DiffSampler: A Differential and Inherently Parallel Sampling Method for Verification

Arash Ardakani

arash.ardakani@berkeley.edu

Berkeley Institute of Data Science (BIDS)
University of California, Berkeley

Motivation: Embedded Systems

- Embedded Systems
  - Data Processing Units are embedded into large products
- Failure in Embedded Systems
  - Functional Failure
  - Safety Risks
  - Disruption of Services
  - Financial Losses
  - Loss of Trust and Confidence

Motivation: Embedded Systems

- Embedded Systems
  - Data Processing Units are embedded into large products
- Failure in Embedded Systems
  - Functional Failure
  - Safety Risks
  - Disruption of Services
  - Financial Losses
  - Loss of Trust and Confidence

Such consequences necessitates design verification and testing, in particular for safety-critical applications!

Motivational Example: Pentium FDIV bug

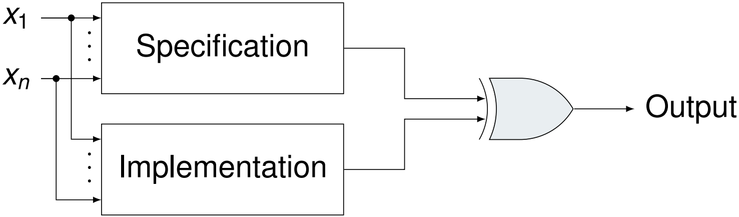
- A hardware bug affecting the floating-point (FP) unit of the early Intel Pentium processors.
- Returns incorrect/inaccurate binary FP results when dividing certain pairs of high-precision numbers.
- Estimated around 1 in 9 billion FP divides with random parameters would produce inaccurate results

Motivational Example: Pentium FDIV bug

- One commonly-reported example is dividing 4,195,835 by 3,145,727:
  - A flawed Pentium processor returns $$\frac{4,195,835}{3,145,727} = 1.333739068902037589$$
  - The correct value of the calculation is $$\frac{4,195,835}{3,145,727} = 1.333820449136241002$$
- Incurred a \$475 million to recover replacement and write-off of these processors

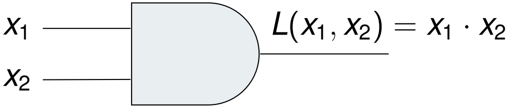
Circuit Design Verification

- Implementation Errors: occur during the mapping of a specification into the final circuit
- Impact: make all produced chips erroneous
- **Formals methods are used to avoid design errors before producing any chip**



Circuit Design Verification–Example

Definition of AND gate: The **output** is **ON** when **both** x_1 **and** x_2 are **ON**.



x_1

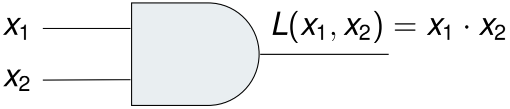
x_2

$$L(x_1, x_2) = x_1 \cdot x_2$$

AND Gate

Circuit Design Verification–Example

Definition of AND gate: The **output** is **ON** when **both** x_1 **and** x_2 are **ON**.

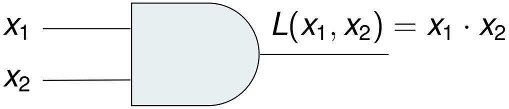


AND Gate

x_1	x_2	$x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

Circuit Design Verification–Example

Definition of AND gate: The **output** is **ON** when **both** x_1 **and** x_2 are **ON**.



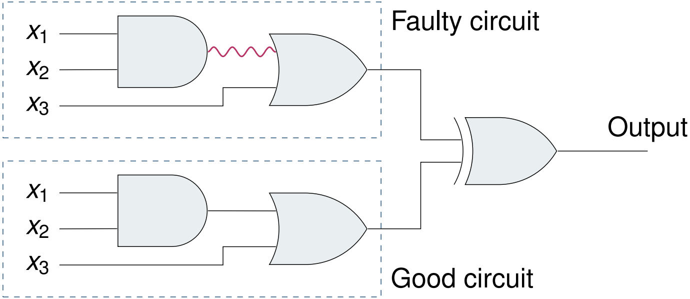
AND Gate

x_1	x_2	$x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

By covering all the possible cases (4 combinations in this example), we can verify the functionality of this circuit.

