Beyond: Undergraduate Research Journal

Volume 6 Article 1

# A Nature-Inspired Approach for Scenario-Based Validation of Autonomous Systems

Quentin Goss
*Embry-Riddle Aeronautical University*, gossq@my.erau.edu

Mustafa Akbas
*Embry-Riddle Aeronautical University*, akbasm@erau.edu

UNDERGRADUATE RESEARCH JOURNAL
Beyond

# *A Nature-Inspired Approach for Scenario-Based Validation of Autonomous Systems*

Quentin Goss, Mustafa Ilhan Akbas Ph.D.

## Abstract

Scenario-based approaches are cost and time effective solutions to autonomous cyber-physical system testing to identify bugs before costly methods such as physical testing in a controlled or uncontrolled environment. Every bug in an autonomous cyber-physical system is a potential safety risk. This paper presents a scenario-based method for finding bugs and estimating boundaries of the bug profile. The method utilizes a nature-inspired approach adapting low discrepancy sampling with local search. Extensive simulations demonstrate the performance of the approach with various adaptations.

## Introduction

With the advances in sensory systems and artificial intelligence (AI) engines, autonomous systems have been increasingly deployed in daily tasks. As autonomous features continue to be deployed and utilized, there is an increasingly important need for these systems to be tested for both safety and security.

There are several methods such as formal methods, simulation and physical real-life testing for the safety and security testing of autonomous systems. Applying formal methods has been difficult due to the increasing complexity of autonomous systems. Physical tests are dangerous, costly and extremely slow for sufficient bug coverage. Hence, methods based on scenario-based testing and scenario generation in simulation have been developed to overcome these challenges.

This paper presents a nature-inspired approach for scenario-based safety and cybersecurity testing of autonomous systems using long walk together with local search. The main contributions of this paper are as follows:

- A scenario-based approach for bug finding during cybersecurity testing.
- A concise approach to generating a bug profile and identifying profile boundaries.
- A performance evaluation which compares exhaustive testing, random testing, and low discrepancy sequence testing, against long walk-with-local search adaptations.
- Finally, a graphical comparison of approaches with and without long walk-with-local search adaptation.

The remainder of the paper is organized as follows: Related work is given in Section II. The methodology of the scenario-based approach is presented in Section III. Then a performance evaluation with approach comparisons is presented in Section IV and finally, the paper concludes in Section V.

## Related Work

Testing and validation of autonomous systems has been a vexing issue, and considerable effort has been directed towards providing safe and secure autonomous systems [1]–[3].

The random test generation has been a common approach in software and autonomous system testing [4]–[6]. Recently, there have been approaches to combine random test generation with formal scenario definition [7], [8]. These approaches create significant benefits in several areas, such as falsification or AI training. However, it has also been argued that the random testing is inefficient such that it requires extremely large number of tests to detect bugs or failures [9], [10].

The random test generation has the problem of high discrepancy and dispersion, which result in the clustering of test cases in certain areas. Hence, low discrepancy sequences are introduced to solve the issues in irregularity of random distribution [11], [12]. The utilization of low discrepancy sequences has been proposed for various fields of numerical analysis applications including software debug testing [13].

Morokoff and Caflisch [14] showed that the discrepancies of low discrepancy sequences depend on the number of dimensions they are used in. The performance of the sequences degrade with increasing number of dimensions in terms of discrepancy. Therefore, several mechanisms have been proposed to extend low discrepancy sequences when they are utilized for large number of dimensions [15]. Another important extension to low-discrepancy sequences for their utilization in testing has been randomizing their deterministic behavior [16].

## Scenario-Based Approach

Scenario based approaches are used to explore complex systems such as autonomous systems through observation. This approach utilizes autonomous systems scenarios of functional abstraction level with parameter ranges (functional scenarios), and scenarios of concrete abstraction level where parameter values are given and there is no ambiguity (concrete scenarios). A concrete scenario is reproducible and its input, i.e. the concrete parameter values, and its output, i.e. outcome of the test (bug or not a bug) are recorded. The details of the scenario do not need to be observed to find a bug, which can be considered as any unexpected outcome.

### *A. Finding bugs*

The approach for finding the bugs is given in Algorithm 1. At a high level, the purpose of the algorithm is to run numerous scenarios and collect a sequentially ordered list of the input parameters of the scenario P where each list item p is an array of real numbers, and whether or not a bug occurred B.

