User Manual of Probabilistic Memory for Intelligent Systems at Edge(PROMISE) V1

Developers: Likai Pei, Emilie Ye

PI: Prof. Ningyuan Cao, University of Notre Dame

Contents

1 Introduction

PROMISE is a comprehensive simulation and modeling framework developed to evaluate the system-level performance of probabilistic AI workloads running on novel reconfigurable probabilistic memory. Similar to how traditional AI accelerators face memory bottlenecks, probabilistic models such as Bayesian Neural Networks (BNNs), probabilistic decision trees, and graphical models also suffer from limited memory bandwidth—especially in edge environments. PROMISE addresses this by introducing a new class of memory that integrates native support for probabilistic computation while maintaining compatibility with existing compute architectures.

At the core of the PROMISE project is PROMISim, a fully Python-based simulation framework built under the PyTorch ecosystem, enabling co-simulation of algorithmic and hardware behaviors for probabilistic AI workloads. PROMISE consists of two main components: a user interface layer, which integrates with PyTorch for seamless model definition and execution; and an internal simulation engine, which models the behavior of reconfigurable probabilistic memory under various hardware configurations. This modular design allows users to easily evaluate architectures by specifying workload-level constraints and hardware parameters (e.g., memory structure, entropy source configuration, technology nodes). The simulator outputs key performance metrics such as inference accuracy, energy consumption, access latency, data retention, and system throughput.

The PROMISE simulator operates in three distinct modes: Demo Mode showcases the fundamental principles and behavior of probabilistic memory; Memory Mode performs detailed simulations of read and write operations, incorporating entropy sources and associated peripheral circuits; and Compute-In-Memory(CIM) Mode integrates probabilistic memory with neural networks to evaluate inference performance across different compute architectures, such as near-memory and in-memory computing. This modular design enables users to explore probabilistic memory from basic concept validation to full-system performance analysis.

PROMISE supports hierarchical modeling from device level (e.g., FeFET-based NAND/AND arrays, MRAM, ReRAM) to architecture level (e.g., near-memory vs. in-memory configurations, entropy-aware memory controllers), and system level (e.g., heterogeneous 2.5D/3D integration and probabilistic runtime adaptation). PROMISim integrates with a probabilistic programming model and compiler, enabling users to define probabilistic data types, memory-mapped inference operations, and runtime compiler optimizations directly in a high-level language.

By combining reconfigurable probabilistic memory with a unified software-hardware co-design approach, PROMISE facilitates robust and energy-efficient decision-making under uncertainty at the edge.

2 System Requirements

The PROMISE simulator is fully developed in Python and currently runs on Linux (Red Hat 8) as the primary development and testing environment. It relies on a set of Python libraries for numerical computation, deep learning integration (via PyTorch), and hardware modeling. Users are expected to have Python ($\geq$3.8) and relevant dependencies installed.

To improve portability and ease of deployment, future versions of PROMISE will be packaged using PyInstaller, allowing the simulator to be distributed as a standalone executable (.exe or equivalent) that can run on any major operating system without requiring manual Python environment setup.

3 Installation and Usage

This section will provide detailed information on the required dependencies and how to set up and run the PROMISE simulator manually if PyInstaller is not used.

Users will be guided through installing all necessary Python packages, configuring the environment, and executing the simulator under the native Python runtime. This includes step-by-step instructions for setting up the PyTorch-based simulation interface and launching different simulation modes (Demo, Memory, CIM).

4 Inteface

4.1 Mode Selection

Upon launching the simulator, the user is first presented with the default mode selection interface, as shown in Fig. 1. Users can select the desired simulation mode and click Next to proceed to the corresponding detailed interface. Each simulation interface also includes a Back button, allowing users to return to the mode selection screen and switch between different simulation modes as needed.

