Table of Contents

Chapter 1

REALM: Reactor Evolutionary Algorithm Optimizer

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

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

- Effective: REALM is well documented, well tested, and version-controlled on Github [1].

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

- Open-source: I utilized a well-documented open-source evolutionary algorithm (EA) Python package to drive the genetic algorithm optimization. I utilized established open-source nuclear transport, OpenMC [7], and thermal-hydraulics, Moltres [5], software to compute the objective function and constraints. I also provide a simple tutorial for future developers to follow for coupling other nuclear software to the REALM package.

- Parallel: REALM runs parallel on high-performance computing (HPC) machines using the `multiprocessing_on_dill` Python package [8].

- Reproducible: Data from every REALM run saves into a unique pickled file, and all results from this work are available on Github.

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

REALM essentially couples an evolutionary algorithm driver with nuclear software such as nuclear transport and thermal-hydraulics codes. Figure **??** from Chapter **??** outlined an evolutionary algorithm's iterative problem solving process. I modified Figure **??** to produce Figure 1.1, which depicts how the nuclear transport and thermal-hydraulics software fit within the process. Therefore, REALM will initially read and validate the JSON input file,

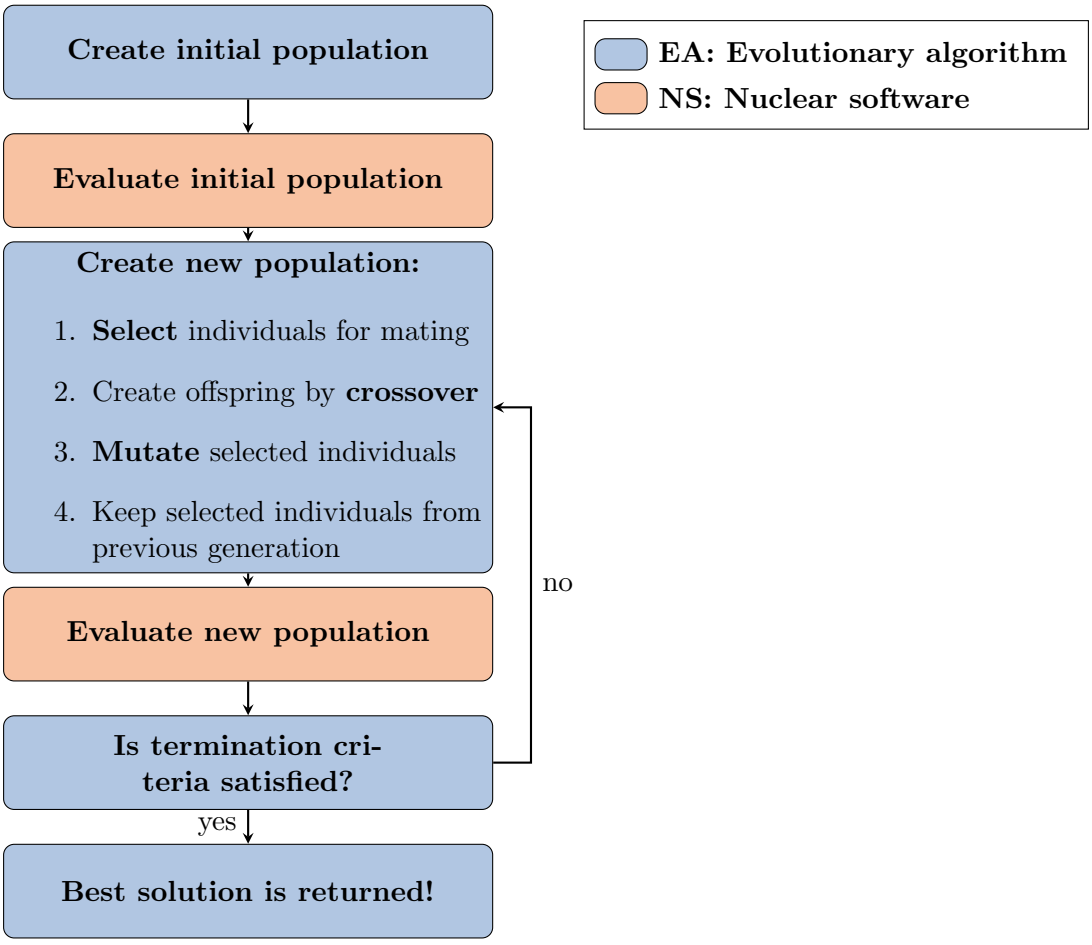

Figure 1.1: Process of finding optimal solutions for a problem with a genetic algorithm with evaluation conducted with nuclear software.

initialize the Distributed Evolutionary Algorithms in Python (DEAP) genetic algorithm hyperparameters and operators, and finally run the genetic algorithm following the flow chart in Figure 1.1 in which the nuclear software evaluates each individual's fitness.

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

1.1 Evolutionary Algorithm Driver

Evolutionary algorithm computation is a sophisticated field with diverse techniques and mechanisms, resulting in even well-designed coded-up frameworks being complicated under the hood. Therefore, utilizing a previously used evolutionary algorithm framework brings up issues in extending implementation intricacies as the user has to edit the source code [2]. Therefore, an evolutionary algorithm computation framework that gives the user the capability to build custom evolutionary algorithms is ideal for this project.