The input of the algorithm is as follows:

- A functional scenario with parameter ranges F, which is the template scenario used when generating concrete scenarios.
- A sequence L to be used for the long walk portion of the algorithm. Each sequence sample should be in the shape of 1 by r where r is the number of parameter ranges in F. If L has randomness, a seed must be used for reproducibility.
- The number of indices z to skip ahead L to warm-up the sequence at the start of the algorithm.
- A rapidly-exploring random tree (RRT) is used for local search. A strategy commonly used in robot path planning, applied to parameter space rather than physical space. The way-points of the RRT are sampled and the path is discarded. Therefore, path optimization is not necessary.
- The number of local searches to perform before reverting to long walk $n_{local}$
- The number of scenarios to test in all $n_{all}$.

**Algorithm 1** Finding bugs.

**input:** long walk sequence $L$, Functional Scenarios with parameter ranges $F$, number of indices in $L$ to skip at the start $z$, RRT, number of local searches $n_{\text{local}}$, number of scenarios $n_{\text{all}}$
**output:** list of input parameters $P$, list of bugs $B$

**procedure** find_bugs($L, F, z$,RRT, $n_{\text{local}}, n_{\text{all}}$) **begin**
  $L$.skip($z$) /* Warmup the sequence. */
  $P \leftarrow$ empty list/* For sampled parameters. */
  $B \leftarrow$ empty list/* For bug status. */
  **while** true **do**
    $P$.append($L$.sample()) /* Long walk. */
    $C \leftarrow F(P$.peek()) /* Concrete Scenario */
    $B$.append($C$.run()) /* true when bug is found */
    $n_{\text{all}}$--
    **if** $n_{\text{all}} = 0$ **then return** P, B
    **else if** $B$.peek() **then**
      RRT.reset_and_center($P$.peek())
      **for** $i$ **in** range(1, $n_{\text{local}}$) **do**
        $P$.append(RRT.step())/* Local search. */
        $C \leftarrow F(P$.peek()) /* Concrete Scenario */
        $B$.append($C$.run())
        $n_{\text{all}}$--
        **if** $n_{\text{all}} = 0$ **then return** P, B
      **end for**
    **end if**
  **end while**
**end procedure**

*Algorithm 1:* Algorithm to find bugs.

After the long walk sequence, and the empty lists for sampled parameters and bug statuses are initialized (lines 2-4), the scenario testing loop (lines 5-21) runs until $n_{all}$ scenarios are executed. The procedure is exited when all $n_{all}$ scenarios are executed (lines 10 and 18).

The procedure begins with a long walk, where a point is sampled from L (line 6). Then, a concrete scenario C is generated by distributing the dimensions of that point to the parameter ranges of F (line 7). Next, the concrete scenario is executed and returns true if a bug is observed and false otherwise (line 8). Here is one example, where F is a functional scenario with two parameter ranges $r_1$ and $r_2$, which looks like F($r_1$,$r_2$ ). Parameters $r_1$ and $r_2$ both have ranges between 0 and 1. L is a random sequence which returns a point of two dimensions when sampled, e.x. (0.2, 0.7). This point is distributed to parameters of F such that C = F (0.2, 0.7).

If the concrete scenario C with parameters sampled from L results in a bug, a local search is started (line 11). The root of the RRT is centered on the last point and all other waypoints and paths in the RRT are cleared. (line 12). Then $n_{local}$ searches are performed (lines 13-18) as the RRT is stepped and then sampled for a point (line 14), which is distributed to F to generate a concrete scenario (line 15) which is executed and the bug status is recorded.

### *B. Constructing bug profiles.*

Testing scenarios in Algorithm 1 returns a list of input parameters P and information about outputs on whether they are bugs or not B. All p in P exist as a point in n-d where n is the length of the array p. The points p form clusters of bugs or not bugs based on the corresponding b in B. This approach also includes Algorithm 2, which identifies points at the boundaries of the bug clusters to form a bug profile. In addition to P and B, two more parameters are input into Algorithm 2:

- Granularity of the testing space k in each dimension. The space must be discretized in order for the neighbor estimation equation at line 6 to be valid.
- Decimal places m is the number of decimal places to round values to at lines 2 and 12. Rounding is used at line 2 to discretize the points. If k = 0.01 then m should be 2. At line 12, rounding to m corrects float rounding error.

