Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: nucleaire technologie

Masterthesis

Analysis of the most critical configuration in nuclear fuel storage using an optimization algorithm

Enes Orhan

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: nucleaire technologie, afstudeerrichting nucleair en medisch

PROMOTOR :

Prof. dr. ir. Gert VAN DEN EYNDE

Gezamenlijke opleiding UHasselt en KU Leuven

UHASSELT KU LEUVEN

Universiteit Hasselt | Campus Diepenbeek | Faculteit Industriële Ingenieurswetenschappen | Agoralaan Gebouw H - Gebouw B | BE 3590 Diepenbeek

Universiteit Hasselt | Campus Diepenbeek | Agoralaan Gebouw D | BE 3590 Diepenbeek
Universiteit Hasselt | Campus Hasselt | Martelarenlaan 42 | BE 3500 Hasselt

UHASSELT KU LEUVEN

2024
2025

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: nucleaire technologie

Masterthesis

Analysis of the most critical configuration in nuclear fuel storage using an optimization algorithm

Enes Orhan

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: nucleaire technologie, afstudeerrichting nucleair en medisch

PROMOTOR :

Prof. dr. ir. Gert VAN DEN EYNDE

UHASSELT KU LEUVEN

Acknowledgements

At the conclusion of this academic journey, I would like to take a moment to express my appreciation to those who have supported me throughout the development of this thesis.

My deepest thanks go to my supervisor, Prof. Dr. Ir. Van den Eynde, whose invaluable guidance, expertise, and encouragement have shaped this work from the very beginning. His deep knowledge and enthusiasm for reactor physics sparked my own interest in the field, and his mentorship has been instrumental in bringing this dissertation to life.

I also wish to sincerely thank Prof. Dr. Bex, whose assistance with the supercomputing infrastructure enabled me to carry out essential simulations. Through his support, I was able to explore computational challenges that have enriched my understanding and practical skills in ways that go far beyond this thesis.

In addition, I would like to thank UHasselt and KU Leuven for providing a strong academic foundation and the opportunity to study within the framework of their nuclear engineering program. The knowledge and skills I acquired during this program have been instrumental in completing this thesis.

Finally, I would like to express my heartfelt gratitude to my family and classmates. Their unwavering support, patience, and encouragement have been a constant source of strength throughout this journey. In particular, I want to thank Sven, Jelle, and Jorrit—their presence, shared determination, and *La Passion*-like perseverance made a meaningful difference in completing this work.

Contents

List of Tables

List of Figures

Nomenclature

ϕ	Neutron flux
φ	Angular neutron flux
$\hat{\Omega}$	Angular direction
$\chi(E)$	Neutron birth energy spectrum
Σ_t	Total macroscopic cross section
σ	Microscopic cross section
∇	Gradient operator

barn	Unit of area equal to 10^{-24} cm^2
eV	Electronvolt, unit of energy
MeV	Mega electronvolt (10^6 eV)
K	Kelvin, unit of temperature
k_{eff}	Effective neutron multiplication factor
^{235}U	Uranium-235, fissile isotope of uranium
^{238}U	Uranium-238, fertile isotope of uranium
^{239}Pu	Plutonium-239, fissile isotope of plutonium
BO	Bayesian Optimization
CPU	Central Processing Unit
DEAP	Distributed Evolutionary Algorithms in Python
Demon core	6.2 kg metallic plutonium sphere involved in criticality accidents
E	Neutron energy
GA	Genetic Algorithm
GB	Gigabyte, unit of digital information
GP	Gaussian process
LET	Linear Energy Transfer

n	Neutron density
N	Atomic number density
NTE	Neutron Transport Equation
OpenMP	Shared-memory multiprocessing API
PWR	Pressurized Water Reactor
RAM	Random Access Memory
Slurm	Simple Linux Utility for Resource Management
VSC	Flemish Supercomputing Center
wICE	Tier-2 supercomputing cluster at KU Leuven/UHasselt

Abstract

Accidental chain reactions, or criticality accidents, can occur when fissile materials like uranium are arranged to sustain a self-amplifying neutron reaction. This thesis investigates how the spatial distribution of uranium dissolved in water influences the risk of reaching a critical state. A cylindrical vessel—commonly used in nuclear storage—was modeled, and thousands of possible configurations were evaluated using the Serpent Monte Carlo code. A genetic algorithm (GA) was implemented to evolve and optimize the most reactive arrangements, supported by a Bayesian optimizer that predicts promising distributions. Simulations were run using 1430 cells, a population of 100, and a 5-day wall time on a supercomputer. The GA successfully identified more critical distributions than the initial biases in several cases. However, some runs failed to achieve criticality, highlighting a trade-off between convergence speed and solution quality. Geometry and uranium mass were found to influence spatial patterns significantly. Limitations included the small population size, high computational demands, and supercomputing-related issues. This study demonstrates the potential of GAs in criticality simulations and suggests future improvements such as adaptive Serpent settings, progressive segmentation, and informed initialization. Fine-tuning algorithmic parameters could further enhance performance and robustness.

Abstract (in Dutch)

Criticaliteitsongevallen kunnen ontstaan wanneer splijtbaar materiaal, zoals uranium, zodanig verdeeld is dat een zichzelf versterkende kettingreactie mogelijk wordt. Deze scriptie onderzoekt hoe de ruimtelijke verdeling van in water opgeloste uranium invloed heeft op het risico op kriticiteit. Een cilindrisch vat, typisch voor nucleaire opslag, werd gemodelleerd. Duizenden configuraties werden geëvalueerd met behulp van de Serpent Monte Carlo-code. Een genetisch algoritme (GA) werd gebruikt om kritische verdelingen te optimaliseren, ondersteund door een Bayesian optimizer die veelbelovende configuraties voorspelt. De simulaties werden uitgevoerd met 1430 cellen, een populatie van 100 individuen en een rekentijd van vijf dagen op een supercomputer. In meerdere gevallen vond het GA kritischer verdelingen dan de oorspronkelijke beginsituaties. Sommige runs bereikten echter geen kriticiteit, wat wijst op een afweging tussen snelheid en nauwkeurigheid. Geometrie en massa bleken bepalend voor de verdelingspatronen. Beperkingen waren onder andere de beperkte populatiegrootte, hoge rekenlast en technische beperkingen bij supercomputing. Deze studie toont het potentieel van GA's in criticaliteitsanalyses en beveelt verbeteringen aan zoals adaptieve Serpent-instellingen, progressieve segmentatie en slimme initialisatie.

Chapter 1

General

1.1 Introduction

Nuclear energy is a powerful phenomenon with a broad range of applications. The energy released during nuclear reactions and radioactive decay can be harnessed for electricity generation, medical treatments, advanced materials research, and more. Among these processes, fission is particularly important due to the tremendous amounts of energy it produces.

When a fission reaction occurs, the surrounding environment can enable a self-sustaining chain reaction. This may be intentional—as in a nuclear reactor—or accidental, which can lead to hazardous situations. The intense energy and radiation released during a criticality event can be lethal to anyone nearby if proper shielding and protocols are not in place.

Several criticality accidents have occurred throughout history. One notable example is the Tokaimura accident in Japan, where two workers died after receiving fatal doses of radiation [1]. The incident was caused by the unauthorized use of stainless steel buckets to dissolve uranium oxide powder, bypassing the mandated dissolution tank designed with a safe geometry for fissile material. This violation led to the formation of a critical mass inside a precipitation tank [2]. The workers' lack of understanding of the parameters influencing criticality ultimately led to the accident.

Furthermore, a criticality event is not excluded to one setting. Criticality risks can also arise during nuclear fuel fabrication, storage, and transportation. Conservative design approaches and strict procedural safeguards are essential to prevent accidental criticality in any system handling fissile material.

It is crucial to carefully manage parameters such as mass, enrichment, geometry, moderation, reflection, density, and interactions between units [3]. However, these are not the only factors that matter. The spatial distribution of fissile material within a system also plays a significant role. For example, the effective multiplication factor of a uranium precipitate in solution can differ substantially from that of a uniformly distributed uranium solution.

1.2 Problem Statement

While criticality safety is a well-researched field, there remains a lack of studies focusing on the most critical distribution of fissile material within a system. The optimal distribution can vary significantly depending on factors such as fissile material mass, enrichment level, and the dimensions of the system. For certain parameters, the most critical configuration might involve a higher concentration of mass at the center of the vessel, while in other cases, a uniform solution might be more dangerous. Most likely, the critical configuration lies somewhere between these extremes.

Understanding whether real-world scenarios could drive a system toward criticality is crucial for enhancing safety measures. Several situations highlight the importance of studying the most critical configuration of fissile material within a system.

One example is precipitation: if uranium powder settles at the bottom of a vessel, a locally critical configuration could develop. Similarly, if liquid is lost through evaporation or leakage, the resulting increase in fuel concentration could push the system closer to criticality.

Mechanical disturbances—such as dropping or shaking a vessel—could also redistribute precipitated material unevenly. In such cases, fissile material might accumulate at the center or along the walls of the vessel, creating a more reactive geometry.

By understanding the critical risks associated with these scenarios, better strategies can be developed to prevent accidents and improve overall system safety.

The objective of this thesis is to determine the most critical geometric distribution of uranium in solution under these conditions. The analysis will explore how variations in total mass, enrichment level, and vessel geometry influence the critical configuration. Criticality calculations will be performed using the Serpent simulation software. An optimization algorithm will simulate different distributions to identify the configuration that presents the highest risk of criticality. Through iterative simulations and detailed analysis, the most critical arrangement of fissile material will be determined.

Chapter 2

Theoretical Framework

2.1 Nuclear Physics

2.1.1 Nuclear Fission

When a nucleus is bombarded with neutrons, there is a probability that it will undergo a nuclear reaction. Among these reactions, fission is of particular interest. When a fissile nucleus, such as ^{235}U, absorbs a neutron, it can split into two smaller daughter nuclei. Together with these fission products, the reaction produces a substantial amount of energy. In the case of ^{235}U, this energy amounts to approximately 200 MeV per fission event (for comparison: the burning of a single carbon atom in a fire produces around 4 eV). In addition to energy, each fission event also emits a few neutrons (see Fig. 2.1).

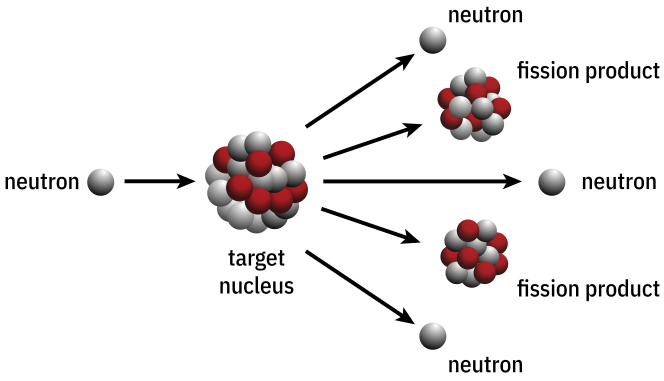


Figure 2.1: Nuclear fission reaction [4]

The exact number of released neutrons depends on the energy of the incident neutron and the nuclide that is being fissioned. But on average, most fissile nuclei emit between two and three neutrons per fission [5]. These newly generated neutrons can subsequently induce further fission reactions in other fissile nuclei, potentially leading to a self-sustaining chain reaction.

2.1.2 Fast, Thermal and Epithermal neutrons

As mentioned earlier, the energy of the incoming neutron is an important factor in determining its interactions with a nucleus. Neutrons are generally classified into three broad categories based on their energy: fast, thermal and epithermal neutrons.

Thermal neutrons derive their name from their interaction with the surrounding medium, where they reach thermal equilibrium with the environment. This means their energy distribution follows the Maxwell-Boltzmann distribution, which describes the statistical distribution of particle energies in a system at thermal equilibrium [6]. Figure 2.2 shows the distribution for different temperatures.

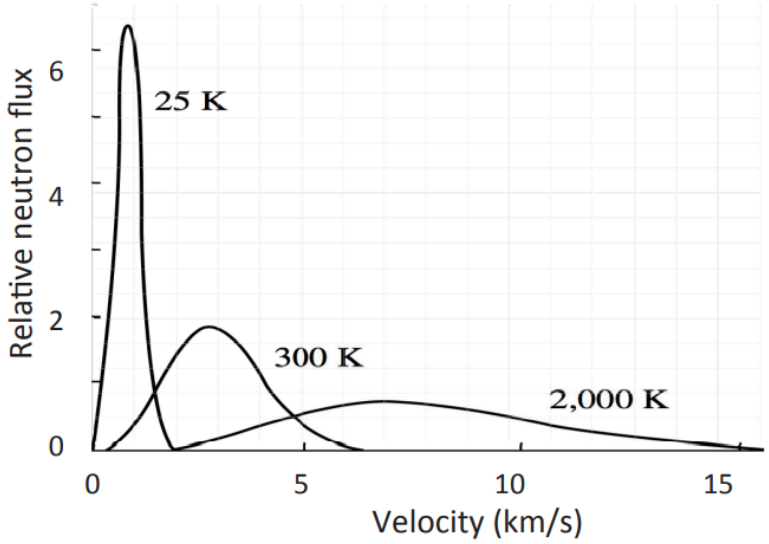


Figure 2.2: The Maxwell-Boltzmann flux distribution for 25 K, 300 K and 2000 K moderator temperature [7]

In most reactor environments, thermal neutrons have energies of approximately 0.025 eV at room temperature (293 K).

In contrast, fast neutrons possess much higher kinetic energies, typically ranging from 1 MeV to 10 MeV. In a nuclear reactor context, these are produced directly during fission reactions. Epithermal neutrons have energies that fall between the thermal and fast neutron energy ranges [6].

In PWRs, thermal neutrons are particularly important because they have a significantly higher probability of inducing fission compared to fast neutrons. Therefore, slowing down fast neutrons — a process known as moderation — is a crucial aspect of reactor operation [6].

2.1.3 Interactions of interest

Before going further, neutrons can experience some other interactions besides fission. These are absorption and scattering. During an absorption reaction, the nuclide captures a neutron but does not undergo fission. For example a hydrogen atom can absorb a neutron and become deuterium, which consists of a proton and a neutron [8].

$$^{1}\mathrm{H} + \mathrm{n} \rightarrow {}^{2}\mathrm{H} + \gamma$$

Furthermore, scattering is another key interaction which can be classified into two types: elastic and inelastic. The distinction lies in how energy is exchanged during the interaction. In an elastic scattering event, a neutron collides with a nucleus—such as a proton—and transfers part of its kinetic energy, causing the neutron to slow down without exciting the nucleus [9]. This process

is fundamental for moderating neutrons in a thermal neutron reactor, as well as for neutron reflection, both of which will be discussed later. In contrast, inelastic scattering involves the neutron transferring enough energy to the nucleus to excite it to a higher energy state, often resulting in less efficient moderation at low energies [9].

2.1.4 Microscopic Cross Section

The microscopic cross section σ, represents the probability that a particle will interact with a specific nucleus and undergo a particular nuclear reaction. It is measured in barns ($1\ barn = 10^{-24} cm^{-2}$), a unit roughly corresponding to the cross-sectional area of a uranium nucleus [10]. For fission, this probability is quantified by the fission cross section, which depends on several factors. The most significant are the type of nucleus and the energy of the incoming neutron [6].

In general, fissile isotopes such as ^{235}U exhibit a much higher fission cross section for thermal (low-energy) neutrons. However, some isotopes primarily absorb neutrons at specific energy ranges rather than undergoing fission. For example, ^{238}U can absorb a neutron and, through subsequent decay, transform into ^{239}Pu — a fissile material with a high fission cross section for thermal neutrons. Because of this, ^{238}U is classified as a fertile nuclide [6], [9].

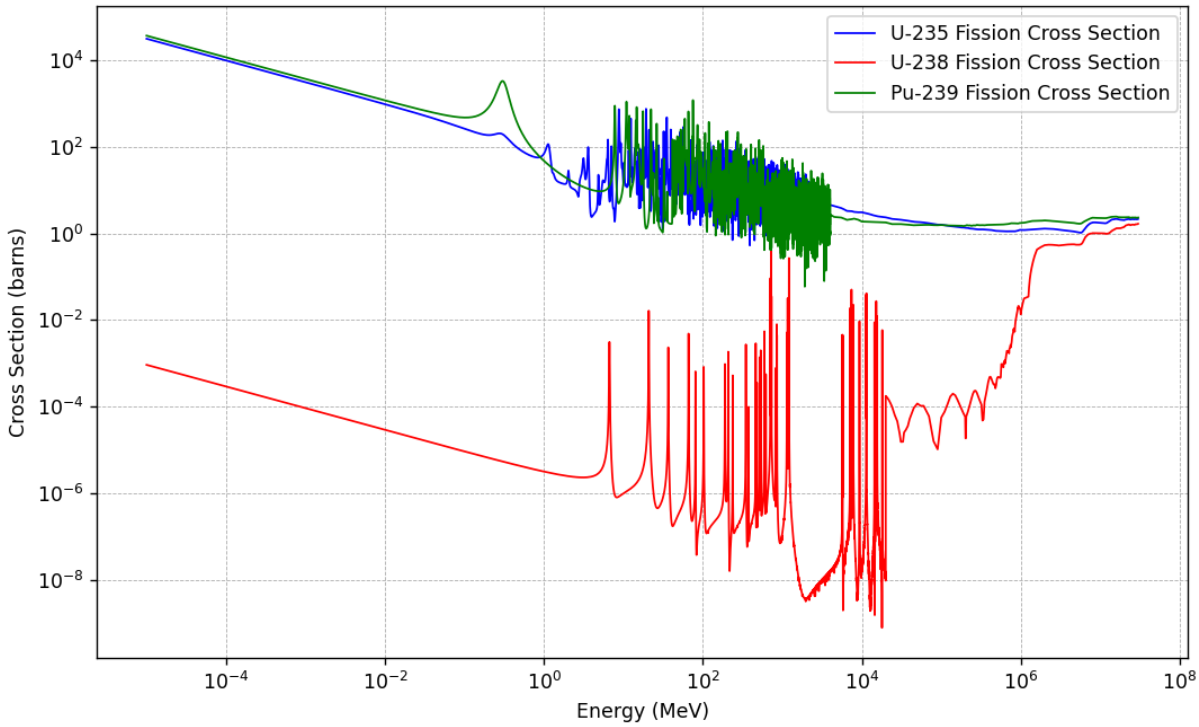


Figure 2.3: Fission cross sections for ^{235}U, ^{238}U and ^{239}Pu [11]

2.1.5 Neutron Flux

The following theoretical development in Sections 2.1.5 through 2.1.7 is primarily based on the course "Reactor Physics" taught by Prof. Dr. Ir. Van den Eynde at Hasselt University [6].

