2023
Volume 33, Number 3

ACM Transactions on

Modeling and Computer Simulation

Association for Computing Machinery

Advancing Computing as a Science & Profession

ACM Transactions on Modeling and Computer Simulation

Vol. 33 • No. 3 • 2023

Articles 8–10

ACM Transactions on Modeling and Computer Simulation

ACM
1601 Broadway, 10th Floor
New York, NY 10019-7434
Tel.: 212-869-7440
Fax: 212-869-0481
https://www.acm.org

Home Page: https://tomacs.acm.org

The *ACM Transactions on Modeling and Computer Simulation* (ISSN 1049-3301) is published quarterly in Spring, Summer, Fall, and Winter by the Association for Computing Machinery (ACM), 1601 Broadway, 10th Floor, New York, NY 10019-7434. Summer 2023. Periodicals class postage paid at New York, NY 10001, and at additional mailing offices. Printed in the U.S.A. POSTMASTER: Send address changes to *ACM Transactions on Modeling and Computer Simulation*, ACM, 1601 Broadway, 10th Floor, New York, NY 10019-7434.

Association for Computing Machinery

Advancing Computing as a Science & Profession

Uncertainty-aware Simulation of Adaptive Systems

JEAN-MARC JÉZÉQUEL, University of Rennes, CNRS, Inria, IRISA, France
ANTONIO VALLECILLO, ITIS Software, Universidad de Málaga, Spain

Adaptive systems manage and regulate the behavior of devices or other systems using control loops to automatically adjust the value of some measured variables to equal the value of a desired set-point. These systems normally interact with physical parts or operate in physical environments, where uncertainty is unavoidable. Traditional approaches to manage that uncertainty use either robust control algorithms that consider bounded variations of the uncertain variables and worst-case scenarios or adaptive control methods that estimate the parameters and change the control laws accordingly. In this article, we propose to include the sources of uncertainty in the system models as first-class entities using random variables to simulate adaptive and control systems more faithfully, including not only the use of random variables to represent and operate with uncertain values but also to represent decisions based on their comparisons. Two exemplar systems are used to illustrate and validate our proposal.

CCS Concepts: • **Software and its engineering → Software design engineering**; **Model-driven software engineering**; • **Computing methodologies → Uncertainty quantification;**

Additional Key Words and Phrases: Model-based software engineering, control systems, self-adaptive systems, uncertainty

ACM Reference format:
Jean-Marc Jézéquel and Antonio Vallecillo. 2023. Uncertainty-aware Simulation of Adaptive Systems. *ACM Trans. Model. Comput. Simul.* 33, 3, Article 8 (May 2023), 19 pages.
https://doi.org/10.1145/3589517

1 INTRODUCTION

An adaptive system is a system that changes its behavior in response to its environment or to changes in its interacting parts. In general, these systems are rather complex to design, prove correct, and optimize, and therefore simulations are used to analyze not only their behavior but also their properties of interest. In this context, models are used to represent the relevant characteristics of the system under study, whereas the simulations represent the evolution of the model over time [43].

Simulations are commonly used in domains where physical artifacts are costly to build and deploy, such as manufacturing [26] or robotics [32]. A typical example is an automated assembly

This work was partially funded by the Spanish Government (FEDER/Ministerio de Ciencia e Innovación–Agencia Estatal de Investigación) under projects PID2021-125527NB-I00 and TED2021-130523B-I00.
Authors' addresses: J.-M. Jézéquel, University of Rennes, CNRS, Inria, IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France; email: jezequel@irisa.fr; A. Vallecillo, ITIS Software, Universidad de Málaga, Bulevar Louis Pasteur 35 (29071) Spain; email: av@uma.es.

1049-3301/2023/05-ART8 $15.00
https://doi.org/10.1145/3589517

line, in which conveyor belts and gantries are used to transport semi-assembled parts from one workstation to another, and the parts are added in sequence until the final assembly is produced. These systems are thoroughly simulated before they are deployed to ensure correct behavior once they are built. However, when deployed, they most often require some fine-tuning. The problem is that their physical parts and elements are never perfect: They contain looseness and small inaccuracies that need to be adjusted for. These inaccuracies are not usually captured by the models, often resulting in parts falling off the trays or clamps not gripping the items when initially deployed, for example.

Reality is indeed different from the model and its simulation, because, e.g., the values obtained from the sensors are actually imprecise, the physical contour of the parts is not exactly the same in all cases, or the moving times are not always precise. Since this uncertainty is usually not explicitly considered—despite being an essential aspect of any physical system [12, 19]—the decisions made by the control system to order movements or to make adaptations are based on imprecise information that can even lead to catastrophic failures.

The usual solution to deal with uncertainty in these situations, including not only manufacturing but also in all types of control systems [10], uses robust control and a conservative strategy that relies on estimating static upper bounds on the variations of the variables and assumes worst-case scenarios. This approach is easy to implement, but it may be too conservative in many situations and therefore sub-optimal—e.g., wasting too many resources or arriving at non-optimal approximations. Adaptive control systems aim at addressing this problem by using dynamic variables (instead of upper bound constants) to control the system's behavior. Although this strategy results in simulations that are more faithful to reality, they are much more difficult to develop and prove correct, because all uncertainty operations, as well as the propagation of uncertainty, need to be manually and explicitly programmed by the software engineer. At the simulation level, this is commonly achieved using some of the existing uncertainty propagation packages, such as in References [2, 22, 23] (see Reference [41] for a comprehensive list), but they are still quite complex to use. More importantly, these packages enable the propagation of uncertainty through arithmetic operations on uncertain numbers, but their comparison is not usually implemented. Indeed, these packages only offer crude support for comparing uncertain values, and we shall see that this is an essential operation for operating with uncertainty in a more precise manner.

In this article, we propose to use random variables as first-class entities in programs and models to represent and operate with the system uncertain values, including their comparison. In this way, the controller is going to be able to manage the uncertainties and then the simulations are going to be much more faithful than they currently are.

Our concrete claims are that (1) we need to include the sources of uncertainty as first-class entities in the system models, and (2) they should be better handled by the type system and not by the programmers. This will enable the natural representation and management of uncertain numbers in models, and the automatic propagation of uncertainty through operations, which are cumbersome and error-prone tasks when performed manually by the programmers. In addition, it will allow the explicit representation of the uncertainty that occurs when two uncertain numbers are compared.

The organization of the article is as follows. After this introduction, Section 2 briefly describes the context and background of our work. Then Section 3 presents our proposal, starting with a running example that serves to illustrate our approach. We then describe how to represent and manage some of the uncertainties that affect that system. Another example is used to illustrate further uncertainties in a more complex setting. After that, Section 4 discusses some of the

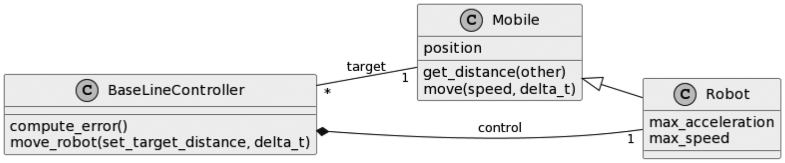


Fig. 1. Architecture of the chasing robot example.

advantages and possible limitations of our proposal. Finally, Section 5 relates our proposal to similar works, and Section 6 concludes with an outlook on future work.

Open research: All the software, artifacts and results described in the article are publicly available from https://github.com/atenearesearchgroup/uncertainty-aware-adaptive-systems.

2 CONTEXT

2.1 Introduction to the Chasing Robot

In this section, we present background concepts using a toy example of the simulation of a Chasing Robot model where a robot must follow a moving target, staying as close as possible to the target but without ever going under a specified safety distance (e.g., 1 m). To keep it simple, we only consider a target moving at a constant speed (e.g., 2 m/s) in straight line.

The chasing robot is controlled by a straightforward **proportional–integral–derivative (PID)** controller,[1] as illustrated in Figure 1 and detailed in the Python code below:

```
class BaseLine:
  def __init__(self, robot: Robot, kp: float, ki: float, kd: float):
    self.robot = robot
    self.speed = 0.0
    self.max_acceleration = 5
    self.Kp = kp
    self.Ki = ki
    self.Kd = kd
    self.target = None
    self.error = [0.0, 0.0, 0.0]

  def get_error(self, target_distance: float):
    distance = self.robot.get_distance(self.target)
    return distance - target_distance

  def compute_target_speed(self, target_distance: float, dt: float) -> float:
    error = self.get_error(target_distance)
    self.error[2] = self.error[1]
    self.error[1] = self.error[0]
    self.error[0] = error
    derivative = (error - self.error[1]) / dt
    integral = (error+self.error[1]+self.error[2]) * dt
    return self.Kp * error + self.Ki * integral + self.Kd * derivative
```

Simulating this model consists in having a loop that

(1) moves the target by calling its `move()` method with a given delta time (`dt`)
(2) asks the controller to move the robot it controls with the same delta time. This is implemented by method `move_robot()` below.

[1]A PID controller is a control loop mechanism that continuously calculates an error value as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms (denoted P, I, and D, respectively) [1].

```
  def move_robot(self, target: Mobile, target_distance: float, dt: float):
    speed = self.compute_target_speed(target_distance, dt)
    speed = max(min(self.robot.max_speed, speed), 0)
    delta_speed = speed-self.speed
    if delta_speed != 0:
      sign = delta_speed/abs(delta_speed)
      if abs(delta_speed) < self.max_acceleration*dt:
        self.speed = speed
      else:
        self.speed += sign * self.max_acceleration*dt
      self.robot.move(self.speed, dt)
```

As expected, after an initial acceleration phase, the chasing robot speed converges toward following its target at the required distance. To assess the controller performance, we consider two indicators: (1) the minimum distance ever reached between the chasing robot and its target and (2) the average distance over the entire course. The goal is, of course, to obtain the smallest possible average distance without ever going under the safety distance.

Our `BaseLine` Controller simulation[2] performs relatively well with respect to these indicators: starting 10 m behind the target, it ends up after 30 seconds with an average distance of 2.0 m and a minimum one of 1.07 m. However, when tried in a real situation (i.e., not a simulated one), the chasing robot would violate the safety distance, and even in some cases crash into the target. The reason is that reality is much more uncertain than our simple model.

2.2 Uncertainty Sources

References [38, 44] summarize the main sources of uncertainty found in cyber physical systems. In this article, we only focus on measurement uncertainty [3, 18]. In the chasing robot example, we can identify two sources of measurement uncertainty: (1) when calculating the distance to the target we rely on, e.g., ultrasound sensors that yield imprecise data, and (2) when setting the speed of the robot, its actual speed might be a bit different due to inertia and friction.

Once again, for the sake of simplicity in this section we will only consider (1), i.e., the distance uncertainty. In our simulation, we can introduce a distance sensor uncertainty by adding a random value X to the computation of the distance by a Mobile. X is chosen within a normal distribution, with an average value of 0 (no skew) and a standard deviation depending on the actual distance (due to the speed of sound) of $\sigma * (1 + 0.25 * real\_distance)$.

Now, if we run again our simulation several times with increasing values of σ, then we can indeed see that as soon as $\sigma \geq 0.0325$ m the chasing robot can violate the safety distance.

2.3 Robust Control

In control theory, robust control is an approach to make a controller work in the presence of uncertainty, assuming that certain variables will be unknown but bounded [1]. These robust methods aim to achieve robust performance and/or stability in the presence of bounded modeling errors. We can thus implement a variant of our `BaseLine` Controller that we call `RobustController` that basically takes a fixed safety margin of $10 \times \sigma$ when computing the error to be minimized in the PID control:

```
class RobustController (BaseLine):
  def __init__(self, robot: Robot, kp: float, ki: float, kd: float):
    super().__init__(robot, kp, ki, kd)

  def get_error(self, target_distance: float):
    distance = self.robot.get_distance(self.target)
    return distance -target_distance -Mobile.sensor_accuracy*10 # margin with initial distance
```

[2]Each simulation is repeated 30 times to deal with pseudo-random number generation issues.

That works well; for example, for $\sigma = 0.0325$ m the chasing robot always stays above the safety distance (at least 2.05 m). However, its overall performance is not so good, with an average distance of only 2.55 m, which is too conservative. The goal of this article is to investigate whether we can do any better by considering the distance returned by the sensors as an explicit random variable, i.e., treating this kind of random variables as first class entities in our adaptive control programs.

3 UNCERTAINTY-AWARE CONTROL SYSTEMS

In this section, we describe our proposal and illustrate it on the chasing robot example discussed above. Our approach consists of three steps:

- Identify the possible sources of uncertainty
- Explicitly represent them so that they can be managed
- Incorporate them into the control loop to improve the decision process of the control system in such a way that it can manage the identified uncertainties.

3.1 The Chasing Robot Example Revisited

In our simple chasing robot example, we only consider the uncertainty due to the estimation of the distance to the target. Since we know that the distance sensor returns a value within a normal distribution with standard deviation of σ, we are going to explicitly model that return value with a random variable X having this σ standard deviation. Then, in the control loop of the robot, we can use X to make decisions (this approach is sometimes called adaptive control).

For most systems, including mission critical ones, reliability is not absolute but estimated in terms of the probability of not failing the mission. Depending on the stakes, this probability can range from 0.999 to 0.9999999999 or more and is usually made readable by "counting the nines." For instance, a probability of 0.999 is called "3 nines."

In the case of our chasing robot, this makes it possible to let the user choose the level of risk she wants to take and use it as a parameter for controlling the robot. For example, for a 3-nines probability of keeping the safety distance, probability theory tells us that we need to take a margin of 3.29σ, while for 5 nines we need 4.41σ. We can easily model that in Python using the class `ufloat` from the `uncertainties` package that provide basic support for random variables:

```
class ProbabilisticController (BaseLine):
  def __init__(self, robot: Robot, kp: float, ki: float, kd: float, risk: float):
    super().__init__(robot, kp, ki, kd)
    self.confidence = self.confidence_interval[risk]

  confidence_interval = {.90:1.64, .99:2.67, .999:3.29, .9999:3.89, .99999:4.41}

  def get_error(self, target_distance: float) -> ufloat:
    distance = self.robot.get_distance(self.target)
    return ufloat(distance, abs((1+0.25*distance)*Mobile.sensor_accuracy))


class PXNinesController (ProbabilisticController):
  def __init__(self, robot: Robot, kp: float, ki: float, kd: float, risk: float):
    super().__init__(robot, kp, ki, kd, risk)

  def get_error(self, target_distance: float) -> float:
    distance = super().get_error(target_distance)
    return distance.nominal_value - distance.std_dev*self.confidence -target_distance # p>risk
```

Figure 2 summarizes the performance of various versions of our chasing robot controller when σ increases. It is obtained by running each chasing robot for 30 seconds and plotting both its real minimum and average distance to the target. The experiment is repeated 30 times to account for random perturbations.

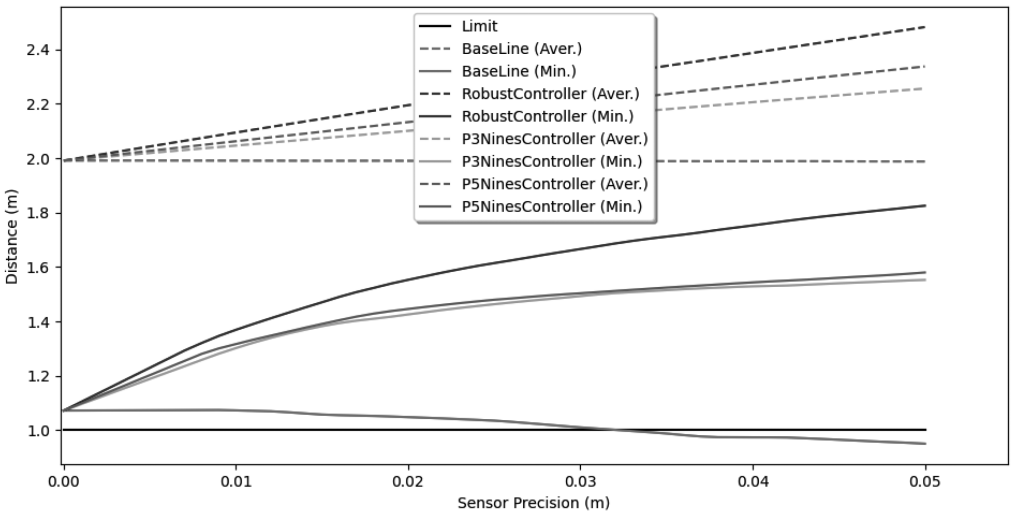


Fig. 2. Performance of various control algorithms for the chasing robot example.

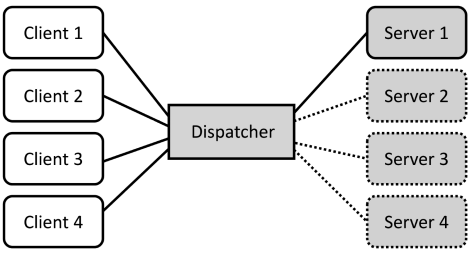


Fig. 3. The ZNN.com system architecture.

When the precision of the distance sensor degrades, the `BaseLine` controller fails, going below the safety distance (1 m) as explained above. In contrast, the `RobustController` is too cautious, and we can see how its performance degrades as the precision of the sensors decreases: It stays too far behind the target. However, our two versions of an uncertainty aware controller (`P3Nines` and `P5Nines`) are making a good tradeoff between quality of service (average distance) and safety (not going under 1 m).

Since this example is very simple, the basic support for random variables currently found in, e.g., Python, is good enough to handle it. However, as soon as computations need to be done manually on random variables, things get much more complicated to handle at the programmatic level without strong support for treating random variables as first class entities, including support for comparison operators among them. Let us demonstrate that in the next section with another example, the ZNN system, which is a bit more complex than the chasing robot system.

3.2 The Znn.com System

Znn.com is a news service that is commonly used as an example of a self-adaptive system [35]. Its architecture is shown in Figure 3. The system comprises several *servers*, some of which can be inactive, and a load balancer (d) in charge of receiving *requests* from *clients* and selecting the active server that will process them. The system monitors the dispatcher and the servers to make decisions to optimize its behavior.

One typical example is the invariant stating that the current system response time R for any request should always stay below some threshold $RMax$. When a request is received, the dispatcher tries to find an active server able to process the request and respond to the originating client within the required time limit. If none is found, then the dispatcher activates one of the inactive servers and sends the pending request to it. If all servers are active and none of them can ensure processing the request within the required timeframe, then the request is denied and returned to the client. Server activation takes some time, the so-called starting latency. If a server is inactive for more than a certain time, then it shuts itself down to save energy and waits for the dispatcher to activate it when required.

Our exemplar system comprises one dispatcher and four servers, all initially inactive. Four clients generate requests at a given pace. For simplicity, we assume that the processing time of all requests is the same, namely 20 time units.

To simulate this ZNN system we decided to use UML executable models to show how our approach can be used with different paradigms, i.e., it can work with both modeling and programming languages. In particular, we have used UML and OCL [28] to specify the ZNN system and the **UML-based Specification Environment (USE)** [14] to execute the UML system specifications. The use of such high-level specifications provides interesting benefits, such as that we can abstract away from any concrete implementation, focusing on high-level models that allow runtime verification of system properties, and thus they require a very lightweight development process. Furthermore, these UML models could be formally analyzed using high-level validation tools [15] or transformed into concrete implementations if needed.

3.2.1 Baseline Behavior. We simulated the system using different workloads, depending on the pace at which the clients issue their requests. In the *low workload* scenario, each client issues a request every 30 time units. Under *medium workload*, requests are issued every 20 time units (i.e., same as the processing time of requests). This simulates a stable system that works at optimal performance. In the *heavy workload* scenario, clients issue their requests every 18 time units to ensure that servers cannot process all incoming requests. Assuming that the response time limit is 62 time units (i.e., $RMax = 62$) and that all clients issue their requests at the same time, the simulations show that no requests are overdue in any scenario and that no requests are denied either under low and medium workloads. However, around 7.53% of the requests are denied under heavy workload, as theoretically expected: The throughput of the four clients is, respectively, 2.6666, 4.0, and 4.4444 requests per time unit in each scenario, while the combined processing capability of the servers is always 4.1333. This means that 7.5267% of the requests ($= 4.4444/4.1333 - 1.0$) should be denied in the heavy workload scenario, as the simulations corroborate.