**Algorithm 2** Locating bugs at the boundary of bug clusters.

**input:** List of points $P$, list of bugs $B$, granularity of testing space $k$, decimal places $m$
**output:** Data frame of unique bugs with neighbors count and boundary information $U$

1: **procedure** find_boundary_bugs($P, B, g, m$) **begin**
2: $P \leftarrow$ list( round($p$,$m$) for all $p$ in $P$ )
3: $R \leftarrow$ redundancy_data_frame($P$,$B$) /* See Table I */
4: $U \leftarrow R$.filter($R.b$ and $R.n_{\text{hits}} = 1$) /* Unique bugs. */
5: $n_{\text{dim}} \leftarrow$ dimension of 1st point $p$ in $P$
6: $h \leftarrow 3^{n_{\text{dim}}} - 1$/* Estimated number of neighbors. */
7: $G \leftarrow$ empty list/* For neighbor w/ bug counts. */
8: $L \leftarrow$ empty list/* For bug-is-on-boundary flags. */
9: **for** row $r$ **in** iterate_rows($U$) **do**
10: $c \leftarrow 0$ /* Counter for neighbors with bugs */
11: **for** point $q$ **in** iterate_items($U.p$) **do**
12: $d \leftarrow$ round($|r.p - q|$, $m$)/* array of size $n_{\text{dim}}$ */
13: **if** all($d > 0$) and all($d \leq k$) **then**
14: $c$++ /* Neighbor w/ bug is found. */
15: **if** $c = h$ **then break**
16: **end for** /* ↑ All neighbors w/ bugs are found */
17: $G$.append($c$) /* $c$ in $\mathbb{W}$ */
18: $L$.append($c < h$) /* $c < h$ in {true, false} */
19: **end for** /* ↓ Trim and add new columns to $U$. */
20: $U \leftarrow U$.drop_columns($b$, $n_{\text{hits}}$)
21: $U \leftarrow U$.add_column(n_neighbors_with_bugs $n_{\text{nwb}}$ = $G$)
22: $U \leftarrow U$.add_column(is_boundary $e$ = L)
23: **return** $U$ /* See Table II. */
24: **end procedure**

*Algorithm 2:* Algorithm to locate bugs at the boundary of the clusters.

To start the procedure, the points are fit to discretized space (line 2) and statistics for the points are generated in a redundancy data frame R (line 3). The columns of the data frame are described in Table I. The row index describes the test. It is sequential and must not be altered. Column $n_{hits}$ in Table I is the count of repeated points up to the current test. Column $n_{bugs}$ is the total number of bugs observed up to the current test. The rows with unique bugs U are selected (line 4) since the calculations for counting neighbors will be inaccurate otherwise. Then the estimated number of neighbors, h, is calculated based on the

discretized space.

| Column Name | | Data Type |
|---|---|---|
| point | $p$ | A point $p$ in n-d space. |
| is_bug | $b$ | $b$ in {true, false} |
| n_hits | $n_{\text{hits}}$ | $n_{\text{hits}}$ in $\mathbb{N}$ |
| n_bugs | $n_{\text{bugs}}$ | $n_{\text{bugs}}$ in $\mathbb{W}$ |

*Table 1:* Redundancy data frame showing the columns of the data frame.

Next step in the approach is the neighbor counting loops (lines 9-19) where the Manhattan distance d of each point to all other points is compared (lines 12-13). For instance, two points are considered as the same point When all d = 0. Neighbors will have a Manhattan distance of the granularity k away from the point. The count of neighbors with bugs is appended to the list G (line 17). Another important piece of information recorded to the list L is whether the point is a bug at the boundary of the cluster (line 18). Finally, G and L are added as columns to the unique bugs data frame U, and unnecessary columns are dropped for efficiency (lines 20-22). A description of the columns of the output U is shown in Table II.

| Column Name | | Data Type |
|---|---|---|
| point | $p$ | A point $p$ in n-d space. |
| n_bugs | $n_{\text{bugs}}$ | $n_{\text{bugs}}$ in $\mathbb{N}$ |
| n_neighbors_with_bugs | $n_{\text{nwb}}$ | $n_{\text{nwb}}$ in $\mathbb{W}$ |
| is_boundary | $e$ | $e$ in {true, false} |

*Table 2:* Unqiue bugs generated as output.

