Chapter 3

Fluoride-Salt-Cooled High-Temperature Reactor Benchmark

The FHR is a reactor concept that uses TRISO fuel and a low-pressure liquid fluoride-salt coolant. FHR technology combines FLiBe coolant from MSRs and TRISO particles from VHTRs to enable a reactor with low operating pressure, large thermal margin, and accident-tolerant qualities. The AHTR is a FHR type that has plate-based fuel in a hexagonal fuel assembly. To address the AHTR modeling challenges, such as multiple heterogeneity and material cross-section data, the OECD-NEA and Georgia Tech initiated the FHR benchmark for the AHTR design in 2019 [2]. In section 2.1, I gave an FHR concept overview, a AHTR design description, a review of previous efforts towards modeling these designs, and how these efforts led to the benchmark initiation.

The FHR benchmark has several phases, starting with a single fuel assembly simulation without burnup and gradually extending to full core depletion. Table 3.1 outlines the complete and incomplete benchmark phases.

Table 3.1: The Fluoride-Salt-Cooled High-Temperature Reactor benchmark's Phases [2].

Phases	**Sub-phases**	**Description**	**Completed?**
Phase I: fuel assembly (2D/3D with depletion)	I-A	2D model, steady-state (no depletion)	✓
	I-B	2D model depletion	✓
	I-C	3D model depletion	
Phase II: 3D full core with depletion	II-A	Steady-state (no depletion)	
	II-B	Depletion	
Phase III: 3D full core with feedback & multicycle analysis	III-A	Full core depletion with feedback	
	III-B	Multicycle analysis	

In the subsequent sections, I will describe the benchmark's specifications for the AHTR design and Phase I. Then, I will share our Phase I-A and I-B results.

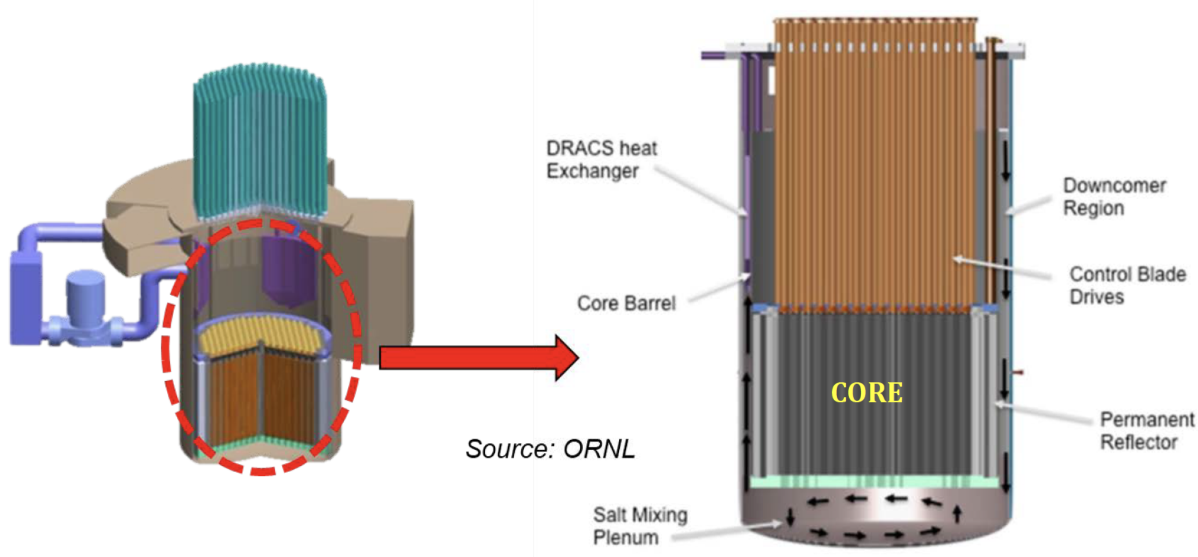


Figure 3.1: Advanced High Temperature Reactor schematic (left) and vessel (right) [2].

3.1 Benchmark Specifications: AHTR Design

The Advanced High Temperature Reactor has 3400 MWt thermal power and 1400 MW electric power [81]. Figure 3.1 shows the reactor schematic and a vertical cut of the reactor vessel. Figure 2.1 shows a single fuel assembly's geometry and all assemblies' configuration in the core. The AHTR's hexagonal fuel assembly detailed 2D view is shown in Figure 3.2. It features plate-type fuel with hexagonal fuel assembly consisting of eighteen planks arranged in three diamond-shaped sectors, with a central Y-shaped structure and external channel (wrapper). The diamond-shaped sections have 120-deg rotational symmetry with each other [81, 60, 2]. The fuel planks have semi-cylindrical spacers attached, their radius being equal to the coolant channel thickness.

Figure 3.3 shows the external channel wrapper and structural Y-shape, which are made of C-C composite and have extra notches to hold the fuel plates in place. The gap between the fuel assemblies and fuel plates is filled with FLiBe coolant. The Y-shaped control rod slot at the center of the Y-shape structure contains FLiBe coolant when the control blade is not in the slot (as seen in Figure 3.2) [81, 60, 2]. Each fuel plank is made of an isostatically

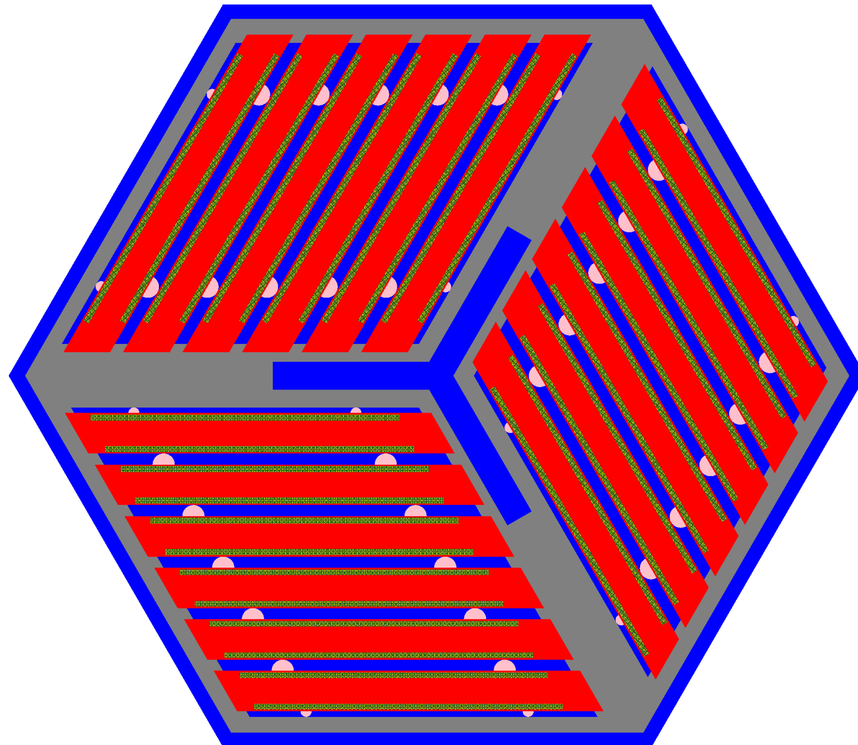


Figure 3.2: Advanced High Temperature Reactor fuel assembly with 18 fuel plates arranged in three diamond-shaped sectors, with a central Y-shaped and external channel graphite structure. Blue: FliBE coolant in between fuel assemblies and plates, and in the control rod slot, Gray: graphite structural components, Red: graphite fuel plank, Pink: graphite spacers, Green: graphite matrix with embedded TRISO particles.

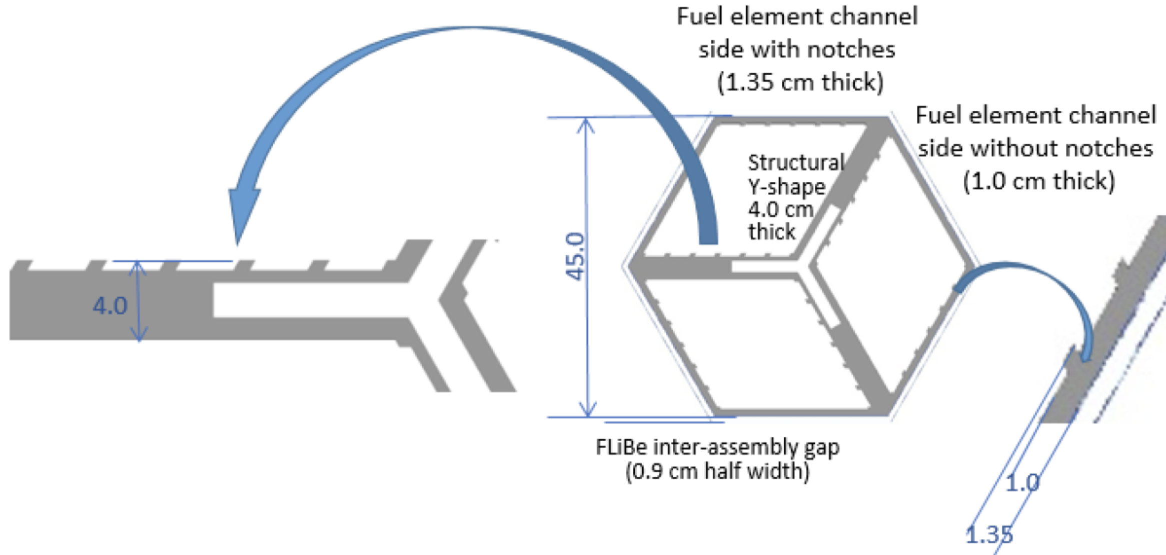


Figure 3.3: Advanced High Temperature Reactor fuel assembly's structural components [2].

pressed carbon with fuel stripes on each outer side of the plank, as seen in Figure 3.4. The fuel stripes are prismatic regions composed of a graphite matrix filled with a cubic lattice of TRISO particles with a 40% packing fraction. The lattice is 210 TRISO particles wide in the x-direction, four particles deep in the y-direction, and 5936 particles tall in the z-direction. Each TRISO particle has five layers (Figure 3.5): oxycarbide fuel kernel, porous carbon buffer, inner pyrolytic carbon, silicon carbide layer, and the outer pyrolitic carbon.

