Politecnico di Torino

Corso di Laurea Magistrale in
Ingegneria Energetica e Nucleare

Model development for the solution of eigenvalue problems in nuclear reactor physics

Supervisor:
Sandra Dulla

Author:
Alberto Calabria

Co-supervisor:
Nicolò Abrate

Torino, March 2022

Abstract

Monte Carlo method has a remarkable variety of applications, from Finance to Biology, from Engineering to Climate Change studies, etc. This widespread use is clearly due to its nature of statistical method which allows to simulate the genuine behaviour of complex systems starting from their fundamental phenomena provided that they are representable through a statistical model. Once defined a random variable of use for the quantity to be evaluated, a numerical code simulates the system behaviour in order to obtain, through a certain number of experiments, the estimation of the mean value of this random variable with its statistical uncertainty.

In the field of Nuclear Engineering this set of methods finds use, for instance, in the crucial evaluation of the criticality condition of a nuclear reactor. A lot of codes can run this task, such as MCNP (Monte Carlo N-Particle Transport Code) or Serpent whose programming languages are respectively Fortran and C.

The purpose of this work is the evaluation of the criticality condition of a reactor with simple geometrical configuration (mono-dimensional slab with an heterogeneous medium and a two-groups-discretized energy spectrum) by investigating both a well-known parameter, such as the κ_0 factor, and a less common one, like the γ_0 factor, through the implementation of codes written in Python, which is one of the most popular programming languages. This work will hopefully grant a Monte Carlo benchmark for future different approaches to the solution of criticality problems, with the possibility of customizing the number of energy groups used to describe the neutron interaction and changing the features of the system under investigation, e.g., the number of different layers composing the geometry.

This thesis presents good results in terms of quality of the calculated factors, which show different behaviours inside the reactor. In order to compute κ_0 and γ_0 properly, a judicious choice of the input parameters is necessary.

Contents

Introduction

Around the middle of the twentieth century, nuclear physics led an astonishing breakthrough in many technological fields, such as military, transports, energetic, medical and aerospace. Behind this incredible advance of the human possibilities, there were the efforts of some of the most brilliant minds of all time, like John Von Neumann, Enrico Fermi, Stanislaw Ulam, Edward Teller, Nicholas Metropolis and many others. The birth of Monte Carlo simulation can be traced back to World War II: the Manhattan project desperately needed methods to study nuclear materials ahead of the atomic bomb assembling. This coincided with another initiative: building the first electronic computer. The first one, ENIAC, was built in 1946 at University of Pennsylvania [1]. Scientists understood that a statistical approach for solving neutron diffusion problems would be the right way and so, Von Neumann prepared an outline of that: he knew that performing exhausting statistical sampling with the consequent large number of calculations was the main problem and that the new computer technology could be the right propellent to solve it. Since Ulam's uncle used to borrow money from his family for gambling at the famous Monte Carlo casino in the Principality of Monaco, this method took this name with which it went down in history[1]. It was immediately realized that the Monte Carlo method was more flexible for simulating complex problems as compared to differential equations and the amount of computation, as previously said, was huge, but this obstacle was quickly overcome, with the development, thanks to Metropolis and Von Neumann's wife (Klari), of a new control system for ENIAC and the creation of FERMIAC (from an idea of the pioneer of the studies in neutron moderation, Fermi). These scientists succeeded in getting the ability to solve several neutron transport problems [1].

As history teaches, Nuclear Science gave birth to Monte Carlo as statistical method used for dealing with complex systems. Hence, with the incumbency of a “nuclear renaissance” for the sake of the global ecosystems, the mastery and the improvement of the available analytical instruments are more needed than ever. This thesis means to act in this direction. Its main contents are illustrated in this brief introduction.

The first chapter will provide a presentation of the Transport Theory, a crucial milestone for every analytical work about Nuclear Physics, with its general model and its geometrical configuration, its boundary and initial conditions, and finally its various simplifications.

The second chapter will develop the mathematical basis of the criticality problem on which this work is focused: the eigenvalue problem and the definitions of the eigenvalues will be the key issues of this part.

The third chapter will begin with the comparison of the two different approaches for the previously mentioned problem: the deterministic method ($\mathrm{P_N}$ approximation method) and the stochastic one (Monte Carlo method). The latter is the core of the entire thesis. Thus, it is fundamental to give essential bases of probability and statistics to understand how this method works. After this general part, the chapter will deal with the theoretical aspects of the physical phenomena that occur in a reactor through a 'Monte Carlo method' standpoint.

The fourth chapter will show every detail of the codes which has been developed to accomplish the eigenvalues' computations, with its programming language, its general structure, its critical issues and the explanation of each code that has been written for that purpose.

The final chapter shall illustrate the results and the criteria to be followed to achieve them. Testing the codes through comparisons with reliable results from other codes will establish the quality of the present thesis.

The chapters are followed by the conclusions that this work has led to, and finally by three appendices with the codes created, sorted according to the 'degree' of sharing: the first appendix collects all the common scripts, shared by both the eigenvalue algorithms, while the last two appendices respectively present scripts which exclusively belong to the single algorithm, κ_0's and then γ_0's.

Chapter 1

The neutron transport

In order to introduce this thesis, it is crucial to begin with the equation that governs neutron behavior and that ”describes, through a statistical approach, how thermodynamic systems that are not in equilibrium work” [2]. Formulated in 1872 by Ludwig Boltzmann as a kernel of kinetic theory of gas, it expresses in an elegant form every phenomenon involving the physical transport of a quantity or of particles [2]. It is non-linear and presents both integral and differential nature.

Neutron transport requires two main assumptions to be added to the original Boltzmann equation: the former states that ”neutrons are point particles and their trajectories are straight lines between two successive collisions with the medium nuclei”, while the latter says ”that ‘neutron-neutron’ interactions are neglected” [2].

The section 1.1.1 of this first chapter shows the derivation of the general form of the Boltzmann equation for neutron transport, with a three-dimensional spatial configuration, continuous energy spectrum and time dependency. The section 1.1.2 gives the definitions of boundary and initial conditions, necessary for every differential equation solution. The last section 1.1.3 provides the illustration of the major simplifications this thesis needs, i.e. plane geometry model and isotropic scattering.

1.1 Boltzmann's Transport Equation

1.1.1 The general physical model

The core of the neutron transport is the balance in time of the expected number of neutrons in a volume element, whose coordinates are not 'classical', they are not only spatial and temporal. They form the so-called *phase space* [2]. Thus, firstly it is important to define which are the dimensions of this phase space:

- The spatial coordinates defined by a triplet of values (x, y and z coordinates summarized with $\vec{r}$) expressed in [cm].
- The directional coordinates of the flight of the neutrons defined by two angles (the *azimuthal angle* φ with a domain that varies between 0 and 2π, and the *co-latitude angle* ϑ between -π and π, both summarized with $\hat{\Omega}$, a unit vector) with steradian as unit of measure [sr].
- The neutron kinetic energy coordinate E, expressed in [MeV].

Thus, there are six values that define the phase space of interest, but that may be summarized in the equation, for the sake of simplicity, with three magnitudes: $\vec{r}$, $\hat{\Omega}$ and E (together with the obvious time dependence t expressed in [s]). In section 1.1.3 a more detailed characterization, particularly of the angular coordinates, will be given. A graphical representation of the phase space is depicted in Fig.:1.1.: the position is measured from the Cartesian coordinate system (x, y, z), while the direction is the vector that coincides with unitary radius of the sphere representing all the possible solid angles. Its coordinate system consists in the following triad: $\hat{e_x}$, $\hat{e_y}$, $\hat{e_z}$. Although the direction is a unit vector, in this figure its length is multiplied times the speed, given by $\sqrt{\frac{2E}{m}}$, being m the neutron's mass.

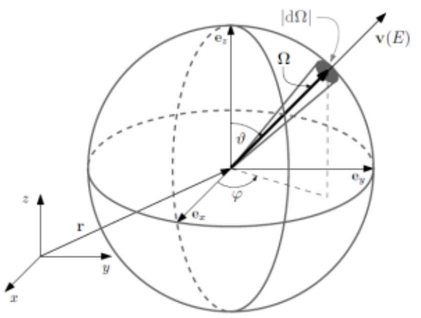


Figure 1.1: Neutron phase space [2]

Together, these coordinates form the variables of a function, the *neutron density* $n(\vec{r},\hat{\Omega},\mathrm{E},\mathrm{t})$ that represents the probable number of neutrons at point $\vec{r}$ per unit volume, per unit energy, per unit solid angle, at time t. It is not a density on time, it is not a rate. Its unit of measure is $[\frac{1}{cm^3 sr MeV}]$. Multiplying the neutron density times the speed of neutrons v (a function of the energy and the mass neutron), the definition of the *neutron angular flux* $\phi(\vec{r},\hat{\Omega},\mathrm{E},\mathrm{t})$ is obtained:

$$\phi(\vec{r},\hat{\Omega},E,t) = n(\vec{r},\hat{\Omega},E,t)v \tag{1.1}$$

whose unit of measure is $[\frac{1}{cm^2 sr MeV s}]$. The term 'angular' emphasizes the dependence on the flight direction of neutron motion. This flux is the total distance covered by neutrons per unit volume, per unit energy, per unit solid angle and per unit time. So it is a measure of 'neutron traffic' rather than a classical flux, i.e. a quantity flowing across a surface. In order to have this kind of information, the elementary current must be defined, by multiplying the flux times the direction unit vector $\hat{\Omega}$:

$$\vec{j}(\vec{r},\hat{\Omega},E,t) = \phi(\vec{r},\hat{\Omega},E,t)\hat{\Omega} \tag{1.2}$$

this elementary current is the number of neutrons per unit area (perpendicular to neutrons' direction), per unit energy, per unit solid angle, and per unit time with the same unit of measure as the flux. Integrating over all directions both the members in Eq.:1.2, the total flux $\Phi(\vec{r},\mathrm{E},\mathrm{t})$ and the net particle current $\vec{J}(\vec{r},\mathrm{E},\mathrm{t})$ are respectively obtained:

$$\Phi(\vec{r},E,t) = \oint \phi(\vec{r},\hat{\Omega},E,t)\,d\Omega \tag{1.3}$$

and

$$\vec{J}(\vec{r}, E, t) = \oint \phi(\vec{r}, \hat{\Omega}, E, t)\hat{\Omega}\, d\Omega \tag{1.4}$$

While defining the density, it has been said that is a ‘probable’ number of neutrons, because this population may fluctuate, since the general balance in the phase space volume element is affected not only by sources of any kind and by the free motion of neutrons, but also by their possible interactions with medium nuclei. These probabilities of collision events are quantified by the cross sections, defined as the probability per unit path traveled by a neutron to experience a specific interaction with the nuclei of matter. Their unit of measure refers to a reciprocal of a length, so it is [1/cm].

The interactions of interest for this dissertation are:

- The total cross section Σ_t, which quantifies the probability per unit path of any type of interaction.
- The absorption cross section Σ_a, characteristic of the absorption events.
- The scattering cross section Σ_s, for collision events (both elastic and inelastic collisions).
- The fission cross section Σ_f, obviously for fission reactions.

Cross sections depend on position (the type of nuclei may change in space) and neutrons’ energy. They also change in time, but according to a completely different timescale with respect to neutron mean life, so the time dependence may be neglected [2].
It is now possible to write down the balance cited above. The variation in time of the number of neutrons in the phase space element depends on:

- Leakage due to streaming
- Leakage due to collisions (absorption events and scattering-out)
- Gain from scattering-in
- Gain from sources

The variation in time of the total number of neutrons in the phase space element has the following form:

$$[n(\vec{r},\hat{\Omega},E,t+dt)-n(\vec{r},\hat{\Omega},E,t)]d\vec{r}dtdEd\Omega \tag{1.5}$$

In order to have the flux in the equation, a modification of Eq.:1.5 is needed:

$$(1/v)[\phi(\vec{r},\hat{\Omega},E,t+dt)-\phi(\vec{r},\hat{\Omega},E,t)]d\vec{r}dtdEd\Omega \tag{1.6}$$

<u>Leakage for streaming</u>

The first term of the list above consists in the neutrons getting into and going out through the boundary surfaces of the domain. It is possible to sum up every surface contribute obtaining a net particle current:

$$\sum_i[\phi(\vec{r},\hat{\Omega},E,t)\hat{\Omega}]\xi_i dA_i dtdEd\Omega \tag{1.7}$$

being ξ_i the vector normal to the single surface. Thanks to the Gauss theorem, the gradient of the current related to a volume might replace the flux through a surface, so:

$$[\nabla\cdot\hat{\Omega}\phi(\vec{r},\hat{\Omega},E,t)]d\vec{r}dtdEd\Omega \tag{1.8}$$

Because of the convention of the sign for in-going and out-going fluxes (positive the latter, negative the former) if the 'in - out term' is wanted, the opposite of 1.8 must be taken. Knowing the chain rule for derivation, 1.8 may also be written in another way:

$$-[\nabla\cdot\hat{\Omega}\phi(\vec{r},\hat{\Omega},E,t)] = -\phi(\vec{r},\hat{\Omega},E,t)\nabla\cdot\hat{\Omega}-\hat{\Omega}\cdot\nabla\phi(\vec{r},\hat{\Omega},E,t) \tag{1.9}$$

Since the unit vector $\hat{\Omega}$ is independent of the coordinates of the system, $\nabla\cdot\hat{\Omega}$ is equal to zero. So the streaming term can be also written like this:

$$-[\nabla\cdot\hat{\Omega}\phi(\vec{r},\hat{\Omega},E,t)]d\vec{r}dtdEd\Omega = -\hat{\Omega}\cdot\nabla\phi(\vec{r},\hat{\Omega},E,t)d\vec{r}dtdEd\Omega \tag{1.10}$$

Leakage for collisions

The second term on the right-hand side of the balance takes into account all the contributes to the removal of neutrons; indeed, the cross section Σ_t includes absorption, scattering, fission, etc. Σ_t is the total sum all the cross sections considered:

$$\Sigma_t(\vec{r}, E)\phi(\vec{r}, \hat{\Omega}, E, t)d\vec{r}dtdEd\Omega \tag{1.11}$$

Gain from scattering-in

This term consists in the integral sum of neutrons, having different energies from the interval (E, E+dE) and different directions from the elemental solid angle dΩ, that may scatter in the 'correct' observation ranges of energy and direction:

$$\oint d\Omega' \int dE'\Sigma_s(\vec{r}, E')\phi(\vec{r}, \hat{\Omega}', E', t)f_s(\vec{r}, E' \Rightarrow E, \hat{\Omega}' \Rightarrow \hat{\Omega})d\vec{r}dtdEd\Omega \tag{1.12}$$

Thus, this term gives to the equation "its integral nature" [2]. The probability per unit path for a neutron of energy E' in the volume d$\vec{r}$ about $\vec{r}$ to undergo a scattering interaction with a nucleus of the background medium is expressed by the scattering cross-section Σ_s. Instead, the probability that a neutron with the initial energy E' and direction $\hat{\Omega}'$ will be scattered in the volume d$\vec{r}$ about $\vec{r}$, with the energy dE about E and direction $\hat{\Omega}$ within the solid angle dΩ is given by the scattering probability function $f_s(\vec{r}, E' \Rightarrow E, \hat{\Omega}' \Rightarrow \hat{\Omega})$. Thus, the scattering kernel f_s (transfer function of scattering) is defined as a probability density function (a concept that will be more deeply investigated in Chapter 3) which, through the integration over the energy range and over all the directions, proves to be normalized [2]:

$$\oint d\Omega' \int_0^\infty dE' f_s(\vec{r}, E' \Rightarrow E, \hat{\Omega}' \Rightarrow \hat{\Omega})d\vec{r}dtdEd\Omega = 1 \tag{1.13}$$

If materials can be supposed isotropic, then the function depends neither on $\hat{\Omega}'$ nor $\hat{\Omega}$, but on the angle between them, that is expressed by their inner product $\hat{\Omega}' \cdot \hat{\Omega}$. Consequently, the transfer scattering function must change:

$$f_s(\vec{r}, E' \Rightarrow E, \hat{\Omega}' \cdot \hat{\Omega}) \tag{1.14}$$

Gain from sources

This term is composed by two contributes: the former is the generation from a external source, not due to fissions. The number of 'new' neutrons generated by a source in the volume d$\vec{r}$, having energy in range (E, E+dE), and directed into dΩ in the time interval dt is given by the following expression:

$$S(\vec{r}, \hat{\Omega}, E, t) d\vec{r} dt dE d\Omega \tag{1.15}$$

where $S(\vec{r}, \hat{\Omega}, E, t)$ is the number of neutrons generated by the source per unit volume, per unit energy, per unit solid angle and per unit time. The second addition is the fission contribution: it is an isotropic interaction, so the number of "new" neutrons within dΩ, with energy range (E, E+dE) has the following expression:

$$\frac{d\Omega}{4\pi} \nu(\vec{r}, E') \chi(\vec{r}, E) dE \tag{1.16}$$

where:

- $\frac{d\Omega}{4\pi}$ is the probability that a neutron is emitted within dΩ (a neutron has the same probability of being emitted in every direction).
- $\nu(\vec{r}, E')$ is the number of neutrons generated per fission that depends on $\vec{r}$ and on the energy of the neutrons that has caused fission.
- $\chi(\vec{r}, E)dE$, i.e. the *fission spectrum* is the probability that a neutron is emitted with energy in the range between E and E+dE. It is called fission spectrum and gives the energy distribution of neutrons generated by fission.

So the fission transfer function is:

$$\frac{1}{4\pi} \nu(\vec{r}, E') \chi(\vec{r}, E) \tag{1.17}$$

And the fission contribution term in the balance is:

$$\frac{\chi(\vec{r}, E)}{4\pi} \oint d\Omega' \int dE' \Sigma_f(\vec{r}, E') \phi(\vec{r}, \hat{\Omega}', E', t) \nu(\vec{r}, E') d\vec{r} dt dE d\Omega \tag{1.18}$$

Finally, it is possible to write down the general, integro-differential form of the Boltzmann neutron transport equation by unifying and re-arranging all the terms composing the balance, by dividing both right and left-hand sides by $d\vec{r}dtdEd\Omega$ and by taking the limit of dt approaching to zero:

$$\begin{aligned}\frac{1}{v}\frac{\partial\phi(\vec{r},\hat{\Omega},E,t)}{\partial t}+\Sigma(\vec{r},E)\phi(\vec{r},\hat{\Omega},E,t)+\hat{\Omega}\cdot\nabla\phi(\vec{r},\hat{\Omega},E,t)\\ =S(\vec{r},\hat{\Omega},E,t)+\oint d\Omega'\int dE'\Sigma_s(\vec{r},E')\phi(\vec{r},\hat{\Omega}',E',t)f_s(\vec{r},E'\Rightarrow E,\hat{\Omega}'\cdot\hat{\Omega})\\ +\frac{\chi(\vec{r},E)}{4\pi}\oint d\Omega'\int dE'\Sigma_f(\vec{r},E')\phi(\vec{r},\hat{\Omega}',E',t)\nu(\vec{r},E')\end{aligned} \tag{1.19}$$

Eq.:1.19 is the time-dependent neutron transport equation in presence of fission reactions and external source. Nevertheless, the immediate emissions from fission events concern only a fraction of the entire 'new-born' population. Some neutrons are called *delayed* [2]. They are indeed emitted in the decay process of some fission products called the delayed neutron precursors (sorted in N families since they are hundreds). If one take into account them, Eq.:1.19 has to be modified twice: the former by multiplying the fission source term times (1-β), i.e. the fraction of prompt neutrons, the latter by adding to the right-hand side of Eq.:1.19 the term with the decay of precursors ($\sum_{i=1}^{N}\lambda_i C_i(\vec{r},t)$, with C_i being the concentration of the i-th precursor family). This renewed equation will form a system of N+1 equations (with the proper set of initial and boundary conditions). Each one of the other N equations will represent the time balance of the corresponding precursor family, with the following form [2]:

$$\frac{\partial C_i(\vec{r},t)}{\partial t}=\beta_i\nu\Sigma_f\phi(\vec{r},t)-\lambda_i C_i(\vec{r},t) \tag{1.20}$$

Between prompt neutrons, i.e those particle which are emitted within fractions of the microsecond by fissile nuclei, have a time advance with respect to the delayed ones that ranges from values of the order of milliseconds up to seconds, nearly: it is function of the half-live values of neutron precursors [2]. For the steady-state problems and criticality evaluations, that are the focus of this work, the neglect of delayed neutrons might be assumed, but they still play a pivotal role in nuclear reactor kinetics and reactor control. [2].

1.1.2 Initial and boundary conditions

Eq.:1.19's left-hand side has a first-order derivative in time and a divergence (a first-order differential operator). The former generates only one initial condition, the latter only one boundary condition. The initial condition usually has the following form:

$$\phi(\vec{r}, E, \hat{\Omega}, t = 0) = \phi_0(\vec{r}, E, \hat{\Omega}) \tag{1.21}$$

Conversely, the boundary conditions are characterized by a certain level of variety. The figure below (Fig.:1.2) depicts an interface boundary surface (Γ) between regions I and II, and unit vectors $\hat{\Omega}$ and $\hat{\Omega}'$ referring to neutrons' directions entering regions I and II, respectively, and $\hat{n}$ is the unit vector normal to surface Γ. Five types of boundary conditions will be discussed: vacuum, specular reflective, albedo, white and periodic [1].

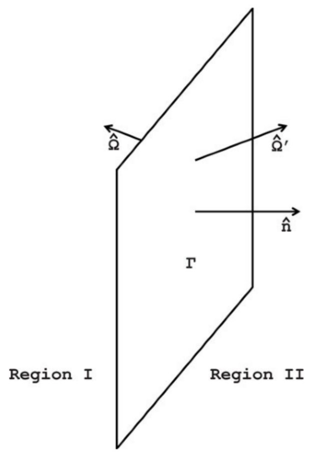


Figure 1.2: Schematic of an interfacial boundary between two regions [1]

In this thesis only the first two types of boundary conditions will taken into account, the former as a fundamental condition, the latter as an attempt of improvement of the computational performance (Chapter 5).

1. *Vacuum boundary condition*: if region II in Fig.1.2 is considered as void, then no particle will be reflected back to region I. Hence, the incoming angular flux to region I is expressed by
$$\phi(\vec{r_b}, E, \hat{\Omega}) = 0 \tag{1.22}$$
for $\hat{n} \cdot \hat{\Omega} < 0$ and $\vec{r_b} \in \Gamma$.

2. *Specular reflective boundary condition*: it is a symmetric condition in which the incoming and the outgoing angular fluxes at the interface are equal, i.e.:
$$\phi(\vec{r_b}, E, \hat{\Omega}) = \phi(\vec{r_b}, E, \hat{\Omega}') \tag{1.23}$$
for $\hat{n} \cdot \hat{\Omega} = -\hat{n} \cdot \hat{\Omega}'$ and $\vec{r_b} \in \Gamma$. In order to achieve such a condition, regions I and II must be identical. If the geometry of the problem allows its use, it results in significant reduction in model size, and therefore in computation time [1].

3. *Albedo boundary condition*: in this case, the incoming and the outgoing angular fluxes are related as follows:
$$\phi(\vec{r_b}, E, \hat{\Omega}) = \alpha(E)\phi(\vec{r_b}, E, \hat{\Omega}') \tag{1.24}$$
for $\hat{n} \cdot \hat{\Omega} = -\hat{n} \cdot \hat{\Omega}'$ and $\vec{r_b} \in \Gamma$. $\alpha(E)$ is the albedo coefficient for the particle with energy E. With this condition, only a fraction $\alpha(E)$ of particles leaving the surface, e.g., along $\hat{\Omega}'$ into region II, will be reflected back to region I. This allows to avoid modelling a region, while still keeping its impact, i.e., the reflection of some particles [1].

4. *White boundary condition*: with this condition, particles leaving region I are reflected back with a certain cosine distribution $p(\mu) = \mu$ (a concept that will be studied in Chapter 3).

5. *Periodic boundary condition*: in problems with physical periodicity, such as fuel assemblies or fuel cells in a reactor, in special cases of an infinite system, one may be able to establish that angular flux distribution on one boundary $\vec{r}$ is equal to the angular distribution on another boundary $\vec{r} + \vec{r_d}$ in a periodic manner [1]:
$$\phi(\vec{r} + \vec{r_d}, E, \hat{\Omega}) = \phi(\vec{r}, E, \hat{\Omega}') \tag{1.25}$$

1.1.3 Plane geometry model

Naturally, going from a three-dimensional case to a one-dimensional situation involves simplifications, but the new model may be useful as well: if the domain, e.g. a reactor environment, has two dimensions that are by far greater than the third one, the analysis may be led by focusing only on the latter: this is a 'slab configuration', i.e. the cases examined in this thesis.

If the neutron angular flux depends on a unique spatial coordinate, for instance the z-coordinate, then it depends only on the angle formed by its direction with the z-axis or, alternatively, on the cosine of this angle, generally denoted as μ [2].

In cartesian coordinates, the expression of the unit direction vector $\hat{\Omega}$, as function of the co-latitude angle ϑ and the azimuthal angle φ, is:

$$\hat{\Omega} = \sqrt{1-\mu^2}\cos(\phi)\hat{e_x} + \sqrt{1-\mu^2}\sin(\phi)\hat{e_y} + \mu\hat{e_z} \tag{1.26}$$

with $\cos(\vartheta) = \mu$ and, obviously, $\sin(\vartheta) = \sqrt{1-\mu^2}$. The formulation of the inner product between $\hat{\Omega}$ and $\hat{\Omega}'$ becomes quite complex with this notation:

$$\hat{\Omega}' \cdot \hat{\Omega} = \sqrt{1-\mu'^2}\sqrt{1-\mu^2}\cos(\varphi' - \varphi) + \mu\mu' \tag{1.27}$$

Most of the last two equations must be eliminated thanks to the plane geometry assumption. Firstly, it turns all the position coordinates into z, then the divergence appearing in Eq.:1.19 becomes a simple spatial derivative with respect to z. The simplifications go on with $\hat{\Omega}$ that reduces to μ, and with the inner product of Eq.:1.27, which becomes $\mu\mu'$:

$$\begin{aligned} \frac{1}{v}\frac{\partial\phi(z,\mu,E,t)}{\partial t} + \mu\frac{\partial\phi(z,\mu,E,t)}{\partial z} + \Sigma_t(z,E)\phi(z,\mu,E,t) \\ = S(z,\mu,E,t) + \int_{-1}^{1} d\mu' \int dE'\Sigma_s(z,E')\phi(z,\mu',E',t)f_s(z,E' \Rightarrow E,\mu'\mu) \\ + \frac{\chi(z,E)}{2}\int_{-1}^{1} d\mu' \int dE'\Sigma_f(z,E')\phi(z,\mu',E',t)\nu(z,E') \end{aligned} \tag{1.28}$$

1.1.4 Isotropic scattering model

The last simplification for the model of interest is about the scattering transfer function's angular dependence. Chapter 3 will discuss its energy dependence to find a way to simplify it. It has been previously said that scattering interaction is not isotropic, but this supposition is legitimate. If it is given, then the probability is constant for all possible value of the polar angle cosine μ. Moreover, the probability for a neutron of being emitted with all potential direction and energies is equal to 1, thus:

$$\int_{-1}^{1} d\mu \int dE f_s(z, E' \Rightarrow E, \mu'\mu) = 1 \tag{1.29}$$

Since the isotropy assumption has been made, the probability density is a constant with respect to μ:

$$f_s(z, E' \Rightarrow E, \mu'\mu) = C f_s(z, E' \Rightarrow E) \tag{1.30}$$

where C is a constant value and, after integrating both sides of this equality over all directions and energy spectrum:

$$\int_{-1}^{1} d\mu \int dE f_s(z, E' \Rightarrow E, \mu'\mu) = C \int_{-1}^{1} d\mu \int dE f_s(z, E' \Rightarrow E) \tag{1.31}$$

The left-hand side is of course equal to 1 (from Eq.:1.29), as well as the integral over the energy spectrum of the right-hand side, which also has an integral over all the directions that is equal to 2. Hence, the constant C is equal to 1/2 and, consequently, the scattering transfer function is:

$$f_s(z, E' \Rightarrow E, \mu'\mu) = \frac{1}{2} f_s(z, E' \Rightarrow E) \tag{1.32}$$

Eq.:1.28, thanks to Eq.:1.32, may be re-written in its ultimate form:

$$\begin{aligned} \frac{1}{v}\frac{\partial \phi(z,\mu,E,t)}{\partial t} + \mu\frac{\partial \phi(z,\mu,E,t)}{\partial z} + \Sigma_t(z,E)\phi(z,\mu,E,t) \\ = S(z,\mu,E,t) + \frac{1}{2}\int_{-1}^{1} d\mu' \int dE' \Sigma_s(z,E')\phi(z,\mu',E',t) f_s(z,E' \Rightarrow E) \\ + \frac{\chi(z,E)}{2}\int_{-1}^{1} d\mu' \int dE' \Sigma_f(z,E')\phi(z,\mu',E',t)\nu(z,E') \end{aligned} \tag{1.33}$$

Fig.:1.3 depicts an example of the heterogeneous environment on which this thesis is focused: a layer of 'fuel' material amid two layers of 'moderator' material.

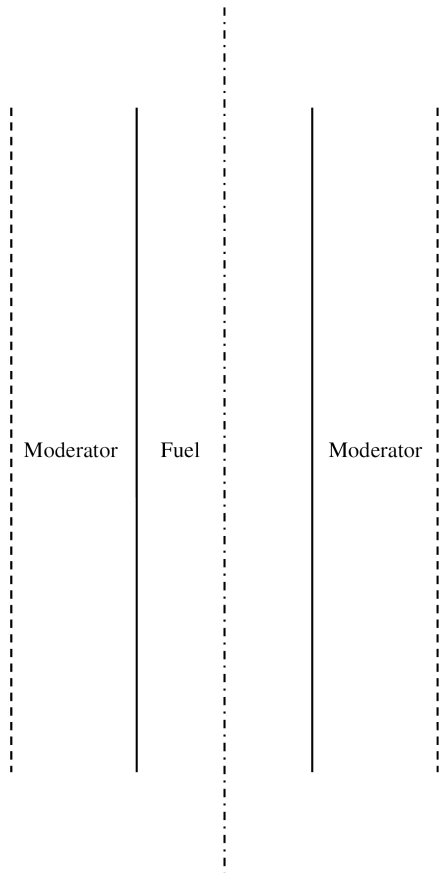


Figure 1.3: The simplest scheme for an heterogeneous slab.

Chapter 2

The criticality problem

A nodal issue in Nuclear Reactor Engineering is understanding whether the neutron population throughout the core is stable in time, decreases or increases with it, during the operations of the reactor life. By definition, a nuclear reactor is able to "self-sustain a controlled fission chain reaction, i.e. the phenomenon in which neutrons generated by fission reactions of elements like ^{235}U or ^{238}U, by interacting with others of these nuclides are able to induce further fission events, and so on" [2].

If a stationary energy production from the fission reactions is the reactor objective, neutrons produced by fission events and those lost due to absorption or leakages out of the boundaries of the system must be in equilibrium [2]. Naturally, human action can safely adjust this balance. Section 2.1 will explain how it is done and section 2.2 will provide the mathematical grounds on which is based.

2.1 Nuclear reactor control

If the aforementioned equilibrium is verified, the neutron population in the reactor is independent of time, and the unique neutron source term is represented by fissions [2]. A reactor in such a condition is referred to as *critical*, with a stationary fission chain reaction. Whether instead the number of neutrons 'born' after fission events is not able to counter-balance the number of neutrons lost due to absorption or leakage out of outer surface, in absence of an external neutron source (different from fission) the chain reaction will die out over time and the reactor is said *subcritical.* In the opposite situation, the system is destined to increase its neutron population "from generation to generation": this reactor is defined as *supercritical* [2]. Generally, nuclear power reactors are designed in such a way that these three conditions can be modified as needed through the *reactor control* procedures. The criticality problem is the research of "the right combination of two aspects of design that allows the reactor to achieve a critical state: the material composition and the geometrical configuration" [2].

Some physical phenomena, as the fuel depletion or the accumulation of fission products in the core as the chain reaction develops, affect the reactor criticality making the device subcritical. The increment of the amount of plutonium, instead, leads the system to supercriticality [2]. To respond to these events, engineers have developed some strategies, such as the insertion of control rods or the use of soluble 'neutron poisons' just to mention a few[2]. Hence, it is important to give a theoretical, basic understanding for that issue and then to deal with its main, specific markers.

2.2 The eigenvalue problem

Mathematically, the criticality problem is approached as an eigenvalue problem, an expression which reads:

$$\hat{A}\varphi = \lambda\varphi \tag{2.1}$$

where $\hat{A}$ is a matrix $\in C^{n\times n}$, λ is a scalar, real or complex, and φ is a non-null ("non-trivial solution") vector such that the expression in Eq.:2.1 is verified. Thus, any λ satisfying Eq.:2.1 is called an *eigenvalue* of the matrix $\hat{A}$ and the corresponding solution vector φ is the associated *eigenvector.* The eigenvectors having in common the same eigenvalue form an *eigenspace.* In summary, $\hat{A}$ is an operator which, when applied to λ, behaves like the operator "multiplication times a constant", where the constant is the eigenvalue of the problem [2]. If the eigenvectors belongs to a function space instead of a vector space, they are called *eigenfunctions* [3].
Chapter 3 will implement one of the many methods to solve Eq.:2.1. From the point of view of the neutron transport equation, eigenvalue equations may be formulated in many ways, but the main ones are four, giving rise to four different eigenvalues:

1. The effective multiplication factor κ.
2. The multiplication factor per collision γ.
3. The fundamental multiplication rate α
4. The effective density factor δ.

In order to have a smarter form of Eq.:1.33, some modifications must be done. Thus, referring to the steady-state (null time derivative), homogeneous version of the integro-differential form of the neutron transport equation 1.33, there would be [2]:

- *Leakage* operator:
$$\hat{L} = \mu \frac{\partial}{\partial z} \tag{2.2}$$
- *Removal* operator:
$$\hat{R} = \Sigma_t(z, E) \tag{2.3}$$
- *Scattering* operator:
$$\hat{S} = \frac{1}{2} \int_{-1}^{1} d\mu' \int dE' \Sigma_s(z, E') f_s(z, E' \Rightarrow E) \tag{2.4}$$
- *Fission* operator:
$$\hat{F} = \frac{\chi(z, E)}{2} \int_{-1}^{1} d\mu' \int dE' \Sigma_f(z, E') \nu(z, E') \tag{2.5}$$
- *Transport* operator:
$$\hat{T} = \hat{L} + \hat{R} \tag{2.6}$$

The transport equation, without external sources, in operator form then yields:
$$\hat{L}\phi(z, \mu, E) + \hat{R}\phi(z, \mu, E) = \hat{S}\phi(z, \mu, E) + \hat{F}\phi(z, \mu, E) \tag{2.7}$$

The four different eigenvalue problems differ on the combination of these specific operators, but the fundamental equation is still Eq.:2.7. Each eigenvalue constitutes an "identifier of the system deviation from criticality, whose amount depends on the specific eigenvalue" [2]. Once chosen one of these

four markers, the set of eigenvalues solution of the equation is denoted as the eigenvalue spectrum, or *eigenspectrum.* The eigenfunctions associated to the elements of this set are then called *modes*, or *harmonics* [2].The eigenvalues may be real or complex numbers, with different multiplicity [3]: the one with the largest modulus is generally referred to as the *spectral radius*, or the *fundamental eigenvalue.* The associated eigenfunction, or the *fundamental mode*, in the case of the neutron transport equation, is the only one that is physically meaningful, having constant sign over all the domain while the other eigenfunctions are called *higher-modes* [2].
After this introduction of the general criticality problem, the next subsections will be dedicated to each eigenvalue previously mentioned. It will not be an in-depth dissertation, knowing that the purpose of this thesis is to focus only on κ and γ.

2.2.1 The κ-eigenvalue

It is surely the most commonly investigated eigenvalue, also known as the *effective multiplication factor* κ: it modifies the fission source term of Eq.:2.7:

$$\hat{L}\phi + \hat{R}\phi = \hat{S}\phi + \frac{1}{\kappa}\hat{F}\phi \tag{2.8}$$

A first definition denotes κ as the "ratio between the amounts of neutrons present in the core in two successive fission generations" [2].The eigenvalue with the largest real part is denoted as the fundamental κ-eigenvalue, generally indicated as κ_0 or κ_{eff}. The solution associated to this eigenvalue is the κ-mode. If $\kappa_0 = 1$, the number of neutrons is stable generation by generation [2]. Hence, in this case the chain reaction is independent on time and the reactor is referred to as *critical.* If $\kappa_0 < 1$, the reactor constitutes instead a *subcritical* system, and if $\kappa_0 > 1$ it is *supercritical* [2].
In order to see Eq.:2.8 in a typical eigenvalue problem form, some modifications must be carried out [2]:

$$(\hat{L} + \hat{R} - \hat{S})\phi = \frac{1}{\kappa}\hat{F}\phi \tag{2.9}$$

Since the $\hat{F}$ operator could be singular (it would not admit the inverse), the inversion is applied to the combination of the other operators which is surely not singular. Finally, the multiplication factor is taken on the other side of the equation leading to the authentic eigenvalue problem form of the equation [2]:

$$\kappa\phi = (\hat{L} + \hat{R} - \hat{S})^{-1}\hat{F}\phi \tag{2.10}$$

2.2.2 The γ-eigenvalue

The γ-eigenvalue, the 'effective multiplication factor per collision', may be seen as a "direct eigenvalue of the integro-differential neutron transport equation" [2]. As its denomination indicates, the γ-eigenvalue is inserted in the Eq.:2.7 as a modification of both the scattering-in and the fission source term in the neutron balance. In operator notation the transport equation with the γ eigenvalue reads:

$$(\hat{L} + \hat{R})\phi = \frac{1}{\gamma}(\hat{S} + \hat{F})\phi \tag{2.11}$$

Likewise the previous eigenvalue type, the set of γ-eigenvalue for which Eq.:2.11 is verified, is referred to as the γ-eigenvalue spectrum whereas the corresponding eigenfunctions ϕ are denoted as γ-modes. The considerations done for κ_0's eigenmodes and spectrum are also valid for γ_0. From a physical standpoint, γ_0 is interpreted as "the ratio between the number of neutrons produced by the scattering-in and the fission source term, and the number of those lost due to collisions and leakage through the system boundaries" [2]. Concerning the criticality condition discussion, a system is referred to as critical if $\gamma_0 = 1$, as subcritical if $\gamma_0 < 1$ and finally as supercritical if $\gamma_0 > 1$ [2].
If, instead of $\hat{S}$ and $\hat{F}$ operators, $\hat{T}$ is taken into account, and $\hat{H}$ is the sum of the right-hand side terms of Eq.:2.11, the latter becomes:

$$\hat{T}\phi = \frac{1}{\gamma}\hat{H}\phi \tag{2.12}$$

With another step, the eigenvalue problem form of the equation is achieved:

$$\gamma\phi = \hat{T}^{-1}\hat{H}\phi \tag{2.13}$$

2.2.3 The α-eigenvalue

It is also referred to as the *decay constant* or the *time eigenvalue* to emphasize the relation between this eigenvalue type (useful to study sub-critical reactors) and the time-dependent behaviour of a neutron population. In this case, a relevant number of higher modes is needed, the fundamental one is no more enough [2]. The starting point is the time-dependent neutron transport equation in its integro-differential form (1.33) without external source and proper, vacuum boundary conditions applied and kept constant in time, together with an initial condition as defined in the previous chapter: these elements form an *initial value problem*, for which the neutron angular flux may be found at any time $t > 0$ as solution of Eq.:1.33. it is demonstrable that this solution may be unique under certain mathematical conditions about cross-sections and the source term [2]. The final form of this equation's solution reads:

$$\phi(z, \mu, E, t) = \phi(z, \mu, E)e^{\alpha t} \tag{2.14}$$

where α may assume all the values that generate a non-trivial solution of the homogeneous time-independent neutron transport equation. The corresponding solutions constitute the eigenfunctions of the problem or α-modes [2]. Hence, by inserting the function 2.14 and the corresponding derivative with respect to time into Eq.:1.33, one obtains:

$$\begin{aligned}
&\frac{\alpha}{v}\phi(z, \mu, E)e^{\alpha t} + \mu\frac{\partial\phi(z, \mu, E)e^{\alpha t}}{\partial z} + \Sigma_t\phi(z, \mu, E)e^{\alpha t} \\
&= \frac{1}{2}\int_{-1}^{1} d\mu' \int dE'\Sigma_s(z, E')\phi(z, \mu', E')e^{\alpha t} f_s(z, E' \Rightarrow E) \\
&+ \frac{\chi(z, E)}{2}\int_{-1}^{1} d\mu' \int dE'\Sigma_f(z, E')\phi(z, \mu', E')e^{\alpha t}\nu(z, E')
\end{aligned} \tag{2.15}$$

Eq.:1.16, in operator formalism, reads:

$$(\hat{T} + \frac{\alpha}{v})\phi = (\hat{S} + \hat{F})\phi \tag{2.16}$$

It may be seen a kind of increment of the total macroscopic cross-section Σ_t by a factor $\frac{\alpha}{v}$, referred to as *fictitious capture* or *time-absorption* term. This latter term may be modulated through the α-eigenvalue, until the criticality is reached [2]. α, differently from the other eigenvalue types, has a unit of measure, the inverse of time. The dissertation of eigenvalues, spectrum and modes is analogous to the ones of the aforementioned factors. Indeed, α_0 is the eigenvalue with the largest real part, and its corresponding solution is the only one to not die out with respect to the higher exponential modes [2]. Thus, the criticality problem reduces to determine the sign of α_0: if it is negative, the system neutron population will decay over time following an exponential behaviour and, as a consequence, the system will be subcritical; on the contrary,the positive sign will make neutron population diverge over time (supercritical system), whereas for $\alpha_0 = 0$ the system is critical [2]. The eigenvalue problem form of Eq.:2.16 reads:

$$\hat{L}\phi + (\hat{R} + \frac{\alpha}{v})\phi = (\hat{S} + \hat{F})\phi \tag{2.17}$$

with a little re-arrangement of terms, i.e. $\hat{B} = (\hat{L} + \hat{R}) - (\hat{S} + \hat{F})$, α stands alone on the right-hand side of the equation:

$$v\hat{B}\phi = \alpha\phi \tag{2.18}$$

2.2.4 The δ-eigenvalue

The so-called effective density factor expresses a modification of the nuclides densities through which criticality may be achieved. Specifically, the δ-eigenvalue problem in operator formalism reads [2]:

$$\hat{L}\phi + \frac{1}{\delta}\hat{R}\phi = \frac{1}{\delta}(\hat{S} + \hat{F})\phi \tag{2.19}$$

in which $\delta\phi$ can be easily isolated:

$$\hat{L}^{-1}[\hat{R} - (\hat{S} + \hat{F})]\phi = \delta\phi \tag{2.20}$$

δ-eigenvalue has influence on both the removal term and on the source terms of the neutron balance, constituted by the scattering-in and the fission contribution. As for the previously presented eigenvalue types, δ-eigenvalue has its own spectrum and set of modes. It might be represented as "the ratio between the difference of the fission and absorption rate, and the leakage rate" [2]. The criticality condition in a nuclear system is characterized by $\delta_0 = 1$, a subcritical system is a reactor such that $\delta_0 < 1$ and, as a consequence, a supercritical system is characterized by $\delta_0 > 1$.

Chapter 3

The stochastic approach to the criticality problem

Solving problems including simultaneously all the known dependencies (space, energy, time and travel direction) requires great efforts. Sections 1.1.3 and 1.1.4 have introduced major simplifications, i.e. the plane geometry and the isotropic scattering. Criticality analysis, regarding the time coordinate, is addressed as an eigenvalue problem, as seen in the previous chapter: time is someway marked by the turnover of neutron generations, even though there is no time dependence. So, it is a "*pseudo-stationary* model for the description of slight changes in reactor temporal behaviour" [2]. However, numerical calculation implies discretization of the variables. For instance, Burrone in [2] analyses a time-independent neutron population in a mono-dimensional, homogeneous environment, with neutrons moving at one speed, while Abrate in [4] adds another energy group. Both works carry out a criticality analysis through the theory of *spherical harmonics* by implementing it through the $\mathrm{P_N}$ approximation method that discretizes the angular coordinate to make the model feasible for a numerical code. It is a deterministic approach based on rigorous mathematical models. The computation of eigenvalues and eigenmodes, in these cases, exploits the "iterative Implicit Restarted Arnoldi Method (IRAM), a suitable way to deal with sparse matrices" [2]. This thesis, instead, aims at following a totally different approach to criticality problems. The Monte Carlo method simulates the random movement in a medium taking advantage of the probability density distributions characteristic of the phenomenon with no discretization of coordinates (except the energy as this chapter illustrates). By repeating this simulation a large number of times, this method mimics physical experiment retrieving the averaged quantity of interest with its uncertainty. Instead of IRAM, the power iteration method is utilized. The main difference between the two methods is that the latter can compute only the spectral radius.

The section 3.1 of this chapter gives an overview of basic probability and statistics principles which Monte Carlo method is based on. The section 3.2, instead, deals with the power iteration method and its application to the criticality problem.

3.1 The theory behind Monte Carlo method

3.1.1 Probability concepts

Every event humans are able to perceive or not, every event humans are able to consciously commit or not has consequences, or outcomes. Outcomes are not nevertheless all equal to each other, even if they might be similar. For instance 'tossing a coin' has a different outcome from 'tossing a coin and check which side can be seen'. The former is 'the coin reaches the ground' and the observer can do an a-priori, deterministic reasoning about what is going to happen to the coin, he does know the outcome; while the latter is 'heads OR tails': the observer cannot know the exact outcome, because it is a *random experiment* that "has different outcomes even though it is repeated in the same manner every time" [5]. Each one of these possible outcomes belongs to the so-called the *sample space* of the experiment and, more precisely, an event is a subset of this space. Set operations may help to create events, such as [5]:

- The union of two events E1 and E2 is the event that consists of all outcomes that are contained in either of the two events. It is denoted as $E1 \cup E2$.
- The intersection of two events E1 and E2 is the event that consists of all outcomes that are contained in both of the two events. It is denoted as $E1 \cap E2$.
- The complement of an event E is the set of outcomes in the sample space that are not in the event E. It is denoted as E'. E and E' has

no intersection naturally, but their union gives out the entire sample space: they are called *exhaustive events*.

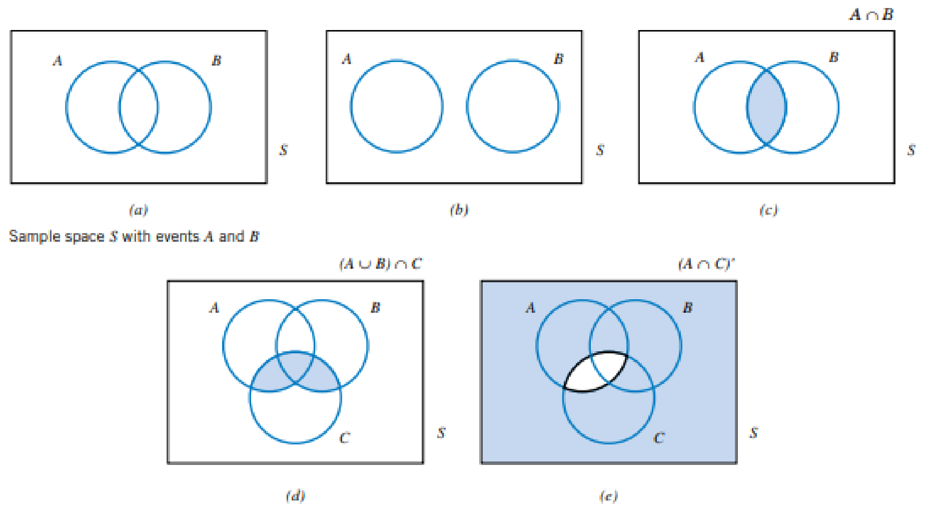


Figure 3.1: Venn diagrams [5].

These operations form a base for more complex combinations, displayable with Venn diagrams, as illustrated in Fig.:3.1.

Although a deterministic argument on a random experiment cannot be asserted, every outcome has its own probability. There are many definitions of probability, but the axiomatic one will be given. It consists in three axioms due to Andrej N. Kolmogorov [5]: "*Probability is a number that is assigned to each member of a collection of events from a random experiment that satisfies the following properties: If S is the sample space and E is any event in a random experiment:*

1. $P(S) = 1$. *It is a sure event, while an impossible event has* $P(E) = 0$
2. $0 \leq P(E) \leq 1$
3. *For two events E1 and E2 with* $E1 \cap E2 = \emptyset$ *(*mutually exclusive events*)* $P(E1 \cup E2) = P(E1) + P(E2)$"

If two events E1 and E2 are not mutually exclusive, their union becomes $P(E1 \cup E2) = P(E1) + P(E2) - P(E1 \cap E2)$. Further complications appear with the union of three or more events [5].

Sometimes an event (E1) may affect the probability of another one (E2) forcing to narrow it down. Probabilities need to be evaluated again as additional information becomes available. This situation is called conditional probability [5]: "*The conditional probability of an event E2 given an event E1, denoted as $P(E2|E1)$, is equal to:*

$$P(E2|E1) = \frac{P(E1 \cap E2)}{P(E1)} \tag{3.1}$$

for $P(E1) > 0$."
For example, if a die is rolled, the probability P(E) of getting 2 is 1/6 (if the die is well-balanced, every die face has the same probability of being the outcome), but given that the outcome is even ($P(F) = 1/2$) there is an additional information and $P(E|F) = \frac{1/6}{1/2}$, so $P(E|F) = \frac{1}{3}$.