There are many evolutionary algorithm computation packages available: DEAP [2], inspyred [4], Pyevolve [6], and OpenBEAGLE [3]. DEAP is the most newly created package and places a high value on code compactness and code clarity [2]. DEAP is the only framework that allows the user to prototype evolutionary algorithms rapidly and define custom algorithms without digging deep into the source code to modify lines, making it the code of choice for the REALM framework's evolutionary algorithm driver component. . DEAP provides building blocks for each optimizer function and allows the user to customize and design a specialized algorithm to fit their project [2].

1.1.1 Distributed Evolutionary Algorithms in Python

DEAP is composed of two simples structures: a *creator* and a *toolbox*. The *creator* module is a meta-factory that allows the run-time creation of classes via inheritance and compo-

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

Figure 1.2: DEAP sample code demonstrating the usage of the *creator* and *toolbox* modules to initialize the genetic algorithm. In REALM, DEAP's *creator* and *toolbox* modules are initialized in the source code based on the genetic algorithm parameters defined by the user in the REALM input file.

sition, enabling individuals and populations' creation from from any data structure: lists, sets, dictionaries, trees, etc [2]. The *toolbox* is a container for the tools/operators that the user manually populates with selected tools for the evolutionary algorithm. The *toolbox* is where the user would define the selection, crossover, and mutation operator types and hyperparameters. For example, the user would register a crossover operator under the 'mate' alias. Then, the evolutionary algorithms would be built with these aliased operators, and if the user wanted to change the crossover operator, they would update the 'mate' alias in the toolbox, and the algorithm would remain unchanged [2].

Figure 1.2 illustrates DEAP's usage of the *creator* and *toolbox* modules. On line 2, a single-objective fitness class `Objective` is created. The first argument defines the name of the derived class, the second argument specifies the inherited base class (`base.fitness`), and the third argument is the weights attribute initialized with a tuple that indicates one-objective

fitness (−1.0 indicates a minimum objective, and +1.0 indicates a maximum objective). On line 3, an `Individual` class is derived from the Python list and composed with our newly created `Objective` object. On lines 5-9, we initialize the DEAP toolbox, register `variable_1` and `variable_2` with their upper and lower bounds, and defined the `individual_creator` function to return an `Individual` initialized with `variable_1`, and `variable_2`. Lines 10-11 and 14-17 have aliases for initializing individuals and population, specifying variation operators (`select`, `mutate`, `mate`), and evaluating individual fitness (`evaluate`) [2]. Lines 12-13 define the evaluation function that returns the fitness values.