For reactivity control, burnable poisons and control rods are included in some configurations of the AHTR. The burnable poisons consist of europium oxide, Eu_2O_3, and have a discrete or integral (dispersed) option. In the discrete option, small spherical Eu_2O_3 particles are stacked axially at five locations in each fuel plank, as shown in Figure 3.6. In the integral option, Eu_2O_3 is homogenously mixed with the fuel plank graphite matrix (including the graphite in fuel stripes matrix and plank ends indented to structural sides, but excluding the graphite in spacers and graphite in TRISO particles). Each control rod is uniformly composed of molybdenumhafnium carbide alloy (MHC) and is inserted into the Y-shaped control rod slot where it displaces the FLiBe that occupies the slot (Figure 3.2).

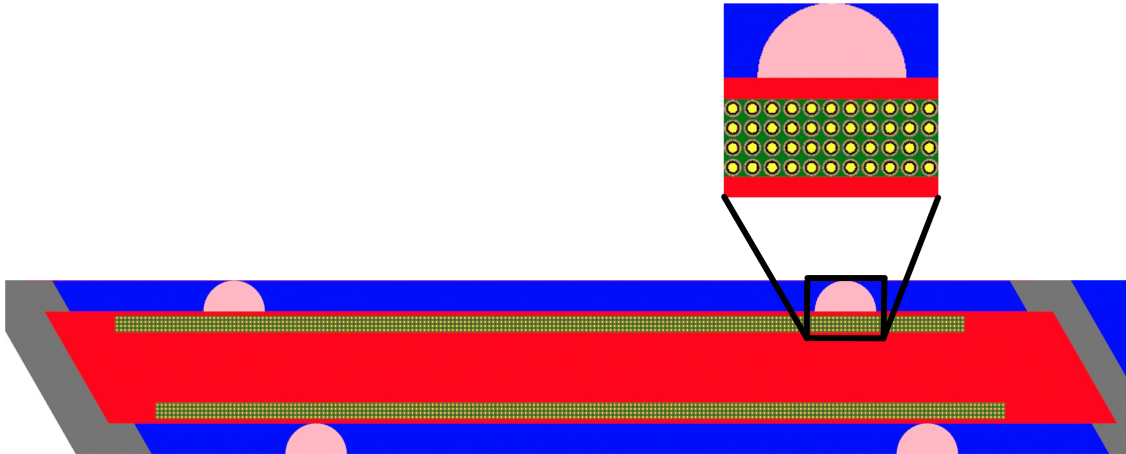


Figure 3.4: Advanced High Temperature Reactor's fuel plank, with the magnification of a spacer and segment of the fuel stripe with embedded TRISO particles.

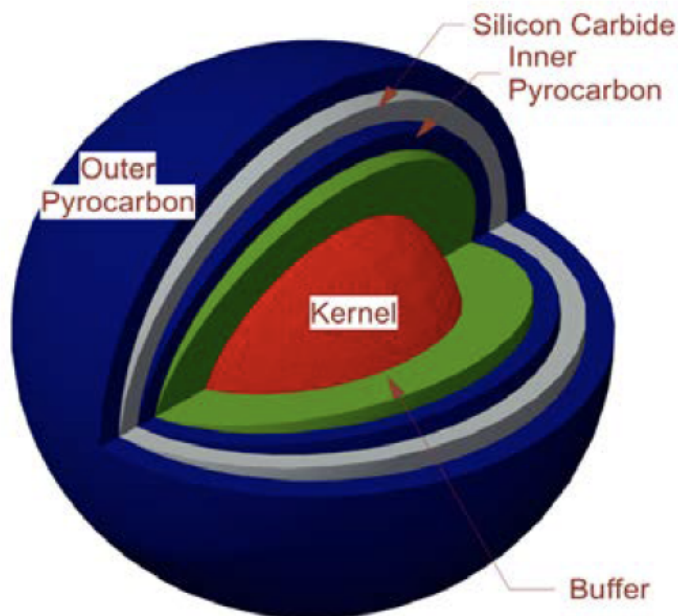


Figure 3.5: Advanced High Temperature Reactor's TRISO particle schematic [2].

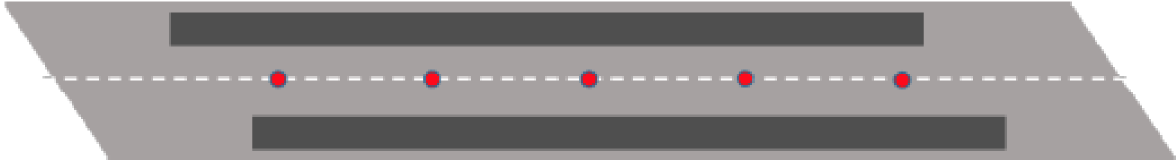


Figure 3.6: Placement of axial stacks of burnable poisons in the Advanced High Temperature Reactor [2].

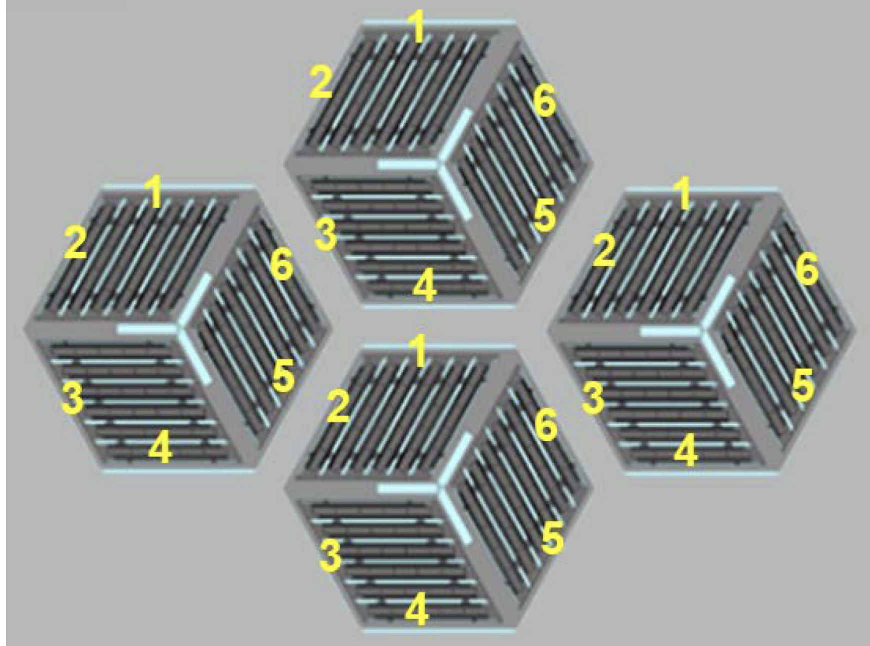


Figure 3.7: Visualization of periodic boundary conditions for a single fuel assembly in the AHTR[2].

3.2 Benchmark Specifications: Phase I

The FHR benchmark's Phase I consists of a steady-state 2D model (Phase I-A) and depletion (Phase I-B) of one FHR fuel assembly. For a single fuel assembly, the internal 120-degree rotational symmetry is represented by periodic boundary conditions, as seen in Figure 3.7. The benchmark required the following results for Phases I-A and I-B:

(a) effective multiplication factor

(b) reactivity coefficients (β_{eff}, fuel Doppler coefficient, FLiBe temperature coefficient, graphite temperature coefficient)

(c) tabulated fission source distribution by one-fifth fuel stripe

(d) neutron flux averaged over the whole model tabulated in three coarse energy groups

(e) neutron flux distribution in three coarse energy groups

(f) fuel assembly averaged neutron spectrum

Next, I report the equations used to calculate these required results.