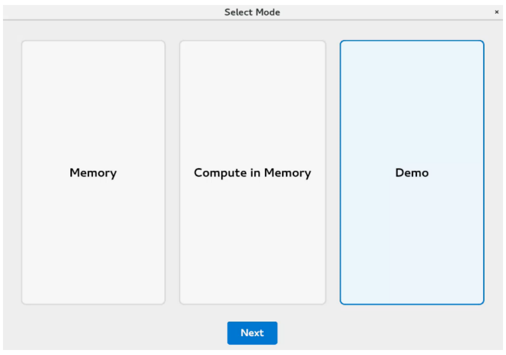
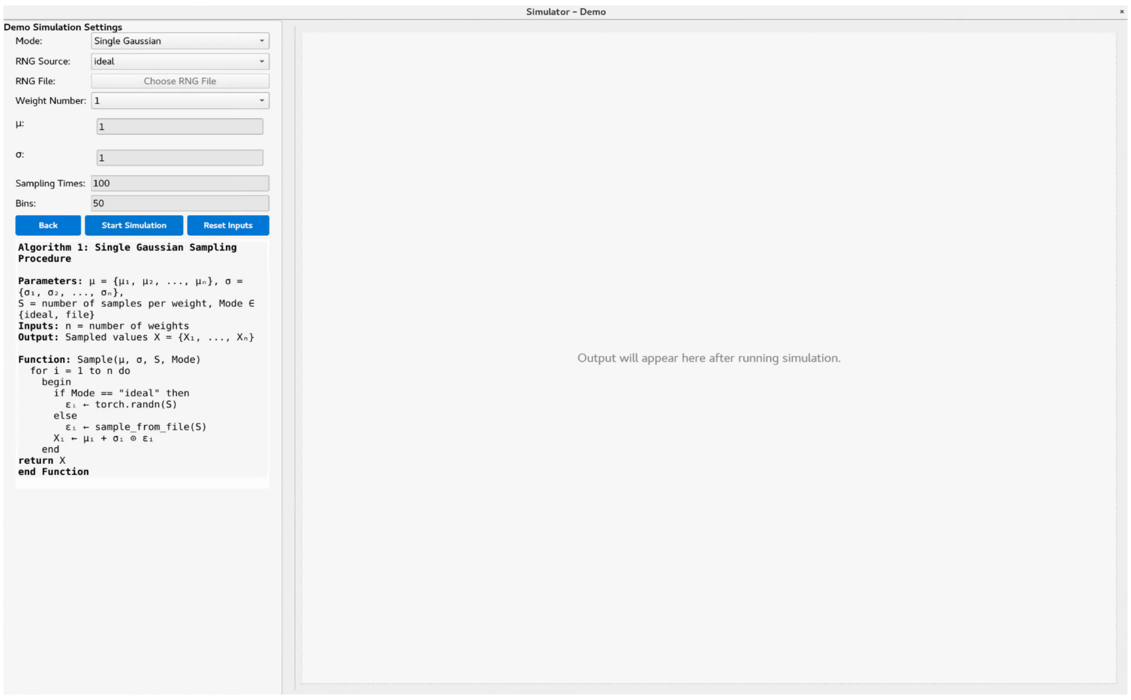


Figure 1: GUI interface for Mode Selection.

4.2 Demo Mode

The Demo Mode is primarily designed to demonstrate the fundamental working principles of probabilistic memory. In this mode, we assume only three memory cells for simplicity but provide two configuration options. The first is the Single Distribution mode, where users can select the number of weights to sample from a single Gaussian distribution. The second is the Gaussian Mixture Model (GMM) mode, where multiple distributions are combined into a GMM, and users can specify the number of distinct distributions.

4.2.1 Default



Demo Simulation Settings

Mode: Single Gaussian
RNG Source: ideal
RNG File:
Weight Number: 1
μ: 1
σ: 1
Sampling Times: 100
Bins: 50

```
Algorithm 1: Single Gaussian Sampling
Procedure

Parameters: $\mu = \{\mu_1, \mu_2, ..., \mu_n\}$, $\sigma =$
$\{\sigma_1, \sigma_2, ..., \sigma_n\}$,
S = number of samples per weight, Mode $\in$
{ideal, file}
Inputs: n = number of weights
Output: Sampled values $X = \{X_1, ..., X_n\}$

Function: Sample($\mu$, $\sigma$, S, Mode)
  for i = 1 to n do
    begin
      if Mode == "ideal" then
        $\varepsilon_i \leftarrow$ torch.randn(S)
      else
        $\varepsilon_i \leftarrow$ sample_from_file(S)
      $X_i \leftarrow \mu_i + \sigma_i \odot \varepsilon_i$
    end
return X
end Function
```