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

1.1.2 General Genetic Algorithm Framework

The creators' of DEAP provided an example of a classical genetic algorithm exposing different explicitness levels [2]. The high-level examples use the in-built DEAP genetic algorithms, whereas the low-level example completely unpacks the genetic algorithm to expose a generational loop. The general genetic algorithm included in the ***Algorithm*** class is based on the low-level example. The algorithm begins by initializing the starting population and evaluating each individual's fitness value; then, it enters a generational loop. During each iteration, selection, mating, and mutation operators are applied to the population, then the new individuals are evaluated, the constraints are applied, and the results are saved.

1.2 Nuclear Software

Many nuclear software have restricted public access. In the proposed work, I enabled REALM to work with open-source nuclear transport and thermal-hydraulics software, OpenMC [7] and Moltres [5]. OpenMC is an open-source Monte Carlo neutron transport code capa-

ble of performing k-eigenvalue calculations on models built using either constructive solid geometry or CAD representation. OpenMC can run in parallel using a hybrid MPI and OpenMP programming model. Moltres is an open-source tool designed to simulate Molten Salt Reactors (MSRs) using deterministic neutronics and thermal-hydraulics, implemented as an application atop the Multiphysics Object-Oriented Simulation Environment (MOOSE) finite-element framework. Moltres solves arbitrary-group neutron diffusion, temperature, and precursor governing equations on a single mesh and can be deployed on an arbitrary number of processing units [5].

OpenMC and Moltres are both open-source, well-documented, well-supported, and Github version-controlled codes that run in parallel on HPC machines, thus, achieving the REALM goals listed at the start of this chapter, making them suitable to be used as REALM's nuclear software. However, REALM users can easily use restricted nuclear software with REALM by using REALM with the restricted software on their local machine. In the REALM documentation [1], I outline how to couple other nuclear software to REALM.

1.3 REALM Input File

REALM's input file is in JSON format. There are four sections that the user must define: `control_variables`, `evaluators`, `constraints`, and `algorithm`. Figure 1.3 shows an example REALM input file. Next, we will describe how to define each section of a REALM input file. Detailed descriptions of setting up REALM can be found in the REALM documentation [1].

1.3.1 Control Variables

The control variables are parameters that the user wants the genetic algorithm to vary. For each control variable, the user must specify the minimum and maximum values (lower and upper bounds). For example, lines 2 to 5 in Figure 1.3 demonstrate that the control

```
{
    "control_variables": {
        "variable1": {"min": 0.0, "max": 10.0},
        "variable2": {"min": -1.0, "max": 0.0}
    },
    "evaluators": {
        "openmc": {
            "input_script": "openmc_inp.py",
            "output_script": "openmc_output.py",
            "inputs": ["variable1", "variable2"],
            "outputs": ["output1", "output2"]
        }
    },
    "constraints": {
        "output_1": {"operator": [">=", "<"], "constrained_val": [1.0, 1.5]}
    },
    "algorithm": {
        "objective": "min",
        "optimized_variable": "output1",
        "pop_size": 100,
        "generations": 10,
        "mutation_probability": 0.5,
        "mating_probability": 0.5,
        "selection_operator": {"operator": "selBest", "k": 1},
        "mutation_operator": {
            "operator": "mutPolynomialBounded",
            "indpb": 0.5,
            "eta": 0.5
        },
        "mating_operator": {"operator": "cxOnePoint"}
    }
}
```

Figure 1.3: REALM sample input file.

```
1  import openmc
2  # templating
3  variable1 = {{variable1}}
4  variable2 = {{variable2}}
5  # run openmc
6  ...
```

```
1  import openmc
2  # templating
3  variable1 = 3.212
4  variable2 = -0.765
5  # run openmc
6  ...
```

Figure 1.4: `openmc_inp.py` input script template (left). Templated `openmc_inp.py` with variable1 and variable2 values defined (right).

variables, `variable1` and `variable2`, can be varied from 0 to 10 and -1 to 0 respectively.

