BRNG: Bidirectional Random Number Generators

Applications in adjoint automatic differentiation

28th EuroAD Workshop - CERN Geneva

10 December 2025

Juan Lucas Rey (Bloomberg)

Bidirectional Random Number Generators

- Most RNG users don’t know it is possible
- In the cryptography community, it is considered a bad thing (“backtracking resistance” is desirable)
- In the context of Adjoint AD or backpropagation, it could be useful

Types of Random Number Generators

- **True Random Number Generators (TRNGs)** ❌
- **Pseudo-Random Number Generators (PRNGs)** ✓
- **Quasi-Random Number Generators (QRNGs)** ✓
- **Cryptographically Secure PRNGs (CSPRNGs)** ✓

Only deterministic RNGs (PRNGs, CSPRNGs, QRNGs) can be made bidirectional

Deterministic RNG Architecture: Two-Function Design

1. State Transition Function f

- Updates the internal state: $state_{i+1} = f(state_i)$
- LCG: $state_{i+1} = (a * state_i + c)\%m$
- Xorshift: $state_{i+1} = state_i \oplus (state_i << 13)$
- Counter-based: $state_{i+1} = state_i + 1$

2. Output Function g

- Transforms state to output: $output_i = g(state_i)$
- May be identity or add mixing
- Examples: PCG permutation, MT tempering, crypto hash

For Bidirectional RNGs,

we only need to be able to invert

the state transition function f!

State Transition f tyical operations and inverse operations

- Counter increment -> counter decrement
  - runtime(forward) == runtime(backward)
- Modular multiplication -> modular multiplicative inverse
  - runtime(forward) == runtime(backward)
- Bitwise XOR -> bitwise XOR
  - runtime(forward) == runtime(backward)
- Bitwise shift and XOR -> reverse shifts and XORs
  - runtime(forward) != runtime(backward) (!!)
  - Number of operations depends on the ratio of word size to shift amount

Bitwise shift and XOR: 8-bit, 4-bit shift -> 1 operation to invert

Forward: `x ^= x << 4`

```
Original x:     [1][0][1][1][0][1][0][0]
x << 4:         [0][1][0][0][0][0][0][0]  (XOR)
                  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
Result:         [1][1][1][1][0][1][0][0]
```

Backward:

```
Result:         [1][1][1][1][0][1][0][0]
x << 4:         [0][1][0][0][0][0][0][0] (XOR)
                  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
Recovered x:    [1][0][1][1][0][1][0][0]
```

Bitwise shift and XOR: 8-bit, 2-bit shift -> 3 operations to invert (!)

Forward: `x ^= x << 2`

```
Original x:     [1][0][1][1][0][1][0][0]
x << 2:         [1][1][0][1][0][0][0][0] (XOR)
                  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
Result:         [0][1][1][0][0][1][0][0]
```

```
Result:                   [0][1][1][0][0][1][0][0]
(x << 2) & mask<2,2>:     [0][0][0][0][0][0][0][0] (XOR)
                            ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
Intermediate x:           [0][1][1][0][0][1][0][0]
(x << 2) & mask<4,2>:     [0][0][0][1][0][0][0][0] (XOR)
                            ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
Intermediate x:           [0][1][1][1][0][1][0][0]
(x << 2) & mask<6,2>:     [1][1][0][0][0][0][0][0] (XOR)
                            ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
Recovered x:              [1][0][1][1][0][1][0][0]
```

BRNG interface

```
static_assert(std::bidirectional_iterator<RNG_TYPE>);

class BRNG
{
    result_type operator*() const;         // function g applied (CONST!!)
    bidirectional_iterator& operator++(); // function f applied
    bidirectional_iterator& operator--(); // function f-1 applied
    bidirectional_iterator operator+=(std::size_t z) -> BRNG &; // fast discard FORWARD
    bidirectional_iterator operator-=(std::size_t z) -> BRNG &; // fast discard BACKWARD
};
```