Circuit Design Test

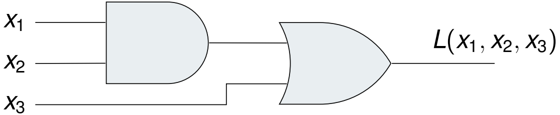
- Production Errors: defects caused during the production of chips which change their functionality
- Causes: A broken transistor switch, a wire shorted to VDD or to ground, unwanted connections, wrong doping, etc.
- **Formals methods to find the production errors**



Circuit Design Test–Example

A good model for such faults is to assume that all faults manifest themselves as some wires being permanently stuck at logic value 0 or 1.

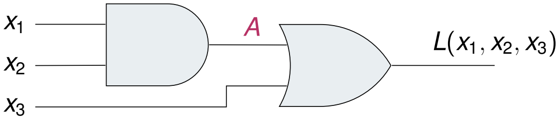
x_1	x_2	x_3	$L(x_1, x_2, x_3)$
0	0	0	0



Circuit Design Test–Example

A good model for such faults is to assume that all faults manifest themselves as some wires being permanently stuck at logic value 0 or 1.

x_1	x_2	x_3	A	$L(x_1, x_2, x_3)$
0	0	0	stuck-at-1	1



The valuation $x_1x_2x_3 = 000$ can detect the occurrence of a stuck-at-1 fault on wire A.

How to find these input combinations?

- The most common approach for circuit design verification and testing is Boolean satisfiability problem (SAT) solving.
- Tremendous performance improvements over years
- State-of-the-art SAT solvers are able to
  - solve problems from real-world applications (e.g., large industrial circuits)
  - handle optimization & enumeration problems, multi-valued domains, hybrid systems

Boolean Satisfiability Problem (SAT)

Given:

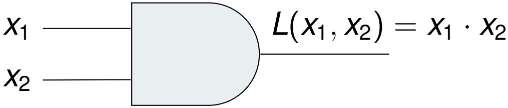
- A Boolean formula φ in Conjunctive Normal Form (CNF)
- A CNF is a conjunction (AND) of clauses: $C_1 \wedge \cdots \wedge C_m$
- A clause is a disjunction (OR) of literals: $(l_1 \vee \cdots \vee l_k)$
- A literal l is a Boolean variable or its negation: l or $\neg l$

Question:

- Is there any valuation of the variables that satisfies φ?

Techniques for solving SAT instances are called SAT solvers

SAT–Example

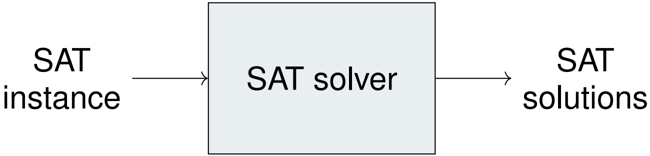


AND Gate

x_1	x_2	$x_1 \cdot x_2$	clauses
0	0	0	$x_1 \vee x_2 \vee \neg L$
0	1	0	$x_1 \vee \neg x_2 \vee \neg L$
1	0	0	$\neg x_1 \vee x_2 \vee \neg L$
1	1	1	$\neg x_1 \vee \neg x_2 \vee L$

- CNF = $(\neg x_1 \vee \neg x_2 \vee L) \wedge (\neg x_1 \vee x_2 \vee \neg L) \wedge (x_1 \vee \neg x_2 \vee \neg L) \wedge (x_1 \vee x_2 \vee \neg L)$

Typical SAT Solving Flow



SAT instance

SAT solver

SAT solutions

Step #1: Real problem in CNF

- $\neg x_1 \vee \neg x_2 \vee L$
- $\neg x_1 \vee x_2 \vee \neg L$
- $x_1 \vee \neg x_2 \vee \neg L$
- $x_1 \vee x_2 \vee \neg L$

Step #2: Adoption of SAT solvers

- Minisat22
- Glucose42
- Cadical195
- CryptoMinisat

Step #3: SAT solution(s)

- $x_1 = 1, x_2 = 1, L = 1$
- $x_1 = 1, x_2 = 0, L = 0$
- $x_1 = 0, x_2 = 1, L = 0$
- $x_1 = 0, x_2 = 0, L = 0$

Intractability Problem with SAT

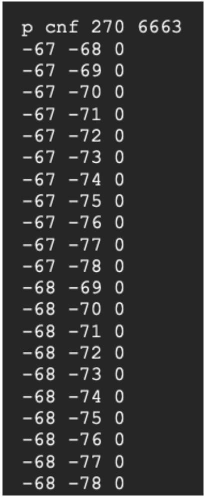
Digital circuits usually take a few operands as inputs. For example, let’s consider an n-bit adder circuit with two operands.