In reactor physics, the neutron flux is a fundamental quantity representing the flow of neutrons through a unit area. However, since the flux can vary with direction, the **scalar flux** is introduced to simplify the description of neutron behaviour.

The scalar flux, denoted as $\phi(\mathbf{r}, E, t)$, is obtained by integrating the angular neutron flux over all directions. It represents the **total neutron flux at a point**, independent of direction. Mathematically, the scalar flux is expressed as:

$$\phi(\mathbf{r}, E, t) = \int_{4\pi} \varphi(\mathbf{r}, \mathbf{\Omega}, E, t)\, d\Omega$$

Here:

- $\varphi(\mathbf{r}, \mathbf{\Omega}, E, t)$ is the **angular flux**, describing the neutron flux as a function of position ($\mathbf{r}$), direction ($\mathbf{\Omega}$), energy (E), and time (t).
- The integration is performed over the entire solid angle (4π).

The scalar flux can be interpreted as the **average number of neutrons passing through a unit area per unit time, regardless of their direction**. Now that we have established a clear understanding of the scalar and angular flux, we can proceed to discuss the Neutron Transport Equation (NTE).

2.1.6 Neutron Transport Equation

The neutron population within a system is not static; it constantly changes as neutrons are gained or lost. The rate of change in the neutron population within a given volume can be expressed as the difference between the neutron gain mechanisms and the neutron loss mechanisms within that volume:

$$\frac{\partial}{\partial t}\left[\int_V n(\mathbf{r}, E, \hat{\Omega}, t)\, d^3\mathbf{r}\right] dE\, d\hat{\Omega} = \textit{Gain in } V \textit{ - Loss in } V$$

Interaction loss is one of the loss mechanisms for neutrons in a system. Neutrons can undergo interactions such as absorption or scattering, which result in their removal from the population being monitored. Since multiple types of interactions can occur, they are collectively represented by the total macroscopic cross section, Σ_t. The corresponding loss term is given by:

$$\int_V \Sigma_t(\mathbf{r}, E, t)\varphi(\mathbf{r}, E, \hat{\Omega}, t)\, dV$$

The total macroscopic cross section $\Sigma_t(\mathbf{r}, E, t)$ represents the probability per unit path length that a neutron at a given position $\mathbf{r}$, with energy E, and at time t, will undergo an interaction with the material.

The macroscopic cross section Σ is calculated as the product of the **microscopic cross section** σ and the **atomic number density** N of the material:

$$\Sigma = \sigma \times N$$

Hence, Σ accounts for the cumulative likelihood of neutron interactions within the material, incorporating both the probability per nucleus and the number of target nuclei per unit volume.

Additionally, neutrons may also leak out of the volume being considered. The leakage term can be initially expressed as:

$$\int_A j(\mathbf{r}, E, \hat{\Omega}, t)\, d\vec{A}$$

However, since all other terms are integrated over the volume, this term is transformed using the Gauss (Divergence) theorem to:

$$\int_V \hat{\Omega} \cdot \nabla\varphi(\mathbf{r}, E, \hat{\Omega}, t)\, dV$$

Furthermore, the population increases through gain mechanisms. These are fission and scattering. Each fission event releases between 2 and 3 neutrons, depending on the energy of the incident neutron. The fission gain term is given by:

$$\int_V \frac{\chi(E)}{4\pi} \int_0^\infty \int_{4\pi} v(E')\Sigma_f(\mathbf{r}, E', t)\varphi(\mathbf{r}, E', \Omega', t)\, d\Omega'\, dE'\, dV$$

The term $\chi(E)$ represents the neutron birth spectrum, or the energy distribution of emitted neutrons. The function $v(E')$ indicates the number of neutrons produced per fission as a function of the energy E' of the inducing neutron.

When neutrons scatter from a different energy and angle to the monitored energy and angle, they contribute to the neutron population. This is called in-scattering and the term is:

$$\int_V \int_0^\infty \int_{4\pi} \Sigma_S(\mathbf{r}, E' \to E, \hat{\Omega}' \to \hat{\Omega}, t)\varphi(\mathbf{r}, E', \Omega', t)\, d\hat{\Omega}'\, dE'\, dV$$

Here, Σ_S represents the scattering cross section, tracking neutrons moving from energy E' to E and from direction Ω' to Ω.

If an independent neutron source exists, an additional gain term is added:

$$\int_V s(\mathbf{r}, E, \hat{\Omega}, t)\, dV$$

This term does not include the neutron population density $n(\mathbf{r}, E, \hat{\Omega}, t)$ since it is independent of it.

Combining all terms together, the neutron transport equation is complete.

$$\int_V \frac{\partial n(r, E, \Omega, t)}{\partial t} dV = -\int_V \Sigma_t(r, E, t)\varphi(r, E, \hat{\Omega}, t) dV$$

$$-\int_V \hat{\Omega}.\nabla\varphi(r, E, \hat{\Omega}, t) dV$$

$$+\int_V \frac{\chi(E)}{4\pi} \int_0^\infty \int_{4\pi} v(E')\Sigma_f(r, E', t)\varphi(r, E', \Omega', t) d\Omega' dE' dV$$

$$+\int_V \int_0^\infty \int_{4\pi} \Sigma_S(r, E' \to E, \hat{\Omega}' \to \hat{\Omega}, t)\varphi(r, E', \Omega', t)d\hat{\Omega}'dE'dV$$

$$+\int_V s(r, E, \hat{\Omega}, t)dV$$

2.1.7 The k-eigenvalue problem

When the loss of neutrons is equal to the gain of neutrons. The neutron population remains stable, and the system is therefore stationary ($\int_V \frac{\partial n(r,E,\Omega,t)}{\partial t} dV = 0$) . In the context of an operating reactor, the independent neutron source is typically absent or negligible so it will be excluded from further consideration. Furthermore, to simplify the equation, it can be written in operator form.

$$\mathcal{M}\varphi(\mathbf{r}, E, \hat{\Omega}) = \mathcal{F}\varphi(\mathbf{r}, E, \hat{\Omega})$$

where

- $\mathcal{M}$ = *Leakage term + total interaction loss term - scattering gain term*
- $\mathcal{F}$ = *Fission term*

This problem is a homogeneous problem but it isn't necessarily singular when looking for a non-trivial solution. To find a stationary solution that is non-zero, a scaling factor k is introduced for the fission operator.

$$\mathcal{M}\varphi(\mathbf{r}, E, \hat{\Omega}) = \frac{1}{k}\mathcal{F}\varphi(\mathbf{r}, E, \hat{\Omega})$$

Here, k is known as the *effective multiplication factor* (k_{eff}), which determines the system's criticality state [9]:

- If $k > 1$, fission production exceeds neutron losses, meaning the system is *supercritical*.
- If $k < 1$, neutron losses exceed fission production, meaning the system is *subcritical*.
- If $k = 1$, the neutron population remains constant, meaning the system is *critical*.

Now this problem becomes a k-eigenvalue problem which can be solved using e.g. the Power iteration method. Solving for this parameter is crucial to asses the criticality of a system containing fissile material. Different approaches exist to determine k_{eff} for a system, each with its own advantages and applications.

This concludes the theoretical foundation based on the aforementioned course material [6].

2.2 MERMAIDS

When fissile material is arranged in a critical configuration, a self-sustaining chain reaction can occur. This reaction releases tremendous amounts of energy in the form of heat and ionizing radiation [1]. If this happens unintentionally, it is referred to as a criticality accident [5], posing an immediate, life-threatening danger to personnel nearby.

Ionizing radiation can damage DNA by breaking molecular bonds, leading to cell death or mutations. At high radiation doses, widespread cellular damage can occur, resulting in severe health effects [12]. The extent of biological harm also depends on the type of radiation: charged particles such as alpha particles (helium nuclei) and beta particles (electrons) have a higher linear energy

transfer (LET) compared to gamma rays. As a result, they cause more intense local damage, even though gamma radiation is more penetrating [12].

Because of these risks, ensuring the safety of systems containing fissile material is crucial. Conservative design practices, simplicity in control mechanisms, and strict adherence to safety protocols are essential. Overly complex systems can introduce opportunities for human error.

Several factors influence the likelihood of reaching a critical state. The acronym MERMAIDS is commonly used to help remember these criticality parameters [6].

MERMAIDS:

- **M** = Mass
- **E** = Enrichment
- **R** = Reflection
- **M** = Moderation
- **A** = Absorption
- **I** = Interaction
- **D** = Density
- **S** = Shape and Size

To fully understand criticality, it is essential to first grasp these underlying parameters. In the following sections, some parameters and their principles will be discussed in detail.

2.2.1 Moderation

The dilution of a fissile material can decrease the critical mass. This is because, in most cases, the fissile material is diluted in an aqueous solution. Due to their similar masses, the light hydrogen nuclei provide optimal energy transfer for neutrons per collision. They can therefore quickly slow down the fast neutrons to thermal energies. Increasing the likelihood of the neutrons inducing a fission [9]. An interpretation of this could be a golf ball that does not go too fast when trying to drop into a hole as shown in figure 2.4.

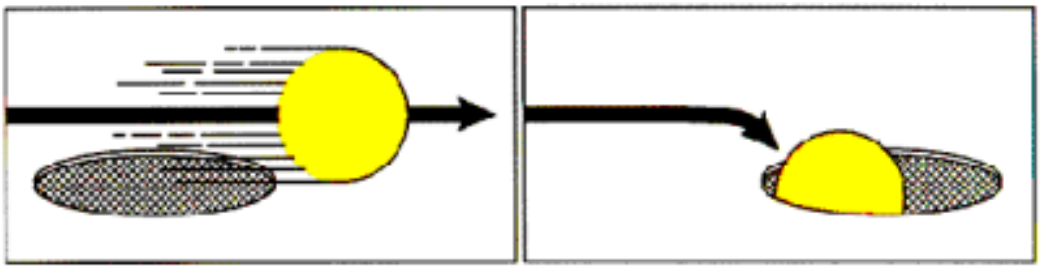


Figure 2.4: A golf ball that rolls slowly is more likely to fall into the hole [3]

This effect is called moderation and it plays a crucial role in keeping control of a chain fission reaction. Still hydrogen nuclei can absorb neutrons and diluting the fissile material too much or surrounding it with too much water can push the system to a over-moderated state. This

reduces the reactivity but is also dangerous since losing water would make the system more critical. Inversely, if a system is under moderated, the introduction of water over a longer period of time could drive the system to criticality. This is a particular concern in the context of dry stored nuclear fuel where cracks could allow the ingress of water in the form of mist or condensation increasing the reactivity over time [5].

2.2.2 Fissile Material Density

One could assume that for a subcritical system to become critical, adding more fissile material would suffice. However, the critical mass of a metallic sphere of ^{239}Pu is much lower then that of a sphere containing fillings or chips of ^{239}Pu [5], [9]. Research shows that if a subcritical metallic sphere of ^{239}Pu is dissolved to a subcritical solution containing ^{239}Pu, somewhere inbetween the system can still become supercritical [13]. This crucial parameter called the fissile material density therefore has a significant impact on the criticality of a system. Some vessels could have a subcritical configuration with precipitated fissile material, but could reach criticality due to stirring after e.g. a fall or other mechanical disturbances [3], [9]. Figure 2.5 illustrates the hazard.

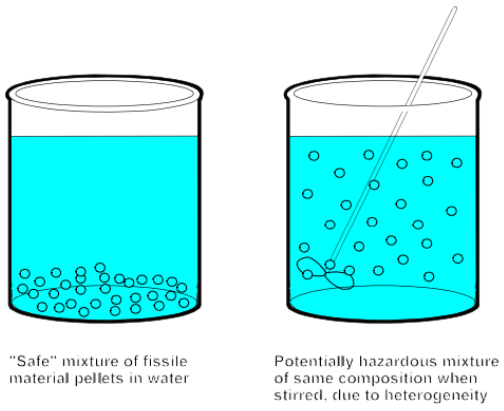


Figure 2.5: Criticality hazard after accidental stirring [3]

Furthermore, an increasing fuel concentration due to water evaporation or leakage out the vessel could also push a system towards criticality as shown in figure 2.6.

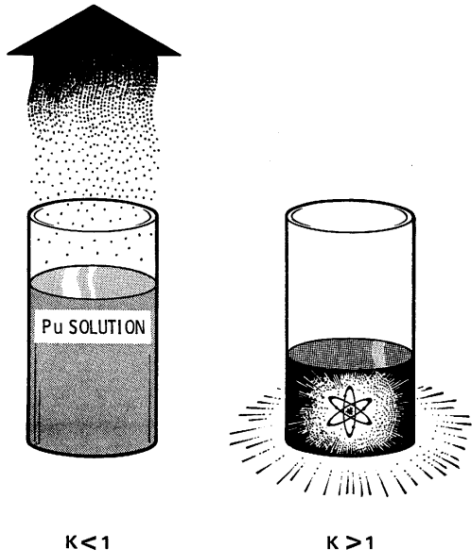


Figure 2.6: Criticality hazard due to evaporation [9]

In short, fuel concentration is crucial as it can significantly reduce the amount of fissile material needed to achieve criticality. For example, a solid metallic sphere of uranium enriched to 97.67% ^{235}U has a critical mass of 21.6 kg when surrounded by a water reflector. In contrast, a homogeneous water-reflected solution of uranium enriched to 98% can become critical with just around 800 grams of fissile material [14]. Therefore, the fuel density has a significant role in determining a system's criticality.

Adequate spacing between atoms is necessary to ensure sufficient moderation. This aspect is particularly important when determining the most critical distribution of uranium in a solution.

2.2.3 Geometric Dependency

Departing from the spherical shape increases surface area and thus leakage. To store fissile material, favored geometries are long and small diameter cylinders or thin slab geometries [5]. Figure 2.7 shows a favorable (favorable as in safe and not sustaining a chain reaction) geometry.

Figure 2.7: Favorable geometry for safety [3]

2.2.4 Reflection

Another important factor to keep in mind is neutron reflection. Surrounding a system with a reflective material reduces neutron leakage and can push the system closer to criticality. Many different materials can act as reflectors, but water is commonly used because it also helps to moderate neutrons. However, neutron reflection has been the cause of several accidents. One of the most famous examples is the Demon Core accident [1]. During a manual experiment, a researcher noticed that the beryllium brick (a strong neutron reflector) he was holding over a plutonium sphere was making the system supercritical. He quickly tried to pull the brick away, but it slipped from his hand and fell onto the sphere, pushing the system even further into a supercritical state. This caused an intense burst of radiation, and the researcher died nine days later from the exposure [1]. Figure 2.8 demonstrates the principle of neutron reflection, where neutrons that would otherwise escape the system are redirected back, enhancing reactivity.

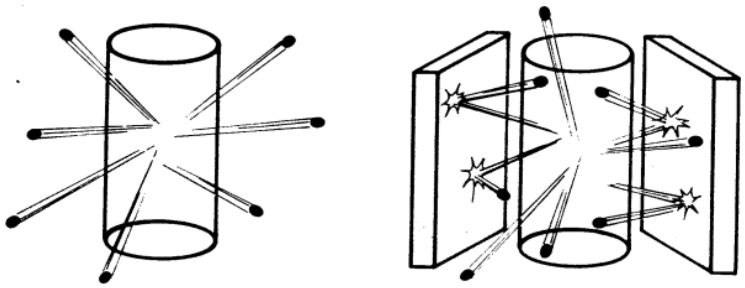


Figure 2.8: Reduction of neutron leakage due to reflecting objects [3]

Furthermore, it is important to remember that the human body itself can act as a neutron reflector, since it is largely made of water. A system that is safely subcritical on its own might become critical if a person stands too close to it.

Another prime example of risk management in this context is assessing the potential flooding of subcritical storage vessels, which could become critical after the introduction of the water [5], [9].

2.2.5 Other factors

Several factors influencing reactivity have already been discussed, but a few more are worth considering. One important factor is the enrichment of the fissile material, which indicates the proportion of fissile nuclides within the material. For instance, if we have 1 kg of uranium with an enrichment of 0.80, approximately 800 grams will consist of ^{235}U, while the remaining 200 grams will be ^{238}U. Higher enrichment increases the number of fissile nuclei per unit volume, thereby enhancing reactivity [3], [5].

Additionally, storing multiple subcritical vessels in close proximity can lead to an unintended increase in reactivity. This interaction between vessels must always be accounted for, as it can push an otherwise subcritical system toward criticality [3] (see Fig. 2.9).