If the left-hand side of Eq.:3.1 is multiplied times P(E1) one obtains, after some passages, another important relation, the *Bayes's Theorem*. Firstly:

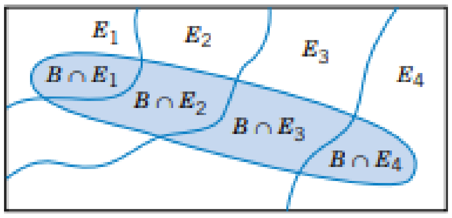
$$P(E1 \cap E2) = P(E2|E1)P(E1) \tag{3.2}$$

Moreover, the equivalency $P(E1 \cap E2) = P(E2 \cap E1)$ is obviously valid and its right-hand side is equal to $P(E1|E2)P(E2)$ from Eq.:3.2. By substituting this into Eq.:3.2 and by letting only $P(E1|E2)$ on its left-hand side, the Bayes's Theorem formula is constituted [5]:

$$P(E1|E2) = \frac{P(E2|E1)P(E1)}{P(E2)} \tag{3.3}$$

Conditional probability is the basis for the consequent the *Total probability rule*, useful when there is a set of N events in a sample space that are mutually exclusive and exhaustive such as the example of Fig.:3.2. Be event B a subset of each one of these events. Then:

$$P(B) = \sum_{i=1}^{N} P(B|E_i)P(E_i) \tag{3.4}$$



$B = (B \cap E_1) \cup (B \cap E_2) \cup (B \cap E_3) \cup (B \cap E_4)$

Figure 3.2: Partitioning an event into several mutually exclusive subsets [5].

"For example, suppose that in semiconductor manufacturing the probability is 0.10 that a chip that is subjected to high levels of contamination during manufacturing causes a product failure. The probability is 0.005 that a chip that is not subjected to high contamination levels during manufacturing causes a product failure. In a particular production run, 20% of the chips are subject to high levels of contamination. What is the probability that a product using one of these chips fails ?"[5] If F is the event 'a product fails', H is 'chip exposed to high levels of contamination' and $P(F|H)$ and $P(F|H')$ their respective conditional probabilities, by applying Eq.:3.4 the following result is got:

$$P(F) = P(F|H)P(H) + P(F|H')P(H') \tag{3.5}$$

that is equal to $0.1 * 0.2 + 0.005 * 0.8 = 0.0235$.
In some cases, the conditional probability of $P(B|A)$ might be equal to $P(B)$. In this special case, the knowledge of the outcome of the event A does not affect the probability of the outcome of event B. Thus, A and B are independent [5]: "*Two events are independent if any one of the three following equivalent statements is true:*

1. $P(A|B) = P(A)$

2. $P(B|A) = P(B)$

3. $P(A \cap B) = P(A)P(B)$

" It is important to focus on the difference between independence and mutual exclusivity: the latter is a strong dependence. Indeed, $P(E \cap F) = P(\emptyset)$ if they are mutually exclusive, while $P(E \cap F) = P(E)P(F)$ if they are independent.

In probability evaluation, it is often necessary to be able to effectively count the number of different ways that a given event can occur and, for sample spaces with large amount of elements, the different ways in which a given event may occur are difficult to handle, because of the dimension of the numbers involved: thus, it is necessary to know the *basic principle of counting.*
"If r experiments that are to be performed are such that the first one may result in any of n_1 possible outcomes, and if for each of these n_1 possible outcomes there are n_2 possible outcomes of the second experiment, and if for each of the possible outcomes of the first two experiments there are n_3 possible outcomes of the third experiment, and so on..., then there are a total of $\prod_{i=1}^{r} n_i$ possible outcomes of the r experiments. For instance, how many *different* ordered arrangements of the letters 'a, b,c' are possible? By direct enumeration it is clear that there are 6; namely, 'abc, acb, bac, bca, cab, cba'. Each one of these ordered arrangements is known as a *permutation.* It is convenient to introduce the factorial notation, n!. In the previous 'a-b-c' example, there are 3! = 6 permutations (it is important to remember that 0! = 1)" [6].
What if the elements to be picked up[1] are three of five items A, B, C, D, E? Since there are 5 ways to select the initial item, 4 ways to then select the next item, and 3 ways to then select the final item, there are thus $5 * 4 * 3$ ways of selecting the group of 3 when the order in which the items are selected is relevant [6]. However, since every group of 3, the group consisting of items A, B, and C, will be counted 6 times (that is, all of the permutations ABC, ACB, BAC, BCA, CAB, CBA will be counted when the order of selection is relevant), it follows that the total number of different groups that can be formed is $(5 * 4 * 3)/(3 * 2 * 1) = 10$ [6]. In general, as the latter expression

[1]Without re-immission

represents the number of different ways that a group of r items could be selected from n items when the order of selection is considered relevant, and since each group of r items will be counted r! times in this count, it follows that the number of different groups of r items that could be formed from a set of n items is the binomial coefficient [6]:

$$\frac{n(n-1)(n-2)(n-r+1)}{r!} = \frac{n!}{(n-r)!r!} = \binom{n}{r} \tag{3.6}$$

This concept has a clear application in probability computation. "For instance, a committee of size 5 is to be selected from a group of 6 men and 9 women. If the selection is made randomly, what is the probability that the committee consists of 3 men and 2 women?" [6] 'Randomly selected' means that each of the $\binom{15}{5}$ possible combinations is equally likely to be selected. Hence, since there are $\binom{6}{3}$ possible choices of 3 men and $\binom{9}{2}$ possible choices of 2 women, it follows that the desired probability is given by [6]:

$$\frac{\binom{6}{3}\binom{9}{2}}{\binom{15}{5}} = \frac{240}{1001} \tag{3.7}$$

3.1.2 Random variables and Sampling

Normally, outcomes are mapped onto numerical values for mathematical treatment [1]. These numerical values are called *random variables* and they might be discrete, like a natural numbers interval from 1 to 6 indicating the possible outcomes of a die tossing, or continuous, a real number interval between 0 and ∞, for instance, standing for the time between two consecutive particles emission from a radioactive material [1]. For any random variable ξ, two functions are defined: the *cumulative distribution function* (or cdf) and the *probability density function* (or pdf).
The cumulative distribution function F can be expressed in terms of the pdf $f(x)$ by $F(a) = \sum_{x \leq a} f(x)$ if ξ is a discrete random variable whose set of possible values are x_1, x_2, x_3, ..., where $x_1 < x_2 < x_3 <$... , then its distribution function F is a 'step function'. That is, the value of F is constant in the intervals $[x_i - 1, x_i)$ and then takes a step (or jump) of size f(x_i) at x_i [6]. If ξ is continuous random variable, then exists a non-negative function

$f(x)$, defined for all real $x \in (-\infty, \infty)$, having the property that for any set B of real numbers:

$$P[\xi \in B] = \int_B f(x)\, dx \tag{3.8}$$

The function $f(x)$ is called the probability density function of the random variable X [2]. In words, Eq.3.8 states that the probability that x will be in B may be obtained by integrating the probability density function over the set B. Since ξ must assume some value, f(x) must satisfy [6]:

$$P[\xi \in (-\infty, \infty)] = \int_{-\infty}^{\infty} f(x)\, dx \tag{3.9}$$

Probabilities about X can be answered in terms of the pdf. For instance, letting $B = [a, b]$, from Eq.3.8:

$$P[a \leq \xi \leq b] = \int_a^b f(x)\, dx \tag{3.10}$$

If a = b in the above, then:

$$P[\xi = a] = \int_a^a f(x)\, dx \tag{3.11}$$

which is zero. In words, this equation states that "the probability that a continuous random variable will assume any *particular* value is zero" [6]. The relationship between the cdf and the pdf is expressed by:

$$F(a) = P[-\infty \leq \xi \leq a] = \int_{\infty}^{a} f(x)\, dx \tag{3.12}$$

Differentiating both sides yields:

$$\frac{\partial F(a)}{\partial a} = f(a) \tag{3.13}$$

So, the pdf is the derivative of the cdf, or better:

$$P[a - \frac{\varepsilon}{2} \leq \xi \leq a + \frac{\varepsilon}{2}] = \int_{a-\frac{\varepsilon}{2}}^{a+\frac{\varepsilon}{2}} f(x)\, dx \approx \varepsilon f(a) \tag{3.14}$$

[2]The probability density functions may also have more than one variable

when ε is small. In other words, "the probability that X will be contained in an interval of length ε around the point a is approximately $\varepsilon f(a)$" [6], a sort of measure of how likely it is that the random variable is in the neighbourhood of a. In order to summarize the features of this couple of functions, it is useful to add the following sum-up. Being ξ the random variable:

- $P[\xi < x] = F_\xi(x^-)$.
- $P[\xi = b] = F_\xi(b^+) - F_\xi(b^-)$.
- $P[\xi \leq b] = F_\xi(b^+)$.
- $P[a < \xi \leq b] = F_\xi(b) - F_\xi(a)$.
- $P[a \leq \xi \leq b] = F_\xi(b) - F_\xi(a^-)$.
- $P[a \leq \xi < b] = F_\xi(b^-) - F_\xi(a^-)$.
- $P[a < \xi < b] = F_\xi(b^-) - F_\xi(a)$.
- The pdf is always positive.
- The cdf is always positive and non-decreasing (so, it is invertible).
- The pdf is normalized such that its corresponding cdf varies in range [0,1].
- If the cdf discontinuous, it is continuous from the right.

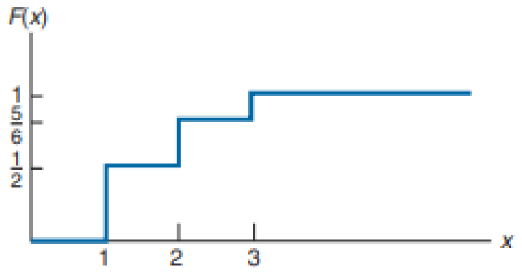


Figure 3.3: An example of step-function cdf [5].

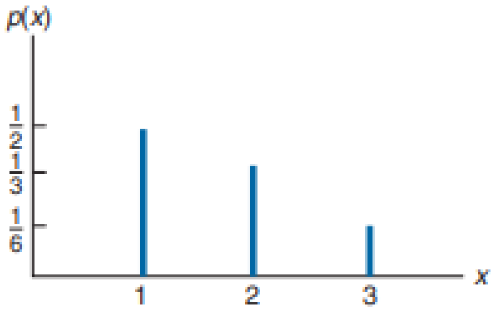


Figure 3.4: The corresponding pdf of the cdf in Fig.3.3 [5].

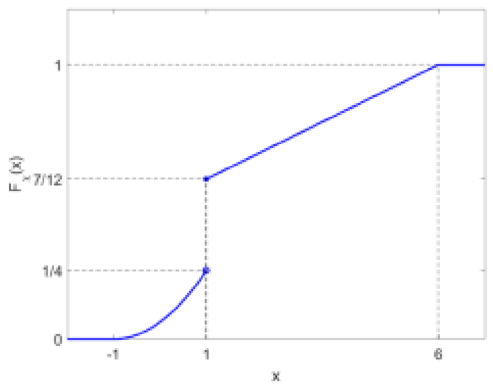


Figure 3.5: An example of continuous, but piecewise cdf [7].

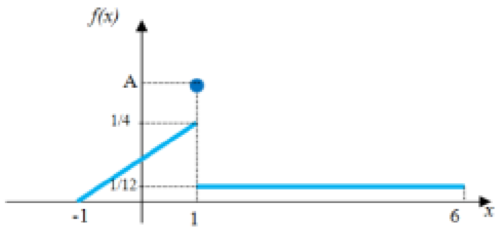


Figure 3.6: The corresponding pdf of the cdf in Fig.3.5 [7].

Figs.3.3-3.4-3.5-3.6 provide an example of two couples of pdf and cdf, the former for a discrete random variable, the latter for a continuous one.

Random numbers sequences have a special feature: no one can predict η_n based on the previous η_{n+1} numbers in the sequence. The section 3.1.3 will concern the specific issue of randomness of numbers and their practical generation. In order to execute a good Monte Carlo simulation, this generation must yield numbers uniformly distributed in a defined range, commonly [0,1] [1]. Random numbers might be seen as a random variable η whose pdf is constant:

$$q(\eta) = 1 \tag{3.15}$$

with $0 \leq \eta \leq 1$.
Therefore, the corresponding cdf reads as:

$$Q(\eta) = \int_0^\eta q(\eta')\, d\eta' \tag{3.16}$$

with $0 \leq \eta \leq 1$.
In a Monte Carlo simulation, physical processes have known probability density functions from which random variable (x) is obtainable and it may be written as [1]:

$$p(x)dx = q(\eta)d\eta \tag{3.17}$$

with $0 \leq \eta \leq 1$ and $a \leq x \leq b$.
Then, one can integrate both sides of Eq.:3.15 respectively over [a, x] and [0,η] to get:

$$\int_a^x p(x')\, dx' = \int_0^\eta 1\, d\eta' \tag{3.18}$$

that is equivalent to write:

$$P(x) = \eta \tag{3.19}$$

Eq.:3.18 gives a relation for getting a continuous random variable x using a random number η. This is the so-called "*Fundamental Formulation of Monte Carlo* (FFMC)" for continuous random variable [1]. If the latter is discrete and assumes certain values, while the random number η is always a continuous variable, the following relation has to be imposed:

$$Min[P(n)|P(n) \geq \eta] \tag{3.20}$$

where $P(n) = \sum_{i=1}^{n} p_i$. This means that n is selected when the minimum of P(n) is greater than or equal to η [1]. Many approaches for solving FFMC exist, but the focus will be in particular on two of them: the *analytical*

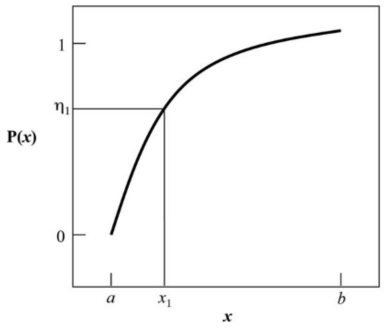


Figure 3.7: Sampling a continuous random variable x [1].

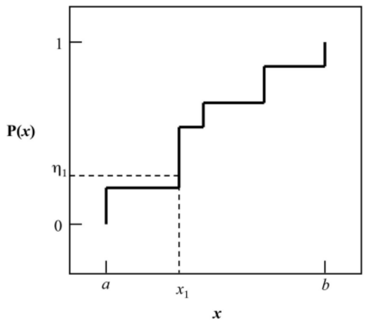


Figure 3.8: Sampling a discrete random variable x_i [1].

inversion and the *rejection technique*. Figs.3.7-3.8 depict the two kinds of FFMC: the former for a continuous random variable, the latter for a discrete one.

In the analytical inversion, the FFMC is inverted to obtain a formulation for a random variable x in terms of a number $\eta \in [0,1]$. Mathematically, this means to have an inverse formulation, $x = P^{-1}(\eta)$. For instance, if the pdf for the random variable x is given by [1]:

$$p(x) = 1/2 \tag{3.21}$$

for $-1 \le x \le 1$.
Then, corresponding FFMC formulation is:

$$\int_{-1}^{x} 1/2\, dx' = \eta \tag{3.22}$$

And x or $P^{-1}(\eta)$ is given by:

$$x = 2\eta - 1 \tag{3.23}$$

If the exact computation of $P^{-1}(\eta)$ is not easy, the rejection technique may be the alternative and four steps have to be followed [1]:

1. Enclose p(x) in a frame bounded by p_{max}, a, and b, as shown in Fig.3.9.
2. Generate two random numbers: η_1 and η_2
3. Sample the random variable x by using:
$$x = a + \eta_1(b - a) \tag{3.24}$$
4. Accept x if:
$$\eta_2 p_{max} \leq p(x) \tag{3.25}$$

In this technique, all the pairs(x, $y = \eta_2 p_{max}$) are accepted if they are under the graph of p(x), otherwise, they are rejected. So, the sample is effectively from the area under the pdf, i.e. the cdf [1]. Because the technique samples from the area, an efficiency might be stated:

$$\varepsilon = \frac{\int_a^b p(x)\,dx}{p_{max}(b-a)} = \frac{1}{p_{max}(b-a)} \tag{3.26}$$

This technique may be very slow for low values of efficiency [1].

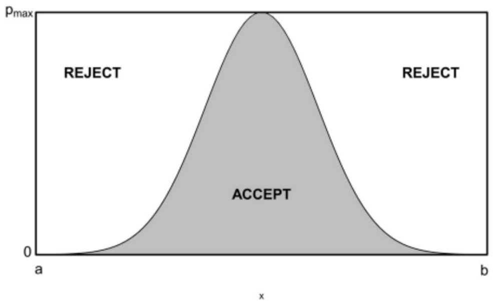


Figure 3.9: Demonstation of the rejection technique [1].

3.1.3 Random number generation

The quality of a Monte Carlo simulation is strictly connected to the quality, or randomness, of the random numbers used. Their generation is implemented through an experimental way (draw balls from an urn, measure the distance of a dart from the centre in a dart game, etc.) or through an algorithmic one. For the latter, this approach is called *pseudo random number generator* (PRNG)[3] [4]and its associated numbers *pseudo random numbers* (PRN). Both have advantages and disadvantages and there are six factors decreeing which approach is better than the other one [1]:

1. *Randomness*: random numbers should assume a uniform distribution. In the experimental approach, it is achieved if the procedure follows a uniform distribution, in the algorithmic one, the sequence has to satisfy several statistical tests.

2. *Reproducibility*: the random number sequence must be reproduced multiple times.

3. *Length of the sequence of the random numbers*: a Monte Carlo simulation for realistic engineering problems needs millions of random numbers.

4. *Computer memory*: the generator should not consume too much computer memory.

5. *Generation time*: the amount of time that it takes to generate a sequence of random numbers should not be significant, e.g., days/months.

6. *Computer time*: the amount of computer time needed to generate the sequence should be significantly shorter than the actual simulation.

The algorithmic approach results to be preferable mainly because its sequence is reproducible and it requires minimal effort (in terms of computer resources and engineer time) [1].
There are two common types of PRNGs: *congruential generators* and *multiple recursive* ones.

[3] *"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin"*- John von Neumann (1951) [1]

[4] *"Random number generators should not be chosen at random"*- Donald Knuth (1986) [1]

Congruential generators

They are integer generators and use the following formulation:

$$x_{k+1} = (ax_k + b), modM \tag{3.27}$$

with b $<$ M.
x_0 is called the *seed*, while a, b, and M are given integers. M is the largest integer representable by a computer, e.g., on binary machine with a 64-bit word length, the largest unsigned integer is $2^{64} - 1$ and the largest signed one is $2^{63} - 1$. The modulus function determines the remainder of the right-hand side of Eq.:3.26 divided by M. If $b \neq 0$, Eq.:3.26 is called *called linear congruential generator*, if $b = 0$, *multiplicative congruential generator*. This method gives integer x in the range [0, M-1]. To convert the random integer generated here into a random number in the range of [0,1), the relationship $\eta = \frac{x}{M-1}$ is used [1]. The sequence shown in Fig.:3.10 is a cycle with $a = 5$,

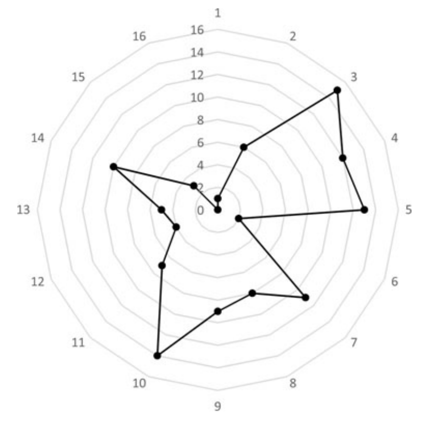


Figure 3.10: Schematic random number cycle for a linear congruential generator [1].

$b = 1$ and $M = 16$, so if the seed changes, only the starting point of the sequence will change. The variation of the multiplier a and of the constant b instead affects the period of PRNG. The obvious goal is to achieve the longest period as possible (M), a full period. Table:3.1 presents the properties a linear congruential generator must have to be 'full-period'. On the other hand, it is demonstrable that a multiplicative congruential generator can only have a

Parameter	Value	Comment
Multiplier (a)	$4N+1$	$N>0$
Contact (b)	odd number	-
Modulus (M)	2^k	$k>1$

Table 3.1: properties of the parameters in a linear congruential generator for achieving full period [1].

partial period [5] (for some values of the multipliers it is 4, for other values is even 2). In summary, a congruential generator with a reasonable period is quite good for executing most of physical simulations, because the physical system introduces randomness by applying the same random numbers to different phenomena [1].

Multiple recursive generators

The multiple recursive generators usually give better results in terms of period[6]. A group of them may be expressed by:

$$x_{k+1} = (a_0 x_k + a_1 x_{k-1} + a_j x_{k-j} + b), modM \tag{3.28}$$

The initial $j+1$ random numbers are selected from simpler generators. The length and the randomness of the generator depend on the values a_j, b and M. One example is the Fibonacci generator that is a good choice for large problems, is floating point. It computes a new number of the sequence by combination (sum, difference, or product) of two preceding terms [1]. For instance:

$$x_k = x_{k-31} - x_{k-13} \tag{3.29}$$

is a Fibonacci generator of lags 31 and 13. Its expected period is $p = (2^{31} - 1)2^n$ with n being the number of bits in the mantissa of x_i. For example, for 32-bit floating point arithmetic, $n = 24$; hence, $p \approx 2^{41}$ or 10^{12}. To start this kind of generator, the 'pre-generation' of the initial random numbers is

[5]In order to maximize the period, M must be a prime number and the multiplier a primitive of M [1]

[6]Python exploits the Mersenne Twister, a type of multiple recursive generator based on the homonymous prime number: it has a period of $2^{19937} - 1$ [8]

needed (in the case of Eq.:3.28 31 numbers). One approach is to represent each of them in their binary form [1]:

$$r = \frac{r_1}{2} + \frac{r_2}{2^2} + \frac{r_m}{2^m} \tag{3.30}$$

for $m \leq n$(mantissa).
Each bit r_i (0 or 1) must be generated, e.g. by means of a linear congruential generator: for instance, r_i might be set to either 0 or 1 depending on whether the output of the congruential generator is greater or less than zero. So, the quality of this simpler generator will state the quality of the Fibonacci generator. It is worth noting that the largest period that a congruential generator can have when using 32-bit format is 2^{32}, or $4.3 * 10^9$ which is significantly smaller than the one of the generator described in Eq.:3.28.

There is a lot of randomness useful tests for evaluating PRNGs [7]. Considering congruential generators , the aims of all of them is to assess the effect of modulating seed, multiplier and constant. The parameters for this assessment are the period of the sequence, as previously seen, and the average, i.e.the first moment of the random numbers (explained in the next section), that must be 0.5 [1].
An impactful way of visualizing the quality of a PRNG is to generate 3-tuple spatial positions using every three consecutive random numbers of the sequence and then mark the positions in a three-dimensional domain, as done in the next three figures.

It is possible to note a strongly correlated set of random numbers in

[7]One of the most important is the $\chi^2 - Test$ [1]

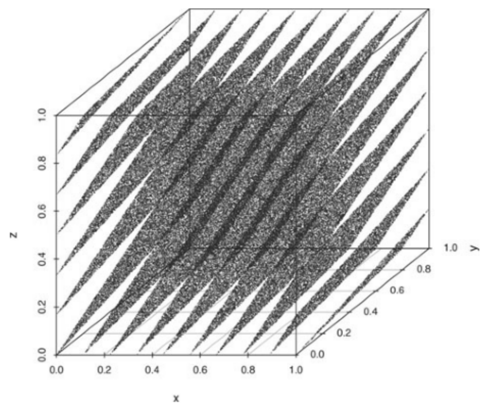


Figure 3.11: 3-tuple distribution of random numbers for a linear congruential generator with $a = 65539$, $b = 1$, seed$= 1$ and mod$= 2^{24}$ [1].

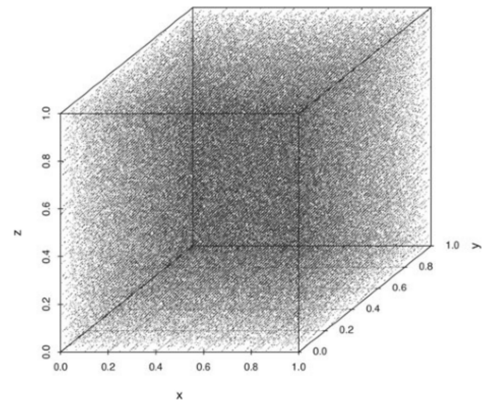


Figure 3.12: 3-tuple distribution of random numbers for a linear congruential generator with $a = 65541$, $b = 1$, seed$= 1$ and mod$= 2^{24}$ [1].

Fig.:3.11 whose generator does not reach a full period. A milder correlation may be seen in Fig.:3.12's sequence. Fig.:3.13's sequence, finally, has no visible correlation.

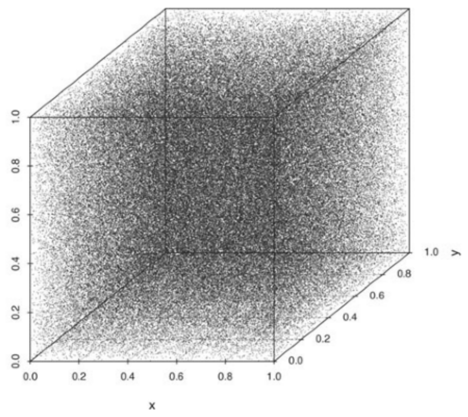


Figure 3.13: 3-tuple distribution of random numbers for a linear congruential generator with $a = 16333$, $b = 1$, seed= 1 and mod= 2^{24} [1].

3.1.4 The Monte Carlo simulation

Statistical procedures are basilar in dealing with random processes. They allow to describe and indicate trends and expectations with a related degree of reliability. In a few words, statistics make use of scientific methods of sampling (collecting and analysing) and interpreting data when the population (in terms of pdf) is unknown. Statistics theory is based on probability theory. Through the latter, one may determine the likelihood that an unknown sample has certain characteristics. The former deals with sampling an unknown population in order to estimate its composition, i.e. probability density. As previously mentioned, the Monte Carlo approach is a statistical method that uses random numbers to sample an unknown population, e.g. particle history, and, consequently, evaluates the expected outcome of a physical process [1].

The first tool to be handled is the *expectation operator*: given a continuous random variable x and its pdf p(x) defined in a range [a, b], the *expectation value* (or *true mean*) of any function g(x) defined in this interval is the following application of the expectation operator.

$$E[g(x)] = \int_a^b p(x)g(x)\,dx \tag{3.31}$$

If the function which the expectation operator is applied to is the random variable itself, the results will be the true mean of x [1]:

$$m_x = E[x] = \int_a^b p(x)x\,dx \tag{3.32}$$

for $a \leq x \leq b$.
In case of a discrete random variable x_i of N outcomes, the formulations of Eqs.3.30-3.31 become [1]:

$$E[g(x_i)] = \sum_{i=1}^{N} p(x_i)g(x_i) \tag{3.33}$$

and

$$m_x = E[x_i] = \sum_{i=1}^{N} p(x_i)x_i, i = 1, N \tag{3.34}$$

For now, only the *first order moment* of random variable x has been taken into account, but the expected value of generic the *k-th power* of x, for continuous random variable, is given by [1]:

$$m_x^k = E[x^k] = \int_a^b p(x)x^k\,dx \tag{3.35}$$

And, for the discrete random variable case, by:

$$m_x^k = E[x_i^k] = \sum_{i=1}^{N} p(x_i)x_i^k \tag{3.36}$$

Additionally, there is also the definition of the *k-th central moment* of random variable x for the two usual situations [1]:

$$E[(x - m_k)^k] = \int_a^b p(x_i)(x - m_x)^k\,dx \tag{3.37}$$

and

$$E[(x_i - m_x)^k] = \sum_{i=1}^{N} p(x_i)(x_i - m_x)^k \tag{3.38}$$

If $k = 2$, the central moment is referred to as the *true variance* of x, expressed by [1]:

$$\sigma_x^2 = E[(x - m_x)^2] = \int_a^b p(x)(x - m_x)^2 \, dx \tag{3.39}$$

and

$$\sigma_x^2 = E[(x_i - m_x)^2] = \sum_{i=1}^{N} p(x_i)(x_i - m_x)^2 \tag{3.40}$$

A more convenient way of computing the true variance is derived from the expansion of the quadratic term [1]:

$$\sigma_x^2 = E[x^2] - E[x]^2 \tag{3.41}$$

Another useful quantity is the square root of the variance, referred to as the *true standard deviation*, i.e [1]:

$$\sigma_x = \sqrt{\sigma_x^2} \tag{3.42}$$

It is an indication of the dispersion of random variable x relative to its mean m_x. The true mean and the true variance are the *population parameters* because they are obtained from a known pdf [1].

Moreover, the linearity of the expectation operator implies the following relationships, with a and b being constant coefficient:

- $E[ag(x) + b] = aE[g(x)] + b$
- $E[ag(x_1) + bg(x_2)] = aE[g(x_1)] + bE[g(x_2)]$
- $E[\sum_{i=1}^{N} a_i g(x_i)] = \sum_{i=1}^{N} a_i E[g(x_i)]$
- $\sigma^2[ag(xi)] = a^2\sigma^2[g(x)]$
- $\sigma^2[\sum_{i=1}^{n} a_i g(x_i)] = \sum_{i=1}^{n} a_i^2 \sigma^2[g(x_i)]$

These relations are useful, for instance, to deal with a combination of two random variables such as $x_3 = c_1x_1 + c_2x_2$. Its true mean and variance are respectively [1]:

$$E[x_3] = c_1m_{x_1} + c_2m_{x_2} \tag{3.43}$$

and

$$\begin{aligned}\sigma^2 = \sigma[c_1x_1 + c_2x_2] = E[(c_1x_1 - c_1m_{x_1})^2] + E[(c_2x_2 - c_2m_{x_2})^2] \\ -2E[(c_1x_1 - c_1m_{x_1})(c_2x_2 - c_2m_{x_2})]\end{aligned} \tag{3.44}$$

which is equal to:

$$\sigma^2 = c_1^2E[(x_1-m_{x_1})^2]+c_2^2E[(x_2-m_{x_2})^2]+2c_1c_2E[(x_1-m_{x_1})(x_2-m_{x_2})] \tag{3.45}$$

and, finally, to:

$$\sigma^2 = c_1^2\sigma_{x_1}^2 + c_2^2\sigma_{x_2}^2 - 2c_1c_2E[(x_1 - m_{x_1})(x_2 - m_{x_2})] \tag{3.46}$$

where $E[(x_1 - m_{x_1})(x_2 - m_{x_2})] = \mathrm{cov}(x_1, x_2)$, the *co-variance*, that helps to define the *correlation coefficient* between the two random variable as follows [1]:

$$\rho_{x_1,x_2} = \frac{\int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 p(x_1, x_2)(x_1 - m_{x_1})(x_2 - m_{x_2})}{\sigma_{x_1}\sigma_{x_2}} \tag{3.47}$$

If x_1 and x_2 are independent ($p(x_1, x_2) = p_1(x_1)p_2(x_2)$), the correlation coefficient formulation reduces to [1]:

$$\rho_{x_1,x_2} = \frac{(m_{x_1} - m_{x_1})(m_{x_2} - m_{x_2})}{\sigma_{x_1}\sigma_{x_2}} = 0 \tag{3.48}$$

Consequently, the formulation of the variance of x_3 changes:

$$\sigma^2 = c_1^2\sigma_{x_1}^2 + c_2^2\sigma_{x_2}^2 \tag{3.49}$$

On the other hand, if x_1 and x_2 are dependent on each other, e.g, $x_1 = \alpha x_2$, then correlation coefficient formulation becomes [1]:

$$\rho_{x_1,x_2} = \frac{(\alpha x_2 - \alpha m_{x_2})(x_2 - m_{x_2})}{\alpha\sigma_{x_2}\sigma_{x_2}} = 1 \tag{3.50}$$

with the new true variance of x_3 defined as:

$$\sigma^2 = (\alpha c_1 + c_2)\sigma_{x_2}^2 \tag{3.51}$$

Unknown populations are Statistics field of interest. Hence, true mean and true variance are no longer meaningful concepts. They are substituted by the *sample mean* and the *sample variance*. Given a sample of size N, the formulation of the sample mean is [1]:

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{3.52}$$

As $N \to \infty$, the sample mean must approach the true one. If one applies the expectation operator to the sample mean, the following relation will be obtained:

$$E[\overline{x}] = \frac{1}{N}\sum_{i=1}^{N} E[x_i] = \frac{1}{N}Nm_x = m_x \tag{3.53}$$

So, sample mean is a good estimation of the true mean [1].
Based on the definition of variance of Eq.:3.40, the sample variance is:

$$s_x^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2 \tag{3.54}$$

Now the expectation operator is applied to both the sides of Eq.:3.54 and, inside the parenthesis of its right-hand side, m_x is added and subtracted in order to express the sample variance as function of the true variance [1]:

$$E[s_x^2] = E[\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2] = E[\frac{1}{N}\sum_{i=1}^{N}(x_i - m_x + m_x - \overline{x})^2] \tag{3.55}$$

which is equal to:

$$\begin{aligned} E[s_x^2] = \frac{1}{N}\sum_{i=1}^{N} E[(x_i - m_x)^2] + \frac{1}{N}\sum_{i=1}^{N} E[(\overline{x} - m_x)^2] \\ +2E[(m_x - \overline{x})\frac{1}{N}\sum_{i=1}^{N}(x_i - m_x)] \end{aligned} \tag{3.56}$$

After some passages, one obtains [1]:

$$E[s_x^2] = \frac{1}{N}\sum_{i=1}^{N}\sigma_x^2 + \frac{1}{N}NE[(\overline{x} - m_x)^2] - 2E[(\overline{x} - m_x)^2] \tag{3.57}$$

and then:

$$E[s_x^2] = \sigma_x^2 - E[(\overline{x} - m_x)^2] \tag{3.58}$$

The second term on the right-hand side of Eq.:3.58 is the variance of the sample average $\overline{x}$. Hence, as consequence of that [1]:

$$\sigma_{\overline{x}}^2 = \sigma[\frac{1}{N}\sum_{i=1}^{N} x_i] = \sum_{i=1}^{N} \frac{1}{N^2}\sigma^2[x_i] = \frac{\sigma_x^2}{N} \tag{3.59}$$

So, an important fact arises from Eq.:3.59: the variance of the sample average decreases with the growth of the sample size. Now, it is possible to substitute Eq.:3.59 into Eq.:3.58 and get [1]:

$$E[s_x^2] = \frac{N-1}{N}\sigma_x^2 \tag{3.60}$$

The above equation demonstrates that the sample variance formulation (Eq.:3.54) is not a good estimate of the true variance. Thus, to define an unbiased formulation for the sample variance, the above equation may be re-written as:

$$E[\frac{N}{N-1}s_x^2] = \sigma_x^2 \tag{3.61}$$

This means indeed that the term of Eq.:3.61 which the expectation operator is applied to, yields an unbiased estimate of the true variance. Therefore, the *unbiased sample variance* S_x^2 is equal to [1]:

$$S_x^2 = \frac{N}{N-1}s_x^2 \tag{3.62}$$

and, consequently, to:

$$S_x^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2 \tag{3.63}$$

The precision associated with the sample average is measured by the *relative statistical* uncertainty or *relative standard deviation*:

$$R_x = \frac{\sigma_x}{\overline{x}} \tag{3.64}$$

On the other hand, the accuracy is the measure of the deviation from the true mean. The accuracy is generally retrieved by performing an experiment or by consulting the literature or the results of another technique that is known to be accurate [1].

Name	Pdf	True mean	True variance
Uniform	$p(x) = \frac{1}{b-a}$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
Bernoulli	$p(n) = p^n(1-p)^{1-n} n = 0, 1$	p	$p - p^2$
Binomial	$p(n) = \binom{N}{n} p^n(1-p)^{N-n} n = 1, N$	Np	$Np(1-p)$
Geometric	$p(n) = (1-p)^{n-1}p$	$\frac{1}{p}$	$\frac{1}{p^2} - \frac{1}{p}$
Poisson	$p(n) = \frac{m^n}{n!}e^{-m}$	m	m
Normal	$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-m)^2}{2\sigma^2}}$	m	σ

Table 3.2: Commonly used density functions [1].

Table:3.2 presents some probability density functions often encountered when dealing with random physical process. The Bernoulli process refers to a random process with only two outcomes whose probabilities remain constant [1].
The binomial density function is referred to a Bernoulli process repeated N times. The sum of the *successes* (only successful experiment with outcome 1 are worth) follows the binomial distribution[8].
The geometric density function gives the probability of achieving a success after $(n-1)$ failures [1].
The Poisson density function may be approached by a binomial density function when $p << 1$, $N >> 1$ and $n << N$.
And finally, the well-known normal density function, with a key role in statistics field. Fig.:3.14 shows an example by depicting both the pdf and cdf. In this case, the mean value is 40 and the variance is 10. Its main characteristics may be summarized in [1]:

- Maximum at the mean value.
- Symmetric about the mean.

[8]If $N \to \infty$, the binomial density function approaches Normal density function [1]

- Points of inflection (where the second order derivative is null) at one standard deviation away from the mean and there, its value is about 60% of the maximum.
- Tangents to the distribution curve at the inflection points intersect the x-axis at $x = m_x \pm 2\sigma$.
- The half-maximum value is at $x = 1.177\sigma$.
- The $\frac{1}{e}$ of maximum value is at $x = 1.414\sigma$.

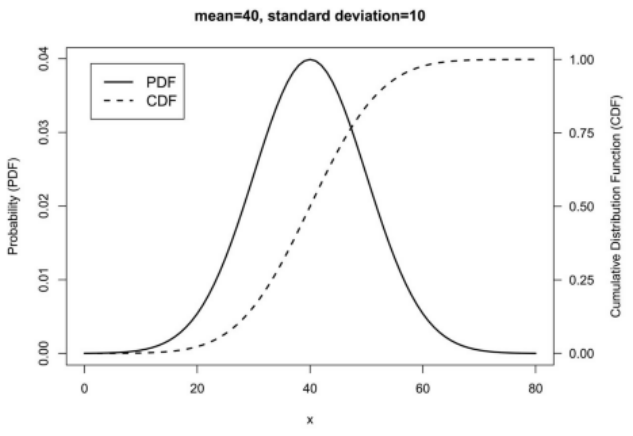


Figure 3.14: Example of normal density function and its cdf [1].

Another important point is the definition of confidence levels, i.e. the probability that the estimation of the sample average lies at a certain distance from the true mean: if the random variable follows the normal density function, since it is symmetric, this probability may be determined within a number of $\pm\ \sigma$ as follows: $P[m_x - n\sigma_x \leq x \leq m_x + n\sigma x]$, where n is the number of standard deviations. This probability is equivalent to [1]:

$$P[m_x - n\sigma_x \leq x \leq m_x + n\sigma x] = \frac{1}{\sqrt{2\pi\sigma_x^2}} [\int_{-\infty}^{m_x+n\sigma_x} dx e^{\frac{-(x-m_x)^2}{2\sigma_x^2}} - \int_{-\infty}^{m_x-n\sigma_x} dx\, e^{\frac{-(x-m_x)^2}{2\sigma_x^2}}] \tag{3.65}$$

The above integrals are the cumulative density functions, i.e. $P(m_x + n\sigma_x)$ and $P(m_x - n\sigma_x)$ respectively.By symmetry, it can be written:

$$P[m_x - n\sigma_x \leq x \leq m_x + n\sigma x] = 2P(m_x + n\sigma_x) - 1 \tag{3.66}$$

- For $n = 1$, Pr$= 68,3\%$.
- For $n = 2$, Pr$= 95,4\%$.
- For $n = 3$, Pr$= 99,7\%$.

The above probabilities indicate that a certain percentage of random variables lie within one, two or three standard deviations from the mean, respectively, and they are true for any normal distribution because of its normalization [1].

It is crucial , at this point, to introduce the fundamental *Central Limit Theorem*: considering the execution of T trials, each of these trials consisting in N independent samples from a common density function with existing mean m_x and variance σ_x^2: then, for any fixed value of N histories per each trial, there is a pdf $f_N(x)$ that describes the distribution of the sample means after repeating these trials. As $N \to \infty$, the Central Limit Theorem states that there is a limiting density function for x and is a normal density function given by [1]:

$$f_N(\overline{x}) = \frac{1}{\sqrt{2\pi\sigma_{\overline{x}}^2}} e^{-\frac{(\overline{x}-m_x)^2}{2\sigma_{\overline{x}}^2}} \tag{3.67}$$

where $\sigma_{\overline{x}}^2 = \frac{\sigma_x^2}{N}$ (Eq.:3.59). The applicability to any random variable with well-defined m_x and σ_x^2 and the emphasis on the sample mean uncertainty rather than the random variable itself make this theorem the most important instrument for the assessment of a Monte Carlo simulation [1].

3.2 Monte Carlo method in transport theory

The previous sections of this chapter have introduced the theoretical grounds of the tools useful to execute the analysis. The next sections aim at illustrating the specific application of these tools to the physical models presented in the first chapter. Firstly, the focus will be directed on the way of solving an eigenvalue problem by means of the power iteration method, as said at the beginning of the chapter; subsequently on the phases of a neutron's 'life'.

3.2.1 The power iteration method

The most common way of solving an eigenvalue problem is the *power iteration technique* (or *Von Mises iteration*) in which the power (in this case of interest the neutron source) is iterated on until it converges within a prescribed tolerance [1]. From this point on, the computation of the random variables (κ_0 or γ_0) for the Monte Carlo simulation shall be executed. Chapter 2 has shown how to obtain the equation for the eigenvalue problem, and Eqs.2.10-2.13 are those which this thesis is focused on. Both of them might be generalized by the following form:

$$\lambda\phi = \hat{M}\phi \tag{3.68}$$

The $\hat{M}$ operator of 3.68 must be a matrix, even with complex elements, defined in a certain vector field, that is diagonalizable: it implies, among other things that has n, distinct eigenvalues[9]. Their absolute values can be sorted in decreasing order: $|\lambda_0| > |\lambda_1| > |\lambda_{n-1}|$. If ϕ_0 is a vector such that its projection on κ_0's eigenspace is not null, then a sequence arises [9]:

$$\phi_k = \hat{M}\phi_{k-1}, k \geq 1 \tag{3.69}$$

It represents the continuous, iterative applications of the $\hat{M}$ operator to the neutron source, and it can be demonstrated that it tends to the eigenvector (eigenfunction in this case) associated to the greatest eigenvalue, i.e. λ_0; k is the generation counter. Since $\hat{M}$ is diagonalizable by hypothesis (it has n distinct eigenvectors, as many as the eigenvalues), each function of the function field on which the $\hat{M}$ operator is defined might be expressed as a linear combination of the n eigenfunctions v_i of $\hat{M}$ (they form a base for the vector field of $\hat{M}$): $\phi_0 = \sum_{i=0}^{n-1} \alpha_i v_i$. Consequently, by definition of eigenvalue of a matrix, the k-th iteration may be written as [9]:

$$\phi_k = \hat{M}^k\phi_0 = \sum_{i=0}^{n-1} \alpha_i \hat{M}^k v_i = \sum_{i=0}^{n-1} \alpha_i \lambda_i^k v_i \tag{3.70}$$

Now, it is possible to make the ratio between two consecutive powers of the sequence expressed in Eq.:3.69, taking the non-zero, j-th components of ϕ and v (to be indicated with a subscript, while the counter of generations

[9]From the fundamental theorem of algebra and corollaries [9]

is a superscript between parentheses, such as i, the counter of the eigenvalues/eigenvectors) [9]:

$$\frac{\phi_j^{(k+1)}}{\phi_j^{(k)}} = \frac{\lambda_0^{k+1}}{\lambda_0^k}\frac{R^{k+1}}{R^k} \tag{3.71}$$

where $R^{k,k+1} = \alpha_0 v_j^{(0)} + \sum_{i=1}^n \alpha_i (\frac{\lambda_i}{\lambda_0})^{k,k+1} v_j^{(i)}$ [10]. Since $\lambda_0 > \lambda_i$ for every value of $i \geq 1$, for $k \to \infty$, $\frac{R^{k+1}}{R^k} \to 1$ (the argument of the sum tends to 0) and the ratio between $\phi_j^{(k+1)}$ and $\phi_j^{(k)}$ will tend to λ_0.
Thus, the method converges slowly if there is an eigenvalue close in magnitude to the dominant eigenvalue [9]. This might be measured by the Dominance Ratio, i.e. $|\frac{\lambda_1}{\lambda_0}|$: if it is too close to 1 i.e., *High Dominance Ratio* (HDR), the convergence will be slow [1].

3.2.2 Neutron interaction models

Power iteration method concerns operators (leakage, removal, scattering and fission as Chapter 2 has illustrated) which are applied to to functions (the angular flux of the n-th generation of neutrons). In a Monte Carlo simulation instead, the concept of collective flux is substituted by the 'life', referred to as *random walk*, of each single neutron that forms the source for a certain generation, and the operators consists in random experiments that affect the coordinates of the single neutron in the phase space (position, energy and direction). Random walks are nothing but the continuous, random alternation of the states that are going to be described. This last section shows how the theoretical concepts analysed in previous sections are applied to the neutrons transport problem of this thesis. The next chapter will show how this sort of 'instructions' has been given to the neutrons through the codes.

Neutron flight

In some sense, the leakage operator is replaced by the so-called *free flight* of the neutrons, the state immediately previous to every kind of interaction. As a stochastic phenomenon, it is described by a proper pdf:

$$f_\Sigma(x) = \Sigma_t(x) e^{-\int_0^x \Sigma_t(x')\,dx'} \tag{3.72}$$

[10]Being k without surrounding parentheses an exponent, not an index!

The pdf expressed in Eq.:3.72 , if multiplied times dx, represents the probability of the intersection of two independent events, thus, it is a product of probabilities: the one of having a collision between x and $x+dx$ ($\Sigma_t dx$) after experiencing no collision between 0 and x ($e^{-\int_0^x \Sigma_t(x')\,dx'}$). In this thesis, the change in space of the cross section is not handled in this integral way [11]. So, the actual form of the pdf in Eq.:3.69 is:

$$f_\Sigma(x) = \Sigma_t e^{-\Sigma_t x} \tag{3.73}$$

The mean value of this pdf is $\frac{1}{\Sigma_t}$ and it is also called *mean free path* and may be used as unit of measure of the slab length. As Chapter 1 has explained, the cross section is function of neutron's energy, and it has already been said that the energy levels are discretized in this work. Therefore, every energy group has its own set of cross sections (shown in Chapter 4). The value of the distance covered by a neutron, since it depends on cross section, is a function of the energy of the neutron: the faster is the neutron, the greater the distance covered will (statistically) be. It may be obtained through analytical inversion: being $F_\Sigma(x) = 1 - e^{-\Sigma_t x}$ the cdf, its inversion, as explained in Eq.:3.22, results:

$$\frac{1 - \ln \eta}{\Sigma(E)} \tag{3.74}$$

Fig.:3.15 depicts the charts of pdf and cdf for two different cross sections (from two different energy groups) used for the computations in the next chapters. The total cross section in the faster group is equal to 0.88721 cm^{-1}, while the other one to 2.9727 cm^{-1} [10].

Collision

At the end of the free flight, many of things may happen: there is a collision, and it ends up with an absorption or with a scattering event. This 'either/or' situation is perfectly described by a Bernoulli experiment (Table.3.2): in this case $p = \frac{\Sigma_a(E)}{\Sigma_t(E)}$ or $p = \frac{\Sigma_s(E)}{\Sigma_t(E)}$, since they are complementary events. For instance, the first option is chosen: to sample this pdf, one must start from the relation 3.20. If this condition is respected, the neutron is absorbed, if not, it is scattered. This Bernoulli experiment is followed by another one, in case of absorption. This new 'either/or' scenario has the same pdf of the

[11]The *virtual collision* approach is used and it will be explained in the next chapter

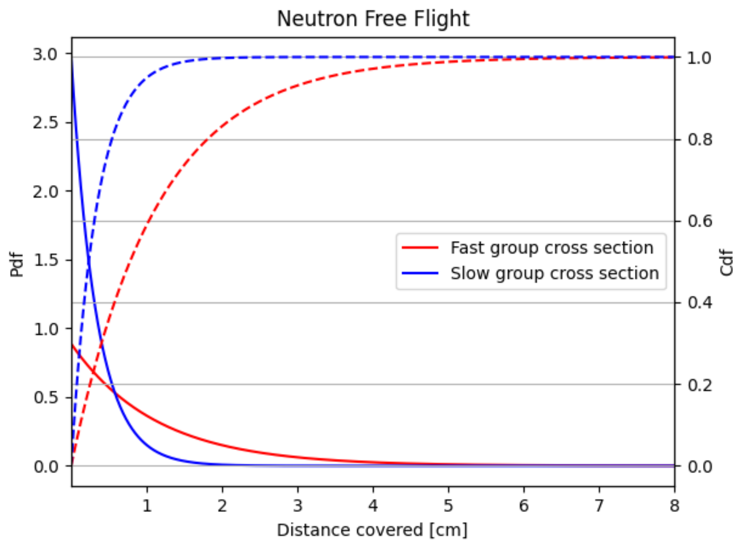


Figure 3.15: $Pdf - Cdf$ couples for different enrgy groups' cross sections (Pdf: solid line; Cdf: dashed line).

previous one but with $p = \frac{\Sigma_f(E)}{\Sigma_a(E)}$ or $p = \frac{\Sigma_c(E)}{\Sigma_a(E)}$ and states whether a fission or a capture occurs. Figs.3.3-3.4 would represent quite well this Bernoulli experiment if the random variable had only two value instead of three.