Reactivity Coefficients (b)

Effective delayed neutron fraction (β_{eff}) is the fraction of delayed neutrons in the core. I assumed one energy group and six delayed groups for β_{eff}. Reactivity coefficient is the change in reactivity (ρ) of the material per degree change in the material's temperature (T). I calculated each reactivity coefficient and its corresponding uncertainty with these equations:

$$\frac{\Delta\rho}{\Delta T} = \frac{\rho_{T_{high}} - \rho_{T_{low}}}{T_{high} - T_{low}} \; [\frac{pcm}{K}] \tag{3.1}$$

$$\delta\frac{\Delta\rho}{\Delta T} = \frac{\sqrt{\delta(\rho_{T_{high}})^2 + (\delta\rho_{T_{low}})^2}}{T_{high} - T_{low}} \; [\frac{pcm}{K}] \tag{3.2}$$

Fission Source Distribution / Fission Density (c)

I calculated fission density (FD) with OpenMC's `fission` tally score (f) for each region divided by the average `fission` tally score of all the regions:

$$FD_i = \frac{f_i}{f_{ave}} \tag{3.3}$$

where

f_i = fission reaction rate in a single region [reactions/src]

f_{ave} = average of all f_i [reactions/src]

The uncertainty calculations for FD_i and f_{ave}:

$$\delta FD_i = |FD_i| \sqrt{(\frac{\delta f_i}{f_i})^2 + (\frac{\delta f_{ave}}{f_{ave}})^2} \tag{3.4}$$

$$\delta f_{ave} = \frac{1}{N} \sqrt{\sum_i^N f_i^2} \tag{3.5}$$

where

$$N = \text{No. of fission score values}$$

Neutron Flux (d, e, f)

OpenMC's `flux` score is in $[\frac{neutrons\ cm}{src}]$ units. For the benchmark, I converted flux to $[\frac{neutrons}{cm^2 s}]$ units using the following equations:

$$\Phi_c = \frac{N \times \Phi_o}{V} \tag{3.6}$$

$$N = \frac{P \times \nu}{Q \times k} \tag{3.7}$$

where

Φ_c = converted flux $[\frac{neutrons}{cm^2 s}]$

Φ_o = original flux $[\frac{neutrons\ cm}{src}]$

N = normalization factor $[\frac{src}{s}]$

V = volume of fuel assembly $[cm^3]$

P = power $[\frac{J}{s}]$

$\nu = \frac{\nu_f}{f}$ $[\frac{neutrons}{fission}]$

Q = Energy produced per fission $[\frac{J}{fission}] = 3.2044 \times 10^{-11}$ J per U_{235} fission

$k = k_{eff}$ $[\frac{neutrons}{src}]$

The flux standard deviation is:

$$\delta\Phi_c = \Phi_c \times \sqrt{(\frac{\delta\Phi_o}{\Phi_o})^2 + (\frac{\delta\nu_f}{\nu_f})^2 + (\frac{\delta k}{k})^2 + (\frac{\delta f}{f})^2} \tag{3.8}$$

I calculated reactor power based on the given reference specific power (P_{sp}) of 200 $\frac{W}{gU}$:

$$P = P_{sp} \times V_F \times \rho_F \times \frac{wt\%_U}{100} \tag{3.9}$$

where

$$V_F = \text{volume of fuel } [cm^3] = \frac{4}{3}\pi r_f^3 \times N_{total}$$

$$r_f = \text{radius of fuel kernel}$$

$$N_{total} = \text{total no. of TRISO particles in fuel assembly} = 101 \times 210 \times 4 \times 2 \times 6 \times 3$$

$$\rho_F = \text{density of fuel } [g/cc]$$

$$wt\%_U = \frac{at\%_{U235} \times AM_{U235} + at\%_{U238} \times AM_{U238}}{\sum(at\%_i \times AM_i)} \times 100$$

$$AM = \text{atomic mass}$$

3.2.1 Benchmark Specifications: Phase I-A

For Phase I-A, the benchmark specifies that each participant must produce a steady-state 2D model of one fresh fuel assembly for nine cases and report the required results listed in Section 3.2. Table 3.2 describes each case.

Table 3.2: Description of the Fluoride-Salt-Cooled High-Temperature Reactor benchmark Phase I-A cases [2].

Case	**Description**
1A	Reference case. Hot full power (HFP), with temperatures of 1110K for fuel kernel and 948K for coolant and all other materials (including TRISO particle layers other than fuel kernel). Nominal (cold) dimensions, 9 wt% enrichment, no Burnable Poison (BP), control rods (CRs) out.
2AH	Hot zero power (HZP) with uniform temperature of 948 K, otherwise same as Case 1A. Comparison with Case 1A provides HZP-to-HFP power defect.
2AC	Cold zero power (CZP). Same as Case 2AH, but with uniform temperature of 773 K. Comparison with Case 2AH provides isothermal temperature coefficient.
3A	CR inserted, otherwise same as Case 1A.
4A	Discrete europia BP, otherwise same as Case 1A.
4AR	Discrete europia BP and CR inserted, otherwise same as Case 1A.
5A	Integral (dispersed) europia BP, otherwise same as Case 1A.
6A	Increased heavy metal (HM) loading (4 to 8 layers of TRISO) decreased C/HM ratio (from about 400 to about 200) and decreased specific power to 100 W/gU, otherwise same as Case 1A.
7A	Fuel enrichment 19.75 wt%, otherwise same as Case 1A.

3.2.2 Benchmark Specifications: Phase I-B

For Phase I-B, the benchmark specifies that each participant must produce depletion results for three cases: 1B, 4B, and 7B. These are the same as cases 1A, 4A, and 7A, but with depletion steps added. The benchmark assumes that depletion occurs only in the fuel and BPs and that the depletion performs under the critical spectrum assumption.

3.3 Results

Several organizations participated in the benchmark with various Monte Carlo and Deterministic neutronics codes, such as Serpent [41], OpenMC [62], and WIMS [42]. University of Illinois at Urbana-Champaign (UIUC) participated in the benchmark with the OpenMC Monte Carlo code [62] and the ENDF/B-VII.1 material library [10]. The `fhr-benchmark` Github repository contains all the results submitted by UIUC for the FHR benchmark [11]. The benchmark used a phased blind approach – participants were asked to submit Phase I-A and I-B results without knowledge of other submissions. Petrovic et al. [56] describes the preliminary results of the benchmark results across several institutions and concludes that the overall observed agreement is satisfactory. In the subsequent sections, I will share the results obtained by UIUC.

3.3.1 Results: Phase I-A

Petrovic et al. [56] compared the effective multiplication factor (k_{eff}) for all participants and Phase I-A cases in the FHR benchmark. They reported that the standard deviation between participants for each case was in the 231 to 514 pcm range, acceptable and notably close given a blind benchmark, assuring us that our Phase I-A results are acceptable and in agreement with other benchmark participants.

Table 3.2 reports Phase I-A k_{eff} and reactivity coefficients results. I ran the simulations on UIUC's BlueWaters supercomputer with 64 XE nodes, which each have 32 cores [48]. To

reduce k_{eff}'s statistical uncertainty to ~10pcm, I ran each simulation with 500 active cycles, 100 inactive cycles, and 200000 neutrons. Each simulation took wall-clock-time (WCT) ranging from 2 to 5 hours.

Table 3.3: University of Illinois at Urbana-Champaign's Fluoride-Salt-Cooled High-Temperature Reactor Benchmark Phase I-A results [11].

Case	**Summary**	**WCT [hr]**	k_{eff}*	β_{eff}**	**Fuel** $\frac{\Delta\rho}{\Delta T}$	**FliBe** $\frac{\Delta\rho}{\Delta T}$	**Graphite** $\frac{\Delta\rho}{\Delta T}$
1A	Reference	2.82	1.39389	0.006534	-2.24±0.15	-0.15±0.15	-0.68±0.15
2AH	HZP	2.82	1.40395	0.006534	-3.14±0.15	-0.20±0.14	-0.85±0.14
2AC	CZP	2.75	1.41891	0.006534	-3.36±0.14	-0.11±0.14	0.07±0.14
3A	CR	2.49	1.03147	0.006534	-4.03±0.28	-0.83±0.27	-3.18±0.29
4A	Discrete BP	5.08	1.09766	0.006542	-4.06±0.24	-1.55±0.23	-6.51±0.24
4AR	Discrete BP + CR	4.59	0.84158	0.006553	-5.60±0.49	-1.78±0.46	-10.44±0.47
5A	Dispersed BP	2.33	0.79837	0.006556	-5.09±0.40	-4.87±0.40	-22.99±0.38
6A	Increased HM	3.52	1.26294	0.006556	-4.46±0.19	0.16±0.20	-0.39±0.20
7A	19.75% Enriched	2.21	1.50526	0.006530	-2.49±0.13	-0.12±0.12	-0.62±0.12

* All k_{eff} values have an uncertainty of 0.00010.

** All β_{eff} values have an uncertainty of 0.000001.

Cases 2AH and 2AC are at zero power, meaning that the fuel assembly is exactly critical but not producing any energy. For both cases, k_{eff} is higher than the reference Case 1A, which I attribute to lower fuel temperatures. At lower fuel temperatures, less doppler broadening occurs, resulting in less neutron capture, thus, increasing k_{eff}. As expected, k_{eff} is lower for Cases 3A, 4AR, and 5A than reference case 1A since these cases introduce burnable poisons and control rods to the fuel assembly. Also, as expected, k_{eff} is higher for Case 7A than reference Case 1A, since it has a higher enrichment. However, Case 6A deviated from expectations with a lower k_{eff} despite an increase in heavy metal loading. This behavior is due to reduced moderation and worsened fuel utilization brought about by self-shielding, demonstrating that an increase in fuel packing fraction does not always correspond with an increased k_{eff}.

β_{eff} increased by 10-20pcm for Cases 4A, 4AR, 5A, and 6A compared to reference Case 1A due to the introduction of control rods and poisons that shift the average neutron velocity

to higher values, resulting in decreased thermal fission and increased fast fission [78]. Table 3.3 reports that most of the temperature coefficients are negative, exemplifying the AHTR's passive safety behavior. Negative reactivity feedback results in a self-regulating reactor; if the reactor's power rises, resulting in temperature increase, the negative reactivity reduces power.

