Random and fixed seeds

Reproducibility is a fundamental requirement in Bayesian analysis. BayesForge incorporates a comprehensive pseudo-random number generator (PRNG) architecture powered by JAX's splittable PRNG keys alongside Python's standard and NumPy random engines.

---

1. Initializing Seeds in `bf()`

When creating a `bf()` instance, the `rand_seed` argument dictates the global seed policy for the entire session:

Python

```python
from BayesForge import bf

# 1. Fixed reproducible seed (explicit integer)
m = bf(rand_seed=42)

# 2. Fixed reproducible default (base seed 0)
m = bf(rand_seed=False)

# 3. Random seed (entropy from wall-clock time)
m = bf(rand_seed=True)
```

R

```r
library(BayesianInference)

# 1. Fixed reproducible seed (explicit integer)
m <- importBI(rand_seed = 42)

# 2. Fixed reproducible default (base seed 0)
m <- importBI(rand_seed = FALSE)

# 3. Random seed (entropy from wall-clock time)
m <- importBI(rand_seed = TRUE)
```

Configuration Modes

`rand_seed` Setting	Base Seed	Reproducibility	Use Case
`rand_seed=<int>`	Specified integer	**Fully Reproducible**	Production runs, publications, benchmarks
`rand_seed=False`	`0`	**Fully Reproducible**	Quick reproducible prototyping
`rand_seed=True`	`time.time_ns()`	**Non-deterministic** (auditable)	Exploratory analysis, testing variance across chains

Auditing and Replaying Random Runs

When `rand_seed=True` is used, BayesForge automatically records the exact base seed derived from `time.time_ns()`. You can retrieve it at any time:

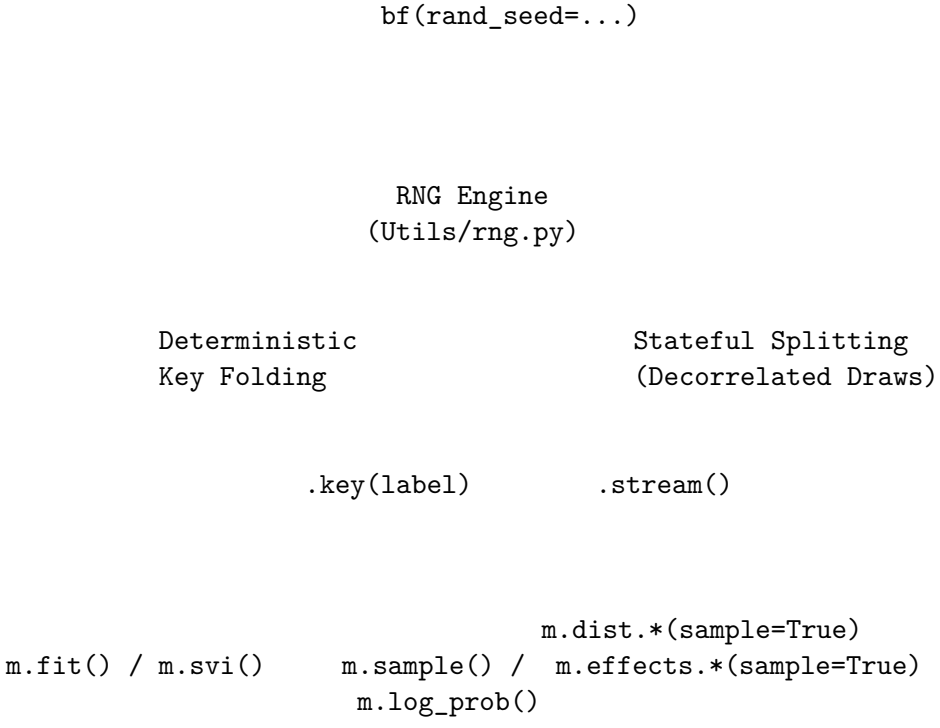
```r
# Retrieve the generated seed to reproduce the run later
recorded_seed = m.rng.get_seed()
print(f"Session Seed: {recorded_seed}")

# Replay the exact same session later:
m_replayed = bf(rand_seed=recorded_seed)
```

---

2. Dual-Layer PRNG Access Architecture

BayesForge implements two distinct PRNG key access mechanisms under the hood via `Utils/rng.py` to meet different sampling requirements:



A. Deterministic Key Folding: `.key(label)`

- **Mechanism:** Computes `jax.random.fold_in(PRNGKey(base), hash(label))`.
- **Behavior:** Purely functional and idempotent. Requesting a key with the same string label (e.g., `'fit'`, `'svi'`, `'prior_predictive'`) always returns the exact same key.
- **Why it matters:** Executing `m.fit()` twice produces identical posterior draws. Furthermore, running ad-hoc prior predictive draws before calling `.fit()` will **not** shift or affect the inference results.

