User Manual: Pebble Bed Reactor Equilibrium Simulation Tool

Ludovic Jantzen

April 7, 2025

Contents

1 Introduction

This document serves as a user manual for the Pebble Bed Reactor (PBR) equilibrium simulation tool. The purpose of this tool is to simulate the transition from fresh fuel to equilibrium by modeling neutron transport, fuel burnup, and pebble movement in a reactor. The simulation is performed using the Serpent Monte Carlo code, enhanced with the Cerberus framework.

2 Installation and Dependencies

To use this tool, ensure you have the following dependencies installed:

- Python (3.8)
- Required Python libraries:
  - pandas
  - numpy
  - matplotlib
  - cerberus

The Cerberus library can be found on the cluster storage under the path:

`/global/home/groups/co_nuclear/HxF_tools/Cerberus_HxF`

The first step will be to clone the given GitHub repository. Clone the repository:

```
git clone https://github.com/ludojantzen/search_equilibrium.
    git
cd search_equilibrium
```

It's recommended to clone it in a location where the simulation will have enough memory to store results (such as `scratch` space for example)

3 Repository Structure

This section describes the organization of the repository, outlining the key directories and files that compose the simulation framework for the Pebble Bed Nuclear Reactor. The repository is structured to facilitate ease of use, modification, and extension of the simulation capabilities.

3.1 Main Directories and Files

The repository is organized into the following main directories and files:

- **Utils/** – Contains the bash script for setting up the conda environment.
  - `setup_Cerberus` – Helps to install conda when you clone the repository for the first time. Refer to the next section 5 to obtain details about how to use the `setup_Cerberus` script.
  - `sss_environment` – Help to load the good modules and necessary serpent environment before running your simulation. See more in section 5.
  - `exe.sub` – Gives an example of sbatch file for the job.
- `search_equilibrium.py` – Main Python script containing the functions, and classes to simulate the reactor equilibrium. This script also contains some input parameters that you'll need to modify manually. For instance, this script needs to be fed with an actual Serpent input (reactor geometry and materials); In `search_equilibrium.py`, you'll point to the file including the Serpent reactor input.
- **Inputs/** – This directory includes the Serpent inputs describing the reactor geometry, material, and main simulation parameters. This directory contains an example of gFHR input. a generic salt-cooled pebble bed reactor. The script `search_equilibrium.py` should point to one of the files included in **Inputs/**.
- **Outputs/** – This directory includes the calculation outputs. The code creates a directory within **Outputs/** containing plots, results, and logs. The name of this subdirectory can be modified within the `search_equilibrium.py` input parameters.
- **Docs/** It contains the README.md and the LaTeX manual.

4 Simulation Framework and Loop Structure

The development of this tool has been driven by the need to reduce the computational time associated with high-fidelity pebble-wise transport and depletion simulations, such as HxF [1]. Instead of resolving neutron transport and depletion at the individual pebble level—which demands significant computational resources and memory—this tool adopts a zone-based approach. The reactor domain is divided into spatial zones where pebbles are depleted or activated based on an averaged neutronic environment, characterized by mean flux. The tool implements pebble motion by updating material compositions across these zones while handling key processes such as fuel discard, discharge, and reinsertion. This methodology inherently assumes a purely vertical fuel motion, making it particularly well-suited for reactors with cylindrical geometries.

The core simulation framework operates within a nested loop structure designed to ensure equilibrium in both neutron transport and fuel motion. The outer loop progresses through successive transport-depletion cycles, iterating until a global equilibrium condition is met—typically determined by the convergence of the effective neutron multiplication factor (k_{eff}). Within each outer iteration, an inner loop manages the coupled fuel motion and activation process, iterating until an equilibrium material distribution is established. This two-level iterative process ensures that the reactor dynamics, including neutron transport, fuel burnup, and material redistribution, are consistently captured while optimizing computational efficiency.

4.1 Outer Loop: Transport and Equilibrium Check

The outer loop is responsible for performing neutron transport simulations and depletion calculations over short time steps to determine the cross-section distribution. The workflow is as follows:

1. **Neutron Transport and Short Depletion:** The simulation advances in time using a very short step to obtain neutron flux and cross-section distributions. This ensures that the flux shape remains representative of the equilibrium conditions, independent of the fuel motion process.

2. **Flux Mapping and Visualization:** The calculated flux is stored in the pebble bed model and visualized using both pebble-wise and zone-wise plots.

3. **Multiplication Factor Tracking:** The current k_{eff} value is extracted and stored, along with its relative uncertainty. The variation of k_{eff} over successive iterations is used as a convergence criterion.

