chi']
xs_lib.domain_type = 'cell'
xs_lib.domains = geometry_file.get_all_material_cells().values()
xs_lib.by_nuclide = False
xs_lib.build_library()
xs_lib.add_to_tallies_file(tallies_file, merge=True)
```

Setting up Depletion and Running Code

First we need to load the deplete class object, then we establish data for the decay chains. Next, we set all the information for the integrate function to run before finally running it through each time step iteratively.

```
In [9]: import openmc.deplete
chain = openmc.deplete.Chain.from_xml("./opt/data/endlf/chain_endfb71_pwr.xml")

In [10]: SGTR_model = openmc.Model(materials = materials_file, geometry = geometry_file, settings = settings_file)

In [11]: operator = openmc.deplete.CoupledOperator(SGTR_model, chain_file="./opt/data/endlf/chain_endfb71_pwr.xml")

In [12]: integrator = openmc.deplete.PredictorIntegrator(operator, time_steps, power, timestep_units='d')
integrator.integrate()
```

Output omitted for Appendix.

```
In [13]: results = openmc.deplete.Results("./depletion_results.h5")
time, k = results.get_keff()
time /= (24 * 60 * 60) # convert back to days from seconds
```

Calling each statepoint and calculating factors

As in the previous notebook, the factor arrays are instantiated, then an iterative loop is run to extract values from each statepoint, calculate the four factors and k, and close the statepoint.

```
In [14]: sp = openmc.StatePoint("./openmc_simulation_n{}.h5".format(0))
sp.tallies

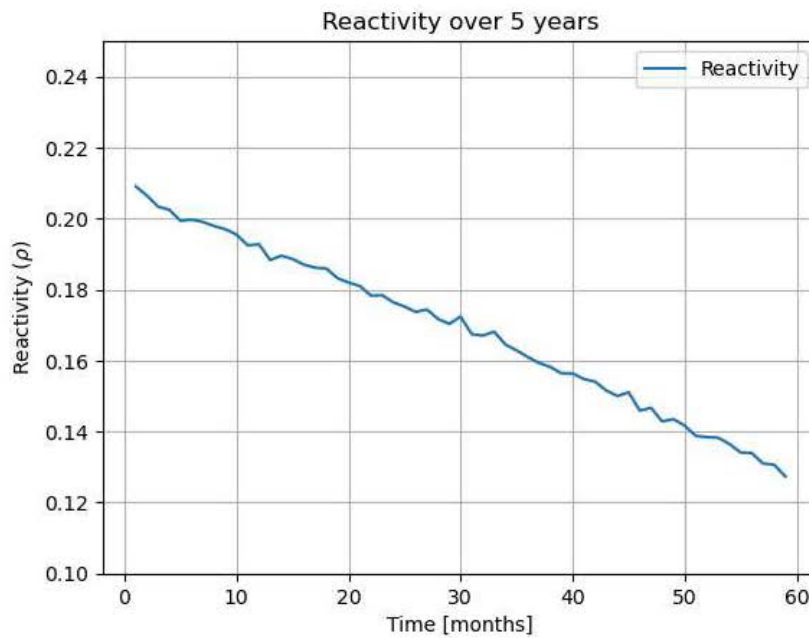
keff_values = unp.nominal_values(k[:,0])
```

```
In [15]: # from matplotlib import pyplot
_, u235 = results.get_mass("6", "U235")
_, u238 = results.get_mass("6", "U238")
_, xe135 = results.get_mass("6", "Xe135")
_, i131 = results.get_mass("6", "I131")
_, cs137 = results.get_mass("6", "Cs137")
_, sr90 = results.get_mass("6", "Sr90")
_, ba141 = results.get_mass("6", "Ba141")
_, kr90 = results.get_mass("6", "Kr90")

uranium = []
rho = []
remaining_235 = []
for diff in range(len(time)):
    uranium.append(u235[diff]+u238[diff])
    remaining_235.append(u235[diff])
    rho.append((keff_values[diff]-1)/keff_values[diff])