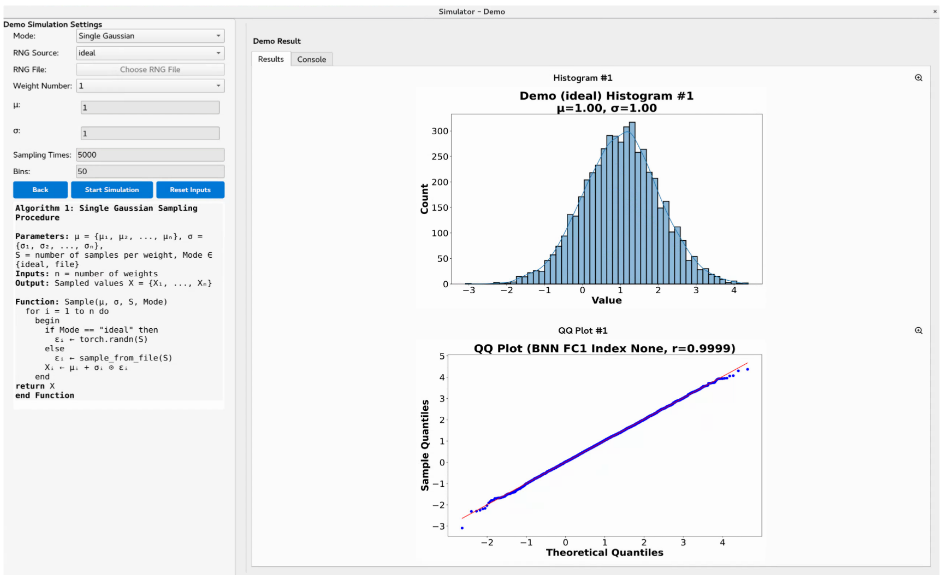
Output will appear here after running simulation.

Figure 2: Default GUI for Demo Mode.

Fig. 2 shows the default interface displayed after entering Demo Mode. The top-left panel allows users to input and select configuration parameters, while the right panel serves as the output display area. By default, the interface is set to the Single Gaussian mode and includes three buttons:

- **Back**: returns to the mode selection screen.
- **Start Simulation**: runs the simulation based on the selected parameters and displays the results on the right.
- **Reset Inputs**: resets all input fields to their default values.