Intractability Problem with SAT

Digital circuits usually take a few operands as inputs. For example, let's consider an n-bit adder circuit with two operands.

input bit-width (n bits)	# input combinations
2	16
3	64
4	256
5	1,024
6	4,096
7	16,384
8	65,536
⋮	⋮
32	18,446,744,073,709,551,616
⋮	⋮
64	340,282,366,920,938,463,463,374,607,431,768,211,456

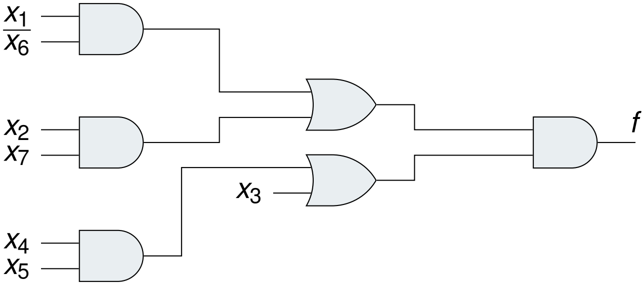
Intangible Representation of SAT



```
p cnf 270 6663
-67 -68 0
-67 -69 0
-67 -70 0
-67 -71 0
-67 -72 0
-67 -73 0
-67 -74 0
-67 -75 0
-67 -76 0
-67 -77 0
-67 -78 0
-68 -69 0
-68 -70 0
-68 -71 0
-68 -72 0
-68 -73 0
-68 -74 0
-68 -75 0
-68 -76 0
-68 -77 0
-68 -78 0
```

- CNF format completely change the structure of multi-level circuits.
- It is almost impossible to understand what circuit CNF describes.
- Make the communication between test and design teams more difficult.

New Testing/Verification Method



Research goal: Find a set of input combinations satisfying any desired output.

- Keep the structure of the circuit intact.
- Fast and inherently parallel method with the support of GPU acceleration.
- Understands both the temporal and spatial nature of computations in hardware.

Proposed Method: ML-based Approach

- The idea is to reframe the testing/verification problem as a supervised multi-output regression task.
- This method is analogous to what we know as adversarial example generation with different goal.
- We use gradient descent to update and obtain valid input combinations for any desired outputs.
- The key to generate valid input combinations is how we can describe basic logic gates.

Modeling Logic Gates

We use **probability** for modeling logic gates.

AND gate: The probability of AND gate outputting 1 is the product of the input operands' probability p_1 and p_2, i.e.,

$$p_{AND} = p_1 \times p_2$$

OR gate: The probability of OR gate outputting 1 is

$$p_{OR} = 1 - (1 - p_1) \times (1 - p_2)$$

NOT gate: The probability of NOT gate outputting 1 is

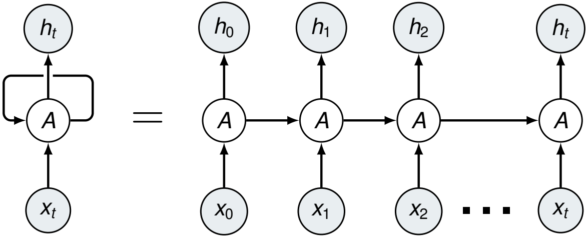
$$p_{NOT} = 1 - p_1$$

Modeling Logic Gates (Combinational Circuits)

- Any other types of gates such as NAND, NOR, XOR and XNOR can be modeled in a similar manner.
- Any combinational digital circuits can be described using the probabilistic models of basic gates.
- The inputs to such models can take any real value between 0 and 1.