plt.title("Reactivity over 5 years")
plt.plot(time[1:-1]/30, rho[1:-1], label='Reactivity')
plt.xlabel("Time [months]")
plt.grid(True)
plt.legend(loc='upper right')
plt.ylabel("Reactivity ($\\rho$)");
plt.ylim(0.1, 0.25)
```

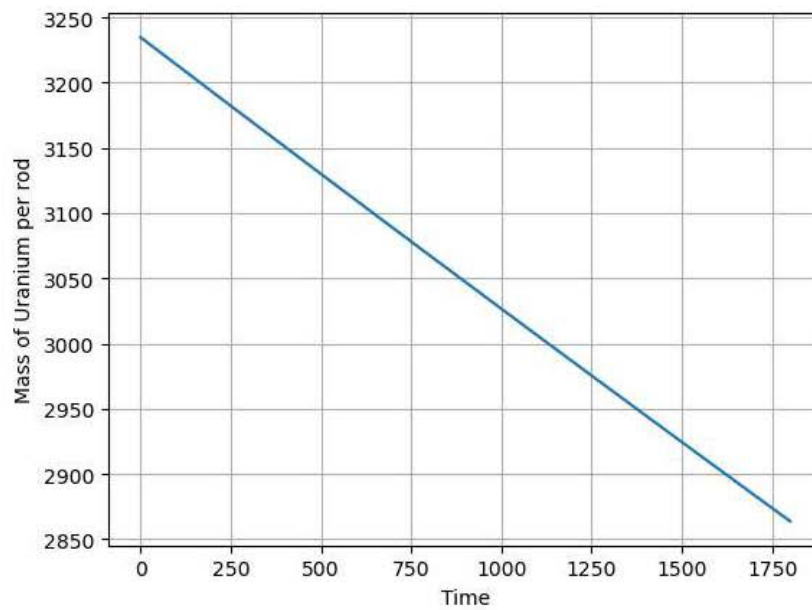
```
Out[15]: (0.1, 0.25)
```

```
In [16]: print(f'Reactor Uranium Mass is {uranium[0]*rod_count/1000} kg, at {uranium[0]} grams in each of the {rod_count} rods')
plt.suptitle("Mass of Uranium per Rod over 60 Months")
plt.plot(time, uranium)
plt.xlabel("Time")
plt.grid(True)
plt.ylabel("Mass of Uranium per rod");
```

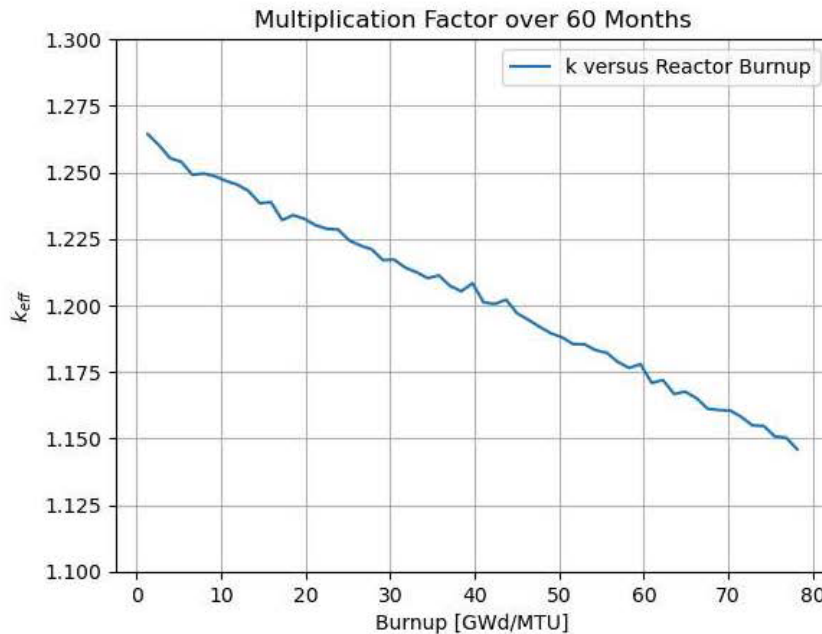
Reactor Uranium Mass is 2264.340722289828 kg, at 3234.7724604140403 grams in each of the 700 rods, depleted at EOL to 2863.8509624866224 grams.

Mass of Uranium per Rod over 60 Months



```
In [17]: plt.title("Multiplication Factor over 60 Months")
plt.plot(time[1:-1]/((uranium[0]))*power/1000, keff_values[1:-1], label='k versus Reactor Burnup')
plt.xlabel("Burnup [Gd/MTU]")
plt.grid(True)
plt.legend(loc='upper right')
plt.ylabel("$k_{eff}$");
plt.ylim(1.1, 1.3)
```

```
Out[17]: (1.1, 1.3)
```



```
In [18]: print(f'During this period, k decreases from {keff_values[0]} to {keff_values[-1]} \
and reactivity decreases from {rho[0]}$ to {rho[-1]}$.'
```

During this period, k decreases from 1.2797141541600923 to 1.144180052309506 and reactivity decreases from 0.2185754945749393\$ to 0.12601168148184483\$.