The three key decisions that the components of the system should make are (1) whether a server can accept a request, because it is able to respond to it in time; (2) whether a request is ready to be responded, because its has been processed; and (3) whether a request is overdue.

These decisions can be implemented by the following query operations of a server (they are specified in OCL):

```
fits(r:Request):Boolean =
    r.finishTime - r.arrivalTime + self.swapTime < self.config.RMax

hasFinished(r:Request,now:Real):Boolean =
    r.finishTime <= now

isOverdue(r:Request):Boolean =
    (self.actualFinishTime - self.arrivalTime) > self.config.RMax
```

Table 1. Baseline System: Percentage of Denied and Overdue Requests under Medium (20) and Heavy (18) Workloads

Proc.Time Uncert.	Denied-20	Overdue-20	Denied-18	Overdue-18
0.0	0.00%	0.00%	7.81%	2.52%
0.1	0.00%	0.00%	7.81%	2.53%
0.2	0.00%	0.00%	7.27%	2.66%
0.3	0.00%	0.00%	7.54%	2.76%
0.4	0.00%	0.00%	7.27%	3.01%
0.5	0.00%	0.03%	7.81%	2.27%
0.6	0.00%	0.03%	7.81%	3.32%
0.7	0.00%	0.03%	7.81%	3.33%
0.8	0.00%	0.05%	7.54%	3.48%
0.9	0.00%	0.13%	7.27%	3.76%
1.0	0.00%	0.16%	7.54%	4.52%

In the last operation, `isOverdue()`, the value of attributes `arrivalTime` and `actualFinishTime` are set by the server when the request is accepted and when it is removed from its queue of pending requests, respectively.

3.2.2 A More Realistic Behavior. As mentioned in the Introduction, reality is different from simulations, especially in the case of physical systems that are subject to different types of uncertainties. In this article, we assume that data collected from the environment can never be directly observed without noise and also that an accurate model of the environment cannot be obtained [7, 27]. These are inherent characteristics of any physical system and therefore cannot be neglected.

Let us consider here two sources of uncertainty that may affect our system:

- The actual time taken by a server to process a request is not a fixed value, but a random variable. This may cause the actual processing time to be greater than the expected one. Thus, a server may accept a request, because, according to its calculations, it is able to respond to it within the required time, but then the actual processing time is longer than expected and the response is delayed.
- Clocks have some imprecision. Even if we assume that the deviations are only of micro-time units (i.e., 10^{-6}), this can cause some comparisons between time variables to fail. We will see how this can cause some requests to be delayed, because when the system checks if a request has finished, the comparison fails and the request has to wait for the next clock cycle to be answered, hence causing unnecessary delays.

These two aspects correspond to measurement uncertainties [18], which affect the values of the variables managed by the simulation. To evaluate the effect of such uncertainties, we developed a simulation system (hereinafter, the `Baseline` system) where the values of the variables could have small variations, due to the lack of precision of the sensors (e.g., the clock readings) or indeterminacy of the environment (e.g., variations in the processing times of the requests due to other concurrent executing tasks running in the server).

To illustrate the effects of such uncertainties, Table 1 shows the percentage of denied and overdue requests under medium and heavy workloads for different levels of processing time imprecision. The first column displays the value of the processing time uncertainty, which ranges between 0 and 5% of the processing time of each request, i.e., between 0 and 1 time units. This is used by the system to assign each request a deviation from its expected processing time, which simulates

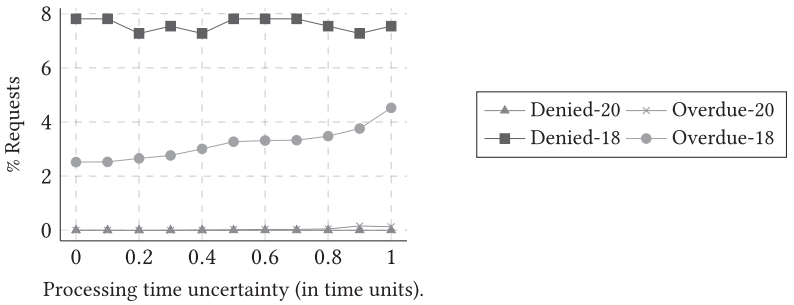


Fig. 4. Baseline system: Percentage of denied and overdue requests under medium (20) and heavy (18) workloads.

a more realistic situation where the actual processing times of requests are not perfect values but random variables. Of course, the heavier the workload the worse the results.

Figure 4 shows these results in a graphical way. Note how the percentages of denied requests maintains around the "expected" theoretical value of 7.53% value. Sometimes it is even lower, because the requests are accepted based on their estimated processing times of 20 time units and not on their actual processing times. In some cases, these decisions led to overdue responses. This is similar to the situation described in the Introduction, where the behavior of the machines and parts on the assembly line did not correspond to what was expected from the simulations, and some parts fell off the trays, or the gantry grippers failed to grasp the parts.

3.2.3 Taming Uncertainty: Robust Approach. As previously mentioned, robust control methods aim to achieve robust performance and/or stability in the presence of bounded uncertainty [1]. These methods normally use interval arithmetic [17, 40]. Thus, instead of representing a value as a single number x, interval arithmetic represents each value as a range of possibilities defined by the interval $[x_m, x_M]$ that contains x. The most common use is in software to keep track of rounding errors in calculations and of uncertainties in the knowledge of the exact values of physical and technical parameters, so that reliable results can be guaranteed.

In the ZNN example, we implemented a robust solution using an interval of $\pm 5\sigma$ to ensure more than 6-nines precision in the processing time (what we have called 1.0 confidence in Table 2). This can be implemented by simply changing method `fits()` to include such a safety interval (note that in our case we use $\sigma = 1$, and therefore $5\sigma = 5$):

```
1   fits(r:Request):Boolean =
2     r.est_finishTime - r.arrivalTime + self.swapTime + 5.0  -- safety interval added
3     < self.config.RMax
```

However, it still does not work. We still get 0.08% overdue requests (see first row of Table 2, under column CI:PTU). Analyzing the causes, we realized that this is because we need to consider the second source of uncertainty, i.e., the imprecision of the clock. As mentioned above, a slight variation of the clock readings may cause that we miss one timestep. For example, the actual finishing time of a request is 30.0, but the clock time is 29.9999999. Then, the comparison `r.finishTime <= now` returns false and the request has to wait for the next timestep.

To tackle this issue using a robust control approach, we substitute the uncertain variable (in this case, the clock readings) by an interval and use the interval in the comparison. With this, query `hasFinished()` is implemented as follows:

```
1   hasFinished(r:Request,now:Real):Boolean =
2     (now - r.finishTime).abs() <= 1.0)
```

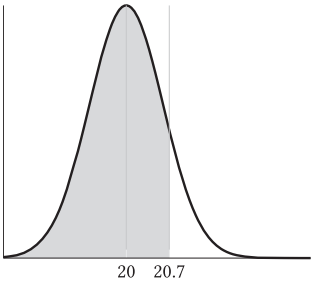


Fig. 5. Comparing one random variable and one real number.

This change has the desired effect, and no overdue requests are produced. However, the approach taken by robust control systems is too coarse grained and conservative, normally wasting too many resources or producing sub-optimal results, as illustrated in the Chasing Robot example. This is where adaptive control systems come into play.

3.2.4 Taming Uncertainty: Adaptive Control. Adaptive control methods do not need *a priori* information about the bounds on the uncertain or time-varying parameters. In contrast, their safety bounds can dynamically adapt to improve the decisions made by the control loop. Random variables are commonly used instead of fixed-length intervals, and fixed-bound intervals become **confidence intervals (CI)**. For example, assuming that the actual processing time of a request follows a normal distribution $X \sim N(x, \sigma)$ with standard deviation σ, we will use such a random variable X instead of the real value x [18]. Comparison are no longer Boolean values but become probabilities [3].

To illustrate the differences among the crisp, robust, and adaptive approaches, consider the real values $x = 20.0$ and $y = 20.7$. Using real arithmetic, $x < y =$ true. Assuming a precision of $\sigma = 0.6$ in the values of x, using a robust control approach we would define an interval of $\pm 5\sigma$ around x, i.e., $\widehat{X} = [17, 23]$. In this case, given that $20.7 \in \widehat{X}$, then $x < y =$ false.

Finally, using the adaptive control strategy, variable x would be modeled by a random variable $X \sim N(20, 0.6)$, and the comparison $x < y$ becomes $P(X < 20.7) = 0.878$. Such probability coincides with the area shaded in blue in Figure 5.

To realize this adaptive control approach, we only need to change the implementation of the two query operations that compare the two random variables of our example, namely `finishTime` and `now`:

```
1   fits(r:Request):Boolean =
2     r.finishTime - r.arrivalTime + self.swapTime
3     + self.config._processingTimeUnc * self.tolerance(config.robustness) < self.config.RMax
4
5   hasFinished(r:Request,now:Real):Boolean =
6     (now-r.finishTime).abs() <= self.config._clockUnc * self.tolerance(config.robustness))
```

In these specifications, variables `_processingTimeUnc` and `_clockUnc` correspond to the precision of the requests processing times and the clock (1.0 and 10^{-6}, respectively). They are both stored as attributes of class `Config`. Operation `tolerance()` returns the number of σ's required to obtain a given robustness, i.e., confidence. For example, assuming that the variables follow normal distributions, to obtain a confidence of 3 nines (0.999) we need 3.29σ.

The results obtained for different levels of confidence using an adaptive control strategy are shown in columns CI:PTU and CI:PTU+CU of Table 2 and graphically in Figure 6. Column CI:PTU shows the percentage of overdue requests taking into account only the request **performance time**

Table 2. Percentage of Overdue Requests Depending on the Confidence Level

Confidence	CI:PTU	CI:PTU+CU	Stochastic
Robust (1.0)	0.08%	0.00%	0.00%
0.9999	0.20%	0.11%	0.00%
0.999	0.72%	0.12%	0.00%
0.99	0.84%	0.15%	0.00%
0.98	0.85%	0.16%	0.00%
0.95	0.86%	0.22%	0.02%

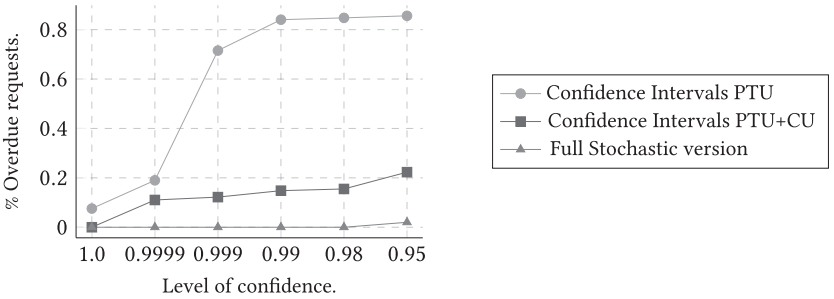


Fig. 6. Percentage of overdue requests depending on the confidence level.

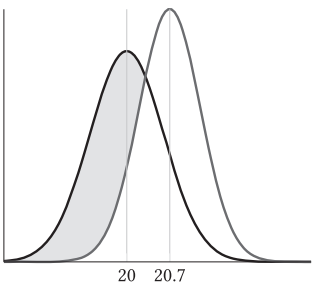


Fig. 7. Comparing two random variables.

uncertainty (PTU). In turn, Column CI:PTU+CU shows the percentage of overdue requests taking into account both the request performance time uncertainty and the **clock uncertainty (CU)**.

3.2.5 Taming Uncertainty with Random Variables. The adaptive strategy that uses random variables and confidence intervals to model some of the values of the system but still uses *crisp* values with the rest. Our claim is that this is not realistic, because in physical systems there are no exact values—they *all* are subject to uncertainty, numerical approximations, or both. This is why physical variables should never be modeled by means of real numbers but using *uncertain* numbers.

For example, assuming that $X \sim N(20, 0.6)$ and y becomes a random variable $Y \sim N(20.7, 0.5)$, we get that $P(X < Y) = 0.48$. This is graphically depicted in Figure 7, where the shaded area shows the value of the comparison (check it against the area in Figure 5). By changing the standard deviation of the variables we obtain different values for that probability. The larger the variance, the more difficult it is to tell the two values apart and vice versa.

For implementing this approach, we have used the Java library of datatypes extended with measurement uncertainty defined in Reference [3], which is also implemented in the tool USE to

support uncertain numbers in UML and OCL [29]. Essentially, this library extends the basic UML and OCL primitive datatypes (Real, Boolean, Integer,. . .) with uncertainty by defining super-types for them, as well as the set of operations defined on the values of these types. Thus, `Real` values with uncertainty are represented in terms of `UReal` values, which are composed of pairs (x, u), also noted as $x \pm u$, where x is the value, and u represents its uncertainty as the standard deviation of its possible variations, according to the GUM international standard [18]. Likewise, a `Boolean` value b is lifted to a `UBoolean` value B, which is a pair $B = (b, c)$ in which c is a real number between 0 and 1 that represents the confidence we assign to b. Comparison operators between `UReal` variables return `UBoolean` values. For example, if $a = 2.0 \pm 0.3$ and $b = 2.5 \pm 0.25$, then $a < b$ = `Boolean(true,0.893)`, meaning that $a < b$ with a confidence of 0.893 [3]. Projection operation `confidence()` applied to an uncertain Boolean returns a probability, i.e., the confidence assigned to that Boolean. Uncertain values then become first-class entities of our models and can be managed and operated in a natural way by the underlying type system. Propagation of uncertainty through operations is transparently taken care of by the type system, and comparisons are lifted to `UBoolean` values when required. This greatly simplifies the management of uncertain numbers in both Java programs and UML/OCL models.

Using these extended datatypes and operations, the critical queries in the ZNN system can be restated as follows:

```
1  fits(r:Request):Boolean =
2      (r.finishTime - r.arrivalTime + self.swapTime < self.config.RMax).confidence()
3        >= self.config.robustness
4
5  hasFinished(r:Request,now:UReal):Boolean =
6      (now >= r.finishTime).confidence() >= self.config.robustness
7
8  isOverdue(r:Request):Boolean =
9      (r.responseTime >  self.config.RMax).confidence() >= self.config.robustness
```

We can see how we can now play with the level of confidence (robustness) required, thus being able to quantify in a more precise way the degree of uncertainty with which we make decisions.

The last column (Stochastic) of Table 2 shows the percentage of overdue requests of a system that is simulated using the strategy of representing physical attributes with random variables, i.e., uncertain reals. This is also shown graphically in Figure 6. The last 0.02% is to be expected, because we are assuming a confidence of only 0.95 in our decisions. As in the example of the chasing robot, with this strategy we can obtain more faithful simulations and therefore more accurate results.

4 DISCUSSION

So far we have shown how we are able to capture the inherent uncertainty of the possible values of the attributes used in a control system by means of random variables, and the benefits of handing them as first-class entities of our programs or models with the appropriate libraries. This section provides some methodological guidance on how our proposal can be used. Then, we discuss some further advantages and possible limitations of our proposal and finish with some open questions.

Note that the two examples we have used to illustrate our approach in this article come from the realm of physical systems, although our proposal is also applicable to scenarios where non-physical systems are considered. Ultimately, what we propose is a more effective modeling approach for any application where uncertainty plays a role by considering uncertainty as a first-class entity. For example, our proposal can be used in scenarios where we are uncertain of the values of some parameters, because the system is virtual, and decisions made by some underlying real system may materialize different values over time. Likewise, it is applicable in situations where we have

to model the duration of tasks in software development environments or where numerical errors in computations may lead to inaccurate results.

4.1 Methodological Guidelines

If one wants to leverage the use of uncertain variables in a consistent way across an application, then the following three steps might be followed:

- First, identify all possible *primary* sources of uncertainty in a model or a program. In cyber-physical systems, that is all the variables that store values that are read from system sensors. In other systems, it might be variables whose values depend on the hardware platform on which the program would execute. This is, for example, what may occur for virtual machines vs. hypervisors, e.g., in terms of the completion time of a task.
- Then, for each uncertain variable, find a law that models its probability distribution (e.g., normal, uniform, etc.). Alternatively, if the law is unknown, then one could resort to measurements and store the measured distribution as a random variable (i.e., Type A evaluation of uncertainty [18]).
- Finally, propagate uncertainty across the source code. That is, each time an uncertain variable is used in a computation, the result of the computation also becomes an uncertain variable. For instance, if x and y are `UReal` variables, then the result b of their comparison, $b = x \leq y$, must be an uncertain variable, namely a `UBoolean`. Some easy-to-implement static analysis can help ensure that this rule is enforced across the source code of an application.

This approach relies on the use of libraries supporting operations across uncertain basic data types (e.g., UBooleans, UIntegers, UReals). Several such libraries already exist for Python (with some limitations with respect to comparing uncertain variables), for Java (for instance, the one we have developed in a previous work and that is freely available in our Github repository https://github.com/atenearesearchgroup/uncertainty), and for UML/OCL, the latter actually relying on the Java one.

4.2 Advantages

First, our proposal allows us to capture the uncertainty of the data collected from the environment of the system in an accurate way. This uncertainty basically depends on the precision of the sources of these data, i.e., on the possible variations of their values. Such precision values become input parameters for the control algorithms.

Using these input parameters, we have shown how explicit uncertainty management allows us to choose the level of "risk" we want to take between the too-naive (i.e., crisp) vs. the too-conservative (i.e., robust) approaches. Thus, such a level of risk (e.g., the acceptable failure rate, the admissible deviations from a theoretical ground truth, or the allowable degree of uncertainty in the value of an attribute) becomes a parameter we can play with—something essential for, e.g., software certification.

In this way, we can make tradeoffs to achieve acceptable compromises depending on the precision of the sensors, which is not the case now, as current control algorithms tend to use an all-or-nothing strategy. For the "all" case, they decide the level of risk they want to take (maybe none in case of critical systems) and then build the control system based on this level. However, in our proposal the level of risk is a parameter of the controller, and we can decide (even at runtime) the tradeoff we want to make and thus the level of risk acceptable for our system.

Working on the opposite direction, based on a given level of risk (or of robustness) and on the expected behavior of the system, we can decide about the required precision of the sensors that we

need to install in our system to ensure that level of risk. This is very useful for systems where the costs of their parts (e.g., the sensors) are important and should be maintained under control but still ensure the required level of precision. Note that the use of random variables provides more accurate estimations than those provided by current control algorithms.

4.3 Potential Limitations (and How to Mitigate Them)

Normality of the distributions. In the first place, our proposal makes some assumptions that might not hold in all situations. For example, we suppose that the random variables that represent the uncertainty of the attributes of our control algorithms follow normal distributions (as usually done in measurement [18]). Should this not be the case, one solution would be to use Chebyshev's inequality to determine the range of standard deviations around the mean and thus decide the level of robustness we are accepting. Note that the Chebyshev's inequality works for any type of distribution. Its practical usage is similar to the 68–95–99.7 rule, which applies only to normal distributions. Chebyshev's inequality is more general, stating that a minimum of just 75% of values must lie within two standard deviations of the mean and 88.89% within three standard deviations. Although this is a more conservative estimation that that used for the normal distribution, it is still very useful to ensure acceptable levels of risk.