4. **Convergence Check:** If the relative difference between successive k_{eff} values is within a predefined tolerance ($\epsilon_{k_{\text{eff}}}$), the simulation terminates, indicating equilibrium.

5. **Initialize Motion and Activation Process:** If equilibrium is not yet achieved, the loop proceeds to the fuel motion step.

4.2 Inner Loop: Fuel Motion and Activation Step

The inner loop captures the dynamic fuel motion and activation processes, iterating until the system achieves equilibrium in material composition and burnup distribution.

1. **Motion Matrix Update:** Motion matrices are constructed to determine how pebbles move through the core. The axial and radial transfer rates are updated based on the predefined daily motion parameters.

2. **Material Redistribution:**

   - Fuel compositions and burnup values are extracted for each zone.
   - Motion matrices dictate the fraction of material transported from adjacent zones.
   - A fraction of fresh fuel is added, while spent fuel exits the core based on the number of accumulated passes.
   - The new material composition and burnup values are assigned back to the simulation.

3. **Activation Step:** Using the newly updated material distributions, a depletion step is performed over a time interval corresponding to the motion step. This accounts for neutron interactions leading to transmutation and fission product buildup. Activation means that the flux is not updated after motion.

4. **Tracking and Visualization:**

   - The discarded fuel compositions and burnup levels are stored.
   - The burnup distribution across radial zones is plotted to monitor depletion behavior.
   - The top 20 isotopes in discarded fuel are identified and plotted.

5. **Inner Convergence Test:**

   - The relative variation of the discarded burnup in the last pass radial zones is computed.
   - If the norm of this variation remains below a predefined threshold (ϵ) over several substeps, equilibrium is declared.
   - Upon convergence, the current material state is saved, and the inner loop exits.

4.3 Equilibrium and Simulation Termination

Once both the inner and outer loops converge, the system has reached a steady-state condition where the fuel motion and neutron flux distributions are self-consistent. The final results, including k_{eff}, flux distributions, and burnup profiles, are saved for post-processing.

This nested loop approach ensures an accurate and self-consistent treatment of neutron transport, depletion, and fuel motion, making it a robust framework for simulating the dynamic behavior of a pebble bed reactor. You can find in Figure 1 .

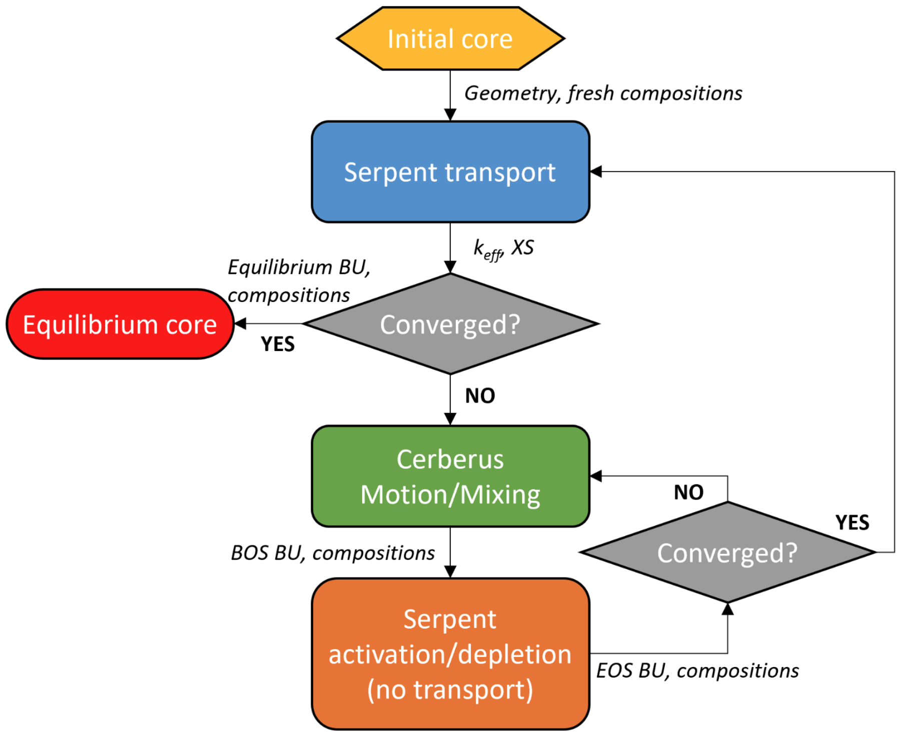

Figure 1: Search-Equilibrium Workflow summary.