(as opposed to the usual C++ RNG interface):

```
class RNG
{
    result_type operator(); // NON CONST!! (applies f and g)
    void discard( unsigned long long z ); // fast discard FORWARD
};
```

Runtime Comparisons (1B sims)

Reproduced, refactored and developed backward sims for all RNGs from:

- C++ std, Practrand, Random123, Python's randomgen libraries
- Anything by Melissa E. O'Neill (PCG family), Pierre L'Ecuyer (WELL, MRG, ...), Makoto Matsumoto (MT and variants)
- and more...
- Many QRNGs are missing, but Sobol is there
- Some winners and participants from eSTREAM competition (HC-128, Trivium, Salsa)
- **120+ RNG implementations tested** - Forward vs. Backward timing

Complete Results, ordered by $runtime(Bwd)/runtime(Fwd)$

RNG	Fwd	Bwd	Ratio		RNG	Fwd	Bwd	Ratio		RNG	Fwd	Bwd	Ratio
tyche	3488	1335	0.38		WELL19937a	3389	2048	0.60		WELL44497a	2962	2018	0.68
WELL1024a	2339	1697	0.73		efiix32x48	1164	851	0.73		efiix64x48	1164	856	0.74
sfmt86243_64	1689	1352	0.80		efiix16x48	1186	1011	0.85		efiix8x48	1185	1009	0.85
xoroshiro128**	2755	2422	0.88		sfmt216091	1670	1511	0.90		gjrand16	2015	1848	0.92
xoroshiro128+	2739	2545	0.93		xoroshiro64*	2764	2604	0.94		salsa20	4579	4357	0.95
squares64	1341	1279	0.95		chacha	4827	4670	0.97		mcw128	1619	1572	0.97
sobol	866	839	0.97		threefry2x64	3467	3408	0.98		splitmix32	421	411	0.98
MRG32k5a	6824	6752	0.99		Newlib	997	992	0.99		sfmt607	1477	1466	0.99
splitmix64	471	467	0.99		Borland	993	992	1.00		philox2x64	2431	2421	1.00
minstd_rand0	3232	3234	1.00		minstd_rand	3232	3234	1.00		ZX81	746	744	1.00
ranqd1	743	745	1.00		RANDU	994	997	1.00		sfmt1279	1466	1473	1.00
sfmt11213	1553	1558	1.00		speck34	11134	11128	1.00		squares32	997	993	1.00

Complete Results, ordered by $runtime(Bwd)/runtime(Fwd)$

RNG	Fwd	Bwd	Ratio		RNG	Fwd	Bwd	Ratio		RNG	Fwd	Bwd	Ratio
threefry2x32	3408	3394	1.00		sfmt44497	1476	1488	1.01		gjrand8	1977	2008	1.02
mt19937	1361	1399	1.03		mt19937_64	1482	1537	1.04		philox4x64	1992	2072	1.04
sfmt86243	1410	1474	1.05		philox2x32	2478	2606	1.05		mcg128	994	1040	1.05
ranlux48	1092	1150	1.05		threefry4x32	3182	3381	1.06		ranlux24	1092	1170	1.07
threefry4x64	3219	3461	1.08		philox4x32	2078	2276	1.10		xoroshiro128++	2508	2765	1.10
gjrand32	1388	1538	1.11		hc128	2475	2747	1.11		gjrand64	1368	1535	1.12
mcg128_fast	883	994	1.13		dsfmt19937	1013	1146	1.13		xoroshiro64**	2309	2631	1.14
dsfmt11213	986	1137	1.15		dsfmt4253	987	1145	1.16		sfmt216091_64	1690	1976	1.17
dsfmt44497	983	1162	1.18		dsfmt132049	965	1136	1.18		dsfmt86243	965	1149	1.19
PCG	815	1040	1.28		sfmt1279_64	1761	2282	1.30		sfmt607_64	1786	2338	1.31
dsfmt2203	1463	1935	1.32		dsfmt1279	1448	1921	1.33		sfmt11213_64	1688	2259	1.34
sfmt44497_64	1690	2260	1.34		dsfmt216091	1438	1939	1.35		sfmt4253	1663	2384	1.43