Variable independence. Second, in this work we have made some assumptions regarding the independence of the attributes when operating with their associated uncertainty using the closed-form solution. If such an independence cannot be ensured, then there are several ways to deal with dependent (i.e., correlated) variables. First, if we know their covariances, then most specifications and implementations support closed-form expressions of the operations with uncertainty when variables are dependent. However, the values of such covariances are rarely known by users, and therefore they are not very useful in common practice. An alternative solution consists of using the implementation of the operations based on samples (i.e., Type A evaluation of uncertainty [18]). This is also the approach proposed by ISO, which is very general and powerful. However, it may have a significant impact on the performance of the evaluation of the operations, given that they have to be applied to the samples, hence introducing an overhead proportional to the sample size.

Precision estimation. Sometimes, estimating the precision of data collected from the environment is not an easy task. Some common factors that make this task difficult include the lack of information about the data sources and their uncertainty; the effect of unreliable communication channels and networks, which can produce large distortions in the values of the input data; or the degradation of data sources or communication channels themselves, which can make the quality of the data received increasingly worse. In this article, we have assumed that the precision of the input data is known and constant. As for the latter, it would not be difficult to deal with variable precision, since functions can be used to define the uncertainty of the `UReal` values. How to deal with unknown and imprecise precision (i.e., a type of second-order uncertainty) remains part of our future work.

Usage complexity. This proposal adds a certain level of complexity related to the need to compare probabilistic values, which is not required in more conventional approaches, such as those based on the inclusion of error bounds. Instead of a single comparison, the developer must provide a piece of code that returns a Boolean value under the probabilistic comparison, depending on the confidence level that can be accepted. While this introduces some additional complexity, at the same time it clearly provides more refined and accurate results in terms of the final quality of the developed model.

4.4 Open Questions

In addition to the potential benefits and limitations of our proposal, this section discusses some open issues that we have found during its evaluation.

Domain expert implication. When incorporating measurement uncertainty information into a model, sometimes it is difficult to identify the attributes that are subject to uncertainty. In general, all attributes that represent physical variables should be subject to uncertainty, but there might be others. For example, some constants should be endowed with uncertainty, too. The variability of the duration of tasks in certain processes, or of the cost of a given product due to currency exchange fluctuations, are uncertainties that need to be estimated. For this, the judgment of the domain expert is essential, and communication with them is needed to clarify which attributes should be endowed with this kind of information.

Representing and operating with uncertainty. There are different ways of representing measurement uncertainty, especially for the uncertainty associated to numeric values. They include ranges, probability distributions of the values, or the standard deviation of the variability of the measured attribute. From all the available alternatives, we decided to use a library that implements the ISO VIM recommended representation and management of measurement uncertainty, as defined in the GUM [18], which is also the notation used in most engineering disciplines. Ranges and other kinds of possible expressions of the measurements deviations can be reduced to this representation [18]. Similarly, Bayesian probability [5], fuzzy logic [46], or uncertainty theory [24] can be used to assign confidence to uncertain Boolean values. All these theories have advantages and limitations (see, e.g., References [6, 21, 24]), but, as previously mentioned, we decided to use Bayesian probability, which is the one that, in our opinion, is the most well known and easily understood by software engineers. A proper comparison analysis between the different approaches is left for future work. Likewise, the use of Type A representation of uncertainty, which uses the value samples instead of closed-form equations to represent and propagate the uncertainty is something that we would like to explore further. Although *a priori* this would have a significant input on the performance of the uncertainty analysis, the use of cheap hardware accelerators such as graphic cards might provide effective solutions for these simple vector operations, and therefore we could deal with uncertainty in a more statistically precise manner.

5 RELATED WORK

Uncertainty in control systems and their simulation has been traditionally represented and managed using two main approaches.

In the first place, intervals to represent the possible values of uncertain attributes have been extensively used in the simulation domain. For example, Fujimoto [11, 25] use time intervals to deal with the concepts of approximate time and approximate time event ordering in the context of DEVS [43]. In their proposal, two events are considered concurrent if the intervals representing their timestamps have a non-empty intersection. Other authors have proposed to introduce uncertainty on the spatial properties of the model for obtaining speed-ups [13, 31]. Saadawi and Wainer also explored replacing time datatype in DEVS models by intervals in their RTA-DEVS formalism [34]. Furthermore, two new extensions to DEVS, called UA-DEVS and IA-DEVS, provide methods to specify uncertainty in the state, input, and output variables in addition to the time variable [40]. The former defines a formal specification of models including uncertainty specifications as intervals. The latter enables the simulation of UA-DEVS models based on computational constraints (time, memory, etc.). This separation of concerns allows the domain expert to define the model once and then simulate it with different constraints without redefining the model. Other

approaches, such as Reference [20], make conservative decisions based on intervals to robustify the specification of controllers of cyber-physical systems so that they satisfy safety requirements under uncertain conditions.

We see two major limitations of approaches based on intervals for specifying the possible values of uncertain variables. On the one hand, they are very coarse grained, as we have seen in the examples shown above, which results in very conservative (also called *cautious*) simulations [42]. On the other hand, specifying and operating with intervals require a significant effort by the modeler, since there is no direct support for making computations with them, such as arithmetic operations or comparisons, which are really burdensome and error-prone tasks.

Other set of works study the relationship the uncertainty of the input parameters and that of the simulation results, aiming at defining measures for risk quantification under input uncertainty. In general, there are two sources of uncertainty in a typical stochastic simulation experiment: the extrinsic uncertainty on input parameters (also called input parameter uncertainty) and the intrinsic uncertainty on output response (referred to as stochastic uncertainty) that reflects the inherent stochasticity of the system. The variability of simulation output response depends on both input uncertainty and stochastic uncertainty. Some authors [16, 45] propose nested Monte Carlo simulation approaches to estimate them. Others [4] propose statistical methods for the calculation of confidence intervals for the mean of a simulation output. As in our case, they obtain more accurate results than those proposals that use interval arithmetic or very conservative (i.e., robust) estimations. However, both the complexity of their calculations and their computational costs might hinder their applicability. In our case, the fact that we assume normal distributions and that uncertainty propagation is achieved using closed-form solutions mitigate these issues.

Another group of papers provides alternative approaches to exploit approximation (hence uncertainty) for improving the tradeoff between performance and representativeness of simulation output, under uncertain event occurrence [11, 31] or using approximated rollbacks [30]. Our proposal is orthogonal to these approaches, as each focuses on different aspects of uncertainty.

Similarly, existing schemes for adaptive control used in industry provide reasonable heuristic approaches, although they have the limitation that parameter uncertainties are not usually taken into account in the design of the controller. This has led to the notion of *dual control* [9, 39], which addresses this issue by considering parameter uncertainties. In particular, explicit dual control algorithms, such as the ones used in our examples, are based on the minimization of cost functions defined in terms of control losses and uncertainty measurements (the measure of precision of the parameter estimation) [37]. Basically, the controller has a dual action: It follows the control goal, i.e., the system output cautiously tracks the desired reference value, and it excites the plant so that the control quality becomes better in future time intervals. One of the known problems with such control algorithms is that they are complicated and not always feasible to implement in practical problems [37], which hinders their applicability in real systems. What we have shown in this article is that the use of a type system that provides basic support for explicitly representing and operating with uncertain attributes and propagating their associated uncertainty transparently greatly simplifies these problems. This makes it possible to obtain the advantages of dual control algorithms while minimizing their limitations.

In this context, the explicit representation of uncertainty is also a challenge, especially in the context of software models. The survey [38] covers current approaches, although significant challenges remain to be addressed. In particular, there are very few libraries for programming or modeling languages that support measurement uncertainty, i.e., the representation and operation of uncertain datatypes [3]. Even those that support the propagation of uncertainty (e.g., References [2, 22, 23, 41]) are quite complex to use and do not support the comparison between uncertain numbers. This is a general problem that we have observed in most uncertainty mod-

eling proposals: They only deal with uncertain reals. However, in the physical world, all other primitive data types also have uncertain values. In particular, logical variables representing decisions or comparisons between quantities rarely have crisp true or false values. Instead, extensions to the Boolean logic enable dealing with this type of uncertainty, including probability theory [5, 8], possibility theory (based on fuzzy logic [33, 46]), plausibility (a measure in the Dempster–Shafer theory of evidence [36]), and uncertainty theory [24]. These approaches assign different probabilities to propositions, rather than truth values, and probability formulas replace truth tables. From the surveyed literature, in this article we use the proposal presented in Reference [3], which provides a Java library that supports all UML and OCL primitive datatypes endowed with uncertainty. Moreover, as mentioned above, probabilities tend to be easier for engineers to understand and manage than other measures that quantify confidence or the likelihood of failure.

6 CONCLUSIONS AND FUTURE WORK

In this article, we have proposed to include the sources of uncertainty in system models as first-class entities using random variables to simulate control systems more faithfully, including not only the use of random variables to represent and operate with uncertain values, but also to represent decisions based on their comparisons. We have illustrated the problem with the toy example of a Chasing Robot and validated our approach on the ZNN case study, which is a standard for the self-adaptive systems community.

Uncertainty is inherent in cyber-physical systems, and we strongly believe that it should be handled explicitly at every level from requirements to design to code and validation. We have shown that this is not so difficult to implement by leveraging emerging libraries for supporting sound computations on random variables. We hope this article would help in triggering a wider adoption of stochastic approaches for software controlling cyber-physical systems.

ACKNOWLEDGMENTS

We thank the reviewers for their very valuable comments and suggestions, which helped us significantly to improve the article.

REFERENCES

[1] Michael Athans. 1971. Editorial on the LQG problem. *IEEE Trans. Autom. Contr.* 16, 6 (1971), 528.

[2] Michaël Baudin, Anne Dutfoy, Bertrand Iooss, and Anne-Laure Popelin. 2016. *OpenTURNS: An Industrial Software for Uncertainty Quantification in Simulation*. Springer, 1–38. https://doi.org/10.1007/978-3-319-11259-6_64-1. https://openturns.github.io/.

[3] Manuel F. Bertoa, Loli Burgueño, Nathalie Moreno, and Antonio Vallecillo. 2020. Incorporating measurement uncertainty into OCL/UML primitive datatypes. *Softw. Syst. Model.* 19, 5 (2020), 1163–1189. https://doi.org/10.1007/s10270-019-00741-0

[4] R. C. H. Cheng and W. Holland. 2004. Calculation of confidence intervals for simulation output. *ACM Trans. Model. Comput. Simul.* 14, 4 (October 2004), 344–362. https://doi.org/10.1145/1029174.1029176

[5] Bruno de Finetti. 2017. *Theory of Probability: A Critical Introductory Treatment*. John Wiley & Sons.

[6] Didier Dubois and Henri Prade. 1993. Fuzzy sets and probability: Misunderstandings, bridges and gaps. In *Proceedings of the IEEE Conference on Fuzzy Systems*. IEEE, 1059–1068. https://doi.org/10.1109/FUZZY.1993.327367

[7] Naeem Esfahani and Sam Malek. 2013. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*. LNCS, Vol. 7475. Springer, 214–238.

[8] W. Feller. 2008. *An Introduction to Probability Theory and Its Applications*. Wiley.

[9] Nikolai M. Filatov and Heinz Unbehauen. 2000. Survey of adaptive dual control methods. *IEEE Proc. Contr. Theory Appl.* 147, 1 (2000), 118–128. https://doi.org/10.1049/ip-cta:20000107

[10] Antonio Filieri et al. 2015. Software engineering meets control theory. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*. IEEE Computer Society, 71–82. https://doi.org/10.1109/SEAMS.2015.12

[11] Richard Fujimoto. 1999. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings of the International Conference on Principles of Advanced Discrete Simulation (PADS'99)*. IEEE Computer Society, 46–53. https://doi.org/10.1109/PADS.1999.766160

[12] David Garlan. 2010. Software engineering in an uncertain world. In *Proceedings of the Future of Software Engineering Research (FoSER'10)*. 125–128. https://doi.org/10.1145/1882362.1882389

[13] Valerio Gheri, Giovanni Castellari, and Francesco Quaglia. 2008. Controlling bias in optimistic simulations with space uncertain events. In *Proceedings of the IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT'08)*. IEEE Computer Society, 157–164. https://doi.org/10.1109/DS-RT.2008.37

[14] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.* 69, 1-3 (2007), 27–34. https://doi.org/10.1016/j.scico.2007.01.013

[15] Martin Gogolla, Frank Hilken, and Khanh-Hoang Doan. 2018. Achieving model quality through model validation, verification and exploration. *Comput. Lang. Syst. Struct.* 54 (December 2018), 474–511. https://doi.org/10.1016/j.cl.2017.10.001

[16] Michael B. Gordy and Sandeep Juneja. 2010. Nested simulation in portfolio risk measurement. *Manage. Sci.* 56 (August 2010), 1833–1848. https://doi.org/10.1287/mnsc.1100.1213

[17] IEEE 1788-2015. 2015. IEEE Standard for Interval Arithmetic. Retrieved from https://standards.ieee.org/ieee/1788/4431/.

[18] JCGM 100:2008. 2008. *Evaluation of Measurement Data—Guide to the Expression of Uncertainty in Measurement (GUM)*. Joint Com. for Guides in Metrology. http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf.

[19] Deepali Kholkar, Suman Roychoudhury, Vinay Kulkarni, and Sreedhar Reddy. 2022. Learning to adapt—Software engineering for uncertainty. In *Proceedings of the International Solvent Extraction Conference (ISEC'22)*. ACM, 21:1–21:5. https://doi.org/10.1145/3511430.3511449

[20] Tsutomu Kobayashi, Rick Salay, Ichiro Hasuo, Krzysztof Czarnecki, Fuyuki Ishikawa, and Shin-ya Katsumata. 2021. Robustifying controller specifications of cyber-physical systems against perceptual uncertainty. In *Proceedings of the NASA Formal Methods Symposium (NFM'21), LNCS, Vol. 12673*. Springer, 198–213. https://doi.org/10.1007/978-3-030-76384-8_13

[21] Bart Kosko. 1990. Fuzziness vs. probability. *Int. J. Gen. Syst.* 17, 2–3 (1990), 211–240. https://doi.org/10.1080/03081079008935108

[22] Eric O. Lebigot. 2016. Uncertainties Package. Retrieved May 30, 2022 from https://pythonhosted.org/uncertainties/.

[23] Abraham Lee. 2013. SOERP Uncertainties Package. Retrieved May 30, 2022 from https://pypi.org/project/soerp/.

[24] Baoding Liu. 2018. *Uncertainty Theory* (5th ed.). Springer.

[25] Margaret L. Loper and Richard M. Fujimoto. 2000. Pre-sampling as an approach for exploiting temporal uncertainty. In *Proceedings of the International Conference on Principles of Advanced Discrete Simulation (PADS'00)*. IEEE Computer Society, 157–164. https://doi.org/10.1109/PADS.2000.847159

[26] Giovanni Lugaresi and Andrea Matta. 2018. Real-time simulation in manufacturing systems: Challenges and research directions. In *Proceedings of the Winter Simulation Conference (WSC'18)*. IEEE, 3319–3330. https://doi.org/10.1109/WSC.2018.8632542

[27] Sara Mahdavi-Hezavehi, Paris Avgeriou, and Danny Weyns. 2017. *A Classification Framework of Uncertainty in Architecture-Based Self-Adaptive Systems With Multiple Quality Requirements*. Morgan Kaufmann, Boston, 45–77. https://doi.org/10.1016/B978-0-12-802855-1.00003-4

[28] Object Management Group. 2014. *Object Constraint Language (OCL) Specification. Version 2.4*. OMG Document formal/2014-02-03.

[29] Victor Ortiz, Loli Burgueño, Antonio Vallecillo, and Martin Gogolla. 2019. Native support for UML and OCL primitive datatypes enriched with uncertainty in USE. In *Proceedings of the International Workshop in OCL and Textual Modeling (OCL 2019) co-located with IEEE/ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS 2019), CEUR Workshop Proceedings, Vol. 2513*. 59–66.

[30] Matteo Principe, Andrea Piccione, Alessandro Pellegrini, and Francesco Quaglia. 2020. Approximated rollbacks. In *Proceedings of the ACM SIGSIM-PADS. International Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'20)*. ACM, 23–33. https://doi.org/10.1145/3384441.3395984

[31] Francesco Quaglia and Roberto Beraldi. 2004. Space uncertain simulation events: Some concepts and an application to optimistic synchronization. In *Proceedings of the International Conference on Principles of Advanced Discrete Simulation (PADS'04)*. IEEE Computer Society, 181–188. https://doi.org/10.1109/PADS.2004.1301299

[32] Jürgen Roßmann, Eric Guiffo Kaigom, Linus Atorf, Malte Rast, Georgij Grinshpun, and Christian Schlette. 2014. Mental models for intelligent systems: eRobotics enables new approaches to simulation-based AI. *Künstl. Intell.* 28, 2 (2014), 101–110. https://doi.org/10.1007/s13218-014-0298-z

[33] Stuart J. Russell and Peter Norvig. 2010. *Artificial Intelligence. A Modern Approach* (3rd ed.). Prentice Hall.

[34] Hesham Saadawi and Gabriel A. Wainer. 2010. Rational time-advance DEVS (RTA-DEVS). In *Proceedings of the Spring Simulation Multi-conference (SpringSim'10)*. SCS/ACM, 143:1–143:8. https://doi.org/10.1145/1878537.1878686

[35] Bradley R. Schmerl, Javier Cámara, Jeffrey Gennari, David Garlan, Paulo Casanova, Gabriel A. Moreno, Thomas J. Glazier, and Jeffrey M. Barnes. 2014. Architecture-based self-protection: Composing and reasoning about denial-of-service mitigations. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security (HotSoS'14)*. ACM, 2:1–2:12. https://doi.org/10.1145/2600176.2600181

[36] Glenn Shafer. 1976. *A Mathematical Theory of Evidence*. Princeton University Press.

[37] Pankaj Swarnkar, Shailendra Kumar Jain, and R. K. Nema. 2014. Adaptive control schemes for improving the control system dynamics: A review. *IETE Techn. Rev.* 31, 1 (2014), 17–33. https://doi.org/10.1080/02564602.2014.890838

[38] Javier Troya, Nathalie Moreno, Manuel F. Bertoa, and Antonio Vallecillo. 2021. Uncertainty representation in software models: A survey. *Softw. Syst. Model.* 20, 4 (2021), 1183–1213. https://doi.org/10.1007/s10270-020-00842-1

[39] Heinz Unbehauen. 2000. Adaptive dual control systems: A survey. In *Proceedings of the IEEE Adaptive Systems for Signal Processing, Communications, and Control Symposium (AS-SPCC'00)*. IEEE, 171–180. https://doi.org/10.1109/ASSPCC.2000.882466

[40] Damián Vicino, Gabriel A. Wainer, and Olivier Dalle. 2022. Uncertainty on discrete-event system simulation. *ACM Trans. Model. Comput. Simul.* 32, 1 (2022), 2:1–2:27. https://doi.org/10.1145/3466169

[41] Wikipedia. List of uncertainty propagation software. https://en.wikipedia.org/wiki/List_of_uncertainty_propagation_software.

[42] B. Wittenmark. 1975. Stochastic adaptive control methods: A survey. *Int. J. Contr.* 21, 5 (1975), 705–730. https://doi.org/10.1080/00207177508922026

[43] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. 2018. *Theory of Modeling and Design: Discrete Event and Iterative System Computational Foundations* (3rd ed.). Academic Press.

[44] Man Zhang, Bran Selic, Shaukat Ali, Tao Yue, Oscar Okariz, and Roland Norgren. 2016. *Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model*. In *Proceedings of the 12th European Conference on Modelling Foundations and Applications (ECMFA'16)*, Vol. 9764. Springer, 247–264. https://doi.org/10.1007/978-3-319-42061-5_16

[45] Helin Zhu, Tianyi Liu, and Enlu Zhou. 2020. Risk quantification in stochastic simulation under input uncertainty. *ACM Trans. Model. Comput. Simul.* 30, 1 (February 2020), 1:1–1:24. https://doi.org/10.1145/3329117

[46] Hans-Jürgen Zimmermann. 2001. *Fuzzy Set Theory—And Its Applications*. Springer Science+Business Media.