4.4 Input Parameters

This workflow relies on a series of parameters that are manually defined in `search_equilibrium.py`, affecting the transport-depletion process, fuel motion, and convergence criteria. Below, we highlight the most significant parameters that influence the overall workflow.

4.4.1 Reactor and Zone Configuration

The reactor domain is divided into spatial zones, where neutron transport and fuel depletion calculations are performed. The key parameters governing the spatial discretization are:

- **Number of passes** (N_{passes}): Number of times pebbles circulate through the core before discharge.
- **Radial and axial divisions**:
  - $N_{\text{zones, radial}}$: Number of radial zones.
  - $N_{\text{zones, axial}}$: Number of axial zones.
  - *Bounds*: The spatial limits of each region in cm.

4.4.2 Fuel and Material Properties

Each material type in the simulation is defined by:

- **Pebble compositions**: Different pebble types (e.g., fuel and graphite).
- **Pebble fractions** (f_i): The fraction of each pebble type in the core.
- **Burnable materials**: Specifies whether a material undergoes depletion.
- **Material volumes**: The volume of each material type in a single pebble.

Basically, at the beginning of the simulation, the simulator will generate pebble composition based on their type and fraction.

4.4.3 Simulation Control

To balance computational efficiency and accuracy, the transport simulation is controlled by:

- **Neutron population per cycle** (N_{init}): Initial number of neutrons per cycle.
- **Multiplication factor** ($\lambda_{\text{neutrons}}$): Scaling factor for neutron population after each cycle.
- **Maximum neutron limit** (N_{max}): Upper bound on neutron count per cycle.
- **Motion step initialization** (Δt_{ini}): Initial time step for pebble movement in seconds.
- **Motion step refinement factor** (λ_{step}): Factor for reducing time step at each iteration.

4.4.4 Pebble Motion and Mixing

Fuel motion and material redistribution are modeled with:

- **Residence time** (t_{res}): Total time pebbles remain in the core across all passes.
- **Zone residence time** (t_{zone}): Time spent in each axial zone.
- **Axial transfer rate** (T_{axial}): Fraction of pebbles moving between axial zones per day.
- **Radial transfer rate** (T_{radial}): Fraction of pebbles moving radially per day.

4.4.5 Convergence Criteria

The simulation iterates over both inner (fuel motion) and outer (transport-depletion) loops, converging based on:

- **Inner loop convergence threshold** (ϵ): Criterion for sub-step convergence.
- **Minimum substeps** ($N_{\text{substeps, min}}$): Ensures a minimum number of mixing steps before exiting the inner loop.
- **Outer loop keff convergence threshold** ($\epsilon_{k_{\text{eff}}}$): Tolerance for overall system equilibrium.

4.4.6 Communication and Debugging

To control verbosity and real-time monitoring:

- **Verbosity level**: Controls the amount of output logged.
- **Plot frequency**: Specifies how often zone-wise data is visualized during the simulation.

These parameters play a fundamental role in defining the simulation workflow, affecting the accuracy and computational cost of the pebble-bed reactor simulation. A detailed breakdown of each individual parameter and its role will be provided in a subsequent section.

5 Setup and Environment Configuration

5.1 Setting Up Cerberus: The `setup_Cerberus` Script

The `setup_Cerberus` script is a crucial component for configuring the environment required to run the simulation tool. It ensures that all necessary dependencies are installed and properly configured before executing any simulations. This script should be executed once before using the tool for the first time.

5.1.1 Purpose and Impact

The script automates the setup of the computing environment, handling:

- Unloading conflicting software modules and loading required ones.
- Installing and configuring Miniconda if it is not already present.
- Creating a dedicated Conda environment for running the simulation.
- Installing necessary Python packages and dependencies.
- Ensuring compatibility with Cerberus and KrakenTools frameworks.

5.1.2 Step-by-Step Breakdown

The script follows these main steps:

1. Module Management To prevent software conflicts, the script first unloads any existing Python, GCC, CMake, and Nano modules before loading the correct versions required for the simulation.

2. Miniconda Installation If Miniconda is not found in the user's home directory, the script downloads and installs it. This ensures a controlled Python environment without affecting system-wide installations.

3. Creating a Custom Conda Environment The script configures a Conda environment named `kraken`, placing it in the scratch directory to optimize storage usage. It also configures Conda to store package installations in a designated location.