Complete Results, ordered by $runtime(Bwd)/runtime(Fwd)$

RNG	Fwd	Bwd	Ratio		RNG	Fwd	Bwd	Ratio		RNG	Fwd	Bwd	Ratio
hc256	2497	3653	1.46		sfmt132049	1469	2245	1.53		dsfmt521	1001	1539	1.54
sfmt19937	1516	2342	1.54		sfmt2281	1468	2345	1.60		sobol64	1096	1782	1.63
xoshiro512**	897	1584	1.77		rarns32	1683	3033	1.80		MRG31k3p	3198	5867	1.83
xoshiro512++	826	1557	1.88		romuquad	658	1277	1.94		MRG32k3a	2541	5120	2.01
romutrio	629	1268	2.02		xoshiro512+	743	1502	2.02		sfmt132049_64	1689	3510	2.08
mcw192	725	1576	2.17		sfmt19937_64	1689	3727	2.21		romuduo	793	1783	2.25
sfmt4253_64	1702	3893	2.29		lfsr113	2533	5830	2.30		sfmt2281_64	1689	3939	2.33
isaac64x256	2560	6198	2.42		tyche_openrand	1171	2985	2.55		taus88	2594	6770	2.61
jsf8	1046	2735	2.61		jsf32	667	1744	2.61		jsf64	751	1989	2.65
mcw256	577	1536	2.66		isaac32x256	2525	6848	2.71		arbee	810	2250	2.78
jsf16	670	1877	2.80		MRG63k3a	4174	11728	2.81		arc4	606	1747	2.88
lxm	1131	3398	3.00		rarns16	1676	5051	3.01		WELL512a	2544	8159	3.21

Complete Results, ordered by $runtime(Bwd)/runtime(Fwd)$

RNG	Fwd	Bwd	Ratio		RNG	Fwd	Bwd	Ratio		RNG	Fwd	Bwd	Ratio
rarns64	1119	3630	3.24		xoshiro256**	960	3378	3.52		xoshiro128**	919	3399	3.70
sfc16	1115	4229	3.79		xoroshiro1024**	847	3218	3.80		xoroshiro1024++	818	3180	3.89
xoshiro256++	792	3335	4.21		xoshiro128++	802	3410	4.25		xoshiro256+	754	3254	4.32
sfc8	1116	5035	4.51		xoshiro128+	764	3544	4.64		xoroshiro1024*	670	3485	5.20
lfsr258	2067	11773	5.70		**sfc32**	702	6080	**8.66**		**sfc64**	774	19305	**24.94**
trivium_engine	3744	130169	**34.77**										

Key Findings

✅ Most ratios are between 0.9 and 3.0, so backward performance is comparable to forward performance

⚠️ **Avoid (>10.0):** trivium (34.8×), sfc64 (24.9×), lfsr258 (5.7×)

- Trivium is a very interesting case study: designed for cryptography, very complex state transitions, hardest case to invert, extremely poor backward performance
- It should have been 64x slower, but I did my best to optimize it
- Nobody would use it for MC simulations anyway

AD application 1: Opcode tape reduction

- A special opcode is created for `*rng` and `++rng`
- Generally, RNG numbers are used to multiply with some active variable
- This means the random number HAS to be recorded on the tape for the reverse pass
- With bidirectional RNGs, we can avoid recording the random number, and instead just perform a `--rng` operation to get the same random number again
- This saves tape space and time spent recording the random number
- Runtime increase is negligible compared to overall AD runtime

AD application 1: A possible RNG interface (1)

```
template <class RNG> class TapeRNG {
  public:
    using value_type = RNG::value_type;

  private:
    mutable Tape *tape;
    RNG rng_;
    value_type cache_;

  public:
    template <class... Args>
    explicit TapeRNG(Tape &t, Args const &...args) : tape(&t), rng_(args...) {
        cache_ = *rng_;
    }

    inline auto val() const -> value_type { return cache_; }

    inline auto operator*() const -> TapeRNG const & { return *this; }

    inline auto operator++() -> TapeRNG & {
        ++this->rng_;
        cache_ = *rng_; // we cache the value for efficiency

        if (tape) {
            tape->ops.push_back(OpCode::RNG_ADV);
        }

        return *this;
    }
```