Received 22 July 2022; revised 7 February 2023; accepted 23 March 2023

Learning to Simulate Sequentially Generated Data via Neural Networks and Wasserstein Training

TINGYU ZHU and HAOYU LIU, Peking University, China
ZEYU ZHENG, University of California, Berkeley, USA

9

We propose a new framework of a neural network-assisted sequential structured simulator to model, estimate, and simulate a wide class of sequentially generated data. Neural networks are integrated into the sequentially structured simulators in order to capture potential nonlinear and complicated sequential structures. Given representative real data, the neural network parameters in the simulator are estimated and calibrated through a Wasserstein training process, without restrictive distributional assumptions. The target of Wasserstein training is to enforce the joint distribution of the simulated data to match the joint distribution of the real data in terms of Wasserstein distance. Moreover, the neural network-assisted sequential structured simulator can flexibly incorporate various kinds of elementary randomness and generate distributions with certain properties such as heavy-tail, without the need to redesign the estimation and training procedures. Further, regarding statistical properties, we provide results on consistency and convergence rate for the estimation procedure of the proposed simulator, which are the first set of results that allow the training data samples to be correlated. We then present numerical experiments with synthetic and real data sets to illustrate the performance of the proposed simulator and estimation procedure.

CCS Concepts: • **Computing methodologies** → *Modeling and simulation;*

Additional Key Words and Phrases: Sequential simulator, neural network, Wasserstein training, statistical properties

ACM Reference format:
Tingyu Zhu, Haoyu Liu, and Zeyu Zheng. 2023. Learning to Simulate Sequentially Generated Data via Neural Networks and Wasserstein Training. *ACM Trans. Model. Comput. Simul.* 33, 3, Article 9 (August 2023), 34 pages.
https://doi.org/10.1145/3583070

1 INTRODUCTION

In many applications, such as finance, transportation, and service systems, stochastic simulation models (simulators) that have a sequential structure are widely used to create sample paths and capture the dynamics of relevant multi-dimensional random objects. A sequential-structured simulator typically involves multiple discrete time periods at a certain resolution and models a stochastic process with multi-dimensional state variables. In each time period, the simulator takes the state from the previous time period as input, and generates, along with some new randomness, a new state passing on the next time period. Such simulators are used to simulate *sequentially*

Authors' addresses: T. Zhu and H. Liu, Peking University, Yiheyuan Rd. 5, Haidian, Beijing, China, 100871; emails: {1800017813, 1800015905}@pku.edu.cn; Z. Zheng, University of California, Berkeley, 4125 Etcheverry Hall, Berkeley, CA, USA; email: zyzheng@berkeley.edu.

1049-3301/2023/08-ART9 $15.00
https://doi.org/10.1145/3583070

generated data, which is slightly more general than time series models. The data is in turn used to evaluate expectations/quantiles of functions of the sample paths or to support decision making tasks. For example, in financial applications, a simulator may be used to simulate sequential data that represents the dynamics of prices and volatilities for multiple correlated assets, possibly as well as other relevant factors that impact the asset prices. The simulated data can then be used to evaluate the risks and performances of a portfolio that are composed of these assets.

For some applications, real data are available to calibrate such sequential-structured simulators. Specifically, representative real data may record the dynamics of some, but maybe not all, dimensions of the stochastic process modeled by the sequential-structured simulator. With real data in hand, there is a natural need to tailor the simulator such that the sequentially generated data from the simulator "matches" real data in the corresponding dimensions. Such tasks have always been challenging in terms of both statistical properties and computational demands, due to large data dimension and/or partial observations. In order to calibrate a sequential-structured simulation model, existing methods largely rely on parametric models with specific distributional assumption, such that maximum likelihood method can be used to estimate the parameters using real data. In general, these methods bring up difficult-to-solve estimation procedures especially when real data only contains partial observations in a subset of dimensions. Moreover, specific distributional assumptions are often needed to compute the likelihood function. In addition to the risk of mis-specification, different distributional assumptions may require completely different optimization, computation, and statistical analysis.

To address such challenges, we propose a new framework of sequential simulation assisted by generative adversarial neural networks and Wasserstein training. At each time step, the neural networks take the state vector of the previous time as input, and generate the next state with some additional randomness known as *elementary randomness*. The elementary randomness is prespecified by users according to domain knowledge, whereas parameters of the neural networks, which aim to capture the potential non-linear and complicated dependence of the dynamics on the previous state, are estimated from data. Such estimation is carried out through a Wasserstein training process, which aims to match the (possibly high-dimensional) joint distribution of the simulated data with that of the real data. More specifically, batches of simulated data and real data are passed to another neural network known as the discriminator, which then produces a loss function indicating the similarity of the underlying distributions of both data. The loss function is then alternately maximized via updating the network parameters of the discriminator and minimized via updating the parameters of the simulator networks. We note that this estimation procedure does not require computing likelihood functions, and does not change significantly if the dimensions increase or the type of elementary randomness changes. With such flexibility, the sequential-structured simulator fitted to real data can provide us with information regarding "what-if" scenarios. By altering some parts of the simulator, such as the elementary randomness and length of the sequences, we can answer questions such as "what happens if the variances increase by 10%" and "what happens if the length of the sequences increases by 10%". Further, such modeling scheme allows us to discuss its statistical aspect, which involves three research questions: What types of underlying sequential-structured simulation model can be consistently learned by such framework? What is the statistical rate of convergence if consistency is achieved? To what extent is correlation allowed to exist between different sequences in the sample set, and how much impact does it have on the convergence rate? To the best of our knowledge, such theoretical guarantees do not exist for general sequential-structured simulators assisted by neural networks, especially when the associated distributions have unbounded support.

The modeling and simulation of sequentially generated data has been introduced and intensively studied in the literature. An important class of models is based on parametric assumptions

to capture the dependence structure in the sequences, of which one most representative example is the **stochastic volatility model (SVM)**. As [29] points out, a key intuition in the SVM literature is that the variations in the level of activity is directed by an underlying stochastic process. As an early example of discrete-time stochastic volatility modeling, [32] models the risky part of returns as a product process, integrating an underlying indicator of volatility which follows a non-zero mean Gaussian linear process. Later, continuous-time SVMs formulated as diffusion processes, such as the Heston model presented by [21], become more favorable in portfolio choice and derivatives pricing. The multivariate generalizations of SVMs are presented by [4]. Estimating such models poses substantial challenge due to difficulties in evaluating the exact likelihood function. It is concluded in [8] that there are mainly three categories of estimation techniques to address the challenge, namely estimators based on the method of moments, such as [25]; estimators based on the maximum likelihood principle, such as [28] and estimators based on an auxiliary model, such as [6]. A most representative application of auxiliary models is model calibration, which uses current information such as the option price in parameter estimation, see [1] for example. However, these techniques can become intractable or computationally demanding when the dimension becomes even moderately high. Moreover, considerable assumptions are required, rendering these techniques vulnerable to model mis-specifications.

An alternative way of modeling sequential data and estimating such models is to use the neural network framework. A representative class of models is the **recurrent neural network (RNN)**, which, along with other variants such as **gated recurrent unit (GRU)** and **long-short-term memory (LSTM)**, aims to capture the transition and dependencies between the state vectors in the time-series. Recently, **variational autoencoder (VAE)** provided by [23] is combined with RNN to model and estimate sequentially generated data with a latent stochastic process, see [9, 17, 24, 35] for examples. We also note that [9] and [35] are among the first that integrate these frameworks with Monte Carlo simulation. Such frameworks in general adopt a likelihood-based statistical inference method, using VAE to learn the posterior distributions of latent process variables whose prior distribution and randomness-generation distribution need to be pre-specified. Another branch of neural network-based sequential modeling, including our work, is based on the **generative adversarial network (GAN)** [18]. A GAN is a generative model consisting of a generator and a discriminator, both of which are often neural networks. The generator produces data of some distribution, and the discriminator compares such distributions with the empirical distribution of the sample set. By alternately maximizing the loss function via updating the discriminator and minimizing the loss function via updating the generator, people train the generator network to produce data with the same underlying distribution as the sample set. What is noteworthy about GAN is that it inherently provides a way to compare (possibly high-dimensional) distributions via capturing the most characteristic features, instead of conducting point-wise comparisons or comparing less informative statistics. This serves as an important foundation for application of GANs in stochastic process modeling, which requires learning distributions from data. Representative works such as [16, 31, 36] and [37] design network architectures and loss functions for various specific purposes, such as incorporating extra information as conditions [16], and capturing long-term dependence [36, 37]. We refer to [7] for an overview of application of GANs in time-series modeling, and [14] for an overview of such application in specifically financial time-series data. As stated in both overviews, instability of training, especially when the sample size is limited and severe randomness is included, and lack of proper measurement of how well the distributions are generated are the main problems suffered by applications of GAN in time-series modeling. Answering to such concerns, we use a more model-based approach instead of completely relying on the neural network architectures to capture the distribution from representative data. Also, we use the **Wasserstein generative adversarial network (WGAN)** training framework with gradient

penalty [19] in the estimation procedure to improve training stability as well as provide a basis for our proof of statistical convergence. Apart from that, we propose our own illustrative measurements in the numerical experiments, which either directly use statistics of distributions or resort to a downstream task to provide assessment from an operational aspect. In terms of the sequential structure, [34] consider a different task from ours, inventory optimization, and proposed an RNN-inspired simulation approach to improve computational capabilities for large-scale inventory management.

The Wasserstein training of our neural network-based simulator is inspired by GAN [18], WGAN [2], and the doubly stochastic WGAN framework by [38]. Representative theoretical works like [5, 11] and [12] serve as fundamentals of the proof of our theorem. We also refer to [13] and [20] for descriptions and efficient estimators for general distribution distance metrics such as Wasserstein distance and Kullback-Liebler divergence. We adopt the use of Wasserstein distance to measure distribution distance in this work. We additionally remark that, in financial applications, the drift term and volatility term are separately and explicitly constructed in our framework, which indicates that the induced stochastic process allows a transition to its risk-neutral distribution. The generated risk-neutral paths can be used to estimate the option prices of underlying assets. Therefore, option data can be incorporated, for instance, by adding the mean square error of estimated option prices into the loss function, to jointly learn the real and risk-neutral dynamics. Moreover, options data can be used to fine-tune the parameters in order to assist the training process on price sequences.

The rest of this paper is organized as follows. Section 2 discusses the model setup of the simulator. Section 3 discusses the estimation framework. Section 3.3 discusses the statistical theory for the estimation method. Section 4 provides numerical experiments.

2 MODEL SETUP

We consider a class of simulators that are used to simulate sequential data. A simulator consists of two functions $\mu(\cdot,\cdot)$ and $\Sigma(\cdot,\cdot)$, which take current information as input to generate information about the next step, and incorporate a sequence of elementary randomness, denoted as $\{\eta_k\}$, which contributes to all the randomness in the simulation process. Such simulators generate the dynamics of a stochastic process $(X_k : k = 0, 1, 2, \ldots)$ that takes value in a d-dimensional multi-dimensional real space. That is, $X_k \in \mathbb{R}^d$ for any k. The simulator sequentially updates $(X_k : k = 0, 1, 2, \ldots)$ according to

$$X_{k+1} = \mu(l_k, X_k) + \Sigma(l_k, X_k)\eta_{k+1}, \quad k = 0, 1, 2, \ldots, \tag{1}$$

in which l_k is a real-valued deterministic label that can be used to represent the time-of-day effect or seasonality associated with time period k. The notion $\mu(\cdot,\cdot)$ is a d-dimensional function of the label and the state of the stochastic process in the previous time period. Similarly, $\Sigma(\cdot,\cdot)$ has the same input variables as $\mu(\cdot,\cdot)$, and outputs a $d \times d'$ matrix. The expressions of $\mu(\cdot,\cdot)$ and $\Sigma(\cdot,\cdot)$ can be further specified to incorporate background knowledge. The notion η_{k+1} is referred to as *elementary randomness*, which represents a d'-dimensional mean-zero random vector that is used by the simulator in the time period $k+1$, and $\eta_k : k = 1, 2, \ldots$ are assumed to be independent and identically distributed.

We consider practical applications in which the probability distribution of the elementary randomness η_k's are specified by the users according to background domain knowledge, whereas both $\mu(\cdot,\cdot)$ and $\Sigma(\cdot,\cdot)$ are unknown functions that need to be estimated from empirical data. For many applications, not all dimensions of X_k and not all time periods of data can be observed. Therefore, we consider a flexible data framework for which only the first $d_1 \leq d$ dimension of X_k can be observed at selected time periods. Specifically, we write X_k as $(S_k, Y_k)^\top$, where S_k denotes the d_1-

dimensional observed process, and Y_k denotes the $(d - d_1)$-dimensional latent process that is not observed in empirical data. In terms of generality, suppose that the sequence of S_k's can only be observed at p selected time periods labeled as $0 \le k_1 < k_2 < \cdots < k_p$. Set $\boldsymbol{S} = (S_{k_i} : i = 1, 2, \ldots, p)$. We presume that the empirical data is composed of n copies of $\boldsymbol{S}$, denoted as

$$\boldsymbol{S}_1, \boldsymbol{S}_2, \ldots, \boldsymbol{S}_n,$$

which are n identically distributed copies of $\boldsymbol{S}$. Note that both the number of unobserved points between k_i and k_{i+1} $(i = 1, 2, \ldots, p-1)$ and the dimension $d - d_1$ can be either known or specified by the user as part of the modeling assumptions.

The copies of $\boldsymbol{S}$ do not need to be mutually independent in practice. Our goal is to provide a statistical and computational framework to train (or equivalently, to estimate) the simulator given by (1) such that the sequentially generated data $(X_k : k = 0, 1, 2, \ldots)$ matches the joint distribution of $\boldsymbol{S}$ on the corresponding dimensions and time periods.

Before discussing the statistical and computational framework, we briefly describe two examples to demonstrate the relevance of the class of simulators of interest, given by the form of (1). The first example is given by the **multivariate stochastic volatility model (MSVM)** formulated by a stochastic differential equation, which is widely used within the fields of financial economics and mathematical finance to capture the dynamics of asset prices. Specifically, the vector of state variables X_t follows a multivariate diffusion process,

$$dX_t = \mu_c(X_t)dt + \Sigma_c(X_t)dW_t, \quad t \in [0, T], \tag{2}$$

where $X_t = (S_t, Y_t)^\top$, with S_t denoting the observed price process and Y_t denoting the latent volatility process, and $(W_t : t \in [0, T])$ is a canonical multi-dimensional Brownian motion. Most practical simulation tools for the multi-dimensional diffusion model use the idea of discretization, at a user-specified discretization resolution. Using the Euler-Maruyama discretization scheme [27] for example, once a discretization resolution Δt is selected, the simulation process fits into the general simulator considered in (1). Specifically, we have

$$\begin{aligned} &\mu(l_k, X_k) = X_k + \mu_c(X_k)\Delta t, \\ &\Sigma(l_k, X_k) = \Sigma_c(X_k)\sqrt{\Delta t}, \\ &\eta_{k+1} \sim \mathcal{N}(0, I_{d'}). \end{aligned} \tag{3}$$

Namely, $(X_k : k = 0, 1, 2, \ldots)$ is sequentially generated according to

$$X_{k+1} = X_k + \mu_c(X_k)\Delta t + \Sigma_c(X_k)\sqrt{\Delta t}\eta_{k+1}, \tag{4}$$

where $\eta_k, k = 1, 2, \ldots$ are independent standard d'-dimensional multi-variate normal random variables. We use the subscript c for μ_c and Σ_c to indicate that the label l_k is a constant. We additionally remark that, when the empirical data process follows a stationary pattern, we can always set l_k as a constant, which is often the case in practical applications. Therefore, in the rest of this work we mostly consider stationary cases where $l_k = c$, and $\mu(\cdot, \cdot)$ and $\Sigma(\cdot, \cdot)$ are viewed as functions of X_k only. Besides that the simulator considered in (1) covers the simulation process of MSVM, our data framework also accommodates a practical possibility that the resolution at which data is observed can be lower than the resolution at which the simulation of the stochastic process is conducted.

Not only does the simulator defined by (1) allow simulation of the MSVM, but our data framework also accommodates sequential data with heavy-tailed distributions. As the second example, this simulator sequentially updates $(X_k, k = 0, 1, 2, \ldots)$ according to

$$X_{k+1} = X_k + \mu_h(X_k)\Delta t + \Sigma_h(X_k)\Delta\eta_{k+1}, \tag{5}$$

where Δt can be any given resolution, and $\Delta\eta_k, k = 1, 2, \ldots$ are i.i.d. variables with some given heavy-tailed distribution, such as the t distribution or Pareto distribution. This simulator can be used for data modeling within the fields of spectroscopy, particle motion, finance, and so on, where heavy-tailed behaviors are frequently observed. As a more specific case, we take

$$\mu_h(X_k) = b(X_k, \alpha), \quad \Sigma_h(X_k) = 1, \quad \Delta\eta_{k+1} = L_{k+1}^{\alpha}\Delta t^{1/\alpha}, \quad \alpha \in (0, 2) \tag{6}$$

where the definition of $b(\cdot)$ is specified in [30], and $L_k^{\alpha} : k = 1, 2, \ldots$ is a sequence of i.i.d. standard symmetric α-stable random variables which have heavy tails when $\alpha \in (0, 2)$. This is the discretized version of a stochastic differential equation driven by a symmetric α-stable Lévy process.

3 METHOD

In this section, we use a new framework to estimate the simulator so as to match its simulated data with real data. More specifically, we use **neural networks (NN)** to approximate $\mu(\cdot)$ and $\Sigma(\cdot)$ of the simulator, and update the NN parameters to minimize the distance between the joint distribution of simulated data and the joint distribution of real data. To achieve this, we need to specify how the output distribution is generated by the NN-based simulator, how the distance between the two distributions is formulated and computed, and how the NN parameters are updated according to the computed distance. In the following part of this section, Sections 3.1 and 3.2 provide answers to the first two questions, and formulate the estimation problem into a minimax optimization problem. Section 3.3 answers the third question by discussing the training process to solve the optimization problem.

3.1 Neural Network-integrated Simulator

Recall that $\boldsymbol{S} = (S_{k_1}, S_{k_2}, \ldots, S_{k_p})$ represents the observed sequence. Let π denote the true joint probability distribution of $\boldsymbol{S}$. The training data are n identically distributed copies of $\boldsymbol{S}$, given by $\boldsymbol{S}_1, \boldsymbol{S}_2, \ldots, \boldsymbol{S}_n$. These sequences can be either independent or weakly correlated. Let $\tilde{\pi}$ denote the empirical distribution of the data.