```
In [19]: # Create the figure and the first y-axis
fig, ax1 = plt.subplots()
fig.suptitle("U-235 Mass Depletion per Channel")

# Plot the first Line on the first y-axis
color = 'tab:blue'
ax1.set_xlabel("Time [months]")
ax1.set_ylabel("U-235 Remaining per Rod [g]", color=color)
ax1.plot(time/30, remaining_235, color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.set_xlim(0, time[-1]/30)

# Create the second y-axis, sharing the same x-axis
ax2 = ax1.twinx()

# Plot the second Line on the second y-axis
color = 'tab:red'
ax2.set_ylabel("Reactivity", color=color)
ax2.plot(time/30, rho, color=color)
ax2.tick_params(axis='y', labelcolor=color)

# Calculate values for burnup in relatable units
burned = (u235[0]-u235[-1]) # g
power_gen = power * time [-1] # Mwd
burnup = power_gen / uranium[0] # Mwd/ton

# Print out statements with calculated numbers
print(f'Reactor U-235 Mass decreases from {u235[0]*rod_count/1000} kg, \
depleted after {time[-1]/30} months to {u235[-1]*rod_count/1000} kg.')
print(f'Over the course of those {time[-1]/30} months, the reactor produces {power_gen*rod_count/10**9} G
```

```

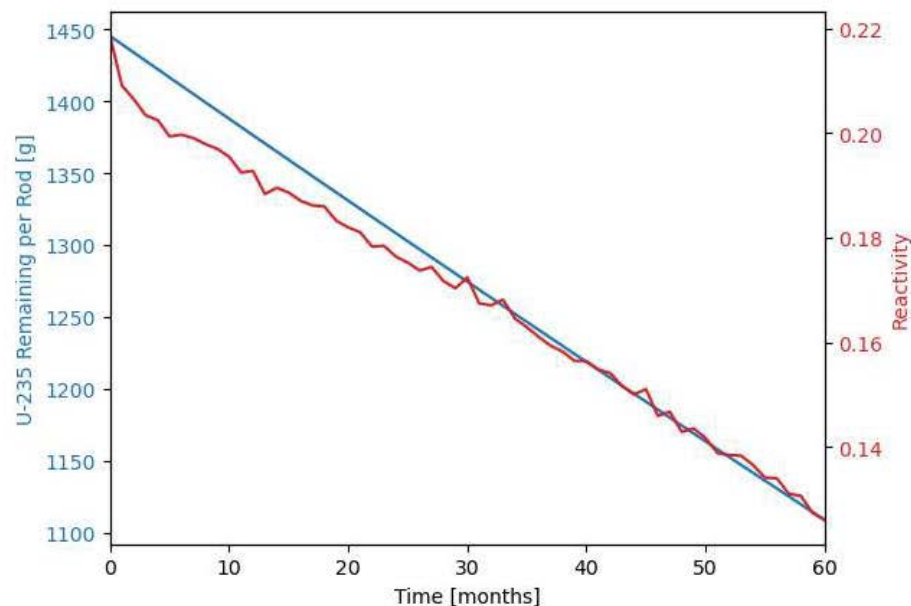
and burns {burned*rod_count/1000} kg of U-235, for a burnup of {burnup/1000} Gwd/MTU.')}

# Display the plot
plt.show()

```

Reactor U-235 Mass decreases from 1811.8348465921173 kg, depleted after 60.0 months to 776.1398278368866 kg.
Over the course of those 60.0 months, the reactor produces 179.99982 Gwd and burns 235.6942187560307 kg of U-235, for a burnup of 79.49325745375197 Gwd/MTU.

U-235 Mass Depletion per Channel



```
In [20]: (remaining_235[0] - remaining_235[-1])/remaining_235[0]
```

```
Out[20]: np.float64(0.23293762405984933)
```

```

In [21]: fig, (ax1, ax2, ax3, ax4) = plt.subplots(4,1, sharex = True)
fig.suptitle("Grams of Fission Product Built In Fuel Rod Over Time")

ax1.plot(time/30, cs137, label="Cs-137", color = "blue")
ax1.plot(time/30, sr90, label="Sr-90", color = "red")
# ax1.set_ylabel("grams of Fission Product");
ax1.legend(loc='upper left')

ax2.plot(time/30, i131, label="I-131", color = "olive")
ax2.plot(time/30, xe135, label="Xe-135", color = "purple")
ax2.legend(loc='center right')