Fission

As Eq.: suggest, the fission transfer function is a complex operator and the phenomenon which describes is less straightforward than the free flight, for instance. First of all, the number of 'newborn' neutrons must be sampled. As Eq.:3.2.2 says, it depends on the fissile element and, weakly, on the colliding neutron's energy. Given $p(n)$ is the probability of the number of fission neutrons born from fission event, then the FFMC for this discrete random variable n is obtained by satisfying the following inequality [1]:

$$P(n-1) < \eta \leq P(n), n = 0, n_{max} \tag{3.75}$$

where $P(n) = \sum_{n'=0}^{n} p(n')$. In practice, however, rather than using Eq.:3.75, the number of fission neutrons are sampled by using the average number of fission neutrons per fission $\overline{\nu}$ given by [1]:

$$\overline{\nu} = \sum_{n'=0}^{n_{max}} n' p(n') \tag{3.76}$$

Using $\overline{\nu}$, the procedure for sampling the number of fission neutrons consists in the following steps [1]:

1. Generate a random number ρ.
2. If $\rho \leq mant(\overline{\nu})$[12], generate $\overline{\nu} - mant(\overline{\nu}) + 1$ fission neutrons.
3. If the above condition is not respected, then generate $\overline{\nu} - mant(\overline{\nu})$

After sampling the number of neutrons per fission, one must focus on their energy. The procedure starts from the fission spectrum defined in Chapter 1. Commonly, the fission spectrum is given by the *Watt spectrum* that for ^{235}U for thermal fission ('slow' energy group) is given by [1]:

$$\chi(\hat{E}) = 0.4527 e^{\frac{\hat{E}}{0.965}} \sinh \sqrt{2.29\hat{E}} \tag{3.77}$$

with $\hat{E} = \frac{E}{E_0}$ and $E_0 = 1$ MeV. Fig.:3.16 allows to visualize this function.

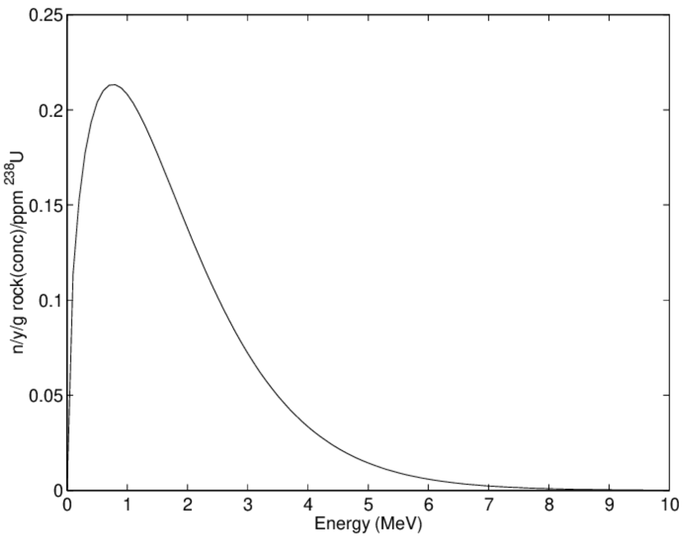


Figure 3.16: An example of Watt spectrum [11].

[12]$mant()$ is the mantissa function

For such a function, the rejection technique is preferable with respect to the analytical inversion (Fig.3.9 at pag.38), but in this thesis the energy spectrum is discretized and the approach of condition 3.20 will be used again.

Scattering

This subsection aims at introducing one last approximation for the model studied in this thesis: the discretization of the energy spectrum together with the already mentioned concept of random walk. A random walk is a path in a certain space that is not predictable a-priori. In the phase space of interest, the transitions among spatial coordinates constitute a continuous random walk, just like the transitions among flight directions, as the next sub-section will show. Transitions among the N energy levels are not only discrete, but also *markovian.* A markovian process is a phenomenon whose evolution depends only on its current condition, not on its previous 'history' [12]. First of all, a phase space, consisting in N possible states, has to be stated. A particle random walk in such phase space is characterized by a first collision in some state i_1, subsequent transmission through a sequence of states i_2, i_3... and finally, a termination in some state i_k. A *discrete random walk process* will therefore be completely specified by [12]:

- a set of first collision probabilities p_i^1 (probability of first collision in state i);
- a set of transmission ones $p_{i,j}$ (probability of transition from state j to state i);
- a set of 'death' probabilities p_i .

With $\alpha = i_1,\ldots,i_k$ denoting a typical random walk and k being the number of collisions made to termination, then [12]:

$$\begin{aligned} p_i^1 &= P[i_1 = i], \\ p_{i,j} &= P[i_{n+1} = i | i_n = j, k > n], \\ p_i &= P[k = n | i_n = i]. \end{aligned} \tag{3.78}$$

It is required that $p_i > 0$ and $p_{i,j} \geq 0$,

$$\begin{gathered}\sum_{i=1}^{N} p_i^1 = 1, \\ \sum_{i=1}^{N} 1 - p_{i,j} \leq 1\end{gathered} \tag{3.79}$$

for all j. The latter condition is valid if multiplication, i.e. fission, is excluded. It can be supposed that α has made collisions at i_1,...i_k. If P_j^n denotes the probability of making collision n at state j [12]:

$$P_j^n = P[i_n = j | k > n-1], n \geq 2 \tag{3.80}$$

and:

$$P_j^1 = P[i_1 = j] = p_j^1 \tag{3.81}$$

It is clear that a recursion formula for P_j^n is:

$$P_j^n = \sum_{k=1}^{N} p_{j,k} P_k^{n-1}, n \geq 2 \tag{3.82}$$

Eq.:3.82 expresses the fact that the probability that collision n takes place at j is the probability that the $(n-1)$-th collision takes place at k, times the probability of transition from k to j, summed over all intermediate states k [12].
For each state j a random variable X_j may be defined such that X_j is the number of collisions made at j. Then X_j is a discrete random variable on the space of all random walks α, which may take on any positive integer value. It is not hard to see that the expected value of X_j may be calculated by [12]:

$$\begin{aligned} E[X_j] = 1 \cdot P_j^1 + 1 \cdot P_j^2 + 1 \cdot P_j^3 + \\ 1 \cdot P_j^n .... = \sum_{n=1}^{\infty} P_j^n \end{aligned} \tag{3.83}$$

Let $E[X_j]$ be the discrete collision density at j with $E[X_j] = P_j$ [12].
To compute P_j, using Eq.:3.82:

$$\sum_{n=2}^{\infty} P_j^n = \sum_{n=2}^{\infty} \sum_{k=1}^{N} p_{j,k} P_k^{n-1} \tag{3.84}$$

i.e. $\sum_{k=1}^{N} p_{j,k} \sum_{n=2}^{\infty} P_k^{n-1}$ and, after a little modification, $\sum_{k=1}^{N} p_{j,k} \sum_{n=1}^{\infty} P_k^{n}$. Therefore:

$$P_j = P_j^1 + \sum_{k=1}^{N} p_{j,k} P_k, 1 \leq j \leq N \tag{3.85}$$

The vector with values $P_1,\ldots,P_n$ might be denoted with $\vec{P}$, the array with values P_1^1,...,P_N^1 with $\vec{P^1}$ and the matrix with entries $p_{i,j}$ by $\hat{K}$, then the system of Eqs.3.85 may be re-written as a matrix equation [12]:

$$\vec{P} = \vec{P^1} + \hat{K}\vec{P} \tag{3.86}$$

The matrix equation just derived relates the discrete collision density $\vec{P}$ to a density of first collisions $\vec{P^1}$ and a operator $\hat{K}$ describing the probability of direct transition from one state (or energy group) to another. The solution of Eq.:3.86 is a vector whose component may be referred to as the densities of particles about to undergo collision in each state. Hence, the source term, $\vec{P^1}$, is interpreted as the density of particles about to undergo a first collision, particles which have already been transferred from their birth state to the state at which first collision will be made [12]. As a realistic model of the above process, one may use the steady-state multi-group neutron transport equations, for an infinite, mono-dimensional, homogeneous medium, without multiplication:

$$\begin{aligned} \mu\frac{\partial \phi^i(z,\mu)}{\partial z} + \Sigma_t \phi^i(z,\mu) \\ = S^i(z,\mu) + \sum_{j=1}^{G} \int_{-1}^{1} d\mu' \Sigma_t^j(z) \phi^j(z,\mu') f_s^{ij}(z,\mu'\mu) \end{aligned} \tag{3.87}$$

where i is the energy index ($1 \leq i \leq G$). ϕ^i is the vector flux in the i-th group. The scattering transfer function, after discretization, still keeps its original, clear task to operate transition from the generic j-th group to the 'correct', i-th one. In an infinite, homogeneous medium, with a constant source, ϕ^i is constant, so that $\mu\frac{\partial \phi^i(z,\mu)}{\partial z}$ is null for every energy group [12]. If, further, one assumes that the source is isotropic, $S^i(\mu) = \frac{S^i}{2}$, Eq.:3.87 becomes:

$$\Sigma_t \phi^i(\mu) = \frac{S^i}{2} + \sum_{j=1}^{G} \int_{-1}^{1} d\mu' \Sigma_t^j \phi^j(\mu') f_s^{ij}(\mu'\mu) \tag{3.88}$$

with $1 \leq i \leq G$.

Now, it is better to consider the total flux (integration over all directions): now it is a scalar: $\int_{-1}^{1} d\mu' \phi^j(\mu') = \Phi^i$. It results [12]:

$$\Sigma_t^i \Phi^i = \sum_{j=1}^{G} \Sigma_t^j f_s^{ij} \Phi^j + S^i, 1 \leq i \leq G \tag{3.89}$$

It is the infinite medium equations for the scalar flux [12]. Now, it is important to focus on f_s^{ij}:

$$f_s^{i,j} = \int_{-1}^{1} d\mu' f_s^{ij}(\mu'\mu) = \frac{\Sigma_s^{ij}}{\Sigma_t^j} \tag{3.90}$$

The matrix Σ_s^{ij} is the transition matrix from energy group j to energy group i:

$$\sum_{i=1}^{G} \Sigma_s^{ij} = \Sigma_s^j \leq \Sigma_t^j \tag{3.91}$$

where Σ_s^j is the macroscopic scattering cross section in group j and where Σ_t^j is the total macroscopic cross section. Now, let $\vartheta^i = \Sigma_t^i \Phi^i$ be the definition of macroscopic scalar collision density in group i. Then Eq.:3.89 becomes [12]:

$$\vartheta^i = \sum_{j=1}^{G} \frac{\Sigma_s^{ij}}{\Sigma_t^j} \vartheta^j + S^i, 1 \leq i \leq G \tag{3.92}$$

under the assumption that $\Sigma_t^j \neq 0$ for all j. Thus, an equation in matrix form arises again: $\vec{\vartheta} = \hat{K}\vec{\vartheta} + \vec{S}$ [12]. On physical grounds, $Q \geq 0$, $K \geq 0$, and $\vartheta \geq 0$. Now it is defined [12]:

$$p_{ij} = \frac{\Sigma_s^{ij}}{\Sigma_t^j} \tag{3.93}$$

so that $0 \leq p_{ij} \leq 1$ and:

$$\sum_{i=1}^{G} p_{ij} = \frac{1}{\Sigma_t^j} \sum_{i=1}^{G} \Sigma_s^{ij} = \frac{\Sigma_s^j}{\Sigma_t^j}. \tag{3.94}$$

Let $p_j = 1 - \frac{\Sigma_s^j}{\Sigma_t^j}$. Further, it is defined the first collision probability: $p_i^1 = \frac{S_i}{\sum_{i=1}^G S_i}$ so that $0 \leq p_i^1 \leq 1$ and, obviously, $\sum_{i=1}^G p_i^1 = 1$. With these definitions, requirements 3.79 for a discrete random walk have been satisfied [12]. Then dividing Eq.:3.86 by the normalizing factor $\frac{1}{\sum_{i=1}^G S_i}$:

$$\frac{1}{\sum_{i=1}^G S_i}\vartheta = \hat{K}\frac{1}{\sum_{i=1}^G S_i}\vartheta + \frac{1}{\sum_{i=1}^G S_i}\vec{S} \tag{3.95}$$

With $\vec{P} = \frac{1}{\sum_{i=1}^G S_i}\vartheta$ and $\vec{P^1} = \frac{1}{\sum_{i=1}^G S_i}\vec{S}$, the Eq.:3.86 is obtained again: this is the identification of the multi-group transport process in an infinite medium as a discrete random walk process. In particular, $\vec{P}$ is the normalized vector collision density whose j-th component, P_j, represents the expected number of collisions in group j per unit source particle [12]. It is worth noting that the source vector $\vec{P^1}$ may be identified either as the density of births or the density of first collisions because of the absence of spatial dependence. [12]. In some sense, the source term in Eq.3.86, in the case of κ_0 computation algorithm for instance, may be seen as the fission spectrum. In this thesis, the matrix $\hat{K}$ has order 2×2 and the denominators of the matrix elements are the scattering macroscopic cross sections of group i (i.e. the sum of all the scattering cross sections from group i to all the others) because, for the code (presented in Chapter 4) the scattering transition is a random experiment that is downstream with respect to the dilemma 'absorption-or-scattering'. The matrix of Eq.:3.86, instead, treats the absorption as the complementary of the scattering, but 'on an equal footing'. Another important difference between $\hat{K}$ and the one used in the codes is that the latter is the transpose of the former. It is worth noting that matrix $\hat{K}$ is usually upper (or lower) triangular. The scattering from a low energy group to a high one, indeed, should be impossible: collisions reduce neutrons' speed, they do not accelerate them. In some case, as Chapter 4 will show, this event is not impossible, but simply highly unlikely. The so-called Scattering Matrix presents as many probability density functions as rows. It is easy to get the corresponding cumulative distribution functions: when a transition occurs, the energy level before the scattering event states which row has to be selected. The new energy level is given by applying the condition of 3.20.
Eq.:3.86 may be re-arranged as a classical linear system with the canonical form $\hat{A}\vec{x} = \vec{b}$:

$$\vec{P}(\hat{I} - \hat{K}) = \vec{P^1} \tag{3.96}$$

where $\hat{I}$ is the identity matrix. Naturally, Eq.:3.96 is solvable via Monte Carlo simulation, but there is a problem with respect to the deterministic version just shown: the sum in the discrete collision density definition must be truncated at $n = N$ for the sake of the random variable sampling that cannot go to ∞. The convergence of the Monte Carlo solution to the analytical one, after a certain number of iterations, is however demonstrable.

Isotropic emission

Since both scattering and fission emissions are isotropic phenomena (the former by hypothesis as previously shown in Chapter 1, the latter by its very nature) the change of direction affects them in the same way, and thus, is the last random experiment to deal with. Since the model of this thesis has a plane geometry (Chapter 1) a pdf with only one variable is needed. It samples only the cosine of the direction. It has, as domain, the range [-1,1] over which has a constant value given by the normalization, i.e. $\frac{1}{2}$. The sampling exploits the analytical inversion (3.22).

Chapter 4

Implementation of the codes

After the necessary, theoretical part, the moment to turn it into practice has come. The chosen language is Python, for its great spreading. The task to be faced, although some simplifications have been introduced in previous chapters, cannot be run by a simple code, but a code system with a certain order of complexity is needed. The section 4.1.1. will be useful to briefly introduce Python. The section 4.1.2. will illustrate how the criticality problems for computing κ_0 and γ_0 are globally organized, while section 4.1.3. explains how the programs operate. The actual codes with relative comments will be presented in the final Appendices A, B and C. The chapter goes on with section 4.2 concerning the main issues of the implementation and the strategies which have been adopted to try to solve them.

4.1 The hierarchy of the codes

4.1.1 Python

"Python is an interpreted high-level, general-purpose programming language, very easy to be read thanks to the use of indentation. Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical codes for small and large-scale projects" [8]. *Object-Oriented Programming* (OOP) instead, is a programming paradigm based on the concept of *objects*, which may contain data and code: data in the form of fields (often referred to as *attributes* or *properties*) and code in the form of procedures (often known as *methods*). A feature of objects is that an object's own pro-

cedures can access and often modify the data fields of itself (objects have a notion *self* in Python). In OOP, computer programs are designed by making them out of objects that interact with one another [8]. OOP gathers in a localized area of the source code (*class*) the statement of the data structures and the procedures operating on them. Classes consist in abstract models which at run-time are called to instantiate or create software objects that are associated with the invoked class. The latter have attributes (variables and/or constants defining the features of the objects that may be created invoking the class) and methods (functions operating on the attributes) according to their respective classes' statements. The part of a program which uses an object is called *client* [13].
A programming language is object-oriented if it allows to implement three schemes with the syntax of the language:

- *Encapsulation*: the separation between class interface and class implementation; in this way, the clients of an object can use the former, but not the latter.
- *Inheritance*: it allows to create classes starting from those which have already been defined.
- *Polymorphism*: the fact that the same executable code might be used with instances of different classes with a *super-class* in common.

Among the advantages of this kind of programming, one may mention [13]:

- OOP provides a natural support to software modeling of real object or of the abstract model to be reproduced.
- It allows an easier management and maintenance for large scale projects.
- The class-ordered organization supports modularity and code reuse.

On the other hand, some mechanisms, that are inherent in the object management, cause overhead from the points of view of time and memory and may induce efficiency problems, as it is mentioned in Section 4.2.1 [13].

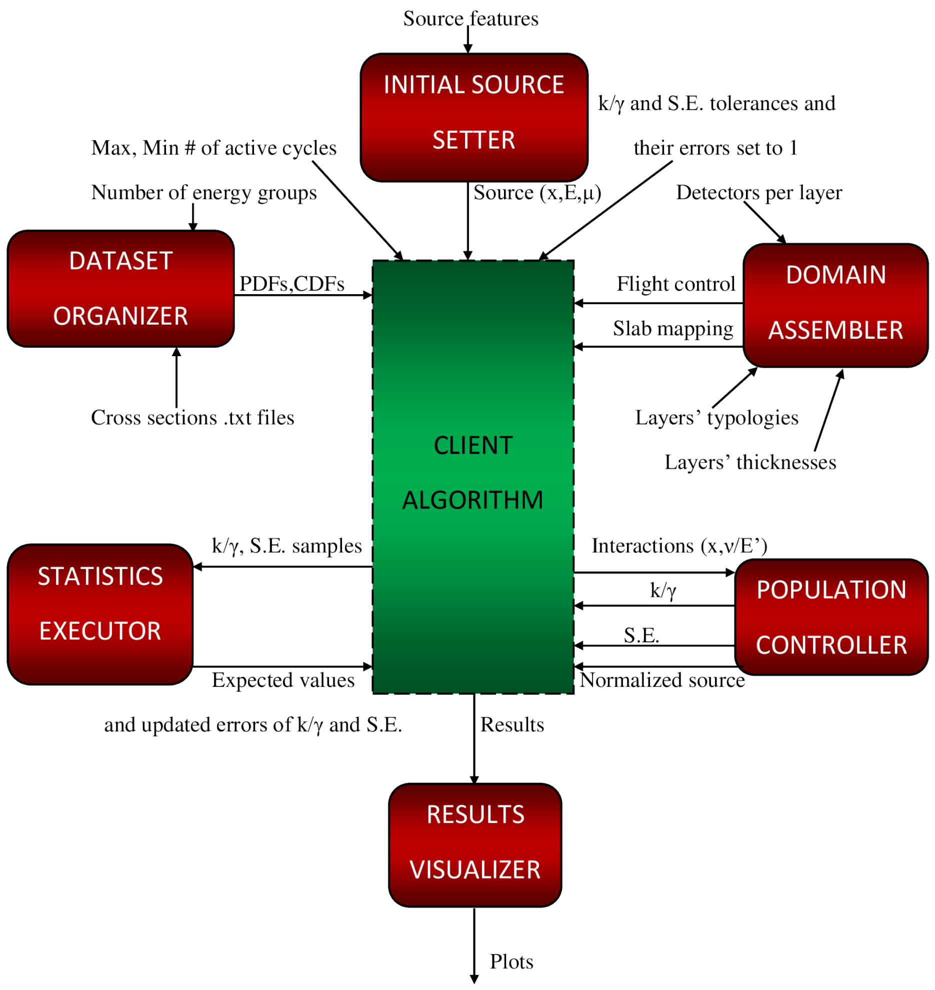


Figure 4.1: General scheme of the codes

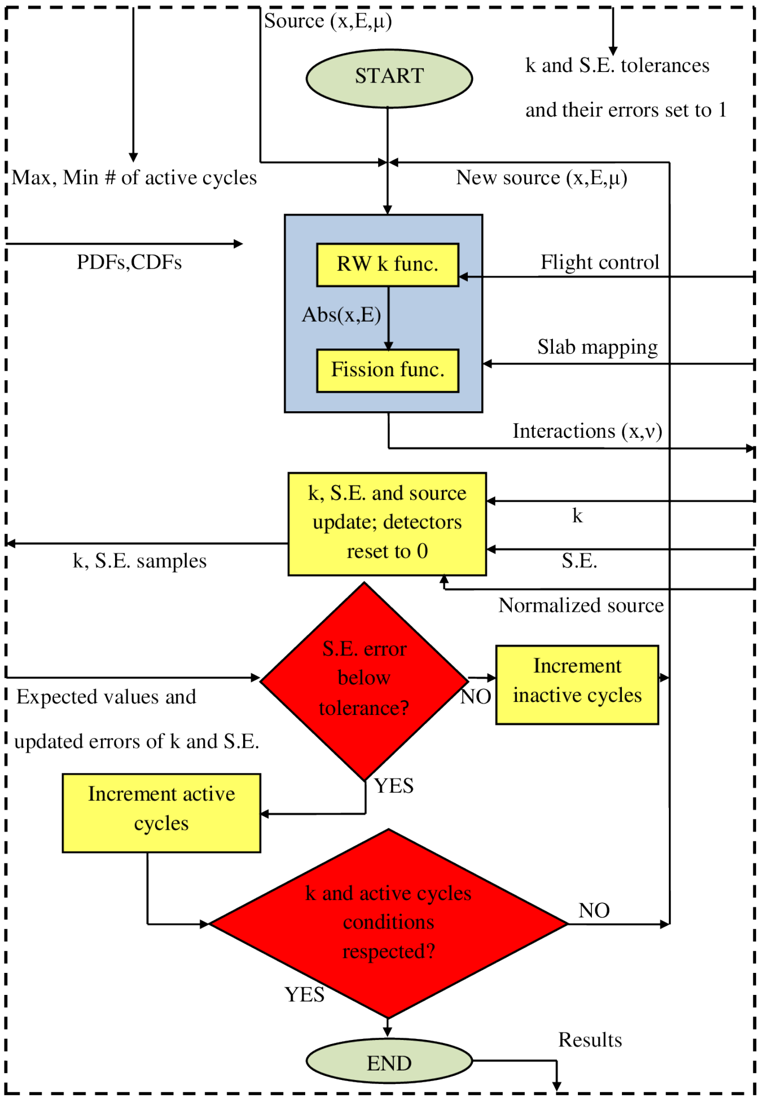


Figure 4.2: Client algorithm for κ_0 computation

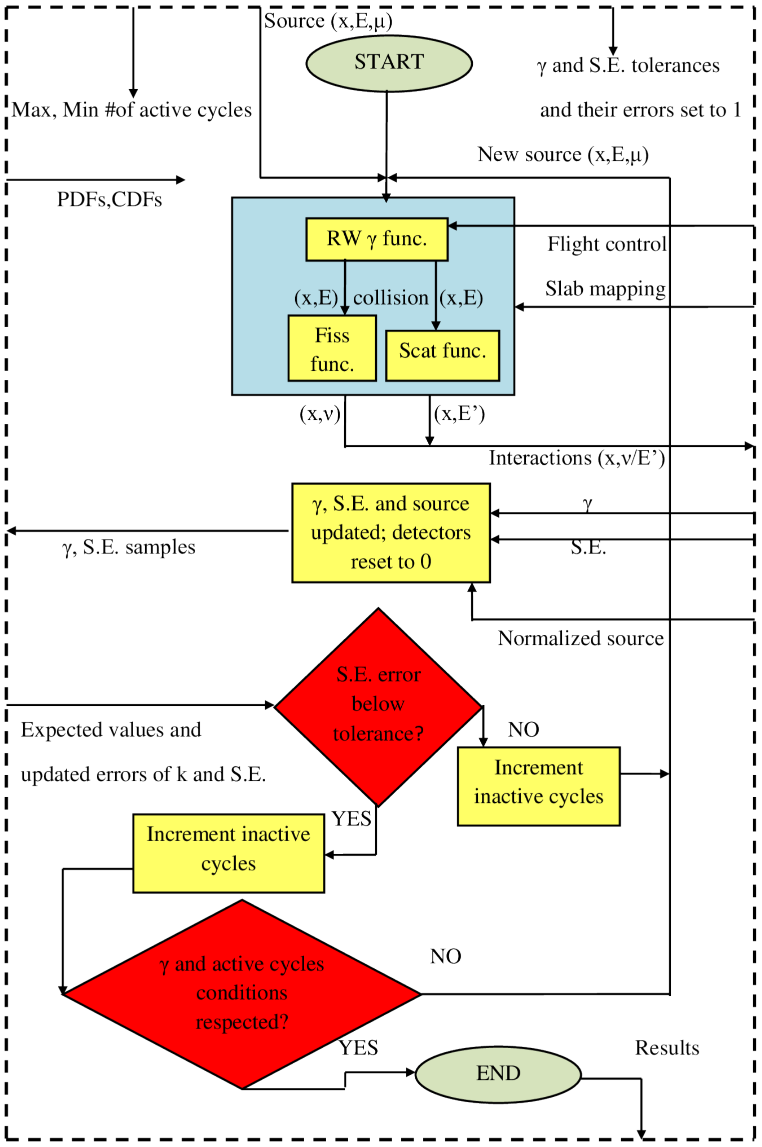


Figure 4.3: Client algorithm for γ_0 computation

4.1.2 Algorithms' working environment

After giving the basic information about the programming language, it is time to concentrate on the working environment itself, starting from an overview. As Fig.4.1 at pag.66 shows, many classes (red background) serve the algorithm computing the eigenvalue (green background). Some external inputs are directed to the classes, others to the client itself. If adequately invoked through the so-called *dot-notation*, all the attributes and methods of a class are accessible both in the algorithm and in the other classes. For the sake of readability, the inner structures of the clients are introduced in Fig.4.2 and Fig.4.3 at pag.67-68, the former for the κ_0-eigenvalue algorithm, the latter for the γ_0-eigenvalue one. Thus, the incoming and outgoing entities in Figs.4.1-4.2-4.3 fit together. In section 4.1.3. a more detailed inspection of each component in these figures will be presented.

These schemes guide the reader through the fundamental way to compute the eigenvalues. In Chapter 5 some modifications take place in order to repeat the same operations by changing one certain parameter at a time, but the essential structure will remain the same.

4.1.3 Code structures

This section is devoted to the description of all the classes, the functions and the algorithms seen before. At the begin of all the scripts the developer must call all the libraries and classes that will be useful for the code. The libraries which appear with most frequency are:

- *Numpy*: it offers comprehensive mathematical functions, random number generators, linear algebra routines , etc.
- *Pathlib*: it set to add additional directories where python will look for modules and packages.
- *Os*: it provides functions for interacting with the operating system.
- *Math*: it gives access to some common mathematical functions and constants.
- *Matplotlib*: it creates static, animated, and interactive visualizations.

Dataset Organizer

The inputs for this class are the number of the energy groups and the name of the file, with '.txt' format, that will be opened and read to create a dictionary with all the useful parameters of the materials forming the slab. In Python, a dictionary is a mutable, not ordered type that contains items formed by a key and a value. Once created, a set of couples 'key-value' is obtained and if one wants to get a value, a unique, predefined key must be used to invoke it. The following tables contain the material data used in eigenvalues' computation:

Σ_{a1}	Σ_{f1}	χ_1	ν_1	$\Sigma_{1\to1}$	$\Sigma_{1\to2}$	Σ_{t1}
0.0984	0.0936	0.5750	3.1000	0.0792	0.0432	0.2208
Σ_{a2}	Σ_{f2}	χ_2	ν_2	$\Sigma_{2\to1}$	$\Sigma_{2\to2}$	Σ_{t2}
0.09984	0.08544	0.42500	2.93000	0.00000	0.23616	0.33600

Table 4.1: 'Fuel' dataset for homogeneous critical slab with 2 energy groups [4].

Σ_{a1}	Σ_{f1}	χ_1	ν_1	$\Sigma_{1\to1}$	$\Sigma_{1\to2}$	Σ_{t1}
0.001940	0.000836	1.000000	2.500000	0.838920	0.046350	0.887210
Σ_{a2}	Σ_{f2}	χ_2	ν_2	$\Sigma_{2\to1}$	$\Sigma_{2\to2}$	Σ_{t2}
0.053633	0.029564	0.000000	2.500000	0.000767	2.918300	2.972700

Table 4.2: 'Fuel' dataset for heterogeneous slab with 2 energy groups [10].

Cross sections are expressed in $[cm^{-1}]$, while other quantities are dimensionless. In Table4.4 the last column is dedicated to the diffusion coefficient, expressed in [cm]. Total cross sections are obtained by summing the absorption and scattering cross sections. The latter, on their turn, are the sum of the transition cross sections from a generic energy group to all the others. Therefore, the so-called Dataset Organizer, after reading the files and creating the dictionaries, is able to make operations on these data to create the necessary probability density functions and cumulative distribution functions to accomplish the samplings during the computation.

Σ_{a1}	Σ_{f1}	χ_1	ν_1	$\Sigma_{1\to1}$	$\Sigma_{1\to2}$	Σ_{t1}
0.00074	0.00000	0.00000	0.00000	0.83975	0.04749	0.88798
Σ_{a2}	Σ_{f2}	χ_2	ν_2	$\Sigma_{2\to1}$	$\Sigma_{2\to2}$	Σ_{t2}
0.018564	0.000000	0.000000	0.000000	0.000336	2.967600	2.986500

Table 4.3: 'Moderator' dataset for heterogeneous slab with 2 energy groups [10].

Σ_{a1}	Σ_{f1}	χ_1	ν_1	$\Sigma_{1\to1}$	$\Sigma_{1\to2}$	Σ_{t1}	D_1
0.0023	0.000	1.0000	3.100	0.0600	0.0600	0.12230	1.5000
Σ_{a2}	Σ_{f2}	χ_2	ν_2	$\Sigma_{2\to1}$	$\Sigma_{2\to2}$	Σ_{t2}	D_2
0.2000	0.0872	0.0000	2.5000	0.000	1.0000	1.2000	0.4000

Table 4.4: 'Fuel' dataset for homogeneous slab with 2 energy groups [14].

Domain Assembler

The inputs for this class are:

- The array of materials in their spatial order, e.g. ['Fuel', 'Moderator'] for a two-regions slab. Only these two types of layer are considered.
- The array of the corresponding layer thicknesses (in [cm]).
- The number of detectors per layer: they are the source sub-regions and play a decisive role in the determination of the initial source and the normalized one (explained in next sub-sections).

This class sets the coordinates both of internal and of the external boarders and controls the free flight of neutrons with two dedicated methods: the former (the function *'Location'*) receives the x-coordinate as input and gives as output the type of the layer in which the neutron lies; the latter (the function *'Boundarycounter'*) takes as input the couple of coordinates of beginning and ending of a free flight and gives back the sorted coordinates of the boundaries crossed by the neutron. Moreover, the so-called Domain Assembler defines the ensemble of the detectors as matrices with layers as rows [1] and detectors in the single layer as columns.

Initial Source Setter

Stating the features of the starting neutrons is crucial to address properly the convergence of the computations (and in Chapter 5 it will be seen). The inputs needed are:

- The number of starting neutrons (N).
- The distribution that affects both neutrons' positions and energy levels: it can be "Uniform", equally spaced in every layer and mono-energetic neutrons from a chosen group or "Fission-source-like" that, as the name suggests, follows the fission spectrum from the energetic point of view and allocates neutrons only in 'Fuel' regions, randomly uniform in each detector.
- The seed for the pseudo random generation (see Section 4.2.2): the neutrons in the single detector are positioned in an uniformly, random way for "Fission-source-like" distribution.
- The direction of the neutrons: it can follow the proper pdf (direction cosine values "Stochastic") or be directed towards the centre of the slab (direction cosine values equal to 1, "Inward") in case of very thin fuel layer.
- The geometry that considers the whole slab ("Non-symmetric") or only one half of it ("Symmetric") if it is symmetric (this case is examined in Chapter 5 for source convergence studies).

The output consists in three arrays of length N: one for the positions, one for the energy groups and one for the direction cosine values. They are put together to form a $N \times 3$ matrix.

[1] Every layer or only the layers of type 'Fuel', it depends on the needs of the algorithm: for instance the sources in the γ_0 case are not only in the fuel regions, but also in the moderator

Random Walk and associated functions

As Figs.4.2-4.3 suggested, the treatment of the random walk is the first great difference between the algorithms of the two eigenvalues' algorithms. The γ_0-eigenvalue algorithm needs to register not only the absorption sites' coordinates in fuel regions, as κ_0-eigenvalue algorithm does, but also scattering sites' coordinates in all the types of material present in the slab. This fact makes the 'life' of a neutron shorter in the former case, but the inputs are common:

- The three coordinates generated in the Initial Source Setter (or in the Population Controller, the next class to be analysed) organized as a $N \times 3$ matrix.
- The seed for the pseudo random generation (Section 4.2.2) for implementing FFMC.

A critical issue is the transition from a layer to another. A change of medium takes place and is treated like a collision, a *virtual* collision. The neutron's new initial position is the boundary itself, the free flight is sampled with the cross section of the 'new' layer, but energy level and direction cosine do not change. It may be noted that the update of the energy level in case of scattering is an example of application of condition 3.20 (the energy group before the collision 'decides' which cdf to be used for the FFMC and the energy group after the collision is sampled). Finally, the sample of the free flight is an example of analytical inversion of a pdf. For the κ_0-eigenvalue algorithm, the random walk function gives as output A pairs of positions and the energy groups of neutrons absorbed in fuel regions in the form of a $A \times 2$ matrix: the fission function, from this inputs, samples the number of neutrons born per site, as explained in Section 3.2.2.

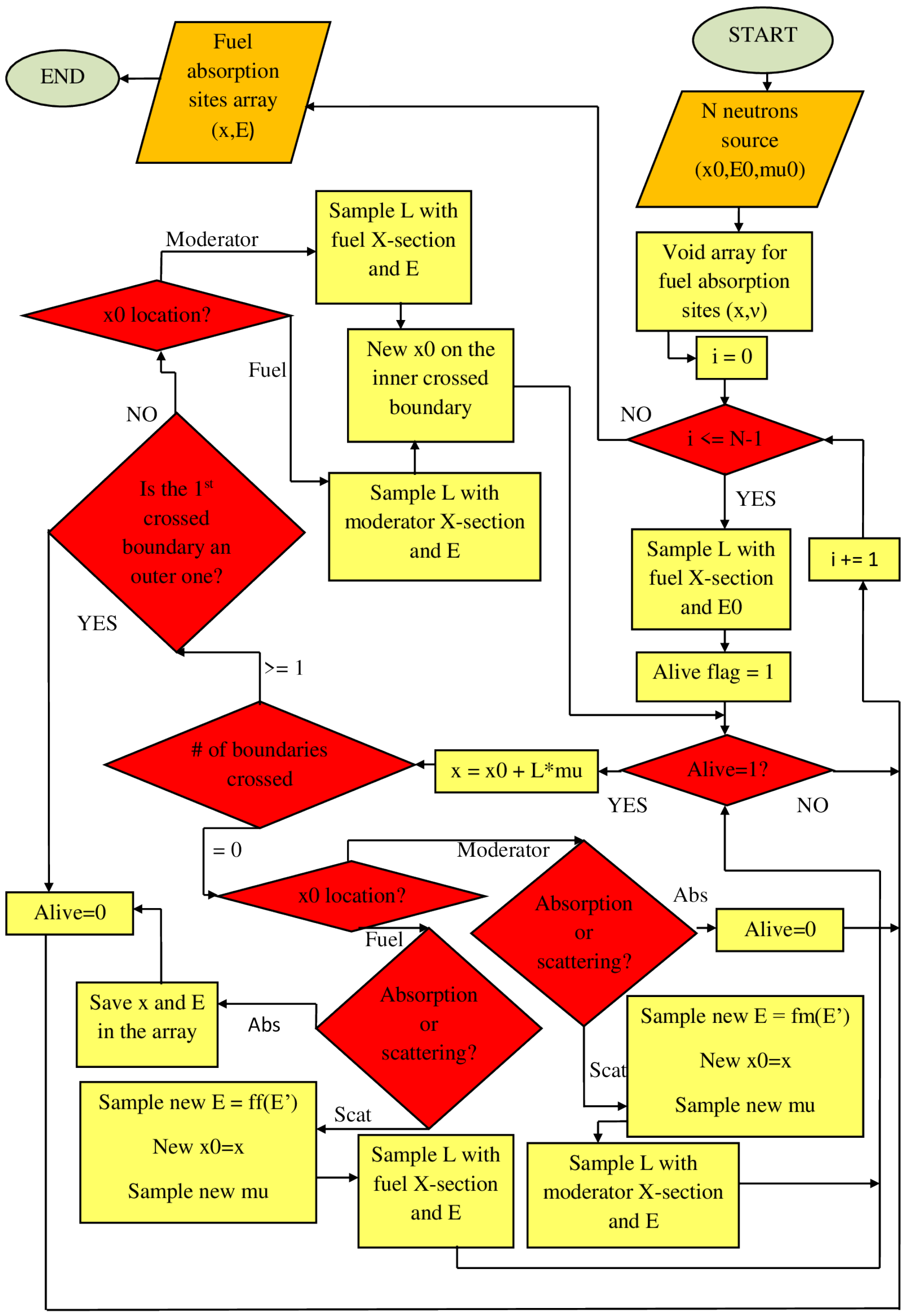

START
N neutrons source (x0,E0,mu0)
Void array for fuel absorption sites (x,v)
i = 0
i <= N-1
NO
YES
Sample L with fuel X-section and E0
i += 1
Alive flag = 1
Alive=1?
YES
NO
x = x0 + L*mu
# of boundaries crossed
>= 1
= 0
Is the 1st crossed boundary an outer one?
NO
YES
x0 location?
Moderator
Fuel
Sample L with fuel X-section and E
New x0 on the inner crossed boundary
Sample L with moderator X-section and E
Alive=0
x0 location?
Moderator
Fuel
Absorption or scattering?
Abs
Scat
Alive=0
Sample new E = fm(E')
New x0=x
Sample new mu
Sample L with moderator X-section and E
Absorption or scattering?
Abs
Scat
Save x and E in the array
Sample new E = ff(E')
New x0=x
Sample new mu
Sample L with fuel X-section and E
Fuel absorption sites array (x,E)
END

Figure 4.4: Flowchart of the random walk for κ_0 computation ('ff' and 'fm' stands for the FFMC of the new energy group respectively in fuel and moderator region).

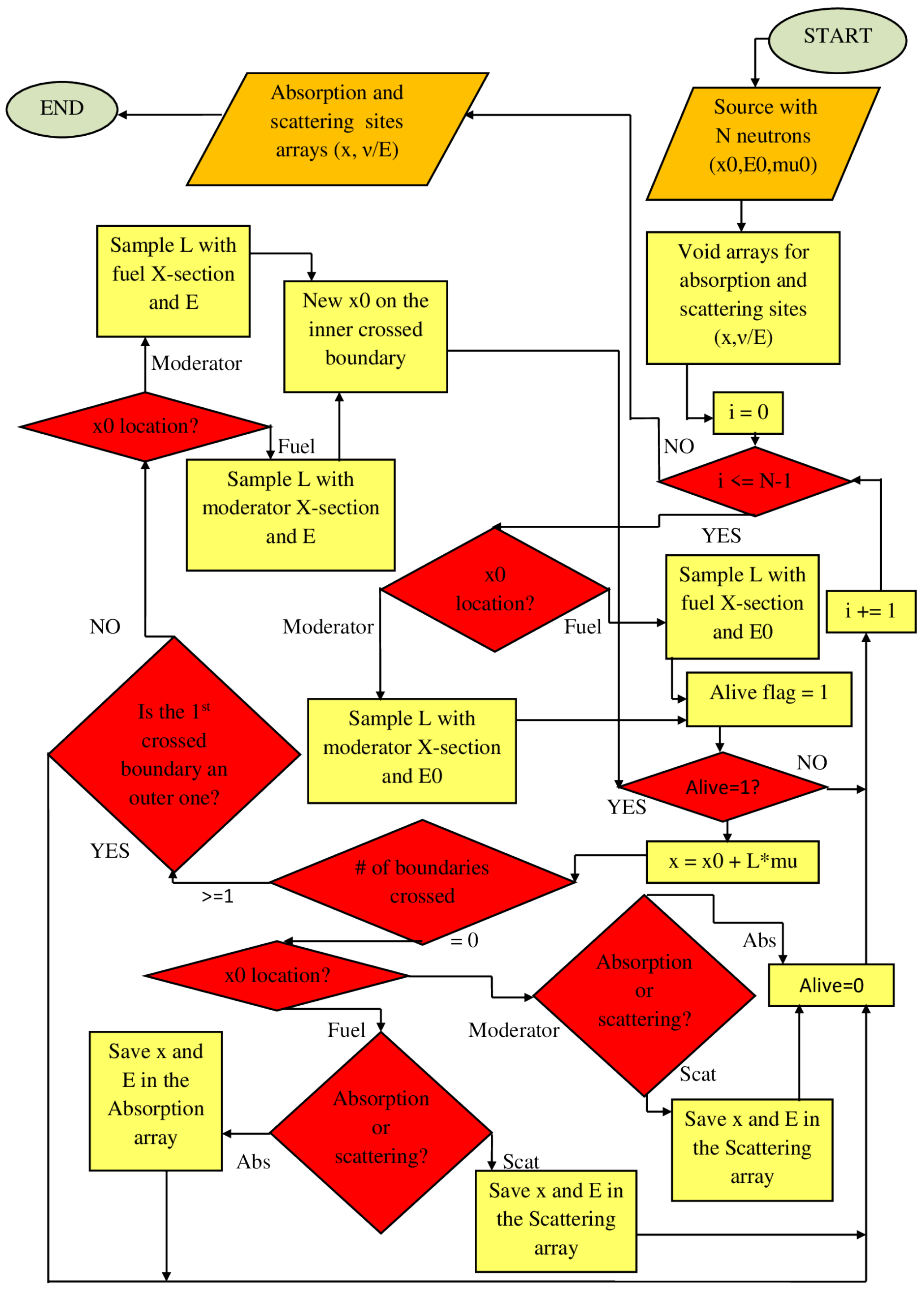

START
Source with N neutrons (x0,E0,mu0)
Void arrays for absorption and scattering sites (x,v/E)
i = 0
i <= N-1
NO
YES
Absorption and scattering sites arrays (x, v/E)
END
i += 1
Sample L with fuel X-section and E0
Alive flag = 1
Alive=1?
NO
YES
x = x0 + L*mu
x0 location?
Moderator
Fuel
Sample L with moderator X-section and E0
# of boundaries crossed
>=1
= 0
Is the 1st crossed boundary an outer one?
NO
YES
x0 location?
Moderator
Fuel
Sample L with fuel X-section and E
Sample L with moderator X-section and E
New x0 on the inner crossed boundary
Absorption or scattering?
Abs
Scat
Alive=0
Save x and E in the Scattering array
x0 location?
Fuel
Moderator
Absorption or scattering?
Abs
Scat
Save x and E in the Absorption array
Save x and E in the Scattering array

Figure 4.5: Flowchart of the random walk for γ_0 computation.

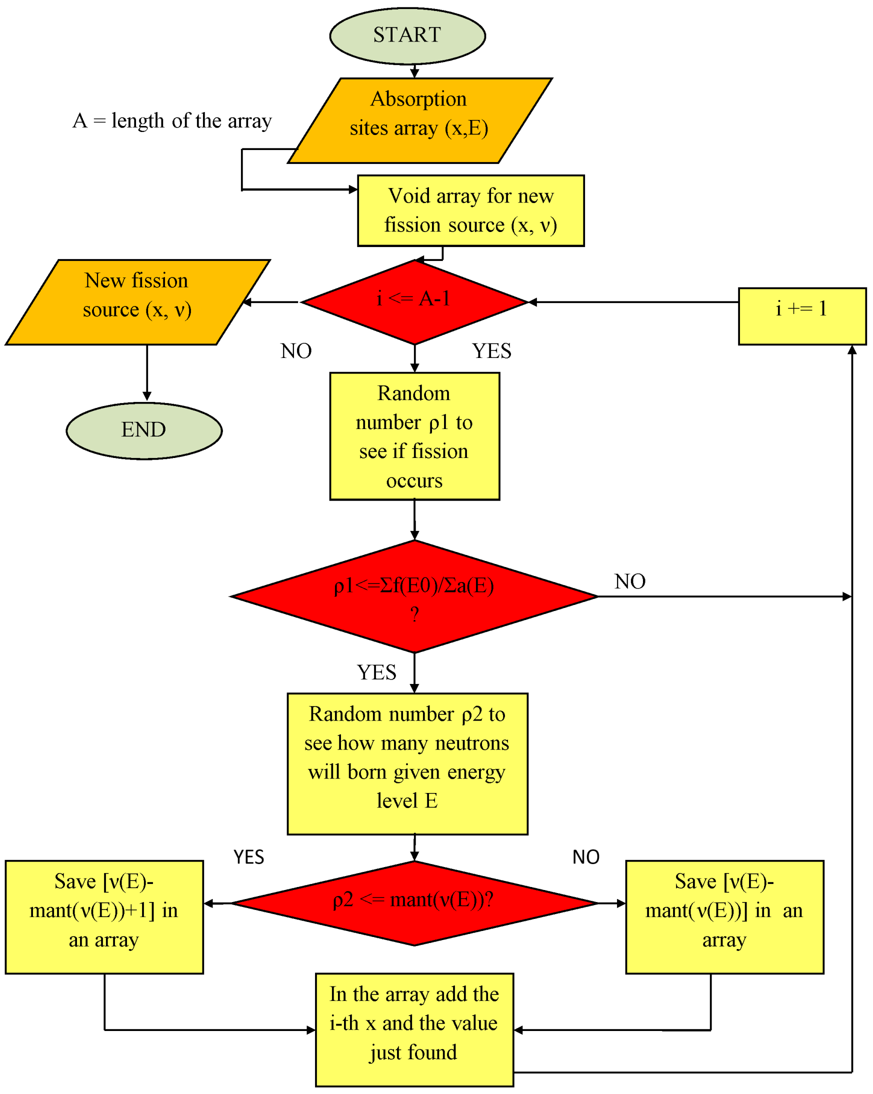

START
Absorption sites array (x,E)
A = length of the array
Void array for new fission source (x, ν)
i <= A-1
NO
YES
New fission source (x, ν)
END
i += 1
Random number ρ1 to see if fission occurs
ρ1<=Σf(E0)/Σa(E) ?
NO
YES
Random number ρ2 to see how many neutrons will born given energy level E
YES
NO
ρ2 <= mant(ν(E))?
Save [ν(E)-mant(ν(E))+1] in an array
Save [ν(E)-mant(ν(E))] in an array
In the array add the i-th x and the value just found

Figure 4.6: Flowchart of the fission function

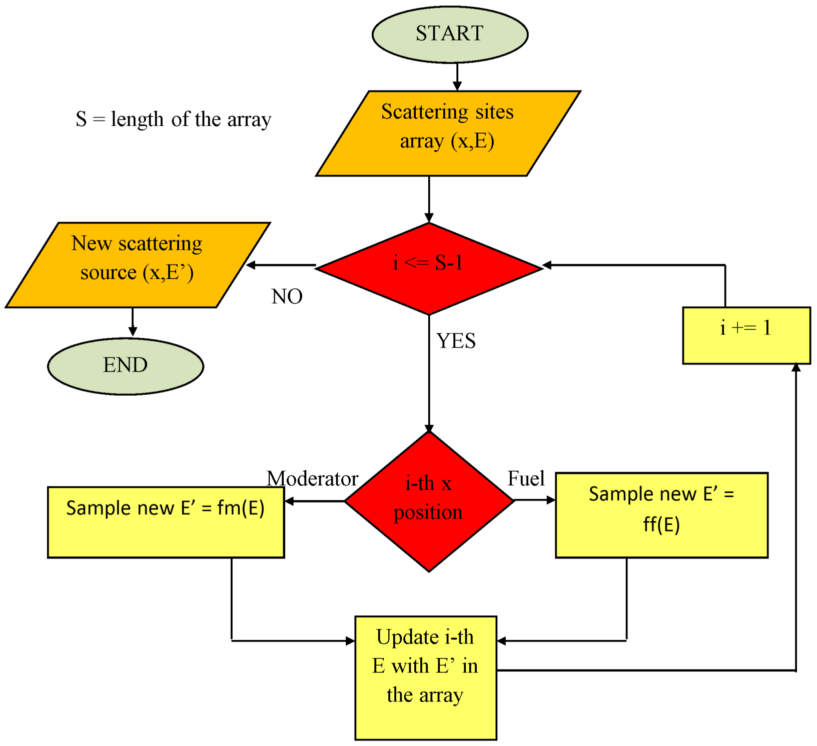


Figure 4.7: Flowchart of the scattering function ('ff' and 'fm' stands for the FFMC of the new energy group respectively in fuel and moderator region).

For the the random walk function of the γ_0-eigenvalue algorithm, gives as output S pairs of positions and energy groups of neutrons absorbed in fuel regions and of neutrons scattered in all the region types in the form of a $S \times 2$ matrix: the fission function acts analogously to the case of the former eigenvalue calculation. The scattering function, after deciding which region the neutron is in, and then it executes the usual FFMC. Figs.4.4-4.5-4.6-4.7 at pages 74-75-76-77 may clarify the previous explanations.