The neural network-based simulator generates a sequence of state vectors $(X_k : k = 1, 2, \ldots)$ according to

$$X_{k+1} = \mu_\theta(X_k) + \Sigma_\phi(X_k)\eta_{k+1}, \tag{7}$$

where $\eta_k, k = 1, 2, \ldots$ are given d'-dimensional random vectors, $\mu_\theta(\cdot)$ is a d-dimensional function of X_k, and $\Sigma_\phi(\cdot)$ is a $d \times d'$-dimensional function of X_k which outputs a $d \times d'$ matrix. Both functions adopt the NN architecture, parameterized by NN parameters θ and ϕ, and are approximations of $\mu(\cdot)$ and $\Sigma(\cdot)$ of the simulator. Specifically, given the number of layers $L \in \mathbb{Z}^+$ and the width of the l-th layer n_l, $l = 1, 2, \ldots, L$, for an input variable $X \in \mathbb{R}^d$, the functional forms of $\mu(X; \theta = (\boldsymbol{W}_\theta, \boldsymbol{b}_\theta))$ and $\Sigma(X; \phi = (\boldsymbol{W}_\phi, \boldsymbol{b}_\phi))$ (expanded forms of μ_θ and Σ_ϕ) are given as

$$\begin{aligned} &\mu_0 = X;\ \mu_k = \sigma(W_{\theta,k} \cdot \mu_{k-1} + b_{\theta,k-1}),\ k = 1, 2, \ldots, L-1; \\ &\mu(X; \theta = (\boldsymbol{W}_\theta, \boldsymbol{b}_\theta)) = W_{\theta,L} \cdot \mu_{L-1} + b_{\theta,L}, \end{aligned} \tag{8}$$

and

$$\begin{aligned} &\Sigma_0 = X;\ \Sigma_k = \sigma(W_{\phi,k} \cdot \Sigma_{k-1} + b_{\phi,k-1}),\ k = 1, 2, \ldots, L-1; \\ &\Sigma(X; \phi = (\boldsymbol{W}_\phi, \boldsymbol{b}_\phi)) = W_{\phi,L} \cdot \Sigma_{L-1} + b_{\phi,L}, \end{aligned} \tag{9}$$

where $(\boldsymbol{W}_\theta, \boldsymbol{b}_\theta)$ and $(\boldsymbol{W}_\phi, \boldsymbol{b}_\phi)$ represent all the parameters in the neural networks, which is the aggregation of $W_{\theta,k}$, $b_{\theta,k}$, $W_{\phi,k}$ and $b_{\phi,k}$ $(k = 1, 2, \ldots, L)$, the parameters of each neural network layer. The notation "$\cdot$" represents a dot product. The dimensionality of $(\boldsymbol{W}_\theta, \boldsymbol{b}_\theta)$ and $(\boldsymbol{W}_\phi, \boldsymbol{b}_\phi)$ can be flexibly adjusted for better simulation outcomes, as long as the dimensionalities of the input and

output of $\mu(\cdot;\theta)$ and the input of $\Sigma(\cdot;\phi)$ match that of X_k, and the output of $\Sigma(\cdot;\phi)$ can be reshaped into a matrix to multiply with the elementary random variables η_{k+1}. To train the neural networks to produce results that fit observations is to search for optimal choices of such parameters. The operator $\sigma(\cdot)$ takes a vector of any dimension as input and is a component-wise operator. We specify the operator $\sigma(\cdot)$ as a *rectified linear activation unit*, or *ReLU* for short. Specifically, for $Z \in \mathbb{R}^d$, we have

$$\sigma(Z) = (\max(Z_1, 0), \max(Z_2, 0), \ldots, \max(Z_d, 0)) . \tag{10}$$

Let X_0 be a given constant or a random vector with a given probability distribution π_0. Recall that $X_k = (S_k, Y_k)$, where S_k is the d_1-dimensional observed process, and Y_k is the $(d-d_1)$-dimensional latent process. We additionally remark that even though $\mu(\cdot)$ and $\Sigma(\cdot)$ of the underlying true model are assumed to be stationary, it is still necessary to generate a full sequence instead of modeling a single step of transition. This is due to our assumption of an existing latent process $(Y_k : k = 1, 2, \ldots)$, which is intractable and has to be sequentially simulated. Finally, the joint probability distribution of the generated d-dimensional observed sequence at the measured data points $\hat{S} = (\hat{S}_{k_1}, \hat{S}_{k_2}, \ldots, \hat{S}_{k_p})$, denoted as $\hat{\pi}$, is taken as the output of the generator. Note that $\hat{\pi}$ and $\hat{S}$ are also functions of θ and ϕ, and are therefore sometimes denoted as $\hat{\pi}(\theta, \phi)$ and $\hat{S}(\theta, \phi)$.

3.2 Wasserstein Distance and Discriminator

Next, we introduce the Wasserstein distance, which is used to quantify the distance between two given distributions. The Wasserstein distance of the generated distribution $\hat{\pi}$ and the real distribution π is given by

$$W(\hat{\pi}, \pi) = \inf_{\gamma \in \Pi(\hat{\pi}, \pi)} \mathbb{E}_{(\hat{S}, S) \sim \gamma}[\|\hat{S} - S\|_2], \tag{11}$$

where $\Pi(\hat{\pi}, \pi)$ denotes the set of all joint distributions of which the marginals are respectively $\hat{\pi}$ and π, and $\|\cdot\|$ denotes the L_2 norm. Since the Wasserstein distance in high dimensions does not have a closed form for computation, we often use the Kantorovich-Rubinstein duality given by

$$W(\hat{\pi}, \pi) = \sup_{\|f\|_L \le 1} \mathbb{E}_{\hat{S} \sim \hat{\pi}}[f(\hat{S})] - \mathbb{E}_{S \sim \pi}[f(S)], \tag{12}$$

where $\|f\|_L \le 1$ denotes the class of all 1-Lipschitz functions f, i.e., $|f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)| \le \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2$ for any $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^{d_\pi}$. Computation of the supremum over all 1-Lipschitz functions is also analytically intractable, but we can use a neural network f_ψ to approximate f, and search over all such approximations parameterized by NN parameters ψ. With the same network architecture as the simulator networks, f_ψ has functional form given as

$$\begin{aligned} &f_0 = X;\ f_k = \sigma(W_{\psi,k} \cdot f_{k-1} + b_{\psi,k-1}),\ k = 1, 2, \ldots, L-1; \\ &f(X; \psi = (\boldsymbol{W}_\psi, \boldsymbol{b}_\psi)) = W_{\psi,L} \cdot f_{L-1} + b_{\psi,L}. \end{aligned} \tag{13}$$

In the framework of the classical WGAN, a function with the same purpose as f_ψ is known as the *discriminator*. Our method aims to match the generated distribution to the real distribution, which can be achieved through minimizing the Wasserstein distance of the two distributions. We formulate the estimation method as solving the following minimax optimization problem:

$$\min_{\theta \in \Theta, \phi \in \Phi} \max_{\psi \in \Psi} \mathbb{E}_{\hat{S} \sim \hat{\pi}(\theta, \phi)}[f_\psi(\hat{S})] - \mathbb{E}_{S \sim \pi}[f_\psi(S)]. \tag{14}$$

The empirical version of problem (14) is given by

$$\min_{\theta \in \Theta, \phi \in \Phi} \max_{\psi \in \Psi} \frac{1}{n} \sum_{j=1}^{n} f_\psi(\hat{S}_j(\theta, \phi)) - \frac{1}{n} \sum_{j=1}^{n} f_\psi(S_j). \tag{15}$$

3.3 Training

In this section, we discuss the training process for model estimation optimization problem (15). We adopt a classical training strategy to solve the minimax problem, which is to alternately update the parameters of the NN-based discriminator and the simulator. Updating the discriminator increases the difference between the two summation terms of (15), which is then attenuated by updating parameters of the simulator. During this process, the discriminator converges to the supreme f over the class of candidate functions, while the output distribution of the simulator converges to the empirical distribution.

We apply the gradient descent method for the training process, which is based on computing gradients of the objective functions to the model parameters. The gradient $\nabla_{\theta,\phi} f_\psi(\hat{\boldsymbol{S}}(\theta,\phi))$ is evaluated through backpropagation using the chain rule, which involves differentiating the entire process of simulation. The diagram for such computation is illustrated in Appendix A.

sectionStatistical Properties In this section, we discuss the statistical property of the estimation method. We prove that the framework proposed in Sections 3.1 and 3.2 can effectively learn distributions of a wide class of sequential data, if the neural network architectures are properly chosen, and the number of copies of $\boldsymbol{S}$, denoted as n, is large enough. In the following Section 3.4, we formulate the statistical convergence problem and describe the requirements and assumptions.

3.4 Formulation of Problem

In this section, we propose three basic requirements on real data, data preparation, and neural network functional class, and explain the reasons for them. Such explanations also shed light on the main ideas of our proof.

3.4.1 Underlying Distribution of Real Data. Let $\boldsymbol{X}_j = (\boldsymbol{S}_j, \boldsymbol{Y}_j)$ denote the sequence. In this section, we prove that the solution of the optimization problem (15) can generate a distribution $\hat{\pi}_S$ of the observed dimensions $\boldsymbol{S}$ that converges to the underlying real distribution π_S of the observed dimensions. Without loss of generality, we assume in this section that all dimensions and time points of the real data are observed, i.e., we have $\boldsymbol{X} = \boldsymbol{S}$. We make this assumption because the convergence of distribution does not include the unobserved dimensions $\boldsymbol{Y}$. Besides, given the value of $\boldsymbol{S}$, the choice of $\boldsymbol{Y}$ has no effect on the optimality of $\boldsymbol{S}$ as a solution of the optimization problem (15). Suppose that the real data, $\boldsymbol{X}_j = (X_{j,0}, X_{j,1}, \ldots, X_{j,p})$, $j = 1, 2, \ldots, n$, is generated as follows: the sequence starts from an initial distribution $X_0 \sim \pi_0$, where $X_0 \in \mathbb{R}^d$, and sequentially proceeds according to

$$X_{i+1} - X_i = \mu(X_i) + \Sigma(X_i)\xi_i. \quad x_i \in \mathbb{R}^d, i = 1, 2, \ldots, p, \tag{16}$$

where ξ_i is some given elementary randomness. The initial distribution π_0 and the integrated functions (μ, Σ) jointly determine the underlying distribution of the real data, denoted as π. To ensure statistical convergence, we assume that π_0 is a known distribution provided to the simulator, and $(\xi_i : i = 1, 2, \ldots, p)$ have the same distribution as the elementary randomness $(\eta_k : k = 1, 2, \ldots, p)$ of the simulator.

Additionally, we can assume different sequences to be independent and identically distributed, or weakly dependent. It is noteworthy that weak correlation allows for applicational situations where all sequences in the sample set are segmented from one single sequence of time-series data. To characterize weak correlation across sequences, we let the first element X_0 of two arbitrary sequences be correlated, namely, $cov(X_0^{(j)}, X_0^{(l)}) \neq 0$ for some $j, l \in \{1, 2, \ldots, n\}$, where n is the sample size. The specific assumption of constraint on correlation will be formulated in the main theorem, where statistical convergence results for both *i.i.d.* and weakly dependent data will also be presented.

3.4.2 Truncation. Classical theoretical results of neural network approximation require the input of the neural network to have bounded support, while in our framework the sequence $(\eta_k : k = 1, 2, \ldots)$ is often set to follow the Gaussian distribution or some heavy-tailed distribution, resulting in unboundedness of the sequence $(X_k : k = 1, 2, \ldots)$. To address the challenge due to unboundedness, we perform truncation methods on both the empirical data and the simulated data. Specifically, we transform the empirical data into its bounded version $\boldsymbol{X}_j^B$ according to

$$X_{j,i}^B = \begin{cases} X_{j,i}, \text{ if } |X_{j,i}| \le B_1; \\ B_1, \text{ else.} \end{cases} \quad i = 0, 1, 2, \ldots p, \quad j = 1, 2, \ldots, n,$$

and the bounded version of simulated data is simulated with bounded elementary randomness given as

$$\eta_k^B = \begin{cases} \eta_k, \text{ if } |\eta_k| \le B_2; \\ B_2, \text{ else.} \end{cases} \quad k = 1, 2, \ldots$$

We note that the real data and simulated data are truncated in different ways. We perform truncation on elementary randomness, instead of truncating after simulating a whole unbounded sequence, because the approximability of μ_θ and Σ_ϕ can only be ensured within bounded areas. We lose control if an unbounded X_i is generated and fed to the networks during the sequential simulation process. However, to ensure that the simulated sequence is also bounded by B_1, we can find some constant C such that $B_2 = C \cdot B_1$, as both truncation bounds B_1 and B_2 increase to infinity. Therefore, for simplicity of notation, we use a common truncation bound B on the truncated data.

We denote the bounded versions of the empirical data and the simulated data as $\boldsymbol{X}_j^B$ and $\hat{\boldsymbol{X}}_j^B$, and their distributions as π^B and $\hat{\pi}^B$. The empirical optimization problem (15) is then transformed into the bounded version:

$$\min_{\theta\in\Theta, \phi\in\Phi} \max_{\psi\in\Psi} \frac{1}{n}\sum_{j=1}^{n} f_\psi(\hat{\boldsymbol{X}}_j^B(\theta,\phi)) - \frac{1}{n}\sum_{j=1}^{n} f_\psi(\boldsymbol{X}_j^B). \tag{17}$$

The Wasserstein distance between the bounded distributions, denoted as $W(\pi^B, \hat{\pi}^B)$, can be controlled. As the size of NN goes to infinity, the truncation bounds also increase to infinity, which, with certain restrictions on η_k that will be presented in the following specific assumption, results in the convergence of the bounded distribution towards its unbounded version, i.e., $\lim_{B\to\infty} W(\pi, \pi^B) = 0$. This convergence enables the controlling of $W(\pi, \pi^B)$, and thus $W(\pi, \hat{\pi}^B)$.

3.4.3 Restrictions on the Discriminator Class. Since taking the supremum over a whole function class, e.g., the bounded 1-Lipschitz class, is computationally intractable, we replace it with a function class of neural networks with bounded size. In this case, the discriminator is a neural network with parameters to be optimized, and defines a metric $d_{\mathcal{F}}(\cdot,\cdot)$ of distributions. A proper discriminator should define a metric under which convergence is equivalent to Wasserstein convergence. This equivalence guarantees that the discriminator effectively distinguishes different distributions, while also making sure that if the training fails to find a solution with small distance under $d_{\mathcal{F}}(\cdot,\cdot)$, then indeed the distributions are far away under Wasserstein distance. Therefore, we impose certain restrictions on the size and components of the discriminator class, so that its discriminative power is proportional to that of the bounded 1-Lipschitz class. Further, such restrictions are helpful when controlling the error induced by weak correlation.

Specifically, restrictions on the discriminator class are imposed through the following definitions.

Definition 3.1 (function class $\mathcal{F}_\Psi$).

$$\mathcal{F}_\Psi(\kappa, L, P, K) = \Bigg\{ f : \mathbb{R}^d \to \mathbb{R} \,\Big|\, f \text{ in the form of ReLU neural networks with } L \text{ layers and width bounded by } P, \|W_i\|_\infty \le \kappa, \|b_i\|_\infty \le \kappa, \sum_{i=1}^{L} \|W_i\|_0 + \|b_i\|_0 \le K \Bigg\}.$$

where W_i and b_i, $i = 1, 2, \ldots, L$ are the weight matrices and bias vectors of the layers.

Definition 3.2 (restricted function class $\mathcal{F}_\Psi$).

$$\mathcal{F}_\Psi(\kappa, L, P, K, \epsilon_f) = \Big\{ f \in \mathcal{F}_\Psi(\kappa, L, P, K) \,\Big|\, \|f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)\| \le \|\boldsymbol{x}_1 - \boldsymbol{x}_2\| + 2\epsilon_f, \forall \boldsymbol{x}_1, \boldsymbol{x}_2 \in [-B, B]^d \Big\}.$$

Throughout our proof, we assume that the discriminator class $\mathcal{F}_\Psi$ is restricted as in Definition 3.2, with parameters $\kappa, L, P, K, \epsilon_f$. In proof of the main theorem, we derive specific order of such parameters with regard to sample size n, to ensure statistical convergence. We remark that the additional $2\epsilon_f$ term in 3.2 serves to allow for approximation, in the sense of infinity norm and within a bounded area, of any 1-Lipschitz function by $\mathcal{F}_\psi(\kappa, L, p, K, \epsilon_f)$, which will be crucial in controlling certain error terms. This is because there is no guarantee of the approximation power of the strictly 1-Lipschitz neural network function class, but with theoretical guarantee that $\mathcal{F}_\psi(\kappa, L, P, K)$, for large enough parameters κ, L, P, K, can approximate any 1-Lipschitz function f within distance ϵ_f, we know that $\mathcal{F}_\psi(\kappa, L, p, K, \epsilon_f)$ is dense enough to approximate any 1-Lipschitz function within a bounded area. Conversely, the fact that $\mathcal{F}_\psi(\kappa, L, P, K, \epsilon_f)$ can be approximated by any 1-Lipschitz function f within distance $2\epsilon_f$ is also a basis for controlling certain error terms.

3.5 Specific Assumption and Theorem

We make the following assumption on the underlying true model of the sample data:

Assumption 1. *The following conditions are satisfied for the generation process of sample data (16):*

(1) *All sequences $\boldsymbol{X}_j$ are independent and identically distributed or weakly dependent. A specified characterization of weak dependence is given as follows:*
Let Π_n denote the joint distribution of $(\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n)$. Let $\{\bar{\boldsymbol{X}}_j\}_{j=1}^n$ be independent and identically distributed variables, where $\bar{\boldsymbol{X}}_j$ has the same distribution as $\boldsymbol{X}_j$. Let $\bar{\Pi}_n$ denote the joint distribution of $(\bar{\boldsymbol{X}}_1, \bar{\boldsymbol{X}}_2, \ldots, \bar{\boldsymbol{X}}_n)$, we have

$$W(\Pi_n, \bar{\Pi}_n) \le O(n^{\frac{1}{2}-\beta}), n \to \infty \tag{18}$$

for some $\beta > 0$.
(2) *$\mu : \mathbb{R}^d \to \mathbb{R}^d$ and $\Sigma : \mathbb{R}^d \to \mathbb{R}^{d \times d'}$ are Lipschitz continuous and bounded on $\mathbb{R}^d$.*
(3) *The tail order of the probability density function of every random variable in the sequence of elementary randomness $(\xi_k : k = 1, 2, \ldots)$ is no more than $x^{-(2+\alpha)}$, for some $\alpha > 0$. Namely, $p_{\xi_k}(x) \le O(x^{-(2+\alpha)}), x \to \infty$, for all $k = 1, 2, \ldots$, where $p_{\xi_k}(x)$ is the density function of ξ_k.*

The first condition is assumed in order to effectively control the statistical error, which we will elaborate on in the following subsection. The two requirements of the second condition are interpreted as follows: the Lipschitz continuous requirement for $\mu(\cdot)$ and $\Sigma(\cdot)$ ensures that they can be sufficiently approximated within a bounded area by neural networks with proper architectures. The bounded requirement for $\mu(\cdot)$ and $\Sigma(\cdot)$ restricts the output, and thus every X_k in the sequence, to a bounded area, when the sequence of elementary randomness $(\eta_k : k = 1, 2, \ldots)$ is also bounded or truncated to its bounded version. The third condition is imposed for controlling the bounding

error, which is defined as the Wasserstein distance between the real distribution and its bounded counterpart, i.e., $W(\pi, \pi^B)$.

The main result of statistical analysis is presented as follows:

THEOREM 3.3. *Suppose that n copies of i.i.d. or weakly correlated data are available and that the underlying generation model satisfies Assumption 1. Under appropriate specifications of the neural network architecture, let θ^* and ϕ^* be the parameters that solve the optimization problem given as (17), and let the truncation boundary B increase to infinity along with n, by order $B = O(n^{\frac{2}{3pd+6}}), n \to \infty$. We have*

$$\mathbb{E}\left[W(\pi, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*}))\right] \leq O\left(n^{-\frac{1}{3pd+6}}(\log n)^{\frac{3}{2}} + n^{-\frac{2\alpha}{3pd+6}} + n^{-\beta}\right), n \to \infty. \tag{19}$$

where α and β have the same meanings as in Assumption 1, p is the length of the observed sequence, d is the dimension of the observed process. We hide constant coefficients that are independent of n and B, but relevant to p, d, α, the Lipschitz constants and bounds of $\mu(\cdot)$ and $\Sigma(\cdot)$.

The implication of the three terms, $n^{-1/(3pd+6)}(\log n)^{3/2}$, $n^{-2\alpha/(3pd+6)}$, and $n^{-\beta}$, can be explained as follows:

- $n^{-1/(3pd+6)}(\log n)^{3/2}$ is the balanced statistical error. Roughly speaking, balancing is to decide on an appropriate size for the discriminator class, and two groups of parameters are taken into consideration.
  (1) The truncation bound B, which is also proportional to the domain size of the discriminator function, and thus the size of the discriminator class. It should increase fast enough to ensure the convergence rate of $W(\pi, \pi^B)$.
  (2) Other parameters that control the size of the neural network, such as width, depth, number of neurons, and maximum weight.