## Performance Evaluation

An experiment is designed to evaluate the approach and also to compare the impact of variations within the approach to the resulting behavior and performance in autonomous systems scenarios. The experiment parameters are described in Table III, organized by procedure parameters input into Algorithm 1 for bug finding and Algorithm 2. Three functional scenarios F are tests:

- A one dimensional (1D) space with one parameter range of [0,1].
- A two dimensional (2D) space with two parameter ranges $r_n$ which is [0,1] for both $r_n$.
- A three dimensional (3D) space with three parameter ranges $r_n$ which is [0,1] for all $r_n$.

| Parameter | | 1D | 2D | 3D |
|---|---|---|---|---|
| *Find bugs. (Algorithm 1)* | | | | |
| Long walk sequences | $L$ | lattice, random, halton sobol, faure | | |
| Parameter ranges for $F$. | $F(\ldots)$ | $[0,1]\forall$ dimensions | | |
| Number of indices to skip. | $z$ | 2243 | — | — |
| RRT implementation | RRT | regular RRT | | |
| Number of local searches. | $n_{\text{local}}$ | $0^{\dagger}, 5^{\ddagger}$ | — | — |
| Number of scenarios to run. | $n_{\text{all}}$ | $101^1$ | $101^2$ | $101^3$ |
| *Locate boundary bugs. (Algorithm 2)* | | | | |
| List of points. | $P$ | Output of Algorithm 1. | | |
| List of bugs. | $B$ | Output of Algorithm 1. | | |
| Granularity of testing space. | $k$ | 0.01 | — | — |
| Decimal places. | $m$ | 2 | — | — |

$^{\dagger}$ Sequence only. Skips RRT part of algorithm when 0.
$^{\ddagger}$ Sequence + RRT.

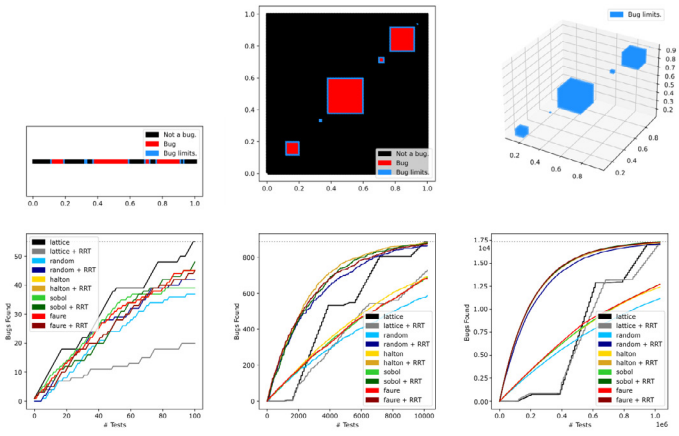
*Table 3:* Experiment parameters as organized by the Algorithms.

Bug clusters are placed in the created spaces. During each set of n_all tests, each assessed approach provides coordinates as a point and the functional scenario returns if the point is within a bug cluster. The following approaches are compared:

- Lattice sequence, which is the baseline. After $101^n$ tests where n is the number of dimensions, the testing space is exhaustively sampled.
- Random sequence, which samples the testing space randomly.
- The low discrepancy sequences Halton, Sobol, and Faure, which use the OpenTurns [17] implementations.
- Variants of the five sequences used for long walk, combined with an RRT for local search. The RRT implementation uses a branch length of 0.01, and a seed of 555.

### *A. Finding Bugs*

One Dimension: There are 55 unique bugs placed across 101 points in the testing space, which are distributed as shown in Figure 1a, along with the summary of bugs found after 101 tests. In one dimension, with just 101 points, exhaustive testing using lattice exceeded bug finding performance of the other approaches. This result shows that the benefits of parameter selection using other approaches or the performance difference they offer is inconclusive in such small state space with a




(a) 1 Dimension. (b) 2 Dimensions. (c) 3 Dimensions.

*Figure 1*: Comparison of approaches from one to three dimensions. (Top) Distribution of bugs. (Bottom) Total unique bugs found as tests are run.

single dimension.

Two Dimensions: There are 887 unique bugs out of 10,201 points in the testing space, which are shown in Figure 1b, along with the summary of bugs found after 10,201 tests.

In two dimensions, the parameter approaches become more positively distinct, with random diverging around 2000 tests. Additionally, adding the local search component to the sequences sharply increases the rate at which unique bugs are found and consistently sooner than lattice. After around 2000 tests, random sequence with RRT diverges from the low discrepancy sequences that are integrated with RRT.

Three Dimensions: There are 17065 bugs out of 1,030,301 points in the testing space, which are shown in Figure 1c, along with the summary of bugs found after 1,030,301 tests. Non-bugs are omitted from the plot since they obscure the bugs profiles in the plot.