1.3.2 Evaluators

Evaluators are the software REALM utilizes to calculate objective functions. Presently, only `openmc` and `moltres` evaluators are available in REALM. In a single REALM input file, a user may define any number of evaluators. For each evaluator, mandatory input parameters are `input_script`, `inputs`, and `outputs`, and the optional input parameters is `output_script`. The `input_script` is an input file template for the evaluator software. The `inputs` parameter lists the control variables that are placed into the input file template. REALM utilizes jinja2 templating to insert the control variable values into the `input_script`. Lines 6 to 12 in the REALM input file (Figure 1.3) demonstrate that `variable1` and `variable2` are `inputs` into the `openmc_inp.py input_script`. Figure 1.4 shows the template and templated openmc script; once the `openmc_inp.py input_script` is templated, {{`variable1`}} and {{`variable2`}} on lines 3 and 4 will be replaced with values selected by the REALM genetic algorithm.

The `outputs` parameters lists the output variables that the user wants to return to the genetic algorithm from the evaluator. These output parameters are also known as the objective functions used to evaluate the individual. There are three methods to returning an output parameter. First, if the output parameter is also an input parameter, REALM will automatically return the input parameter's value. Second, the user can use predefine

evaluations. For example, we have predefined a keff evaluation for OpenMC. The user may also add predefine evaluations to `OpenMCEvaluation` or `MoltresEvaluation`, or any other coupled codes' evaluation file. Third, the user may include an `output_script` that returns the desired output parameters. The `output_script` must include a line that prints a dictionary containing the output parameters' names and their corresponding value as key/value pairs.

1.3.3 Constraints

In constraints, the user can choose to constrain any output parameter. Any individual that does not meet the constraints will be removed from the population, encouraging the proliferation of individuals that meet the constraints. For each constrained output parameter, the user lists the `operator`s and `constrained_val`s as in line 15 of the REALM input file (Figure 1.3). Thus, for this REALM simulation, `output_1` is constrained to be >= 1.0 and < 1.5.

1.3.4 Algorithm

In the algorithm section, the user defines all the hyperparameters for the genetic algorithm. The mandatory input parameters include `optimized_variable`, `objective`, `pop_size`, and `generations`. The user specifies an `optimized_variable` which must be an output parameter from the evaluators' `outputs`. The user has the option to maximize or minimize this `optimized_variable` by defining the `objective` variable as `max` or `min`. The user must also specify the population size (`pop_size`) and no. of generations (`generations`) in the genetic algorithm.

The optional input parameters include `mutation_probability`, `mating_probability`, `selection_operator`, `mutation_operator`, and `mating_operator`. As mentioned previously in section **??**, it is important to select genetic algorithm hyperparameters that balance

Table 1.1: Selection, mutation, and mating operators available in REALM and their corresponding hyperparameters.

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

the extent of exploration and exploitation. The user can define the mutation and mating probability or use default values of 0.3 and 0.4, respectively. For each operator, the user can choose from a list of operators and define the hyperparameters required for them. Table 1.1 shows the available operators and their respective hyperparameters. The default selection operator is `selNSGA2`, with `k` being two-thirds of the population size. The default mutation operator is `mutPolynomialBounded` with an `eta` of 0.3 and `indpb` of 0.3. The default mating operator is `cxBlend` with an `alpha` of 0.4. Lines 17 to 31 in the example REALM input file (Figure 1.3) demonstrate `algorithm` specifications.

1.4 REALM Software Architecture

In this section, I will describe REALM v1.0's software architecture and how all the parts come together to meet the goal of optimizing reactor design. Table 1.2 outlines the classes in the REALM software and describes each class's purpose. Figure 1.5 depicts REALM's software architecture. When the user runs a REALM input file, the ***Executor*** class drives REALM's execution from beginning to end. The ***Executor*** calls ***InputValidation*** to parse the input file to ensure that the user defined all mandatory parameters and used

Table 1.2: Classes that makeup REALM's architecture and their description.