Figure 3.8 shows the fission source distribution by one-fifth fuel stripe for Cases 1A and 3A. Case 4AR has a similar fission source distribution as Case 3A since both cases have control rod insertion. All other cases have similar fission source distribution shape to Case 1A. For Case 1A, intuitively, I would assume that the highest fission source would occur in the center of the diamond fuel segment; however, the opposite is true. Power peaking occurs on exterior stripes and is minimum on the interior stripes. Gentry et al. [25] reported similar power peaking phenomena towards the lattice cell's exterior closest to the Y-shaped carbon support structure where the thermal flux is most elevated. The lowest power is found in the interiors of the lattice tri-sections. This fission source distribution is caused by diminished resonance escape probability in the interior due to the higher relative fuel-to-carbon volume ratio. For Case 3A with an inserted control rod, the fission source is lower in the one-fifth stripes closer to the control rod. Cases 6A and 7A demonstrate a further diminished fission source in the interior stripes due to the higher fuel-to-carbon ratio. This is seen in Figure 3.8 in which case 1A and 6A have similar fission distribution shapes, but case 6A's has a bigger fission source value range.

Figure 3.9 shows the average neutron flux in the fuel assembly in three coarse energy groups. Most of the cases have the most flux in the intermediate group, followed by the thermal group, and the least flux in the fast group. Figure 3.10 shows the neutron flux distribution in a 100×100 mesh for Cases 1A, 3A, and 6A for three coarse energy groups. For all three cases, fast-flux peaks in the diamond-shaped sectors containing the fuel stripes, whereas thermal flux peaks outside of the diamond-shaped sectors. This is attributed to fission occurring at thermal energies in the fuel stripe area. For Case 3A, the thermal and

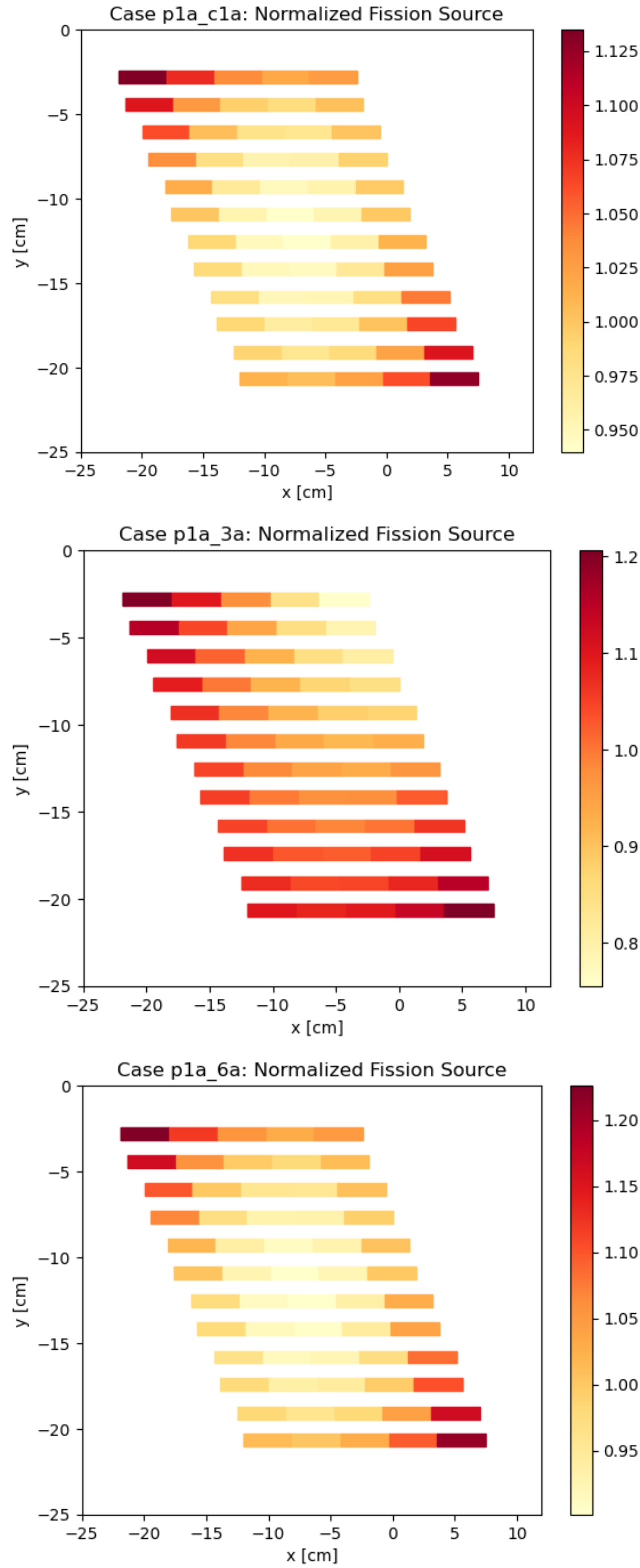


Figure 3.8: Fission Source Distribution per one-fifth fuel stripe for Fluoride-Salt-Cooled High-Temperature Reactor Benchmark's Phase I-A Case 1A (top), Case 3A (middle), and Case 6A (bottom).

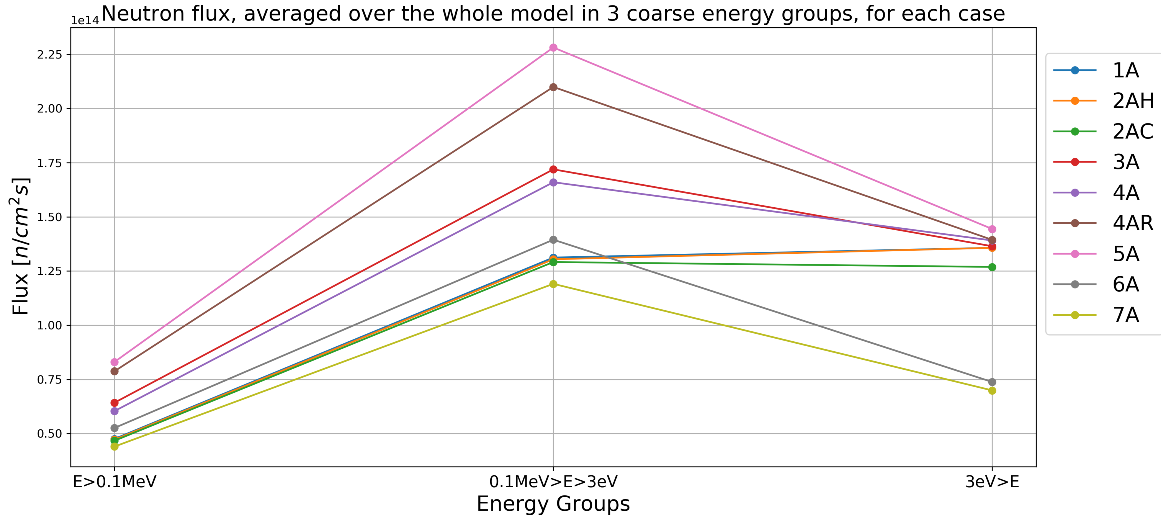


Figure 3.9: Fluoride-Salt-Cooled High-Temperature Reactor Benchmark's Neutron flux, averaged over the whole model, tabulated in three coarse energy groups for each Phase I-A case.

intermediate neutron flux is depressed in the fuel assembly's control rod region. Case 6A has an increased heavy metal loading, resulting in a more pronounced fast-flux peaking and thermal flux dip in the fuel stripe area.

Figure 3.11 shows the neutron spectrum for Cases 1A and 6A. Case 7A has a similar neutron spectrum as Case 6A since both cases have higher fuel content. All other cases have a similar neutron spectrum to Case 1A. The neutron spectrum is faster for Cases 6A and 7A due to more heavy metal loading and higher enrichment, respectively.

3.3.2 Results: Phase I-B

Figure 3.12 shows the k_{eff} evolution during depletion for Cases 1B, 4B, and 7B. The k_{eff} at zero burnup corresponds to each case's corresponding Phase I-A k_{eff} value reported in Table 3.3. Case 1B is the reference case with 9% fuel enrichment and no BPs. Case 1B's k_{eff} steadily decreases until it reaches 0.967845 at the final 70 GWd/tU burnup. Case 4B includes burnable poisons resulting in a lower initial k_{eff}. Its k_{eff} decreases at a slower rate