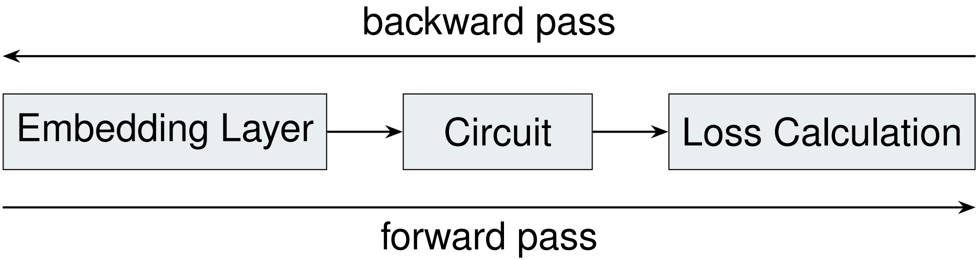
Modeling Memory Elements

- Memories (latches and flip-flops/registers) are often referred to as the “state” of the circuit.
- Modeling states (i.e., memories) of circuits is analogous to modeling recurrent neural networks.



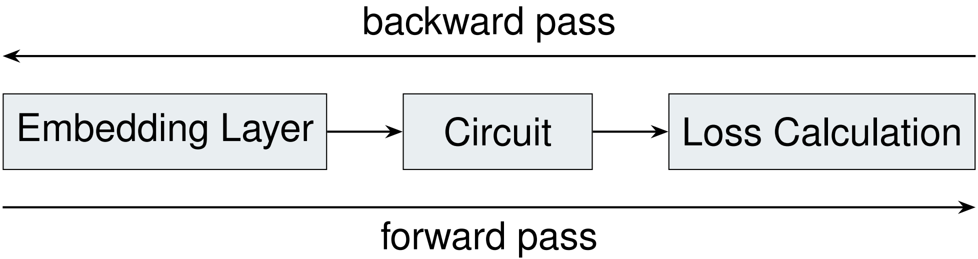
In this figure, *A* can be an adder circuit incrementing the value of state by 1 at each time step (i.e., clock cycle) as an example.

Our ML-based Verification/Testing Flow



- **Embedding layer:** Stores learning parameters which are inputs to the circuit
- **Circuit:** Describes the functionality of the under-test circuit (UTC) using probability
- **Loss Calculation:** Calculates the loss between the expected output and the output of UTC

Our ML-based Verification/Testing Flow



- **Forward pass:** Performs forward computations according to the functionality of the under-test circuit on inputs (inputs are randomly initialized at the beginning)
- **Backward pass:** Gradients w.r.t. inputs are calculated and used to update the soft-valued inputs

Application: SAT Instances

- CNF format can be interpreted as two-level circuit
- The output of such circuit is specifiable when the output of AND gate is 1
- There might be multiple solutions to the problem

DiffSampler: Formulation of SAT

Embedding layer:

We encode n variables of the SAT instance as parameters (i.e., $\mathbf{V} \in \mathbb{R}^{b \times n}$) of an embedding layer where b denotes the batch size.

$$\mathbf{E} = \sigma(\mathbf{V})\left[\mathbf{C}\right] \in \left[0, 1\right]^{b \times m \times l_{\text{max}}},$$

- We express m clauses as the matrix $\mathbf{C} \in \mathbb{Z}^{m \times l_{\text{max}}}$ where l_{max} denotes the maximum number of literals in any single clause in the SAT instance.
- The padding index 0 is reserved for the noncontributory 0-valued e_{ijk} in the OR operation.
- The clause matrix $\mathbf{C}$ contains indices to the variables where the positive and negative indices denote variables in their true and complementary forms, respectively.
- The embedded element $e_{ijk} \in \mathbf{E}$ is equal to $v_{it} \in \mathbf{V}$ when its corresponding index is positive; otherwise $1 - v_{it}$ is assigned to e_{ijk}.
- We use the sigmoid function σ to ensure values between zero and one.

DiffSampler: Formulation of SAT

Circuit:

The OR operations are computed by

$$\mathbf{Y} = 1 - \prod_{l_{\text{max}}} (1 - \mathbf{E}) \in [0, 1]^{b \times m},$$

- Due to the large number of clauses, the AND would results in a value close to 0, resulting in gradient vanishing during the backpropagation.
- Instead of enforcing AND gate to output 1, we enforce the OR between literals in each clause to be 1.

DiffSampler: Formulation of SAT

Loss calculation:

The ℓ_2-loss function $\mathcal{L}$ is obtained by

$$\mathcal{L} = \sum_{b,m} ||1 - \mathbf{Y}||_2^2 .$$

- We use gradient descent to minimize this loss function.
- Upon convergence, the batch of soft values (i.e., **V**) are converted to hard values (i.e., $\widetilde{\mathbf{V}} \in \{0, 1\}^{b,n}$) based on their distance from binary values as *b* solutions to the SAT problem.