AD application 1: A possible RNG interface (2)

```
    inline auto operator--() -> TapeRNG & {
        --this->rng_;
        cache_ = *rng_;
        return *this;
    }

    template <class RNG2>
    friend auto operator*(const active_type &lhs, const TapeRNG<RNG2> &rhs)
        -> active_type;
};

template <class RNG>
inline auto operator*(const active_type &lhs, const TapeRNG<RNG> &rhs)
    -> active_type {
    active_type result(lhs.value * rhs.cache_, lhs.tape);

    if (lhs.is_active() && rhs.tape) {
        // do not store any value, only the opcode and the ids
        lhs.tape->record_unary(OpCode::RNG_MUL, lhs, result);
    }

    return result;
}
```

AD application 1: Client code

```
TapeRNG<standard_normal<mt19937>>
    rng(tape, static_cast<std::uint_fast32_t>(148969043));

for (std::size_t j = 0; j < N_RNG; ++j) {
    for (std::size_t i = j; i < N_RNG; ++i) {
        modelstate[i] += cholesky_decomposition(i, j) * (*rng);
    }
    ++rng;
}
```

AD application 1: Backpropagation loop

```
switch (opcode) {
    case OpCode::MUL: {
        ...
    }
    case OpCode::EXP: {
        ...
    }
    ...
    case OpCode::RNG_MUL: {
        id_idx -= 2;
        std::size_t const arg_id = ids[id_idx];
        std::size_t const res_id = ids[id_idx + 1];
        // RNG has to be available on the backpropagation loop!
        // we grab the multiplication value directly from the rng object
        // NOT from the tape
        double const mul_val = rng.cache_;

        double const der_value = derivatives[res_id];
        add_derivative(arg_id, mul_val * der_value);
        break;
    }
    case OpCode::RNG_ADV: {
        --rng;
        break;
    }
}
```

AD application 1: Memory savings

- Example pricing a 5 year Bermudan Swaption with quarterly options, using Hull-White 2 factor model with Mersenne-Twister RNG
- Monthly simulation: 0.90% tape reduction (meh)
- Weekly simulation: 2.20% tape reduction (ok)
- Daily simulation: 10.92% tape reduction (now we're talking)
- Runtime change is unnoticeable (<1%)

AD application 2:

- Avoiding storing the state of full RNG when recording a callback function
- Mersenne-Twister state is 624 words (4992 bytes)!
- With a counter outside the RNG state, we can reproduce the state simulating back
- Only need to store the counter and a reference to the RNG instead of the full state
- Some RNGs don't have a counter so we provide a wrapper that maintains a counter

AD application 2: Counter RNG wrapper

```
template <class RNG> class CounterRNG {
  private:
    RNG rng_;
    mutable value_type cache_;
    mutable std::size_t counter_{0};

  public:
    template <class... Args>
    explicit CounterRNG(Args const &...args)
        : rng_(args...), cache_(*rng_) {}

    auto count() const -> std::size_t { return counter_; }

    void set_to_count(std::size_t c) { // this allows to set the RNG to a given count
        if (c < counter_) {            // (can be before or after current count)
            rng_ -= (counter_ - c);
        } else {
            rng_ += (c - counter_);
        }
        cache_ = *rng_;
        counter_ = c;
    }

    auto operator*() const -> value_type { return cache_; }

    auto operator++() -> CounterRNG & {
        ++rng_;
        cache_ = *rng_;
        ++counter_;
        return *this;
    }
    // etc...
};
```