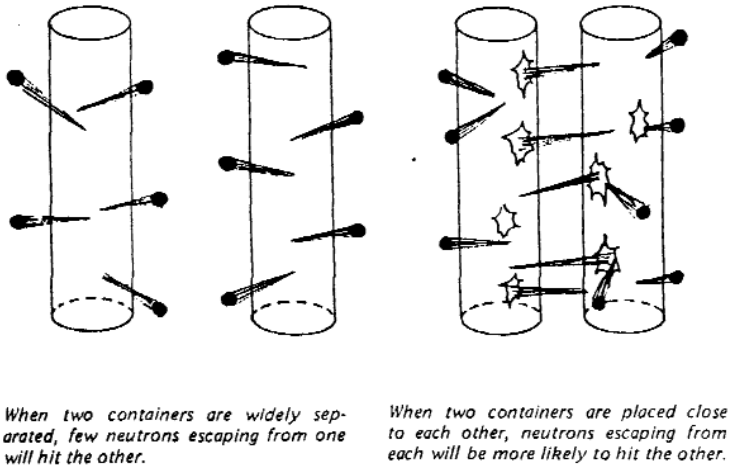


Figure 2.9: Interaction between vessels increasing overall reactivity [3]

2.3 Monte Carlo Simulations

While it is possible to solve the NTE using numerical methods, doing so becomes impractical for this research due to the large number of varying configurations involved. Implementing direct methods for each of these would be complex and inefficient—particularly in cases where the fuel distribution exhibits strong gradients [6]. Since our main interest lies in determining the k_{eff}, which is exactly what Monte Carlo methods are designed to compute, they are a suitable and efficient choice for this application.

The Monte Carlo method operates as follows: it is a computational method used to estimate the probability of various outcomes in random processes. When the probability of a specific event is known, a random number can be used to determine whether or not the event occurs. By repeating this simulation many times, reliable statistical estimates can be obtained [6].

A simple analogy can illustrate this concept. Suppose a football striker has a 60 percent chance of scoring a goal when receiving the ball in the box. To simulate a single attempt, one could generate a random number between 0 and 100. If the number is less than or equal to 60, the striker scores; if it is greater, the striker misses. Repeating this simulation over many attempts for all players of the team would provide a good approximation of the teams performance — assuming the random numbers are truly random.

This is why Monte Carlo methods are widely used in nuclear physics, where systems often involve complex interactions and countless particles. Because these systems can exhibit a large variety of possible outcomes, Monte Carlo simulations are a powerful tool to model their behaviour and predict statistical trends.

2.3.1 Serpent Monte Carlo Code

Serpent is a Monte Carlo-based neutron transport code designed to evaluate the criticality of nuclear systems. To run a simulation, the user must first define the system's configuration in an input file. This file contains detailed material and geometry definitions necessary for modelling the system [15].

The material definition section specifies the composition of materials present in the system, including the types of nuclides and their relative atomic or mass fractions. The geometry is constructed by dividing the system into cells, which are defined using basic geometrical shapes such as cylinders, planes, spheres, and cubes. Each cell is then assigned a material from the material definitions, effectively mapping the physical setup of the configuration [6].

Once the setup is complete, the simulation begins. Serpent generates a specified number of source neutrons and distributes them randomly throughout the defined cells [15]. For each neutron, it calculates the track length, which is the distance a particle can travel before interacting. If this length exceeds the distance to the nearest cell boundary, the neutron moves into the adjacent cell [6], [16].

Within each cell, Serpent uses probabilistic methods based on the isotope composition to determine which nuclide the neutron will interact with, and what type of interaction will occur (e.g., fission, absorption, scattering). If the neutron survives the interaction, it continues its path; otherwise, it is removed from the simulation [6], [16]. The general flow that the particles follow is shown in Figure 2.10.

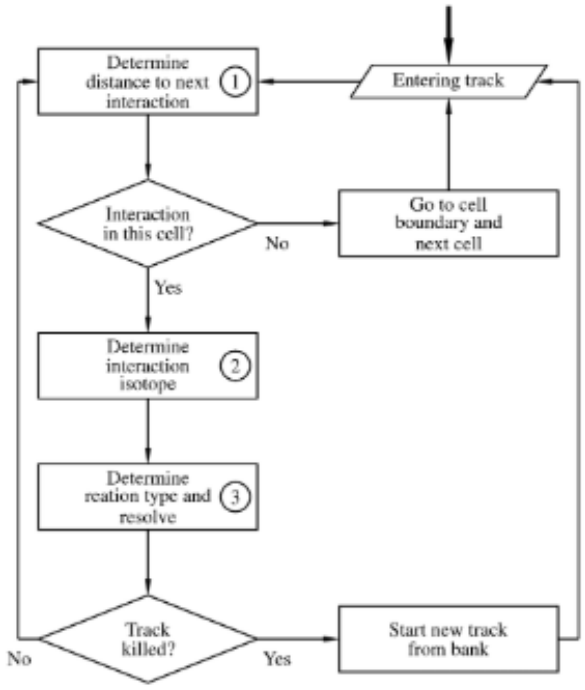


Figure 2.10: General Monte Carlo simulation flow [16]

For example, if 4000 neutrons are simulated in a single generation and only 3000 survive after accounting for all interactions, the resulting effective multiplication factor would be 0.75. Serpent repeats this process over many neutron generations to calculate an average k_{eff}, providing a reliable assessment of system criticality. As the system size and the number of particles increase, so do the computational demands. Fortunately, Monte Carlo simulations can be parallelized because each simulation is independent of the others. This makes it possible to distribute the computational workload across multiple CPU cores and aggregate the statistical information from the independent runs in a final result.

2.4 Genetic Algorithms

Identifying the most critical distribution of uranium within the vessel is fundamentally an optimization problem. The objective is to find the configuration that maximizes the system's reactivity, representing the most dangerous scenario from a criticality safety perspective. Given the complex and non-linear relationship between uranium distribution and neutron behavior, solving this problem directly is challenging. Genetic algorithms offer an effective solution for this type of optimization task.

Their ability to find global optima even in complex, non-linear, and high-dimensional problems where traditional methods might get stuck in local optima. Additionally, GAs are non-intrusive: they do not require detailed knowledge of the internal structure of the problem, only a way to evaluate how good a solution is. This makes them extremely versatile and useful for a wide range of applications.

Genetic algorithms are computational models of biological evolution. Here, individuals (bit strings) undergo natural selection comparable to the natural world. First, a population of individuals is created where slight variations between individuals make some more fit than others.

Based on a fitness parameter individuals will survive or be eliminated. Selected (surviving) individuals will be copied to the next generation. But not entirely, the selected individuals will be crossed and create offspring. Some of the newly produced offspring can also undergo mutation (random bit flipped) [17].

The following figure illustrates the working principle of the genetic operators—selection, crossover, and mutation—in greater detail.

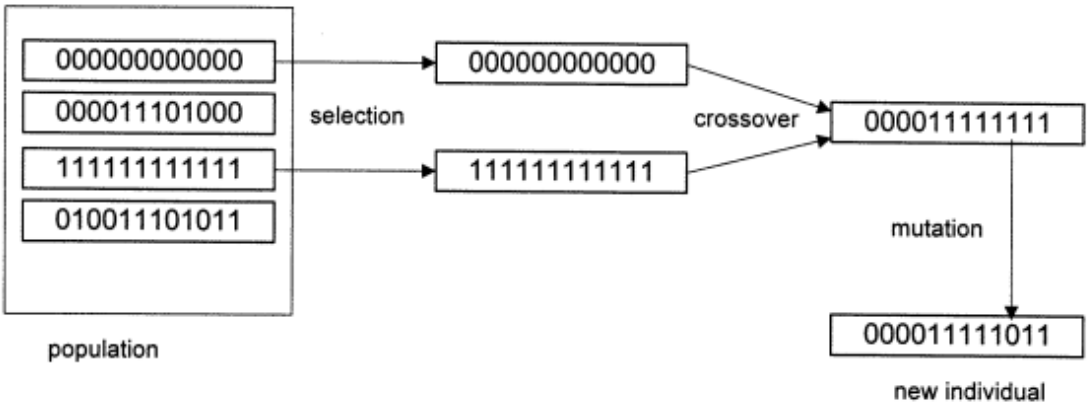


Figure 2.11: Genetic operators : selection, crossover and mutation [18]

Over successive generations, the population evolves toward increasingly optimal solutions. Ideally, each new generation is, on average, fitter than the one before. By repeating this evolutionary cycle for many generations, genetic algorithms can effectively converge on the best—or near-best—solution to complex optimization problems.

2.5 Bayesian Optimization

Bayesian Optimization (BO) is a sample-efficient technique for finding the optimum of objective functions that are expensive to evaluate and analytically unknown. These so-called *black-box functions* do not provide derivatives, nor do they have a closed-form expression that can be directly analysed. In such cases, the only way to learn about the function is by evaluating it at specific points—and since each evaluation can be costly, as in large-scale simulations or physical experiments, it is crucial to minimize the number of evaluations required [19].

Bayesian Optimization addresses this challenge by constructing a *surrogate model*, typically a *Gaussian Process (GP)*, which is updated after each function evaluation. The GP provides both a mean prediction and a measure of uncertainty (variance) over the entire domain. Instead of sampling blindly, the algorithm uses an *acquisition function* to determine where to evaluate the function next. This function balances two competing objectives:

- **Exploration**: sampling where the surrogate model is uncertain.
- **Exploitation**: sampling where the surrogate model predicts high performance.

The result is a method that intelligently navigates the search space to find the global optimum using relatively few evaluations.

A typical run of BO is shown in Figure 2.12. The dashed line represents the true (unknown) objective function, while the solid line shows the surrogate model's prediction. The shaded region indicates the model's uncertainty. The green curve represents the acquisition function, which peaks in regions of high expected improvement. In each iteration, the acquisition function is maximized to select the next point to evaluate, which is then used to update the surrogate model. Over time, this iterative process refines the model and converges toward the optimal solution [19].

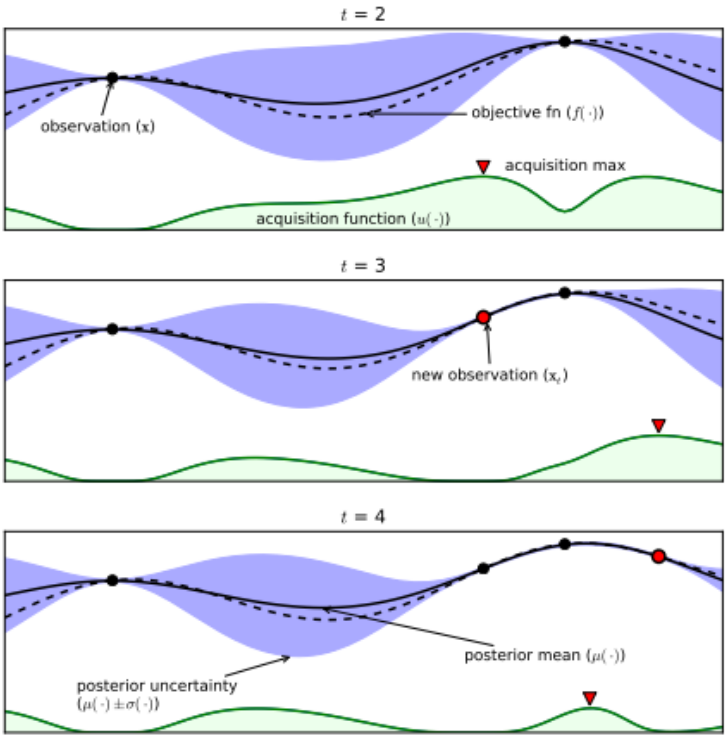


Figure 2.12: Bayesian Optimization in action [19]

This approach is especially useful in optimization problems where evaluations are slow or expensive, or where the objective function has multiple local optima. It offers a principled way to make decisions under uncertainty and adaptively improves over time.

However, Bayesian Optimization also comes with limitations that must be considered:

- **Limited information:** Often, the function is expensive to sample, so data is sparse. In such cases, one must either make strong (and possibly incorrect) assumptions about the function or rely on weak priors, both of which affect the optimization performance.
- **Exploration vs Exploitation:** Tuning the acquisition function is a challenge. Excessive exploration wastes evaluations without progress, while excessive exploitation risks converging to a local optimum instead of the global one.
- **Scalability with dimension:** As the number of parameters increases, the search space grows exponentially. This "curse of dimensionality" means more samples are needed to cover the space adequately. It also increases the number of kernel parameters and acquisition function hyperparameters, making optimization slower and more sensitive to poor choices.

Despite these limitations, Bayesian Optimization remains one of the most effective tools available for sample-efficient optimization in black-box settings [19].

Chapter 3

Methodology

The general methodology follows a structured process. A cylindrical vessel is defined with a set of fixed parameters: radius, height, total mass of uranium in solution, and the enrichment level of the uranium. This information is passed to the algorithm, which then randomly distributes the uranium mass within the cylinder. This step is repeated across multiple vessels to generate an initial population of candidate configurations. A cylinder is chosen because from a engineering perspective it is the most relevant storage unit but the algorithm can be adjusted to create e.g. a cuboid vessel. Additionally, the composition of both the dissolved fissile material and the solvent can be modified to explore other configurations of interest.

For each individual in the population, a Monte Carlo simulation is performed using Serpent. The simulation returns the k_{eff} corresponding to each configuration. This initial population forms generation one.

From this population, a predefined percentage of the most fit individuals—those with the highest k_{eff} values—are selected. These selected individuals are then combined (crossed) to produce offspring, forming the next generation. The process of selection, crossover, and evaluation via Monte Carlo simulation is repeated over multiple generations. The algorithm continues until an optimal or sufficiently high k_{eff} value is reached, indicating the most critical configuration found.

Additionally, by varying the mass of uranium in the solution while keeping the vessel dimensions constant, the algorithm can be rerun to identify the most critical distribution for each mass. This approach allows for a comparison of results across different masses. Similarly, by maintaining the mass constant and changing only the vessel dimensions, the effect of the vessel size on the critical distribution can be studied. However, not all dimensions are relevant. For instance, the estimated minimal critical diameter for an infinitely long cylinder of homogeneous water-moderated plutonium is approximately 15 cm for a concentration of 0.25 kg/L [20]. Simulating smaller diameters would therefore yield uninteresting results. Moreover, the critical mass for an aqueous ^{235}U solution in spherical geometry, optimized for maximum moderation, is approximately 784 g [21]. Therefore, using a smaller quantity of fissile material would not be suitable when aiming to achieve criticality.

3.1 Vessel segmentation

Since Serpent only allows one material definition per cell, creating a non-uniform material distribution inside the vessel requires splitting it into multiple cells, each with its own material definition. This is done by subdividing the original cylindrical vessel into smaller regions, so that different material compositions can be assigned where needed.

To make it more concrete, take the following example: if the vessel needs to have four regions from bottom to top, each with an increasing fuel density, the vessel must be sliced into four separate cells. Four material definitions would then be created, each corresponding to one of these cells. If a finer gradient in fuel density is desired, even more cells and material definitions must be added.

In short, achieving a high-resolution distribution requires many cells. To accomplish this, three types of segmentation are applied to the cylinder:

- **Axial segmentation**: The cylinder is sliced horizontally into discs at different heights.
- **Radial segmentation**: Each disc is divided into concentric rings using cylinders of increasing radii.
- **Planar segmentation**: Each ring is sliced into wedge-shaped sectors using vertical planes, similar to cutting a pizza into slices.

Figures 3.1, 3.2 and 3.3 illustrate the performed segmentations.

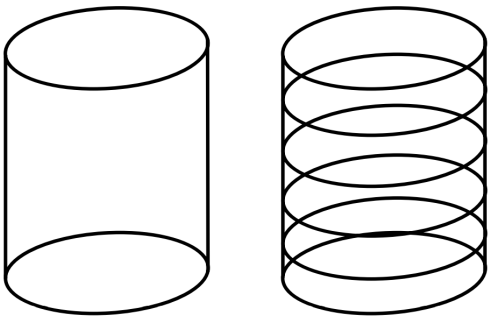


Figure 3.1: Axial segmentation of the vessel

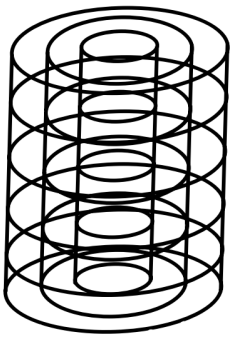


Figure 3.2: Radial segmentation of the vessel

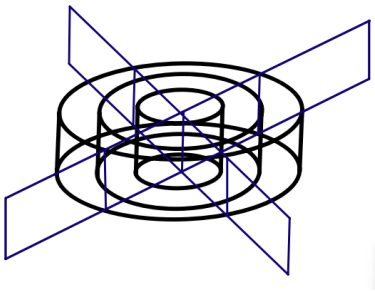


Figure 3.3: Planar segmentation of the vessel

After completing the segmentation, the vessel is divided into numerous cells, each representing a distinct segment of the structure. Figure 3.4 shows the result.