In three dimensions with more total points to test, there are several notable observations:

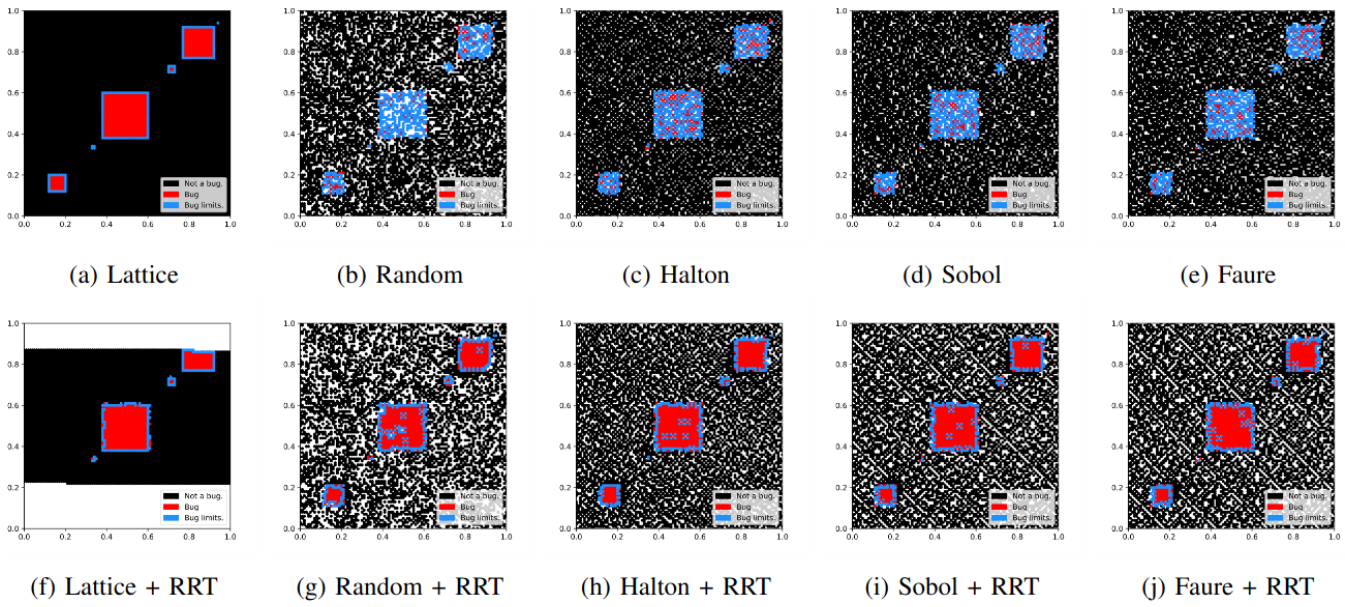
- Low discrepancy sequences perform significantly better compared to random selection. Random parameter selection diverges from the low discrepancy sequences, and random with RRT diverges from the low discrepancy sequences with RRT around 10% of tests.
- The low discrepancy approaches are effective, and their performances are close to each other.
- At test 533,119 or 51.74% of all tests, lattice meets Faure at 8,396 or 49.20% of bugs found, while Faure with RRT has found 15,901 or 93.18% of bugs.

### *B. Bug Profiles*

To visualize the bug profiles, the scatter plots of input parameters are shown for the 2D scenarios after 10, 201 tests in Figure 2. The 2D tests are selected as the example for the simplicity and uncluttered plotting. Observations of adding a local search to sequence are as follows:

- The density of the points outside of bug clusters is lower and the density of the points within the bug profile is higher compared to the case where RRT is not used. This result shows targeted exploration around the bug profile.
- Each bug profile has a “shadow” around the profile limits. This scenario dense area is caused when a long walk test results in a bug, and the unweighted RRT in the local search portion grows outside of the bug profile.

Note again, that Figure 2 is in 2D for a graphical inspection of the approaches. The lattice, i.e. exhaustive testing, is the best option for only the case with a single dimension, however as seen in Figure 1c as the amount of dimensions increase and the resource requirement for running scenarios increases, long walk with local search targets




(a) Lattice (b) Random (c) Halton (d) Sobol (e) Faure

(f) Lattice + RRT (g) Random + RRT (h) Halton + RRT (i) Sobol + RRT (j) Faure + RRT

*Figure 2*: Found bugs in two dimensions, after 10, 201 tests.

scenarios in the bug profile in increasingly fewer tests.