4.2.2 Single Gaussian Distribution

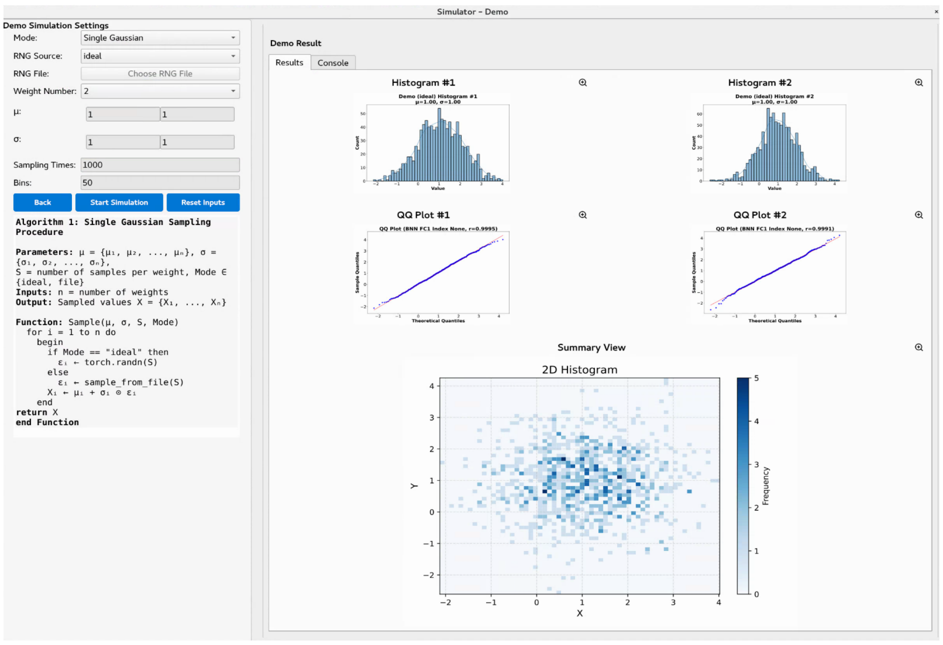

Simulator – Demo

Demo Simulation Settings

Mode: Single Gaussian
RNG Source: ideal
RNG File: Choose RNG File
Weight Number: 1
μ: 1
σ: 1
Sampling Times: 5000
Bins: 50

```
Algorithm 1: Single Gaussian Sampling
Procedure

Parameters: $\mu = \{\mu_1, \mu_2, ..., \mu_n\}$, $\sigma =$
$\{\sigma_1, \sigma_2, ..., \sigma_n\}$,
S = number of samples per weight, Mode ∈
{ideal, file}
Inputs: n = number of weights
Output: Sampled values $X = \{X_1, ..., X_n\}$

Function: Sample(μ, σ, S, Mode)
  for i = 1 to n do
    begin
      if Mode == "ideal" then
        $\varepsilon_i \leftarrow$ torch.randn(S)
      else
        $\varepsilon_i \leftarrow$ sample_from_file(S)
      $X_i \leftarrow \mu_i + \sigma_i \odot \varepsilon_i$
    end
return X
end Function
```

Figure 3: One Weight Results for Single Gaussian In the Demo Mode.


Simulator – Demo

Demo Simulation Settings

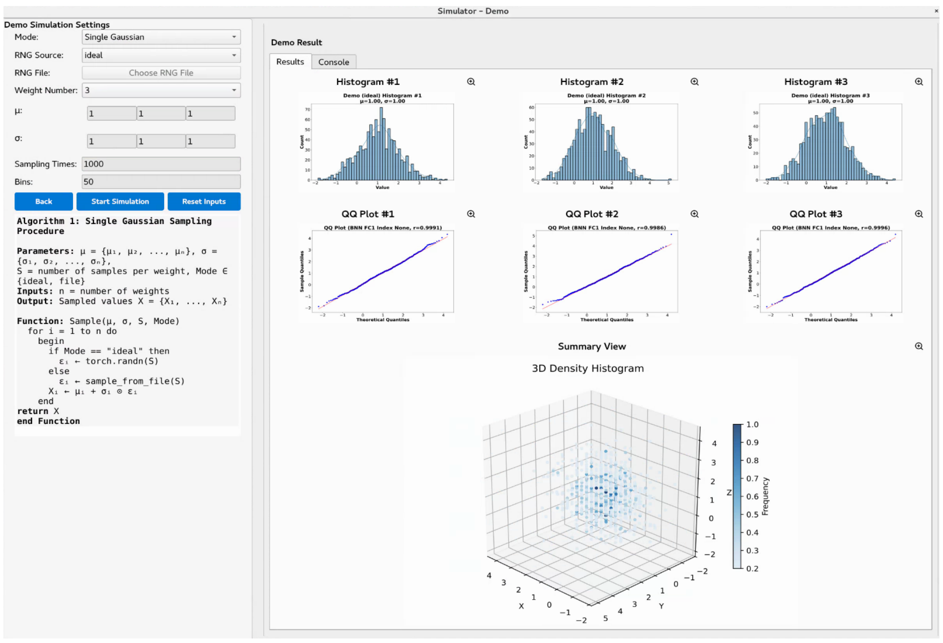
Mode: Single Gaussian
RNG Source: ideal
RNG File: Choose RNG File
Weight Number: 2
μ: 1 1
σ: 1 1
Sampling Times: 1000
Bins: 50

```
Algorithm 1: Single Gaussian Sampling
Procedure

Parameters: $\mu = \{\mu_1, \mu_2, ..., \mu_n\}$, $\sigma =$
$\{\sigma_1, \sigma_2, ..., \sigma_n\}$,
S = number of samples per weight, Mode ∈
{ideal, file}
Inputs: n = number of weights
Output: Sampled values $X = \{X_1, ..., X_n\}$

Function: Sample(μ, σ, S, Mode)
  for i = 1 to n do
    begin
      if Mode == "ideal" then
        $\varepsilon_i \leftarrow$ torch.randn(S)
      else
        $\varepsilon_i \leftarrow$ sample_from_file(S)
      $X_i \leftarrow \mu_i + \sigma_i \odot \varepsilon_i$
    end
return X
end Function
```

Figure 4: GUI interface for Mode Selection.