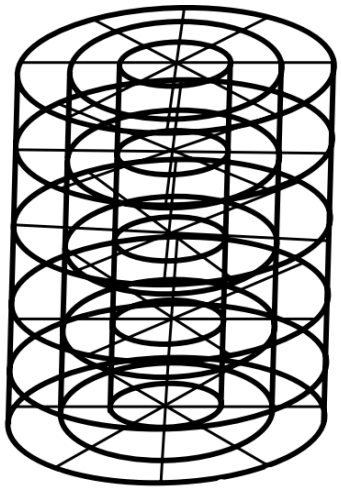


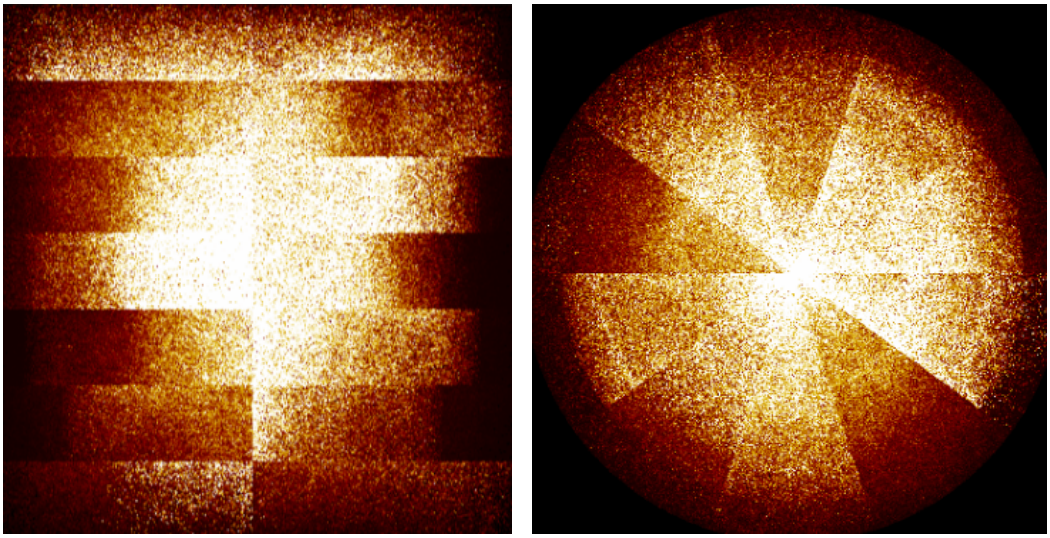
Figure 3.4: Final form of vessel divided into cells

Carefully keeping track of the geometry of each resulting cell is essential, because each material definition must have the correct mass fractions of uranium and water assigned to it. Flexibility in segmentation is also important: the number of slices can be chosen differently depending on the needed resolution. While increasing the number of segments gives a finer distribution and better accuracy, it also leads to a significant increase in computational workload. To handle this, high-performance computing resources, such as a supercomputer, will be used.

Once the segmentation scheme is established, the algorithm assigns each cell a fraction of the total uranium mass according to the defined distribution, allowing for accurate simulation in Serpent.

3.2 Fissile distributions

As previously mentioned, the individuals in the initial population are generated randomly. This randomness is crucial for maintaining diversity within the population, which reduces the risk of the algorithm becoming trapped in a local optimum. Figure 3.5 shows a random distribution that has 7 axial layers, 7 radial layers and 5 planar segmentations. This gives a total of 490 cells.

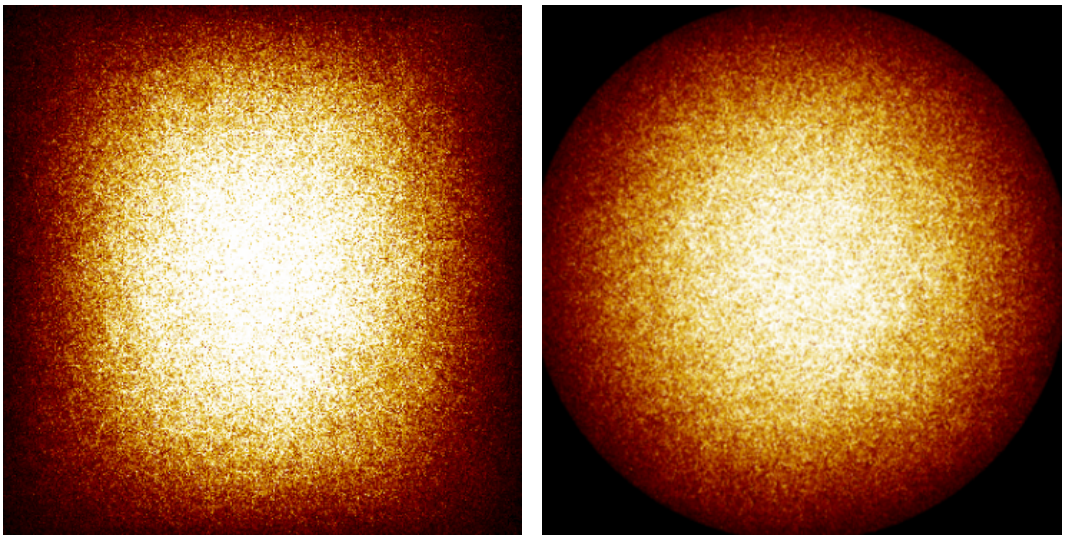


(a) Lateral view (b) Top view

Figure 3.5: Relative fission power plot of random distribution

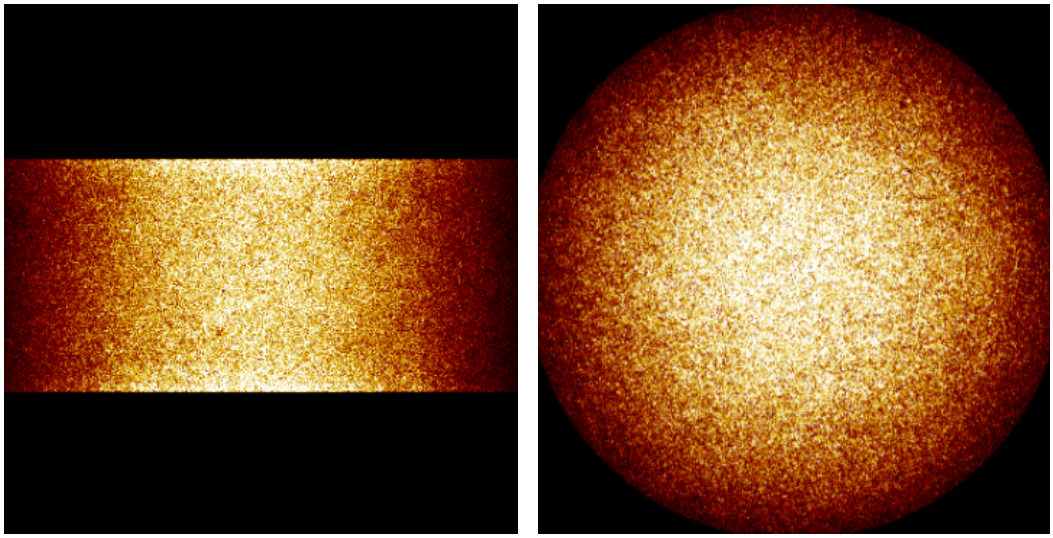
However, it is also possible to introduce intentionally biased initial distributions. These strategically chosen configurations are incorporated into the population to ensure that certain "genes" are present from the start, promoting faster convergence of the simulation.

In this context, three specific initial distributions are considered: a uniform distribution, where the uranium mass is evenly spread throughout the vessel, an axially centered distribution, where the majority of the mass is located near the axial middle of the vessel and a concentrated distribution which concentrates the mass axially and radially. Appendix A.1 contains the functions that are used to generate these initial distributions. Including these extremes provides a broader search space and increases the likelihood of identifying the most critical configuration. Figures 3.6 , 3.7 and 3.8 display the biased distributions.



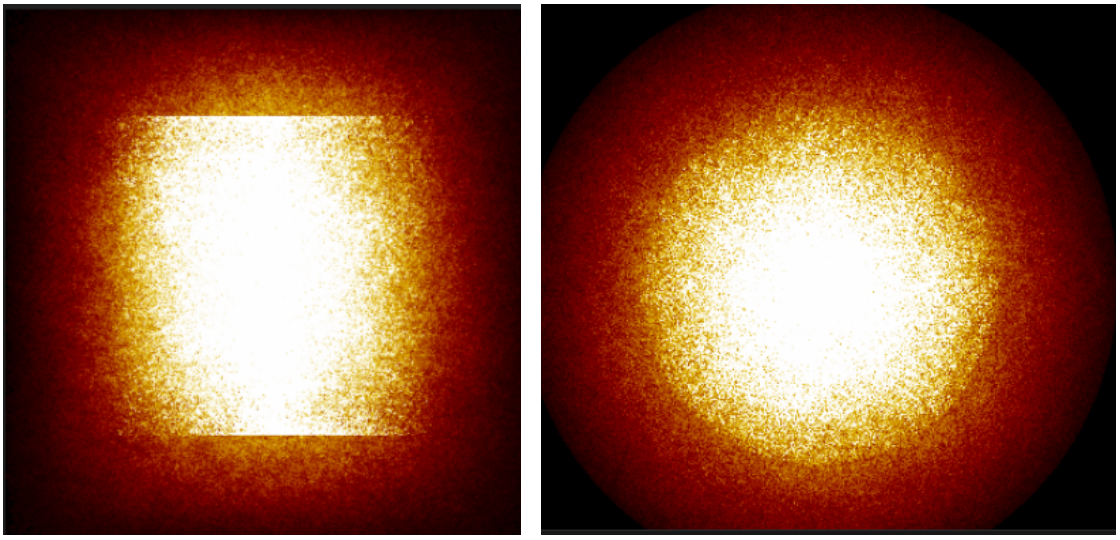
(a) Lateral view (b) Top view

Figure 3.6: Relative fission power plot of uniform distribution



(a) Lateral view (b) Top view

Figure 3.7: Relative fission power plot of axially centered distribution



(a) Lateral view (b) Top view

Figure 3.8: Relative fission power plot of concentrated distribution

3.3 Optimization Algorithm

The optimization algorithm employed in this study is a genetic algorithm, inspired by the principles of natural evolution [17]. The process begins with the generation of an initial population of individuals, each representing a unique mass distribution within the vessel.

Each individual is represented by a "DNA" array, where each element corresponds to the mass of uranium assigned to a specific cell in the geometry. The structure of this array reflects the logical ordering of the geometry: starting from the first axial layer, then iterating through each radial ring and planar slice.

Depending on the chosen distribution method—uniform, axially centered, concentrated, or random—each cell is assigned an appropriate mass value. This assignment proceeds cell by cell

until the entire DNA array is constructed. The resulting array is then used to generate the corresponding Serpent input file. Starting from cell number 1, a material definition "mat solution 1" is created which contains the mass fractions and density for that cell. This is performed for all cells in the configuration. Next, the geometry definitions are created for the cells which is dependant on the axial, radial and planar segmentations entered in the algorithm. The functions shown in Appendix A.2 calculate the dimensions of the cylinders needed for the segmentations and the parameters for the planes. Afterwards, the created cells can be assigned the corresponding material definition. Figure 3.9 shows part of a created input file.

```
/************************
* Material definitions *
************************/

mat solution1    -1.027439
92235.02c     -0.026773
92238.02c     -0.001409
1001.02c      -0.107980
8016.02c      -0.863839

mat solution2    -1.000000
92235.02c     -0.000000
92238.02c     -0.000000
1001.02c      -0.111111
8016.02c      -0.888889

mat solution3    -1.093366
92235.02c     -0.085606
92238.02c     -0.004506
1001.02c      -0.101099
8016.02c      -0.808790

mat solution4    -1.004937
92235.02c     -0.004925
92238.02c     -0.000259
1001.02c      -0.110535
8016.02c      -0.884281
```

(a) Material definitions

```
/************************
* Geometry definitions *
************************/
surf p_plane1 plane 0.0 -1.0 0 0
surf p_plane2 plane 0.5878 -0.809 0 0
surf p_plane3 plane 0.9511 -0.309 0 0
surf p_plane4 plane 0.9511 0.309 0 0
surf p_plane5 plane 0.5878 0.809 0 0
surf s0 cyl 0.0 0.0 3.6363636363636362 0.0 5.384615384615385
cell c0_sega0 0 solution1 -s0 p_plane1 p_plane5
cell c0_segb0 0 solution2 -s0 p_plane1 -p_plane2
cell c0_segc0 0 solution3 -s0 -p_plane1 p_plane2
cell c0_segd1 0 solution4 -s0 p_plane2 -p_plane3
cell c0_sege1 0 solution5 -s0 -p_plane2 p_plane3
cell c0_segd2 0 solution6 -s0 p_plane3 -p_plane4
cell c0_sege2 0 solution7 -s0 -p_plane3 p_plane4
cell c0_segd3 0 solution8 -s0 p_plane4 -p_plane5
cell c0_sege3 0 solution9 -s0 -p_plane4 p_plane5
cell c0_segx4 0 solution10 -s0 -p_plane1 -p_plane5
surf s10 cyl 0.0 0.0 7.2727272727272725 0.0 5.384615384615385
cell c10_sega0 0 solution11 -s10 p_plane1 p_plane5
cell c10_segb0 0 solution12 -s10 p_plane1 -p_plane2
cell c10_segc0 0 solution13 -s10 -p_plane1 p_plane2
cell c10_segd1 0 solution14 -s10 p_plane2 -p_plane3
cell c10_sege1 0 solution15 -s10 -p_plane2 p_plane3
cell c10_segd2 0 solution16 -s10 p_plane3 -p_plane4
cell c10_sege2 0 solution17 -s10 -p_plane3 p_plane4
cell c10_segd3 0 solution18 -s10 p_plane4 -p_plane5
cell c10_sege3 0 solution19 -s10 -p_plane4 p_plane5
cell c10_segx4 0 solution20 -s10 -p_plane1 -p_plane5
```

(b) Geometry definitions

Figure 3.9: Input file containing material and geometry definitions

After generating the input file for an individual, a Monte Carlo simulation is performed using Serpent. The output of this simulation provides the effective multiplication factor k_{eff}, which serves as the fitness value for that individual. The functions in Appendix A.3 are used to run the input file and extract k_{eff} from the output file.

Following the fitness evaluation, a fixed percentage of the fittest individuals is selected to form the next generation. These individuals are crossed to create offspring, which are then introduced as new individuals. During crossover, each cell of the new offspring inherits a fraction of material from one parent and the complementary fraction from the other parent. Typically, this fraction is set to one-half, ensuring an even contribution from both parents. This strategy preserves genetic diversity while allowing gradual convergence towards more optimal solutions. The DEAP library [22] is utilized to implement the genetic algorithm efficiently.

To maintain diversity and reduce the risk of premature convergence, mutation is introduced. Mutations occur at a predefined rate and with a specific effect size, where random modifications are applied to some offspring. For each cell in a mutated individual, a small random value—drawn from a Gaussian distribution—is either added or subtracted based on the mutation effect.

Since crossover and mutation may introduce small deviations in the total uranium mass, normalization is applied to each newly created individual. This is achieved by calculating the total mass of the offspring and scaling each cell's value such that the overall mass matches the original target. Specifically, each cell is multiplied by the ratio of (initial mass / current total mass of

the individual).

Additionally, a Bayesian optimization algorithm is integrated to enhance the search process. This algorithm records evaluated individuals and predicts optimal configurations based on acquired k_{eff} values. In each generation, one individual predicted by the Bayesian algorithm is added to the population. Due to the high dimensionality of the DNA array (often exceeding 1000 elements), the Bayesian algorithm focuses on axial distributions only. It sums the cell masses within each axial layer for every evaluated individual, thus generating predictions at the axial level. As a result, the algorithm outputs mass values for each axial layer, which are then evenly redistributed across the cells within that layer. While this approach improves prediction capability, it also reduces the axial "distribution details" of the predicted individual. Nevertheless, the additional diversity brought by these predictions benefits the optimization process.

Once all offspring are created and normalized, the new generation undergoes evaluation, and the cycle of fitness calculation, selection, crossover, mutation, and normalization continues. This iterative process is repeated for a predefined number of generations or until convergence to an optimal k_{eff} value is achieved. Appendix A.4 contains the Python code for the Genetic and Bayesian optimization algorithm.

3.4 VSC Network

The number of particles, interactions, cells, individuals, and generations needed to solve this optimization problem would likely require several weeks of computation on a standard laptop. However, since Monte Carlo simulations can be parallelized, it is possible to spread the thousands of individual simulations over multiple CPU cores [15]. This makes the use of a supercomputer essential for this project. Thanks to Hasselt University's connection to the Flemish Supercomputing Center (VSC), it is possible to run the algorithm on a supercomputing infrastructure, drastically reducing the total wall clock time and allowing for a much finer resolution of the distribution.

Chapter 4

Results

The initial development and testing of the code were carried out on a local laptop. Early tests primarily focused on verifying basic functionality and ensuring proper input/output handling. These preliminary runs used approximately 200–400 spatial cells, as higher resolutions would exceed the memory limitations imposed by Serpent on the local machine. Due to these hardware constraints, it became essential to transition to a supercomputing infrastructure to effectively analyze the algorithm's performance with larger populations, increased spatial resolution, and extended generational depth.

4.1 Analysing Supercomputer Performance

Before obtaining the final results, tests were conducted on the VSC network. These initial tests aimed to enhance the understanding of algorithm efficiency and the computational capabilities of the infrastructure.

Serpent, the primary simulation tool used, features built-in support for OpenMP, which significantly simplifies parallelization. This allows to distribute the load of computing over multiple cores which in turn decreases computing time.

To make optimal use of this feature, trial simulations were conducted using a configuration of 6000 neutrons per simulation in Serpent. The number of generations was fixed at 100 active and 20 passive generations, to get a good statistical accuracy for each individuals performance.