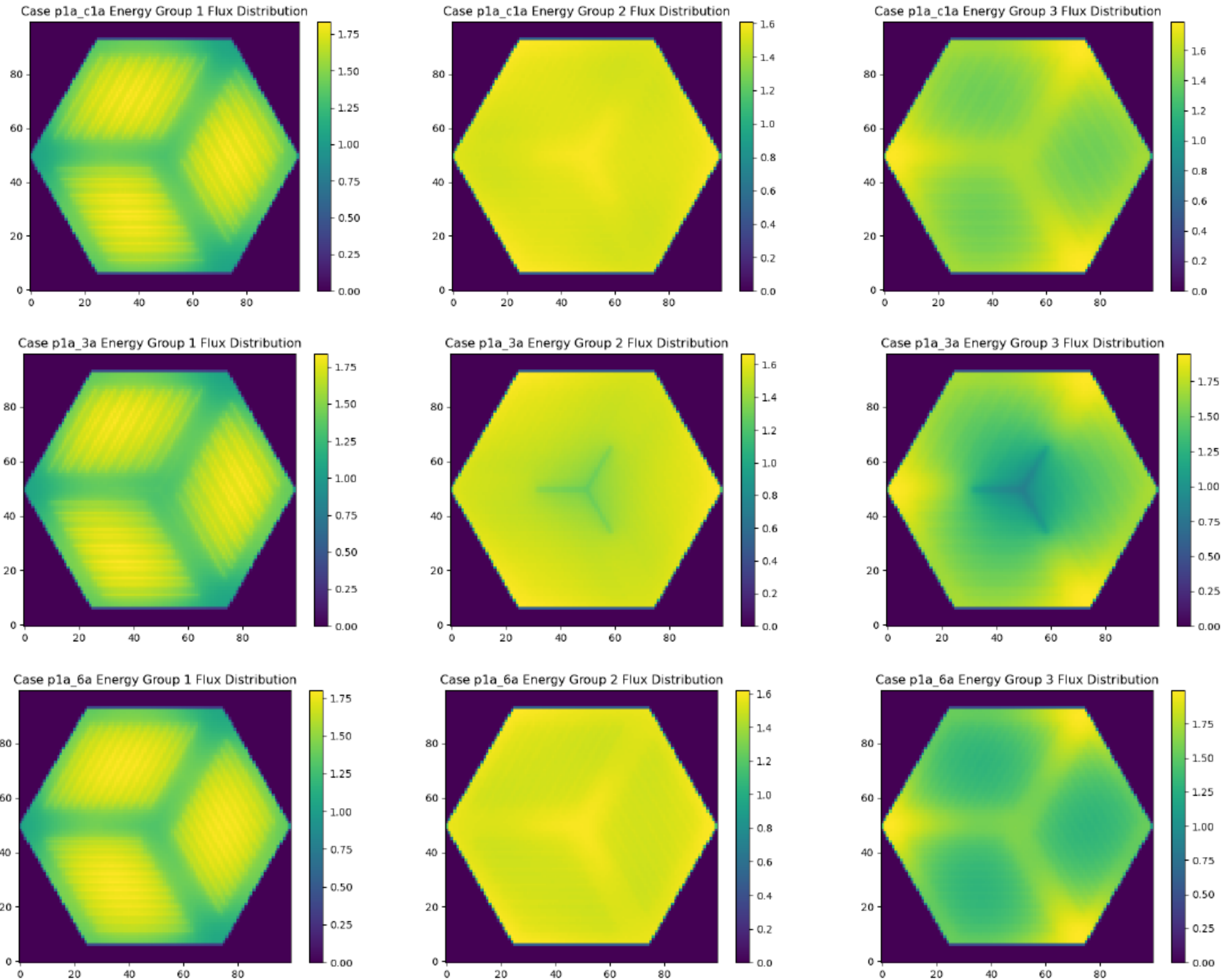
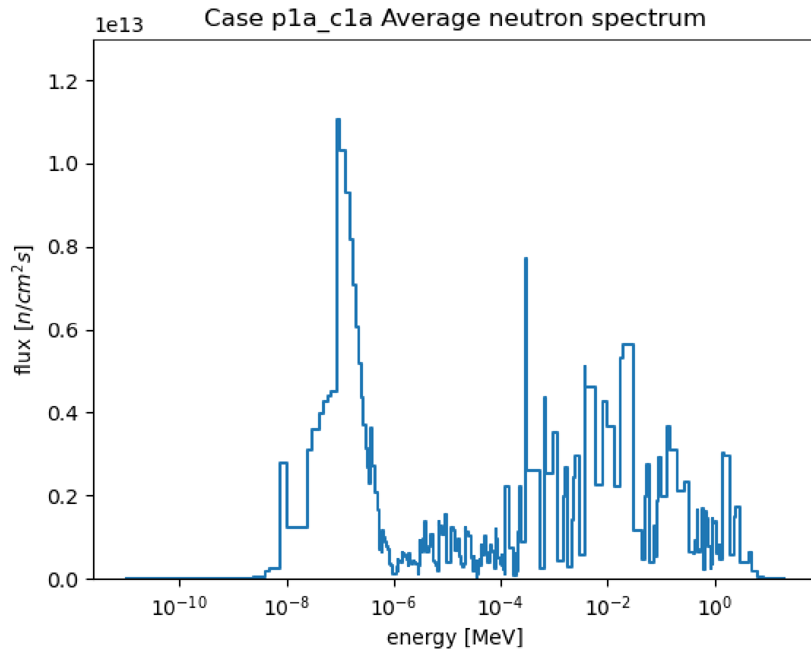


Figure 3.10: Fluoride-Salt-Cooled High-Temperature Reactor Benchmark's Neutron flux distribution in 100×100 mesh for three coarse energy groups: Case 1A (above), Case 3A (middle), Case 6A (below). Energy group 1: $E > 0.1$ MeV, Energy group 2: $3 \times 10^{-6} < E < 0.1$ MeV, Energy group 3: $E < 3 \times 10^{-6}$ MeV.



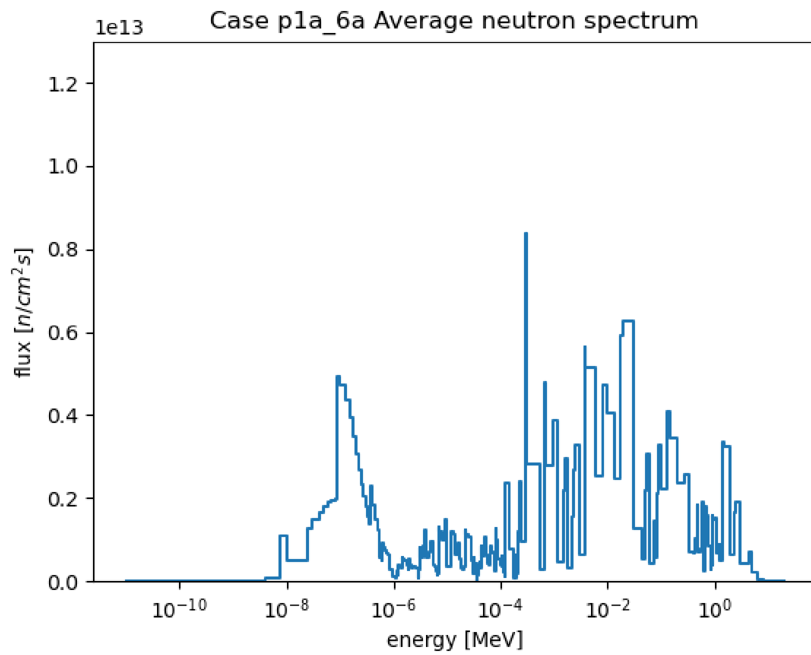


Figure 3.11: Neutron spectrum for Fluoride-Salt-Cooled High-Temperature Reactor Benchmark's Phase I-A Case 1A (left) and Case 6A (right).

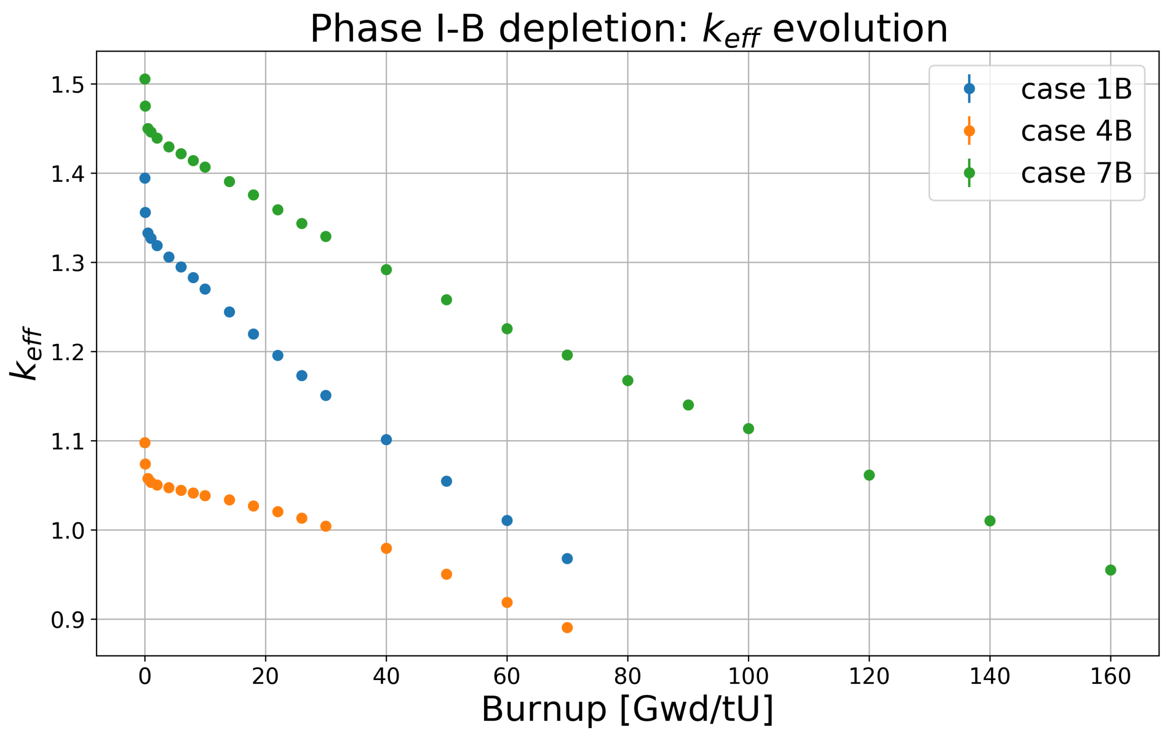


Figure 3.12: Fluoride-Salt-Cooled High-Temperature Reactor Benchmark's Phase I-B depletion k_{eff} evolution for Cases 1B, 4B, and 7B. Case 1B is the reference case, Case 4B is the discrete BP case, and Case 7B is the 19.75% enrichment case. Error bars are included but are barely visible due to the low uncertainty of ~40pcm.

in the beginning due to the presence of burnable poisons, which decreases flux in the core. At approximately 20 GWd/tU, k_{eff} begins decreasing at a faster rate, assumedly due to burn-up of the poison material. Case 7B has a 19.75% fuel enrichment, resulting in a higher initial k_{eff}. With a higher enrichment, the fuel can achieve a final burnup of 160 GWd/tU.