4. Installing Dependencies Within the new Conda environment, the script installs essential Python libraries such as:

- `serpentTools` for processing Serpent Monte Carlo outputs.
- `numpy, scipy, pandas` for numerical and data manipulation.
- `matplotlib, seaborn` for visualization.
- `mpi4py` for parallel computing.

5. Linking Cerberus and KrakenTools To ensure proper integration with the simulation framework, the script links the Cerberus and KrakenTools directories to the Conda environment.

6. Verification Finally, the script deactivates and reactivates the environment to confirm that Cerberus is correctly installed by running a simple import test.

5.1.3 Execution

To set up the environment, run:

```
bash ./Utils/setup_Cerberus
```

This should only be run once before using the simulation tool. If any updates are made to the environment, re-executing the script ensures a clean and correct setup.

5.2 Activating Kraken: The `sss_environment` Script

The `sss_environment` helps you to load the correct simulation for your coming simulation, export the good serpent path, and activate kraken, the environment used to run the searc_equilibrium tool.

6 Execution Workflow

The simulation follows a structured workflow:

1. Prepare input files in the `Inputs/` directory, ensuring correct reactor configuration and material properties.
2. Update the inputa parameters in `search_equilibrium.py`.
3. Run the code `search_equilibrium.py`.
4. Extract results from the `Outputs/` directory for your specific case, analyzing neutron flux, fuel burnup, and material distributions.

6.0.1 Execution

To set up the environment, run:

```
source ./Utils/sss_environment kraken
```

This should only be run once before using the simulation tool. If any updates are made to the environment, re-executing the script ensures a clean and correct setup.

6.1 Testing the environment and Kraken activation

After sourcing the `sss_environment` script, you should be able to do a few checks. The first one will be to see what the current path related to the accessible Python module:

```
[ludovicjantzen@ln002 search_equilibrium]$ which python
/global/scratch/users/ludovicjantzen/conda/kraken/bin/python
```

Then by running accessing the python through the Kraken environment:

```
[ludovicjantzen@ln002 search_equilibrium]$ python
Python 3.11.11 | packaged by conda-forge |
(main, Mar  3 2025, 20:43:55) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cerberus
>>> cerberus
<module 'cerberus' from '/global/home/groups/co_nuclear/
HxF_tools/Cerberus_HxF/cerberus/__init__.py'>
>>>
```

You should be able to import Cerberus and access its location without error. This means that your Kraken environment has been set up correctly.

A Detailed Input Parameters

This appendix provides a comprehensive description of all input parameters used in the simulation framework. Each parameter is categorized based on its function within the simulation.

A.1 File and Case Configuration

- **case_name** (string): Name of the simulation case; defines the folder where results are stored.
- **main_input_file** (string): Name of the primary Serpent input file. This file must be located in the same directory as the script.
- **pos_file** (string): Name of the pebble position file, required for spatial tracking of pebbles.

A.2 Reactor Zone Configuration

- **N_passes** (integer): Number of pebbles passes through the core before discharge.
- **division** (dictionary): Defines radial and axial spatial zoning.
  - **Nzones** (integer): Number of subdivisions in the specified direction.
  - **bounds** (list): Defines the spatial limits for each region in cm.
- **N_radial** (integer): Number of radial zones, extracted from `division['radial']['Nzones']`.
- **N_axial** (integer): Number of axial zones, extracted from `division['axial']['Nzones']`.

A.3 Fuel and Material Definition

- **materials** (dictionary): Maps pebble material names to their respective Serpent universe identifiers.
- **fracs** (list): Fractional distribution of different pebble types in the pebble bed.
- **is_fuel** (list): Boolean flag indicating whether each pebble type undergoes depletion.
- **vol_per_pebble** (list): Volume of material per pebble type, used for normalization.
- **pbed_universe** (string): Name of the Serpent universe corresponding to the pebble bed.
- **N_types** (integer): Number of distinct pebble types, derived from the `materials` dictionary.
- **material_names** (list): Names of each material type in the simulation.

A.4 Transport and Simulation Control

- **sss_exe** (string): Path to the Serpent executable.
- **MPI_mode** (boolean): Flag indicating whether to use MPI for parallel execution.
- **ncores** (integer): Number of OpenMP cores used for simulation. So far this option does not work because of mpirun to spawn process on the cluster.
- **ini_neutrons** (integer): Initial number of neutrons per cycle.
- **neutrons_multiplier** (float): Multiplicative factor applied to the neutron population per cycle.
- **max_neutrons** (integer): Upper limit for neutron count in a cycle.