The spatial segmentation of the simulation was configured with 15 axial cuts, 15 radial cuts, and 15 planar cuts. Since the total number of cells is calculated as the product of the number of axial cuts, radial cuts, and twice the number of planar cuts, this setup resulted in a total of 6,750 cells. In Serpent, computational demand increases significantly with the number of cells, as the software continuously tracks the positions of neutrons throughout the simulation [15]. Consequently, increasing the resolution of the distribution directly amplifies the computational load.

To optimize computational resources, the OpenMP parallel calculation setting in Serpent was configured to use the maximum number of available cores. The request was set to "max," and 72 cores were requested from the VSC computing cluster. The computing cluster operates using job scripts to allocate resources, which requires specifying the desired computational power and

estimated runtime [23]. The simulations were executed on wICE, KU Leuven/UHasselt's latest Tier-2 cluster. This system consists of nodes equipped with two CPUs, each containing 36 cores, allowing for parallel processing with a total of 72 cores per node. The large memory capacity of 256 GB RAM per node is also essential for Serpent to handle the extensive particle tracking required during simulations [23].

During the initial setup with 6,750 cells, a simulation involving a population of 100 individuals running for 100 generations required approximately three days of continuous computation on the wICE cluster. Further testing was conducted with an increased number of cells, reaching 16,000. This higher resolution required about 210 GB of the available RAM and extended the simulation time to approximately eight days for the same number of individuals and generations. These results clearly demonstrate that increasing the resolution of the uranium distribution significantly impacts computational requirements, both in terms of memory and processing time. Balancing the resolution with the available computational resources was therefore a critical consideration in the optimization process.

4.2 Determining an Effective Algorithm Setup

To determine the optimal setup for the simulations, several parameters were carefully considered. It was essential to generate a sufficient number of results to enable meaningful comparisons. Given that the total available computation time was limited, evaluating the algorithm's wall time was a key factor.

To this end, four test runs were conducted using different configurations to assess the algorithm's performance under varying conditions. These configurations involved changes in the geometry dimensions, uranium mass, number of cells, and population size. Each simulation was allowed a maximum wall time of 70 hours, and the resulting outcomes were analyzed.

The primary metrics used for comparison were the number of generations completed within the allotted time and whether convergence was achieved. In this context, convergence refers to the algorithm's ability to discover a distribution that outperforms the initially provided biased distributions. For each setup, it was important to evaluate how effectively and efficiently the algorithm improved the configuration over time. Table 4.1 provides a summary of the parameters and results of the test runs.

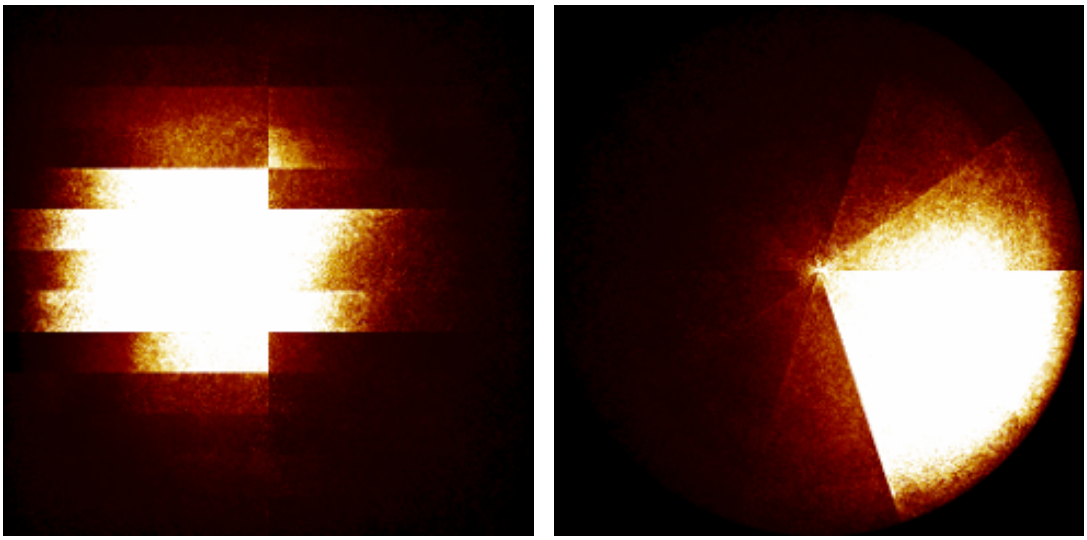
Table 4.1: Overview of test runs and their parameters

Parameter	**Run 1**	**Run 2**	**Run 3**	**Run 4**
Radius [cm]	40	40	20	10
Height [cm]	70	70	50	50
Uranium mass [g]	4000	3000	3000	4000
Number of cells	1430	3500	4000	2002
Population size	100	200	200	100
Number of generations	135	78	80	97
Computation Time [h]	70	70	70	22
Converged	yes	no	no	no

The table shows that the configuration with 1430 cells successfully converged after 135 generations. In contrast, the setups with 3500 and 4000 cells did not show improvement, even after 78 and 80 generations respectively, despite having populations twice as large. This clearly indicates that increasing the number of cells slows down convergence.

This observation is expected, as the DNA array of each individual grows with the number of defined cells. A higher resolution leads to a larger search space, making it more challenging for the algorithm to efficiently find an optimal configuration. Therefore, it becomes important to balance resolution with computational feasibility, depending on the level of detail required for the analysis.

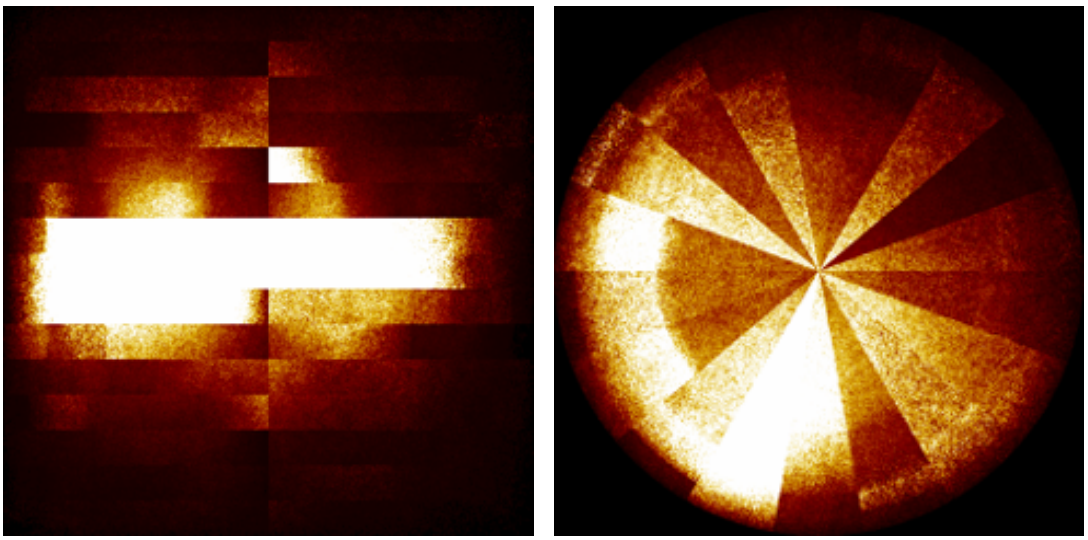
The following figures illustrate the final configurations at the end of the 70-hour wall time for runs 1, 2, and 3. Unfortunately, the fission power plots for run 4 were lost and could not be included.



(a) Lateral view (b) Top view

Figure 4.1: Relative fission power plot of test run 1

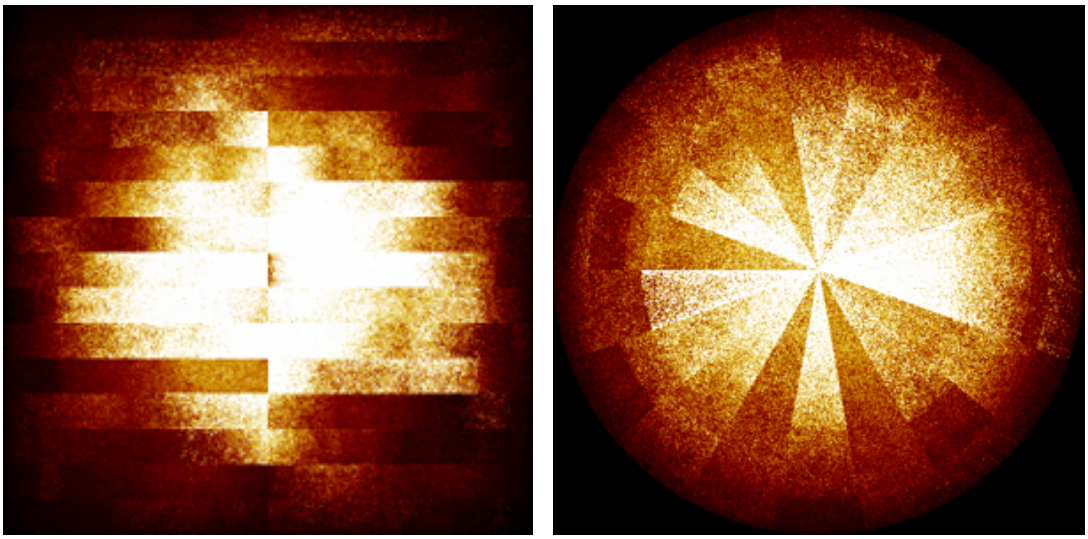
Test run 1 demonstrates convergence of the configuration toward an optimal solution, successfully identifying a distribution with a higher k_{eff} than the initial biased distributions.



(a) Lateral view (b) Top view

Figure 4.2: Relative fission power plot of test run 2

In contrast, test runs 2, 3, and 4 did not achieve such improvement. It is worth noting, however, that test run 4 was only executed for 22 hours, as it was started at a later time. While test runs 1 and 2 have similar configurations, they would be expected to yield comparable results. However, as shown in Figure 4.2, this is not the case.



(a) Lateral view (b) Top view

Figure 4.3: Relative fission power plot of test run 3

Although the increase in resolution across the runs is evident, it was determined that Run 1 provides a sufficiently detailed resolution for producing meaningful results. Its fast convergence is the most critical factor when designing an effective setup.

It is also worth noting that the algorithm can be executed for different configurations simultaneously. This can be achieved by duplicating the algorithm into separate directories and modifying the setup parameters for each instance. In this way, multiple runs can proceed in parallel from different working directories.

With this parallel approach in mind, the configuration for an efficient algorithm setup was defined. The number of cells selected for the configuration is 2002, calculated as the product of the number of axial slices, radial slices, planes, and a factor of 2. Table 4.2 summarizes the selected parameters for the algorithm.

Table 4.2: Overview of the selected algorithm setup

Parameter	**Value**
Population size	200
Number of generations	200
Number of cells	2002
Mutation rate	0.2
Mutation effect	0.2
Crossover blend	0.5
Number of axial slices	13
Number of radial slices	11
Number of planes	7
Neutron generation size	6000
Passive generations in Serpent	60
Active generations in Serpent	20

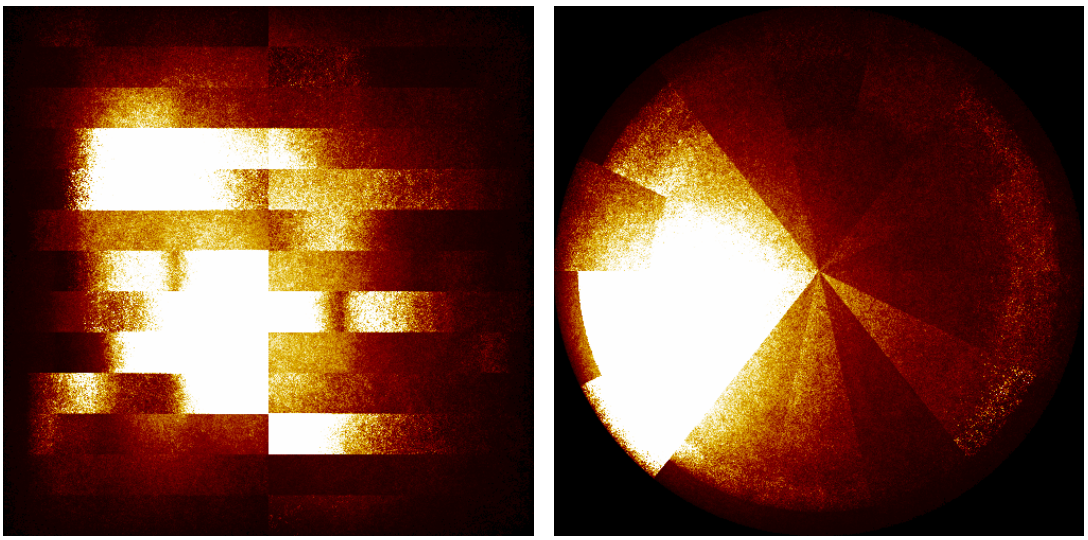
4.3 Results for Initial Two Configurations

To ensure sufficient computation time, the maximum estimated wall time was approximately four days. Consequently, a wall time of five days was requested from the Slurm workload manager to provide a safety margin. Two algorithm configurations were submitted for simulation, as summarized in Table 4.3.

Table 4.3: Setup of initial two configurations

Parameter	**Configuration 1**	**Configuration 2**
Cylinder radius [cm]	40	40
Cylinder height [cm]	70	70
Uranium mass [g]	3000	2000
Enrichment	0.95	0.95
Population size	200	200

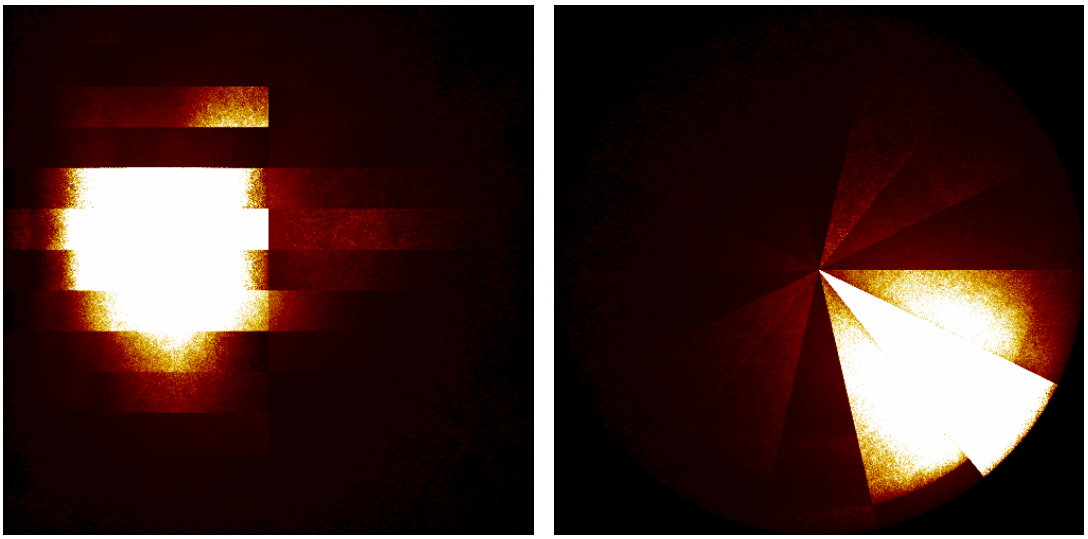
The following figures present the results of the two initial configurations, along with their respective results.



(a) Lateral view ($k_{eff} = 0.80338$) (b) Top view ($k_{eff} = 0.80338$)

Figure 4.4: Result for setup – 40cm × 70cm, 3000g, 2002 cells, 123 gens

This configuration completed 123 generations within the five-day wall time. Beyond this point, the algorithm was unable to discover a distribution with a higher k_{eff} than the initially provided, biased axially concentrated distribution ($k_{eff} = 1.00433$).



(a) Lateral view ($k_{eff} = 0.920051$) (b) Top view ($k_{eff} = 0.920051$)

Figure 4.5: Most critical distribution – 40cm × 70cm, 2000g, 2002 cells, 109 gens

This configuration setup reached 109 generations after 5 days of wall time with a population of 200 individuals. The algorithm was able to identify a distribution with a higher k_{eff} than the biased distributions which were added initially.

Despite the estimated wall time of four days for running 200 generations with a population size of 200 individuals, neither of the two configurations completed all 200 generations within the five-day Slurm allocation. One configuration failed to progress beyond 110 generations. This indicated that something was significantly slowing down the algorithm.

One likely cause identified was the way simulation output from Serpent was handled. During execution, the algorithm redirected console output to a continuously growing text file stored on the VSC Data nodes. Over the span of several hours or days, these files could grow to several gigabytes in size. Writing to such large files—especially if new output had to be appended to specific positions—may have introduced substantial I/O overhead, thereby slowing down the overall progress of the algorithm.

To address this issue, two changes were introduced in the subsequent simulations. First, the console output was disabled to eliminate potential performance degradation due to large output file sizes. Second, the population size was reduced to 100 individuals in order to accelerate convergence and reduce total runtime.