Population Controller

Once collected the positions of interactions and the number of new born neutrons, this two pieces of information serve to know which detector must be filled up with how many neutrons. The Population Controller is the class created for this task by operating on the matrix provided by the Domain Assembler. This task is the one for κ_0 algorithm; for γ_0's one , there are two matrices to be filled up: one for fission sites, the other for the scattering sites. At this point, a major problem arises. It concerns both the supercritical and the subcritical systems. From the formula that gives a generic eigenvalue, a relation between two consecutive generations may be stated [1]:

$$N^{(n)} = K^n N^{(0)} \tag{4.1}$$

as suggested by the power iteration method. This is a quite good estimate of the trend of neutron populations as function of the number of cycles. If $K > 1$, the number of particles increases significantly generation by generation, with two important consequences: also the amount of computer time increases while the source distribution has not converged. On the other hand, if $K < 1$, there is the opposite problem: the systems runs out of particles before converging to a solution. The need to normalize every population to the initial value derives from this issue. Obviously, it is faced after the computation of the eigenvalue, i.e. the ratio with as numerator the amount of 'new' neutrons for the next cycle and as denominator the amount of neutrons at the beginning of the previous cycle (both passed as inputs). The number of neutrons must be subsequently normalized to the initial value, introduced by the Initial Source setter. This process is very different for the two eigenvalues.
For the κ_0 algorithm, things are simpler: each detector-source is multiplied

times a normalization factor, i.e. the ratio between the population of the initial source (passed to this class as input) and the population of the new one. Each normalized source is then rounded up (the number of neutrons must be an integer). In the single detector, the new positions of the neutrons are chosen randomly, with a uniform distribution (the seed appears again among the inputs). Once known the amount of the normalized, new source and each new position, the process may go on with the sampling of the new energy groups through the fission spectrum and of the new energy directions: the source for the next cycle is ready.
The normalization in the γ_0 algorithm is more complicated. The issue is the energy groups of scattered neutrons inside each detector. The proportions of the energy groups have to be preserved, because the meaningfulness of the phenomenon of scattering must be preserved. So, the two matrices of collectors must be summed to execute the normalization as usual for the array of new positions. The number of new neutrons from fission is increased/decreased proportionally and by difference the numbers of new neutrons from scattering is obtained. The energy of the neutrons from fission is sampled via fission spectrum (only in 'fuel' detectors, obviously) and the proportions among energy groups of the neutrons from scattering is kept constant: the normalization causes their round-up and to restore the correct number of scattering neutrons (to have the correct total number of particles per detector) the most represented energy group undergoes a proper subtraction. The process goes on with the normal creation of the other array that constitutes the new source.

Shannon Entropy

The Population Controller class also deals with the evaluation of the source convergence by computing the main indicator of this quantity: the Shannon Entropy (S.E.). From the information theory, the entropy is considered as "a measure of the minimum number of bits for representing a probability density on a computer, or a measure for predicting the outcomes of an experiment" [1]. Another way to see it, is considering it as a measure of how likely the messages emitted from a source are. The lower the entropy is, the more probable an emission becomes. Given an experiment of m possible outcomes with probability p_i for each outcome, the corresponding Shannon Entropy is given by [1]:

$$H = -C \sum_{i=1}^{m} p_i log_2 p_i \tag{4.2}$$

with C an arbitrary constant. This formula might be modified to be useful in the evaluation of the neutron source convergence: C would be replaced by 1 and p_i with the ratio between the normalized population in the i-th detector and the total normalized population (obtained by summing the contributes from all the detectors):

$$H_s^{(n)} = -\sum_{i=1}^{m} q_i log_2 q_i \tag{4.3}$$

One way of using the S.E. is to examine its behaviour from one cycle to next; if the source has converged, then it is expected that S.E. fluctuates about an average value [1]: the power iteration method has achieved its goal and statistics may begin to be computed. The next chapter will present a more detailed description of the S.E. function and its behaviour.

Statistics Executor

After some passages, the initial input from the Initial Source Setter has become the outputs of the Population Controller, consisting in two different random variable arrays (the eigenvalue and the S.E., both of them with as many elements as the number of total cycles performed) and the new source ready for the next iteration. For the S.E., the expectation values may be computed just from the first generation, while the other, the eigenvalue, has to wait for the former to converge. Thus, the Statistics Executor needs as inputs, in addition to the random variable array, the number of active and inactive cycles, because only a part of the array that collects a specific random variable must be used to compute the expectation values. If the random variable is the S.E. the inactive cycles do not exist, all the cycles are active. If the random variable is the eigenvalue, a fraction of the total number of cycles must be skipped. The number of the inactive cycles is updated at every cycle until the error of the S.E.'s sample average is above a certain tolerance (an input of the client algorithm). When this error goes below this given value, the number of active cycles begins to be incremented

at each cycle and the error on the eigenvalue's sample average is computed. If this error goes below its own tolerance (another client algorithm input) the iterations do not stop as long as a minimum number of active cycles is not reached. After a maximum number of active cycles, iterations stop regardless of the value reached by the error.
The attributes of the Statistics Executor are:

- The sample average.
- The second order moment of the sample.
- The variance of the sample.
- The relative standard deviation of the sample average.

When the sample has only one element, the variance is, by construction, null, but this fact means, obviously, neither that the event is sure, nor that the condition on the tolerance is respected. The error is updated to the value of the relative standard deviation only if the variance is greater than zero and this possible bias is prevented. Moreover, in order to avoid possible, precocious counting of active cycles, due to momentary oscillations below the tolerance from the error on the sample mean of the S.E., a minimum number of inactive cycles is established (e.g. 10).

Results Visualizer

After every active cycle, the attributes of the Statistics Executor are stored in specific arrays that constitute the inputs for the Results Visualizer that creates with the charts of:

- The random variable (as a function of the total number of cycles).
- The sample average evolution.
- The evolution of the second order moment of the sample.
- The evolution of the variance of the sample.
- The evolution of the relative standard deviation of the sample mean.

- The evolution of the error bar to be applied to the sample average graph.

All the quantities of the previous charts, random variable excluded, are considered, naturally, as functions of the active cycles.

4.2 Issues of the codes

As previous sections have shown, the amount of data to be treated and operations to be run is remarkable: this fact affects the amount of the computational time, but also the domain has a certain influence due to, for instance, the dimension of the slab (trivially the more thicker the slab is, the longer the single random walk will be) or its heterogeneity (if there is a moderator layer, the overall number of scattering events will increase with respect to the absorptions and leakages) or even the type of material (e.g. if two homogeneous slabs have the same thickness, but different cross sections, so different number of absorption events with respect to scattering ones). As previously said, also the type of client algorithm has influence on the duration. Random walks for γ_0-algorithm last less than κ_0-algorithm ones, but on the other hand the normalization for the latter is simpler.
In order to contrast the problem of time needed by the system to operate , the usage of parallel processing is explored.

4.2.1 Code parallelization

Parallel processing refers to "being able to process different data and/or instructions on more than a CPU"[1], a situation achievable in two environments: several computers connected via a network, and a parallel computer that is comprised of several CPUs. Computers are characterized by the type of architecture, classified into four groups considering the number of concurrent instructions and data stream [1]:

- SISD (Single Instruction Single Data): a serial computer that does not have any parallelism in either instruction or data stream.

- SIMD (Single Instruction Multiple Data): a parallel computer that processes multiple data streams through a single instruction.
- MISD (Multiple Instruction Single Data): not been considered.
- MIMD (Multiple Instruction Multiple Data): a computer environment allowing multiple instruction on multiple system.

Thank to its flexibility, MIMD is the base for the majority of parallel computers nowadays. They could furtherly be divided into:

- Shared-memory MIMD: processors share the same memory or a group of memory modules.
- Distributed-memory MIMD: each processor has a local memory and information is exchanged over a network, i.e., message passing.

The performance of a parallel algorithm is measurable by the following factors:

1. *Speed-up*, defined by the ratio of the 'wall-clock' time of the serial processing and the 'wall-clock' time of the parallel one.
2. *Efficiency*, i.e. the ratio between the speed-up and the number of processors used (P), expressed in percentage.

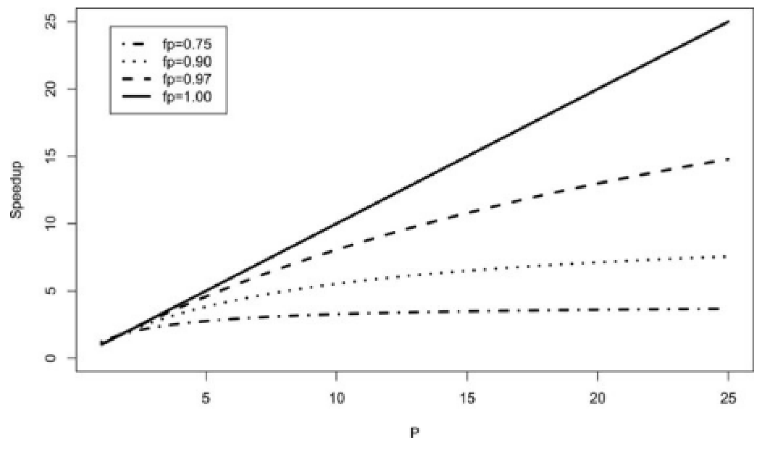


Figure 4.8: Speed-up as a function of the number of processors used (theoretical). [1].

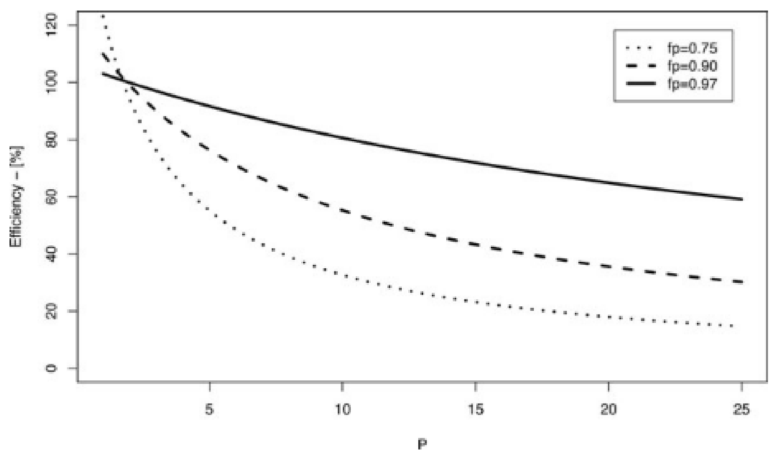


Figure 4.9: Efficiency as a function of the number of processors used (theoretical) [1].

These factors might be compared to the theoretical speed-up predicted by the *Amdahl's law* [1]:

$$S_t = \frac{1}{(1 - f_p) + f_p \frac{1}{P}} \tag{4.4}$$

with f_p is the fraction of the code that is parallelized. It represents a theoretical upper limit (depicted in Figs.4.8-4.9) that might be compared with the performance of real cases. Parallel performance is influenced by some factors [1]: "

- *Load-balancing*: the number of operations on different processors has to be balanced.
- *Granularity*: the number of operations performed per number of communications (distributed memory) or synchronizations (shared memory) is called the grain size (or granularity). The parallel performance deteriorates if the algorithm allocates a low amount of computation to each processor relative to the processor's capacity, while processors require significant number of communications.
- *Message passing*: on distributed memory MIMD or SIMD computers and in distributed computing environments, information is exchanged among processors over a network; this is called message passing. The

number of messages and their size and relation to the network affects the parallel performance.

- *Memory contention*: on shared memory MIMD computers, if different processors access the same memory location, a memory contention would occur.

" In this thesis the parallelization exploits the library multiprocessing, in particular the function pool. The function to be parallelized, if it has other inputs, must be modified by means of the tool *partial* from *functools* library: it fixes some inputs as constants, while the input which can be divided among the processors (in this case the source coordinates) remains as the only actual variable parameter. Then the function *pool* more or less equal fractions of the input among the desired number of processors. The outputs do not appear in the same way as they did in the non-parallel version. Each output is a list of the single outputs performed by each thread. This list must be recomposed before restarting the normal computation.
The parallel processing has been tested on different slabs for both the algorithms to decide whether parallelization of random walks function is a good choice or not: 10 cycles have been performed with 10000 neutrons per cycle for two different slab: an homogeneous one (Table.4.1) with a thickness of 3.5912040 cm, and an heterogeneous one (Tables:4.2-4.3) with a scheme ['Moderator', 'Fuel', 'Moderator'] and respective thicknesses of [5.630757, 9.726784, 5.630757] cm. They are both critical.

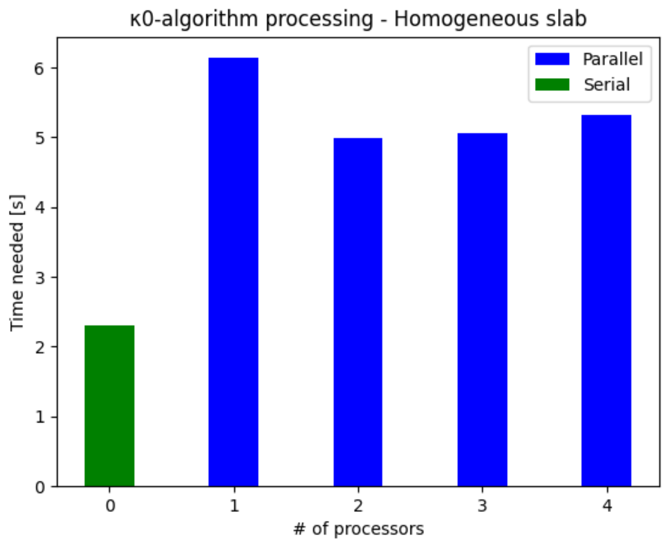


Figure 4.10: Processors performances with homogeneous critical slab for κ_0-algorithm.

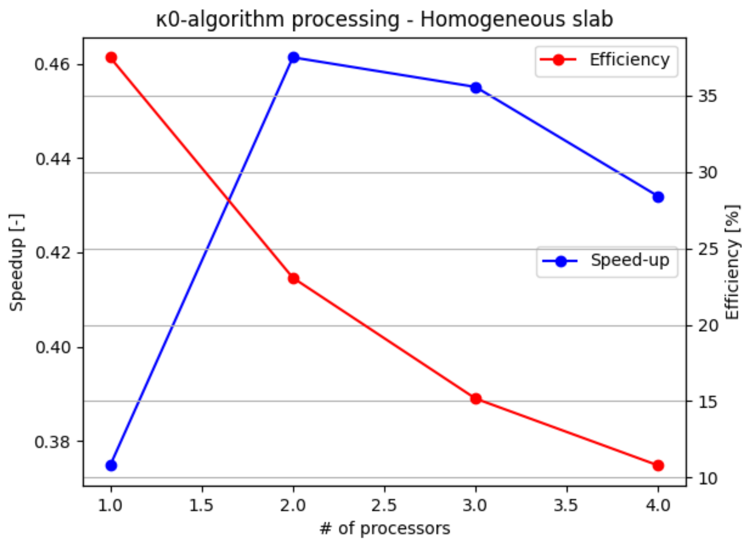


Figure 4.11: Speed-up and Efficiency of parallel processing with homogeneous critical slab for κ_0-algorithm.

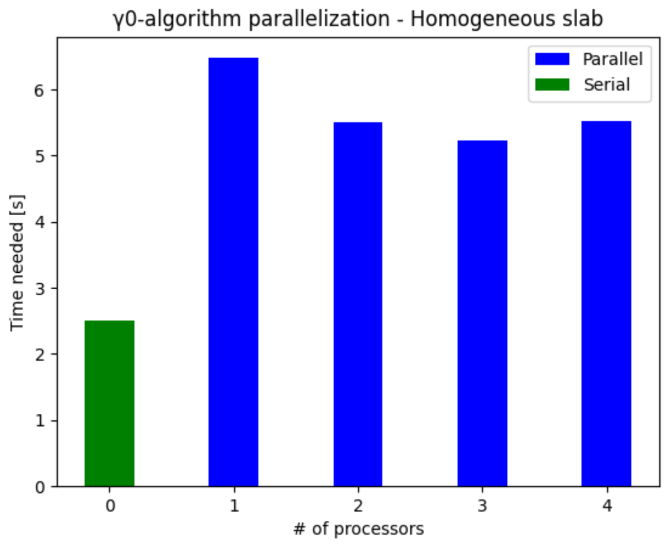


Figure 4.12: Processors performances with homogeneous critical slab for γ_0-algorithm.

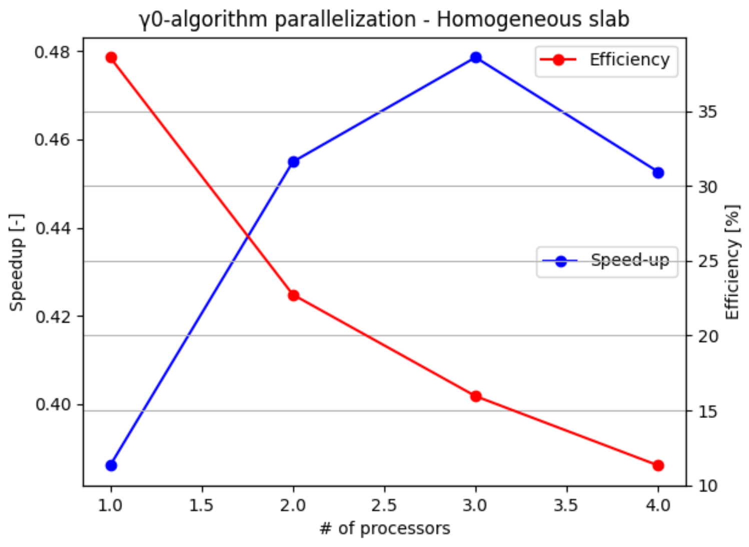


Figure 4.13: Speed-up and Efficiency of parallel processing with homogeneous critical slab for γ_0-algorithm.

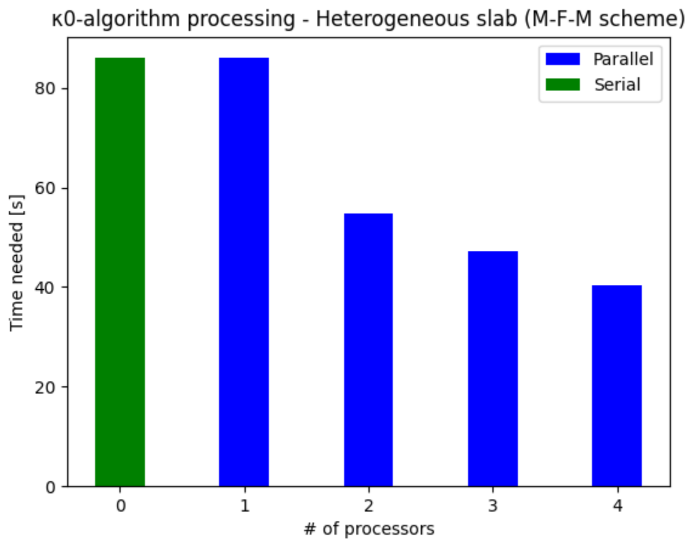


Figure 4.14: Processors performances with heterogeneous critical slab for κ_0-algorithm.

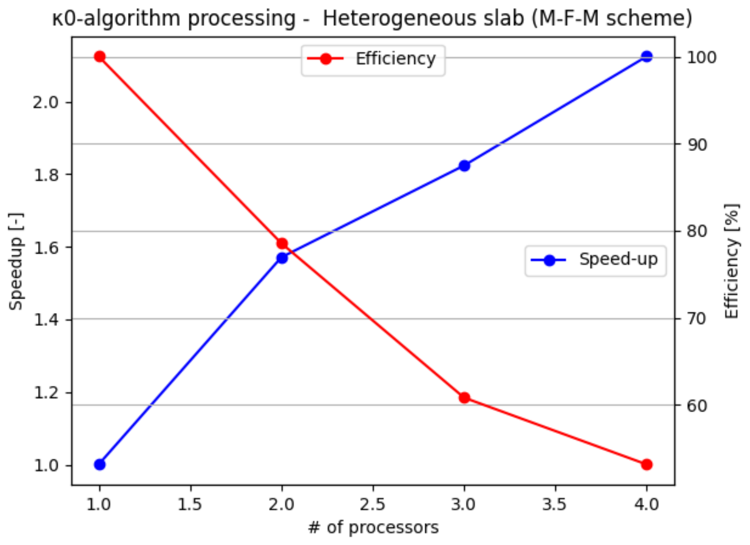


Figure 4.15: Speed-up and Efficiency of parallel processing with heterogeneous critical slab for κ_0-algorithm.

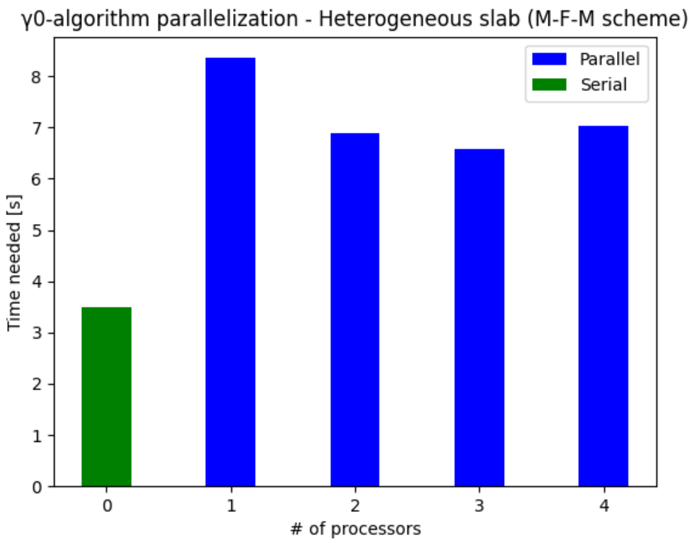


Figure 4.16: Processors performances with heterogeneous critical slab for γ_0-algorithm.

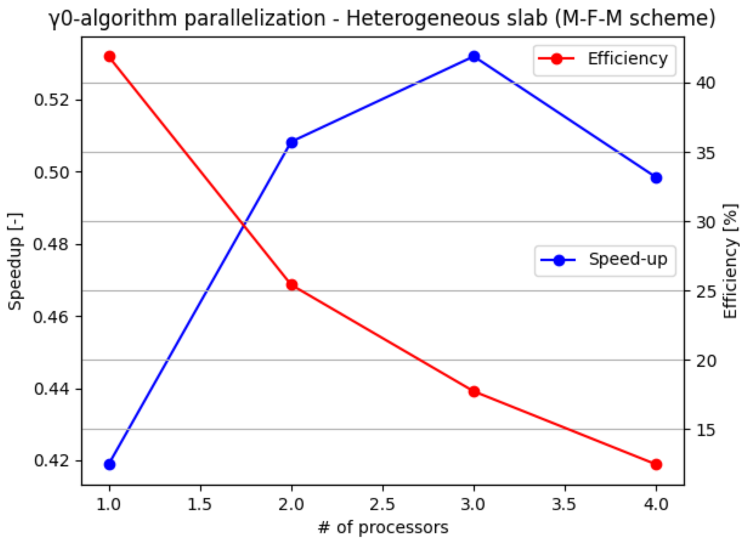


Figure 4.17: Speed-up and Efficiency of parallel processing with heterogeneous critical slab for γ_0-algorithm.

Figs:4.10-4.12 compare the performances for computing respectively κ_0 and γ_0 in the aforementioned homogeneous slab in terms of average time needed to execute a cycle for serial processing (green bar) and with variable number of threads (blue bars). Figs:4.11-4.13 depict the trends of speed-up and efficiency associated to each type of processing. Figs:4.14-4.15-4.16-4.17 have the same purpose, but for the heterogeneous slab. The results of this series of tests clearly demonstrate that this type of parallelizing processes in Python is not always a good solution to reduce the computational time: function calls in Python have significant overhead. Hence, multiprocessing is a trade-off and one cannot know a-priori whether parallelizing is something worth trying to use; because maybe copying the input data into each of the four logical processors, doing the operation in each threads and then copying the output data back is more time consuming than choosing a serial procedure. It depends on the complexity of the processing: sometimes it works quite well, as in heterogeneous computation of κ_0-eigenvalue[2], sometimes it does not.

4.2.2 Seed setting

During the development of the code, checking the results after every little modification is important. The quality of the latter may suggest if the way of proceeding is good or 'going back' is better. The nature of the Monte Carlo simulations might deceive the user: the new result comes from the modifications the user made, or it is just because of the stochastic process? A deterministic problem, since is not dealing with stochastic events, denotes easier check-ups. The solution is that the simulation must be run with the same sequence of random numbers, anytime. The reproducibility of the results has to be guaranteed. It can be done by fixing the seed (Chapter 3). Python's library *Numpy* provides a classical instrument to state the seed the function $np.random.seed(h)$, with h a natural number. In this thesis the algorithm involves nevertheless many imported packages or other scripts of functions or classes, they could reset the global random seed leading to undesirable output changes and the results would not be reproducible anymore. *Numpy* still offers another function to remedy these shortcomings: $np.random.defaultrng(h)$ that must be include among other inputs

[2]In this case the 'heaviness' of the overhead is overcome by the time saving of the multi-processing

for classes or functions involving the use of random numbers. The topic of the seed fixing is also crucial for the parallelization: the seed is obviously passed to random walk function for its own sampling. With a parallelized process with P threads and the seed passed as a constant, every processor would use it: so, instead of N independent random walks, they would be N/P. So, a possible gain in computation time would pay the price for a lower precision. This problem is avoided by passing N different seeds as a new component of the source matrix, and every update of the source must include an update of the seed.

Chapter 5

Results

After presenting the codes and the datasets, the time for examining the results has come. The way of achieving the eigenvalues is not straightforward: it is necessary to understand how neutrons behave in the slab reactor in order to correctly set the parameters denoting the convergence of the power method. Only after this phase, the code verification may take place. With this perspective, section 5.1 is organized in such a way as to give the reader an overview of the neutrons source evolution in simplified slabs and of the diagnostics instruments for studying its convergence (Section 5.1.1). The successive step is to decide which is the right source to optimize the eigenvalue computation (Section 5.1.2) starting from what has been learnt in the previous section. Sections 5.1.3 and 5.1.4 will provide important evaluations about the relationship between eigenvalues and, respectively, the population per cycle and the slab structure. The final part, Section 5.2, will be dedicated to the actual Monte Carlo method validation, i.e. it will be seen if there is consistency between results and statistical theory and if these results are coherent with the ones obtained from literature or through other methods.

5.1 The behavior of neutrons in the domain

First of all, a fact must be emphasized: the set of eigenvalues (κ_0 or γ_0) at each generation is a time series, i.e. a sequence of successive points in time spaced at uniform time intervals. If one wants to do a statistical analysis (and so describe this time series by its mean, variance, etc.) this series has to be stationary, i.e. statistical property do not change with cycles. Without a stationary series, to estimate κ_0 or γ_0 is not achievable [1]. As Chapter 4 has shown, the most widely used and simplest numerical method allowing the neutron population to converge to the fundamental eigenmode is the power iteration. Two key issues are known to affect the neutron population during power iteration: fission source convergence and correlations [15]. Concerning the former, "a slow exploration of the viable phase space by the population implies a poor source convergence." In particular, it has been shown that κ_0 might converge faster than the associated fundamental eigenmode, since the former is an integral property of the system and the latter is a local property [15]. As previously mentioned in Chapter 4, the Shannon Entropy is an important tool to track the source distribution, but its diagnostics may have some critical aspects.

5.1.1 Main issues of source convergence

The entropy function provides a measure of the phase space exploration as a function of the number of generations: when the neutron distribution attains its stationary shape, the entropy converges. So, by a scalar value the required information on the spatial repartition is condensed. Moreover, as apparent from Eq.:4.3, the entropy of the source distribution at the g-th generation is bounded, namely [15]:

$$0 \leq S(g) \leq \log_2 B \tag{5.1}$$

where B is the number of cells of the spatial mesh, i.e. the detectors. This fact ensures that the highest value of S.E. is reached in case of *perfect equipartition* [15]. The issue of the S.E. use is the impact of correlations induced by fission events: obviously, a neutron can only be generated in the presence of a parent particle, so a 'generation-to-generation' correlation arises, with neutrons clustering close to each other after a few generations. There is an asymmetry between correlated fission 'births' and uncorrelated 'deaths' by capture and leakage. This is problem is expected to affect the convergence of Monte Carlo simulation results and makes Central Limit Theorem applicability objectionable. The impact of this issue is inversely proportional to the number of neutrons per generation [15]. Hence, the entropy function might in turn be ineffective at detecting these potential deviations of the neutron population with respect to the expected equilibrium because of compensation of terms due to the 'integral' nature of the Shannon Entropy, which may lead to a false converge for high dominance ratio (See chapter 4) or loosely coupled problems in which the source distribution does not change much, with the S.E. remaining constant for the same not converged source distribution [1]. This problem can be overcome with the parallel use, if it is possible, of the S.E. and the Centre Of Mass (C.O.M.) techniques. Obviously the latter is meaningful only for symmetric, homogeneous domains, because it consists in computing the vector position $\vec{r_i}$ of each sub-region (i.e. detector) relative to the geometric centre of the model. Its formulation is given by [1]:

$$\vec{R^{(g)}} = \sum_{i=1}^{N} q_i^{(g)} \vec{r_i} \tag{5.2}$$

where g refers to the cycle number and q_i to the normalized source of the i-th detector. If there is source convergence, C.O.M. coordinate will (more or less) mildly oscillate around the zero.
In order to test it, it is possible to analyse a simple, homogeneous slab with data taken by Table4.4 and varying population per cycle, thickness of the single layer and number of detectors using both κ_0 and γ_0 algorithms. The 0-th generation neutrons start their random walks at the centre of the slab. 100 cycles are performed.
The first parameter to be modulated is the dimension of the slab (L).

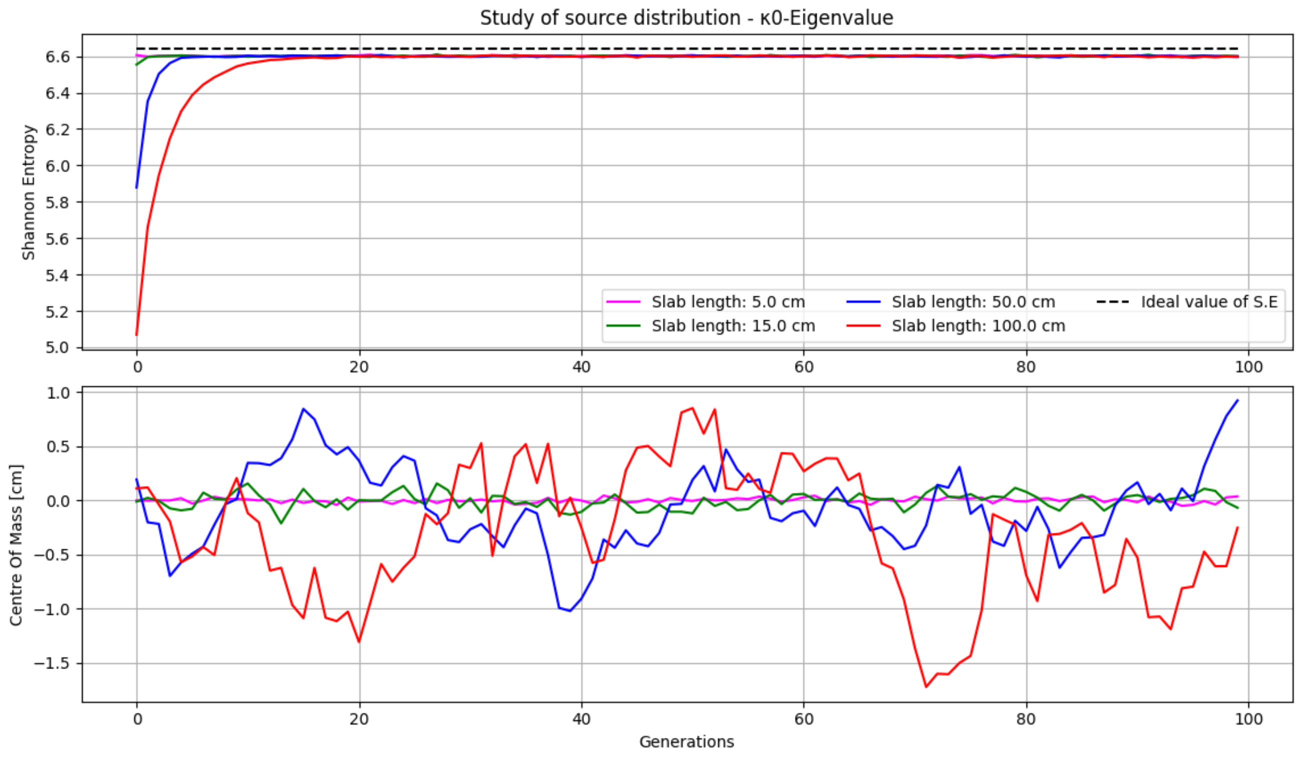


Figure 5.1: Shannon entropy and Centre Of Mass coordinate evolutions in a slab with variable length: κ_0-eigenvalue.

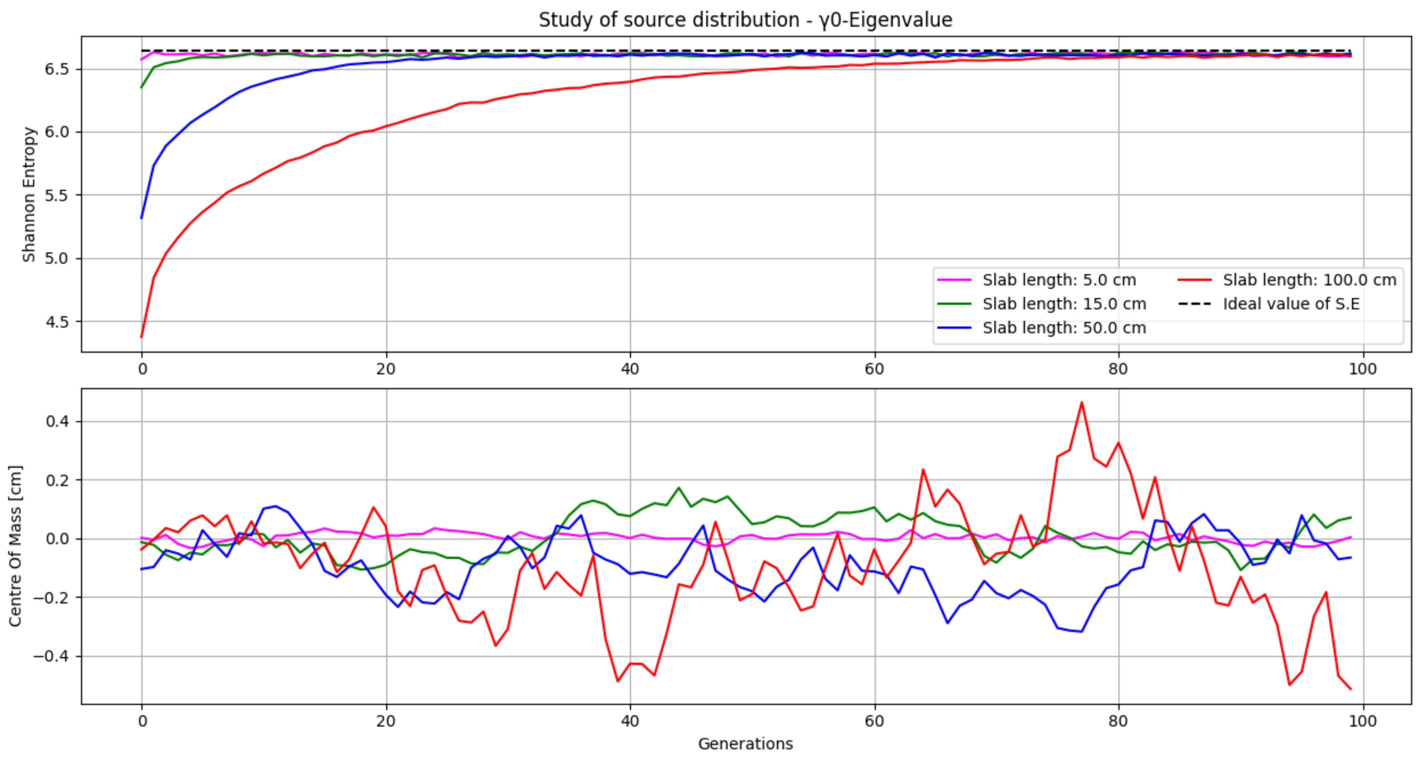


Figure 5.2: Shannon entropy and Centre Of Mass coordinate evolutions in a slab with variable length: γ_0-eigenvalue.

Figs.5.1-5.2 show the evolution of S.E. and C.O.M. coordinate for each eigenvalue. The number of neutrons per cycle (N) is fixed at 10000 and the number of detectors (B) at 100. When the neutron density is high (i.e., L is small for a given N), the fission sites converge to an equilibrium configuration where neutrons are homogeneously spread over the whole slab, with mild fluctuations mostly due to scattering. As the thickness increases, spatial fluctuations due to the competing mechanisms of fission, absorption and scattering become more apparent, in particular if one observes for C.O.M. coordinate, while these fluctuations are almost imperceptible for the S.E.: the asymptotic value of S.E., being N and B constant, is quite similar for all the slabs and close to the ideal value and upper bound of relation 5.1, i.e. $\log_2(100)$ (but the greater is the slab, the more empty cells there will be). For even larger L, the neutron population displays patchiness, with neutrons randomly moving around the slab grouped into a large cluster. The effects of spatial correlations becomes stronger, and the evolution of C.O.M. coordinate becomes increasingly erratic [15]. S.E. evolution, on the other hand, shows that the needed number of generations to reach an asymptotic value is different for each slab. This fact has physical meaning: if all the initial neutrons start their random walks at the centre, the larger is L, the more generations reaching all the detectors will take [1]. This number of cycle is smaller, for equal slab dimension, in κ_0 source evolution than in γ_0 one. The reason is simple: each neutron covers a smaller distance if scattering sites must be stored too. C.O.M. coordinate, on the contrary, presents slighter oscillations around the zero for γ_0's source evolution: because, again, the average distance covered by a neutron generation is smaller for γ_0.

[1]The number m of generations taken by the neutron population to achieve spatial convergence (i.e., to explore the whole reactor) starting from a point source can be roughly estimated by the ratio between the slab length and mean square displacement of a particle per generation [15]

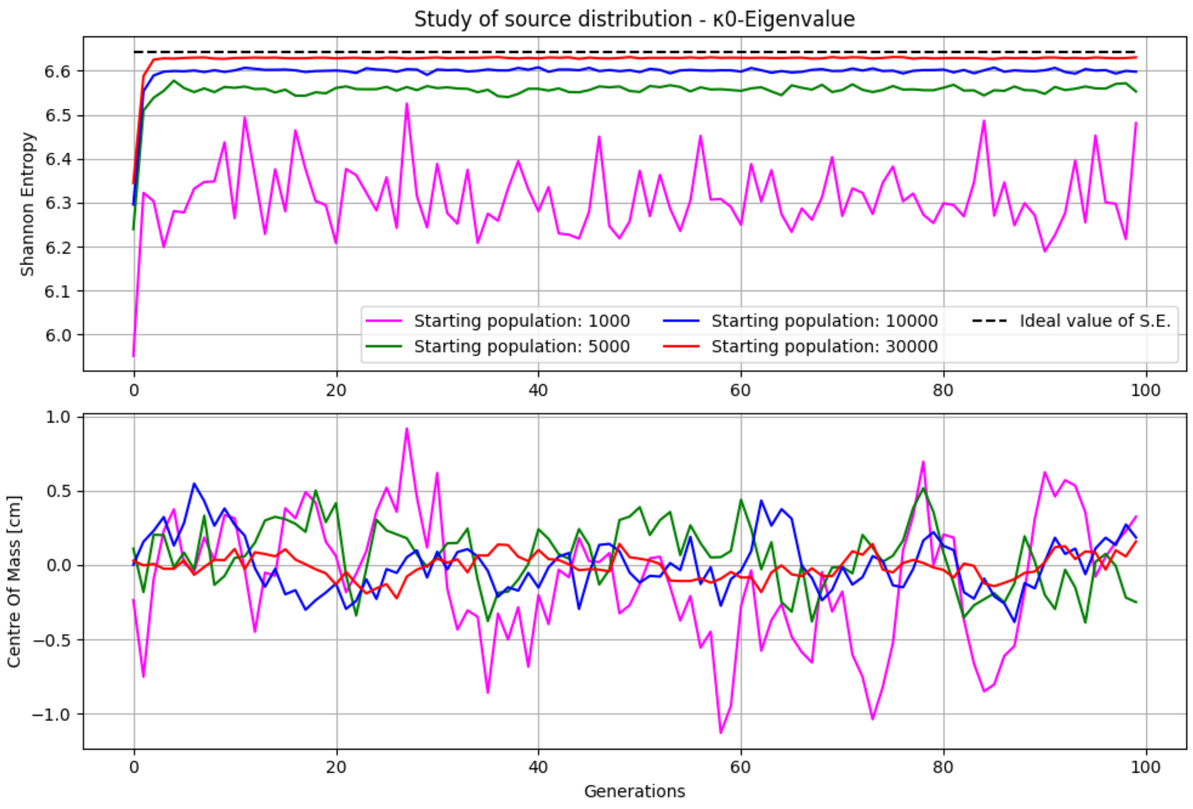


Figure 5.3: Shannon entropy and Centre Of Mass coordinate evolutions in a slab with variable population per cycle: κ_0-eigenvalue.

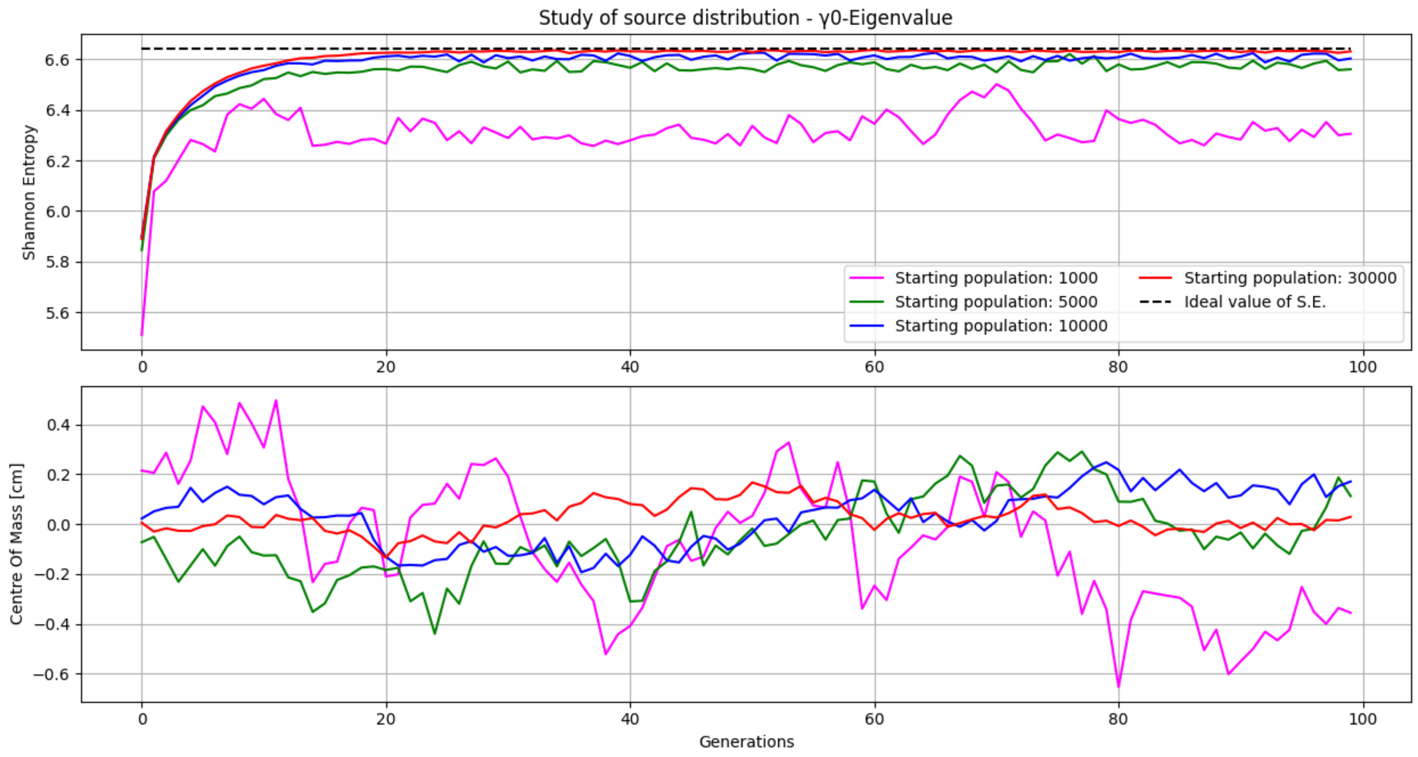


Figure 5.4: Shannon entropy and Centre Of Mass coordinate evolutions in a slab with variable population per cycle: γ_0-eigenvalue.

If L and B are kept constant and N is modulated as shown in Figs.5.3-5.4, things will change. Now L remains equal to 30 cm and B to 100. When the parameter to be changed was the slab dimension, spatial correlation effects were more visible in C.O.M. coordinate evolution (except for the number of cycle needed to reach the asymptote for S.E.). Now fluctuations are evident in both of the graphs of the single figure. First of all, it is clear that S.E. reaches its asymptotic value after the same number of cycles for every population (indeed this number of cycle is not a function of the number of neutrons in the reactor). The presence of a certain amount of empty cells with a low population per cycle is also clearer in this case: the less neutrons a cycle has, less widely distributed their sources will be, with a consequent decrease of the S.E.. So, as N decreases, spatial fluctuations become more apparent, and for even smaller populations neutron clustering eventually sets in [15]. Also in this case, γ_0's C.O.M. coordinate has slighter oscillations with respect to corresponding quantity of κ_0.

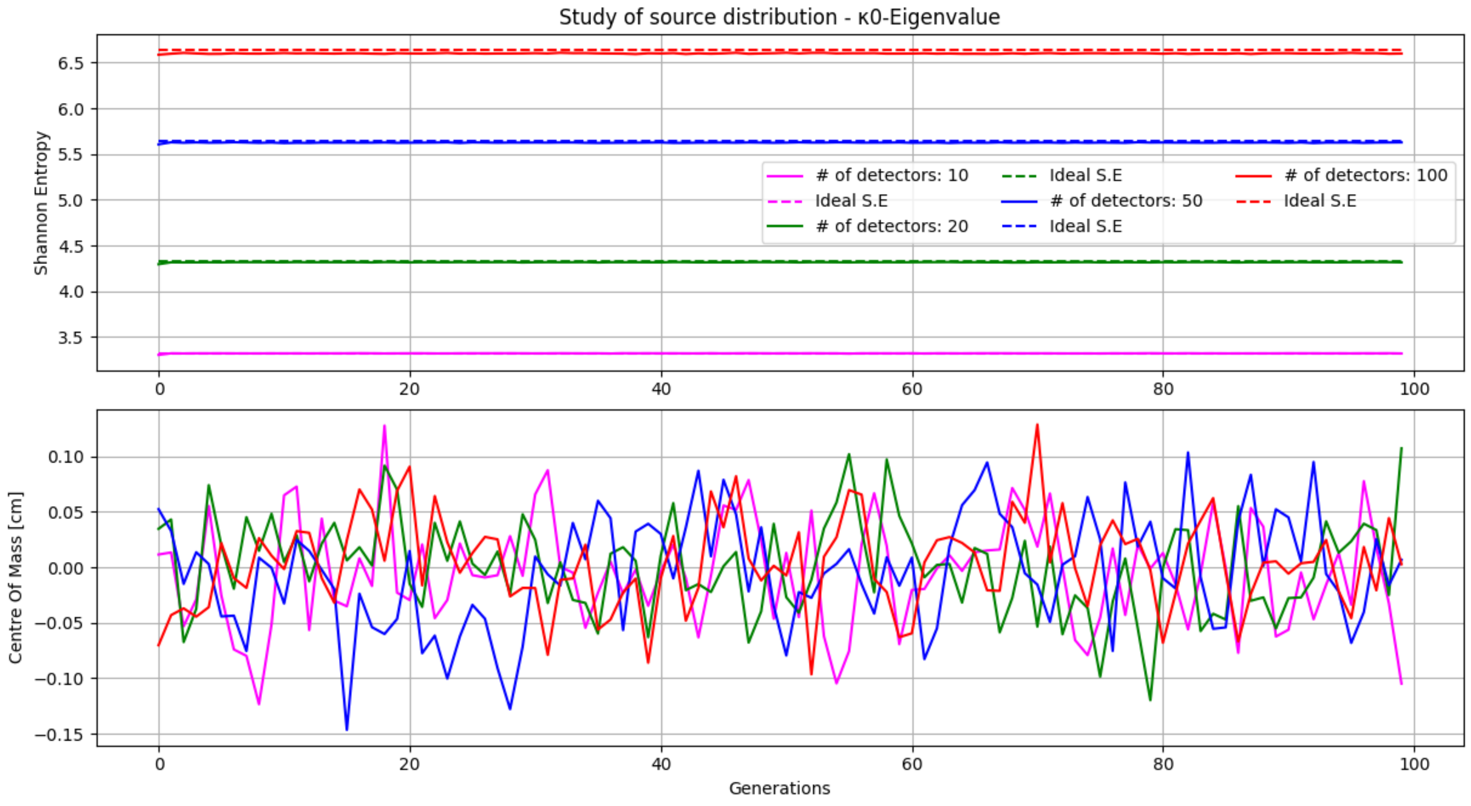


Figure 5.5: Shannon entropy and Centre Of Mass coordinate evolutions in a slab with variable number of detectors: κ_0-eigenvalue.