AD application 2: Client code (bad)

```
tape->switch_to_passive();
// we make a copy of the RNG
// BEFORE using it to generate random numbers
auto rng_before = rng;
T y = g(x, rng);
tape->switch_to_active();

tape->register_variable(y);

// BAD! we copy the full RNG state into the tape
auto fill_gap = [tape, x, y, rng = rng_before]() mutable {
    auto p = tape->get_position();

    T yl = g(x, rng);

    dco::derivative(yl) = dco::derivative(y);

    tape->interpret_adjoint_and_reset_to(p);
};

// this returns a large lambda due to the full RNG capture
std::cout << "lambda size " << sizeof(fill_gap) << std::endl;

tape->insert_callback(std::move(fill_gap));
```

AD application 2: Client code (good)

```
tape->switch_to_passive();
// we only copy the counter of the RNG
std::size_t counter = rng.count();
T y = g(x, rng);
tape->switch_to_active();

tape->register_variable(y);

// GOOD! we copy only RNG reference and counter into the tape
// note: mutable is not needed anymore
auto fill_gap = [tape, x, y, &rng, counter]() {
    auto p = tape->get_position();
    rng.set_to_count(counter);

    T yl = g(x, rng);

    dco::derivative(yl) = dco::derivative(y);

    tape->interpret_adjoint_and_reset_to(p);
};

// this returns a small lambda due to the RNG reference capture
std::cout << "lambda size " << sizeof(fill_gap) << std::endl;

tape->insert_callback(std::move(fill_gap));
```

Applications outside AD 1: Longstaff-Schwartz method

quick reminder of Longstaff-Schwartz method:

- Simulate many paths of underlying asset prices
- At last exercise date, regress continuation value against basis functions of state variables
- On next to last exercise date, decide whether to exercise or continue based on regression on the last exercise date
- Move backward to previous exercise date, repeat regression and decision process
- Regression requires storing all simulated states at exercise dates!!
- Memory usage grows linearly with number of exercise dates (and number of paths)

Applications outside AD 1: Longstaff-Schwartz method

- With bidirectional RNGs, we can avoid storing all option date states, only need to store a constant number of slices, simulate backward to previous option, regress, repeat
- Memory usage is now constant with number of exercise dates!
- Runtime difference is again negligible compared to overall Longstaff-Schwartz runtime

Applications outside AD 2: Faster discard methods for RNGs

- Many RNGs appply fast discarding by applying discard in powers of 2
- If we discard by, for example, 4095 simulations (111111111111 in binary), we would have to apply 12 operations
- It's faster to discard 4096 (1000000000000 in binary) by 1 operation, then go back 1 step
- This divides runtime, on average, by 2!

Funny findings: C++ Standard Library

- C++ defined RNGs have some minor "flaws"
- RNG can be created in "degenerate" states, with 2 RNGS having different states but producing the same output sequence
- Some counterexamples can be constructed where we create an RNG, advance a full cycle, and the state is not the same as the initial state (but produces same sequence)
- With bidirectional RNGs, we can fix these flaws by performing a single backward and forward step on initialisation to "normalize" the state!
- This happens with std::mersenne_twister_engine and std::subtract_with_carry_engine

Funny findings: "Backtracking Resistance" in CSPRNGs

- BACKTRACKING RESISTANCE IS AN ILLUSION
- Most RNGs that claim to be backtracking resistant rely on a hidden variable that scrambles the internal state
- However, NIST defines backtracking resistance as "an adversary that has knowledge of the state of the RBG would be unable to distinguish between observations of ideal random bitstrings and (previously unseen) bitstrings that are output by the RBG at or prior to time T"
- If you assume the adversary does not have knowledge of the hidden variable, then sure, backtracking resistance is achieved
- But that's kind of cheating, isn't it??

Future directions

- Given the particular example of Trivium where the state transition is invertible but is an order of magnitude expensive to invert, can we design RNGs with efficient forward and tremendously ineffient backward operations? "Annoyingly Expensive Backtracking Resistance"??
- Ray-tracing?
- Other AD applications? Let me know!

Questions?

Thank you!

TechAtBloomberg.com