3.4 Summary

This chapter described the FHR benchmark specifications, AHTR design, and Phase I-A and I-B results obtained by the UIUC team. The benchmark results highlight the AHTR's passive safety behavior with negative temperature coefficients. Results such as a lower k_{eff} for the AHTR configuration with higher heavy metal loading demonstrated how increased fuel packing does not always correspond with increased k_{eff} due to self-shielding effects. These results hint at the possibility of minimizing fuel required by optimizing for inhomogeneous fuel distributions within the core. This will be further explored in the later chapters.

Chapter 4

REALM: Reactor Evolutionary Algorithm Optimizer

In this chapter, I introduce the Reactor Evolutionary Algorithm Optimizer (REALM) framework developed for the proposed work. REALM is a Python package that applies evolutionary algorithm optimization techniques to nuclear reactor design. Applying evolutionary algorithms to nuclear design problems is not new, as I previously discussed in Section 2.3, and available evolutionary algorithm packages can be customized for reactor design optimization problems. However, evolutionary algorithm setup is highly customizable with an assortment of genetic algorithm designs and operators. A reactor designer unfamiliar with evolutionary algorithms will have to go through the cumbersome process of customizing a genetic algorithm for their needs and determine which operators and hyperparameters work best for their problem. Furthermore, computing fitness values with nuclear software is computationally expensive, necessitating using supercomputers and setting up parallelization for the genetic algorithm.

Therefore, the motivation behind creating REALM is to limit these inconveniences and facilitate using evolutionary algorithms for reactor design optimization. REALM provides a general genetic algorithm framework, sets up parallelization for the user, and promotes usability with an input file that only exposes mandatory parameters. REALM also strives to be effective, flexible, open-source, parallel, reproducible, and usable. I briefly summarize how REALM achieves these goals:

- Effective: REALM is well documented, well tested, and version-controlled on Github [12].

- Flexible: The proposed work aims to utilize REALM to explore arbitrary reactor geometries and inhomogeneous fuel distributions. However, future users might want to utilize REALM with other arbitrary parameters. Thus, I designed the REALM framework accordingly. The user can vary any imaginable parameter because REALM uses a templating method to edit the coupled software's input file.

- Open-source: I utilize a well-documented, open-source evolutionary algorithm Python package to drive the optimization process, and established open-source nuclear software, OpenMC [62] and Moltres [43], to compute the objective function and constraints. I also provide a simple tutorial for future developers to follow for coupling other nuclear software to REALM.

- Parallel: REALM runs parallel on HPC machines using the `mpi4py` Python package [14].

- Reproducible: Data from every REALM run saves into a unique, pickled file (pickle is a Python module that serializes Python objects), and all results from this work are available on Github.

- Usable: I did not reinvent the wheel–instead, I combined available evolutionary algorithm, nuclear transport, and thermal-hydraulics software to create an optimization tool for easy leveraging of evolutionary algorithms to construct arbitrary reactor designs.

REALM essentially couples an evolutionary algorithm driver with nuclear software, such as nuclear transport and thermal-hydraulics codes. Figure 2.3 from Chapter 2 outlines an evolutionary algorithm's iterative problem solving process. I modified Figure 2.3 to produce Figure 4.1, which depicts how the nuclear transport and thermal-hydraulics software fit within the process. Therefore, REALM initially reads and validates the JSON input file, initializes the Distributed Evolutionary Algorithms in Python (DEAP) genetic algorithm

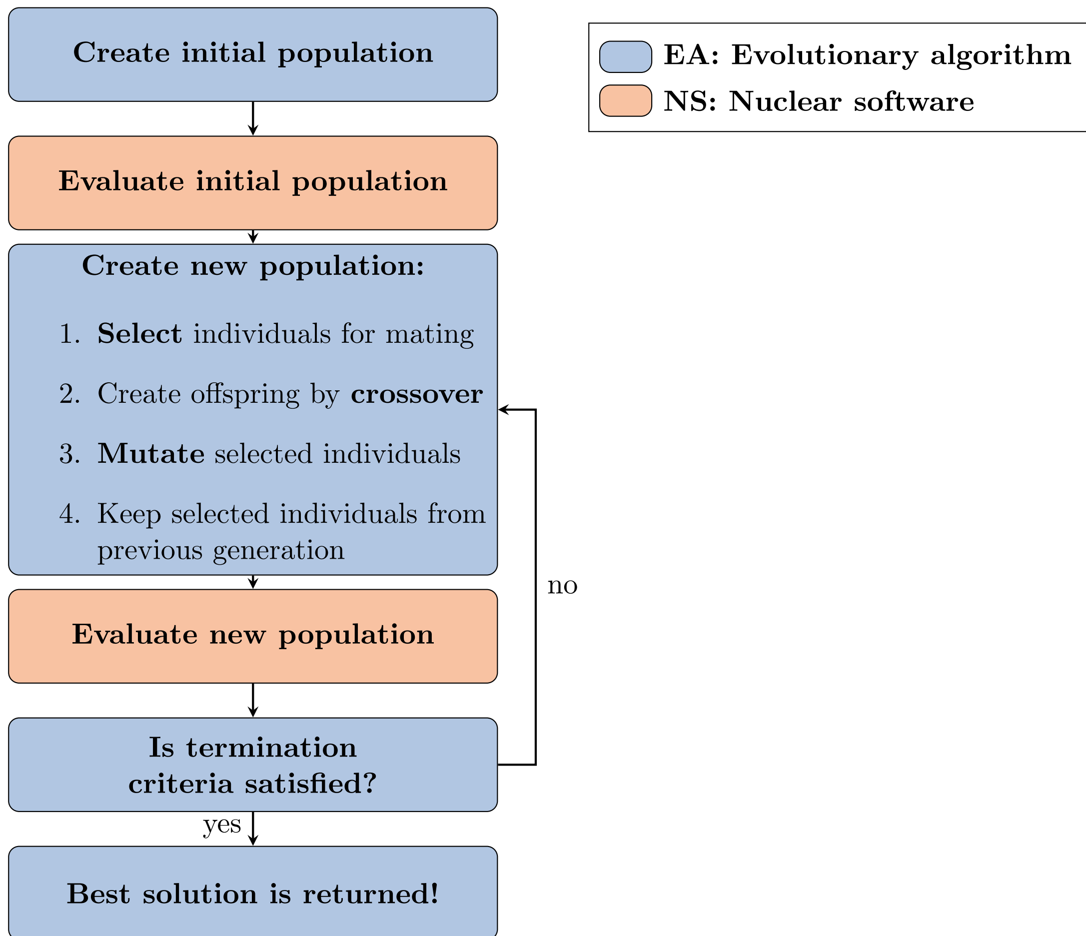


Figure 4.1: Process of finding optimal solutions for a problem with a genetic algorithm. Nuclear software evaluates each new population.

hyperparameters and operators, and finally runs the genetic algorithm following the flow chart in Figure 4.1, in which the nuclear software evaluates each individual's fitness.

In the subsequent sections, I describe the evolutionary algorithm software that drives REALM, the nuclear software coupled to REALM, and details about the REALM framework, such as the input file format and software architecture.

4.1 Evolutionary Algorithm Driver

Evolutionary algorithm computation is a sophisticated field with diverse techniques and mechanisms, resulting in even the most well-designed, coded-up frameworks being complicated under the hood. Utilizing an existing evolutionary algorithm framework brings up issues in extending implementation intricacies as the user has to edit the source code [22]. Therefore, an evolutionary algorithm computation framework that gives the user the capability to build custom evolutionary algorithms is ideal for this project.

There are many evolutionary algorithm computation packages available: DEAP [22], inspyred [24], Pyevolve [54], and OpenBEAGLE [23]. DEAP is the newest package and places a high value on code compactness and clarity [22]. DEAP is the only framework that allows the user to prototype evolutionary algorithms rapidly and define custom algorithms without digging deep into the source code to modify lines. This makes DEAP the code of choice for the REALM framework's evolutionary algorithm driver component. DEAP provides building blocks for each optimizer function and allows the user to customize a specialized algorithm to fit their project [22].