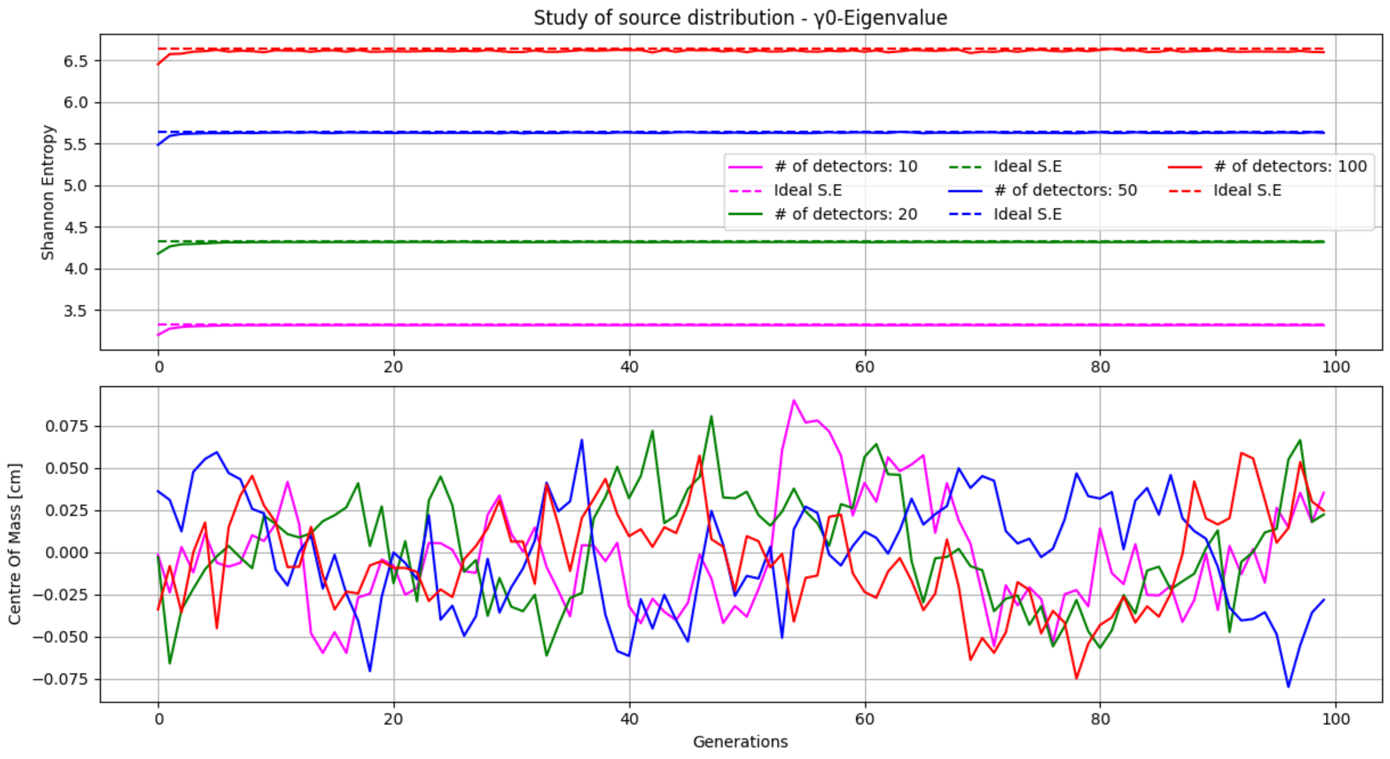


Figure 5.6: Shannon entropy and Centre Of Mass coordinate evolutions in a slab with variable number of detectors: γ_0-eigenvalue.

The last parameter to be modulated is B. It is done by keeping N and L constant, respectively equal to 10 cm and 10000 neutrons per cycle. As depicted in Figs.5.5-5.6, the C.O.M. coordinate evolution is useless in this case, because $\vec{R^{(g)}}$ is, by construction, independent on the number of spatial meshes [15]. Hence, this time, more information might be collected from S.E. evolution chart. It presents, at fixed N, a little distance between asymptotic value and ideal value of S.E. that increases with B, because when the number of mesh B is larger, the number of particles N required to mitigate the effects of correlations in each detector must be also larger [15]. Therefore, a too high number of detectors with respect to the neutron population per cycle is quite detrimental in convergence of S.E., but, on the other hand, if there are a few detectors, when normalization processes sample the new neutrons' positions (as Chapter 4 has shown), the new coordinate may have been varied dramatically, causing unexpected leakages for instance. A good choice would

be a detector dimension that is a fraction of the mean free path in such a way that the new position after normalization is not too much different from the 'original' one.
These diagnostic tests have demonstrated that many factors play a role in source convergence. A good 'rule of thumb' may be derived from the diffusion theory approximation: some previous investigations have found that phenomena of spatial clustering are quenched when $L^2 << M^2 N$ where M^2 is the migration area, the sum of thermal and fast neutron diffusion areas [15]. Thanks to the data of Table:4.4, it is possible to evaluate this inequality for every single treated case. Being the fast diffusion area L_1^2 equal to D_1/Σ_{a1} and the thermal diffusion area L_2^2 equal to D_2/Σ_{a2}, it results that L_1^2 is 652.17 cm^2 and L_2^2 is 2.0 cm^2. Hence, their sum is 654.17 cm^2.
For instance, when the slab dimension was 5 cm with fixed $N = 10000$ neutrons per cycle, the inequality was $25cm^2 << 6.5 \times 10^6 cm^2$, while with a thickness of 100 cm, it was $10^4 cm^2 << 6.5 \times 10^6 cm^2$. When the fixed parameter was the slab dimension (30 cm) and variable population per cycle (respectively 5000 and 30000) the relations were: $900cm^2 << 3.3 \times 10^6 cm^2$ and $900cm^2 << 1.9 \times 10^7 cm^2$.
The inequalities above demonstrate that spatial clustering, although it is present, is more or less always negligible in the case of the plotted examples, but the issue of spatial correlations cannot be neglected.

5.1.2 The choice of the best source

In the Chapter 4 dealing with high computational time was seen from the perspective of parallel processing. Another fundamental way of reducing it is to properly configure the source at the 0-th generation. The situations analysed in Section 5.1.1, with their initial source at the centre of the slab, wanted to stress the fact that two eigenvalues reach the convergence with different 'speeds', but those source choices were not accurate at all. The graphs this section is going to present will show how the parameters that define the initial source (from Chapter 4) affect the evolution of the S.E. and of the eigenvalue. The slab to be examined have materials from Tables:4.2-4.3 with a [Moderator-Fuel-Moderator] scheme of layers and each of them has a thickness of 2 cm.

Fission-source-like distribution

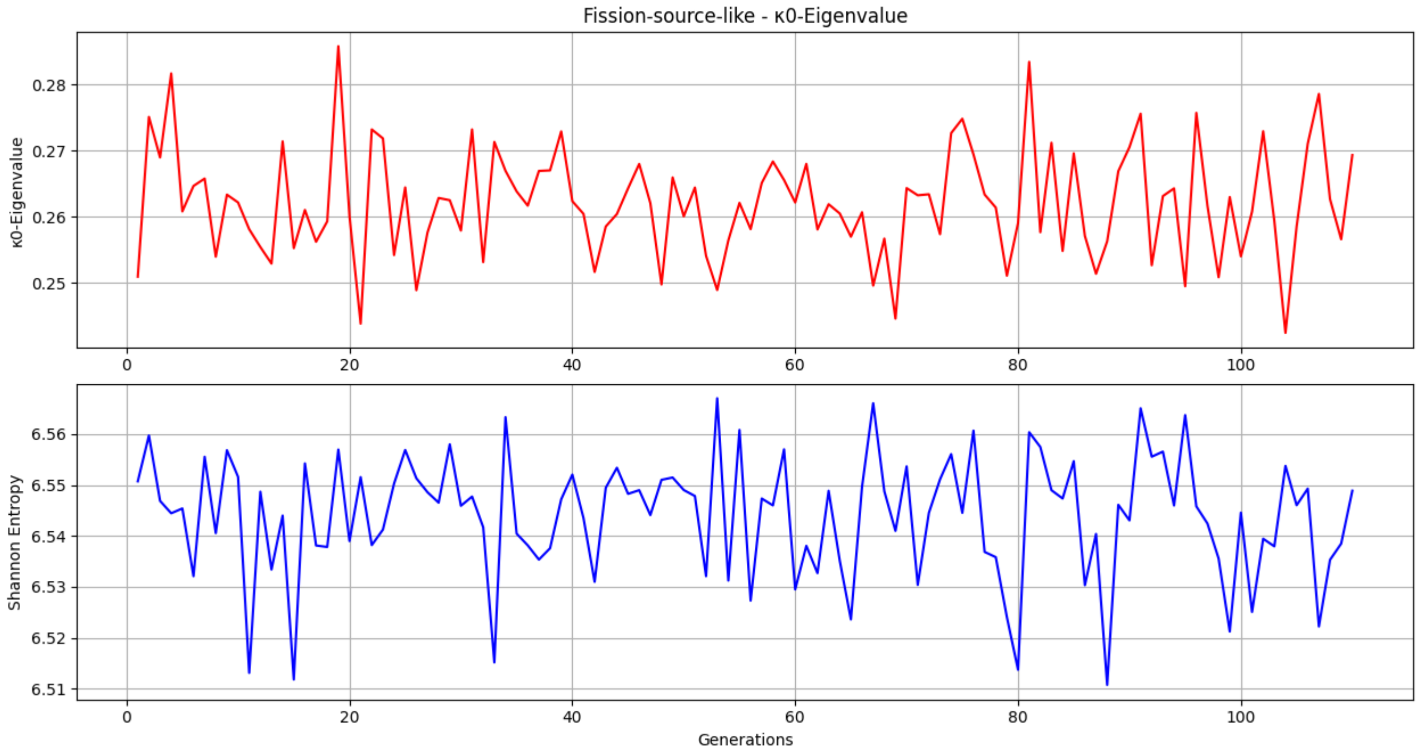


Figure 5.7: κ_0-eigenvalue and Shannon entropy evolutions with a 'Fission-source-like' distribution.

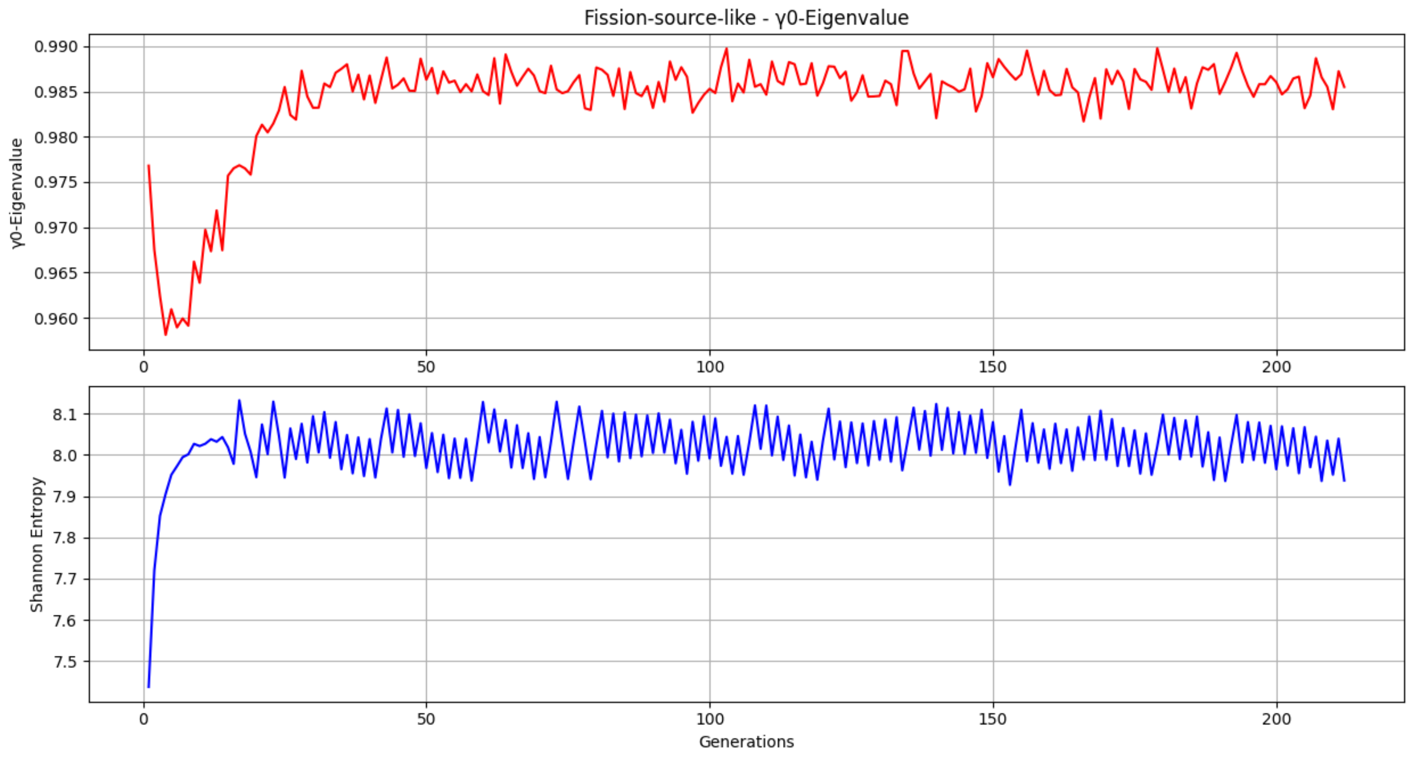


Figure 5.8: γ_0-eigenvalue and Shannon entropy evolutions with a 'Fission-source-like' distribution.

As section 4.1.3 have illustrated, the 'Fission-source-like' distribution sorts the neutrons with a random, uniform distribution into the detectors of fuel layers with energy groups that follow the fission spectrum. It is a good choice for the computation of κ_0, because it follows the recommendation to cover all fissionable regions [16]. If the eigenvalue to be computed is γ_0 instead, the region hosting neutron sources are the moderator layers too. Thus, this distribution becomes less useful for this purpose, because the propagation in the other layers take some cycles. Figs.5.7-5.8 show clearly this difference between the two eigenvalues. In particular, γ_0 at the beginning presents a kind of 'overestimation' due to the large number of neutrons in fuel region with more probability of causing fissions. After the propagation through other regions, the γ_0 sees its value drop and then grow again to the ultimate stationarity values. The S.E. trend seems to prove that this distribution is not very suitable for γ_0 computation.

Uniform distribution

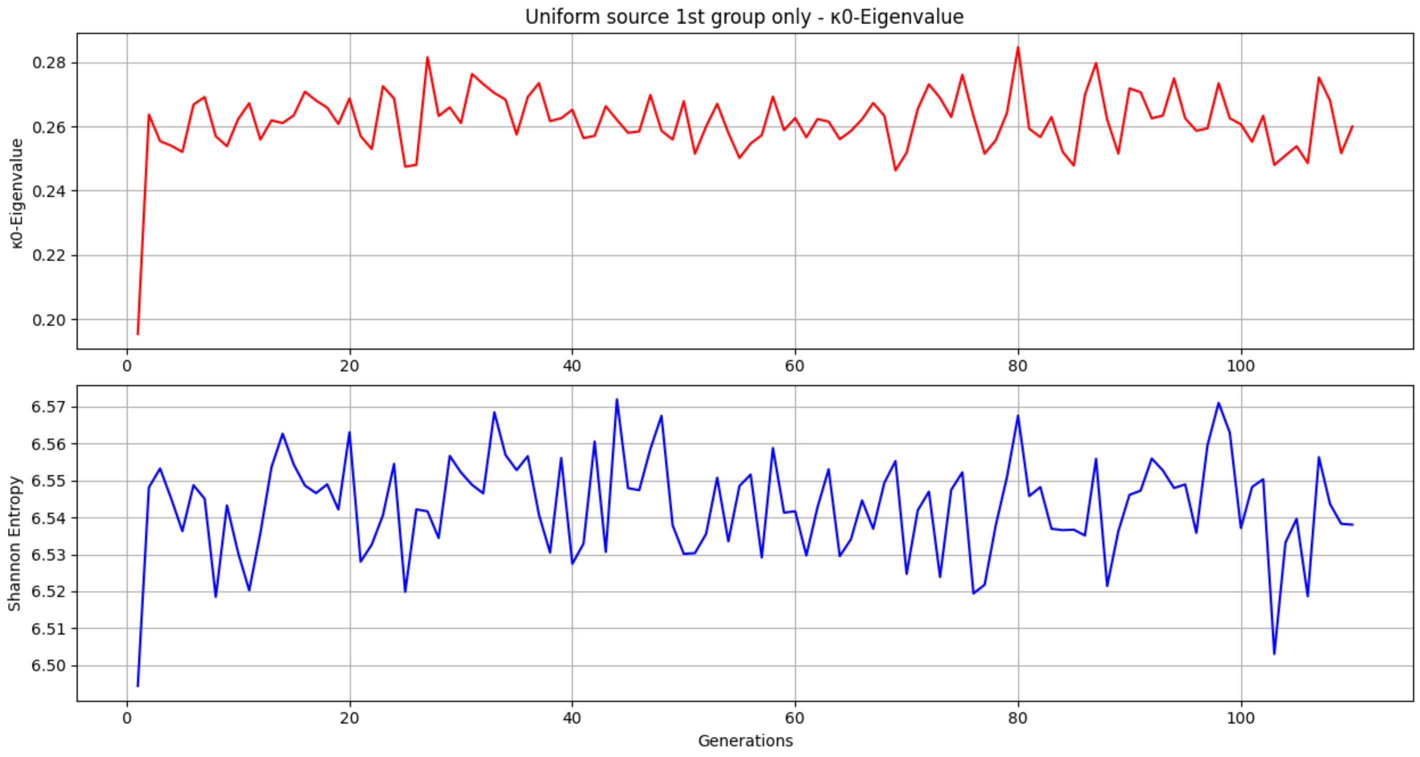


Figure 5.9: κ_0-eigenvalue and Shannon entropy evolutions with a 'Uniform' distribution and fast neutrons

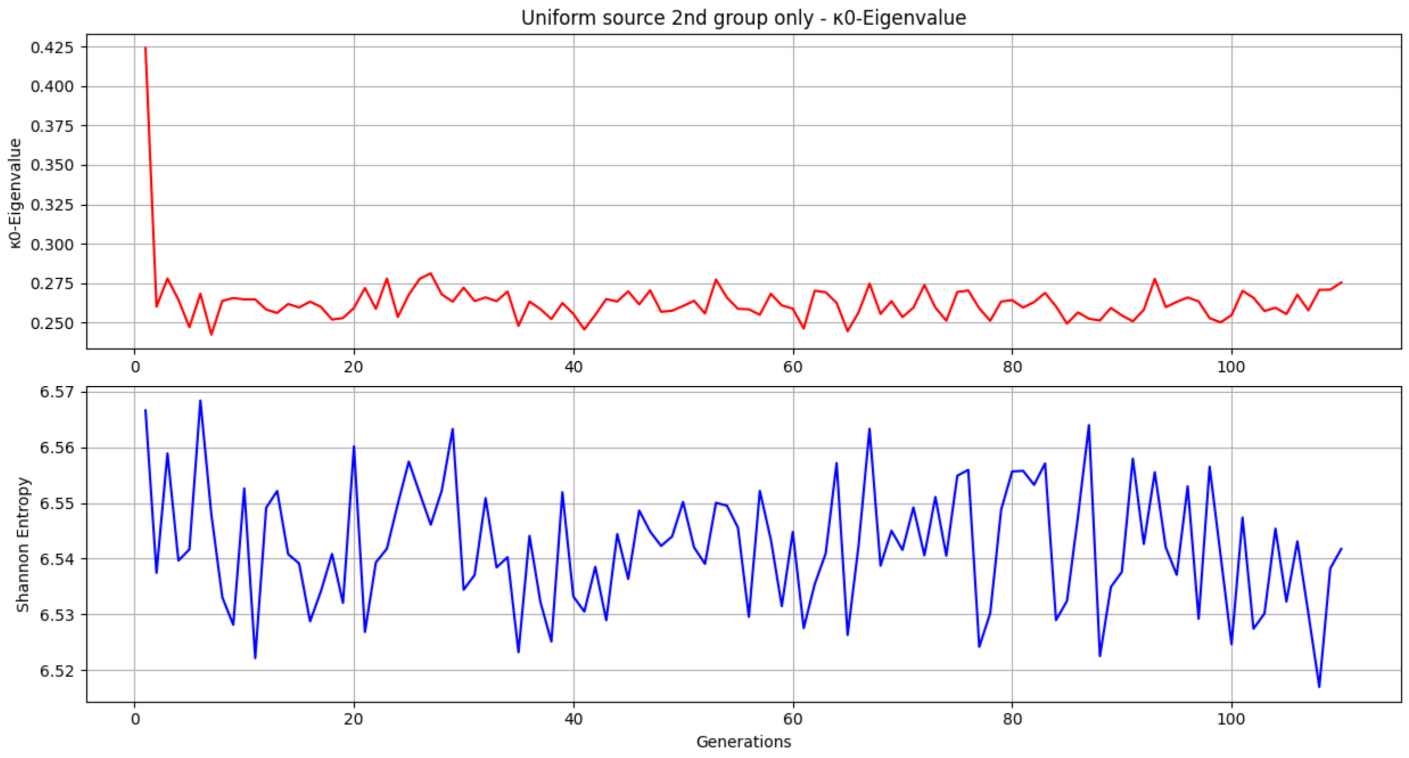


Figure 5.10: κ_0-eigenvalue and Shannon entropy evolutions with a 'Uniform' distribution and thermal neutrons

With 'Uniform' distribution, the initial neutrons are equally spaced throughout the slab, regardless of the detectors or the type of layers. Since this thesis deals only with two energy groups, every neutron of this distribution will be from the first level (*fast group*) or from the second level (*thermal group*). Figs.5.9-5.10 namely depict these two different energy distributions for κ_0-eigenvalue case. Since the faster neutrons move, the less probable interactions will be, neutrons starting from 'thermal' energy group (Fig.5.9) undergo more interactions and, thus, more fissions before reaching the 'real' range of values. This source, in some way, overestimates the criticality, while the source in which every neutron is fast underestimates it at the beginning (leakages are more probable), but the correct range of eigenvalue is reached immediately afterwards. Hence, the 'Fission-source-like' distribution is generally better for the κ_0-eigenvalue calculation, even if the difference in speed of convergence is not so huge, since the overestimation/underestimation dies out in a few cycles. S.E. has no great peculiarity to be stressed (except for a very mild, initial underestimation and overestimation respectively in the 'fast' situation and in the thermal one).

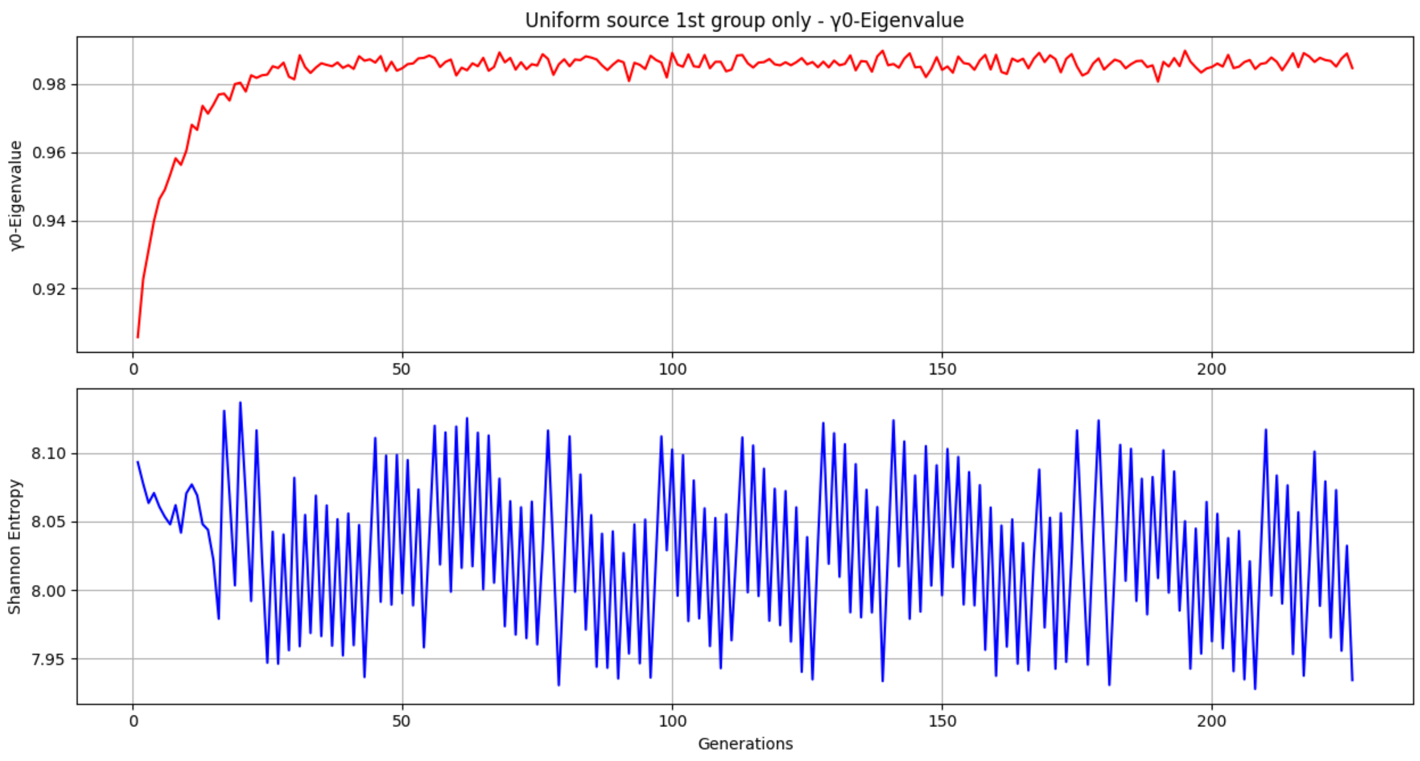


Figure 5.11: γ_0-eigenvalue and Shannon entropy evolutions with a 'Uniform' distribution and fast neutrons

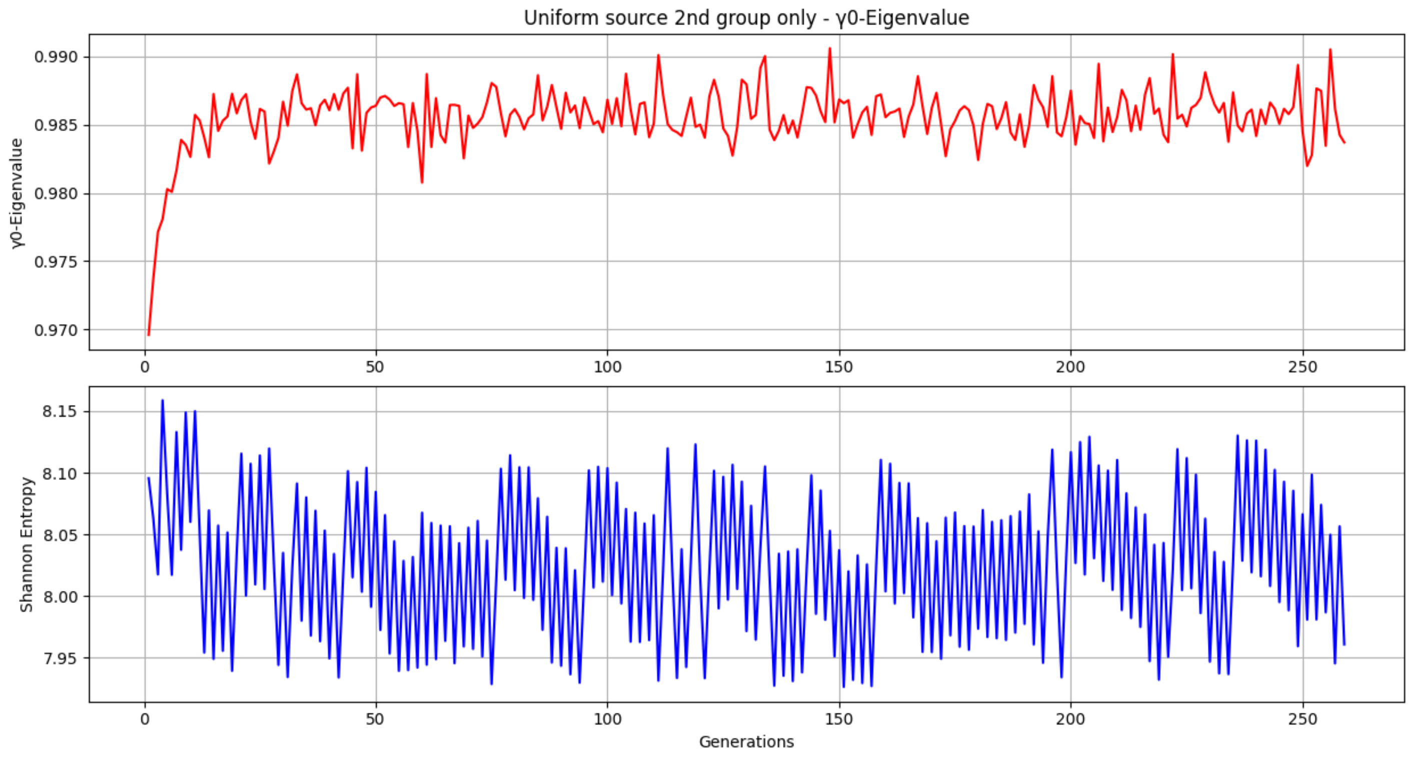


Figure 5.12: γ_0-eigenvalue and Shannon entropy evolutions with a 'Uniform' distribution and thermal neutrons

Figs.5.11-5.12 show the same situation, but in the case of γ_0-eigenvalue. At the beginning of the cycles the overestimation/underestimation of the eigenvalue, that was so evident for κ_0 case, is still present, but is hidden by the fact that the majority of interactions are scattering events; so if the initial neutrons are all thermal, they will likely undergo more fission events (more likely in this energy level) with respect to the case with all fast starting neutrons. This fact makes γ_0 start to grow from higher value for thermal neutron source with respect to fast neutron source, but always below the range of stationarity oscillations. On the other hand, the S.E. already oscillates in the stationary range at first cycles for the 'initially fast' situation while the other one is slightly higher: since thermal neutrons, if they do not do fission, tend to stay thermal (up-scattering, at least, very unlikely) and cover smaller distances, they take time to get concentrated in central regions and also the leakages will be less frequent. So, if 'fast source' has a better S.E.'s initial range, on the other hand has a lower, starting γ_0. The two situations are not so different, but for the code validation in the last part of the chapter, 'thermal source' has been chosen.

Symmetry

Not only the position and the energy of initial neutrons are important, but also which environment they interact with: taking into account only one half of the domain is very useful, if the slab geometry allows it, to reduce the size of the reactor, the number of detectors (it will be easier to fill them up) and the dominance ratio [16]. Thus, this change implies a better distribution of neutrons[2], a consequent convergence improvement and so a faster computation. Figs.5.13-5.14 may help to understand by comparing them with the corresponding 'full-slab' situations (Figs.5.7-5.12): in particular, γ_0 shows a better convergence situation in both of the charts, also in terms of cycles needed to the eigenvalue to reach stationarity.

[2]Obviously, with fixed number of neutrons per cycle

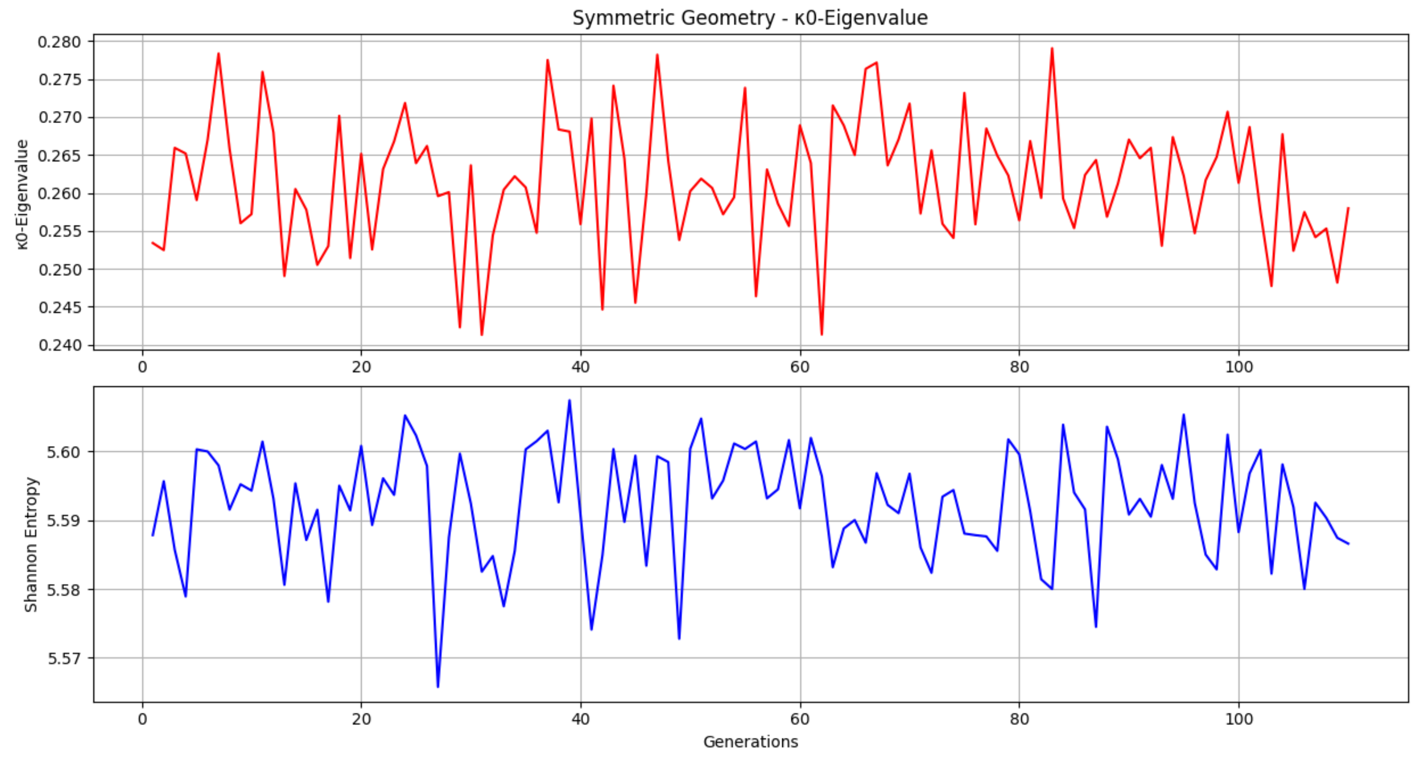


Figure 5.13: κ_0-eigenvalue and Shannon entropy evolutions with a 'Fission-source-like' distribution in half-domain.

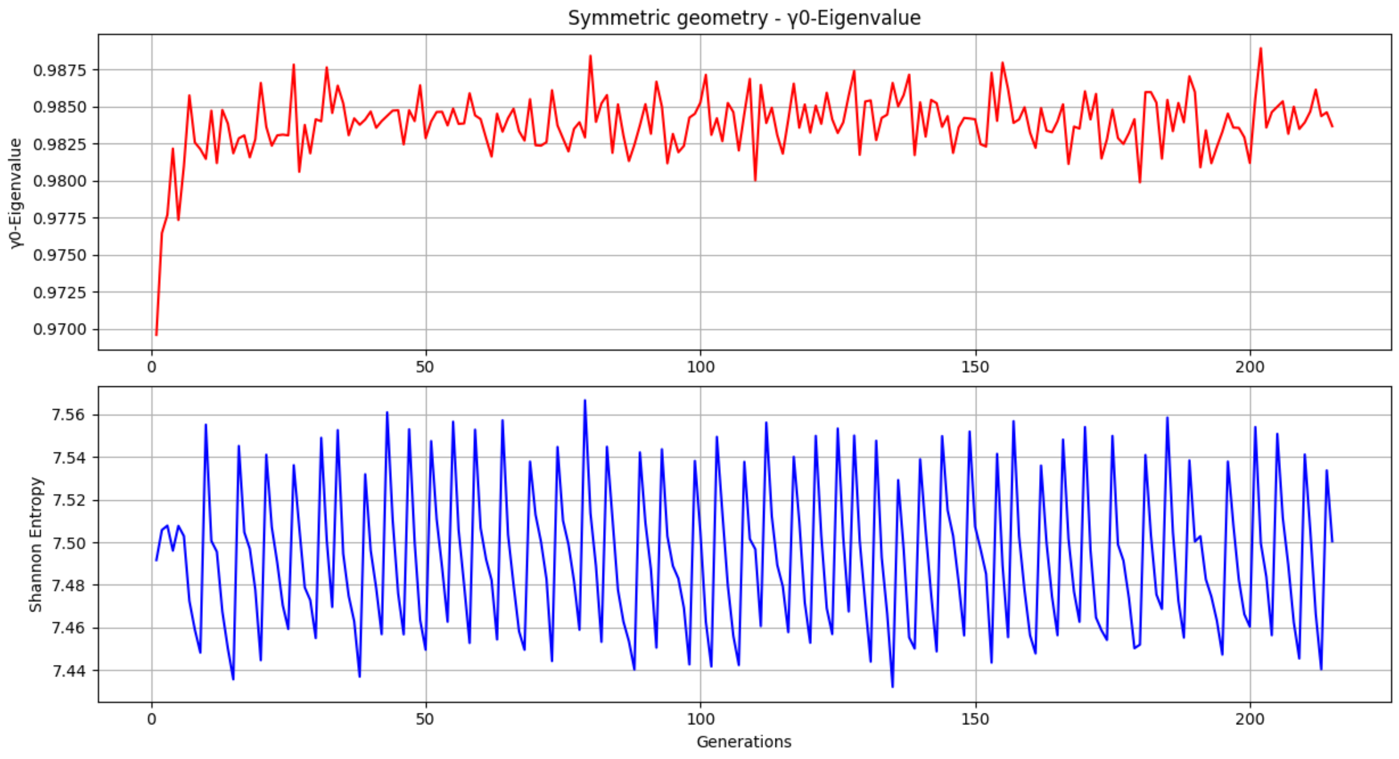


Figure 5.14: γ_0-eigenvalue and Shannon entropy evolutions with a 'Uniform' distribution (all thermal neutrons) in half-domain.

Before applying the symmetric geometry, the slab symmetry must be verified. For this purpose, a new method is added to the Domain Assembler class: the function *'Issymmetric'* whose output is a Boolean variable (True if the slab is symmetric and False if not). It checks if the two halves are equal from the standpoint of materials and from the standpoint of thicknesses. The random walk function has to be modified too. Another virtual collision must be take into account, the one at the axis of symmetry: energy remains constant, the starting neutron position is updated to the axis of symmetry, the cosine of the direction changes its sign and the free flight length is sampled according to the material of the layer where the axis of symmetry lies. The method registering the crossed boundaries must be consequently modified to consider the axis of symmetry's abscissa too.

Direction

If the central layer is very thin, for instance the scheme is [2.0-0.5-2.0] cm, one may think to make things easier by directing all the initial neutron towards the centre of the slab (cosine directions equal to 1 or -1). For both the eigenvalues the distribution is Uniform and thermal, so the effects for random directions are analogous to the ones of Figs.5.10-5.12(but γ_0's S.E. has oscillations already in the 'correct' ,Fig.:5.17, range because of convergence made easier by the reduced dimension of the slab). If the κ_0 computation starts with inward-directed neutrons, the aforementioned, initial overestimation is furtherly increased, as depicted in Figs.5.15-5.16. Then, κ_0 lowers to the usual correct range of values. S.E. seems not to be greatly affected by this type of directions.

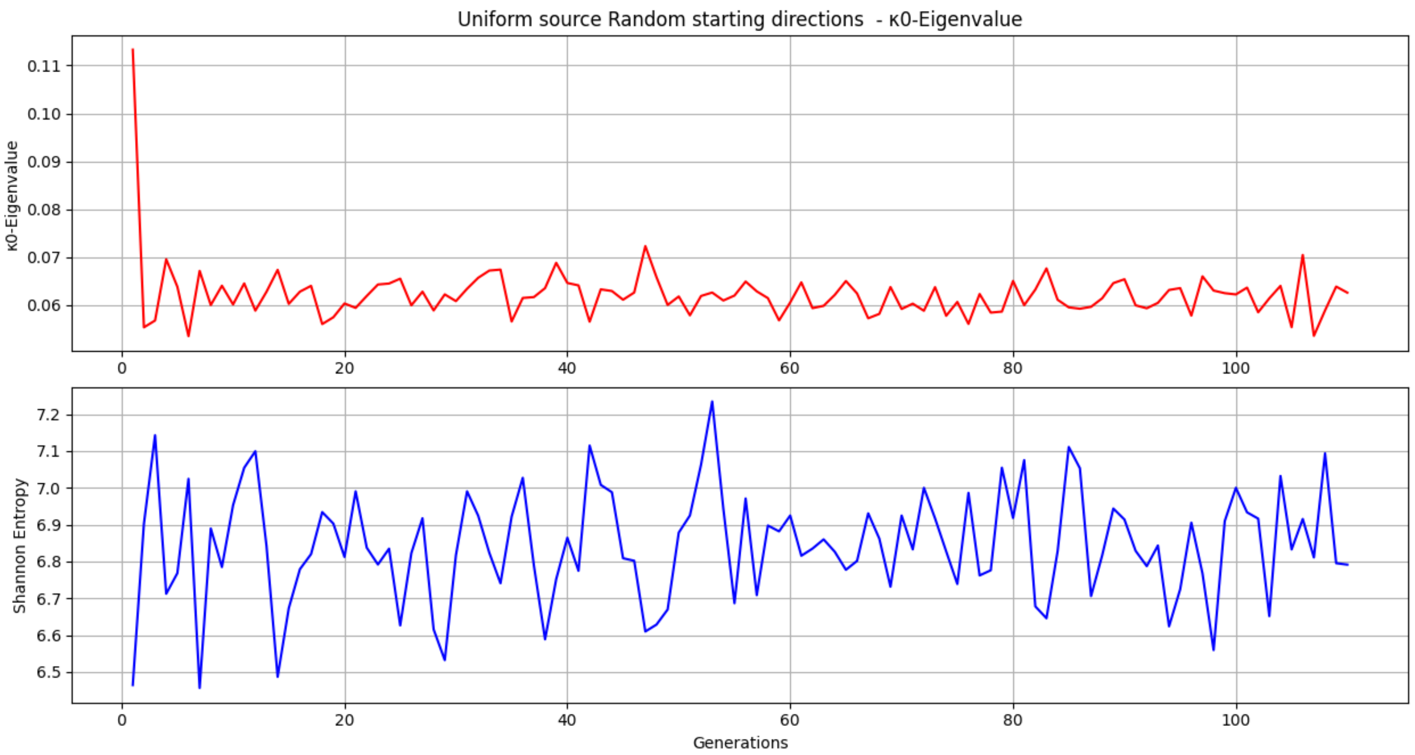


Figure 5.15: κ_0-eigenvalue and Shannon entropy evolutions with a 'Uniform' distribution (all thermal neutrons) and randomly-directed initial neutrons.

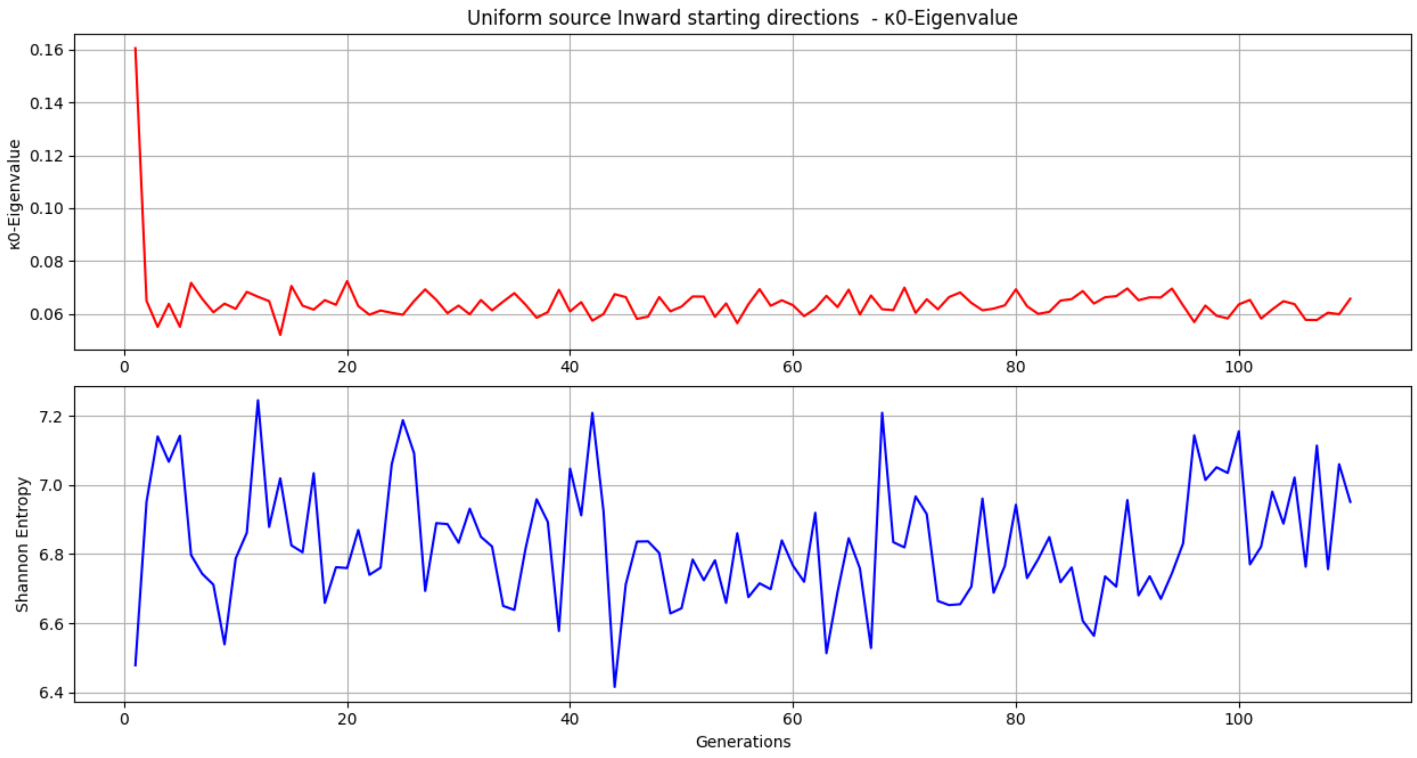


Figure 5.16: κ_0-eigenvalue and Shannon entropy evolutions with a 'Uniform' distribution (all thermal neutrons) and inward-directed initial neutrons.

This time, the overestimation is clearly visible also for γ_0 (Fig.:5.18); then it lowers to the normal range which, instead, is reached more quickly than in the 'random-directions' case. At first, S.E. is higher than the 'stationarity' oscillations range because leakages are absolutely absent.

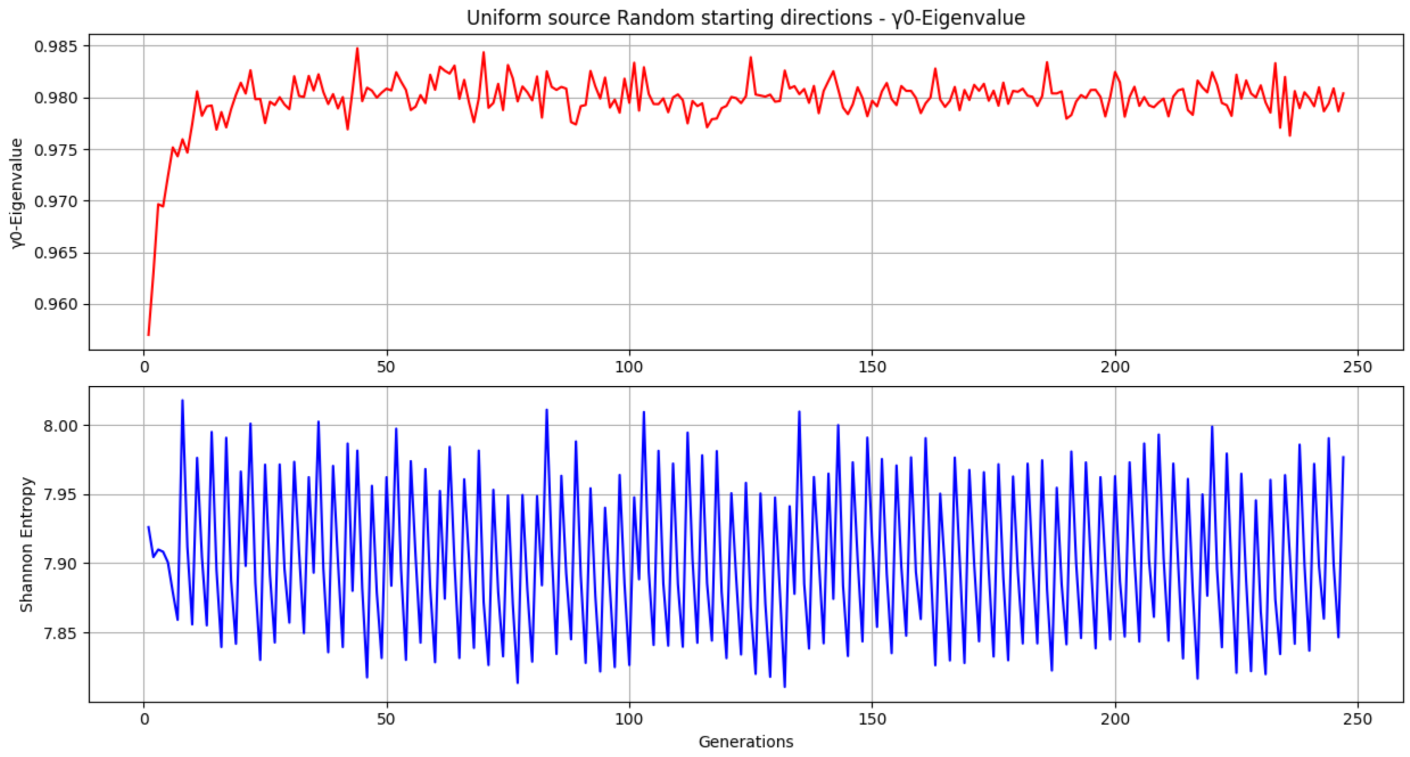


Figure 5.17: γ_0-eigenvalue and Shannon entropy evolutions with a 'Uniform' distribution (all thermal neutrons) and randomly-directed initial neutrons.