Class	Description
InputValidation	The ***InputValidation*** class contains methods to read the JSON REALM input file and conduct a validation to ensure the user defined all key parameters, and if they did not, REALM raises an exception to tell the user which parameters are missing.
Evaluation	DEAP's fitness evaluator (as mentioned in Section 1.1.1) requires an evaluation function to evaluate each individual's fitness values. The ***Evaluation*** class contains a method that returns an evaluation function that runs the nuclear software and returns the fitness values listed in the user input file.
OpenMCEvaluation	The ***OpenMCEvaluation*** class contains built-in methods for evaluating OpenMC output files. Developers can update this file with methods to evaluate frequently used OpenMC outputs.
ToolboxGenerator	The ***ToolboxGenerator*** class initializes DEAP's *toolbox* and *creator* modules with genetic algorithm hyperparameters defined in the user input file.
Constraints	The ***Constraints*** class contains methods to initialize constraints defined in the user input file and apply the constraints by removing individuals that do not meet the constraint.
BackEnd	The ***BackEnd*** class contains methods to save genetic algorithm population results into a pickled checkpoint file and to restart a partially completed genetic algorithm from the checkpoint file.
Algorithm	The ***Algorithm*** class contains methods to initialize and execute the genetic algorithm. It executes a general genetic algorithm framework that uses the hyperparameters defined in the ***ToolboxGenerator***, applies constraints defined in ***Constraints***, evaluates fitness values using the evaluation function produced by ***Evaluation***, and saves all the results with ***BackEnd***.
Executor	The ***Executor*** class drives the REALM code execution with the following steps: 1) User input file validation with ***InputValidation***. 2) Evaluation function generation with ***Evaluation***. 3) DEAP toolbox initialization with ***ToolboxGenerator***. 4) Constraint initialization with ***Constraints***. 5) Genetic algorithm execution with ***Algorithm***.

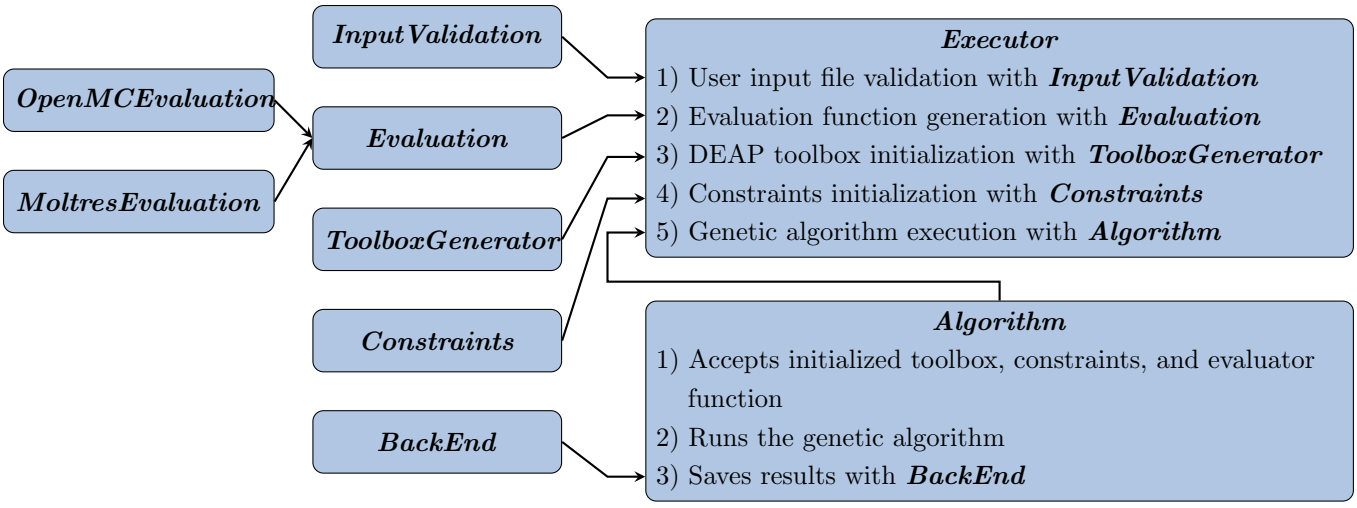

Figure 1.5: Visualization of REALM architecture.