In Single Gaussian Mode, users can select the RNG source, which can be either an ideal generator or externally imported data. They can also specify the number of weights, and input the corresponding values for mean (μ) and standard deviation (σ). Additionally, users can configure the sampling time to perform multiple sampling iterations, and adjust the number of bins to generate a more detailed histogram of the resulting distribution.

Fig. 3 shows the interface after setting a single weight in the Single Gaussian mode. In the bottom-left corner, detailed pseudocode of the algorithm used to generate the probabilistic memory output is displayed. On the right side, the interface presents a histogram of the resulting Gaussian distribution along with a corresponding Q-Q plot for visualizing distribution normality.



Demo Simulation Settings
Mode: Single Gaussian
RNG Source: ideal
RNG File: Choose RNG File
Weight Number: 3
μ: 1 1 1
σ: 1 1 1
Sampling Times: 1000
Bins: 50

```
Algorithm 1: Single Gaussian Sampling
Procedure

Parameters: $\mu = \{\mu_1, \mu_2, \ldots, \mu_n\}$, $\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$,
S = number of samples per weight, Mode ∈ {ideal, file}
Inputs: n = number of weights
Output: Sampled values $X = \{X_1, \ldots, X_n\}$

Function: Sample(μ, σ, S, Mode)
  for i = 1 to n do
    begin
      if Mode == "ideal" then
        $\varepsilon_i \leftarrow$ torch.randn(S)
      else
        $\varepsilon_i \leftarrow$ sample_from_file(S)
      $X_i \leftarrow \mu_i + \sigma_i \odot \varepsilon_i$
    end
return X
end Function
```

Figure 5: GUI interface for Mode Selection.

Fig. 4 and Fig. 5 illustrate the cases with two and three weights, respectively. On the right, the results display the histogram and Q-Q plot for each individual distribution. Additionally, a 2D/3D histogram is included at the bottom, enabling users to better visualize the relationships and interactions between multiple random variables.

4.2.3 Gaussian Mixture Mode

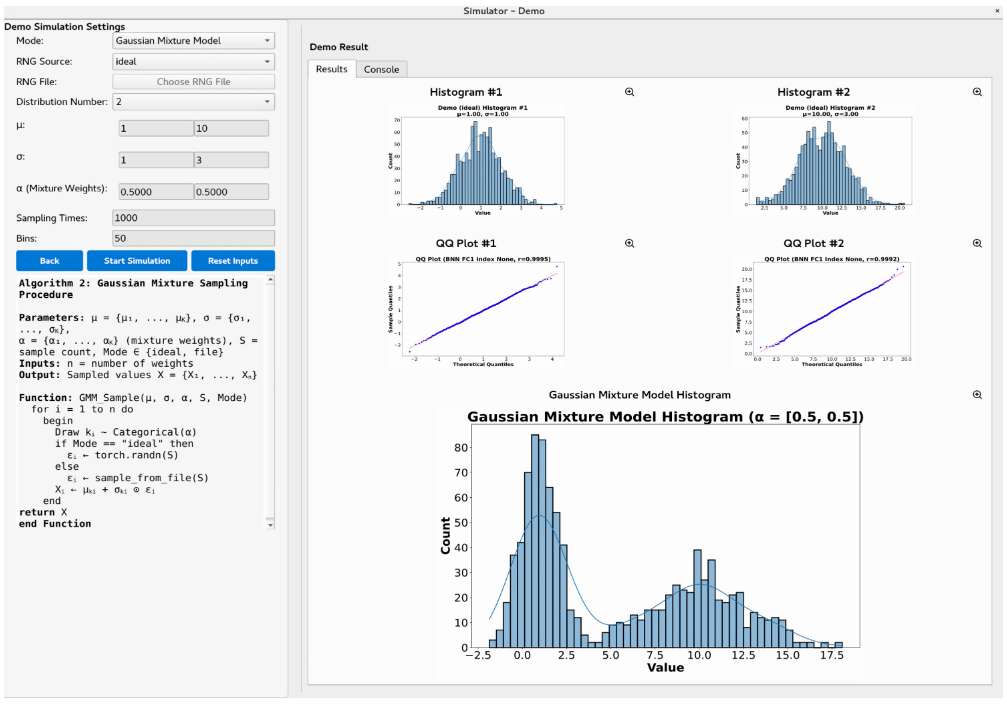
In GMM Mode, the probabilistic memory output is modeled as a Gaussian Mixture Model (GMM). The GMM is expressed as:

$$p(x) = \sum_{i=1}^{K} \alpha_i \cdot \mathcal{N}(x \mid \mu_i, \sigma_i^2) \tag{1}$$

where each component $\mathcal{N}(x \mid \mu_i, \sigma_i^2)$ is a Gaussian distribution with its own mean μ_i and variance σ_i^2, and α_i is the mixture weight associated with the i-th distribution, satisfying $\sum_{i=1}^{K} \alpha_i = 1$.

In this mode, users can configure all related parameters, including the number of distributions, their individual means and variances, and the corresponding mixture weights.

As shown in Fig. 6 and Fig. 7, the bottom-left corner of the input interface provides pseudocode for GMM sampling, while the output panel visualizes the resulting GMM histogram, clearly reflecting the combined effect of all component distributions.



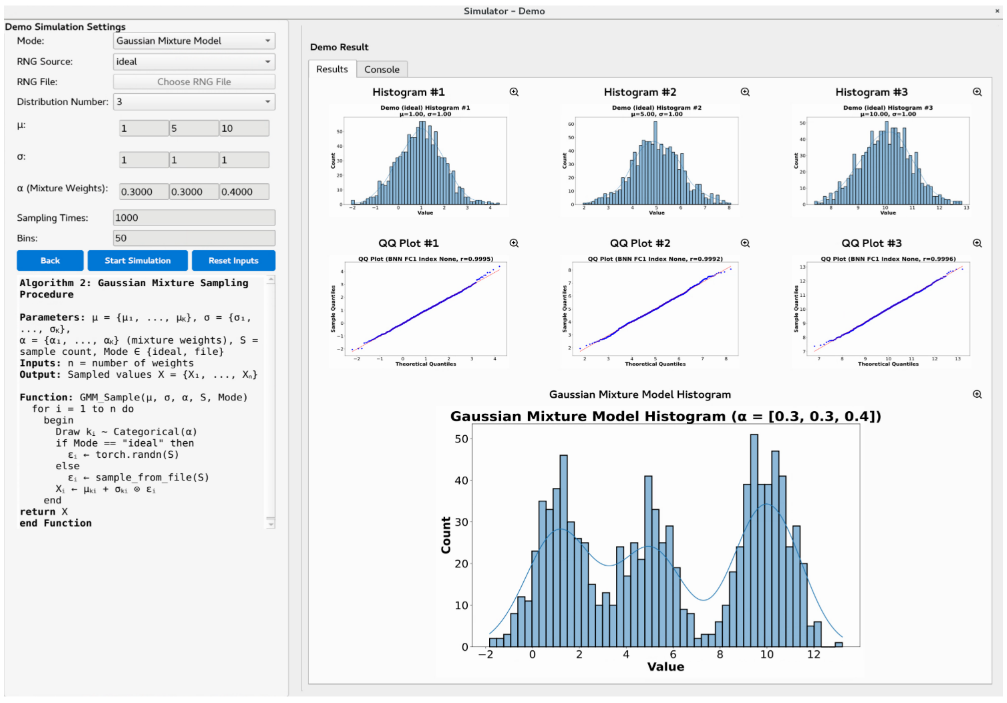
Simulator – Demo
Demo Result
Results
Console
Histogram #1
Demo (ideal) Histogram #1 μ=1.00, σ=1.00
Histogram #2
Demo (ideal) Histogram #2 μ=10.00, σ=3.00
QQ Plot #1
QQ Plot (BNN FC1 Index None, r=0.9995)
QQ Plot #2
QQ Plot (BNN FC1 Index None, r=0.9992)
Gaussian Mixture Model Histogram
Gaussian Mixture Model Histogram (α = [0.5, 0.5])
Count
Value
Back
Start Simulation
Reset Inputs