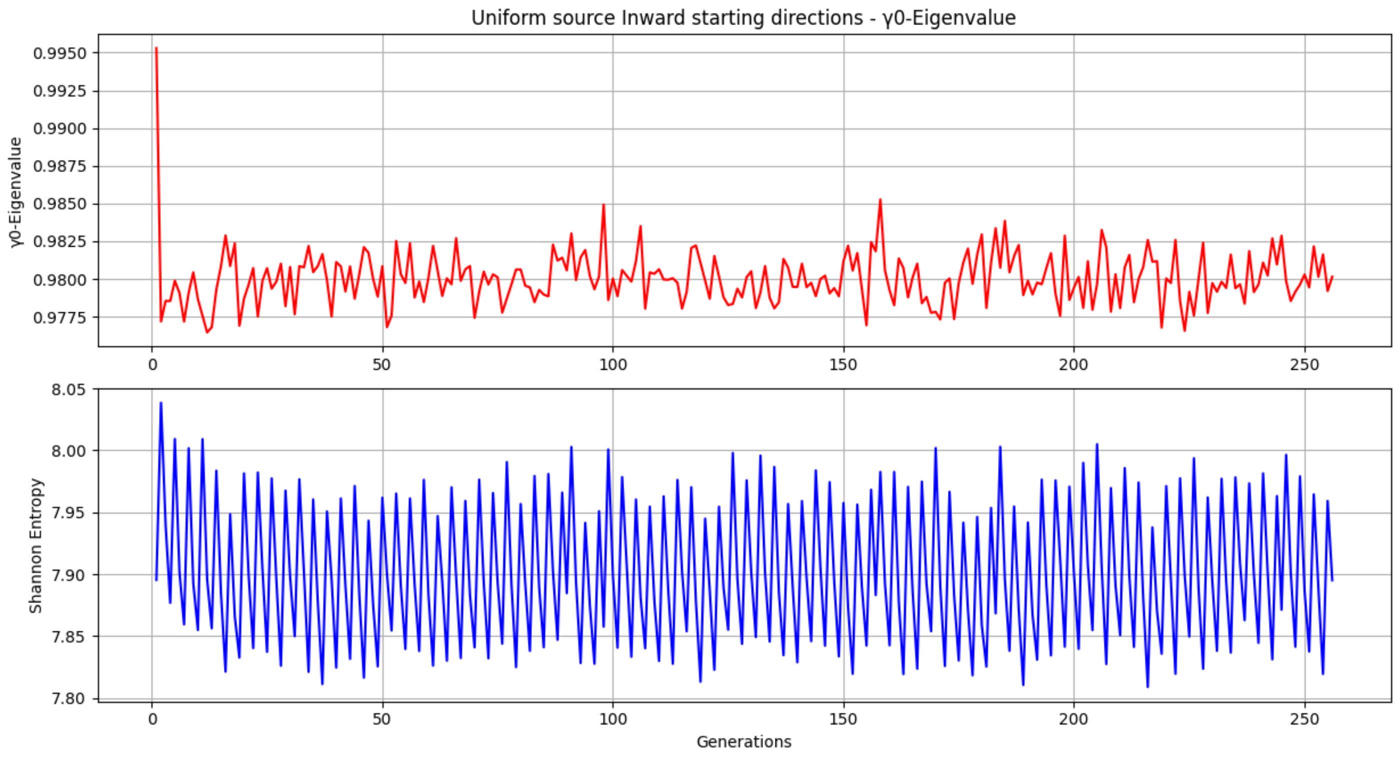


Figure 5.18: γ_0-eigenvalue and Shannon entropy evolutions with a 'Uniform' distribution (all thermal neutrons) and inward-directed initial neutrons.

These graphs show clearly the lack of utility in directing in this way neutrons for computing κ_0, but the improvement, at least for the eigenvalue (not for the S.E.) is tangible for γ_0. So this type of direction for the initial source is not completely useless for thin fuel regions and it deserves the investigation as much as the other types of source modification.

These experiments have shown that the only truly decisive source parameters are the distribution and the geometry. They also have displayed that γ_0 and its S.E. are slower to converge. Energy levels and direction changes have been useful only for illustrating the initial neutron behaviour with different source's features.

If one compares every S.E. evolution chart of κ_0-eigenvalue with the one of γ_0-eigenvalue, once reached the 'stationarity range of oscillations', he/she will not only note the difference in the scale of the oscillation range (γ_0-eigenvalue case has more detectors) but also a kind of periodicity in oscillations, at least in limited sections of the entire set of cycles, that S.E. of the κ_0-eigenvalue does not seem to have. After the initial transient, S.E. for γ_0-eigenvalue has a 'saw-like' trend (Fig.:5.11) that decreases for some cycles before going up again. The reason for this difference between the two examined Shannon Entropies could be that the one of κ_0-eigenvalue does not visualize the periodical migration of neutrons from fuel regions to moderator regions and vice versa, in contrast to S.E. of γ_0-eigenvalue, whose decreasing stretches could represent the progressive fission neutrons build-up, and the following growth after reaching a kind of periodical minimum could indicate the subsequent spread all over the slab; perhaps, if the thicknesses of materials are similar and the distance between the outer boundary and the materials' interface is more or less close to 1 mean free path, this sort of 'pulsed' behaviour is more prominent and more visible through the inspection of the S.E., because neutron sources, with not too thick moderator layers, can spread more deeply in this type of regions and lie not only close to the interfaces with fuel.

5.1.3 Population impact on the eigenvalues

Figs.5.3-5.4 have shown quite well the effects of a not sufficiently high number of neutrons per cycle on the source convergence, but that strongly erratic behaviour also affects the eigenvalue, causing a certain degree of undersampling. Actually, a kind of proportionality between the eigenvalue bias and the population per cycle exists [16] and the two following pictures (Fig.:5.19-Fig.:5.20) depict it (with as abscissas the inverse of the populations per cycle), respectively for κ_0-eigenvalue and for γ_0-eigenvalue of an homogeneous slab with a critical thickness of 3.591204 cm (data from Table:4.1). The active cycles are fixed at 30.

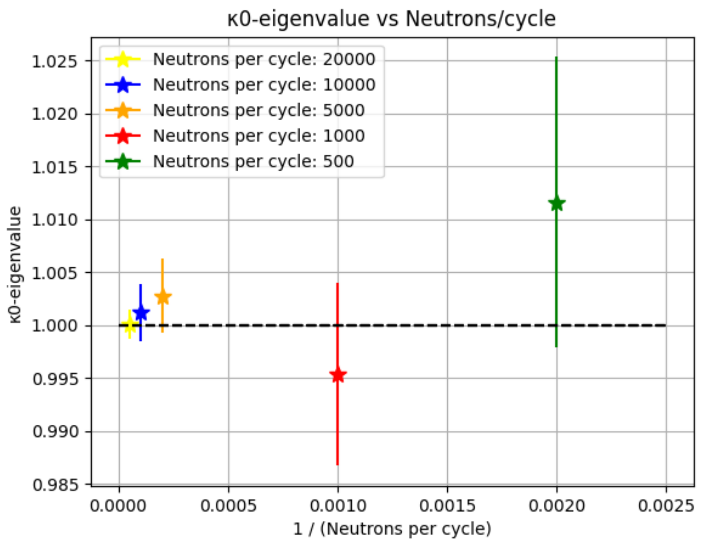


Figure 5.19: Bias in κ_0.

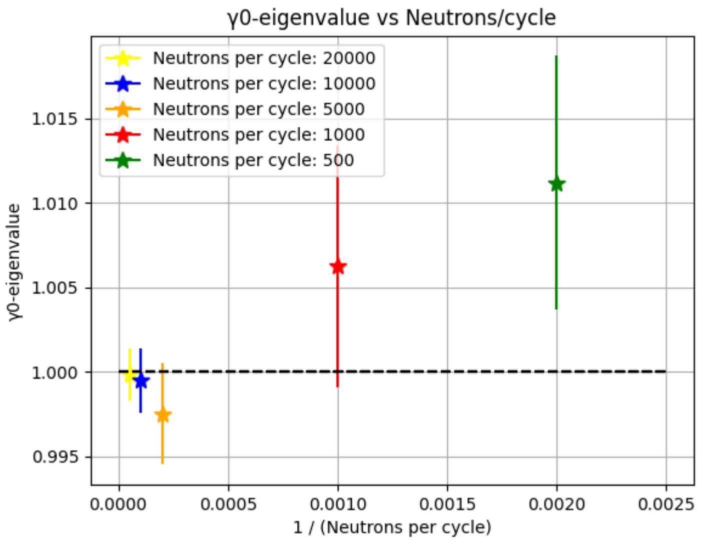


Figure 5.20: Bias in γ_0.

These charts illustrate clearly the gradual increase of distance between the expected value and the sample averages, but also the increase of the degree of uncertainty for the latter values. Obviously, the growth of the population per cycle implies a trade-off with the concomitant, increasing computational time.

5.1.4 Impact of the layers on the eigenvalues

The previous analysis took into account κ_0 and γ_0 behaviours separately. It is worth trying to visualize them together, for instance with a simultaneous variation of the slab, starting from the previous critical slab and then progressively increase/decrease the length by 0.25 cm, as occurred in Fig.:5.21.

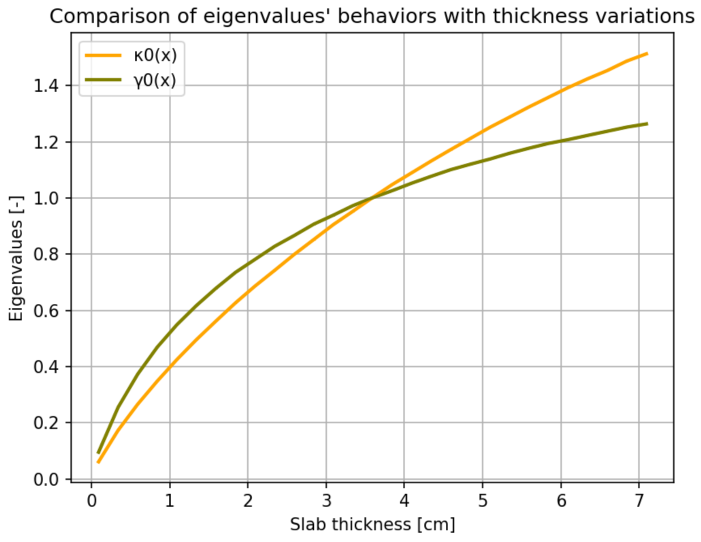


Figure 5.21: Parametrization of κ_0 and γ_0 with the slab length (Homogeneous case).

It is clear that κ_0 is more sensible to the variation of the slab length, if the focus is set on the neighbourhood of the critical thickness: for instance, the focus may be directed on the increment. If the thickness increases of dx, there will be an increment of the interactions: for γ_0 they consist in the

'first' fission events and the scattering events, i.e. every interaction but the leakages and the captures; for κ_0 they consist in the 'first' fission events too, but also 'post-scattering' fission events: so, a kind of transfer from the scattering events to the fission ones takes place. Thus, the fission events caused by the thickness increment are more numerous for κ_0. If the total number of new events is normalized to a certain, common amount for each eigenvalue, the total number of new particles will be greater for κ_0 because its average multiplication factor is bigger (it can vary from 2 to 4 sometimes, while the one of γ_0 is made lower by the major presence of scattering events whose contribute is unitary). Analogously, the decrease of κ_0 is more prominent than the one of γ_0.
It is also interesting to observe how the two eigenvalues react to the modification of the slab length in the heterogeneous case: that is the aim of the following figure. Initially the slab (data from Tables4.2-4.3) has the usual scheme ['Moderator', 'Fuel', 'Moderator'] and thicknesses [1.0-1.0-1.0] cm. Then, they are increased by 2 cm (the fuel region before the moderator region) and by keeping the symmetry. Fig.:5.22 may help to visualize:

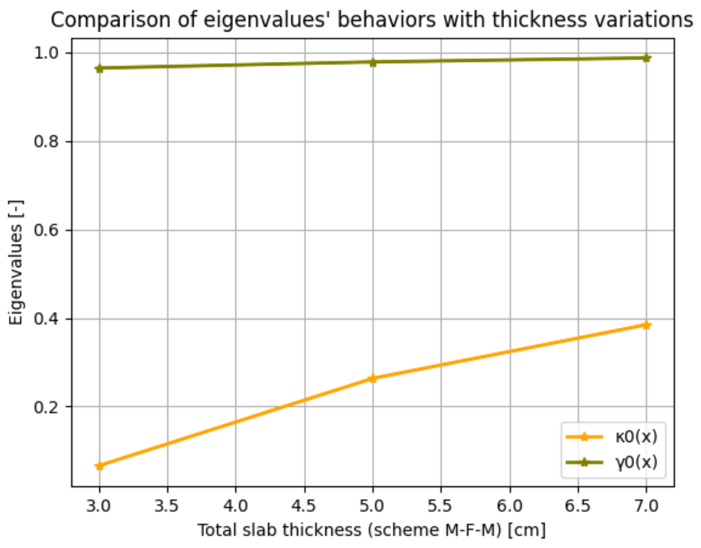


Figure 5.22: Parametrization of κ_0 and γ_0 with the slab length (Heterogeneous case).

The fastest growth still characterizes κ_0 (the lower line), but both the eigenvalues prove to be more sensible to the fuel addition.

5.2 Validation of the codes

Verification is defined as “the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of the phase or as a proof of correctness which is defined as a formal technique used to prove mathematically that a computer program satisfies its specified requirements” [10]. In contrast to verification, validation is “the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements” [10]. Thus code verification checks that the implemented code precisely reflects the intended calculations and that these calculations have been executed correctly. Code validation compares the accuracy of these calculated results usually with experimental data or with other computer codes [10].
The final algorithms turn out to be benchmarks, which are always precious: their continuous development or improvement is important, although they are not used in everyday processes for the dramatic computational cost. Since they simulate how a physical model really evolves, their outputs may be considered as 'the truth' and be compared either with simple diffusion problem outputs or with more complex $\mathrm{P_N}$ approximation ones.

Actually, Figs.5.19-5.20-5.21 have already proven that the results from this code are good because the criticality of their configurations come from the works of others ([4]-[10]). Something is still missing: for instance, is the Central Limit Theorem respected? And the decrease of uncertainty of the sample mean follows its typical behaviour? These questions almost overpass the boarder between validation and verification, because they are equivalent to asking whether the code solves problems correctly rather than asking if the code solves the correct problem. This last part is a kind of fusion of this two concepts and involves data from both [4] and [10]. From the former the check of the Central Limit Theorem is implemented by performing 100 trials with 2, 10 and 100 experiments per trial (obviously for both the eigenvalues in critical situation); from the latter a single trial is performed for each eigenvalue, but with different, critical thicknesses: κ_0 is calculated for a slab with the usual scheme ['Moderator', 'Fuel', 'Moderator'] and lengths [1.126152, 13.393604, 1.126152] cm, and γ_0 with lengths [5.630757, 9.726784, 5.630757] cm. In the former situation the moderator is 1 mean free path long, while in the latter is 5 mean free paths long.

Central Limit Theorem (CLT)

Being the number of operations huge, a parallel processing is needed, but in this case the threads do not share the random walks, but entire trials, whose mutual independency is guaranteed by the input, i.e. the seeds: there are as many seeds as trials and each thread pick up a fraction of them. Other inputs are kept constant with the *partial* function applied to the Monte Carlo simulations executor. The number of particles per cycles are 10000 and detectors 100.

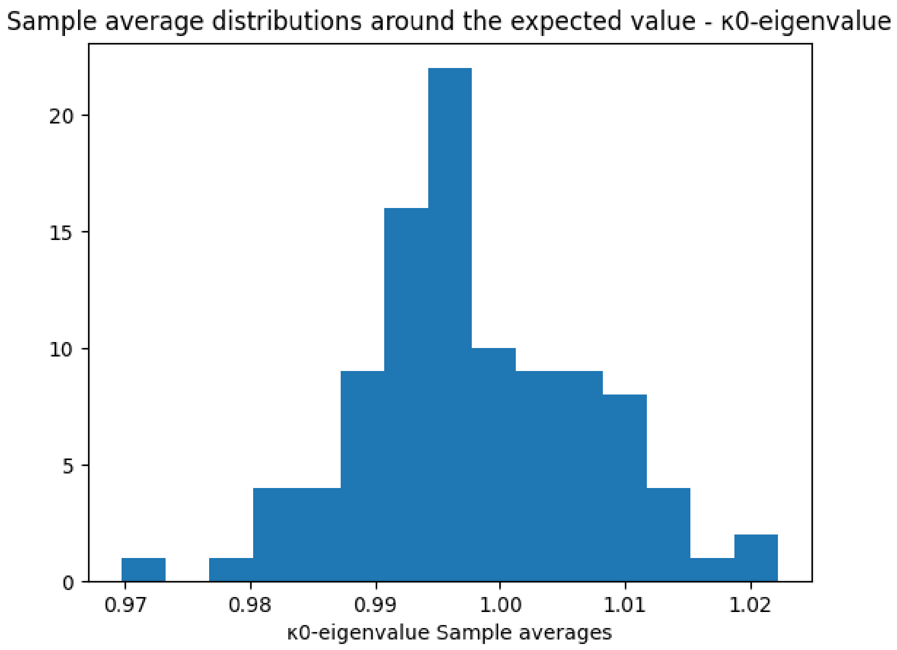


Figure 5.23: κ_0 CLT test $\rightarrow$ Trials: 100; Histories: 2.

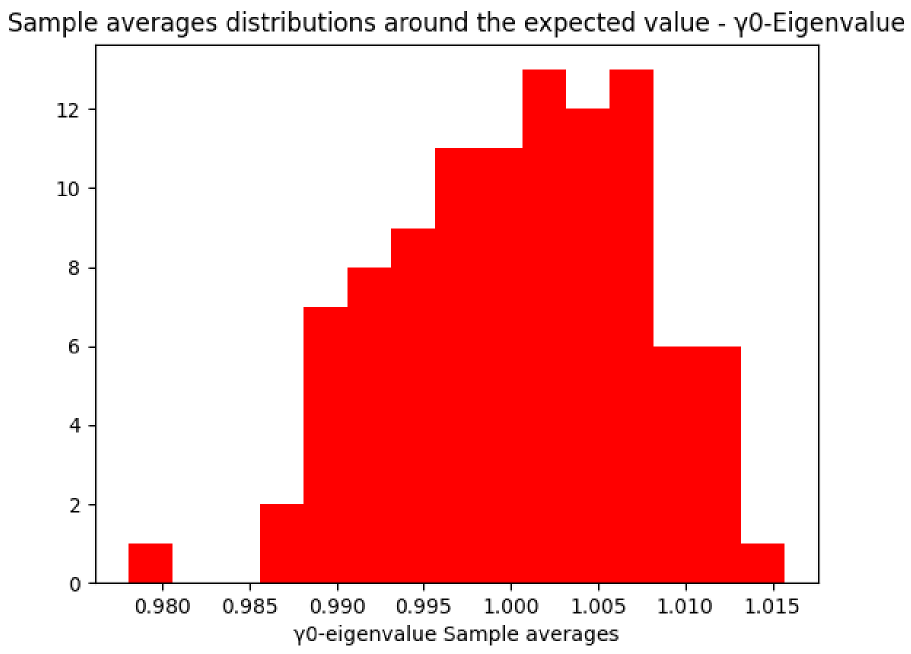


Figure 5.24: γ_0 CLT test $\rightarrow$ Trials: 100; Histories: 2.

The low number of experiments per trial (i.e. the active cycles) makes the variance of the distribution particularly high, with a significant shift of the maximum away from the true mean. The shape itself is completely different from the normal one, both in Fig.:5.23 and in Fig.:5.24.

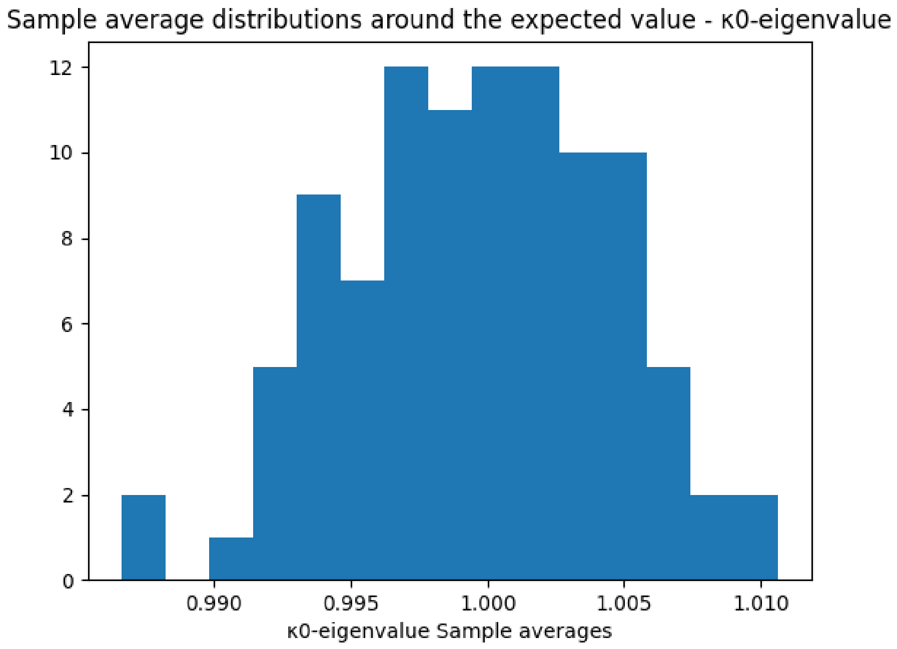


Figure 5.25: κ_0 CLT test $\rightarrow$ Trials: 100; Histories: 10.

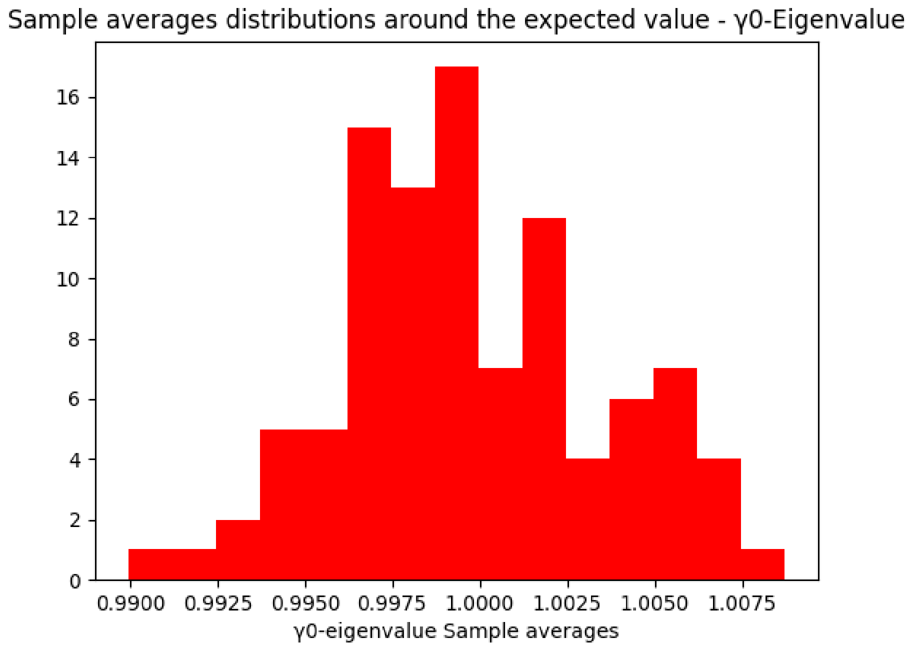


Figure 5.26: γ_0 CLT test $\rightarrow$ Trials: 100; Histories: 10.

The increase of the number of experiments per trial is beneficial and apparent in Figs.:5.25-5.26: the maximum of the distribution is closer to the true mean for both the eigenvalues and the variance is reduced.

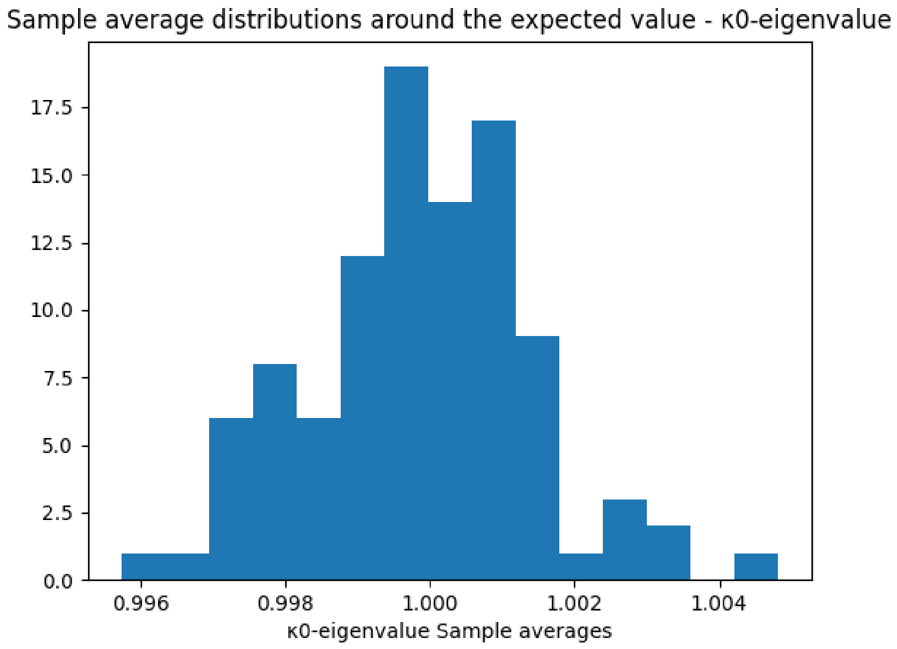


Figure 5.27: κ_0 CLT test $\rightarrow$ Trials: 100; Histories: 100.

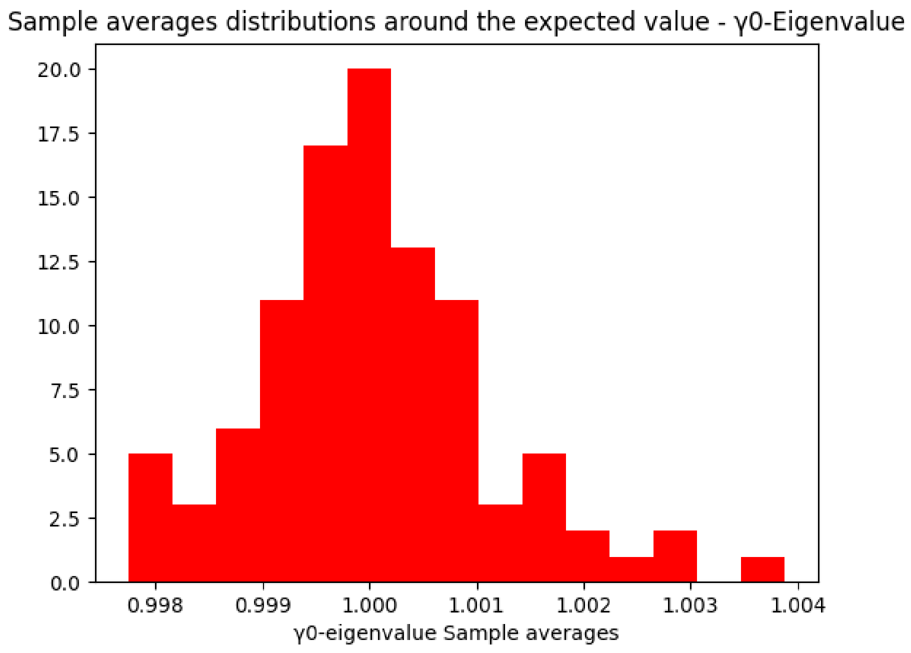


Figure 5.28: γ_0 CLT test $\rightarrow$ Trials: 100; Histories: 100.

With such a number of active cycles, the precision is furtherly improved, and the two distributions of Fig.:5.27-5.28 are more similar to the Gaussian curves.

κ_0	$1-\sigma$	$2-\sigma$	$3-\sigma$
2 histories	0.42	0.65	0.78
10 histories	0.59	0.87	0.98
100 histories	0.66	0.92	0.99

Table 5.1: Confidence levels of κ_0 for experiments with 100 trials

γ_0	$1-\sigma$	$2-\sigma$	$3-\sigma$
2 histories	0.36	0.63	0.75
10 histories	0.61	0.92	0.97
100 histories	0.68	0.95	0.98

Table 5.2: Confidence levels of γ_0 for experiments with 100 trials

The tables above (4.7-4.8) sum up the approaching of the confidence levels of the previous distributions to the typical values of a normal one (Chapter 4). Running many independent trials is useful in eigenvalue convergence study [16].

Single trial

By drawing lessons about the source choice from the previous sections, the Monte Carlo method is now tested on single trials. κ_0 computations have a population per cycle of 12000, a tolerance for the error on the S.E. sample mean of 0.0005 and 200 active cycles as maximum. There are 25 detectors (obviously only in the central layer). The charts of κ_0's statistics come first.

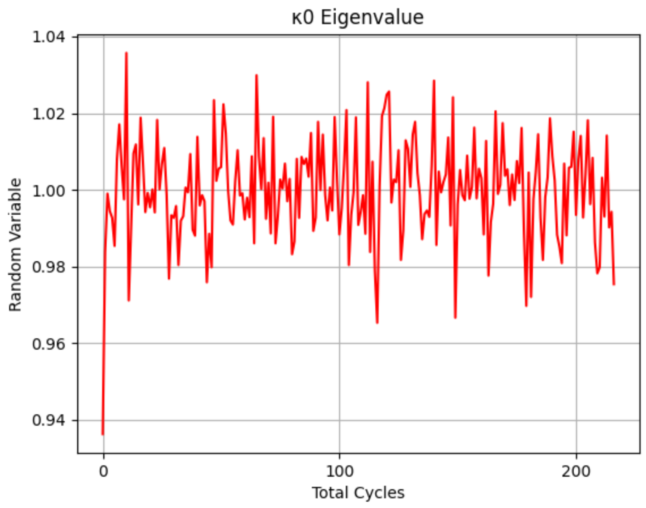


Figure 5.29: κ_0-eigenvalue: Sample

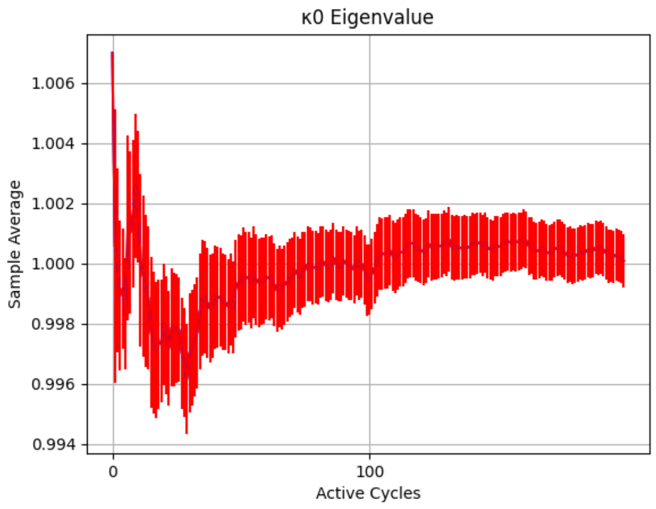


Figure 5.30: κ_0-eigenvalue: Evolution of Sample Average

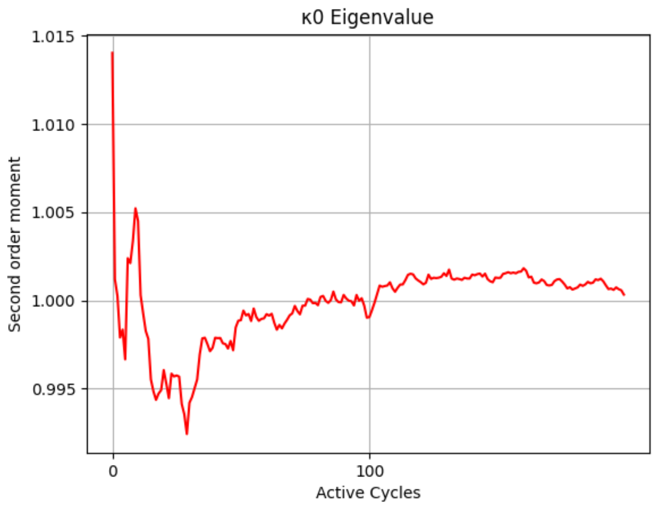


Figure 5.31: κ_0-eigenvalue: Evolution of Second Order Moment of Sample

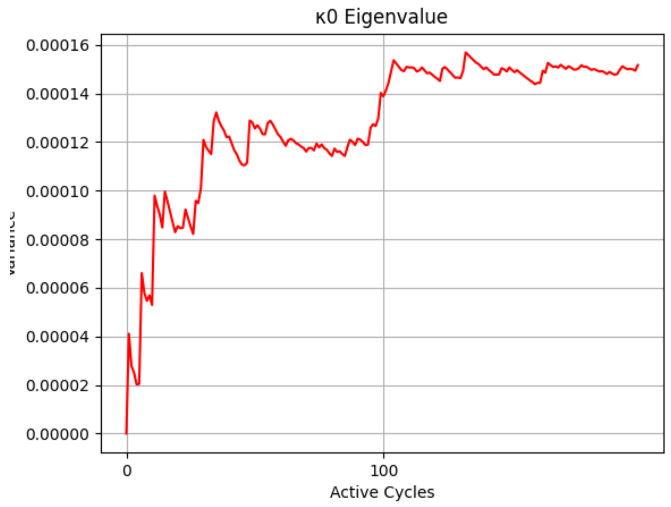


Figure 5.32: κ_0-eigenvalue: Evolution of Variance of the Sample

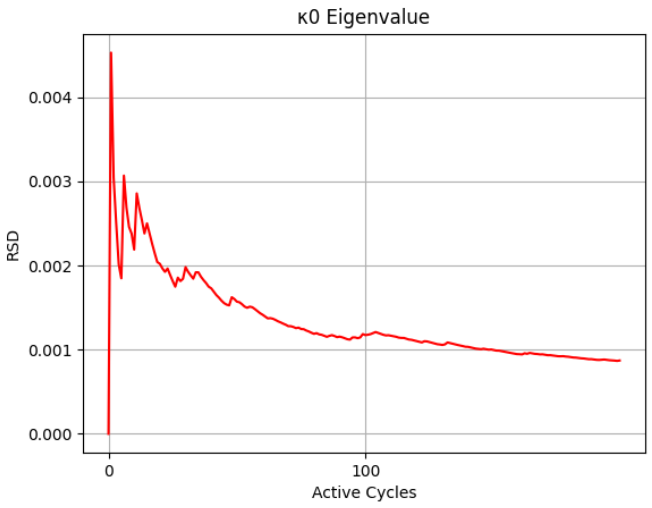


Figure 5.33: κ_0-eigenvalue: Evolution of Relative Standard Deviation of the Sample Average

Now it is the turn of κ_0 S.E.'s statistics graphs.

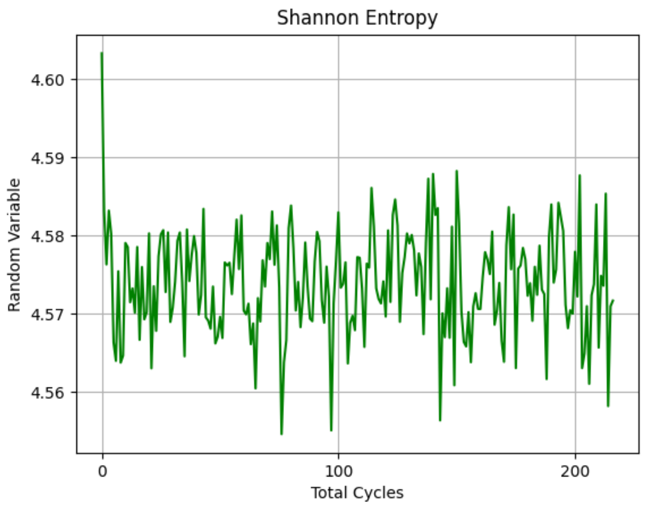


Figure 5.34: κ_0-eigenvalue's S.E.: Sample

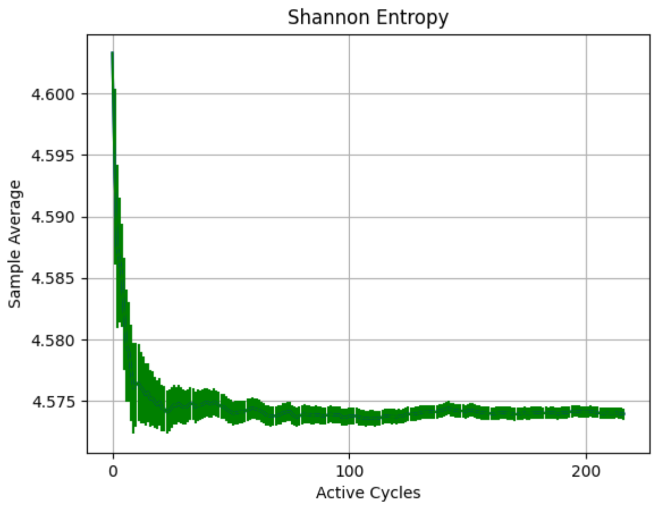


Figure 5.35: κ_0-eigenvalue's S.E.: Evolution of Sample Average

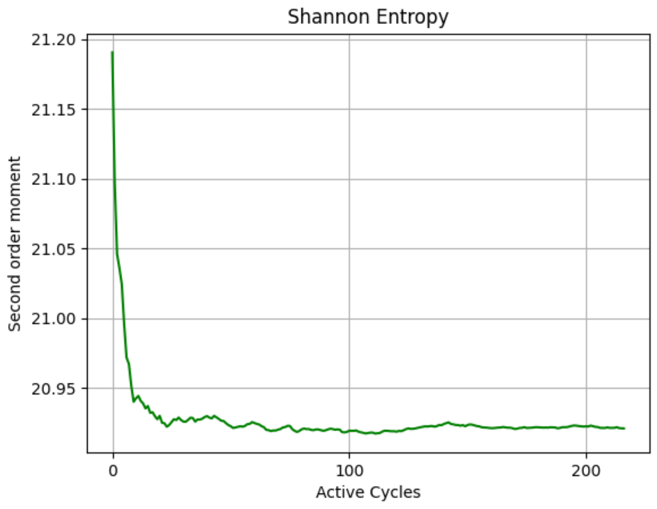


Figure 5.36: κ_0-eigenvalue's S.E.: Evolution of Second Order Moment of Sample

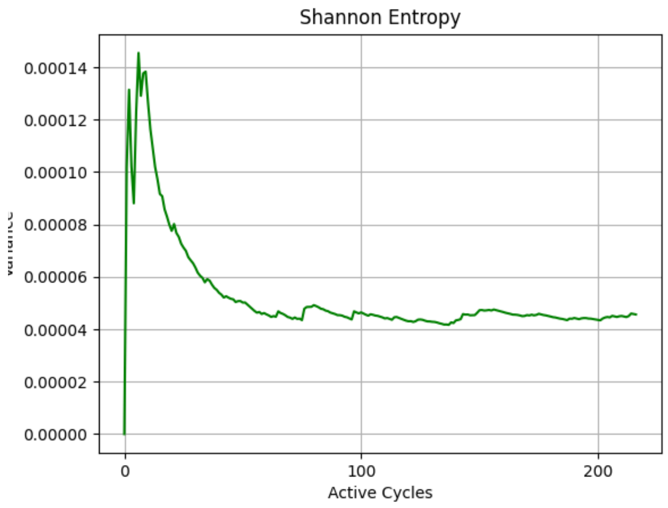


Figure 5.37: κ_0-eigenvalue's S.E.: Evolution of Variance of the Sample

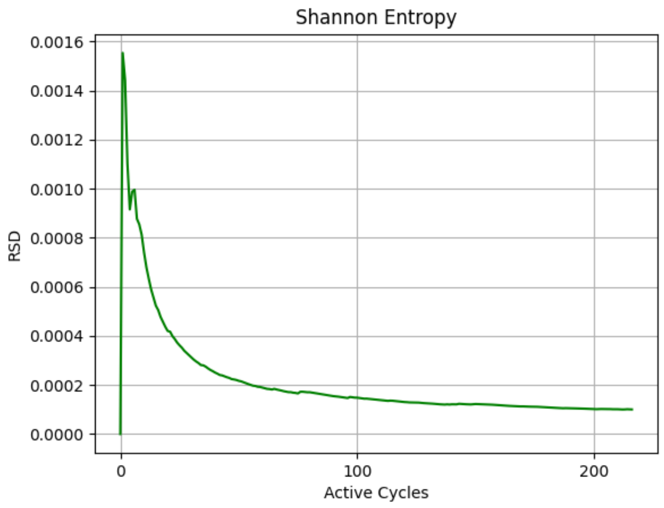


Figure 5.38: κ_0-eigenvalue's S.E.: Evolution of Relative Standard Deviation of the Sample Average

The final result is $\kappa_0 = 1.00009 \pm 0.00087$. The two samples (Figs.5.29-5.34) are initially far from their respective oscillation ranges (perhaps the

length of the moderator is such that leakages are not hard to occur), but they soon reach them. Figs.5.30-5.31 (Sample Average and Second Order Moment) show almost identical trends due to the fact that the second power of values that are close to 1 are close to 1 themselves. On the contrary, the corresponding charts of S.E. (Figs.5.35-5.36), although they are similar between each other, have different scales.
The evolution of variance (Fig.:5.32) stops having an increasing trend after about 100 active cycles, i.e. when the extrema of the random variable have already been explored. On the other hand, Fig.:5.37 shows a decreasing behaviour for S.E.'s variance evolution: since the maximum value of random variable is reached at the first generation, all the other values are smaller.
The evolutions of relative standard deviations in (Figs.5.33-5.38) are characterized by strong initial oscillations, particularly the one of κ_0. The trend of both the graphs, however, is quite 'loyal' to the inverse of the square root of the active cycles. As a consequence, the error bar widths decrease cycle after cycle (Fig.:5.30).
γ_0 computations have a population per cycle of 10000, a tolerance for the error on the S.E. sample mean of 0.00035 and 200 active cycles as maximum. There are 25 detectors in each layer (50 globally because there are only two layers thanks to symmetry of the slab). The charts of γ_0's statistics come first.

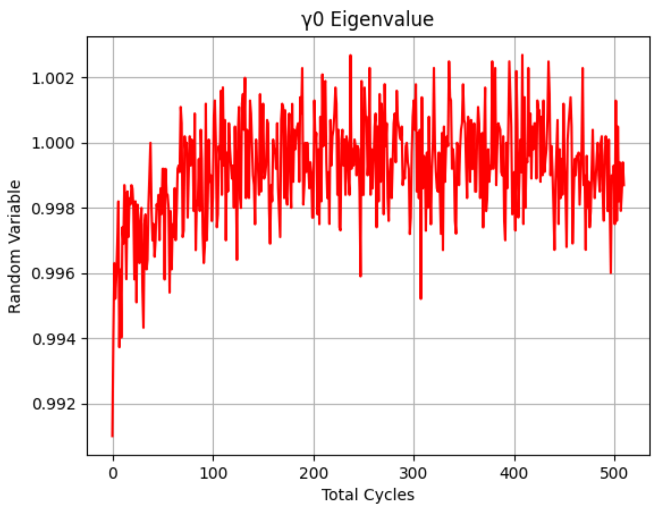


Figure 5.39: γ_0-eigenvalue: Sample

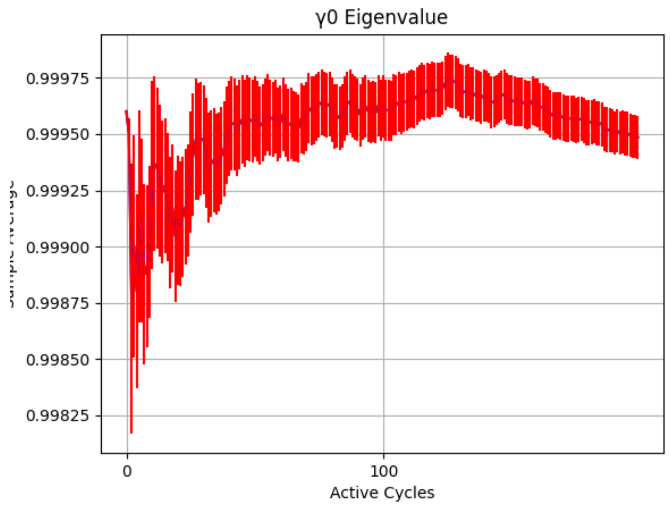


Figure 5.40: γ_0-eigenvalue: Evolution of Sample Average

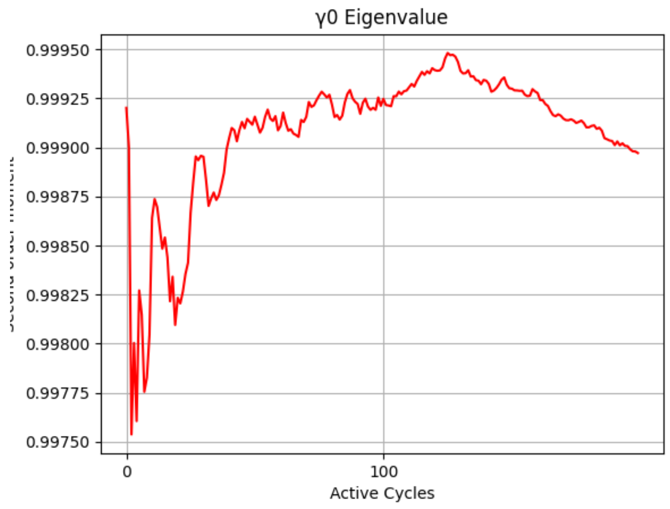


Figure 5.41: γ_0-eigenvalue: Evolution of Second Order Moment of Sample

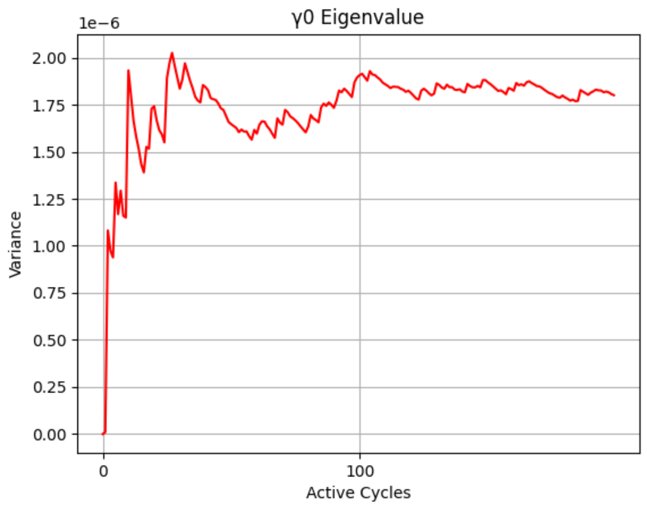


Figure 5.42: γ_0-eigenvalue: Evolution of Variance of the Sample

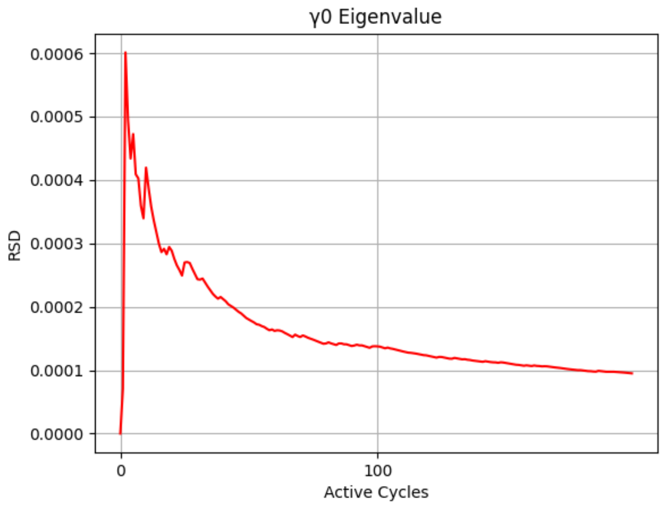


Figure 5.43: γ_0-eigenvalue: Evolution of Relative Standard Deviation of the Sample Average

Now it is the turn of γ_0 S.E.'s statistics graphs.