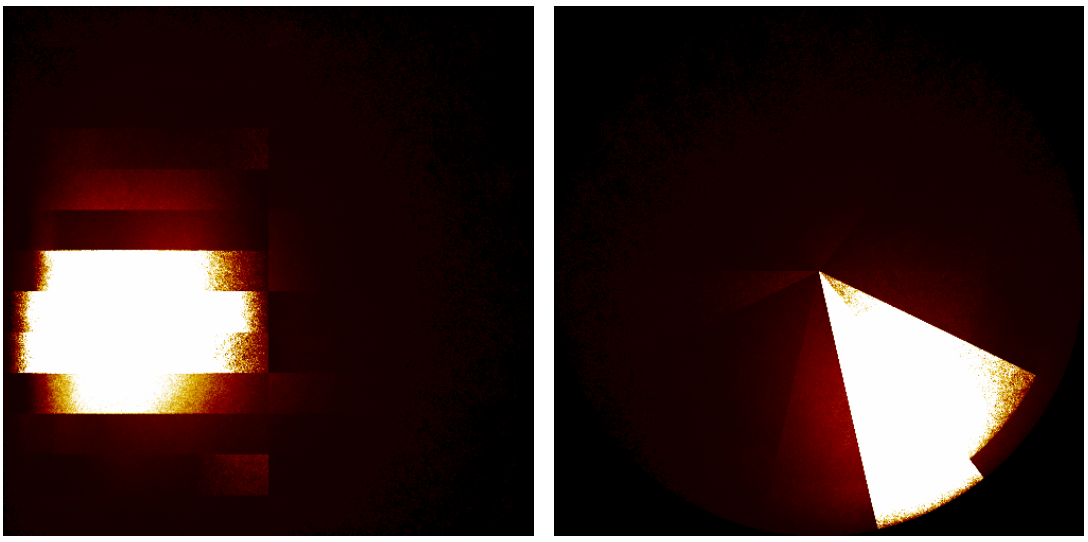
4.4 Results from Adjusted Algorithm Setup

As previously mentioned the following results were obtained with a smaller population size, 3 setups were put into the algorithm to analyse the effect of varying vessel dimensions and mass. Table 4.4 shows the 3 configuration setups.

Table 4.4: Adjusted algorithm: setup of three test configurations

Parameter	**Configuration 1**	**Configuration 2**	**Configuration 3**
Cylinder radius [cm]	40	20	20
Cylinder height [cm]	70	50	50
Uranium mass [g]	1000	2000	3000
Enrichment	0.95	0.95	0.95
Population size	100	100	100

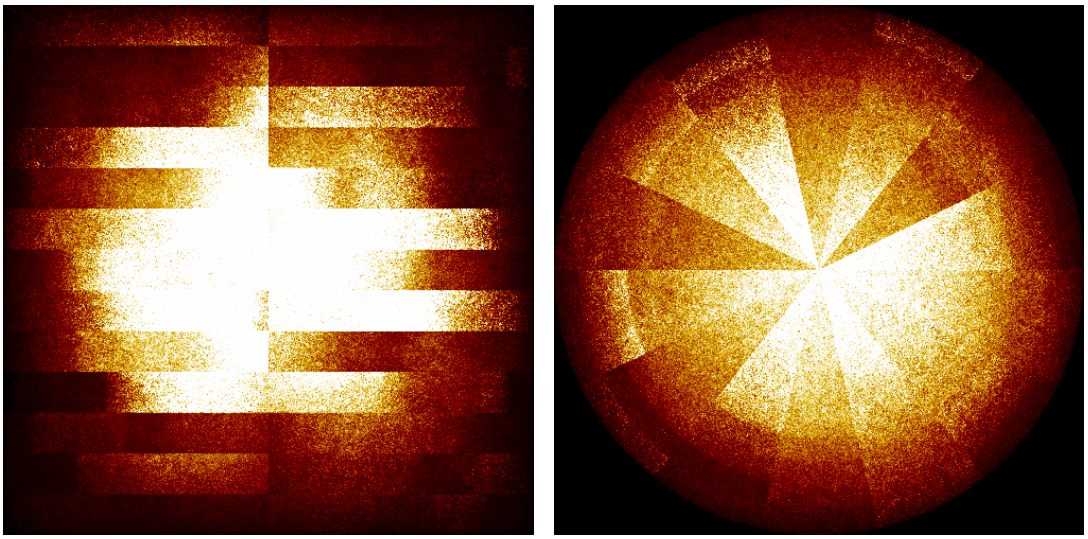
A wall time of four days was requested on the computing cluster via the Slurm workload manager. The following three figures present the results for the three configurations generated using the adjusted algorithm setup.



(a) Lateral view ($k_{eff} = 0.766168$) (b) Top view ($k_{eff} = 0.766168$)

Figure 4.6: Most critical distribution – 40cm × 70cm, 1000g, 2002 cells, 220 gens

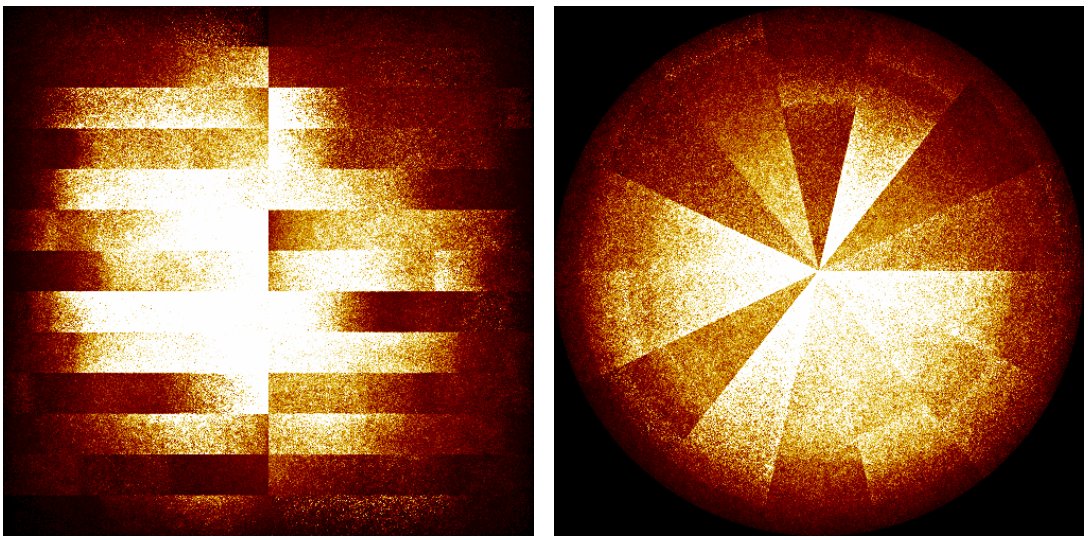
The algorithm was able to identify a distribution with a higher k_{eff} than the initially biased distributions. However, this solution remains far from optimal, as the resulting k_{eff} is still significantly below 1. Given that a mass of 1000 g of uranium enriched to 95% should, in principle, be sufficient to reach criticality, this suggests that additional generations would have been required for the algorithm to converge towards the most critical configuration.



(a) Lateral view ($k_{eff} = 0.879674$) (b) Top view ($k_{eff} = 0.879674$)

Figure 4.7: Result for setup – 20cm × 50cm, 2000g, 2002 cells, 200 gens

Here, the algorithm was not able to discover a configuration with a higher k_{eff} than the initially added homogeneous distribution ($k_{eff} = 0.995007$).



(a) Lateral view ($k_{eff} = 0.982272$) (b) Top view ($k_{eff} = 0.982272$)

Figure 4.8: Result for setup – 20cm × 50cm, 3000g, 2002 cells, 186 gens

In this case as well, the algorithm did not succeed in identifying a configuration with a higher k_{eff} than the initially provided homogeneous distribution ($k_{eff} = 1.09308$).

These results lack improvement within the projected wall time and are far from the optimal solution. So final adjustments were made to the algorithm to get optimal results quickly and

efficiently. The neutron generation size was adjusted to 5000 neutrons, furthermore the total number of cells per distribution was decreased from 2002 cells to 1430 cells. This setup is intended to facilitate convergence toward an optimal solution. Furthermore, the number of generations was not constrained; only the total runtime was limited. Table 4.5 summarizes the adjustments.

Table 4.5: Overview of the fine-tuned algorithm setup

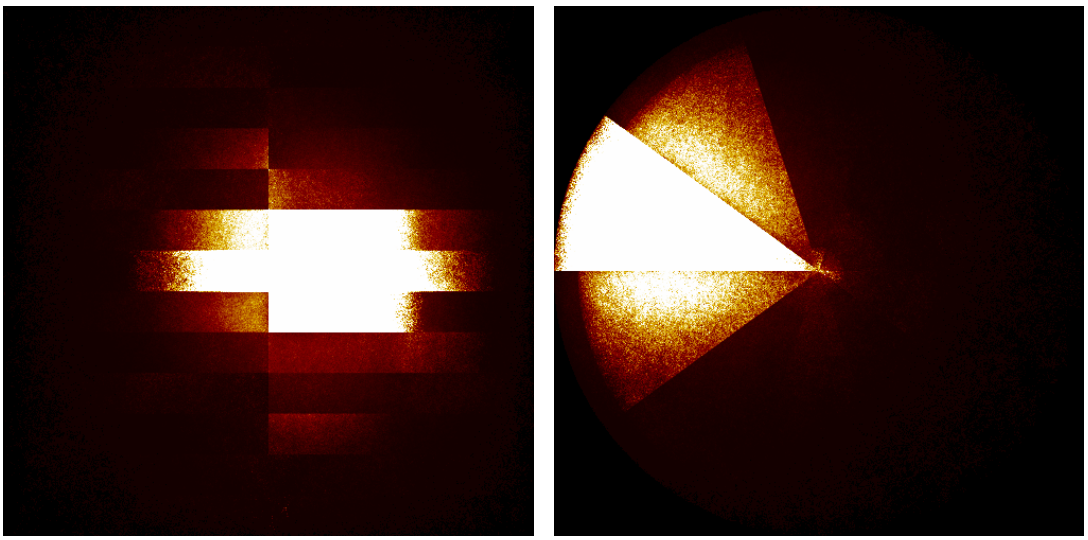
Parameter	**Value**
Population size	100
Number of generations	Limited to timeframe
Number of cells	1430
Number of axial slices	13
Number of radial slices	11
Number of planes	5
Neutron generation size	5000

4.5 Results from the Fine-Tuned Algorithm Setup

Because of time and resource limitation, only a selected amount of configurations could be simulated on the supercomputer cluster with the fine-tuned setup. Four configurations which were thought to be interesting were selected. Each configuration was allocated a wall time of five days via the Slurm workload manager, Appendix A.5 shows an example of a jobscript. Table 4.6 presents the the different configurations.

Table 4.6: Setup of final four configurations

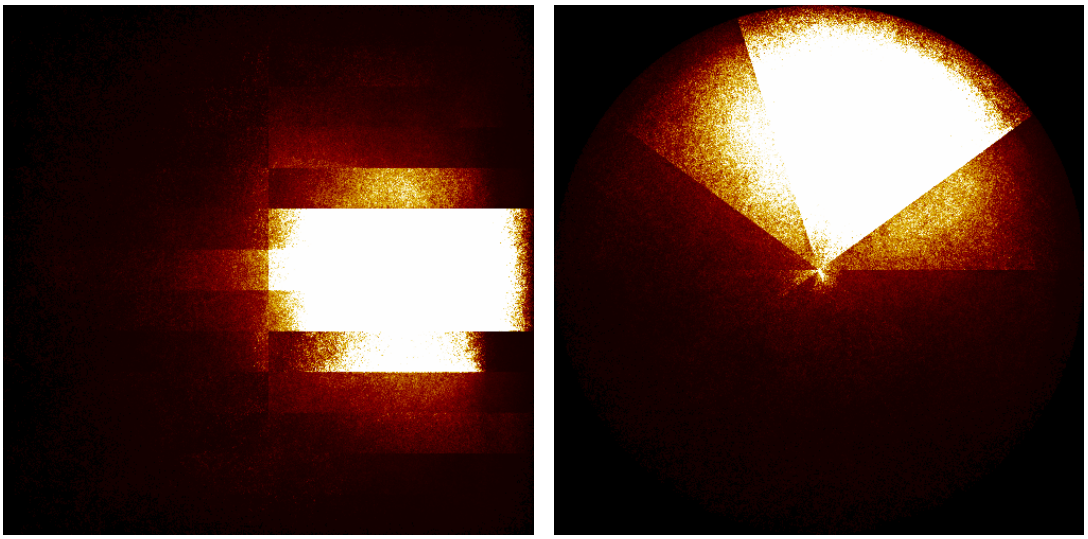
Parameter	**Configuration 1**	**Configuration 2**	**Configuration 3**	**Configuration 4**
Cylinder radius [cm]	40	40	20	20
Cylinder height [cm]	70	70	50	50
Uranium mass [g]	1500	3000	1500	3000
Enrichment	0.95	0.95	0.95	0.95
Population size	100	100	100	100



(a) Lateral view ($k_{eff} = 0.968386$) (b) Top view ($k_{eff} = 0.968386$)

Figure 4.9: Most critical distribution – 40cm × 70cm, 1500g, 1430 cells, 242 gens

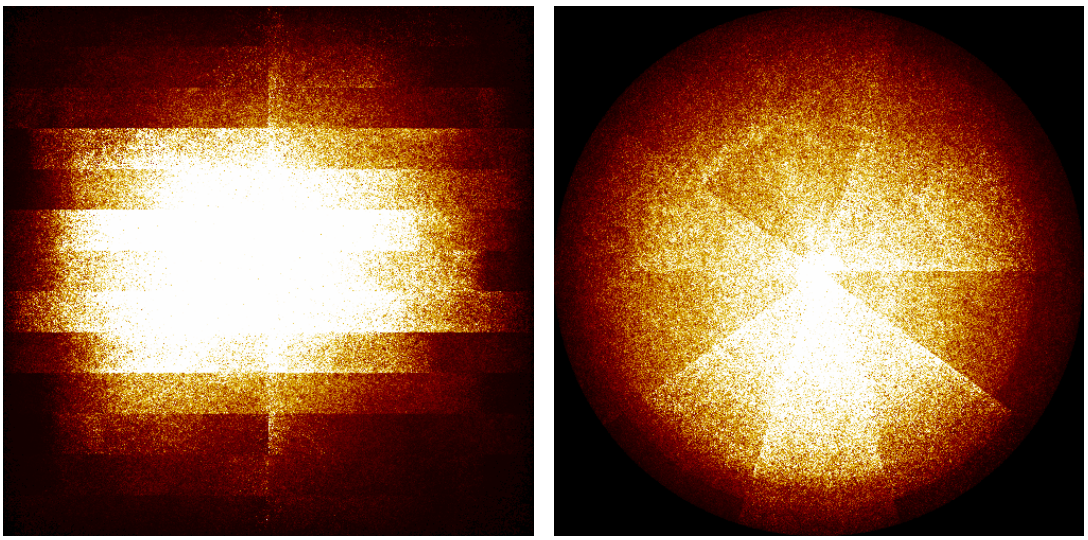
The algorithm successfully identified a distribution with a higher level of criticality than the initially biased configurations. The fissile mass appears to be concentrated axially near the center of the vessel. Figure 4.9(b) shows a concentration of material in the corner, distributed over three sections, with the highest relative fission power observed in the central region. Nonetheless, the fact that the resulting $k_{\text{eff}} = 0.968386$ remains below 1 indicates that this is not yet the most critical configuration.



(a) Lateral view ($k_{eff} = 1.08485$) (b) Top view ($k_{eff} = 1.08485$)

Figure 4.10: Most critical distribution – 40cm × 70cm, 3000g, 1430 cells, 180 gens

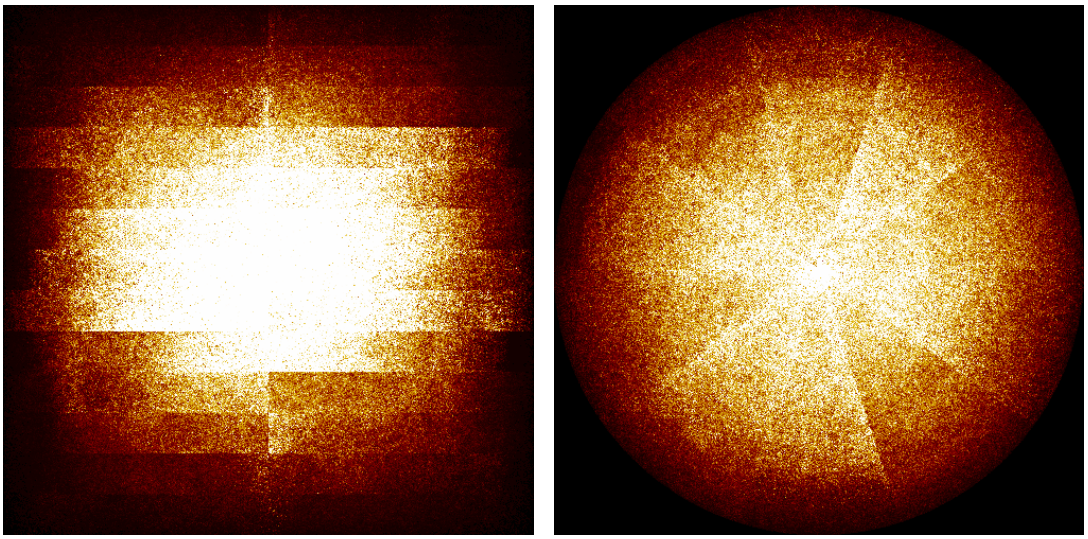
This vessel contains twice the amount of fissile material compared to the previous configuration. The algorithm identified a distribution with axial concentration at the center and angular concentration over four sections, resulting in the most critical configuration found. The relative fission power is equally high in the two central sections. Although criticality was achieved, the relatively low number of generations (180) suggests that the optimization process may not have fully converged and that further improvement of k_{eff} could still be possible.