  The size of the discriminator class should increase at a proper rate, so that the discriminative power of the discriminator class is proportional to that of the 1-Lipschitz function class, but does not become oversize to induce an uncontrollable statistical error. We refer to Section 3.6.4 for details.
- $n^{-2\alpha/(3pd+6)}$ is the bounding error induced by truncation, which is balanced together with the statistical error term. The order of B is selected to make both error terms convergent at similar rates. The bigger α is, the smaller is the tail order of π, and thus also the bounding error.
- $n^{-\beta}$ is the statistical error induced by weak correlation among data. Note that the bigger β is, the bigger are both the weak correlation and this portion of statistical error.

These terms together demonstrate the impact of the dimensionality of data, tail order, and weak correlation on the statistical convergence rate.

Compared with existing work on GAN theory, our statistical theory discusses situations when the input of the discriminator network is sequentially generated and unbounded. Further, most theoretical work such as [3] and [5] lower bounds the discriminative power and upper bounds the generalization error with regard to the discriminator class, but does not derive the statistical convergence rate of $W(\pi, \pi_n^*)$, where π_n^* denotes the optimal solution of the empirical minimax optimization problem for GAN training, and π is the underlying real distribution.

We additionally remark that the assumptions, as well as the truncation strategies, are imposed only to guarantee statistical convergence in the following theorem, and are sometimes unnecessary in practical applications, especially when we have a moderate tolerance for approximation error. For example, to achieve the numerical results in Section 4, we did not perform truncation on the empirical data or the elementary randomness.

3.6 Analysis

In this section, we briefly describe how Theorem 3.3 is proved. We refer to the appendix for details.

3.6.1 Decomposition of Error. The main framework of proof is to control the Wasserstein distance of real distribution and learned simulated distribution using an oracle inequality, decomposing it into generator approximation error, discriminator approximation error, bounding error, and statistical error.

Adopting the idea of [12], we have

$$W(\pi, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*})) \overset{(i)}{\leq} W(\tilde{\pi}^B, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*})) + \underbrace{W(\pi^B, \tilde{\pi}^B)}_{\text{statistical error}} + \underbrace{W(\pi, \pi^B)}_{\text{bounding error}}, \tag{20}$$

$$W(\tilde{\pi}^B, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*})) = d_{\mathcal{F}_\Psi}(\tilde{\pi}^B, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*})) + \underbrace{W(\tilde{\pi}^B, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*})) - d_{\mathcal{F}_\Psi}(\tilde{\pi}^B, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*}))}_{\text{discriminator approximation error I}}, \tag{21}$$

$$d_{\mathcal{F}_\Psi}(\tilde{\pi}^B, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*})) \overset{(ii)}{\leq} d_{\mathcal{F}_\Psi}(\tilde{\pi}^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi)) \overset{(iii)}{\leq} d_{\mathcal{F}_\Psi}(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi)) + \underbrace{d_{\mathcal{F}_\Psi}(\tilde{\pi}^B, \pi^B)}_{\text{statistical error}}, \tag{22}$$

$$d_{\mathcal{F}_\Psi}(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi)) = \underbrace{W(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi))}_{\text{generator approximation error}} + \underbrace{d_{\mathcal{F}_\Psi}(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi)) - W(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi))}_{\text{discriminator approximation error II}}. \tag{23}$$

Here we carry out the explanation of the inequalities along with specification of some notations. As a general rule, upper subscript B denotes truncated data bounded by some constant B, hat $\hat{\cdot}$ is a notation for simulated distributions, and tilde $\tilde{\cdot}$ is for the empirical distribution of real data, (θ^*, ϕ^*) and (θ, ϕ) denote solutions of different optimization problems described in the following.

- For two distributions P and Q, we have

$$d_{F_\Psi}(P, Q) = \sup_{f_\psi \in \mathcal{F}_\Psi} \mathbb{E}_{\boldsymbol{X}_P \sim P}[f_\psi(\boldsymbol{X}_P)] - \mathbb{E}_{\boldsymbol{X}_Q \sim Q}[f_\psi(\boldsymbol{X}_Q)].$$

  We remark that $d_{\mathcal{F}_\Psi}(\cdot, \cdot)$, which serves as an approximation to $W(\cdot, \cdot)$, is not necessarily nonnegative, but satisfies the inequality

$$d_{F_\Psi}(P, Q) \leq d_{F_\Psi}(P, R) + d_{F_\Psi}(R, Q),$$

  which serves as the reason for (iii). Also, note that (i) is due to the triangular inequality of Wasserstein distance.
- $\tilde{\pi}^B$ is the bounded empirical distribution, of which distance from the bounded real distribution π^B is due to
  (1) limitation in sample size
  (2) weak correlation among the sample sequences, if we allow for it to exist
- The two pairs of parameters for the neural networks integrated in the generator, namely (θ^*, ϕ^*) and (θ, ϕ), are solutions to different optimization problems defined as follows:

$$(\theta^*, \phi^*) = \arg\min_{\theta, \phi} d_{\mathcal{F}_\Psi}(\tilde{\pi}^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi)), \tag{24}$$

$$(\theta, \phi) = \arg\min_{\theta, \phi} \|\mu_\theta - \mu\|_{L^\infty([-B,B])} + \|\Sigma_\phi - \Sigma\|_{L^\infty([-B,B])}. \tag{25}$$

  Note that (24) is the reason for (ii).

- μ_θ and Σ_ϕ are likely to be different from the optimal solutions μ_{θ^*} and Σ_{ϕ^*} for the following reasons:
  (1) In the minimax optimization problem, the distribution simulated with μ_{θ^*} and Σ_{ϕ^*} is directed towards $\tilde{\pi}^B$ but not π.
  (2) The difference between the function classes $\mathcal{F}_\Psi = \mathcal{F}_\Psi(\kappa, L, P, K, \epsilon_f)$ and $\mathcal{F}_{\text{Lip}} := \{f : \|f\|_L \leq 1\}$ also induces some difference between the "optimal solutions" and the "optimal networks". This portion of error is bounded by *discriminator approximation error* I and II.

3.6.2 Network Approximation. Before controlling the error terms, we introduce the foundation of proof, which is the deep neural network approximation theory. We first present a theorem of approximating C_L-Lipschitz functions on $[-B, B]^d$. Ideas and proofs of this theorem are adopted from [11, 12] and [15].

Theorem 3.4 (Approximation with Explicit Order Dependence on Bound B). *For*

$$f \in [B, B]^d \text{ s.t. } \|f(\boldsymbol{x}) - f(\boldsymbol{y})\| \leq C_L \|\boldsymbol{x} - \boldsymbol{y}\|,$$

we have a neural network Φ_f *with* $\mathcal{L}(\Phi_f) = O(\log B + \log \delta^{-1})$, $\mathcal{W}(\Phi_f) = O(B^d \delta^{-d})$, $\mathcal{B}(\Phi_f) = O(B\delta^{-1})$ *and* $\mathcal{M}(\Phi_f) = O((\log B + \log \delta^{-1}) \cdot B^d \delta^{-d})$, $B \to \infty$ *and* $\delta \to 0$, *satisfying*

$$\|\Phi_f(\boldsymbol{x}) - f(\boldsymbol{x})\|_{L^\infty([-B,B]^d)} \leq \delta. \tag{26}$$

where notations for the size of ReLU network Φ *are defined as*

- *depth* $\mathcal{L}(\Phi) = L$
- *width* $\mathcal{W}(\Phi) = \max_{l=0,\ldots,L} N_l$, *where* N_l *is the width of the* l*th layer*
- *weight magnitude* $\mathcal{B}(\Phi) = \max_{l=1,\ldots,L} \max\{\|W_l\|_\infty, \|b_l\|_\infty\}$
- *number of neurons* $\mathcal{M}(\Phi) = \sum_{l=1}^{L} \|W_l\| + \|b_l\|$

The proof of Theorem 3.4 consists of two steps: approximating Lipschitz functions with interpolation polynomials, and approximating the polynomials with neural networks. We refer to Appendix B for details.

3.6.3 Controlling Error Terms. In this section, we describe how each error term is controlled. We defer more details to the appendix to complete the proof.

First, for bounding error $W(\pi, \pi^B)$, applying the definition of Wasserstein distance and using the condition that $p_{\eta_k}(x) \leq O(x^{-(2+\alpha)})$, $x \to \infty$, we have

$$W(\pi, \pi^B) \leq O(B^{-\alpha}), \quad B \to \infty. \tag{27}$$

Second, for generator approximation error $W(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi))$, which is induced by the errors of approximating μ and Σ with optimal neural networks μ_θ and Σ_ϕ, denoted as ϵ_μ and ϵ_Σ, we apply the law of total expectation on the sequence to derive that the generator approximation error can be bounded as

$$W(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi)) \leq O(\epsilon_\mu + \epsilon_\Sigma), \quad \epsilon_\mu \to 0 \text{ and } \epsilon_\Sigma \to 0. \tag{28}$$

Note that, replacing δ in Theorem 3.4 with ϵ_μ and ϵ_Σ, we can derive the required sizes of generator networks μ_θ and Σ_ϕ to achieve levels ϵ_μ, ϵ_Σ of approximation errors.

Next, we control the discriminator approximation error terms. We have the following lemma, the proof of which is given in Appendix D.

Lemma 3.5. *The two function classes* $\mathcal{F}_\Psi(\kappa, L, P, K, \epsilon_f)$ *and* 1*-Lipchitz class* $\mathcal{F}_{Lip}$ *can approximate each other within the bounded area* $[-B, B]^d$, *namely,*

(1) $\forall f \in \mathcal{F}_{Lip}$, $\exists f_\psi \in \mathcal{F}_\Psi$, *such that* $\|f - f_\psi\|_{L^\infty[-B,B]^d} \le \epsilon_f$;
(2) $\forall f \in \mathcal{F}_\Psi$, $\exists f_\psi \in \mathcal{F}_{\text{Lip}}$, *such that* $\|f - f_\psi\|_{L^\infty[-B,B]^d} \le 3\epsilon_f$.

Using Lemma 3.5, we have for the discriminator approximation error terms,

$$\begin{aligned}
&W(\tilde{\pi}^B, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*})) - d_{\mathcal{F}_\Psi}(\tilde{\pi}^B, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*})) \\
&= \sup_{\|f\|_L \le 1} [\mathbb{E}_{\boldsymbol{X}\sim\tilde{\pi}^B} f(\boldsymbol{X}) - \mathbb{E}_{\boldsymbol{X}\sim\hat{\pi}^B} f(\boldsymbol{X})] - \sup_{f_\psi \in \mathcal{F}_\Psi} \left[\mathbb{E}_{\boldsymbol{X}\sim\tilde{\pi}^B} f_\psi(\boldsymbol{X}) - \mathbb{E}_{\boldsymbol{X}\sim\hat{\pi}^B} f_\psi(\boldsymbol{X})\right] \\
&= \inf_{f_\psi \in \mathcal{F}_\Psi} \sup_{\|f\|_L \le 1} \mathbb{E}_{\boldsymbol{X}\sim\tilde{\pi}^B} \left[f(\boldsymbol{X}) - f_\psi(\boldsymbol{X})\right] + \inf_{f_\psi \in \mathcal{F}_\Psi} \sup_{\|f\|_L \le 1} \mathbb{E}_{\boldsymbol{X}\sim\hat{\pi}^B} \left[f_\psi(\boldsymbol{X}) - f(\boldsymbol{X})\right] \\
&\le \inf_{f_\psi \in \mathcal{F}_\Psi} \sup_{\|f\|_L \le 1} 2\|f - f_\psi\|_\infty \le 2\epsilon_f.
\end{aligned} \tag{29}$$

Also,

$$\begin{aligned}
&d_{\mathcal{F}_\Psi}(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi)) - W(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi)) \\
&= \sup_{f_\psi \in \mathcal{F}_\Psi} \left[\mathbb{E}_{X\sim\pi^B} f_\psi(\boldsymbol{X}) - \mathbb{E}_{\boldsymbol{X}\sim\hat{\pi}^B} f_\psi(\boldsymbol{X})\right] - \sup_{\|f\|_L \le 1} [\mathbb{E}_{\boldsymbol{X}\sim\pi^B} f(\boldsymbol{X}) - \mathbb{E}_{\boldsymbol{X}\sim\hat{\pi}^B} f(\boldsymbol{X})] \\
&= \inf_{\|f\|_L \le 1} \sup_{f_\psi \in \mathcal{F}_\Psi} \mathbb{E}_{\boldsymbol{X}\sim\pi^B} \left[f(\boldsymbol{X}) - f_\psi(\boldsymbol{X})\right] + \inf_{\|f\|_L \le 1} \sup_{f_\psi \in \mathcal{F}_\Psi} \mathbb{E}_{\boldsymbol{X}\sim\hat{\pi}^B} \left[f_\psi(\boldsymbol{X}) - f(\boldsymbol{X})\right] \\
&\le \inf_{\|f\|_L \le 1} \sup_{f_\psi \in \mathcal{F}_\Psi} 2\|f - f_\psi\|_\infty \le 6\epsilon_f,
\end{aligned} \tag{30}$$

After that, we control the statistical error terms $d_{\mathcal{F}_\Psi}(\tilde{\pi}^B, \pi^B)$ and $W(\pi^B, \tilde{\pi}^B)$. We have

$$\begin{aligned}
\mathbb{E}\left[d_{\mathcal{F}_\Psi}(\tilde{\pi}^B, \pi^B)\right] &= \mathbb{E}\left[\sup_{f_\psi \in \mathcal{F}_\Psi} \frac{1}{n}\sum_{j=1}^n f_\psi(\boldsymbol{X}_j) - \mathbb{E}_{Y\sim\pi^B}[f_\psi(\boldsymbol{Y})]\right] \\
&\le \mathbb{E}_{\boldsymbol{X}} \mathbb{E}_{\boldsymbol{Y}_j\sim\pi^B, i.i.d.} \left[\sup_{f \in \mathcal{F}_\Psi} \frac{1}{n}\sum_{j=1}^n [f_\psi(\boldsymbol{X}_j) - f_\psi(\boldsymbol{Y}_j)]\right].
\end{aligned} \tag{31}$$

If $\boldsymbol{X}_j$ are correlated, according to the assumption of weak correlation, we created independent variables $\bar{\boldsymbol{X}}_j$ such that the joint distribution of $(\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n)$ and $(\bar{\boldsymbol{X}}_1, \bar{\boldsymbol{X}}_2, \ldots, \bar{\boldsymbol{X}}_n)$, denoted as $\gamma(\Pi_n, \bar{\Pi}_n)$, satisfies

$$\gamma^*(\Pi_n, \bar{\Pi}_n) = \arg \inf_{\gamma \in \gamma(\Pi_n, \bar{\Pi}_n)} \mathbb{E}_{(\boldsymbol{X}, \bar{\boldsymbol{X}})\sim\gamma(\Pi_n, \bar{\Pi}_n)} \left[\|\boldsymbol{X} - \bar{\boldsymbol{X}}\|\right].$$

According to the definition of function class $\mathcal{F}_\Psi(\kappa, L, P, K, \epsilon_f)$, we have

$$\begin{aligned}
&\mathbb{E}_{\boldsymbol{X}} \mathbb{E}_{\boldsymbol{Y}_j\sim\pi^B, i.i.d.} \left[\sup_{f \in \mathcal{F}_\Psi} \frac{1}{n}\sum_{j=1}^n [f_\psi(\boldsymbol{X}_j) - f_\psi(\boldsymbol{Y}_j)]\right] \\
\le &\mathbb{E}\left[\sup_{f \in \mathcal{F}_\Psi} \frac{1}{n}\sum_{j=1}^n [f_\psi(\bar{\boldsymbol{X}}_j) - f_\psi(\boldsymbol{Y}_j)]\right] + \mathbb{E}\left[\frac{1}{n}\sum_{j=1}^n \|\boldsymbol{X}_j - \bar{\boldsymbol{X}}_j\|\right] + 2\epsilon_f.
\end{aligned}$$

[12] suggests that

$$\mathbb{E}\left[\sup_{f_\psi \in \mathcal{F}_\Psi} \frac{1}{n}\sum_{j=1}^n [f_\psi(\bar{\boldsymbol{X}}_j) - f_\psi(\boldsymbol{Y}_j)]\right] \le 2 \inf_{0<\delta<M} \left(2\delta + \frac{12}{\sqrt{n}} \int_\delta^M \sqrt{\log \mathcal{N}(\epsilon, \mathcal{F}_\Psi, \|\cdot\|_\infty)} d\epsilon\right), \tag{32}$$

where $M = \text{diam}(\mathcal{F}_\Psi(\kappa, L, P, K, \epsilon_f)) \le R = O(B)$, and $\mathcal{N}(\epsilon, \mathcal{F}_\Psi, \|\cdot\|_\infty)$ is the covering number [26] of ϵ-balls over the functional class $\mathcal{F}_\Psi$ under distance $\|\cdot\|_\infty$. Further, let $f_\psi, f_{\psi'} \in \mathcal{F}_\Psi(\kappa, L, P, K)$ with

all weight parameters at most h from each other, [12] shows that with bounded support $\|x\|_\infty \le B$, we have

$$\|f_\psi - f_{\psi'}\|_\infty \le hL(PB+2)(\kappa P)^{L-1}.$$

Discretizing each parameter uniformly into κ/h grids yields a δ-covering on $\mathcal{F}_\Psi$, therefore,

$$\mathcal{N}(\delta, \mathcal{F}_\Psi(\kappa, L, P, K), \|\cdot\|_\infty) \le (LP^2)^K \left(\frac{2\kappa}{h}\right)^K,$$

where

$$h = \frac{\delta}{L(PB+2)(\kappa P)^{L-1}}.$$

Further, we have

$$\mathcal{N}(\delta, \mathcal{F}_\Psi(\kappa, L, P, K, \epsilon_f), \|\cdot\|_\infty) \le \mathcal{N}\left(\frac{\delta}{2}, \mathcal{F}_\Psi(\kappa, L, P, K), \|\cdot\|_\infty\right) \le \left(\frac{4L^2(PB+2)(\kappa P)^{L+1}}{\delta}\right)^K. \quad (33)$$

Substituting (33) into (32) and taking $\delta = 1/n$ yields

$$\mathbb{E}\left[\sup_{f\in\mathcal{F}_\Psi} \frac{1}{n}\sum_{j=1}^n [f_\psi(\bar{X}_j) - f_\psi(Y_j)]\right] \le O\left(B\sqrt{n^{-1}KL\log(nL\kappa PB)}\right).$$

Also, by Chebyshev inequality,

$$\mathbb{E}\left[\frac{1}{n}\sum_{j=1}^n \|X_j - \bar{X}_j\|\right] \le \frac{1}{\sqrt{n}} W(\Pi_n, \bar{\Pi}_n) = O\left(n^{-\beta}\right).$$

Similarly, for the Wasserstein statistical error $W(\pi^B, \tilde{\pi}^B)$, we have

$$\mathcal{N}(\delta, \mathcal{F}_{\text{Lip}}, \|\cdot\|_\infty) \le \left(\frac{2B}{\delta}+1\right)^{pd(2B/\delta+1)}.$$

Therefore, taking $\delta^{-1} = n^{(pd-1)/(pd+1)}$, we have

$$2\inf_{0<\delta<B}\left(2\delta + \frac{12}{\sqrt{n}}\int_\delta^B \sqrt{\log \mathcal{N}(\delta, \mathcal{F}_{\text{Lip}}, \|\cdot\|_\infty)}d\epsilon\right) \le O\left(B^{\frac{3}{2}} n^{-\frac{1}{pd+1}}\sqrt{\log(Bn)}\right).$$