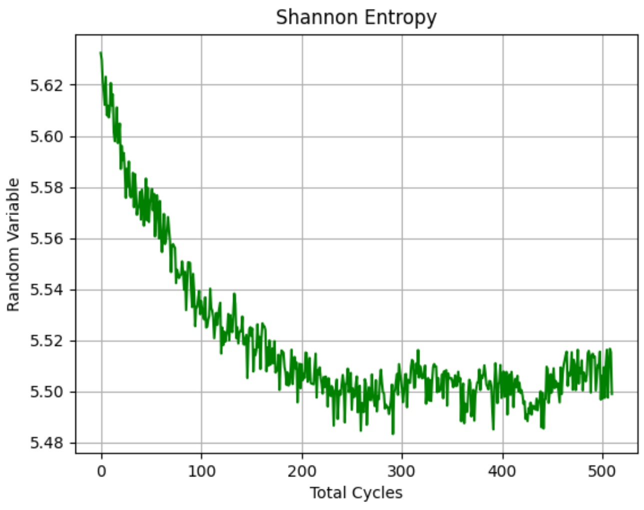


Figure 5.44: γ_0-eigenvalue's S.E.: Sample

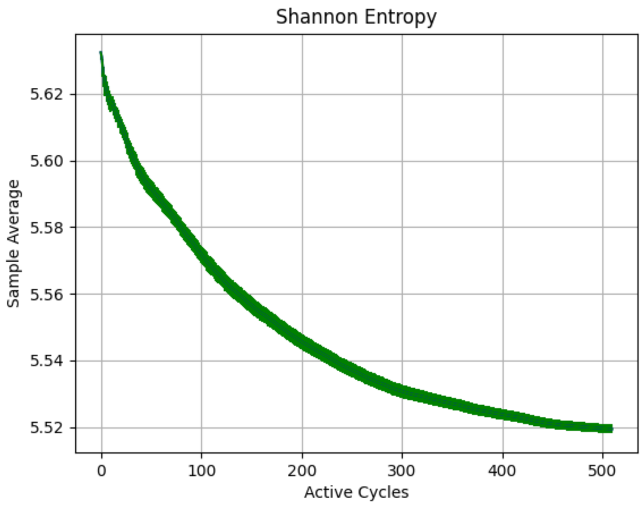


Figure 5.45: γ_0-eigenvalue's S.E.: Evolution of Sample Average

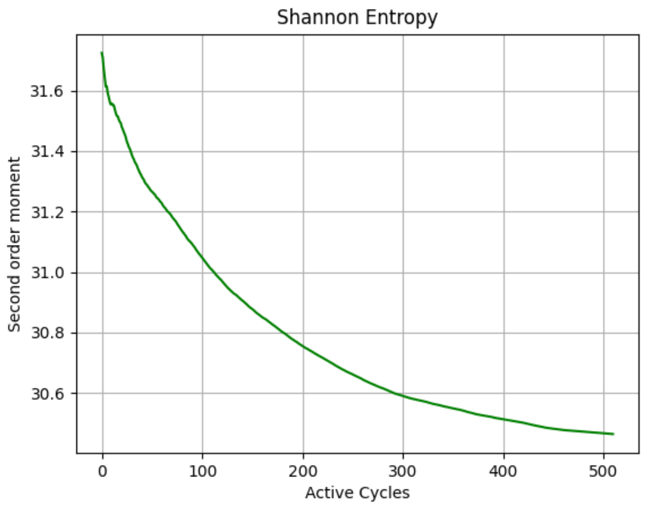


Figure 5.46: γ_0-eigenvalue's S.E.: Evolution of Second Order Moment of Sample

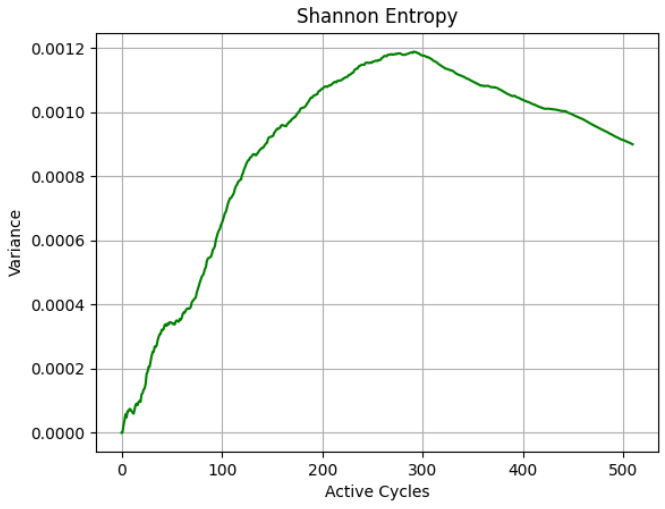


Figure 5.47: γ_0-eigenvalue's S.E.: Evolution of Variance of the Sample

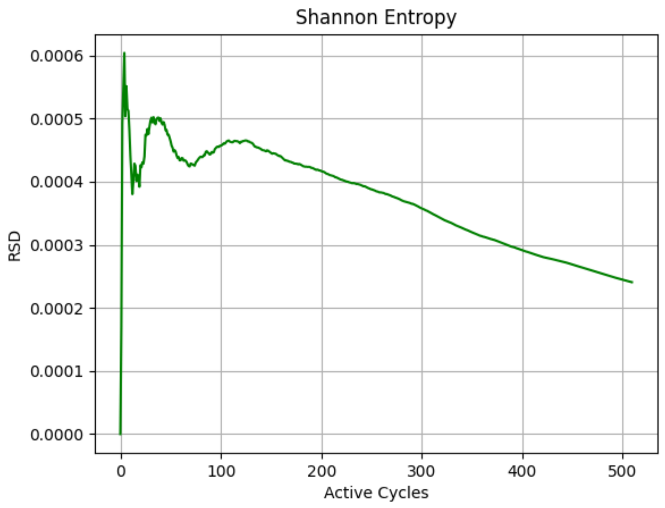


Figure 5.48: γ_0-eigenvalue's S.E.: Evolution of Relative Standard Deviation of the Sample Average

The final result is $\gamma_0 = 0.99948 \pm 0.000094$. The charts dedicated to quantities related to γ_0 (Figs.:5.39-5.40-5.41-5.42-5.43) denote similar trends to the ones of κ_0: skipping the inactive cycles eliminates the transient before the stationarity oscillation range. This cannot be done for the S.E., and its initial overestimation affects each evolution charts (Figs.:5.44-5.45-5.46-5.47-5.48). Both γ_0 and its S.E., as already explored in previous sections, have more difficulties in converging, so the statistics graphs, in particular the ones of the S.E., show a very different trend. At first, γ_0's S.E. error tolerance was set to 0.0005 as for κ_0 but it was not enough to reach the stationarity range for γ_0 (5.39), that starts after about the 200-th generation; once again the seed setting has demonstrated its utility allowing to 'adjust the shot'.

It is worth noting that γ_0's Shannon Entropies plotted in Section 5.1.2 had growing trends (Fig.:5.8) or already stable and quite close to the stationarity oscillation range (Figs.:5.11-5.12 and others); but the one of Fig.:5.44 decreases until the convergence is reached. The reason stays behind the quality of the initial source guess: it might be the one with a greater neutron concentration (i.e. 'Fission-source-like') which underestimate the real fundamental eigenmode distribution, or the one which equally distributes initial neutrons throughout the slab (i.e. 'Uniform') and consequently overestimates the S.E.. When the 'Uniform' distribution was studied in Section 5.1.2, γ_0's S.E. had initial values quite close to the 'stationarity range of oscillations'. Therefore, it was a good guess for the eigenmode distribution: but in those cases the slab had layers with equal length (2 cm) and very thin moderator material in terms of mean free path. So, the distribution of source between fuel and moderator regions was quite balanced and thus, closer to 'Uniform' as starting situation rather than 'Fission-source-like'. The following figures depict the same experiment for γ_0, with same data, but a 'Fission-source-like' source instead of 'Uniform' in order to compare them with Figs:5.39-5.40-5.44. Another difference is the augmented number of detectors per layer (100) and of active cycles (250).

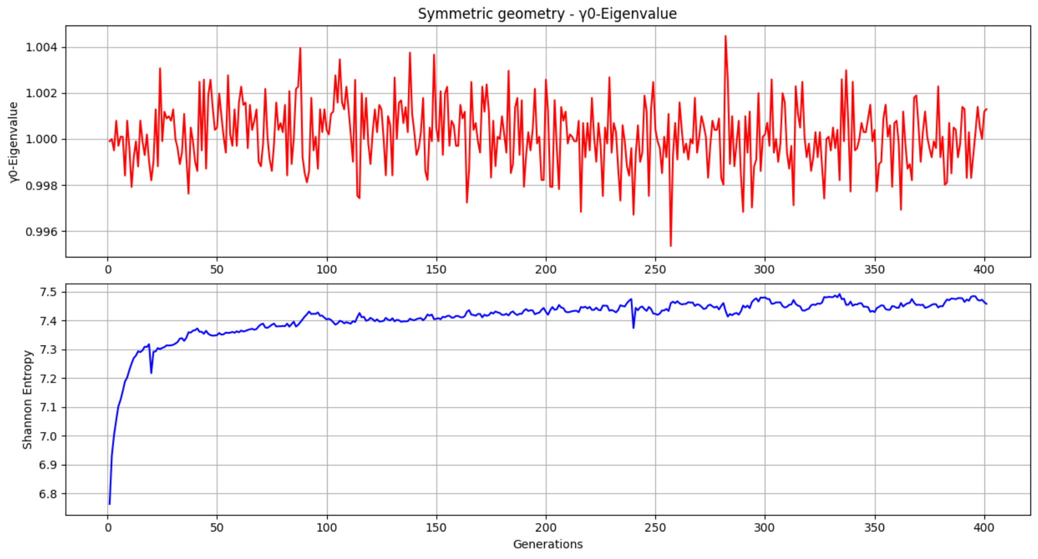


Figure 5.49: γ_0-eigenvalue and its S.E.: Sample-'Fission-source-like' initial distribution

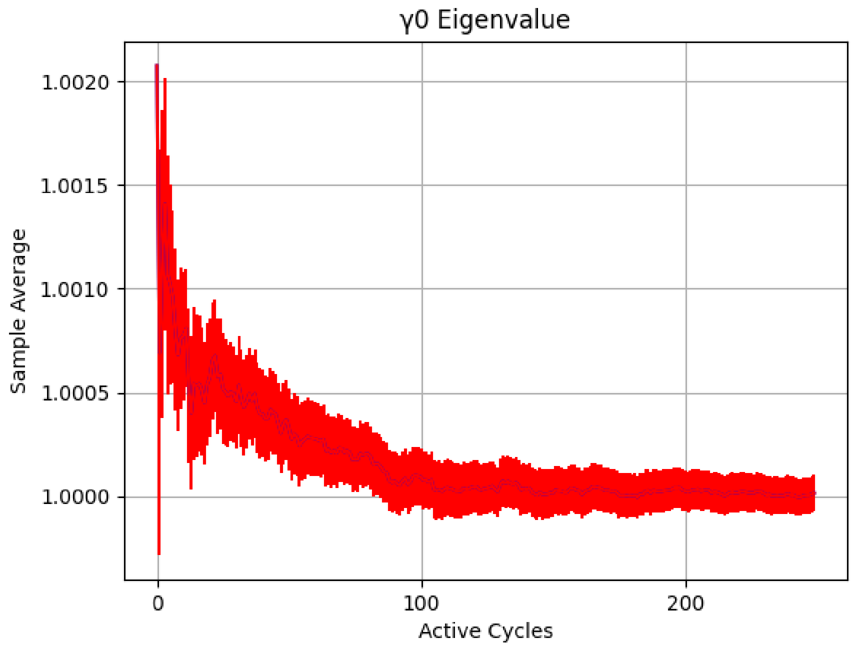


Figure 5.50: γ_0-eigenvalue: Evolution of the Sample Average of 5.49

The result is $\gamma_0 = 1.0000139 \pm 0.000087$. In this situation, both 'Uniform' and 'Fission-source-like' are away from the stationary range of oscillations, but the slab is different from the one of Section 5.1.2: fuel layer is prominent in length and moderator layers are of the order of 5 mean free paths: so, neutron source are rarefied in outer regions and denser toward the materials' interface. Hence, the choice of 'Fission-source-like' distribution for starting neutrons now is 'less' wrong than it was in Fig.(5.8). In Fig.:5.49, the distance between the initial S.E. value and the stationarity one is larger than in Fig.:5.44 and with 'different' sign. On the contrary, the eigenvalue is in the 'correct' range from the beginning with respect to the one depicted in Fig.:5.39. The number of cycles before S.E. convergence is rather similar both in 'Fission-source-like' case and in 'Uniform' case (about 200 cycles) and the precision of both results are comparable with each other, as shown by the sample mean with its error bar Fig.:5.50.

Conclusions

After explaining the fundamental theory which this work is based on (Chapters 1-2-3) the digital device to put it into practice by writing the codes has been developed (Chapter 4), and finally, the outcomes of all the efforts made (Chapter 5).

The true motivation at the base of this thesis has been the need for possess a code able to provide a useful support to other codes for the computation of two criticality eigenvalues, particularly γ_0, whose computation via Monte Carlo simulation had never been explored.
This thesis has also demonstrated the feasibility of a criticality problem solution via Monte Carlo simulation without using a professional code (although there are overhead troubles in parallel processing) with quite good results by following a set of recommendations:

1. To avoid bias in eigenvalue:
    - Use 10000 or more neutrons per cycle (100000 only for full-core situations); more efficient parallel calculations.
    - Discard enough initial cycles.
    - Always check convergence of both the eigenvalue and the source distribution.

2. To help with convergence:
    - Take advantage of problem symmetry, if possible.
    - Choose a good source such as to cover all fissionable regions.
    - Run at least 100 active cycles to compute reliable statistics.

- Make multiple, independent runs.

The evolution of neutron source distribution over the generations turned out to differ according to which factor had to be computed, in particular with an heterogeneous medium: the presence of a moderator induces a slowdown of source convergence throughout the slab, already affected by a larger number of detectors to be filled up. These facts made γ_0 source converge worse than the other one, although the single cycle for the computation γ_0 takes less time. This is due to the shorter duration of the single random walk, which is strictly related to the smaller mean displacement of a particle per generation for γ_0 with respect to κ_0.
The two factors react differently for the same addition of medium (in terms of quantity and quality), with κ_0 being faster to grow up or to decline than γ_0. If the choice of the source distribution is quite trivial for κ_0, on the other hand the optimum for γ_0 is harder to see: it could be uniform in all the domain in case of slabs with similar layer thicknesses and moderator layers not so thick, or uniform only in fuel layers if the reactor has a prominent fuel layer length and thick moderator layers which do not allow neutrons to spread uniformly throughout the moderator itself.
Another important feature of the code developed in this work is its customizable nature from the point of view of the energy groups. It has been tested only with two energy groups, but it has a structure such as to accept every (integer) number of energy levels: the more they are, the closer to reality the problem will be.

Appendix A

In this section the scripts shared by both the algorithms are presented. If the line starts with '#', it is a comment which helps to understand the purpose of the lines below.

Dataset organizer

'#' is also the marker that indicates the key for the dictionary in the '.txt' file with the cross-sections.

```
import numpy as np
from os import path
from pathlib import Path

# Dictionary keys
#- Abs---> Absorption cross section
#- Fiss---> Fission cross section
#- Chit---> Energy spectrum (PDF to state in which group
# the fission neutron will be born)
#- Nubar---> Number of born neutrons per single fission
#- S0---> Scattering matrix

# fname ---> Name of the file
# nE : Number of energy groups

class Dataset_organizer:
    def __init__(self, fname, nE):

        # Organise the data-set as a dictionary (key-->value)
```

```
selfdic = self.__dict__
G = None
# Open the file
lines = open(fname).read().split('\n')

for il, line in enumerate(lines):
    # If the current line of the file begins
    # with '\#' the key of an item
    # of the dictionary is determined by
    # the first word after that symbol
    if line.startswith('#'):
        key = (line.split('#')[1]).strip()
        matrix = None

    elif line == '':
        continue

    else:
        # Every splitted element of the line from the file
        # is treated as a float type
        # and put in an array
        data = np.asarray([float(val) for val in line.split()])
        if G is None:
            G = len(data)

        if G != nE:
            raise OSError('Number of groups in line g
                                  is not consistent!', il)

        if key.startswith('S'):
            # multi-line data (scattering matrix)
            if matrix is None:
                matrix = np.asarray(data)
            else:
                matrix = np.c_[matrix, data]

            if matrix.shape == (G, G):
                selfdic[key] = matrix.T
            elif matrix.shape == (G, ):
                selfdic[key] = matrix

        else:
            # single-line data (not a scattering matrix)
            selfdic[key] = np.asarray(data)

# Total scattering cross section (for each energy group)
self.Xs_scat_tot = np.sum(self.S0, axis=1)

# Total cross section (for each energy group)
self.Xs_tot = self.Xs_scat_tot + self.Abs

# Capture cross section (for each energy group)
self.Xs_capt = self.Abs - self.Fiss

# Number of energy groups
self.n_gr = nE

# Matrix of probability of scattering
# (single element of the scattering matrix
# divided by the respective Total scattering cross section
# of the energy group---> PDF calculated along the the columns)
self.mat_prob_scat = np.zeros((nE,nE))

# Matrix of the cumulative distribution
# function (CDF) of the previous PDF
self.mat_cum_scat = np.zeros((nE,nE + 1))

# Matrix of the cumulative distribution
# function (CDF) of the energy spectrum
self.En_fiss_cum = np.zeros(nE + 1)

for j in range(nE):
    self.En_fiss_cum[j + 1] =
                        self.En_fiss_cum[j] + self.Chit[j]
    for k in range(nE):
        self.mat_prob_scat[j][k] =
                        self.S0[j][k]/self.Xs_scat_tot[j]
              self.mat_cum_scat[j][k+1] =
                        self.mat_cum_scat[j][k]
                      + self.mat_prob_scat[j][k]
```

Fission function

```
import numpy as np
from os import path
from pathlib import Path
import sys
sys.path.append('.')

# LEGEND
# double_abs.T[0] ---> Column of a matrix with the positions
# of the absorbed neutrons after a free flight
# double_abs.T[1] ---> Column of a matrix with the energy groups
# of the absorbed neutrons after a free flight
# slabobj ---> Domain definition object (with boundaries,
# material qualification, detector defiition, ...)
# flibobj ---> Data set of fuel material
# seed ---> seed of the pseudo-random generator

def fission_func(double_abs, slabobj, flibobj, seed):

    Fission_loc = []
    Fission_nu_per_site = []

    for ct in range(double_abs.shape[0]):

        po = double_abs[ct][0]
        gr = int(double_abs[ct][1])


        if slabobj.location(po) == 'Fuel':
            #Absorption occurred in the fuel region
            ran1 = seed.random()
            if ran1 <= flibobj.Fiss[gr]/flibobj.Abs[gr]:
            # Fission occurred
                Fission_loc = np.append(Fission_loc, po)

            ran2 = seed.random()
            if ran2 <= (flibobj.Nubar[gr]-np.floor(flibobj.Nubar[gr])):
                # How many fission neutrons are born ?
                Fission_nu_per_site = np.append(Fission_nu_per_site,
                                    int(np.floor(flibobj.Nubar[gr])) + 1)
            else:
                Fission_nu_per_site = np.append(Fission_nu_per_site,
                                   int(np.floor(flibobj.Nubar[gr])))

    Fiss_matrix = np.c_[Fission_loc, Fission_nu_per_site]

    return Fiss_matrix
```

Domain assembler

```
import numpy as np
import sys
sys.path.append('.')
import math

# LEGEND
# v_lay: vector with the material of each thickness
# v_thic: vector with the lengths of each layer
# n_det: number of detectors in each layer of material

class Domain_assembler:
    def __init__(self, v_lay, v_thic, n_det, seed):

        self.num_lay = len(v_lay)
        lt = len(v_thic)

        if self.num_lay != lt:
            raise OSError('Number of layers not consistent
                    with the associated number of thicknesses!')

        # Thickness of the slab
        self.length = float(np.sum(v_thic))

        # Symmetry axis
        self.xmid = self.length / 2

        # Vector with the material (FUEL or MODERATOR) of each layer
        self.layers = np.asarray(v_lay)

        # Vector with the length of each layer
        self.thicknesses = np.asarray(v_thic)

        # Matrix with the coordinates of layers' boundaries on each row
        self.m_ends = np.zeros((lt,2))

        # Vector with the coordinates of boundaries (inner and outer)
       # of the slab
        self.v_ends = np.zeros(lt+1)
        for end in range(1,lt+1):
            self.v_ends[end] = self.v_ends[end-1] + v_thic[end-1]

        for el in range(lt):
            for i in range(2):
                self.m_ends[el][i] = self.v_ends[i + el]

        self.det_per_region = n_det

        # Lengths of the single, generic detector (dx_v)
        # for each layer in order to have the same
        # number of detectors
        self.dx_v = self.thicknesses/self.det_per_region

        # Vector with the lengths of the FUEL layers
        self.th_vf = []

        # Matrix with the coordinates of FUEL layers'
```

```
  # boundaries on each row
  self.mat_lay_fuel = np.zeros(2)
  rf = 0
  for p in range(len(self.layers)):
      if self.layers[p] == 'Fuel':
          rf += 1
          self.th_vf = np.append(self.th_vf, self.thicknesses[p])
          self.mat_lay_fuel =
                  np.append(self.mat_lay_fuel, self.m_ends[p])
          self.mat_lay_fuel = self.mat_lay_fuel.reshape(rf+1, 2)

  self.mat_lay_fuel = np.delete(self.mat_lay_fuel, 0, 0)

  # Lengths of the single fuel detector (dx_vf)
  # for each layer in order to have the
  # same number of detectors
  self.dx_vf = self.th_vf/self.det_per_region

  # Number of fuel layers
  self.fuel_layers = rf

  # Matrix with all the detectors: the row states the layer,
  # the column the detector
  # inside the specific layer. The elements will be
  # the fission neutrons born in each detector (source)
  rr = self.num_lay
  cc = self.det_per_region
  self.det_matr_s = np.zeros((rr,cc))
  self.det_matr_f  = np.zeros((rr,cc))

  # Same of the above matrix but the elements
  # will be "normalized" neutrons (in order to keep
  # constant the population for the next computations)
  self.norm_matr_t = np.zeros((rr,cc))

# Vector with the coordinates of boundaries
# (inner and outer) of the slab,
# including the symmetry axis
```

```
 self.v_ends_m = self.v_ends
 for zzz in range(1,len(self.v_ends)):
     if (self.xmid > self.v_ends[zzz-1])
                         and (self.xmid <= self.v_ends[zzz]):
         self.v_ends_m =
                 np.insert(self.v_ends, zzz, self.xmid, axis=0)
         # Index of symmetry axis in v_ends_m
         self.imid = zzz

 if self.v_ends_m[self.imid] ==
             self.v_ends_m[self.imid+1]:
    self.v_ends_m =
            np.delete(self.v_ends_m, self.imid+1, axis=0)

# Eliminate the parts of vectors/matrices previously
# created that are useless for symmetric case
 self.v_ends_s = self.v_ends_m

 for ui in range(len(self.v_ends_m) - self.imid - 1):
     self.v_ends_s =
          np.delete(self.v_ends_s, self.imid + 1, axis=0)

 self.m_ends_s = np.zeros((len(self.v_ends_s) - 1,2))

 for el in range(len(self.v_ends_s) - 1):
     for i in range(2):
         self.m_ends_s[el][i] = self.v_ends_s[i + el]

 stndrd = 0

 for bn in range(len(self.v_ends)):
     if self.v_ends[bn] == self.xmid:
         stndrd = 1

# Vector with the length of each layer including the
# two extra-layers defined by the symmetry axis
 if stndrd == 0:
     self.thicknesses_m = np.empty(1)
```

```
        for hhh in range(len(self.v_ends_m) - 1):
            if (hhh == self.imid):
                self.thicknesses_m =
                  np.append(self.thicknesses_m,
                       self.thicknesses[hhh - 1]/2)
                self.i_thic = hhh - 1
            if (hhh == (self.imid - 1)):
                self.thicknesses_m =
                  np.append(self.thicknesses_m,
                    self.thicknesses[hhh]/2)
            if (hhh < (self.imid - 1)):
                self.thicknesses_m =
                    np.append(self.thicknesses_m,
                              self.thicknesses[hhh])
            if (hhh > (self.imid)):
                self.thicknesses_m =
                       np.append(self.thicknesses_m,
                                        self.thicknesses[hhh - 1])

        self.thicknesses_m =
                    np.delete(self.thicknesses_m, 0 ,axis=0)
    else:
        self.thicknesses_m = self.thicknesses
        self.i_thic = self.imid - 1

    self.thicknesses_s = self.thicknesses_m

    for uj in range(int(len(self.thicknesses_m)/2)):
        self.thicknesses_s =
           np.delete(self.thicknesses_s, self.i_thic + 1, axis=0)

    # Vector with the lengths of the FUEL
    # layers (symmetric case)
    self.th_vf_s = []

   # Matrix with the coordinates of FUEL layers' boundaries
   # on each row (symmetric case)
    self.mat_lay_fuel_s = np.zeros(2)
```

```
        rfs = 0
        for p in range(len(self.thicknesses_s)):
            if self.layers[p] == 'Fuel':
                rfs += 1
                self.th_vf_s =
                    np.append(self.th_vf_s, self.thicknesses_s[p])
                self.mat_lay_fuel_s =
                    np.append(self.mat_lay_fuel_s, self.m_ends_s[p])
                self.mat_lay_fuel_s =
                    self.mat_lay_fuel_s.reshape(rfs + 1, 2)

        self.mat_lay_fuel_s = np.delete(self.mat_lay_fuel_s, 0, 0)

        # Number of fuel layers (symmetric case)
        self.fuel_layers_s = rfs

      # Number of layers (symmetric case)
        self.num_lay_s = len(self.thicknesses_s)

      # Dimension of a fuel detector (symmetric case)
        self.dx_vf_s = self.th_vf_s/self.det_per_region
      # Dimension of a detector (symmetric case)
        self.dx_v_s = self.thicknesses_s / self.det_per_region

      # Create matrices with the same purpose
      # of the previous ones but for symmetric case
        self.det_matr_f_s =
                np.zeros((self.num_lay_s, self.det_per_region))
        self.det_matr_s_s =
                np.zeros((self.num_lay_s, self.det_per_region))
        self.norm_matr_t_s =
                np.zeros((self.num_lay_s, self.det_per_region))

# Function Location: INPUT—> coordinate
#                            OUTPUT—> material
# of the layer in which the coordinate is
def location(self, y):
    for wh in range(self.num_lay):
        if y >= self.m_ends[wh][0] and y <=
            self.m_ends[wh][1]:
            return self.layers[wh]

    if  y < 0 or y > self.length:
        return 'Out of the slab'

# Function boundarycounter: INPUT---> start point
#                           and end point of a free flight
#                           OUTPUT---> boundaries
#                           crossed during
#                           the free flight ordered from first
#                           to last encountered

def boundarycounter(self, y_in, y_fin):
    boundary_crossed = []

    for cc in range(len(self.v_ends)):
        if (self.v_ends[cc] >
             y_in and self.v_ends[cc] <
             y_fin) or (self.v_ends[cc] <
             y_in and self.v_ends[cc] > y_fin):

            boundary_crossed =
                    np.append(boundary_crossed,self.v_ends[cc])

        boundary_crossed_ordered =
                    np.zeros(len(boundary_crossed))

        if y_in < y_fin:
            boundary_crossed_ordered =
                          boundary_crossed.copy()
        else:
            boundary_crossed_ordered[:] =
                         np.flipud(boundary_crossed[:])

    return boundary_crossed_ordered
```

```
# Function issymmetric: it states if a slab is symmetric or not.
# It creates two couples of arrays: the first one
# is a sequence of '1' and '0' to indicate which
# material the layers are made of. The 2 elements of the couple
# (layers from the 2 halves) are compared to
# verify if both ones are specular in terms of composition.
# The second couple of arrays instead of '1' and '0' has
# the lengths of each layer in both halves to verify if both ones
# are specular in terms of dimension.
def issymmetric(self):

    flg1 = 0
    flg2 = 0

    materials_m = []
    for ttt in range(len(self.v_ends_m)-1):
        if self.location(self.v_ends_m[ttt]+1e-7) == 'Fuel':
            materials_m = np.append(materials_m, 1)
        if self.location(self.v_ends_m[ttt]+1e-7) == 'Moderator':
            materials_m = np.append(materials_m, 0)

    m1n = materials_m[0:self.imid]
    m2n = materials_m[self.imid:len(materials_m)]

    m1m = self.thicknesses_m[0:self.imid]
    m2m = self.thicknesses_m[self.imid:len(materials_m)]
    if len(m1n) != len(m2n):
        answer = False
    else:
        for yyy in range(len(m1n)):
            if m1n[yyy] == np.flipud(m2n)[yyy]:
                flg1 += 1

        for vvv in range(len(m1m)):
            if m1m[vvv] == np.flipud(m2m)[vvv]:
                flg2 += 1
            if flg1 == len(m1n) and flg2 == len(m1m):
                answer = True
            else:
                answer= False

        return answer

    # Function boundary_counter_s: same purpose of the
    # boundary_counter function but modified for symmetric cases
    def boundarycounter_s(self, y_in, y_fin):
        boundary_crossed = []

        for cc in range(len(self.v_ends_s)):
            if (self.v_ends_s[cc] > y_in and self.v_ends_s[cc] < y_fin)
                   (self.v_ends_s[cc] <
               y_in and self.v_ends_s[cc] > y_fin):

                boundary_crossed =
                        np.append(boundary_crossed, self.v_ends_s[cc])

            boundary_crossed_ordered =
                        np.zeros(len(boundary_crossed))

            if y_in < y_fin:
                boundary_crossed_ordered =
                        boundary_crossed.copy()
            else:
                boundary_crossed_ordered[:] =
                        np.flipud(boundary_crossed[:])

        return boundary_crossed_ordered
```

Initial source setter

```
import numpy as np
from os import path
from pathlib import Path
```

```
import sys
sys.path.append('.')

# LEGEND
# flibobj ---> Data set of fuel material
# slabobj ---> Domain definition
# object (with boundaries, material qualification, detector definition,
# Neo ---> Amount of neutrons of the 0-th population
# seed ---> seed of the pseudo-random generator
# If the geometry is Symmetric, the second
# half of the slab can be neglected.
# If the geometry is Point-source-like,
# neutrons positions are at the geometric centre.
# If the distribution is Fission-source-like, neutrons start the
# flight from fuel regions with energy group given from the proper CDF.
# If the distribution is Uniform, neutrons are mono-energetic
# (2nd group, if Uniform_0 1st group) and start the flight
# both in fuel and moderator regions.
# If the direction is Inward, the neutrons will start
# the flight towards the centre of the slab
# (useful with thin layer of fuel).

class Initial_source_setter:
    def __init__(self, flibobj, slabobj, Ne0, seed, geometry,
                     distribution, direction):

        if geometry == 'Symmetric' and distribution ==
                 'Fission-source-like' and direction == 'Inward':
            # How many neutrons start from each slot?

            self.neut_per_slot = int(
                np.ceil(Ne0 / (slabobj.fuel_layers_s *
                slabobj.det_per_fuel_region)))

            self.start_positions = []
            self.start_energies = []

            # Positions (Abscissas) of the 0-th generation
        # inside each slot

        for ii in range(slabobj.fuel_layers_s):
            for jj in range(slabobj.det_per_fuel_region):

                self.start_positions =
                np.append(self.start_positions,
                seed.uniform(slabobj.mat_lay_fuel_s[ii][0] +
                jj*slabobj.dx_vf_s[ii],
                slabobj.mat_lay_fuel_s[ii][0] +
                (jj+1)*slabobj.dx_vf_s[ii], self.neut_per_slot))

        # Energy groups of the 0-th generation inside each slot
        for tt in range(len(self.start_positions)):
            self.start_energies =
            np.append(self.start_energies,
            np.argmax(
            np.where(flibobj.En_fiss_cum - seed.random() < 0)))

        #Initial directions of the 0-th generation inside each slot
        self.start_directions = np.ones(len(self.start_energies))

    if geometry == 'Symmetric' and distribution ==
       'Fission-source-like' and direction == 'Stochastic':

        # How many neutrons start from each slot?
        self.neut_per_slot = int(
            np.ceil(
            Ne0 /
            (slabobj.fuel_layers_s * slabobj.det_per_region)))

        self.start_positions = []
        self.start_energies = []

        # Positions (Abscissas) of the 0-th generation inside
        # each slot
        for ii in range(slabobj.fuel_layers_s):
            for jj in range(slabobj.det_per_region):
                self.start_positions = np.append(
                self.start_positions,
                seed.uniform(slabobj.mat_lay_fuel_s[ii][0] +
                jj*slabobj.dx_vf_s[ii],
                slabobj.mat_lay_fuel_s[ii][0] +
                (jj+1)*slabobj.dx_vf_s[ii], self.neut_per_slot))

        # Energy groups of the 0-th generation inside each slot
        for tt in range(len(self.start_positions)):
            self.start_energies = np.append(
            self.start_energies, np.argmax(
            np.where(flibobj.En_fiss_cum - seed.random() < 0)))

        # Initial directions of
        # the 0-th generation inside each slot
        self.start_directions = seed.uniform(-1,
                     1, len(self.start_energies))

    if geometry == 'Symmetric' and distribution ==
                            'Uniform' and direction == 'Inward':

        # How many neutrons start from each slot?

        self.neut_per_slot = int(
        np.ceil(
        Ne0 /
        (slabobj.num_lay_s * slabobj.det_per_region)))

        self.start_positions = []

        # Positions (Abscissas) of the 0-th
        # generation inside each slot

        for ii in range(slabobj.num_lay_s):
            for jj in range(slabobj.det_per_fuel_region):

                self.start_positions =
                    np.append(self.start_positions,
                    seed.uniform(slabobj.m_ends_s[ii][0] +
                    jj*slabobj.dx_v_s[ii],
                    slabobj.m_ends_s[ii][0] +
                    (jj+1)*slabobj.dx_v_s[ii],
                    self.neut_per_slot))

        # Energy groups of the 0-th generation inside each slot
        self.start_energies = np.ones(len(self.start_positions))

        #Initial directions of the 0-th generation inside each slot
        self.start_directions = np.ones(len(self.start_energies))

    if geometry == 'Symmetric' and distribution ==
                      'Uniform' and direction == 'Stochastic':

        # How many neutrons start from each slot?

        self.neut_per_slot = int(
            np.ceil(
            Ne0 /
           (slabobj.num_lay_s * slabobj.det_per_region)))

        self.start_positions = []

        # Positions (Abscissas) of the
       # 0-th generation inside each slot
        self.start_positions = np.linspace(0, slabobj.xmid, Ne0)

        # Energy groups of the 0-th generation inside each slot
        self.start_energies = np.ones(len(self.start_positions))

        #Initial directions of the 0-th generation inside each slot
        self.start_directions =
        seed.uniform(-1,
        1, len(self.start_energies))
```

```
if geometry == 'Non-symmetric' and distribution ==
   'Fission-source-like' and direction == 'Inward':
    # How many neutrons start from each slot?

    self.neut_per_slot =
        int(
        np.ceil(Ne0 /
        (slabobj.fuel_layers *
        slabobj.det_per_fuel_region)))

    self.start_positions = []
    self.start_energies = []
    self.start_directions = []

    # Positions (Abscissas) of the
    # 0-th generation inside each slot

    for ii in range(slabobj.fuel_layers):
        for jj in range(slabobj.det_per_fuel_region):

            self.start_positions =
               np.append(self.start_positions,
               seed.uniform(slabobj.mat_lay_fuel[ii][0] +
               jj*slabobj.dx_vf[ii],
               slabobj.mat_lay_fuel[ii][0] +
               (jj+1)*slabobj.dx_vf[ii], self.neut_per_slot))

    # Energy groups of the 0-th generation inside each slot
    # Initial directions of the 0-th generation
    # inside each slot
    for tt in range(len(self.start_positions)):
        self.start_energies =
        np.append(self.start_energies,
        np.argmax(
        np.where(
        flibobj.En_fiss_cum - seed.random() < 0)))
        if tt <= int(np.ceil(len(self.start_positions)/2)):
            self.start_directions = np.append(
                self.start_directions, 1.)
        else:
            self.start_directions = np.append(
                self.start_directions, -1.)

if geometry == 'Non-symmetric' and distribution ==
    'Fission-source-like' and direction == 'Stochastic':
    # How many neutrons start from each slot?

    self.neut_per_slot = int(
        np.ceil(Ne0 /
        (slabobj.fuel_layers *
        int(slabobj.det_per_region))))

    self.start_positions = []
    self.start_energies = []

    # Positions (Abscissas) of the
    # 0-th generation inside each slot

    for ii in range(slabobj.fuel_layers):
        for jj in range(int(slabobj.det_per_region)):

            self.start_positions =
                   np.append(
                   self.start_positions,
                   seed.uniform(
                   slabobj.mat_lay_fuel[ii][0] +
                   jj*slabobj.dx_vf[ii],
                   slabobj.mat_lay_fuel[ii][0] +
                   (jj+1)*slabobj.dx_vf[ii],
                   self.neut_per_slot))

    # Energy groups of the 0-th generation inside each slot
    for tt in range(len(self.start_positions)):
        self.start_energies = np.append(self.start_energies,
        np.argmax(
        np.where(flibobj.En_fiss_cum - seed.random() < 0)))

    #Initial directions of the 0-th generation inside each slot
    self.start_directions =
    seed.uniform(-1,
    1, len(self.start_energies))

if geometry == 'Non-symmetric' and distribution ==
                'Uniform' and direction == 'Inward':

    # How many neutrons start from each slot?

    self.neut_per_slot = int(
        np.ceil(Ne0 /
        (slabobj.num_lay *
        slabobj.det_per_region)))

    self.start_positions = []
    self.start_directions = []
    # Positions (Abscissas) of the
    # 0-th generation inside each slot
    self.start_positions = np.linspace(0,slabobj.length, Ne0)

    # Energy groups of the 0-th generation inside each slot
    self.start_energies = np.ones(len(self.start_positions))

    # Initial directions
    # of the 0-th generation inside each slot
    for fg in range(len(self.start_positions)):
        if fg <= int(np.ceil(len(self.start_positions)/2)):
            self.start_directions = np.append(
                self.start_directions, 1.)
        else:
            self.start_directions = np.append(
                self.start_directions, -1.)

if geometry == 'Non-symmetric' and distribution ==
 'Uniform_0' and direction == 'Stochastic':
    # How many neutrons start from each slot?

    self.neut_per_slot = int(
        np.ceil(
        Ne0 / (slabobj.num_lay *
        slabobj.det_per_region)))

    self.start_positions = []
    self.start_energies = []

    # Positions (Abscissas) of the
    # 0-th generation inside each slot

    self.start_positions = np.linspace(0,slabobj.length, Ne0)

    # Energy groups of the 0-th generation inside each slot
    self.start_energies = np.zeros(len(self.start_positions))

    #Initial directions of the
    # 0-th generation inside each slot
    self.start_directions =
    seed.uniform(-1,
    1, len(self.start_energies))

if geometry == 'Point-source-like' and distribution ==
  'Fission-source-like' and direction == 'Stochastic':

    # Positions (Abscissas) of the 0-th generation
    self.start_positions =
    seed.uniform(
    slabobj.xmid-1e-3,
    slabobj.xmid+1e-3, Ne0)

    self.start_energies = []
    # Energy groups of the 0-th generation inside each slot
    for tt in range(len(self.start_positions)):
        self.start_energies =
        np.append(
        self.start_energies, np.argmax(
        np.where(
        flibobj.En_fiss_cum - seed.random() < 0)))

    #Initial directions of the
    # 0-th generation inside each slot
    self.start_directions =
    seed.uniform(-1,
    1, len(self.start_energies))

if geometry == 'Point-source-like' and distribution ==
     'Uniform' and direction == 'Stochastic':

    # Positions (Abscissas) of the 0-th generation
    self.start_positions =
    seed.uniform(
    slabobj.xmid-1e-3,
    slabobj.xmid+1e-3, Ne0)

    # Energy groups of the 0-th generation inside each slot
    self.start_energies = np.ones(len(self.start_positions))

    #Initial directions of the 0-th generation
    # inside each slot
    self.start_directions =
    seed.uniform(-1,
    1, len(self.start_energies))

if geometry == 'Symmetric' and distribution ==
        'Uniform_0' and direction == 'Stochastic':

    # How many neutrons start from each slot?

    self.neut_per_slot = int(
        np.ceil(Ne0 /
        (slabobj.num_lay_s *
        slabobj.det_per_region)))
```

```
        self.start_positions = []

        # Positions (Abscissas) of the
        # 0-th generation inside each slot
        self.start_positions = np.linspace(0, slabobj.xmid, Ne0)


        # Energy groups of the
        # 0-th generation inside each slot
        self.start_energies = np.zeros(len(self.start_positions))

        #Initial directions of the 0-th generation
        # inside each slot
        self.start_directions =
        seed.uniform(-1,
        1, len(self.start_energies))

    if geometry == 'Non-symmetric' and distribution ==
            'Uniform' and direction == 'Stochastic':

        # How many neutrons start from each slot?

        self.neut_per_slot = int(
            np.ceil(Ne0 /
            (slabobj.num_lay *
            slabobj.det_per_region)))

        self.start_positions = []
        self.start_energies = []

        # Positions (Abscissas) of the
        # 0-th generation inside each slot

        self.start_positions = np.linspace(0, slabobj.length, Ne0)

        # Energy groups of the
        # 0-th generation inside each slot
```

```
            self.start_energies = np.ones(len(self.start_positions))

            #Initial diretions of the
            # 0-th generation inside each slot
            self.start_directions =
            seed.uniform(-1,
            1, len(self.start_energies))
```

Statistics Executor

```
import numpy as np
import sys
sys.path.append('.')
import math

# rv_v ---> Vector whose elements are
# the random variable at each cycle
# a_c ---> Number of active cycles
# i_c ---> Number of inactive cycles

class Statistics_executor:
    def __init__(self, rv_v, a_c, i_c, K_amp):

        # Number of total cycles
        t_c = a_c + i_c

        # Take only elements of the random
        # variable from active cycles
        vv = rv_v[i_c : t_c]
        # Vector with the update of the statistics
        # for the subsequent plot visualization
        self.MC_update = np.zeros((5,1))
        # Update Sample average
        self.MC_update[0] = np.sum(vv)/(a_c)
        # Update Second order moment
        # of the sample
        self.MC_update[1] = np.sum(vv**2)/(a_c)
        # Update Variance of the sample
        self.MC_update[2] = self.MC_update[1] - self.MC_update[0]**2
        # Update Relative standard deviation
        # of the sample mean
        self.MC_update[3] =
        math.sqrt(self.MC_update[2]/(a_c))/abs(self.MC_update[0])
        # Update Error bar for the sample mean
        self.MC_update[4] =
        self.MC_update[3] * abs(self.MC_update[0])
```

Results Visualizer

```
import numpy as np
import sys
sys.path.append('.')
import matplotlib.pyplot as plt

# LEGEND
# v_rc ---> Vector with the values of the
# random variable to be plotted
# v_sa ---> Vector with the values of the
# sample average to be plotted
# v_2o ---> Vector with the 2nd order moment
# of the sample average to be plotted
# v_va ---> Vector with the values of the
# variance to be plotted
# v_rsd ---> Vector with the values of the
# relative standard deviation to be plotted
# v_eb ---> Vector with the values of the
# of the error bar to be plotted
# i_c ---> Number of inactive cycles
# t_c ---> Number of total cycles
# name ---> Name of the quantity
# to be plotted

class Results_visualizer:
    def __init__(self, comv, v_rv, v_sa, v_2o, v_va,
         v_rsd, v_eb, i_c, t_c, name):

        xx = np.arange(len(v_sa))
        x = np.arange(t_c)
        if name == '\u03B30 Eigenvalue':
            plt.figure('RV_k')
            plt.plot(x, v_rv, 'r')
            ax = plt.figure('RV_k').gca()
            ax.set_xticks(np.arange(0, t_c, 100))
            plt.xlabel('Total Cycles')
            plt.ylabel('Random Variable')
            plt.title(name)
            plt.grid()

            plt.figure('SA_k')
            plt.plot(xx, v_sa, 'b')
            yerr = v_eb
            ax = plt.figure('SA_k').gca()
            ax.set_xticks(np.arange(0, t_c - i_c, 100))
            plt.errorbar(xx, v_sa, yerr = yerr, color = 'r')
            plt.xlabel('Active Cycles')
            plt.ylabel('Sample Average')
            plt.title(name)
            plt.grid()

            plt.figure('2nd_ord_mom_k')
            plt.plot(xx, v_2o, 'r')
            ax = plt.figure('2nd_ord_mom_k').gca()
            ax.set_xticks(np.arange(0, t_c - i_c, 100))
            plt.xlabel('Active Cycles')
            plt.ylabel('Second order moment')
            plt.title(name)
            plt.grid()

            plt.figure('VAR_k')
            plt.plot(xx, v_va, 'r')
            ax = plt.figure('VAR_k').gca()
            ax.set_xticks(np.arange(0, t_c - i_c, 100))
```

```
        plt.xlabel('Active Cycles')
        plt.ylabel('Variance')
        plt.title(name)
        plt.grid()

        plt.figure('RSD_k')
        plt.plot(xx, v_rsd, 'r')
        ax = plt.figure('RSD_k').gca()
        ax.set_xticks(np.arange(0, t_c - i_c, 100))
        plt.xlabel('Active Cycles')
        plt.ylabel('RSD')
        plt.title(name)
        plt.grid()
    else:
        plt.figure('RV_ks')
        plt.plot(x, v_rv, 'g')
        ax = plt.figure('RV_ks').gca()
        ax.set_xticks(np.arange(0, t_c, 100))
        plt.xlabel('Total Cycles')
        plt.ylabel('Random Variable')
        plt.title(name)
        plt.grid()

        plt.figure('SA_ks')
        plt.plot(xx, v_sa, 'b')
        yerr = v_eb
        ax = plt.figure('SA_ks').gca()
        ax.set_xticks(np.arange(0, t_c - i_c, 100))
        plt.errorbar(xx, v_sa, yerr = yerr, color = 'g')
        plt.xlabel('Active Cycles')
        plt.ylabel('Sample Average')
        plt.title(name)
        plt.grid()

        plt.figure('2nd_ord_mom_ks')
        plt.plot(xx, v_2o, 'g')
        ax = plt.figure('2nd_ord_mom_ks').gca()
        ax.set_xticks(np.arange(0, t_c - i_c, 100))
    plt.xlabel('Active Cycles')
    plt.ylabel('Second order moment')
    plt.title(name)
    plt.grid()

    plt.figure('VAR_ks')
    plt.plot(xx, v_va, 'g')
    ax = plt.figure('VAR_ks').gca()
    ax.set_xticks(np.arange(0, t_c - i_c, 100))
    plt.xlabel('Active Cycles')
    plt.ylabel('Variance')
    plt.title(name)
    plt.grid()

    plt.figure('RSD_ks')
    plt.plot(xx, v_rsd, 'g')
    ax = plt.figure('RSD_ks').gca()
    ax.set_xticks(np.arange(0, t_c - i_c, 100))
    plt.xlabel('Active Cycles')
    plt.ylabel('RSD')
    plt.title(name)
    plt.grid()

plt.show()
```

Appendix B

In this appendix, scripts exclusively for the computation κ_0 are presented. Also the random walk of 'Symmetry' case is taken into account.