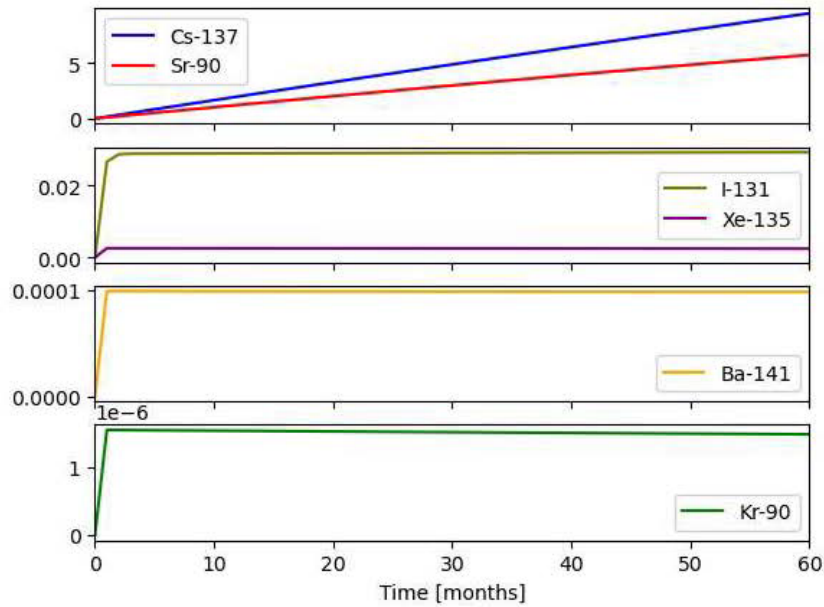
ax3.plot(time/30, ba141, label="Ba-141", color = "orange")
ax3.legend(loc='lower right')

ax4.plot(time/30, kr90, label="Kr-90", color = "green")
ax4.legend(loc='lower right')
ax4.set_xlabel("Time [months]")
ax4.set_xlim(0,time[-1]/30)

```

```
Out[21]: (0.0, 60.0)
```

Grams of Fission Product Built In Fuel Rod Over Time

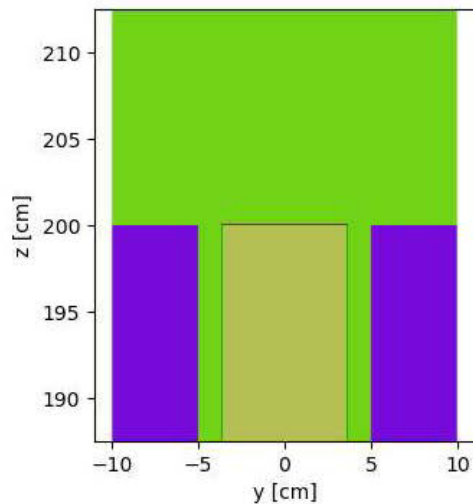


Conclusion

All graphs shows that the value of reactivity retains a value above 0.12 for the first 60 months. Multiplication factor remains above 1.13 during this time.

```
In [22]: root.plot(basis='yz', width=[r_g**2.2, half_z/8], color_by='material', origin=[0., 0., half_z], pixels=[3
```

```
Out[22]: <Axes: xlabel='y [cm]', ylabel='z [cm]')>
```



The above plot is generated for the purpose of the report to demonstrate the geometry at a cross sectional view.

```
In [23]: import pandas as pd
dft=pd.DataFrame(time)
print(dft)
```

```

0
0    0.0
1    30.0
2    60.0
3    90.0
4   120.0
..    ...
56  1680.0
57  1710.0
58  1740.0
59  1770.0
60  1800.0
```

[61 rows x 1 columns]

```
In [24]: dfr=pd.DataFrame(rho)
print(dfr)
```

```

0
0    0.218575
1    0.209075
2    0.206471
3    0.203375
4    0.202490
..    ...
56    0.133956
57    0.130997
58    0.130633
59    0.127336
60    0.126012
```

[61 rows x 1 columns]

```
In [25]: dfb=pd.DataFrame(time/((uranium[0]))*power/1000)
print(dfb)
```

```

0
0    0.000000
1    1.324888
2    2.649775
3    3.974663
4    5.299550
..    ...
56   74.193707
57   75.518595
58   76.843482
59   78.168370
60   79.493257
```

[61 rows x 1 columns]

```
In [26]: array=[]

for month in range(len(time)):
    temp_r = rho[month]
    temp_b = time[month]/((uranium[0]))*power/1000
    array.append([temp_r, temp_b])
```

```
# Specify column names:
df = pd.DataFrame(array, columns=['Reactivity', 'Burnup'])

# Print the DataFrame
print(df)
```

	Reactivity	Burnup
0	0.218575	0.000000
1	0.209075	1.324888
2	0.206471	2.649775
3	0.203375	3.974663
4	0.202490	5.299550
..	...	...
56	0.133956	74.193707
57	0.130997	75.518595
58	0.130633	76.843482
59	0.127336	78.168370
60	0.126012	79.493257

[61 rows x 2 columns]

```
In [27]: # Export to Excel file
df.to_clipboard(header=True, index=False)
```