And finally,

$$W(\pi^B, \tilde{\pi}^B) \le O\left(B^{\frac{3}{2}} n^{-\frac{1}{pd+1}}\sqrt{\log(nB)} + n^{-\beta}\right) + 2\epsilon_f.$$

3.6.4 Balancing. Finally, we balance the error terms relevant to the discriminator class. Summarizing the four parts of errors, we have

$$\begin{aligned}&\mathbb{E}\left[W(\pi, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*}))\right] \\ &= O\left(B^{-\alpha} + \epsilon_\Sigma + \epsilon_\mu + \epsilon_f + B\sqrt{n^{-1}KL\log(nL\kappa PB)} + B^{\frac{3}{2}} n^{-\frac{1}{pd+1}}\sqrt{\log(Bn)} + n^{-\beta}\right). \quad (34)\end{aligned}$$

Also, since ϵ_f is the approximation tolerance of the discriminator class, we have from Theorem 3.4, $L = O(\log B + \log \epsilon_f^{-1})$, $\kappa = O(B\epsilon_f^{-1})$, $P = O(B^{pd}\epsilon_f^{-pd})$ and $K = O((\log B + \log \epsilon_f^{-1}) \cdot B^{pd}\epsilon_f^{-pd})$. To

balance these terms, we mostly pay attention to $B\sqrt{n^{-1}KL\log(nL\kappa PB)}$ and $B^{\frac{3}{2}}n^{-\frac{1}{pd+1}}\sqrt{\log(Bn)}$. To make sure that these two terms converge to 0 as $n \to \infty$, let $B = n^{k_1}$ and $\epsilon_f = n^{-k_2}$, we have

$$\begin{cases} \left(\frac{pd}{2}+1\right)k_1 + \frac{pd}{2}k_2 < \frac{1}{2}. \\ \frac{3}{2}k_1 < \frac{1}{pd+1}, \end{cases}$$

We can set $k_1 = \frac{2}{3pd+6}$ and $k_2 = \frac{1}{3pd+6}$. Also, make the generator network classes large enough so that ϵ_Σ and ϵ_μ are not of leading order, then

$$\mathbb{E}\left[W(\pi, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*}))\right] \le O\left(n^{-\frac{1}{3pd+6}}(\log n)^{\frac{3}{2}} + n^{-\frac{1}{(pd+2)(pd+1)}}(\log n)^{\frac{1}{2}} + n^{-\frac{2\alpha}{3pd+6}} + n^{-\beta}\right).$$

With $pd \ge 3$, we have

$$\mathbb{E}\left[W(\pi, \hat{\pi}^B(\mu_{\theta^*}, \Sigma_{\phi^*}))\right] \le O\left(n^{-\frac{1}{3pd+6}}(\log n)^{\frac{3}{2}} + n^{-\frac{2\alpha}{3pd+6}} + n^{-\beta}\right). \tag{35}$$

and this is achieved by taking $B = O(n^{\frac{2}{3pd+6}})$,

$$L = O(\log n),\ \kappa = O\left(n^{\frac{1}{pd+2}}\right),\ P = O\left(n^{\frac{pd}{pd+2}}\right),\ K = O\left(n^{\frac{pd}{pd+2}}\log n\right),$$

for the discriminator, where $n \to \infty$. For the generator networks μ_θ and Σ_ϕ, we take

$$\epsilon_\mu = \epsilon_\Sigma = O\left(\min\left\{n^{-\frac{1}{3pd+6}}(\log n)^{\frac{3}{2}},\ n^{-\frac{2\alpha}{3pd+6}},\ n^{-\beta}\right\}\right),$$

and the network sizes of μ_θ and Σ_ϕ are accordingly

$$L = O(\log n),\ \kappa = O\left(n^{\frac{2}{3pd+6}}\epsilon_\mu^{-1}\right),\ P = \left(n^{\frac{2pd}{3pd+6}}\epsilon_\mu^{-pd}\right),\ K = O\left(n^{\frac{2pd}{3pd+6}}\epsilon_\mu^{-pd}\log n\right)$$

This result provides insights for selecting the neural network sizes, when the training set is large and high precision of modeling is expected to be achieved.

4 NUMERICAL EXPERIMENTS

In this section, we evaluate the performance of the simulator estimated by our proposed framework, using four sets of synthetic data (Sections 4.1, 4.2, 4.3, and 4.4) and one set of real data (Section 4.5) as the training data.[1] In all experiments, we illustrate the use of the simulator by considering scenarios where the simulator is applied to generate sequences of prices of multiple correlated assets. Our proposed framework then aims to estimate the simulators such that the joint distribution of the simulated data has a close Wasserstein distance compared to that of the training data. To demonstrate the performance of estimated simulators in their practical use, we first consider the task of evaluating the distribution of the **maximal drawdown (MDD)** for all the assets in consideration. In each experiment, we consider the distribution of the MDD of all observed dimensions of the sequential data. We compare the MDD distribution of the simulated data-based portfolio against the MDD distribution of the empirical data-based portfolio. The simulator that simulates the sequential price data is estimated by our proposed method. In this way, we aim to demonstrate the performance of our method from an operational performance point of view. We present the definition of maximal drawdown, which is derived from [10]:

[1] see https://github.com/Goldenbean0521/Sequential-code for code.

Definition 4.1. Let $(H_i \in \mathbb{R} : i = 0, 1, 2, \ldots, p)$ be a portfolio sequence, and let H_i be the portfolio value at time step i. The portfolio *drawdown* at time step i is defined by

$$D_i = \max_{0 \le l \le i} \frac{H_l - H_i}{H_i}, \tag{36}$$

and the *maximal drawdown* of a sequence is the maximum value of D_i over all time steps $i = 0, 1, 2, \ldots, p$, namely, $M = \max_{0 \le i \le p} D_i$.

Apart from that, we demonstrate the ability of the network to capture the correlation among the observed dimensions. Specifically, we use the trained networks to simulate 5,000 copies of sequences, estimate the correlation matrices of all observed dimensions at certain time points, and compare such matrices to that of the real (synthetic) data. Such operation is then replicated 100 times to produce a mean value and a standard deviation of the estimations.

4.1 Multi-dimensional Heston Model

In this subsection, we use a synthesized data set of three stock prices that is generated by a multi-dimensional Heston model.

4.1.1 Underlying Model for Synthetic Data: Multi-dimensional Heston. The observed data is 3-dimensional. For each dimension, the underlying stochastic process is formulated by a stochastic differential equation known as the *Heston model* (see [1], which also presents the values of the model parameters):

$$d\begin{pmatrix} S_t \\ Y_t \end{pmatrix} = \begin{pmatrix} \mu S_t \\ \kappa(\gamma - Y_t) \end{pmatrix} dt + \begin{pmatrix} S_t\sqrt{(1-\rho^2)Y_t} & \rho S_t \sqrt{Y_t} \\ 0 & \sigma\sqrt{Y_t} \end{pmatrix} dW_t \tag{37}$$

where $\mu, \kappa, \gamma, \rho, \sigma$ are given parameters, S_t is the observed price process, Y_t is the latent volatility process, and $W_t = (W_t^S, W_t^Y)^\top$ is the 2-dimensional canonical Brownian motion. We use a matrix L to induce correlation among the three stochastic processes, namely, we let

$$d\begin{pmatrix} X_{1,t} \\ X_{2,t} \\ X_{3,t} \end{pmatrix} = \begin{pmatrix} \mu_{1,t} \\ \mu_{2,t} \\ \mu_{3,t} \end{pmatrix} dt + (L \otimes I_2) \cdot \begin{pmatrix} \Sigma_{1,t} & 0 & 0 \\ 0 & \Sigma_{2,t} & 0 \\ 0 & 0 & \Sigma_{3,t} \end{pmatrix} d\begin{pmatrix} W_{1,t} \\ W_{2,t} \\ W_{3,t} \end{pmatrix}, \tag{38}$$

where

$$X_{i,t} = \begin{pmatrix} S_{i,t} \\ Y_{i,t} \end{pmatrix}, \quad \mu_{i,t} = \begin{pmatrix} \mu_i S_{i,t} \\ \kappa_i(\gamma_i - Y_{i,t}) \end{pmatrix}, \quad \Sigma_{i,t} = \begin{pmatrix} S_{i,t}\sqrt{(1-\rho_i^2)Y_{i,t}} & \rho_i S_{i,t}\sqrt{Y_{i,t}} \\ 0 & \sigma_i\sqrt{Y_{i,t}} \end{pmatrix},$$
$$W_{i,t} = \begin{pmatrix} W_{i,t}^S \\ W_{i,t}^Y \end{pmatrix},$$

and $\otimes$ denotes the Kronecker product.

The model parameters are set as in Table 1. Additionally, we have $\mu = r - d$, and L is the Cholesky decomposition of P, given as

$$LL^\top = P = \begin{pmatrix} 1 & 0.3 & 0.2 \\ 0.3 & 1 & 0.4 \\ 0.2 & 0.4 & 1 \end{pmatrix}.$$

We next describe how the data set is synthesized. The initial values are given by $\boldsymbol{S}_0 \sim \mathcal{N}((100, 100, 100), 70P)$, and $Y_0 = (0.1, 0.1, 0.1)$. There are $n = 5{,}000$ sequences generated in total, each having $p = 15$ transitions, with the weekly frequency $\Delta = 7/365$ as the resolution for observation. We use an Euler discretization of the process, setting 30 sub-intervals between every two

Table 1. Parameters of the Underlying Multi-dimensional Heston Model

	r	d	σ	ρ	κ	γ
d_1	0.04	0.015	0.25	−0.8	3	0.1
d_2	0.04	0.015	0.2	−0.75	2.7	0.11
d_3	0.04	0.015	0.15	−0.85	3.3	0.09

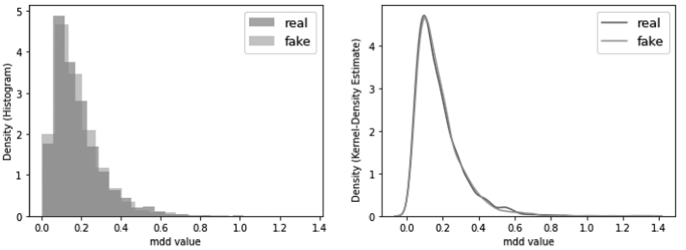


Fig. 1. Maximal draw down distribution, 3-dimensional Heston model, first dimension.

observations, which implies that the resolution for generation is set as $7/(365\times 30)$. This synthetic data set is then used as input data for the discriminator.

4.1.2 Training Process and Results. We specify the structure of the simulator as

$$X_{k+1} = X_k + \mu_\theta(X_k)\Delta t + \Sigma_\phi(X_k)\sqrt{\Delta t}\eta_{k+1} \tag{39}$$

where $\eta_k : k = 1, 2, \ldots$ are independent 6-dimensional canonical normal variables, Δt is set as $0.01/15$, which is not the same as the length of the sub-intervals used in synthetic data generation, but induces only scaling differences that can be eliminated by network parameters. The parameterization of μ_θ is given by $L = 2$, $\tilde{n} = (n_1, n_2) = (100, 6)$. The parameterization of Σ_ϕ is given by $L = 2$, $\tilde{n} = (n_1, n_2) = (100, 36)$. The parameterization of f_ψ is given by $L = 3$, $\tilde{n} = (n_1, n_2, n_3) = (500, 500, 1)$. The initialization of all the parameters of the weight matrices W_l's of μ_θ, Σ_ϕ, f_ψ are given by independent Gaussian random variables with mean 0 and variance 0.1. The vectors b_l's are initialized as constant 3. The gradient penalty coefficient for f_ψ is set as 1, and the batch size for sampling from synthetic data and for simulator generation is set as 256. The initial values S_0 of each simulation process is set to be the same as the initial values of the sample batch, and Y_0 is set as $(0.1, 0.1, 0.1)$. The training process is carried out with 500 iterations using the Adam optimizer [22] with coefficients $\beta_1 = 0.5$ and $\beta_2 = 0.9$. Within each iteration, f_ψ is updated five times. The learning rate of μ_θ, Σ_ϕ and f_ψ decays exponentially from $1e-4$ to $1e-6$. The training takes about 10 minutes using the GPU resource on Google Colab.

For evaluation of training, we first illustrate the comparison between maximal drawdown distribution of the three dimensions. The sizes of the synthesized data set and the simulated data set used for comparison are both 5,000. The results are illustrated in the following Figures 1, 2, and 3.

We also investigate the correlation matrix of the three observed dimensions of data at the 15th observation. By simulating 5,000 sequences using the trained networks μ_θ and Σ_ϕ each time to estimate the correlation matrix, and replicating 100 times to derive the mean value and standard deviation of simulated estimations, we have the following result in Table 2.

4.2 Multi-dimensional Heston Model with Highly Correlated Training Sequences

In this subsection, we demonstrate the performance of our proposed framework when the training sequences are correlated with each other.

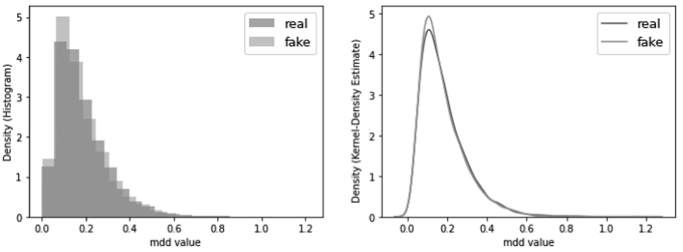


Fig. 2. Maximal draw down distribution, 3-dimensional Heston model, second dimension.

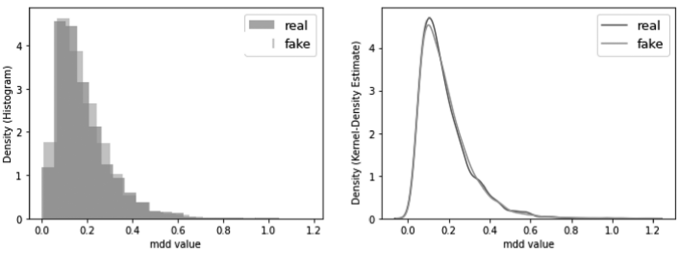


Fig. 3. Maximal draw down distribution, 3-dimensional Heston model, third dimension.

Table 2. Simulated Correlation of the Three Observed Dimensions at Time Step $i = 15$, Multi-dimensional Heston Model

	True value	Sim. mean	Std. dev.
$\text{corr}(S_1, S_2)$	0.3	0.295	0.012
$\text{corr}(S_1, S_3)$	0.2	0.200	0.014
$\text{corr}(S_2, S_3)$	0.4	0.383	0.011

4.2.1 Underlying Model for Synthetic Data: Multi-dimensional Heston. The underlying stochastic process is formulated by the same stochastic differential equation as (37) and (38).

We next describe how the data set is synthesized. The initial values are given by $\boldsymbol{S}_0 \sim \mathcal{N}((100, 100, 100), 70P)$, and $\boldsymbol{Y}_0 = (0.1, 0.1, 0.1)$. There are $n = 5{,}000$ sequences generated in total, each having $p = 15$ transitions, with the weekly frequency $\Delta = 7/365$ as the resolution for observation. We use an Euler discretization of the process, setting 30 sub-intervals between every two observations, which implies that the resolution for generation is set as $7/(365 \times 30)$. Namely, $(X_{i,k} : i = 1, 2, 3; k = 0, 1, 2, \ldots)$ is sequentially generated according to

$$\begin{pmatrix} X_{1,k+1} \\ X_{2,k+1} \\ X_{3,k+1} \end{pmatrix} = \begin{pmatrix} X_{1,k} \\ X_{2,k} \\ X_{3,k} \end{pmatrix} + \begin{pmatrix} \mu_{1,k} \\ \mu_{2,k} \\ \mu_{3,k} \end{pmatrix} \Delta t + (L \otimes I_2) \cdot \begin{pmatrix} \Sigma_{1,k} & 0 & 0 \\ 0 & \Sigma_{2,k} & 0 \\ 0 & 0 & \Sigma_{3,k} \end{pmatrix} d \begin{pmatrix} \sqrt{\Delta t}\eta_{1,k+1} \\ \sqrt{\Delta t}\eta_{2,k+1} \\ \sqrt{\Delta t}\eta_{3,k+1} \end{pmatrix}, \quad (40)$$

where

$$X_{i,k} = \begin{pmatrix} S_{i,k} \\ Y_{i,k} \end{pmatrix}, \quad \mu_{i,k} = \begin{pmatrix} \mu_i S_{i,k} \\ \kappa_i(\gamma_i - Y_{i,k}) \end{pmatrix}, \quad \Sigma_{i,k} = \begin{pmatrix} S_{i,k}\sqrt{(1-\rho_i^2)Y_{i,k}} & \rho_i S_{i,k}\sqrt{Y_{i,k}} \\ 0 & \sigma_i\sqrt{Y_{i,k}} \end{pmatrix},$$

$$\eta_{i,k+1} = \begin{pmatrix} \eta^S_{i,k+1} \\ \eta^Y_{i,k+1} \end{pmatrix}.$$

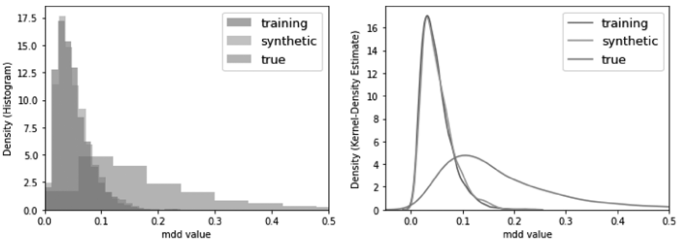


Fig. 4. Maximal draw down distribution, 3-dimensional Heston model with highly correlated training sequences, first dimension.

$\eta^j_{i,k}, i = 1,2,3; k = 1,2,\ldots, j = S, Y$ are independent standard normal random variables. To create correlation among the 5,000 training-sequences, for each i, k, and j, the $\eta^j_{i,k}$'s in the 5,000 sequences are simultaneously generated from a 5,000-dimensional multivariate normal distribution $\mathcal{N}(0, \Sigma)$, where

$$\Sigma = \begin{pmatrix} 1 & \rho & \cdots & \rho \\ \rho & 1 & \cdots & \rho \\ \vdots & \vdots & \ddots & \vdots \\ \rho & \rho & \cdots & 1 \end{pmatrix}.$$

We set $\rho = 0.9$ in our synthetic data set. In other words, for each $i \in \{1,2,3\}, k \in \{1,2,\ldots\}, j \in \{S, Y\}$, the correlation of $\eta^j_{i,k}$'s (i.e., the random noises) in any two different sequences is 0.9, which ensures that the training-sequences are highly correlated with each other. This synthetic data set is then used as input data for the discriminator.

4.2.2 Training Process and Results. We specify the structure of the simulator to have the same form as (39). The parameterization and initialization of the neural networks μ_θ, Σ_ϕ and f_ψ, as well as the iterative optimization process are similar to those of the first experiment. Note that although the training sequences are highly correlated with each other, the sequences in the simulated set generated by our framework are independent. The training takes about 10 minutes.

Since the training sequences are highly correlated with each other, the empirical distribution of the training data might deviate from the real distribution of the underlying process. Considering this difference, we generate another data set with independent sequences (hereafter "uncorrelated set") according to (40). The statistics of the uncorrelated set will be unbiased estimators of the real statistics of the underlying process. For evaluation of training, we compare the distribution of our simulated data set with the distribution of both the training set and the uncorrelated set.

First, we illustrate the comparison among the maximal drawdown distribution of the three dimensions. The sizes of the synthesized data set and the simulated data set used for comparison are both 5,000. We have the following results in Figures 4, 5, and 6. The results indicate that although the empirical distribution of the training data has deviated from the real distribution of the underlying process, the distribution of the synthesized data set is still comparable to the distribution of the training set.

We also investigate the correlation matrix of the three observed dimensions of data at the 15th observation. By simulating 5,000 sequences using the trained networks μ_θ and Σ_ϕ each time to estimate the correlation matrix, and replicating 100 times to derive the mean value and standard deviation of simulated estimations, we have the following result in Table 3.