Random walk - κ_0

```
import numpy as np
import sys
sys.path.append('.')
import math
from fermi import fission_func

# LEGEND
# triple.T[0] ---> Column of the array with the starting
# positions of the neutrons (always in a fuel layer)
# triple.T[1] ---> Column of the array with the starting
# energy group of the neutrons
# triple.T[2] ---> Column of the array with the starting
# directions of neutrons
# slabobj ---> Domain definition object (with boundaries,
# material qualification, detector definition, ...)
# flibobj ---> Data set of fuel material
# mlibobj ---> Data set of moderator/reflector material
# seed ---> seed of the pseudo-random generator



def Random_walk_k(triple, slabobj, flibobj, mlibobj, seed):
```

```
Abs_fuel_matrix = []

for fc in range(len(triple)):
    # starting abscissa, energy group and direction (cosine)
    x0 = triple[fc][0]
    E0 = int(triple[fc][1])
    mu = triple[fc][2]

    if (x0 < 0) or (x0 > slabobj.length):
        raise OSError('Out of the slab !')

    if slabobj.location(x0) == 'Fuel':
        # distance covered if it starts from a fuel region
        l = - math.log(1 - seed.random())/flibobj.Xs_tot[E0]
    else:
        # distance covered if it starts from a moderator region
        l = - math.log(1 - seed.random())/mlibobj.Xs_tot[E0]
    # flag setting
    alive = 1
    # Start the random walk
    while(alive):

        # new position in the slab
        x = x0 + l*mu

        # how many inner boarders did the neutron cross ?
        b_cross = slabobj.boundarycounter(x0, x)

        if (len(b_cross) == 0) and (slabobj.location(x0) ==
            'Fuel'):
            # the neutron has not crossed any inner
            # boundaries and is still in a fuel layer
            rho1 = seed.random()
            if rho1 <= flibobj.Abs[E0]/flibobj.Xs_tot[E0]:
                # absorption occurred
                end_point = x
                end_group = E0
                Abs_fuel_matrix = np.append(
            Abs_fuel_matrix, np.append(x,E0))
            alive = 0
        else:
            # scattering occurred: energy, cosine direction,
            # starting point abscissa
            # and distance covered change
            E  = np.argmax(
            np.where(
            flibobj.mat_cum_scat[E0] - seed.random() < 0))
            E0 = int(E)
            mu = seed.uniform(-1,1)
            x0 = x
            l =
              - math.log(1 - seed.random())/flibobj.Xs_tot[E0]
            continue


    if (len(b_cross) == 0) and (slabobj.location(x0) ==
        'Moderator'):
        # the neutron has not crossed any inner boundaries
        # and is still in a moderator layer
        rho2 = seed.random()
        if rho2 <= mlibobj.Abs[E0]/mlibobj.Xs_tot[E0]:
            # absorption occurred
            alive = 0
        else:
            # scattering occurred: energy, direction,
            # starting point
            # and distance covered change
            E  = np.argmax(
            np.where(
            mlibobj.mat_cum_scat[E0] - seed.random() < 0))
            E0 = int(E)
            mu = seed.uniform(-1,1)
            x0 = x
            l =
              - math.log(1 - seed.random())/mlibobj.Xs_tot[E0]
            continue
        if (len(b_cross) == 1) and (b_cross[0] == 0. or
            b_cross[0] == slabobj.length):
            # the neutron has crossed one outer
            # boundary---> out of the slab
            alive = 0


        if (len(b_cross) >= 1) and ((b_cross[0] != 0.) and
           (b_cross[0] != slabobj.length)):
            # the neutron has crossed at least one inner
            # boundary---> virtual collision
            # change only starting point abscissa
            # and distance covered
            if slabobj.location(x0) == 'Fuel':
                # the neutron has entered a moderator region
                l =
                    - math.log(1 - seed.random())/mlibobj.Xs_tot[E0]
            if slabobj.location(x0) == 'Moderator':
                # the neutron has entered a fuel region
                l =
                  - math.log(1 - seed.random())/flibobj.Xs_tot[E0]

            x0 = b_cross[0]

# Change the shape of the vector in an N x 2 matrix
# to have couples of positions and energy groups
Abs_fuel_matrix = np.reshape(
    Abs_fuel_matrix, (int(len(Abs_fuel_matrix)/2),2))

return fission_func(Abs_fuel_matrix, slabobj, flibobj, seed)
```

Random walk - κ_0 - Symmetric case

```
import numpy as np
import sys
sys.path.append('.')
import math
from fermi import fission_func

# LEGEND
# triple.T[0] ---> Column of the array with the starting
# positions of the neutrons (always in a fuel layer)
# triple.T[1] ---> Column of the array with the starting
# energy group of the neutrons
# triple.T[2] ---> Column of the array with the starting
# directions of neutrons
# slabobj ---> Domain definition object (with boundaries,
# material qualification, detector definition, ...)
# flibobj ---> Data set of fuel material
# mlibobj ---> Data set of moderator/reflector material
# seed ---> seed of the pseudo-random generator



def Random_walk_k_symm(triple, slabobj, flibobj, mlibobj, seed):

    Abs_fuel_matrix = []

    for fc in range(len(triple)):
        # starting abscissa, energy group and direction (cosine)
        x0 = triple[fc][0]
        E0 = int(triple[fc][1])
        mu = triple[fc][2]

        if (x0 < 0) or (x0 > slabobj.length):
            raise OSError('Out of the slab !')

        if slabobj.location(x0) == 'Fuel':
            # distance covered if it starts from a fuel region
        l = - math.log(1 - seed.random())/flibobj.Xs_tot[E0]
    else:
        # distance covered if it starts from a moderator region
        l = - math.log(1 - seed.random())/mlibobj.Xs_tot[E0]
    # flag setting
    alive = 1
    # Start the random walk
    while(alive):

        # new position in the slab
        x = x0 + l*mu

        # how many inner boarders did the neutron cross ?
        b_cross = slabobj.boundarycounter_s(x0, x)

        if (len(b_cross) == 0) and (slabobj.location(x0) ==
            'Fuel'):
            # the neutron has not crossed any inner
            # boundaries and is still in a fuel layer
            rho1 = seed.random()
            if rho1 <= flibobj.Abs[E0]/flibobj.Xs_tot[E0]:
                # absorption occurred
                end_point = x
                end_group = E0
                Abs_fuel_matrix = np.append(
                Abs_fuel_matrix, np.append(x,E0))
                alive = 0
            else:
                # scattering occurred: energy, cosine direction,
                # starting point abscissa
                # and distance covered change
                E  = np.argmax(
                np.where(
                flibobj.mat_cum_scat[E0] - seed.random() < 0))
                E0 = int(E)
                mu = seed.uniform(-1,1)
                x0 = x
                l =
              - math.log(1 - seed.random())/flibobj.Xs_tot[E0]
            continue


    if (len(b_cross) == 0) and (slabobj.location(x0) ==
        'Moderator'):
        # the neutron has not crossed any inner boundaries
        # and is still in a moderator layer
        rho2 = seed.random()
        if rho2 <= mlibobj.Abs[E0]/mlibobj.Xs_tot[E0]:
            # absorption occurred
            alive = 0
        else:
            # scattering occurred: energy, direction,
            # starting point
            # and distance covered change
            E  = np.argmax(
            np.where(
            mlibobj.mat_cum_scat[E0] - seed.random() < 0))
            E0 = int(E)
            mu = seed.uniform(-1,1)
            x0 = x
            l =
              - math.log(1 - seed.random())/mlibobj.Xs_tot[E0]
            continue


    if (len(b_cross) == 1) and (b_cross[0] == 0. or
        b_cross[0] == slabobj.length):
        # the neutron has crossed one outer
        # boundary---> out of the slab
        alive = 0


    if (len(b_cross) >= 1) and ((b_cross[0] != 0.) and
       (b_cross[0] != slabobj.length)):
        # the neutron has crossed at least one inner
        # boundary---> virtual collision
```

```
                # change only starting point abscissa
                # and distance covered
                if slabobj.location(x0) == 'Fuel':
                    # the neutron has entered a moderator region
                    l =
                       - math.log(1 - seed.random())/mlibobj.Xs_tot[E0]
                if slabobj.location(x0) == 'Moderator':
                    # the neutron has entered a fuel region
                    l =
                      - math.log(1 - seed.random())/flibobj.Xs_tot[E0]

                x0 = b_cross[0]
        if (len(b_cross) >= 1) and (b_cross[0] ==
           slabobj.xmid):
            # the neutron has crossed the symmetry axis
            # and for each neutron that leaves the first half,
            # another enters it
            # with opposite cosine ---> virtual collision

            x0 = b_cross[0]
            mu *= -1

            if slabobj.location(x0) == 'Fuel':
                # the symmetry axis is in a fuel region
                l = - math.log(
                 1 - seed.random())/flibobj.Xs_tot[int(E0)]
            if slabobj.location(x0) == 'Moderator':
                # the symmetry axis is in
                # a moderator region
                l =
                  - math.log(1 - seed.random())/mlibobj.Xs_tot[int(E0)]

    # Change the shape of the vector in an N x 2 matrix
    # to have couples of positions and energy groups
    Abs_fuel_matrix = np.reshape(
        Abs_fuel_matrix, (int(len(Abs_fuel_matrix)/2),2))

    return fission_func(Abs_fuel_matrix, slabobj, flibobj, seed)
```

Population controller - κ_0

The case of multiprocessing is taken into account. So, the sum of the list of detector matrices must take place.

```
import numpy as np
import sys
sys.path.append('.')
import math


# LEGEND
# double_f ---> List of matrices whose elements are
# the neutronic sources in each detector
# flibobj ---> Dictionary of fuel material
# slabobj ---> Domain definition object
# (with boundaries, material qualification, detector definition, ...)
# old_pop ---> Amount of neutrons in
# the previous generation
# N0 ---> Amount of neutrons in the 0-th generation
# seed ---> seed of the pseudo-random number generator

class Population_controller:
    def __init__(self, double_f, flibobj, slabobj, old_pop,
          N0, seed):

        # Unify the single matrices of the list
        slabobj.det_matr_f = np.sum(
                    np.asarray(double_f), axis=0)

        # Sum the previous matrix to get the
        # new population generated
        self.new_gene = np.sum(
        np.sum(
        np.asarray(double_f), axis=0))

        self.zero_gene = N0
        self.old_gene = old_pop
```

```
# Compute the K-eigenvalue
self.K_eig = self.new_gene / self.old_gene


# Compute the normalization factor
fact = self.zero_gene / self.new_gene

# Compute the normalized ('future old')
# generation and the elements
# for the Shannon entropy.
# Update positions, energy groups and directions
#for the next random walks;
# also compute different
# random number generator
#for each new RW
self.new_born_sites = []
self. new_energy_leve = []
self.norm_new_gene = 0

# If it is possible, compute the C.O.M. for
# the new generation of neutrons
self.COM = 0

for ii in range(slabobj.fuel_layers):
    for jj in range(slabobj.det_per_fuel_region):

        if slabobj.det_matr_f[ii][jj] == 0.:
            continue

        # Normalize each detector source
        slabobj.norm_matr_t[ii][jj] =
          slabobj.det_matr_f[ii][jj] * fact
        norm_per_slot = int(
        np.ceil(slabobj.norm_matr_t[ii][jj]))

        left_ex = slabobj.mat_lay_fuel[ii][0] +
                  jj*slabobj.dx_vf[ii]
        right_ex = slabobj.mat_lay_fuel[ii][0] +
                (jj+1)*slabobj.dx_vf[ii]

        # Add the new positions inside the detector
        self.new_born_sites = np.append(
           self.new_born_sites, seed.uniform(
              left_ex, right_ex, norm_per_slot))

        # Update C.O.M. ()
        self.COM += ((left_ex + right_ex)  *
          0.5 - slabobj.xmid) * norm_per_slot

        # Update the normalized generation counter
        self.norm_new_gene += norm_per_slot
# Weighted average for C.O.M.
self.COM /= self.norm_new_gene

# Update energy groups and seeds for next RWs
extr = int(np.ceil(seed.random()*1e8))
self.rnd_sd_v = seed.permutation(
          [np.random.default_rng(i) for i in range(
            extr*self.norm_new_gene,
            (extr+1)*self.norm_new_gene)])

for sss in range(len(self.new_born_sites)):
    self.new_energy_leve =
    np.append(
    self.new_energy_leve, np.argmax(
    np.where(flibobj.En_fiss_cum - seed.random() < 0)))


# Update directions for next RWs
self.new_cosines = seed.uniform(
 -1, 1, len(self.new_born_sites))

# Compute S.E.
self.Shannon_en = 0

for ii in range(slabobj.fuel_layers):
```

```
            for jj in range(slabobj.det_per_fuel_region):
                if slabobj.norm_matr_t[ii][jj] != 0:
                    self.Shannon_en -=
                    (slabobj.norm_matr_t[ii][jj] / self.norm_new_gene)*
                    math.log2(
                    slabobj.norm_matr_t[ii][jj] / self.norm_new_gene)
```

General algorithm - κ_0- Multi-thread case

```
import matplotlib.pyplot as plt
import numpy as np
import sys
sys.path.append('.')
from Read_data import Dataset_organizer
from slab import Domain_assembler
from flight import Random_walk_k
from flight_s import Random_walk_k_symm
from controls import Population_controller_symm
from control import Population_controller
from postproc import Statistics_executor
from primasource import Initial_source_setter
from images import Results_visualizer
import time
import multiprocessing as mp
from functools import partial

def main():
    # Define the domain : a multi-layer slab
    # made of alternating thicknesses of fuel
    # and moderator/reflector with the respective
    # lengths (expressed in cm).
    # Define the number of detectors in the fuel regions.
    # The definition of other inputs are written in class files

    Detector_per_fuel_layer = 50
    Reactor = Domain_assembler(
    ['Moderator', 'Fuel', 'Moderator'], [2., 2., 2.],
   Detector_per_fuel_layer)

# State the number of processors involved
# in the parallelization
omp =  mp.cpu_count()
# State the seed for the pseudo-random
# generation of numbers for
# the MC simulation (to be passed to every function
# and class)
rng = np.random.default_rng(81)

# Read the data from the given files and organise
# them according to the chosen number of energy group
Num_en_group = 2
Comb = Dataset_organizer('sood_fuel.txt', Num_en_group)

Moder = Dataset_organizer('sood_reflector.txt', Num_en_group)

#Define the number of initial neutrons in each detector
Neut_orig = 12000

# Symmetry flag
flag_symm = 0

# Is the slab symmetric ?
# Define the number of initial neutrons
# in each detector. Specify the geometry, the
# neutronic distribution and
# the angular direction of the initial source
if Reactor.issymmetric() == True:
    Gen_zero = Initial_source_setter(
    Comb, Reactor, Neut_orig, rng, 'Symmetric',
    'Fission-source-like', 'Stochastic')
    flag_symm = 1
else:
    Gen_zero = Initial_source_setter(
    Comb, Reactor, Neut_orig, rng, 'Non-symmetric',
    'Fission-source-like', 'Stochastic')
```

```
# Get the positions of the 0-th generation of
# neutrons, their directions
# and their energy groups from the object
# defined above.
# Generate one random number generator
# seed per starting neutron
# to avoid sampling the same neutrons
# multiple times
extr = int(np.ceil(rng.random()*1e8))
rngs = rng.permutation(
[np.random.default_rng(i) for i in range(
extr*Neut_orig, (extr+1)*Neut_orig)])
start_inputs = np.c_[Gen_zero.start_positions,
Gen_zero.start_energies,
Gen_zero.start_directions, rngs]

# State the number of inactive cycles for the Monte Carlo
# computation of eigenvalue K and Shannon Entropy (S.E.)
# and the arrays for the respective random variables
Ina_cycles = 0
Ina_cycles_sh = 0
Kappa_vect = []
S_entr = []
COM_vec = []

# Set the errors to 1 and state
# the tolerances for
# K-eigenvalue and S.E.
error1 = 1
error2 = 1

tol1 = 0.00001
tol2 = 0.0005


# Initialize the population and the matrix
# for the MC calculation
# (1st row Sample average,
# 2nd row Second order moment, 3rd row
# Variance, 4th row RSD, 5th row error bar)
# for Kappa and S.E.
# The active cycles for K-eigenvalue must
# be set to zero,
# such as the counter of the iterations
old_popu = Neut_orig
cycle = 0
Act_cycles = 0
MC_kappa_matr = np.empty((5,1))
MC_Shan_matr = MC_kappa_matr.copy()

# Measure the time needed for all
# the loops (set to zero)
time_tot = 0

# State the minimum number of active cycles and
# the maximum one
min_n = 100
max_n = 100

# Modify the functions of random walk and fission in
# order to have only a variable input (the 'iterable' for
# the parallelization)
if flag_symm == 1:
    Random_walk_fiss_k_par = partial(
     Random_walk_k_symm, slabobj = Reactor,
     flibobj = Comb,
     mlibobj = Moder)
else:
    Random_walk_fiss_k_par = partial(
     Random_walk_k, slabobj = Reactor, flibobj = Comb,
     mlibobj = Moder)



# Start the while loop for K-eigenvalue calculation
```

```
while ((error1 > tol1) or (Act_cycles < min_n)) and
    (Act_cycles < max_n):

    # Start the timer
    t1 = time.time()
    cycle += 1
    print('Cycle number', cycle)

    # For the S.E. every cycle is active
    Act_cycles_sh = cycle

    # Random walk begins and then tallying
    if flag_symm == 1:
        # Parallelization of the 'Symmetric' RW function
        pool = mp.Pool(omp)
        ext = pool.map(Random_walk_fiss_k_par, start_inputs)

        # Tallying, Sorting and controlling
        # the neutronic population
        # Computing the K-eigenvalue, S.E.
        # and other quantities
        results = Population_controller_symm(ext, Comb, Reactor,
        old_popu, Neut_orig, rng)
    else:
        # Parallelization of the RW function
        pool = mp.Pool(omp)
        ext = pool.map(Random_walk_fiss_k_par, start_inputs)

        # Tallying, sorting and controlling
        # the neutronic population
        # Computing the K-eigenvalue, S.E.
        # and other quantities
        results = Population_controller(
            ext, Comb, Reactor, old_popu, Neut_orig, rng)

    # Updating inputs for the next iteration
    old_popu = results.norm_new_gene
    start_inputs = np.c_[results.new_born_sites,
    results.new_energy_leve,
    results.new_cosines, results.rnd_sd_v]

    # Add the results of the running loop to
    # the respective arrays
    Kappa_vect = np.append(Kappa_vect, results.K_eig)
    S_entr = np.append(S_entr, results.Shannon_en)
    COM_vec = np.append(COM_vec, results.COM)

    # Set detector matrix to zero for the new iteration
    if flag_symm == 1:
        Reactor.det_matr_f_s = np.zeros((Reactor.fuel_layers_s,
        Reactor.det_per_fuel_region))
    else:
        Reactor.det_matr_f = np.zeros((Reactor.fuel_layers,
        Reactor.det_per_fuel_region))

    print('K:  ', results.K_eig)
    print('S.E.:  ', results.Shannon_en)

    # Compute the update of the statistical
    # measurements and
    # add them to the appropriate array (S.E.)
    sh_prc = Statistics_executor(
    S_entr, Act_cycles_sh, Ina_cycles_sh)
    MC_Shan_matr = np.append(
    MC_Shan_matr, sh_prc.MC_update, axis=1)

    if sh_prc.MC_update[2] > 0.:
        # Update the value of the error (S.E.)
        error2 = sh_prc.MC_update[3]
        print('Error of S.E.:', error2)

    if (error2 > tol2) or (cycle <= 10):
        # Build up the number of inactive cycles until the
        # appropriate condition is satisfied
        Ina_cycles += 1
```

```
        # Since at the first iterations the error can
        # randomly fluctuate below
        # the tolerance, a minimum number of cycle
        # is needed to avoid this
        if (error2 <= tol2) and (cycle > 10):
            # Updating of the number of active cycles
            Act_cycles = cycle - Ina_cycles

            # Compute the update of the statistical
            # measurements and add them
            # to the appropriate array (K-eigenvalue)
            p_prc = Statistics_executor(
            Kappa_vect, Act_cycles, Ina_cycles,
            amp_err_barr)
            MC_kappa_matr = np.append(
            MC_kappa_matr, p_prc.MC_update,
            axis=1)

            if p_prc.MC_update[2] > 0.:
                # Update the value of the error (K-eigenvalue)
                error1 = p_prc.MC_update[3]
                print('Error: ', error1)

        # Stop the timer
        t2 = time.time()
        print("This cycle took",t2 - t1,"seconds")
        print('\n')

        # Increment of the time needed to run the operations
        time_tot += (t2 - t1)

    Tot_cycles = cycle

    MC_kappa_matr = np.delete(MC_kappa_matr, 0, axis=1)
    MC_Shan_matr = np.delete(MC_Shan_matr, 0, axis=1)


    # Print the significant results
    print("Total duration:", time_tot,"seconds")
    print("\n")
    print("Average cycle duration:", time_tot/Tot_cycles, "seconds")
    print('\n')
    print("Total number of cycles: ", Tot_cycles)
    print('\n')
    print("Number of active cycles: ", Act_cycles)
    print('\n')
    print("Number of inactive cycles: ", Ina_cycles)
    print('\n')
    print("K-eigenvalue:", MC_kappa_matr[0][-1], "+-",
    MC_kappa_matr[-1][-1])


    # Plot all the charts
    p_k = plotting(COM_vec, Kappa_vect, MC_kappa_matr[0],
    MC_kappa_matr[1],
    MC_kappa_matr[2], MC_kappa_matr[3], MC_kappa_matr[4],
    Ina_cycles,
    Tot_cycles, "\u03BA0 Eigenvalue")
    p_s = plotting(COM_vec, S_entr, MC_Shan_matr[0],
    MC_Shan_matr[1],
    MC_Shan_matr[2], MC_Shan_matr[3],
    MC_Shan_matr[4],
    Ina_cycles_sh, Tot_cycles, "Shannon Entropy")

    plt.show()

if __name__ == "__main__":
    main()
```

Appendix C

This final appendix, analogously to the previous one, is dedicated just for γ_0 and its peculiar classes/functions.

Random walk - γ_0

```
import numpy as np
from os import path
from pathlib import Path
import sys
sys.path.append('.')
import math
from crash import scatt_func
from fermi import fission_g

# LEGEND
# triple.T[0] ---> Column of the array with the
# starting positions of the neutrons (always in a fuel layer)
# triple.T[1] ---> Column of the array
# with the starting energy group of the neutrons
# triple.T[2] ---> Column of the array with the
# starting directions of neutrons
# slabobj ---> Domain definition object
# (with boundaries, material qualification, detector definition, ...)
# flibobj ---> Data set of fuel material
# mlibobj ---> Data set of moderator/reflector material
# seed ---> seed of the pseudo-random generator
```

```
def Random_walk_g(self, triple, slabobj, flibobj,
  mlibobj, seed):

    abs_pos_vec = []
    abs_group_vec = []

    sca_pos_vec = []
    sca_group_vec = []

    for fc in range(len(triple)):
        # starting abscissa, energy group and
        # direction (cosine)
        x0 = triple[fc][0]
        E0 = int(triple[fc][1])
        mu = triple[fc][2]

        if (x0  < 0) or (x0 > slabobj.length):
            raise OSError('Out of the slab !')


        if slabobj.location(x0) == 'Fuel':
            # distance covered if it starts from a fuel region
            l =
             - math.log(1 - seed.random())/flibobj.Xs_tot[E0]
        else:
            # distance covered if it starts
            # from a moderator region
            l =
            - math.log(1 - seed.random())/mlibobj.Xs_tot[E0]
        # flag setting
        alive = 1
        # Start the random walk
        while(alive):

            # new position in the slab
            x = x0 + l * mu
```

```
# how many inner boarders did
# the neutron cross ?
b_cross = slabobj.boundarycounter(x0, x)

if (len(b_cross) == 0) and
    (slabobj.location(x0) == 'Fuel'):
    # the neutron has not crossed any inner
    # boundaries and is still in a fuel layer
    rho1 = seed.random()
    if rho1 <= flibobj.Abs[E0]/flibobj.Xs_tot[E0]:
        # absorption occurred
        abs_pos_vec = np.append(abs_pos_vec, x)
        abs_group_vec =
        np.append(abs_group_vec, E0)
        alive = 0
    else:
        # scattering occurred
        sca_pos_vec =
        np.append(sca_pos_vec, x)
        sca_group_vec =
        np.append(sca_group_vec, E0)
        alive = 0


if (len(b_cross) == 0) and
      (slabobj.location(x0) == 'Moderator'):
    # the neutron has not crossed any inner
    # boundaries and is still in a moderator layer
    rho2 = seed.random()
    if rho2 <=
        mlibobj.Abs[E0]/mlibobj.Xs_tot[E0]:
        # absorption occurred
        alive = 0

    else:
        # scattering occurred
        sca_pos_vec =
        np.append(sca_pos_vec, x)
```

```
                    sca_group_vec =
                    np.append(sca_group_vec, E0)
                    alive = 0


            if (len(b_cross) == 1) and (b_cross[0] == 0 or
                    b_cross[0] == slabobj.length):
                # the neutron has crossed
                # one outer boundary---> out of the slab
                alive = 0


            if len(b_cross) >= 1 and (b_cross[0] != 0
                   and b_cross[0] != slabobj.length):
                # the neutron has crossed at least
                # one inner boundary---> virtual collision
                # change only starting point abscissa
                # and distance covered
                if slabobj.location(x0) == 'Fuel':
                    # the neutron has entered a moderator region
                    l =
                    - math.log(1 - seed.random())/
                      mlibobj.Xs_tot[E0]
                if slabobj.location(x0) == 'Moderator':
                    # the neutron has entered a fuel region
                    l =
                    - math.log(1 - seed.random())/
                      flibobj.Xs_tot[E0]

                x0 = b_cross[0]
                continue


    # Apply the proper function to each matrix to get
    # the number of new neutrons from fission and
    # the new energy from
    # scattering
    end_matrix_fiss =
        fission_g(
        np.c_[abs_pos_vec, abs_group_vec], slabobj, flibobj, seed)
        end_matrix_scat =
        scatt_func(
        np.c_[sca_pos_vec, sca_group_vec],
        slabobj, flibobj, mlibobj, seed)

        return end_matrix_fiss, end_matrix_scat
```

Random walk - γ_0 - Symmetric case

```
import numpy as np
from os import path
from pathlib import Path
import sys
sys.path.append('.')
import math
from crash import scatt_func
from fermi import fission_g

# LEGEND
# triple.T[0] ---> Column of the array with the
# starting positions of the neutrons (always in a fuel layer)
# triple.T[1] ---> Column of the array
# with the starting energy group of the neutrons
# triple.T[2] ---> Column of the array with the
# starting directions of neutrons
# slabobj ---> Domain definition object
# (with boundaries, material qualification, detector definition, ...)
# flibobj ---> Data set of fuel material
# mlibobj ---> Data set of moderator/reflector material
# seed ---> seed of the pseudo-random generator



    def Random_walk_g_symm(self, triple, slabobj, flibobj,
      mlibobj, seed):
```

```
abs_pos_vec = []
abs_group_vec = []

sca_pos_vec = []
sca_group_vec = []

for fc in range(len(triple)):
    # starting abscissa, energy group and
    # direction (cosine)
    x0 = triple[fc][0]
    E0 = int(triple[fc][1])
    mu = triple[fc][2]

    if (x0  < 0) or (x0 > slabobj.length):
        raise OSError('Out of the slab !')


    if slabobj.location(x0) == 'Fuel':
        # distance covered if it starts from a fuel region
        l =
         - math.log(1 - seed.random())/flibobj.Xs_tot[E0]
    else:
        # distance covered if it starts
        # from a moderator region
        l =
        - math.log(1 - seed.random())/mlibobj.Xs_tot[E0]
    # flag setting
    alive = 1
    # Start the random walk
    while(alive):

        # new position in the slab
        x = x0 + l * mu

        # how many inner boarders did
        # the neutron cross ?
        b_cross = slabobj.boundarycounter_s(x0, x)
```

```
if (len(b_cross) == 0) and
    (slabobj.location(x0) == 'Fuel'):
    # the neutron has not crossed any inner
    # boundaries and is still in a fuel layer
    rho1 = seed.random()
    if rho1 <= flibobj.Abs[E0]/flibobj.Xs_tot[E0]:
        # absorption occurred
        abs_pos_vec = np.append(abs_pos_vec, x)
        abs_group_vec =
        np.append(abs_group_vec, E0)
        alive = 0
    else:
        # scattering occurred
        sca_pos_vec =
        np.append(sca_pos_vec, x)
        sca_group_vec =
        np.append(sca_group_vec, E0)
        alive = 0


if (len(b_cross) == 0) and
      (slabobj.location(x0) == 'Moderator'):
    # the neutron has not crossed any inner
    # boundaries and is still in a moderator layer
    rho2 = seed.random()
    if rho2 <=
        mlibobj.Abs[E0]/mlibobj.Xs_tot[E0]:
        # absorption occurred
        alive = 0

    else:
        # scattering occurred
        sca_pos_vec =
        np.append(sca_pos_vec, x)
        sca_group_vec =
        np.append(sca_group_vec, E0)
        alive = 0
```

```
if (len(b_cross) == 1) and (b_cross[0] == 0 or
         b_cross[0] == slabobj.length):
    # the neutron has crossed
    # one outer boundary——> out of the slab
    alive = 0


if len(b_cross) >= 1 and (b_cross[0] != 0
        and b_cross[0] != slabobj.length):
    # the neutron has crossed at least
    # one inner boundary——> virtual collision
    # change only starting point abscissa
    # and distance covered
    if slabobj.location(x0) == 'Fuel':
        # the neutron has entered a moderator region
        l =
        - math.log(1 - seed.random())/
          mlibobj.Xs_tot[E0]
    if slabobj.location(x0) == 'Moderator':
        # the neutron has entered a fuel region
        l =
        - math.log(1 - seed.random())/
          flibobj.Xs_tot[E0]

    x0 = b_cross[0]
    continue
    if (len(b_cross) >= 2) and
    (b_cross[1] == slabobj.xmid):
    # the neutron has crossed the symmetry
    # axis and for each
    # neutron that leaves the first half,
    # another enters it
    # with opposite cosine; other
    # parameters do not change

       x0 = b_cross[1]
```

```
                    mu *= -1

                    if slabobj.location(x0) == 'Fuel':
                    # the symmetry axis is in a fuel region
                        l = - math.log(1 - seed.random())/
                        flibobj.Xs_tot[E0]
                    if slabobj.location(x0) == 'Moderator':
                    # the symmetry axis is in a moderator region
                        l = - math.log(1 - seed.random())/
                       mlibobj.Xs_tot[E0]


        # Apply the proper function to each matrix to get
        # the number of new neutrons from fission and
        # the new energy from
        # scattering
        end_matrix_fiss =
        fission_g(
        np.c_[abs_pos_vec, abs_group_vec], slabobj, flibobj, seed)
        end_matrix_scat =
        scatt_func(
        np.c_[sca_pos_vec, sca_group_vec],
        slabobj, flibobj, mlibobj, seed)

        return end_matrix_fiss, end_matrix_scat
```

Scattering function

```
import numpy as np
from os import path
from pathlib import Path
import sys
sys.path.append('.')

# LEGEND
# arr_bis.T[0] ---> Column of a matrix
# with the positions of the scattered neutrons
# arr_bis.T[1] ---> Column of a matrix with
# the energy groups of the scattered neutrons
# slabobj ---> Domain definition object
# (with boundaries, material qualification, detector definition, ...)
# flibobj ---> Data set of fuel material
# seed ---> seed of the pseudo-random generator

# The outputs are the positions of these neutrons
# (first column unchanged) and their energy groups

def scatt_func(arr_bis, slabobj, flibobj, mlibobj, seed):

    new_gr = np.zeros(arr_bis.shape)
    new_gr.T[0] = arr_bis.T[0]

    for gg in range(len(arr_bis)):

        if slabobj.location(arr_bis[gg][0]) == "Fuel":
            new_gr[gg][1] =
            np.argmax(
            np.where(
            flibobj.mat_cum_scat[int(arr_bis[gg][1])] - seed.random()
            < 0))
        if slabobj.location(arr_bis[gg][0]) == "Moderator":
            new_gr[gg][1] =
            np.argmax(
            np.where(
            mlibobj.mat_cum_scat[int(arr_bis[gg][1])] - seed.random()
            < 0))

    return new_gr
```

Population controller - γ_0

The case of multiprocessing is taken into account. So, the sum of the list of detector matrices must take place.

```
import math
import numpy as np
import sys
sys.path.append('.')

# LEGEND
# fiss_2da.T[0] ---> Column of the matrix whose elements
# are the positions of the fission sites
# fiss_2da.T[1] ---> Column of the matrix whose elements
# are neutrons emitted per fission site
# scatt_2da.T[0] ---> Column of the matrix whose elements
# are the positions of the scattering sites
# scatt_2da.T[1] ---> Column of the matrix whose elements
# are neutrons emitted per scattering energy groups
# scatt_cord.T[0] ---> Column of the matrix whose elements
# are the positions of the scattering sites
# scatt_cord.T[1] ---> Colum of the matrix whose elements are
# the energy group of the neutrons after scattering events
# flibobj ---> Data set of fuel material
# slabobj ---> Domain definition object
# (with boundaries, material qualification,
# detector definition, ...etc)
# old_pop ---> Amount of neutrons in the previous generation
# N0 ---> Amount of neutrons in the 0-th generation
# seed ---> seed of the pseudo-random generator

class Population_controller:
    def __init__(self, fiss_2da, scatt_2da,
         scatt_coord, flibobj, slabobj, old_pop, N0, seed):

        sct_coord = np.asarray(scatt_coord)
        inner = sct_coord[:, 0] != -2.
        sct_coord = sct_coord[inner]

        # Unify the sources (Fissions and scattering events separated)
        # in each detector
        slabobj.det_matr_f =
        np.sum(np.asarray(fiss_2da), axis=0)
        slabobj.det_matr_s =
        np.sum(np.asarray(scatt_2da), axis=0)

        # Merge the 2 matrices
        det_matr_t =
        slabobj.det_matr_f + slabobj.det_matr_s

        # Sum the previous matrix
        #to get the new population generated
        self.new_gene = np.sum(det_matr_t)

        self.old_gene = old_pop
        self.zero_gene = N0

        # Compute the Gamma-eigenvalue
        self.Gamma_eig =
        self.new_gene / self.old_gene

        # Compute the normalization factor
        fact = self.zero_gene / self.new_gene

        #Compute the normalized ('future old')
        # generation and the elements
        # for the Shannon entropy. Update
        # positions, energy groups and directions
        # for the next random walks; also
        # compute different random number
        # generator for each new RW

        self.new_born_sites = []
        self.new_energy_leve = []

        # Compute the C.O.M. (if it is possible)
        # for the new generation of neutrons
        self.COM = 0

        # Normalization and update of position
        # and energy group for the next RWs
        for ii in range(slabobj.num_lay):
```

```
for jj in range(slabobj.det_per_region):

    if det_matr_t[ii][jj] == 0.:
        continue

    # Normalize each detector source
    slabobj.norm_matr_t[ii][jj] =
    det_matr_t[ii][jj] * fact
    tot_norm_slot =
    int(
    np.ceil(
    slabobj.norm_matr_t[ii][jj]))

    left_ex =
    slabobj.m_ends[ii][0] +
    jj*slabobj.dx_v[ii]
    right_ex =
    slabobj.m_ends[ii][0] +
    (jj+1)*slabobj.dx_v[ii]

    # Update COM
    self.COM += ((left_ex + right_ex)
    * 0.5 - slabobj.xmid)
    * tot_norm_slot

    # Add the new positions inside the detector
    self.new_born_sites =
    np.append(
    self.new_born_sites,
    seed.uniform(left_ex, right_ex, tot_norm_slot))

    # Before the normalization, in each
    # detector there was this amount of neutrons
    tot_slot = det_matr_t[ii][jj]

    # Before the normalization, in each detector
    # there was this amount of fission neutrons
    f_slot = slabobj.det_matr_f[ii][jj]
```

```
# After the normalization, in each detector
# there must be this amount of fission neutrons
f_norm_slot = int(np.ceil(tot_norm_slot * f_slot /
tot_slot))

# After the normalization, in each detector
# there must be this amount of scattering neutrons
s_norm_slot = tot_norm_slot - f_norm_slot

# Energy groups of the new fission neutrons,
# according to the appropriate cumulative
# distribution function
if f_norm_slot > 0:
    for kk in range(f_norm_slot):
        self.new_energy_leve = np.append(
        self.new_energy_leve,
        np.argmax(
        np.where(
        flibobj.En_fiss_cum - seed.random() < 0)))

# How many scattering neutrons
# are there in a certain detector?
# Of which energy group?
lev_in_slot = []
for mm in range(len(sct_coord)):
    if (sct_coord[mm][0] >= left_ex) and
     (sct_coord[mm][0] <= right_ex):
        lev_in_slot =
        np.append(
        lev_in_slot, sct_coord[mm][1])


# If there is no scattering neutron in
# this detector or their amount is exactly
# equal to the correct normalized
# scattering neutronic population, the loop
# will continue with the analysis of the
# next detector after appending
# the lev_in_slot vector to the one
# of the general energy levels
if (len(lev_in_slot) == s_norm_slot) or
  (len(lev_in_slot) == 0):
    self.new_energy_leve =
     np.append(
     self.new_energy_leve, lev_in_slot)
    continue

# If there are scattering neutrons in this detector,
# the scattering population must be modified
# in a proportional
# way according to the percentage of
# the various energy group of the pre-normalized
# scattering population in this
# detector
if len(lev_in_slot) != s_norm_slot:
    # Define a vector whose elements will be
    # the number of neutrons per each energy group
    perc_v = np.zeros(flibobj.n_gr)

    # Define a vector whose elements will be as much
    # as the normalized scattering
    # sources in the detector and
    # every element is the energy group itself of that
    # particular neutron (similar to lev_in_slot)
    lev_in_slot_b = []

    # Define the cumulative of perc_v in order to
    # know which group has the
    # most neutrons in the detector
    # and where to find them in lev_in_slot_b
    cum_v = np.zeros(len(perc_v) + 1)

    # Loop to fill up perc_v
    for ww in range(len(perc_v)):
        for rr in range(len(lev_in_slot)):
            if lev_in_slot[rr] == ww:
                perc_v[ww] += 1

        # Update cum_v
        cum_v[ww+1] =
        int(cum_v[ww] + perc_v[ww])

        # Update lev_in_slot_b
        nrm_lev_num =
        int(np.ceil(
        (1 / len(lev_in_slot)) * s_norm_slot *
        perc_v[ww]))
        lev_in_slot_b =
        np.append(lev_in_slot_b, ww *
        (np.ones(nrm_lev_num)))

    # The new number of scattering
    # neutrons (lev_in_slot_b)
    # is surely bigger, for rounding up,
    # than the
    # correct number (s_norm_slot):
    # The difference between
    # them (delta) is cleared by killing
    # delta neutrons from the most
    # represented energy group
    cum_v = np.delete(cum_v, 0, axis=0)

    # How many scattering neutrons from the most
    # represented energy group must be deleted ?
    delta = int(len(lev_in_slot_b) - s_norm_slot)

    # Which energy group has the biggest number
    # of neutrons ?
    max_ind = np.argmax(perc_v)

    # Delete 'delta' scattering neutrons
    # from that energy group
    for hh in range(delta):
```

```
                lev_in_slot_b = np.delete(
                lev_in_slot_b , int(
                cum_v[max_ind] - perc_v[max_ind]), axis=0)

            # Append lev_inslot_b to
            # general energy levels vector
            self.new_energy_leve = np.append(
            self.new_energy_leve, lev_in_slot_b)


# Update directions for the next RWs
self.new_cosines =
seed.uniform(-1, 1, len(self.new_born_sites))

# Update the normalized generation counter
self.norm_new_gene = len(self.new_born_sites)
# Weighted average for C.O.M.
self.COM /= self.norm_new_gene

# Compute S.E.
self.Shannon_en = 0

for ii in range(slabobj.num_lay):
    for jj in range(slabobj.det_per_region):
        if slabobj.norm_matr_t[ii][jj] != 0:
            self.Shannon_en -=
            (slabobj.norm_matr_t[ii][jj] /
            self.norm_new_gene) *
            math.log2(slabobj.norm_matr_t[ii][jj] /
            self.norm_new_gene)

# Set the seeds for next RWs
extr = int(np.ceil(seed.random()*1e8))
self.rnd_sd_v = seed.permutation(
np.random.default_rng(i) for i in range(
extr*self.norm_new_gene, (extr+1)*self.norm_new_gene)])

# Assemble these 4 new vectors in a matrix
```

```
        self.new_gen_matrix =
        np.c_[self.new_born_sites, self.new_energy_leve,
        self.new_cosines, self.rnd_sd_v]
```

General algorithm - γ_0- Multi-thread case

```
import matplotlib.pyplot as plt
import multiprocessing as mp
import numpy as np
import sys
sys.path.append('.')
from Read_data import Dataset_organizer
from slab import Domain_assembler
from control1 import Population_controller
from control1s import Population_controller_symm
from postproc import Statistics_executor
from primasource import Initial_source_setter
from images import Results_visualizer
from functools import partial
import time

def main():
    # Define the domain : a multi-layer slab
    # made of alternating thicknesses of fuel
    # and moderator/reflector with their
    # respective lengths (expressed in cm).
    # Define the number of detectors in the
    # every region and their total initial amount.
    # The definition of other inputs are written in
    # class files (Domain and Material data; see there)

     Detector_per_layer = 100
     Neut_orig = 10000

    # State the seed for the pseudo-random generation
    # of numbers for the MC
    # simulation (to be passed to every function
# and class)
rng = np.random.default_rng(81)

Reactor = Domain_assembler(['Moderator', 'Fuel', 'Moderator'],
[5.630757, 9.726784, 5.630757], Detector_per_layer, rng)

# Read the data from the given files and organize
# them according to the chosen number of energy groups
Num_en_group = 2
Comb =
Dataset_organizer('sood_fuel.txt', Num_en_group)
Moder =
Dataset_organizer('sood_reflector.txt', Num_en_group)


# Symmetry flag
flag_symm = 0

# Is the slab symmetric ? Define the number
# of initial neutrons in each detector.
# Specify the geometry, the
# neutronic distribution and the
# angular direction of the source
if Reactor.issymmetric() == True:
    Gen_zero = Initial_source_setter(
    Comb, Reactor, Neut_orig, rng, 'Symmetric',
    'Fission-source-like', 'Stochastic')
    flag_symm = 1
else:
    Gen_zero = Initial_source_setter(
    Comb, Reactor, Neut_orig, rng,
    'Non-symmetric', 'Uniform', 'Stochastic')

# Get the positions of the 0-th generation of
# neutrons (first column of the source_matrix), their energy
# groups (second column) and their directions
# (third column) from the object defined above;
# Generate one random number generator seed
# per starting neutron to avoid sampling the same
# neutrons multiple times
extr = int(np.ceil(rng.random()*1e8))
rngs = rng.permutation(
[np.random.default_rng(i) for i in range(
extr*Neut_orig, (extr+1)*Neut_orig)])
start_inputs = np.c_[
Gen_zero.start_positions, Gen_zero.start_energies,
Gen_zero.start_directions, rngs]

# State the number of inactive cycles for the
# Monte Carlo computation of eigenvalue Gamma and
# the Shannon entropy of the neutronic sources,
# also the vectors for random variables
Ina_cycles = 0
Ina_cycles_sh = 0
Gamma_vect = []
S_entr_g = []
COM_vec = []

# Impose the errors and the relative tolerances
# for Gamma and Shannon entropy
error1 = 1
error2 = 1

tol1 = 0.00001
tol2 = 0.001

# Initialize the population and the matrix for the MC
# calculation updates (1st row Sample average,
# 2nd row Second order moment, 3rd row Variance,
# 4th row RSD, 5th row error bar)
# for Gamma and Shannon entropy,
# the total number of cycles aand the number of
# active cycles for MC computation (for Gamma and Shannon entropy)
cycle = 0
Act_cycles = 0
MC_Gamma_matr = np.empty((5,1))
```

```
MC_Shan_matr = MC_Gamma_matr.copy()

# Initialize the 'old' population and the time
# needed for complete the requested calculation
old_popu = Neut_orig
time_tot = 0

# State the minimum number of active
# cycles and the maximum one
min_n = 200
max_n = 250

# Modify the functions of random walk and fission
# in order to have only a variable input (the iterable for
# the parallelization)
if flag_symm == 1:
    Random_walk_fiss_scat_g_par = partial(
    Random_walk_g_symm, flibobj = Comb, mlibobj = Moder)
else:
    Random_walk_fiss_scat_g_par = partial(
    Random_walk_g, flibobj = Comb, mlibobj = Moder)

# State the number of processors involved in the parallelization
omp =  mp.cpu_count()

# Start the while loop for Gamma-eigenvalue calculation
while ((error1 > tol1) or (Act_cycles < min_n)) and
   (Act_cycles < max_n):

    # Start the timer
    t1 = time.time()
    cycle += 1
    print('Cycle number', cycle)

    # For the S.E. every cycle is active
    Act_cycles_sh = cycle

    # Random walk begins and then tallying
    if flag_symm == 1:
        # Parallelization of the 'Symmetric' RW function
        pool = mp.Pool(omp)
        o1, o2, o3 = zip(
        *pool.map(
        Random_walk_fiss_scat_g_par, start_inputs))

        # Tallying, Sorting and controlling
        # the neutronic population
        # Computing the Gamma-eigenvalue,
        # S.E. and other quantities
        results =
        Population_controller_symm(
        o1, o2, o3, Comb, Reactor, old_popu, Neut_orig, rng)
    else:
        # Parallelization of the RW function
        pool = mp.Pool(omp)
        o1, o2, o3 = zip(
        *pool.map(
        Random_walk_fiss_scat_g_par, start_inputs))

        # Tallying, Sorting and controlling
        # the neutronic population
        # Computing the K-eigenvalue,
        # S.E. and other quantitie
        results =
        Population_controller(
        o1, o2, o3, Comb, Reactor, old_popu, Neut_orig, rng)

    # Updating inputs for the next iteration
    old_popu = results.norm_new_gene
    start_inputs = results.new_gen_matrix

    # Add the results of the running loop
    # to the respective arrays
    Gamma_vect =
    np.append(
    Gamma_vect, results.Gamma_eig)
```

```
S_entr_g =
np.append(
S_entr_g, results.Shannon_en)
COM_vec =
np.append(
COM_vec, results.COM)

print('Gamma: ', results.Gamma_eig)
print('S.E.: ', results.Shannon_en)

# Set detector matrices to zero
# for the next iteration
if flag_symm == 0:
    Reactor.det_matr_s =
    np.zeros((Reactor.num_lay,
    Reactor.det_per_region))
    Reactor.det_matr_f =
    Reactor.det_matr_s.copy()
else:
    Reactor.det_matr_s_s =
    np.zeros(
    (Reactor.num_lay_s, Reactor.det_per_region))
    Reactor.det_matr_f_s =
    Reactor.det_matr_s_s.copy()

# Compute the update of the statistical
# measurements and add them to the
# appropriate array (S.E.)
sh_prc =
Statistics_executor(
S_entr_g, Act_cycles_sh, Ina_cycles_sh)
MC_Shan_matr =
np.append(
MC_Shan_matr, sh_prc.MC_update,axis=1)

if sh_prc.MC_update[2] > 0.:
    # Update the value of the error (S.E.)
    error2 = sh_prc.MC_update[3]
```

```
        print('Error of S.E.:', error2)

    if (error2 > tol2) or (cycle <= 10):
        # Build up the number of inactive
        # cycles until the appropriate
        # condition is not satisfied
        Ina_cycles += 1

    # Since at the first iterations the error can
    # randomly fluctuate below the tolerance,
    # a minimum number of cycle
    # is needed to avoid this
    if (error2 <= tol2) and (cycle > 10):
        # Updating of the number of active cycles
        Act_cycles = cycle - Ina_cycles

        # Compute the update of the statistical
        # measurements and add them to the
        # appropriate array (Gamma-eigenvalue)
        p_prc =
        Statistics_executor(
        Gamma_vect, Act_cycles, Ina_cycles)
        MC_Gamma_matr =
        np.append(
        MC_Gamma_matr, p_prc.MC_update, axis=1)


        if p_prc.MC_update[2] > 0.:
            # Update the value of the
            # error (Gamma-eigenvalue)
            error1 = p_prc.MC_update[3]
            print('Error: ', error1)

    # Stop the timer
    t2 = time.time()
    print("This cycle took",(t2 - t1),"seconds")
    print('\n')
```

```
        # Increment of the time
        # needed to run the operations
        time_tot += (t2 - t1)

    Tot_cycles = cycle
    MC_Gamma_matr =
    np.delete(MC_Gamma_matr, 0, axis=1)
    MC_Shan_matr =
    np.delete(MC_Shan_matr, 0, axis=1)

    # Print the significant results
    print("Total number of cycles: ", Tot_cycles)
    print('\n')
    print("Number of active cycles: ", Act_cycles)
    print('\n')
    print("Number of inactive cycles: ", Ina_cycles)
    print('\n')
    print("Total duration:", time_tot, "seconds")
    print('\n')
    print("Average cycle duration:", time_tot/Tot_cycles, "seconds")
    print('\n')
    print("Gamma-eigenvalue:",
    MC_Gamma_matr[0][-1], "+-", MC_Gamma_matr[-1][-1])


    # Plot all the charts
    p_k = Results_visualizer(COM_vec, Gamma_vect,
    MC_Gamma_matr[0],
    MC_Gamma_matr[1],
    MC_Gamma_matr[2], MC_Gamma_matr[3], MC_Gamma_matr[4],
    Ina_cycles, Tot_cycles, "\u03B30 Eigenvalue")
    p_s = Results_visualizer(COM_vec, S_entr_g,
    MC_Shan_matr[0], MC_Shan_matr[1],
    MC_Shan_matr[2], MC_Shan_matr[3], MC_Shan_matr[4],
    Ina_cycles_sh, Tot_cycles, "Shannon Entropy")

    plt.show()
```

```
if __name__ == "__main__":
    main()
```

Bibliography

[1] Haghighat A. *Monte Carlo Methods for Particle Transport.* 2nd ed. Boca Raton, Florida: CRC Press, Taylor & Francis Group, 2021.

[2] Burrone M. *Study of eigenvalue formulations in the P_N approximation of the neutron transport equation.* Master Thesis, Politecnico di Torino, 2018.

[3] Courant R. and Hilbert D. *Methods of Mathematical Physics vol.1.* New Rochelle, New York: Interscience Publishers, Inc., 1953.

[4] Abrate N. et al. "On some features of the eigenvalue problem for the $\mathrm{P_N}$ approximation of the neutron transport equation". In: *Annals of Nuclear Energy* 163.108477 (2021), pp. 3–10.

[5] Montgomery D.C. and Runger G. C. *Applied Statistics and Probability for Engineers.* 3rd ed. New York, New York: John Wiley & Sons, Inc, 2003.

[6] Ross S. M. *Introduction to probability and statistics for engineers and scientists.* 3rd ed. Burlington, Massachusetts: Elsevier, 2004.

[7] Dulla S. *Lecture notes in Monte Carlo Methods.* 2019-2020.

[8] Kuhlman D. *A Python Book: Beginning Python, Advanced Python, and Python Exercises.* MIT Press, 2013.

[9] von Mises R. and Pollaczek-Geiringer H. "Praktische Verfahren der Gleichungsauflösung, (German) [Practical methods of solving equations]". In: *Zeitschrift für Angewandte Mathematik und Mechanik, (German) [Journal of Applied Mathematics and Mechanics]* 9.58–77 (1929), pp. 152–164.

[10] Sood A., Forster R. A., and Parsons D. K. "Analytical Benchmark Test Set For Criticality Code Verification". In: *Progress in Nuclear Energy* 42.1 (2003), pp. 55–106.

[11] Wulandari H., Jochum J., and von Feilitzsch. “Neutron flux at the Gran Sasso underground laboratory revisited”. In: *Astroparticle Physics* 22.3–4 (2003), pp. 313–322.

[12] Spanier J. and Gelbard E. M. *Monte Carlo Principles and Neutron Transport Problems*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1969.

[13] Kindler E. and Krivy I. “Object-Oriented Simulation of systems with sophisticated control”. In: *International Journal of General Systems* 40.3 (2011), pp. 313–343.

[14] Montagnini B. *Appunti del corso di Trasporto dei neutroni*. 1999-2000.

[15] Nowak M. et al. “Monte Carlo power iteration: Entropy and spatial correlations”. In: *Annals of Nuclear Energy* 94 (2016), pp. 856–868.

[16] Brown F. et al. “Reactor Physics Analysis with Monte Carlo”. In: ANS PHYSOR-2010 Conference Workshop, 9 May 2010, Pittsburgh, PA. 2010.