Experimental Setup

- A prototype for DIFFSAMPLER, implemented in Python using a high-performance numerical computing library (JAX).
- We compare our sampler implementation against SOTA baselines (UNIGEN3 and CMSGEN) in terms of the run time performance and uniformity of the solutions.
  - UNIGEN3: M. Soos et al., “Tinted, detached, and lazy cnf-xor solving and its applications to counting and sampling,” in Proceedings of International Conference on Computer-Aided Verification (CAV), 2020.
  - CMSGEN: Golia et al., “Designing samplers is easy: The boon of testers,” in Proc. of Formal Methods in Computer-Aided Design (FMCAD), 2021.
- A public domain benchmark suite (60 SAT instances of different sizes) utilized for comparison purposes
  - `https://zenodo.org/records/3793090`

Experimental Setup

- Both baseline samplers were executed on server-grade Intel Xeon Gold 6134 CPU with 3.2GHz clock rate and 1TB RAM
- DIFFSAMPLER results are from running on a system equipped with an Intel Xeon E5-2698 with 2.2GHz clock rate and 8 32GB NVIDIA V100 GPUs.

Experimental Results: Runtime Performance for Representative Subset of 10 Benchmarks

- Run time performance, measured in terms of unique solution throughput.
- Throughput is measured under the case where each method is aimed to produce 1000 unique solutions.

Benchmark	DiffSampler	UniGen3	CMSGen
or-50-10-7-UC-10	75,040.1	64.7	36,693.5
or-70-5-5-UC-30	13,665.6	616.0	36,344.1
or-100-20-8-UC-50	33,728.7	84.4	26,888.6
blasted_1_b12_1	25.8	400.4	10,767.4
blasted_1_b14_3	88.8	97.8	16,495.0
tire-1	35.4	36.8	16,271.4
blasted_1_12_even2	0.7	12.2	1,246.9
blasted_1_14_even	3.3	15.9	4,288.2
modexp-8-4-1	NA	2.8	6.4
hash-02	NA	8.0	1.0

Experimental Results: Runtime Performance for Representative Subset of 10 Benchmarks

- Run time performance, measured in terms of unique solution throughput.
- Throughput is measured under the case where each method is aimed to produce 1000 unique solutions.

Benchmark	DIFFSAMPLER	UNIGEN3	CMSGEN
or-50-10-7-UC-10	75,040.1	64.7	36,693.5
or-70-5-5-UC-30	13,665.6	616.0	36,344.1
or-100-20-8-UC-50	33,728.7	84.4	26,888.6
blasted_1_b12_1	25.8	400.4	10,767.4
blasted_1_b14_3	88.8	97.8	16,495.0
tire-1	35.4	36.8	16,271.4
blasted_1_12_even2	0.7	12.2	1,246.9
blasted_1_14_even	3.3	15.9	4,288.2
modexp-8-4-1	NA	2.8	6.4
hash-02	NA	8.0	1.0

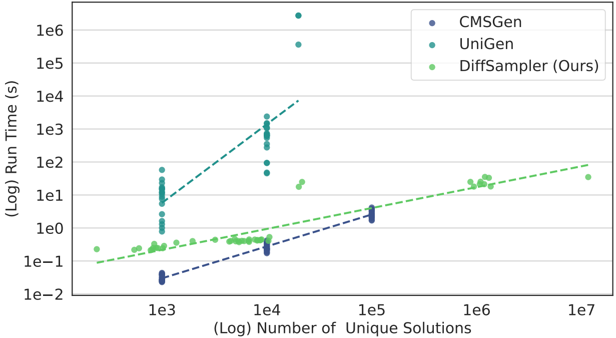
Experimental Results: Runtime Performance for Representative Subset of 10 Benchmarks

- Run time performance, measured in terms of unique solution throughput.
- Throughput is measured under the case where each method is aimed to produce 1000 unique solutions.