## Conclusion

This paper provides a scenario-based testing methodology to find bugs and vulnerabilities for autonomous systems. The approach integrates quasi-random sequences with RRT to locate bugs and find boundaries of bug clusters. The modularity of the approach allows utilization of different quasi-random sequences along with different local search algorithms. Extensive simulations are run to demonstrate the performance of the approach. The results show that the scenarios generated by using quasi-random sequences lead to faster discovery of bugs in the system compared to the random generation of scenarios. This is critical for both the early detection of problems and also the efficient usage of computation time. Our results also show that there are performance differences among quasi random sequences. Therefore, the operational design domain of a system must be analyzed first before selecting the quasi random sequence in our solution. The results demonstrate the significant impact of local search on the identification of errors when integrated with the quasi-random generated scenarios. For all of the sequences, addition of RRT improved the speed of finding bugs. The performance evaluation included comparison to exhaustive search, as well. For the cases where the state space is not extremely large, exhaustive search could be a viable option as well. However, our approach had significantly better performance for all cases other than the single-dimension and the performance difference became larger as the number of dimensions increased.

As future work, we plan to apply the approach in an existing autonomous system setting such as an autonomous vehicle intersection scenario or an unmanned aerial vehicle sense-and-avoid scenario. Furthermore, we plan to explore the impact of local search parameters in the overall performance.

## References

[1] A. Corso, R. J. Moss, M. Koren, R. Lee, and M. J. Kochenderfer, "A survey of algorithms for black-box safety validation," arXiv preprint arXiv:2005.02979, 2020.

[2] M. I. Akbas¸, "Testing and Validation Framework for Autonomous Aerial Vehicles," Journal of Aviation/Aerospace Education & Research, vol. 30, no. 1, pp. 1–19, 2021.

[3] C. Medrano-Berumen and M. I. Akbaş, "Validation of decision-making in artificial intelligence-based autonomous vehicles," Journal of Information and Telecommunication, vol. 5, no. 1, pp. 83–103, 2021.

[4] R. Hamlet, "Random testing," Encyclopedia of software Engineering, vol. 2, pp. 971–978, 1994.

[5] C. Ebert and M. Weyrich, "Validation of autonomous systems," IEEE Software, vol. 36, no. 5, pp. 15–23, 2019.

[6] Q. Goss, Y. AlRashidi, and M. I. Akbaş, "Generation of modular and measurable validation scenarios for autonomous vehicles using accident data," in IEEE Intelligent Vehicles Symposium (IV), July 2021.

[7] S. Hallerbach, Y. Xia, U. Eberle, and F. Koester, "Simulation-based Identification of Critical Scenarios for Cooperative and Automated Vehicles," tech. rep., SAE Technical Paper: 01-1066, 2018.

[8] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta, "Formal scenario-based testing of autonomous vehicles: From simulation to the real world," in IEEE ITSC, pp. 1–8, IEEE, 2020.

[9] G. J. Myers, The Art of Software Testing. John Wiley and Sons Ltd, 1979.

[10] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," IEEE Transactions on Software Engineering, no. 4, pp. 438–444, 1984.

[11] H. Niederreiter, "Low-discrepancy and low-dispersion sequences," Journal of number theory, vol. 30, no. 1, pp. 51–70, 1988.

[12] P. Bratley, B. L. Fox, and H. Niederreiter, "Implementation and tests of low-discrepancy sequences," ACM Transactions on Modeling and Computer Simulation (TOMACS), vol. 2, no. 3, pp. 195–213, 1992.

[13] T. Y. Chen and R. Merkel, "Quasi-random testing," IEEE Transactions on Reliability, vol. 56, no. 3, pp. 562–568, 2007.

[14] W. J. Morokoff and R. E. Caflisch, "Quasi-random sequences and their discrepancies," SIAM Journal on Scientific Computing, vol. 15, no. 6, pp. 1251–1279, 1994.

[15] H. Chi, Scrambled quasirandom sequences and their applications. PhD thesis, The Florida State University, 2004.

[16] H. Liu and T. Y. Chen, "Randomized quasi-random testing," IEEE Transactions on Computers, vol. 65, no. 6, pp. 1896–1909, 2015.

[17] M. Baudin, A. Dutfoy, B. Iooss, and A.-L. Popelin, OpenTURNS: An Industrial Software for Uncertainty Quantification in Simulation, pp. 1–38. Cham: Springer International Publishing, 2016.