B. Sequential Stream Splitting: `.stream()`

- **Mechanism:** Mutates and advances the internal key state:

$$\text{key}_{\text{next}}, \text{sub-key} = \text{jax.random.split}(\text{key}_{\text{current}})$$

- **Behavior:** Each call produces a fresh, independent sub-key.
- **Why it matters:** When taking interactive sample draws (e.g., `m.dist.normal(0, 1, sample=True)`), consecutive draws are guaranteed to be decorrelated rather than accidentally repeating the same random values, while remaining completely reproducible from the session's base seed.

C. Global Seeding: `seed_globals()`

- Automatically synchronizes the seed with Python's standard `random.seed()` and `numpy.random.seed()` so that non-JAX code paths and data preprocessing routines remain deterministic.

---

3. Submodule Propagation

BayesForge ensures that all specialized submodules—including random effects (`m.effects`), network modeling (`m.net`), Gaussian processes, mixture models (DPMM, GMM), and survival analysis—share the exact same `RNG` instance and active stream state as `m.dist`.

When random effect structures or network layers are sampled with `sample=True`, their draws are seamlessly coordinated with the session's random seed.

---

4. Hands-on Examples

Example 1: Reproducible Model Fitting

Python

```python
import jax.numpy as jnp
from BayesForge import bf

# Simulated data
X = jnp.linspace(-2, 2, 50)
Y = 1.5 + 2.0 * X + 0.3 * jnp.array(jnp.ones(50))

def linear_model(Y, X):
    a = m.dist.normal(0, 1)
```

```
    b = m.dist.normal(0, 1)
    sigma = m.dist.exponential(1)
    m.dist.normal(a + b * X, sigma, obs=Y)

# Run 1
m1 = bf(rand_seed=123)
m1.fit(linear_model, Y=Y, X=X, samples=500, warmup=200)
post1 = m1.get_samples()

# Run 2 with identical seed
m2 = bf(rand_seed=123)
m2.fit(linear_model, Y=Y, X=X, samples=500, warmup=200)
post2 = m2.get_samples()

# Verify exact match
import numpy as np
np.testing.assert_allclose(post1['a'], post2['a'])
print("Results are perfectly identical across runs.")
```

R

```
library(BayesianInference)

X <- seq(-2, 2, length.out = 50)
Y <- 1.5 + 2.0 * X + 0.3

linear_model <- function(Y, X) {
    a = m$dist$normal(0, 1)
    b = m$dist$normal(0, 1)
    sigma = m$dist$exponential(1)
    m$dist$normal(a + b * X, sigma, obs = Y)
}

# Run with fixed seed
m <- importBI(rand_seed = 123)
m$fit(linear_model, Y = Y, X = X, samples = 500, warmup = 200)
post <- m$get_samples()
```

Example 2: Interactive Sampling with Seed Decorrelation

When generating direct distribution draws via `sample=True`, each draw advances the RNG stream:

Python

```python
from BayesForge import bf

m = bf(rand_seed=42)

# Sequential interactive sampling
draw_1 = m.dist.normal(0, 1, sample=True)
draw_2 = m.dist.normal(0, 1, sample=True)

# Draws are decorrelated (draw_1 != draw_2)
print(f"Draw 1: {draw_1:.4f}")
print(f"Draw 2: {draw_2:.4f}")

# Random effects sampling also respects the stream
varying_draws = m.effects.varying_effects(
    N=10,
    sigma=1.0,
    label="group_effect",
    sample=True
)
print("Varying effects sample:", varying_draws)
```

R

```r
library(BayesianInference)

m <- importBI(rand_seed = 42)

draw_1 <- m$dist$normal(0, 1, sample = TRUE)
draw_2 <- m$dist$normal(0, 1, sample = TRUE)

print(draw_1)
print(draw_2)
```

Example 3: Independence of Interactive Draws and `.fit()`

Because inference methods use deterministic key folding via `.key(label)`, intermediate sampling does not alter the posterior results:

```
from BayesForge import bf

# Model A: Direct fit
mA = bf(rand_seed=999)
mA.fit(linear_model, Y=Y, X=X, samples=300, warmup=100)
samples_A = mA.get_samples()

# Model B: Several ad-hoc samples drawn before fit
mB = bf(rand_seed=999)
_ = mB.dist.normal(0, 1, sample=True)
_ = mB.dist.gamma(2, 2, sample=True)
_ = mB.effects.varying_effects(N=5, sample=True)

# Fitting produces the exact same posterior as Model A
mB.fit(linear_model, Y=Y, X=X, samples=300, warmup=100)
samples_B = mB.get_samples()

import numpy as np
np.testing.assert_allclose(samples_A['b'], samples_B['b'])
print("Posterior samples match perfectly regardless of prior ad-hoc sampling.")
```

---

5. Summary & Best Practices

1. **For Publications & Production:** Always specify an explicit integer `rand_seed` (e.g. `rand_seed=42`) to guarantee reproducibility.
2. **For Exploratory Work with `rand_seed=True`:** Always save or log `m.rng.get_seed()` if you intend to re-examine or verify surprising findings later.
3. **Multi-Instance Concurrency:** If running parallel models on multiple processes, initialize each process with its own distinct integer seed (e.g. `seed + worker_id`).