A.5 Pebble Motion and Mixing

- **res_time** (float, days): Total residence time of pebbles in the core overall passes.
- **t_zone** (float, days): Time pebbles spend in each axial zone, computed as:
$$t_{\text{zone}} = \frac{t_{\text{res}}}{N_{\text{passes}} N_{\text{axial}}} \tag{1}$$
- **uniform_axial_transfer_per_day** (list): Fraction of pebbles transferring to the next axial zone per day.
- **uniform_radial_transfer_per_day** (list): Fraction of pebbles transferring radially per day to model cross-mixing.
- **ini_motion_step** (float, seconds): Initial mixing and motion time step, calculated as:
$$\Delta t_{\text{ini}} = t_{\text{zone}} \times 86400 \tag{2}$$
- **step_multiplier** (float): Multiplicative factor for reducing time step per iteration.

A.6 Convergence Criteria

- **eps** (float): Convergence criterion for inner loop substeps.
- **mini_substeps** (integer): Minimum number of inner loop substeps before exiting.
- **n_converged_substeps** (integer): Number of consecutive substeps satisfying the convergence criterion before exiting.
- **eps_keff** (float): Convergence criterion for the outer loop based on k_{eff}.

A.7 Communication and Monitoring

- **verbosity_mode** (integer): Defines the verbosity level of logging output.
- **plot_every** (integer): Frequency of plotting zone-wise simulation data.

This detailed parameter breakdown ensures clarity on each variable's function and its role in controlling the reactor simulation.

B SLURM Script for Parallel Execution

B.1 Single-Core Execution

The following SLURM script runs a single-core job on Savio. Since this script is not parallelized, it will only utilize one core, making it inefficient if a full node is allocated.

```
#!/bin/bash
#SBATCH --job-name=single_core_job
#SBATCH --partition=savio3_htc
#SBATCH --account=co_nuclear
#SBATCH --ntasks=1
#SBATCH --qos=savio_lowprio
#SBATCH --time=02:00:00
#SBATCH --output=job_output.txt

module load python/3.11.6-gcc-11.4.0
python search_equilibrium.py
```

Since this code is not parallelized, it is inefficient to book an entire node when only one core is being used. Instead, we should run multiple independent single-core jobs in parallel. This can be achieved using `mpirun`.

B.2 Parallel Execution with MPI

To take advantage of cluster resources, we can launch multiple single-core instances of the Python script in parallel using `mpirun`. The following script allocates an entire node and runs multiple independent jobs using MPI.

```
#!/bin/bash
#SBATCH --job-name=parallel_jobs
#SBATCH --partition=savio3_htc
#SBATCH --account=co_nuclear
#SBATCH --nodes=1
#SBATCH --ntasks=20  # Number of parallel single-core jobs
#SBATCH --cpus-per-task=1
#SBATCH --qos=savio_lowprio
#SBATCH --time=02:00:00
#SBATCH --output=parallel_output_%j.txt
```

```
#SBATCH --error=parallel_error_%j.txt

module load python/3.11.6-gcc-11.4.0
module load mpi  # Load MPI module if required

# Run multiple independent jobs in parallel using mpirun
mpirun -np $SLURM_NTASKS python search_equilibrium.py
```

B.3 Example: Parallel Execution with `mpi4py`

The following Python script demonstrates how to use `mpi4py` to distribute computations across multiple processes:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()  # Get current process ID
size = comm.Get_size()  # Get total number of processes

def run_simulation(process_id):
    """Example function representing a simulation task."""
    print(f"Process␣{process_id}␣is␣running␣a␣simulation.")

# Distribute workload among available processes
run_simulation(rank)

comm.Barrier()  # Ensure all processes finish before exiting
```

B.4 Explanation

- `--ntasks=20`: Runs 20 independent Python instances in parallel.
- `mpirun -np $SLURM_NTASKS python search_equilibrium.py`: Ensures multiple instances of `search_equilibrium.py` run simultaneously.
- `--nodes=1`: Allocates a single node.
- `--cpus-per-task=1`: Each job uses one core.
- `--output=parallel_output_%j.txt`: Outputs results for each SLURM job.
- `module load mpi`: Ensures MPI is available for `mpirun`.

The `mpi4py` script above distributes the workload across multiple processes, ensuring that each core runs an independent simulation. This parallelization significantly reduces the overall computational time compared to a single-core execution.

References

[1] Yves Robert, Tatiana Siaraferas, and Massimiliano Fratoni. Hyper-fidelity depletion with discrete motion for pebble bed reactors. *Scientific Reports*, 13(1), August 2023.