Benchmark	DiffSampler	UniGen3	CMSGen
or-50-10-7-UC-10	75,040.1	64.7	36,693.5
or-70-5-5-UC-30	13,665.6	616.0	36,344.1
or-100-20-8-UC-50	33,728.7	84.4	26,888.6
blasted_1_b12_1	25.8	400.4	10,767.4
blasted_1_b14_3	88.8	97.8	16,495.0
tire-1	35.4	36.8	16,271.4
blasted_1_12_even2	0.7	12.2	1,246.9
blasted_1_14_even	3.3	15.9	4,288.2
modexp-8-4-1	NA	2.8	6.4
hash-02	NA	8.0	1.0

Experimental Results: Runtime Performance for Representative Subset of 18 Benchmarks

- Log-Log plot of sampler run time in microseconds against the count of unique satisfying solutions found within that run time. A representative subset of 18 SAT problems from the evaluation benchmark are used (or-50-10-7, or-60-20, or-70-5-5, and or-100-20-8).



Experimental Results: Uniformity Measurement for Representative Subset of 3 Benchmarks

- For all benchmarks where DIFFSAMPLER discovered solutions, BARBARIK framework (which evaluates the uniformity of solutions) returned “Accept”, confirming the uniformity of the generated solutions.
  - Pote et al., “On scalable testing of samplers,” in Advances in Neural Information Processing Systems (NeurIPS), 2022.

Experimental Results: Uniformity Measurement for Representative Subset of 3 Benchmarks

- Hamming distance distribution statistics between satisfying solutions.

Benchmark	tire-2			blasted_case_1_b12_1			or-100-20-8-UC-10		
Sampler	DiffSampler	UniGen3	CMSGen	DiffSampler	UniGen3	CMSGen	DiffSampler	UniGen3	CMSGen
Range	[2,121]	[3,123]	[3,131]	[7, 157]	[3,30]	[6,200]	[20,75]	[27,91]	[34,111]
Avg	46.0	48.2	69.1	72.7	16.9	116.4	48.0	57.5	74.2
Std	13.1	12.3	19.7	15.5	3.3	24.7	5.9	7.1	8.2
Entropy	5.7	5.6	6.3	6.0	3.8	6.7	4.6	4.9	5.1

Experimental Results: Uniformity Measurement for Representative Subset of 3 Benchmarks

- Hamming distance distribution statistics between satisfying solutions.
- The entropy quantifies the degree of variability in the solutions. Higher entropy suggests that solutions are scattered throughout the search space, making it less likely for them to be uniformly distributed, whereas lower entropy suggests more structured and constrained solutions.

Benchmark	tire-2			blasted_case_1_b12_1			or-100-20-8-UC-10		
Sampler	DiffSampler	UniGen3	CMSGen	DiffSampler	UniGen3	CMSGen	DiffSampler	UniGen3	CMSGen
Range	[2,121]	[3,123]	[3,131]	[7, 157]	[3,30]	[6,200]	[20,75]	[27,91]	[34,111]
Avg	46.0	48.2	69.1	72.7	16.9	116.4	48.0	57.5	74.2
Std	13.1	12.3	19.7	15.5	3.3	24.7	5.9	7.1	8.2
Entropy	5.7	5.6	6.3	6.0	3.8	6.7	4.6	4.9	5.1

What About Multi-Level Circuits?

- Unfortunately, there are a very few benchmarks available for hardware verification.
- The existing benchmarks require the conversion from HDL codes (Verilog or VHDL) to PyTorch, which takes engineering efforts.
- We are currently preparing benchmarks for our tool, stay tuned!

Future Works

- Preprocessing clauses by applying two-level minimization
- Preparing parser that can convert HDL codes to their corresponding PyTorch description
- Measurement of experimental results on GPUs in terms of run time and the number solutions found

Conclusion

- Presented a differentiable sampling method, called DiffSampler
- Reframed the SAT problem as a supervised multi-output regression task
- Independent generation of satisfying solutions using GD
- Support of GPU acceleration due to the parallel nature of the computing paradigm
- Comparable run time performance and uniformity compared to SOTA sampling techniques.

Thank you for your attention

- **Arash Ardakani**, Minwoo Kang, Kevin He, Vighnesh Iyer, Suhong Moon, John Wawrzynek, “Differential and Massively Parallel Sampling of SAT Formulas”, accepted for publication in DAC’24.
- Code is available at `https://github.com/LBR-DAC-2024/DiffSampler`