Demo Simulation Settings

Mode: Gaussian Mixture Model
RNG Source: ideal
RNG File: Choose RNG File
Distribution Number: 2
μ: 1 10
σ: 1 3
α (Mixture Weights): 0.5000 0.5000
Sampling Times: 1000
Bins: 50

```
Algorithm 2: Gaussian Mixture Sampling
Procedure

Parameters: $\mu = \{\mu_1, ..., \mu_K\}$, $\sigma = \{\sigma_1, ..., \sigma_K\}$,
$\alpha = \{\alpha_1, ..., \alpha_K\}$ (mixture weights), S =
sample count, Mode ∈ {ideal, file}
Inputs: n = number of weights
Output: Sampled values $X = \{X_1, ..., X_n\}$

Function: GMM_Sample(μ, σ, α, S, Mode)
  for i = 1 to n do
    begin
      Draw $k_i \sim$ Categorical(α)
      if Mode == "ideal" then
        $\varepsilon_i \leftarrow$ torch.randn(S)
      else
        $\varepsilon_i \leftarrow$ sample_from_file(S)
      $X_i \leftarrow \mu_{k_i} + \sigma_{k_i} \odot \varepsilon_i$
    end
return X
end Function
```

Figure 6: GUI interface for Mode Selection.



Simulator – Demo
Demo Result
Results
Console
Histogram #1
Demo (ideal) Histogram #1 μ=1.00, σ=1.00
Histogram #2
Demo (ideal) Histogram #2 μ=5.00, σ=1.00
Histogram #3
Demo (ideal) Histogram #3 μ=10.00, σ=1.00
QQ Plot #1
QQ Plot (BNN FC1 Index None, r=0.9995)
QQ Plot #2
QQ Plot (BNN FC1 Index None, r=0.9992)
QQ Plot #3
QQ Plot (BNN FC1 Index None, r=0.9996)
Gaussian Mixture Model Histogram
Gaussian Mixture Model Histogram (α = [0.3, 0.3, 0.4])
Count
Value
Back
Start Simulation
Reset Inputs

Demo Simulation Settings

Mode: Gaussian Mixture Model
RNG Source: ideal
RNG File: Choose RNG File
Distribution Number: 3
μ: 1 5 10
σ: 1 1 1
α (Mixture Weights): 0.3000 0.3000 0.4000
Sampling Times: 1000
Bins: 50

```
Algorithm 2: Gaussian Mixture Sampling
Procedure

Parameters: $\mu = \{\mu_1, ..., \mu_K\}$, $\sigma = \{\sigma_1, ..., \sigma_K\}$,
$\alpha = \{\alpha_1, ..., \alpha_K\}$ (mixture weights), S =
sample count, Mode ∈ {ideal, file}
Inputs: n = number of weights
Output: Sampled values $X = \{X_1, ..., X_n\}$

Function: GMM_Sample(μ, σ, α, S, Mode)
  for i = 1 to n do
    begin
      Draw $k_i \sim$ Categorical(α)
      if Mode == "ideal" then
        $\varepsilon_i \leftarrow$ torch.randn(S)
      else
        $\varepsilon_i \leftarrow$ sample_from_file(S)
      $X_i \leftarrow \mu_{k_i} + \sigma_{k_i} \odot \varepsilon_i$
    end
return X
end Function
```

Figure 7: GUI interface for Mode Selection.

4.3 Memory Mode

4.4 CIM Mode

5 Chip Level Architectures

6 Device Level: Technology File

7 Algorithm Level: PyTorch Wrapper

8 How to run PROMISE

References