(a) Lateral view ($k_{eff} = 0.917669$) (b) Top view ($k_{eff} = 0.917669$)

Figure 4.11: Results for setup – 20cm × 50cm, 1500g, 1430 cells, 228 gens

The results for this setup indicate a concentration of fissile material at the center of the vessel, both axially and radially. However, this configuration was not more critical than the initially biased axially concentrated distribution, which yielded a $k_{\text{eff}} = 0.955383$. This suggests that the algorithm required additional generations to identify a more critical configuration. Moreover, the distribution exhibits no angular dependence, as the fissile material is uniformly spread across all angles. The fact that the resulting k_{eff} remains below 1 confirms that criticality was not achieved. Given that 1500 g of fissile material should, in principle, be sufficient to reach criticality, further optimization of the spatial distribution is necessary.



(a) Lateral view ($k_{eff} = 1.10611$) (b) Top view ($k_{eff} = 1.10611$)

Figure 4.12: Most critical distribution – 20cm × 50cm, 3000g, 1430 cells, 232 gens

The same observations apply to this distribution: it exhibits an even angular spreading and is centrally concentrated both axially and radially. With a resulting $k_{\text{eff}} \approx 1.11$, the configuration is clearly supercritical. However, despite containing twice the amount of fissile material compared

to the previous setup, no significant differences in the spatial distribution of the fissile mass are observed.

Chapter 5

Discussion

5.1 Algorithm Performance and Key Findings

The results demonstrate that the genetic algorithm generally functions as intended. In multiple configurations, it successfully identified spatial distributions yielding a higher k_{eff} than those of the initially biased input configurations. However, criticality was not achieved across all runs, even when the fissile mass should theoretically have been sufficient. This inconsistency suggests that convergence to the global optimum was not always reached within the allocated wall time.

This outcome highlights a common trade-off in heuristic optimization: balancing convergence speed with solution quality. In time-limited scenarios, the GA may converge prematurely to locally optimal solutions, particularly for configurations that require a larger number of generations to explore the solution space effectively.

5.2 Impact of Geometry and Mass

The spatial distribution of fissile material was found to depend on both the geometry of the vessel and the amount of fissile mass. This dependency is particularly evident in Figures 4.9 and 4.10, where doubling the mass leads to a noticeable difference in the distribution. The number of cells with a high relative fission power increases with increasing fissile mass. These configurations also suggest that the most critical distribution tends to concentrate in a corner of the vessel for those dimensions, indicating strong asymmetry.

In contrast, Figures 4.11 and 4.12 show minimal changes in distribution with increasing mass. However, since the configuration in Figure 4.11 does not represent the most critical distribution for that setup, no definitive conclusions can be drawn. Interestingly, both of these distributions exhibit axial and radial centralization, along with uniform angular spreading around the vessel.

To draw more reliable conclusions about the influence of vessel geometry, fissile mass, and enrichment on the spatial distribution and criticality, a broader range of configurations must be simulated over a greater number of generations.

5.3 Computational and Methodological Limitations

One of the most significant constraints was the long wall time required to obtain near-optimal solutions. The use of a relatively small population (100 individuals) limited the GA's ability to explore the solution space thoroughly. In general, larger populations (e.g., 1000+) enhance exploration and improve convergence robustness but require substantially more computational resources.

A further limitation was the resolution of the spatial model. Although using more cells to discretize the vessel geometry would yield more accurate spatial distributions, it also dramatically increases the dimensionality of the search space. Consequently, the number of generations and the computational cost per generation rise significantly, since each fitness evaluation involves a full Serpent simulation. While access to high-performance computing infrastructure was available, practical issues such as stalled runs, incorrect inputs/outputs, and inaccurate wall time predictions often disrupted the workflow.

Learning to operate efficiently in an HPC environment, while simultaneously tuning the algorithm, posed a steep challenge. Additionally, interpolation methods used to predict maximum wall times for job scheduling frequently proved unreliable.

Despite these constraints, considerable improvements were made during the development phase, particularly in optimizing simulation speed and reducing computational overhead. Additional performance gains could be realized through more effective parallelization strategies—such as distributing the evaluation of individuals across multiple cores within a single generation.

5.4 Bayesian Optimization and Algorithm Parameters

While a Bayesian Optimizer was integrated into the framework, it was not extensively tuned. Fine-tuning the BO's exploration-exploitation balance could improve convergence rates. Additionally, the current BO implementation generates predictions only on the axial level before renormalization, which limits its utility when the optimal solution exhibits non-uniform axial behavior. Preserving detail in axial segments—rather than collapsing it during normalization—may improve the performance of the BO.

Beyond the BO, several algorithmic parameters were not optimized but could substantially influence convergence. These include:

- Mutation rate and effect size
- Crossover blend strategies
- Mutation function types
- Number of parents per offspring (e.g., using more than two)
- Selection mechanisms (e.g., tournament, rank-based, roulette-wheel)

Conducting a systematic parameter sensitivity study could yield faster and more reliable convergence.

5.5 Future Improvements and Practical Considerations

To enhance practicality, future implementations must aim to reduce required wall time. One potential strategy is to adapt Serpent settings over time—for instance, starting with smaller neutron generation sizes and lower active cycle counts in early GA generations, where highly accurate keff estimates are not yet needed. This would reduce simulation time for suboptimal individuals early in the search process.

Another strategy is progressive segmentation. Starting with coarse segmentation allows the algorithm to converge to a broadly optimal region of the search space. Finer segmentation can then be introduced in later generations to refine the solution.

Also, future implementations could benefit from initializing the population with distributions derived from prior simulations. These informed starting points, if matched appropriately to the configuration, may help the algorithm bypass unproductive regions of the solution space and accelerate convergence toward critical distributions.

The current setup also supports practical extension to different configurations. The type of fissile material and solvent can be easily changed, offering flexibility in criticality studies.

Future research should focus on streamlining the simulation-optimization pipeline, improving the robustness of job execution, and developing a set of best practices for parameter tuning. A general, adaptable algorithm setup would be beneficial for applying this methodology across various fissile systems.

Chapter 6

Conclusion

This study demonstrates the potential of a genetic algorithm, coupled with Serpent simulations, to optimize the spatial distribution of fissile material in aqueous systems for criticality analysis. In several configurations, the algorithm was able to discover distributions with higher k_{eff} than the initial biased inputs, confirming its basic functionality and effectiveness. However, consistent achievement of criticality or the most critical distribution was not guaranteed, highlighting the importance of convergence speed and solution quality in heuristic optimization, especially under time constraints.

The geometry of the vessel and the amount of fissile mass both influenced the resulting spatial distribution. While some configurations showed clear trends—such as asymmetric concentration or mass-dependent spreading—others did not, suggesting that geometry may play a more dominant role than mass in certain setups. These findings emphasize the need for further simulations across a broader range of configurations and enrichment levels to draw more robust conclusions.

Computational limitations posed significant challenges throughout the study. A relatively small population size and modest spatial resolution restricted the algorithm's exploratory power, while each fitness evaluation's dependence on a full Serpent simulation imposed high computational costs. Despite access to high-performance computing infrastructure, practical issues—such as job failures, wall time estimation errors, and input/output misconfigurations—hindered workflow efficiency.

Nonetheless, several improvements were successfully implemented, including enhanced simulation speed and reduced overhead. Additional gains could be realized through finer-grained parallelization and smarter job management. Although Bayesian Optimization was integrated, its tuning and architecture require further refinement to fully benefit from its potential. Similarly, key algorithmic parameters—such as mutation rate, selection strategy, and crossover blending—remain unexplored and may offer opportunities for performance optimization.

Looking forward, a number of enhancements could make the algorithm more practical and efficient. These include adaptive Serpent settings based on GA progression, dynamic segmentation strategies, and smarter initialization using distributions from previous simulations. Such approaches could significantly reduce wall time while maintaining or improving convergence reliability.

Importantly, the framework developed here is flexible and extensible. The type of fissile material

and solvent can be easily adjusted, enabling its application to a wide range of criticality scenarios. To maximize its utility, future research should focus on streamlining the simulation–optimization pipeline, improving robustness in HPC environments, and establishing a generalizable setup for parameter tuning. With these refinements, the methodology has the potential to become a valuable tool for investigating and optimizing fissile systems.

Bibliography

[1] T. P. McLaughlin, S. P. Monahan, N. L. Pruvost, V. V. Frolov, B. G. Ryazanov, and V. I. Sviridov, "A Review of Criticality Accidents: 2000 Revision," Tech. Rep. LA-13638, Los Alamos National Laboratory, 2000.

[2] S. Tsuchiya, A. Tanabe, T. Narushima, K. Ito, and K. Yamazaki, "An analysis of Tokaimura nuclear criticality accident: A systems approach," in *The 19th International Conference of the System Dynamics Society*, (Atlanta, Georgia), 2001.

[3] NRC, *Nuclear Engineering: Module 4 - Criticality Safety [course].* Chattanooga, USA: USNRC Technical Training Center, 2012.

[4] Atomic Archive, "Nuclear Fission: Basics," Available: https://www.atomicarchive.com/science/fission/index.html [Accessed: 19 May 2025].

[5] N. L. Pruvost and H. C. Paxton, "Nuclear Criticality Safety Guide," Tech. Rep. LA-12808, Los Alamos National Laboratory, 1996.

[6] G. Van den Eynde, *Reactor Physics [course].* SCK CEN, Faculteit Industriële Ingenieurswetenschappen KU Leuven & UHasselt, 2025.

[7] S. Pirani, *Study of the Superconducting Medium Beta Cavity of the European Spallation Source [PhD thesis].* Lund, Sweden: Lund University, 2020.

[8] International Atomic Energy Agency, *Basic Professional Training Course on Nuclear Safety: Module I – Nuclear Physics and Nuclear Reactor Principles.* Vienna, Austria: IAEA, 2015.

[9] E. D. Clayton, A. W. Prichard, B. E. Durst, D. Erickson, and R. J. Puigh, "Anomalies of Nuclear Criticality, Revision 6," Tech. Rep. PNNL-19176, Pacific Northwest National Laboratory, Richland, Washington, 2010.

[10] W. M. Stacey, *Nuclear Reactor Physics.* Atlanta, GA: John Wiley & Sons, 2018.

[11] N. Soppera, M. Bossant, and E. Dupont, "JANIS 4: An Improved Version of the NEA Java-based Nuclear Data Information System," *Nuclear Data Sheets*, vol. 120, p. 294, 2014.

[12] J.-P. Pouget and S. Mather, "General Aspects of the Cellular Response to Low- and High-LET Radiation," *European Journal of Nuclear Medicine*, vol. 28, pp. 541–561, 2001.

[13] C. O. Brown and R. D. Carter, "Reanalysis of Idealized Plutonium Dissolvers and the "Always Safe" Conditions," Tech. Rep. ARH-LD–109, Atlantic Richfield Hanford Co., Richland, WA, 1975.

[14] N. L. Pruvost and H. C. Paxton, "Critical Dimensions of Systems Containing U-235, Pu-239, and U-233," Tech. Rep. LA-10860-MS, Los Alamos National Laboratory, 1987.

[15] VTT Technical Research Centre of Finland, "Installing and Running Serpent - Parallel Calculation using MPI," Available: https://serpent.vtt.fi/mediawiki/index.php [Accessed: 7 May 2025].

[16] W. Haeck, *An Optimum Approach to Monte Carlo Burn-Up [PhD thesis]*. Gent, Belgium: UGent - Gent University, 2007.

[17] S. Forrest, "Genetic algorithms," *ACM Comput. Surv.*, vol. 28, p. 77–80, 1996.

[18] V. Podgorelec, J. Brest, and P. Kokol, "Power of Heterogeneous Computing as a Vehicle for Implementing E3 Medical Decision Support Systems," *Studies in Health Technology and Informatics*, vol. 68, pp. 703–708, 1999.

[19] E. Brochu, V. M. Cora, and N. de Freitas, "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning," *CoRR*, vol. abs/1012.2599, p. 49, 2010.

[20] D. G. Bowen and R. D. Busch, "Hand Calculation Methods for Criticality Safety – A Primer," Tech. Rep. LA-14244-M, Los Alamos National Laboratory, 2006.

[21] European Nuclear Society, "Critical Mass," Available: https://www.euronuclear.org/glossary/critical-mass/ [Accessed: 19 May 2025].

[22] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary Algorithms Made Easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, 2012.

[23] Vlaams Supercomputer Centrum (VSC), "wICE Hardware - VSC Documentation," Available: `https://docs.vscentrum.be/leuven/tier2_hardware/wice_hardware.html` [Accessed: 7 May 2025.

Appendix A

Code Listings

A.1 Libraries, input parameters and functions to create distributions

```
import subprocess
import re
import math
import random
import numpy as np
import time
import shutil
from deap import base, creator, tools
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern
from deap import algorithms


# === Cylinder Parameters ===
Cilinder_radius = 40    # cm
Cil_height = 70         # cm
Uranium_enrichment = 0.95 # Enrichment
Massa_U =   1500      # g
density_U = 19.1        # g/cm^3

# Radial, axial and planar segmentation
n_radial =  11  # Segmentation of radial shells
n_axial =   13 # Segmentation of axial layers
n_planes =  5  # Segmentation is two times the amount of added planes!

# Genetic Algorithm Parameters
population_size =  100   #Genetic Algorithm population size
generations = 500        # Iteration of generations
mutation_rate = 0.20     # Mutation rate
mutationeffect = 0.20    # Mutation effect
```

```
crossover_blend = 0.5    # fraction of parents material during crossover

def planar_slices(n_planes):
    """Generates coefficients for planar slices dividing the cylinder."""
    angles = np.linspace(0, np.pi, n_planes, endpoint=False)
    slices = []
    for angle in angles:
        A = math.sin(angle)
        B = -math.cos(angle)
        C = 0
        D = 0
        slices.append((A, B, C, D))
    return slices

def segment_volume(r_min, r_max, segment_height):
    """Calculates the volume of a cylindrical segment."""
    return math.pi * (r_max**2 - r_min**2) * segment_height


def generate_homogeneous_fuel():
    """Generates a homogeneous uranium distribution."""
    radial_edges = np.linspace(0, Cilinder_radius, n_radial + 1)
    axial_edges = np.linspace(0, Cil_height, n_axial + 1)
    segment_volumes = [segment_volume(radial_edges[j], radial_edges[j+1], axial_edges[i+1] - axial_edges[i])
                       for i in range(n_axial) for j in range(n_radial)]
    total_volume = sum(segment_volumes)
    segment_densities = [Massa_U / total_volume] * len(segment_volumes)
    masses = [density * volume for density, volume in zip(segment_densities, segment_volumes)]
    slicedmass = []

    # Splits each mass into n_planes parts
    for mass in masses:
        for _ in range(2*n_planes):
            slicedmass.append(mass / (2 * n_planes))
    return slicedmass

def generate_random_distribution():
    """Generates a random uranium distribution."""
    while True:
        segment_fractions = np.random.dirichlet(np.ones(n_axial * n_radial))
        fuel_segments = segment_fractions * Massa_U
        radial_edges = np.linspace(0, Cilinder_radius, n_radial + 1)
        axial_edges = np.linspace(0, Cil_height, n_axial + 1)
```

```
74          valid_distribution = True
75
76          for i in range(n_axial):
77              for j in range(n_radial):
78                  r_min, r_max = radial_edges[j], radial_edges[j+1]
79                  z_min, z_max = axial_edges[i], axial_edges[i+1]
80                  segment_height = z_max - z_min
81                  cell_volume = segment_volume(r_min, r_max,
                        segment_height)
82                  uranium_volume = fuel_segments[i * n_radial + j] /
                        density_U
83                  if uranium_volume > cell_volume:
84                      valid_distribution = False
85                      break
86              if not valid_distribution:
87                  break
88
89          if valid_distribution:
90              fuel_segments *= Massa_U / np.sum(fuel_segments)
91          slicedmass = []
92          for mass in fuel_segments:
93              random_fractions = np.random.dirichlet(np.ones(2 * n_planes
                    ))
94              for fraction in random_fractions:
95                  slicedmass.append(mass * fraction)
96          return slicedmass
97
98
99  def generate_cilinder_distribution():
100     """Generates axially centered uranium distribution in the cylinder.
            """
101     radial_edges = np.linspace(0, Cilinder_radius, n_radial + 1)
102     axial_edges = np.linspace(0, Cil_height, n_axial + 1)
103     segment_masses = []
104
105     for i in range(n_axial):
106         for j in range(n_radial):
107             r_min, r_max = radial_edges[j], radial_edges[j + 1]
108             z_min, z_max = axial_edges[i], axial_edges[i + 1]
109             segment_height = z_max - z_min
110             volume = segment_volume(r_min, r_max, segment_height)
111             segment_masses.append((volume * density_U))
112
113     total_mass = sum(segment_masses)
114     sphere_volume = Massa_U / density_U
115     sphere_radius = (sphere_volume / (4/3 * math.pi))**(1/3)
116
117
118     # Finds the closest index to the radius of the sphere
```

```
    sphere_height_up = Cil_height / 2 + sphere_radius
    sphere_height_down = Cil_height / 2 - sphere_radius
    i_min = sorted(range(len(axial_edges)), key=lambda i: abs(
       axial_edges[i] - sphere_height_down))[0]
    i_max = sorted(range(len(axial_edges)), key=lambda i: abs(
       axial_edges[i] - sphere_height_up))[0]
    jrad = sorted(range(len(radial_edges)), key=lambda i: abs(
       radial_edges[i] - sphere_radius))[0]

    if total_mass > Massa_U:
        excess_mass = total_mass - Massa_U
        # Loops radially from outside to inside, but only over the half
            of the segments
        for i in range(n_axial):
            for j in range(n_radial):
                if i < i_min -1 or i > i_max or j > jrad +1:
                    index = i * n_radial+ j
                    if segment_masses[index] <= excess_mass:
                        excess_mass -= segment_masses[index]
                        segment_masses[index] = segment_masses[index] *
                            0.05
                    else:
                        segment_masses[index] -= excess_mass
                        excess_mass = 0

            if excess_mass <= 0:
                break
        slicedmass = []

    # Splits each mass into n_planes parts
    for mass in segment_masses:
        for _ in range(2*n_planes):
            slicedmass.append(mass / (2*n_planes))
    return slicedmass

def generate_cilinder_distribution2():
    """Generates concentrated uranium distribution in the cylinder."""
    radial_edges = np.linspace(0, Cilinder_radius, n_radial + 1)
    axial_edges = np.linspace(0, Cil_height, n_axial + 1)
    segment_masses = []

    for i in range(n_axial):
        for j in range(n_radial):
            r_min, r_max = radial_edges[j], radial_edges[j + 1]
            z_min, z_max = axial_edges[i], axial_edges[i + 1]
            segment_height = z_max - z_min
            volume = segment_volume(r_min, r_max, segment_height)
            segment_masses.append((volume * density_U))