4.1.1 Distributed Evolutionary Algorithms in Python

DEAP is composed of two simples structures: a *creator* and a *toolbox*. The *creator* module is a meta-factory that allows the run-time creation of classes via inheritance and composition, enabling individual and population creation from from any data structure: lists, sets,

```
1    from deap import creator, base, tools, algorithms
2    creator.create("Objective", base.Fitness, weights=(-1.0,)) # minimum
3    creator.create("Individual", list, fitness=creator.Objective)
4
5    toolbox = base.Toolbox()
6    toolbox.register("variable_1", random.uniform, 0.0, 10.0)
7    toolbox.register("variable_2", random.uniform, -1.0, 0.0)
8    def individual_creator():
9        return creator.Individual([toolbox.variable_1(), toolbox.variable_2()])
10   toolbox.register("individual", individual_creator())
11   toolbox.register("population", tools.initRepeat, list, toolbox.individual)
12   def evaluator_fn(individual):
13       return tuple([sum(individual)])
14   toolbox.register("evaluate", evaluator_fn)
15   toolbox.register("select", tools.selBest, k=5)
16   toolbox.register("mutate", tools.mutPolynomialBounded, eta=0.5, low=[0, -1], up=[-1, 0])
17   toolbox.register("mate", tools.cxOnePoint)
```

Figure 4.2: DEAP sample code demonstrating the usage of the *creator* and *toolbox* modules to initialize the genetic algorithm. In REALM, DEAP's *creator* and *toolbox* modules are initialized in the source code based on the genetic algorithm parameters defined by the user in the REALM input file.

dictionaries, trees, etc [22]. The *toolbox* is a container that the user manually populates. In the *toolbox*, the user defines the selection, crossover, and mutation operator types and hyperparameters. For example, the user registers a crossover operator under the 'mate' alias, and a selection operator under the 'select' alias. Then, the evolutionary algorithm uses these aliased operators from the *toolbox*. If the user wants to change the crossover operator, they would update the 'mate' alias in the *toolbox*, while keeping the evolutionary algorithm unchanged [22].

Figure 4.2 illustrates DEAP's usage of the *creator* and *toolbox* modules. Line 2 creates a single-objective fitness class, `Objective`. The first argument defines the name of the derived class, the second argument specifies the inherited base class, `base.fitness`, and the third argument indicates the objective fitness (−1.0 indicates a minimum objective, +1.0 indicates a maximum objective). Line 3 derives an `Individual` class from the standard Python list

type, and defines its fitness attribute to be the newly created `Objective` object. Lines 5-9 initialize the DEAP toolbox, register `variable_1` and `variable_2` with their upper and lower bounds, and define the `individual_creator` function to return an `Individual` initialized with `variable_1`, and `variable_2`. Lines 10-11 and 14-17 are aliases for initializing individuals and population, specifying variation operators (`select`, `mutate`, `mate`), and evaluating individual fitness (`evaluate`) [22]. Lines 12-13 define the evaluation function that returns the fitness values.

In REALM, DEAP's *creator* and *toolbox* modules are initialized in the source code based on the genetic algorithm parameters defined by the user in the REALM input file. The evaluation function runs the nuclear software and returns user-defined fitness values.

4.1.2 General Genetic Algorithm Framework

The creators' of DEAP provided variations of a classical genetic algorithm exposing different explicitness levels [22]. The high-level examples use the in-built DEAP genetic algorithms, whereas the low-level example completely unpacks the genetic algorithm to expose a generational loop. The general genetic algorithm included in the ***Algorithm*** class is based on the low-level example. The algorithm begins by initializing the starting population and evaluating each individual's fitness value. Then, it enters a generational loop. During each iteration, selection, mating, and mutation operators are applied to the population, then, the new individuals are evaluated, the constraints are applied, and the results are saved.

4.2 Nuclear Software

Many nuclear software have restricted public access. In the proposed work, I enabled REALM to work with open-source nuclear transport and thermal-hydraulics software, OpenMC [62] and Moltres [43]. OpenMC is an open-source Monte Carlo neutron transport code capable of performing k-eigenvalue calculations on models built using either constructive solid

geometry or CAD representation. OpenMC can run in parallel using a hybrid Message Passing Interface (MPI) and OpenMP programming model. Moltres is an open-source tool designed to simulate MSRs using deterministic neutronics and thermal-hydraulics implemented as an application atop the Multiphysics Object-Oriented Simulation Environment (MOOSE) finite-element framework. Moltres solves arbitrary-group neutron diffusion, temperature, and precursor governing equations on a single mesh and can be deployed on an arbitrary number of processing units [43].

OpenMC and Moltres are both open-source, well-documented, well-supported, and Github version-controlled codes that can run in parallel on HPC machines. Thus they achieve the REALM goals listed at the start of this chapter, making them suitable to be used as REALM's nuclear software. However, users can easily use restricted nuclear software with REALM by coupling REALM with the restricted software on their local machine. In the REALM documentation [12], I outline how to couple other nuclear software to REALM.

4.3 REALM Input File

REALM's input file is in JSON format. There are four sections that the user must define: `control_variables`, `evaluators`, `constraints`, and `algorithm`. Figure 4.3 shows an example REALM input file. In this simulation, REALM uses a genetic algorithm with the defined hyperparameters to minimize the `output1` parameter which is calculated using the OpenMC evaluator that accepts input parameters: `variable1` and `variable2`.

Next, I will describe how to define each section of a REALM input file. The REALM documentation [12] provides further descriptions for setting up a REALM input file.

4.3.1 Control Variables

Control variables are parameters the genetic algorithm will vary. For each control variable, the user must specify its minimum and maximum values. For example, Lines 2 to 5 in Figure

```
{
    "control_variables": {
        "variable1": {"min": 0.0, "max": 10.0},
        "variable2": {"min": -1.0, "max": 0.0}
    },
    "evaluators": {
        "openmc": {
            "input_script": "openmc_inp.py",
            "output_script": "openmc_output.py",
            "inputs": ["variable1", "variable2"],
            "outputs": ["output1", "output2"]
        }
    },
    "constraints": {
        "output1": {"operator": [">=", "<"], "constrained_val": [1.0, 1.5]}
    },
    "algorithm": {
        "objective": "min",
        "optimized_variable": "output1",
        "pop_size": 100,
        "generations": 10,
        "mutation_probability": 0.5,
        "mating_probability": 0.5,
        "selection_operator": {"operator": "selBest", "k": 1},
        "mutation_operator": {
            "operator": "mutPolynomialBounded",
            "indpb": 0.5,
            "eta": 0.5
        },
        "mating_operator": {"operator": "cxOnePoint"}
    }
}
```

Figure 4.3: Reactor Evolutionary Algorithm Optimizer (REALM) sample JSON input file.

```python
import openmc
# templating
variable1 = {{variable1}}
variable2 = {{variable2}}
# run openmc
...
```

```python
import openmc
# templating
variable1 = 3.212
variable2 = -0.765
# run openmc
...
```

Figure 4.4: `openmc_inp.py` input script template (left). Templated `openmc_inp.py` with variable1 and variable2 values defined (right).

4.3 demonstrate that the control variables, `variable1` and `variable2`, will be varied from 0 to 10 and -1 to 0, respectively.

4.3.2 Evaluators

Evaluators are the nuclear software REALM utilizes to calculate objective functions. Presently, only `openmc` and `moltres` evaluators are available in REALM. In a single REALM input file, a user may define any number of evaluators. For each evaluator, mandatory input parameters are `input_script`, `inputs`, and `outputs`, and the optional input parameter is `output_script`. The `input_script` is input file template's name for the evaluator software. The user must include a input file template in the same directory as the REALM input file. The `inputs` parameter lists the control variables that are placed into the input file template. REALM utilizes jinja2 templating to insert the control variable values into the `input_script`. Lines 6 to 12 in the REALM input file (Figure 4.3) demonstrate that `variable1` and `variable2` are `inputs` into the `openmc_inp.py` `input_script`. Figure 4.4 shows the template and templated openmc script; once the `openmc_inp.py` `input_script` is templated, {{`variable1`}} and {{`variable2`}} on Lines 3 and 4 will be replaced with values selected by the REALM genetic algorithm.

The `outputs` parameter lists the output variables that the evaluator will return to the genetic algorithm. These output parameters are also known as the objective functions used to evaluate the individual. REALM uses three methods to return an output parameter.

First, if the output parameter is also an input parameter, REALM will automatically return the input parameter's value. Second, the user can use predefined evaluations. For example, in `OpenMCEvaluation`, there is a predefined k_{eff} evaluation. The user may also add predefined evaluations to `OpenMCEvaluation` or `MoltresEvaluation`, or any other coupled codes' evaluation file. Third, the user may include an `output_script` that returns the desired output parameters. The `output_script` must include a line that prints a dictionary containing the output parameters' names and their corresponding value as key-value pairs.

4.3.3 Constraints

In the constraints section, the user can define constraints on any output parameter. Any individual that does not meet the defined constraints is removed from the population, encouraging the proliferation of individuals that meet the constraints. For each constrained output parameter, the user lists the `operators` and `constrained_vals` as in Line 15 of the REALM input file (Figure 4.3). Thus, for this REALM simulation, `output_1` is constrained to be >= 1.0 and < 1.5.

4.3.4 Algorithm

In the algorithm section, the user defines all the hyperparameters for the genetic algorithm. The mandatory input parameters include `optimized_variable`, `objective`, `pop_size`, and `generations`. The user specifies an `optimized_variable`, which must be an output parameter from the evaluators' `outputs`. The user has the option to maximize or minimize this `optimized_variable` by defining the `objective` variable as `max` or `min`. The user must also specify the population size (`pop_size`) and number of generations (`generations`) in the genetic algorithm.