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

In the REALM Github repository [1], I included a tests directory that contains unit tests for all methods in the classes described above.

1.4.1 Installing and Running REALM

There are two ways to install REALM. First, a user can utilize The Python Package Index (PyPI) to install REALM: `pip install realm`. Second, a user can download the REALM Github repository [1] and install it from source.

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

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

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

1.4.2 REALM Results Analysis

The ***BackEnd*** class manages REALM's results. ***BackEnd*** puts all the results in a pickled dictionary (pickle is a Python module that serializes Python objects), and it is saved as `checkpoint.pkl` in the same directory as the input file. The checkpoint file can then be reloaded into a Jupyter notebook and organized to produce desired plots. Examples of REALM results analysis can be found in the REALM documentation [1].

For closer inspection of the evaluator software's output files, the evaluation function creates a new directory for each software, generation, and individual and stores the templated input file and output files associated with that particular run. The generation and individual values are indexed by zero. For example, the directory containing files associated with an OpenMC run for the tenth individual in the genetic algorithm's third generation will be named: `openmc_2_9`.

1.5 Summary

This chapter described the Reactor Evolutionary Algorithm Optimizer (REALM) framework developed for the proposed work; REALM is a Python package that applies evolutionary algorithm optimization techniques to nuclear reactor design using the Distributed Evolutionary Algorithms in Python (DEAP) module and OpenMC and Moltres nuclear software. The motivation for REALM's inception is to enable reactor designers to utilize robust evolutionary algorithm optimization methods without going through the cumbersome process of setting up a genetic algorithm framework, selecting appropriate hyperparameters, and setting up its parallelization. REALM is designed to be effective, flexible, open-source, parallel, reproducible, and usable and is hosted on Github [1].

References

[1] Gwendolyn Chee. arfc/realm, 2021.

[2] Flix-Antoine Fortin, Franois-Michel De Rainville, Marc-Andr Gardner, Marc Parizeau, and Christian Gagn. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13(Jul):2171–2175, 2012.

[3] Christian Gagn and Marc Parizeau. Open BEAGLE: A New Versatile C++ Framework for Evolutionary Computation. In *GECCO Late Breaking Papers*, pages 161–168. Citeseer, 2002.

[4] Aaron Garrett. inspyred: Bio-inspired Algorithms in Python. *URL: https://pypi. python. org/pypi/inspyred (visited on 11/28/2016)*, 2014.

[5] Alexander Lindsay, Gavin Ridley, Andrei Rykhlevskii, and Kathryn Huff. Introduction to Moltres: An application for simulation of Molten Salt Reactors. *Annals of Nuclear Energy*, 114:530–540, April 2018. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0306454917304760`, `doi:10.1016/j.anucene.2017.12.025`.

[6] Christian S. Perone. Pyevolve: a Python open-source framework for genetic algorithms. *Acm Sigevolution*, 4(1):12–20, 2009.

[7] Paul K. Romano and Benoit Forget. The OpenMC Monte Carlo particle transport code. *Annals of Nuclear Energy*, 51:274–281, January 2013. URL: `http://www.sciencedirect.com/science/article/pii/S0306454912003283`, `doi:10.1016/j.anucene.2012.06.040`.

[8] Robert Smallshire. multiprocessing_on_dill 3.5.0a4 : A friendly fork of multiprocessing which uses dill instead of pickle. URL: `https://github.com/sixty-north/multiprocessing_on_dill`.