```

```
    total_mass = sum(segment_masses)
    sphere_volume = Massa_U / density_U
    sphere_radius = (sphere_volume / (4/3 * math.pi))**(1/3)


    # Finds the closest index to the radius of the sphere
    sphere_height_up = Cil_height / 2 + sphere_radius
    sphere_height_down = Cil_height / 2 - sphere_radius
    i_min = sorted(range(len(axial_edges)), key=lambda i: abs(
       axial_edges[i] - sphere_height_down))[0]
    i_max = sorted(range(len(axial_edges)), key=lambda i: abs(
       axial_edges[i] - sphere_height_up))[0]
    jrad = sorted(range(len(radial_edges)), key=lambda i: abs(
       radial_edges[i] - sphere_radius))[0]

    if total_mass > Massa_U:
        excess_mass = total_mass - Massa_U
        # Loops radially from outside to inside, but only over the half
            of the segments
        for i in range(n_axial):
            for j in range(n_radial):
                if i < i_min -1 or i > i_max:
                    index = i * n_radial+ j
                    if segment_masses[index] <= excess_mass:
                        excess_mass -= segment_masses[index]
                        segment_masses[index] = 0.1
                    else:
                        segment_masses[index] -= excess_mass
                        excess_mass = 0

            if excess_mass <= 0:
                break
        slicedmass = []

    # Splits each mass into n_planes parts
    for mass in segment_masses:
        for _ in range(2*n_planes):
            slicedmass.append(mass / (2*n_planes))
    return slicedmass
```

A.2 Function to create the Serpent input files

```
def generate_geometry(fuel_segments):
    """Generates Serpent input file."""
    input_data = """
    set acelib "/data/leuven/376/vsc37601/xsdata/JEFF-3.3.0.xsdata"
/************************
* Material definitions *
```

```
**********************/
"""
    radial_edges = np.linspace(0, Cilinder_radius, n_radial + 1)
    axial_edges = np.linspace(0, Cil_height, n_axial + 1)
    planes = planar_slices(n_planes)
    cell_id = 1

    for i in range(n_axial):
        for j in range(n_radial):
            for k in range(2*n_planes):
                r_min, r_max = radial_edges[j], radial_edges[j + 1]
                z_min, z_max = axial_edges[i], axial_edges[i + 1]
                segment_height = z_max - z_min

                cell_volume = segment_volume(r_min, r_max,
                   segment_height) / n_planes
                uranium_mass = 2 * fuel_segments[i * n_radial *2*
                   n_planes + j *2* n_planes + k]
                uranium_volume = uranium_mass / density_U
                water_volume = max(cell_volume - uranium_volume, 0)
                water_mass = water_volume
                total_mass = uranium_mass + water_mass

                density_solution = total_mass / cell_volume if
                   cell_volume > 0 else 0
                mass_percentage_U235 = uranium_mass *
                   Uranium_enrichment / total_mass if total_mass > 0
                   else 0
                mass_percentage_U238 = uranium_mass * (1 -
                   Uranium_enrichment) / total_mass if total_mass > 0
                   else 0
                mass_percentage_H = (water_mass / total_mass) * 2/18 if
                    total_mass > 0 else 0
                mass_percentage_O = (water_mass / total_mass) * 16/18
                   if total_mass > 0 else 0

                input_data += f"""
mat solution{cell_id}    -{density_solution:.6f}
92235.02c    -{mass_percentage_U235:.6f}
92238.02c    -{mass_percentage_U238:.6f}
1001.02c     -{mass_percentage_H:.6f}
8016.02c     -{mass_percentage_O:.6f}
"""
                cell_id += 1

    input_data += """
/***********************
* Geometry definitions *
**********************/
```

```
"""

    cell_id = 0
    # Define the planes only once
    for k, (A, B, C, D) in enumerate(planes):
        input_data += f"surf p_plane{k + 1} plane {round(A, 4)} {round(
           B, 4)} {round(C, 4)} {round(D, 4)}\n"

    for i in range(n_axial):
        for j in range(n_radial):
            r_min, r_max = radial_edges[j], radial_edges[j + 1]
            z_min, z_max = axial_edges[i], axial_edges[i + 1]

            # Define the cylindrical shell for each radial and axial
               segment
            input_data += f"surf s{cell_id} cyl 0.0 0.0 {r_max} {z_min}
                {z_max}\n"

            solution_id = 1
            # Add cells using planes that are next to each other to
               define slices
            for k in range(n_planes):
                if k == 0: #1st plane
                    input_data += f"cell c{cell_id}_sega{k} 0 solution{
                       solution_id+cell_id} -s{cell_id} p_plane{k + 1}
                       p_plane{n_planes}\n"
                    solution_id += 1
                    input_data += f"cell c{cell_id}_segb{k} 0 solution{
                       solution_id+cell_id} -s{cell_id} p_plane{k+1} -
                       p_plane{k + 2}\n"
                    solution_id += 1
                    input_data += f"cell c{cell_id}_segc{k} 0 solution{
                       solution_id+cell_id} -s{cell_id} -p_plane{k+1}
                       p_plane{k + 2}\n"
                    solution_id += 1
                elif k == n_planes - 1: #last plane
                    input_data += f"cell c{cell_id}_segx{k} 0 solution{
                       solution_id+cell_id} -s{cell_id} -p_plane1 -
                       p_plane{k+1}\n"
                    solution_id += 1
                else: #middle planes
                    input_data += f"cell c{cell_id}_segd{k} 0 solution{
                       solution_id+cell_id} -s{cell_id} p_plane{k+1} -
                       p_plane{k + 2}\n"
                    solution_id += 1
                    input_data += f"cell c{cell_id}_sege{k} 0 solution{
                       solution_id+cell_id} -s{cell_id} -p_plane{k+1}
                       p_plane{k + 2}\n"
                    solution_id += 1
```

```
            cell_id += n_planes *2

    # Outer boundary surface
    input_data += f"surf s{cell_id} cyl 0.0 0.0 {Cilinder_radius} 0 {Cil_height}\n"
    input_data += f"cell c{cell_id} 0 outside s{cell_id}\n"

    input_data += """
/******************
* Run parameters *
******************/
set pop 5000 60 20
mesh 1  500  500
mesh 3  500  500
"""

    with open("/user/leuven/376/vsc37601/Serpent/Masterthesis/Inputfile_fiss_geometry", "w") as file:
        file.write(input_data)
```

A.3 Functions to run the input files and extract the value of keff

```
def run_serpent_script():
    """Runs the Serpent input file."""
    command = ["/user/leuven/376/vsc37601/2.2-main/src/sss2", "-omp", "max", "Inputfile_fiss_geometry"]
    subprocess.run(command, check=True)

def get_keff_value():
    """Extract the keff value from the Serpent output file."""
    with open("/user/leuven/376/vsc37601/Serpent/Masterthesis/Inputfile_fiss_geometry_res.m", "r") as file:
        content = file.read()
    matches = re.findall(r'IMP_KEFF\s*\(idx, \[1:\s*2\]\) = \[\s*([\d.E+-]+)\s*[\d.E+-]+\s*\];', content)
    return float(matches[-1]) if matches else None
```

A.4 Optimization algorithm function consisting of the Deap genetic algorithm and Bayesian optimization function

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
toolbox = base.Toolbox()
```

```
4
5  # Register the individual creation functions
6  toolbox.register("individual_homogeneous", tools.initIterate, creator.
       Individual, generate_homogeneous_fuel)
7  toolbox.register("individual_cilinder", tools.initIterate, creator.
       Individual, generate_cilinder_distribution)
8  toolbox.register("individual_cilinder_2", tools.initIterate, creator.
       Individual, generate_cilinder_distribution2)
9  toolbox.register("individual_random", tools.initIterate, creator.
       Individual, generate_random_distribution)
10
11 best_keff_tracker = {'keff': 0.0}
12
13 def normalize(individual):
14     """Normalizes the uranium distribution when needed."""
15     individual = [max(0, mass) for mass in individual]
16     total_mass = sum(individual)
17     return [x * (Massa_U / total_mass) for x in individual]
18
19 def evaluate(individual):
20     """Evaluates the keff of an individual using Serpent."""
21     # Save current individual's distribution to a Serpent input file
22     generate_geometry(individual)
23
24     # Run Serpent
25     run_serpent_script()
26
27     # Parse output to get keff
28     keff = get_keff_value()
29
30
31     # If best so far, copy output images immediately
32     if keff - best_keff_tracker['keff'] > 0.0009:
33         best_keff_tracker['keff'] = keff
34         with open("/user/leuven/376/vsc37601/Serpent/Masterthesis/
               Generations_population.txt", "a") as f:
35             f.write(f"new best keff = {keff}\n")
36         shutil.copy(
37             "/user/leuven/376/vsc37601/Serpent/Masterthesis/
                   Inputfile_fiss_geometry_mesh1.png",
38             "/user/leuven/376/vsc37601/Serpent/Masterthesis/
                   Sideview_mostcritical_config.png"
39         )
40         shutil.copy(
41             "/user/leuven/376/vsc37601/Serpent/Masterthesis/
                   Inputfile_fiss_geometry_mesh2.png",
42             "/user/leuven/376/vsc37601/Serpent/Masterthesis/
                   topview_mostcritical_config.png"
43         )
```

```
44      # Divide the array into n_axial segments and sum every group of
            values
45      segment_length = len(individual) // (n_axial)
46      reduced_individual = [sum(individual[i * segment_length:(i + 1) *
            segment_length]) for i in range(n_axial )]
47      evaluated_X.append(np.array(reduced_individual))  # Save the
            reduced array for surrogate
48      evaluated_y.append(keff)
49      return (keff,)
50
51  def deap_crossover(p1, p2):
52      """Performs crossover between two parents."""
53      child1, child2 = tools.cxBlend(p1, p2, alpha=crossover_blend)  #
            Blend crossover (alpha can be adjusted)
54      child1 = normalize(child1)  # Normalize the child
55      child2 = normalize(child2)  # Normalize the child
56      return child1, child2
57
58  def deap_mutate(individual):
59      """Performs mutation on an individual using a gaussian."""
60      mutated_individual = tools.mutGaussian(individual, mu=0, sigma=
            mutationeffect, indpb=0.2)[0]
61      mutated_individual = normalize(mutated_individual)  # Normalize the
             mutated individual
62      return mutated_individual,
63
64  # Register the genetic algorithm functions
65  toolbox.register("population", tools.initRepeat, list, toolbox.
        individual_homogeneous)
66  toolbox.register("mate", deap_crossover)
67  toolbox.register("mutate", deap_mutate)
68  toolbox.register("select", tools.selTournament, tournsize=3)
69  toolbox.register("evaluate", evaluate)
70
71  # BO configuration
72  evaluated_X = []      # List of previous individuals
73  evaluated_y = []      # Corresponding keff values
74
75  def genetic_algorithm_deap():
76      """Runs the DEAP genetic algorithm."""
77      # Clear the log file before the GA starts
78      with open("/user/leuven/376/vsc37601/Serpent/Masterthesis/
            Generations_population.txt", "w") as f:
79          f.write("Starting simulation......")  # This clears the file
80
81      # Create the initial population
82      population = [
83          toolbox.individual_homogeneous(),
84          toolbox.individual_cilinder(),
```

```
        toolbox.individual_cilinder_2()]

    population += [toolbox.individual_random() for _ in range(
        population_size-3)]

    # Normalize and evaluate
    population = [creator.Individual(normalize(ind)) for ind in
        population]
    fitnesses = list(map(toolbox.evaluate, population))
    for ind, fit in zip(population, fitnesses):
        ind.fitness.values = fit

    last_bo_generation = -1
    # Run the genetic algorithm
    for gen in range(generations):
        print(f"Generation {gen + 1}")
        current_best = tools.selBest(population, 1)[0]
        current_keff = current_best.fitness.values[0]

        # Log progress
        message = f"Generation {gen + 1}, best keff = {
            best_keff_tracker}, best keff in this generation = {
            current_keff}"
        print(message)
        with open("/user/leuven/376/vsc37601/Serpent/Masterthesis/
            Generations_population.txt", "a") as f:
            f.write(message + "\n")

        # Select top individuals and produce offspring
        top_half = tools.selBest(population, len(population) // 2)
        children = []
        while len(children) < population_size-1:
            p1, p2 = map(toolbox.clone, random.sample(top_half, 2))
            if random.random() < 0.8:
                toolbox.mate(p1, p2)
                del p1.fitness.values
                del p2.fitness.values
            if random.random() < mutation_rate:
                toolbox.mutate(p1)
                del p1.fitness.values
            if random.random() < mutation_rate:
                toolbox.mutate(p2)
                del p2.fitness.values
            p1[:] = normalize(p1)
            p2[:] = normalize(p2)
            children.extend([p1, p2])

        #make room for one bayesian prediction
        population[:] = children[:population_size-1]
```

```
129
130                 # Bayesian Optimization: Run if enough data is
                        collected and at regular intervals
131             if len(evaluated_X) > population_size-1 and gen !=
                    last_bo_generation:
132                 print(f"Running Bayesian Optimization at generation {gen +
                        1}...")
133                 time.sleep(1)
134                 last_bo_generation = gen
135                 # Convert evaluated data to numpy arrays
136                 X = np.array(evaluated_X)
137                 y = np.array(evaluated_y)
138
139                 # Check if X is non-empty and has the correct shape
140                 if X.size == 0 or X.shape[1] == 0:
141                     print("Error: Empty or invalid shape for X.")
142                     continue
143
144                 # Fit surrogate model
145                 kernel = Matern(nu=2.5)
146                 gp = GaussianProcessRegressor(kernel=kernel, normalize_y=
                        True)
147                 try:
148                     gp.fit(X, y)
149                 except Exception as e:
150                     print(f"Error fitting Gaussian Process: {e}")
151                     continue
152
153                 # Generate one BO individual
154                 best_acq = -np.inf
155                 candidate = None
156                 for _ in range(100):
157                     try:
158                         # Generate a random candidate scaled by the maximum
                                of each feature
159                         x_try = np.random.rand(X.shape[1]) * np.max(X, axis
                            =0)
160
161                         # Predict using the surrogate model
162                         mu, sigma = gp.predict(x_try.reshape(1, -1),
                            return_std=True)
163                         ucb = mu + 1.96 * sigma
164                         if ucb > best_acq:
165                             best_acq = ucb
166                             candidate = x_try
167                     except Exception as e:
168                         print(f"Error during candidate prediction: {e}")
169                         continue
170
```

```
            # Check if a valid candidate was found
            if candidate is None:
                print("No valid candidate found through Bayesian Optimization.")
                continue

            # Renormalize the candidate to match the format
            renormalized_candidate = []
            for mass in candidate:
                for _ in range(2 * n_planes * n_radial):
                    renormalized_candidate.append(mass / (2 * n_planes * n_radial))

            # Create the BO individual and normalize it
            bo_ind = creator.Individual(list(normalize(renormalized_candidate)))
            bo_ind.fitness.values = toolbox.evaluate(bo_ind)

            # Add BO individual to the population
            print("Adding BO individual to the population")
            bo_ind.is_bo = True

            children.append(bo_ind)

        population[:] = children[:population_size]

        # Log the addition of the BO individual
        with open("/user/leuven/376/vsc37601/Serpent/Masterthesis/Generations_population.txt", "a") as f:
            f.write(f"BO individual added at generation {gen + 1}, keff: {bo_ind.fitness.values[0]}\n")

        if len(population) != population_size:
            population += [toolbox.individual_random() for _ in range(population_size - len(population))]

        # Evaluate new individuals
        invalid_ind = [ind for ind in population if not ind.fitness.valid]
        fitnesses = map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

    # Final result
    best_individual = tools.selBest(population, 1)[0]
    return best_individual, best_individual.fitness.values[0]


if __name__ == "__main__":
```

```
    best_dist, best_keff = genetic_algorithm_deap()
    print(f"Best keff: {best_keff}")
```

A.5 Example of a Slurm jobscript

```
#!/bin/bash -l
#SBATCH --account=lp_h_vsc37601
#SBATCH --cluster=wice
#SBATCH --partition=batch_icelake_long
#SBATCH --time=120:00:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=72

source venv/bin/activate
export SERPENT_DATA="$VSC_DATA/xsdata/JEFF-3.3.0"
module load libgd/2.3.3-GCCcore-12.3.0

python Algoritme_thesis_Enes_Orhan.py > /dev/null 2>&1
```