The optional input parameters include `mutation_probability`, `mating_probability`, `selection_operator`, `mutation_operator`, and `mating_operator`. As mentioned previ-

Table 4.1: Selection, mutation, and mating operators available in Reactor Evolutionary Algorithm Optimizer (REALM) and their corresponding hyperparameters.

Operator	Available Options	Hyperparameters
Selection	`selTournament`	`tournsize`: no. of individuals in each tournament `k`: no. of individuals to select
	`selNSGA2`	`k`: no. of individuals to select
	`selBest`	`k`: no. of individuals to select
Mutation	`mutPolynomialBounded`	`eta`: crowding degree of the mutation `indpb`: independent probability for each attribute to be mutated
Mating	`cxOnePoint`	-
	`cxUniform`	`indpb`: independent probability for each attribute to be exchanged
	`cxBlend`	`alpha`: Extent of the interval in which the new values can be drawn for each attribute on both side of the parents attributes

ously in Section 2.4.1, it is important to select genetic algorithm hyperparameters that balance the extent of exploration and exploitation. The user can define the mutation and mating probability or not define them which results in the default values of 0.3 and 0.4, respectively. For each operator, the user can choose from a list of operators and define each of their required hyperparameters. Table 4.1 shows the available operators and their respective hyperparameters. The default selection operator is `selNSGA2` with a default `k` value, two-thirds the population size. The default mutation operator is `mutPolynomialBounded` with default `eta` and `indpb` values of 0.3. The default mating operator is `cxBlend` with a default `alpha` of 0.4. Lines 17 to 31 in the example REALM input file (Figure 4.3) demonstrate `algorithm` specifications.

4.4 REALM Software Architecture

In this section, I will describe REALM v1.0's software architecture and how all the parts come together to optimize reactor design. Table 4.2 outlines the classes in the REALM software and describes each class's purpose. Figure 4.5 depicts REALM's software architecture. When

Table 4.2: Classes that makeup REALM's architecture and their description.

Class	Description
InputValidation	The ***InputValidation*** class contains methods to read and validate the JSON REALM input file to ensure the user defined all key parameters. If they did not, REALM raises an exception to tell the user which parameters are missing.
Evaluation	DEAP's fitness evaluator (as mentioned in Section 4.1.1) requires an evaluation function to evaluate each individual's fitness values. The ***Evaluation*** class contains a method that creates an evaluation function that runs the nuclear software and returns the required fitness values, defined in the input file.
OpenMCEvaluation	The ***OpenMCEvaluation*** class contains built-in methods for evaluating OpenMC output files. Developers can update this file with methods to evaluate frequently used OpenMC outputs.
ToolboxGenerator	The ***ToolboxGenerator*** class initializes DEAP's *toolbox* and *creator* modules with genetic algorithm hyperparameters defined in the input file.
Constraints	The ***Constraints*** class contains methods to initialize constraints defined in the input file and applies the constraints by removing individuals that do not meet the constraint.
BackEnd	The ***BackEnd*** class contains methods to save genetic algorithm population results into a pickled checkpoint file and to restart a partially completed genetic algorithm from the checkpoint file.
Algorithm	The ***Algorithm*** class contains methods to initialize and execute the genetic algorithm. It executes a general genetic algorithm framework that uses the hyperparameters defined in the ***ToolboxGenerator***, applies constraints defined in ***Constraints***, evaluates fitness values using the evaluation function produced by ***Evaluation***, and saves all the results with ***BackEnd***.
Executor	The ***Executor*** class drives the REALM code execution with the following steps: 1) User input file validation with ***InputValidation*** 2) Evaluation function generation with ***Evaluation*** 3) DEAP toolbox initialization with ***ToolboxGenerator*** 4) Constraint initialization with ***Constraints*** 5) Genetic algorithm execution with ***Algorithm***

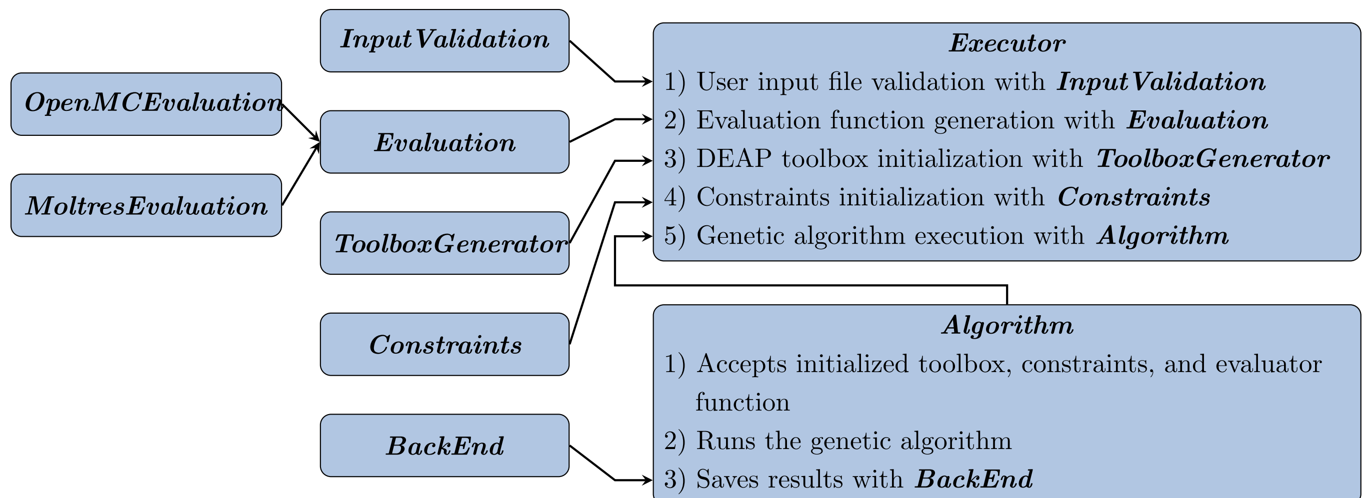


Figure 4.5: Visualization of REALM architecture.

the user runs a REALM input file, the ***Executor*** class drives REALM's execution from beginning to end. The ***Executor*** calls ***InputValidation*** to parse the input file to ensure that the user defined all mandatory parameters and used the correct formatting. Next, it initializes an ***Evaluator*** object based on the `evaluators` specifications in the input file. It uses the ***Evaluator*** object to create a function that will run each evaluator software with the desired input parameters and return the output parameters calculated by the evaluator software. Next, it uses the ***ToolboxGenerator*** to create an initialized DEAP toolbox object based on the input file's `algorithm` specifications. The ***ToolboxGenerator*** object accepts the ***Evaluator*** object and registers it as the toolbox's 'evaluate' tool. Then, it initializes a ***Constraints*** object to contain `constraints` specified in the input file. Next, the ***Executor*** initializes an ***Algorithm*** object that accepts the initialized DEAP toolbox and ***Constraints*** object. Finally, the ***Executor*** class uses a method in the ***Algorithm*** object to run a general genetic algorithm with hyperparameters from the DEAP toolbox, apply constraints defined in the ***Constraints*** object, and calculate objective functions using the evaluation function created by the ***Evaluator*** object, all the while saving the results using the ***BackEnd*** class.

In the REALM Github repository [12], I included a tests directory that contains unit

tests for all methods in the classes described above.

4.4.1 Installing and Running REALM

There are two ways to install REALM. First, a user can utilize The Python Package Index (PyPI) to install REALM: `pip install realm`. Second, a user can download the REALM Github repository [12] and install it from source.

REALM is run from the command line interface. A user should first set up the REALM JSON input file and evaluator scripts in a directory. When running REALM from the command line, there are two mandatory arguments and one optional argument. The mandatory arguments are the input file (`-i`) and objective (`-p`). The optional argument is the checkpoint file (`-c`). Thus, the structure of a command line input for running REALM is:

```
python realm -i <input file name> -p <max or min> -c <checkpoint file name>
```

The checkpoint file holds the results from the REALM simulation and also acts as a restart file. Thus, if a REALM simulation ends prematurely, the checkpoint file can be used to restart the code from the most recent population and continue the simulation.

4.4.2 REALM Results Analysis

The ***BackEnd*** class manages each REALM simulation's results. ***BackEnd*** puts all the results in a pickled dictionary, and it is saved as `checkpoint.pkl` in the same directory as the input file. The checkpoint file can then be reloaded into a Jupyter notebook and organized to produce desired plots. Examples of REALM results analysis can be found in the REALM documentation [12].

The evaluation function creates a new directory for each software, generation, and individual and stores the templated input file and output files associated with that particular run. The generation and individual values are indexed by zero. For example, the directory

containing files associated with an OpenMC run for the tenth individual in the genetic algorithm's third generation will be named: `openmc_2_9`.

4.5 Summary

This chapter described the Reactor Evolutionary Algorithm Optimizer (REALM) framework developed for the proposed work. REALM is a Python package that applies evolutionary algorithm optimization techniques to nuclear reactor design using the Distributed Evolutionary Algorithms in Python (DEAP) module, OpenMC, and Moltres nuclear software. The motivation for REALM's inception is to enable reactor designers to utilize robust evolutionary algorithm optimization methods without going through the cumbersome process of setting up a genetic algorithm framework, selecting appropriate hyperparameters, and setting up its parallelization. REALM is designed to be effective, flexible, open-source, parallel, reproducible, and usable and is hosted on Github [12].