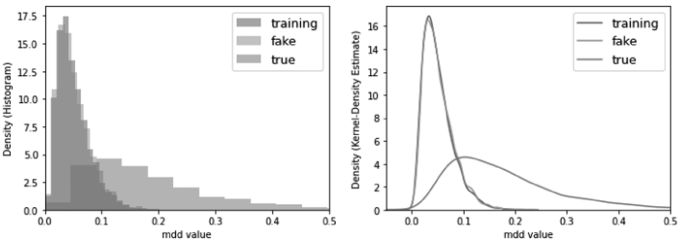


Fig. 5. Maximal draw down distribution, 3-dimensional Heston model with highly correlated training sequences, second dimension.

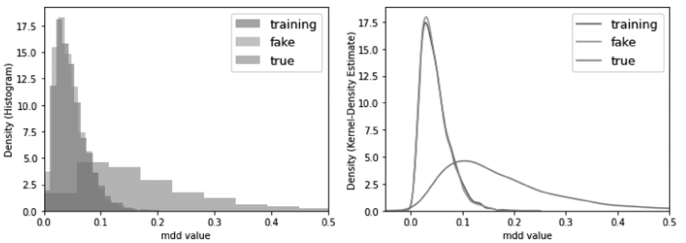


Fig. 6. Maximal draw down distribution, 3-dimensional Heston model with highly correlated training sequences, third dimension.

Table 3. Simulated Correlation of the Three Observed Dimensions at Time Step $i = 15$, Multi-dimensional Heston Model with Highly Correlated Training Sequences

	Training Set	Uncorrelated Set	Sim. mean	Std. dev.
$\text{corr}(S_1, S_2)$	0.28	0.27	0.29	0.008
$\text{corr}(S_1, S_3)$	0.20	0.19	0.17	0.009
$\text{corr}(S_2, S_3)$	0.42	0.42	0.43	0.008

4.3 Multi-dimensional Polynomial Model

In this subsection, we use a synthesized data set generated by a nonlinear SDE-based stochastic process.

4.3.1 Underlying Model for Synthetic Data: Multi-dimensional Polynomial. The underlying stochastic process is formulated by a stochastic differential equation given by:

$$\begin{aligned} &d\,(\boldsymbol{S}_t, \boldsymbol{Y}_t)^\top \\ &= \mu(\boldsymbol{S}_t, \boldsymbol{Y}_t)dt + \Sigma(\boldsymbol{S}_t, \boldsymbol{Y}_t)d\boldsymbol{W}_t \end{aligned} \tag{41}$$

where

$$\begin{aligned} &\mu(\boldsymbol{S}_t, \boldsymbol{Y}_t) \\ &= \left(S_{1,t}^{0.2} + S_{2,t}^{0.2} + 1 \quad S_{2,t}^{0.3} + 0.02 S_{1,t} S_{3,t} \quad S_{3,t}^{0.25} + 0.01 S_{1,t} \quad Y_{2,t} + Y_{3,t} \quad Y_{3,t} + Y_{1,t} \quad Y_{1,t} + Y_{2,t} \right)^\top, \end{aligned}$$

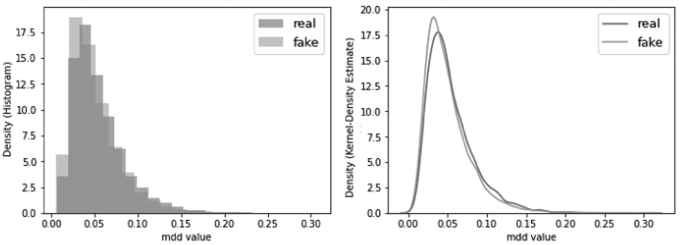


Fig. 7. Maximal draw down distribution, 3-dimensional polynomial model, first dimension.

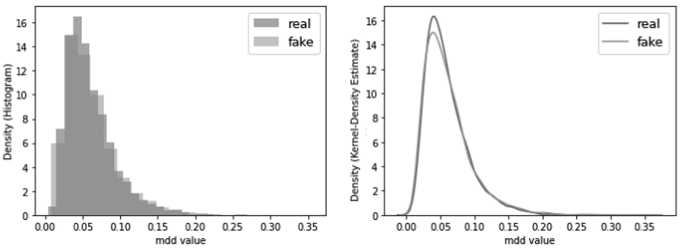


Fig. 8. Maximal draw down distribution, 3-dimensional polynomial model, second dimension.

and

$$\Sigma(S_t, Y_t) = \begin{pmatrix} -2S_{1,t}^{1.2}Y_{1,t} & S_{1,t}S_{2,t}Y_{2,t} & 2S_{1,t}S_{3,t}Y_{3,t} & 0.1S_{1,t}Y_{1,t} & 0.5S_{1,t}Y_{2,t} & 0.7S_{1,t}Y_{3,t} \\ 01.5S_{1,t}S_{2,t}Y_{1,t} & 7S_{2,t}^{1.1}Y_{2,t} & S_{1,t}S_{3,t}Y_{3,t} & 0.1S_{2,t}Y_{1,t} & 0.2S_{2,t}Y_{2,t} & 0.2S_{2,t}Y_{2,t} \\ 3S_{1,t}S_{3,t}Y_{1,t} & S_{2,t}S_{3,t}Y_{2,t} & -Y_{3,t}S_{1,t}S_{3,t}^{1.2} & 0.2S_{3,t}Y_{1,t} & 0.6S_{3,t}Y_{2,t} & 0.3S_{3,t}Y_{3,t} \\ Y_{1,t} & 0 & 0 & Y_{2,t}Y_{3,t} & Y_{3,t}Y_{1,t} & Y_{1,t}Y_{2,t} \\ 0 & Y_{2,t} & 0 & Y_{1,t}Y_{3,t} & Y_{1,t}Y_{2,t} & Y_{3,t}Y_{2,t} \\ 0 & 0 & Y_{3,t} & Y_{2,t}Y_{1,t} & Y_{3,t}Y_{2,t} & Y_{1,t}Y_{3,t} \end{pmatrix}.$$

We next describe how the data set is synthesized. The initial values are given by $S_0 \sim \mathcal{N}((25, 25, 15), 1)$, where all three dimensions are independent, and $Y_0 = (0.1, 0.1, 0.1)$. There are $n = 5{,}000$ sequences generated in total, each having $p = 25$ observed points with frequency $\Delta = 0.01$ as the resolution for observation. We use an Euler discretization of the process, setting 15 sub-intervals between every two observations, which implies that the resolution for generation is set as $\Delta t = 0.00067$. This synthetic data set is then used as input data for the discriminator.

4.3.2 Training Process and Results. We specify the structure of the simulator to have the same form as (39). The parameterization and initialization of the neural networks μ_θ, Σ_ϕ and f_ψ, as well as the iterative optimization process are similar to those of the first experiment. The training takes about 10 minutes.

For evaluation of training, we first illustrate the comparison between maximal drawdown distribution of the three dimensions. The sizes of the synthesized data set and the simulated data set used for comparison are both 5,000. The results are illustrated in the following Figures 7, 8, and 9.

We also investigate the correlation matrix of the three dimensions of data at the 25th observation. By generating 5,000 sequences using the trained networks μ_θ and Σ_ϕ each time to estimate the correlation matrix, and replicating 100 times to derive the mean value and standard deviation of simulated estimations, we have the following results in Table 4. Note that the true value is now estimated from real (synthetic) data.

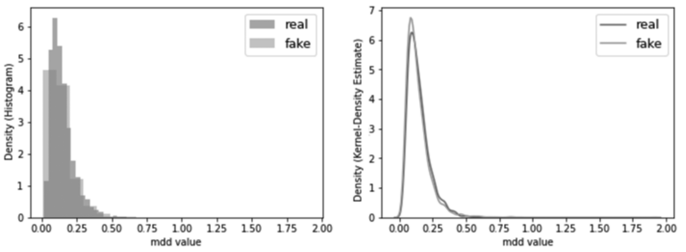


Fig. 9. Maximal draw down distribution, 3-dimensional polynomial model, third dimension.

Table 4. Simulated Correlation of the Three Observed Dimensions at Time Step $i = 15$, Polynomial Model

	Est. true value	Sim. mean	Std. dev.
$\mathrm{cor}(S_1, S_2)$	0.187	0.213	0.014
$\mathrm{cor}(S_1, S_3)$	−0.157	−0.164	0.015
$\mathrm{cor}(S_2, S_3)$	0.442	0.441	0.010

4.4 Multi-dimensional Polynomial Model with Heavy-tailed Noises

In this subsection, we demonstrate the performance of our proposed framework when the distribution of the elementary randomness (i.e., η_k's) are heavy-tailed.

4.4.1 Underlying Model for Synthetic Data: Multi-dimensional Polynomial with Heavy-tailed Randomness. The underlying stochastic process is sequentially generated according to

$$(S_{k+1}, Y_{k+1})^\top = (S_k, Y_k)^\top + \mu(S_k, Y_k)\Delta t + \Sigma(S_k, Y_k)\Delta t \Delta\eta_{k+1}, \tag{42}$$

where $S_k : k = 0, 1, 2, \ldots$ and $Y_k : k = 0, 1, 2, \ldots$ are 3-dimensional vectors; $\mu(S_k, Y_k)$ and $\Sigma(S_k, Y_k)$ have the same form with (41). Specifically, we have

$$\mu(\boldsymbol{S}_k, \boldsymbol{Y}_k) = \left(S_{1,k}^{0.2} + S_{2,k}^{0.2} + 1 \quad S_{2,k}^{0.3} + 0.02 S_{1,k} S_{3,k} \quad S_{3,k}^{0.25} + 0.01 S_{1,k} \quad Y_{2,k} + Y_{3,k} \quad Y_{3,k} + Y_{1,k} \quad Y_{1,k} + Y_{2,k} \right)^\top,$$

and

$$\Sigma(\boldsymbol{S}_k, \boldsymbol{Y}_k) = \begin{pmatrix} -2S_{1,k}^{1.2} Y_{1,k} & S_{1,k} S_{2,k} Y_{2,k} & 2S_{1,k} S_{3,k} Y_{3,k} & 0.1 S_{1,k} Y_{1,k} & 0.5 S_{1,k} Y_{2,k} & 0.7 S_{1,k} Y_{3,k} \\ 01.5 S_{1,k} S_{2,k} Y_{1,k} & 7S_{2,k}^{1.1} Y_{2,k} & S_{1,k} S_{3,k} Y_{3,k} & 0.1 S_{2,k} Y_{1,k} & 0.2 S_{2,k} Y_{2,k} & 0.2 S_{2,k} Y_{2,k} \\ 3S_{1,k} S_{3,k} Y_{1,k} & S_{2,k} S_{3,k} Y_{2,k} & -Y_{3,k} S_{1,k} S_{3,k}^{1.2} & 0.2 S_{3,k} Y_{1,k} & 0.6 S_{3,k} Y_{2,k} & 0.3 S_{3,k} Y_{3,k} \\ Y_{1,k} & 0 & 0 & Y_{2,k} Y_{3,k} & Y_{3,k} Y_{1,k} & Y_{1,k} Y_{2,k} \\ 0 & Y_{2,k} & 0 & Y_{1,k} Y_{3,k} & Y_{1,k} Y_{2,k} & Y_{3,k} Y_{2,k} \\ 0 & 0 & Y_{3,k} & Y_{2,k} Y_{1,k} & Y_{3,k} Y_{2,k} & Y_{1,k} Y_{3,k} \end{pmatrix}.$$

$\Delta\eta_k : k = 1, 2, \ldots$ are i.i.d. 6-dimensional vectors. In each $\Delta\eta_k$, the 6 dimensions are i.i.d. variables following t-distribution with 2.5 degrees of freedom.

We next describe how the data set is synthesized. The initial values are given by $S_0 \sim \mathcal{N}((25, 25, 15), 1)$, where all three dimensions are independent, and $Y_0 = (0.1, 0.1, 0.1)$. The resolution for generation is set as $\Delta t = 0.00067$. There are $n = 5{,}000$ sequences generated in total, each having $p = 25$ observed points with frequency $\Delta = 0.01$ as the resolution for observation. Thus,

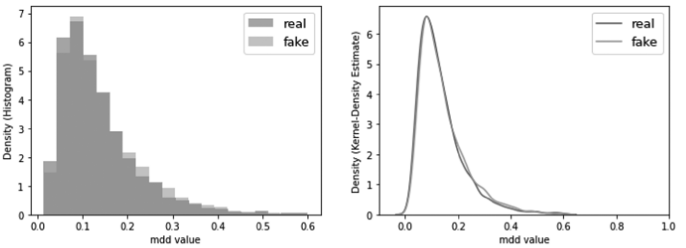


Fig. 10. Maximal draw down distribution, 3-dimensional polynomial model with heavy-tailed noises, first dimension.

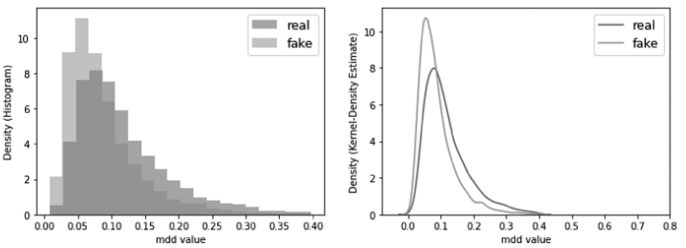


Fig. 11. Maximal draw down distribution, 3-dimensional polynomial model with heavy-tailed noises, second dimension.

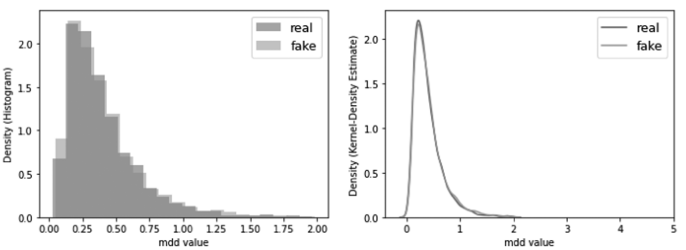


Fig. 12. Maximal draw down distribution, 3-dimensional polynomial model with heavy-tailed noises, third dimension.

there are 15 sub-intervals between every two observations. This synthetic data set is then used as input data for the discriminator.

4.4.2 Training Process and Results. We specify the structure of the simulator to have the same form as (39). The parameterization and initialization of the neural networks μ_θ, Σ_ϕ and f_ψ, as well as the iterative optimization process are similar to those of the first experiment. The training takes about 10 minutes.

For evaluation of training, we first illustrate the comparison between maximal drawdown distribution of the three dimensions. The sizes of the synthesized data set and the simulated data set used for comparison are both 5,000. The results are illustrated in the following Figures 10, 11, and 12.

We also investigate the correlation matrix of the three dimensions of data at the 25th observation. By generating 5,000 sequences using the trained networks μ_θ and Σ_ϕ each time to estimate the correlation matrix, and replicating 100 times to derive the mean value and standard deviation of

Table 5. Simulated Correlation of the Three Observed Dimensions at Time Step $i = 15$, Polynomial Model

	Est. true value	Sim. mean	Std. dev.
$\text{cor}(S_1, S_2)$	0.31	0.29	0.10
$\text{cor}(S_1, S_3)$	−0.24	−0.12	0.14
$\text{cor}(S_2, S_3)$	0.62	0.56	0.15

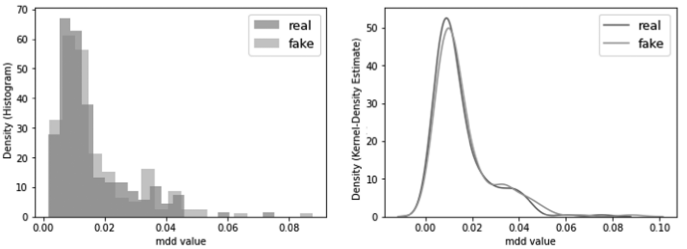


Fig. 13. Maximal draw down distribution, real stock price.

simulated estimations, we have the following results in Table 5. Note that the true value is now estimated from real (synthetic) data.

4.5 Stock Price

4.5.1 The Real Data Set. In this subsection, we use a real data set from a data vendor from a platform *Wind* to test the performance of our estimated simulator. The data set consists of the price variations of a stock (Facebook) from Oct. 8th, 2020 to Mar. 22nd, 2021. The observation frequency is 15 minutes, and 26 data points $(S_i : i = 0, 1, 2, \ldots, 25)$ are recorded for every transaction day. The empirical data is processed as follows. Since stock returns are usually stationary while prices are not, we take logarithm of the data points $(S_i : i = 0, 1, 2, \ldots, 25)$, and derive the log return sequence $(R_i : i = 1, 2, \ldots, 25)$, where $R_i = \log S_i - \log S_{i-1}$. With such transformation, all log return sequences can be regarded as weakly correlated identical copies of an underlying real distribution. After removing the sequences with missing data, we retain $n = 186$ such copies.

4.5.2 Training Process and Results. We specify the structure of the simulator as

$$X_{k+1} = X_k + \mu_\theta(X_k)\Delta t + \Sigma_\phi(X_k)\sqrt{\Delta t}\eta_{k+1} \tag{43}$$

where $X_k = (R_k, Y_k)^\top$, $\eta_k : k = 1, 2, \ldots$ are independent 2-dimensional canonical normal variables. The latent volatility process Y_k is assumed to be 1-dimensional. We first simulate a sequence of $\log \hat{S}_i$ using the sequential simulator, and derive the log return sequence as part of the input of the discriminator. Thus, the discriminator compares distributions of log returns. The resolution is set as the daily frequency with $\Delta t = 1/252$ year. The parameterization of μ_θ is given by $L = 2$, $\tilde{n} = (n_1, n_2) = (50, 2)$. The parameterization of Σ_ϕ is given by $L = 2$, $\tilde{n} = (n_1, n_2) = (80, 4)$. The parameterization of f_ψ is given by $L = 3$, $\tilde{n} = (n_1, n_2, n_3) = (200, 200, 1)$. The neural network initialization of μ_θ, Σ_ϕ and f_ψ, as well as the iterative optimization process are similar to those of the first experiment. The training takes about 1.5 minutes.

We next evaluate the training results. Since the stock price is 1-dimensional, we take itself as the portfolio, i.e., $H_t = S_t$. The comparison between the distribution of the maximal drawdown of real data-based H_t and that of the simulated data-based $\hat{H}_t$ is illustrated in the following Figure 13. The sizes of the real data set and the simulated data set used for comparison are both 186.

5 CONCLUSION

We propose a new framework of a sequential-structured simulator assisted by neural networks and Wasserstein training to model, estimate, and simulate a wide class of sequentially generated data. Neural networks are integrated into the sequentially structured simulators to capture potential nonlinear and complicated sequential structures. Given representative real data, the neural network parameters in the simulator are estimated and calibrated through a Wasserstein training process, which matches the joint distribution of the simulated data and real data in terms of Wasserstein distance. Moreover, the neural network-assisted sequential structured simulator can flexibly incorporate various kinds of elementary randomness and generate distributions with certain properties such as heavy-tail, without the need to redesign the estimation and training procedures. Further, regarding statistical properties, we provide results on consistency and convergence rate for the estimation procedure of the proposed simulator, which are the first set of results that allow the training data samples to be correlated.

ACKNOWLEDGMENTS

The authors are thankful to the anonymous reviewers and editors for their very helpful comments and suggestions, which have significantly benefited this work. The authors thank the participants and organizers of 2021 INFORMS Simulation Society Workshop (I-Sim) for helpful comments and feedback. A preliminary conference version of this work, [33], has appeared in the Proceedings of the Winter Simulation Conference 2021. The theory and analysis in this manuscript are new, compared to the conference version.

APPENDICES

A BACKPROPOGATION DIAGRAM ON GRADIENT EVALUATION

The backpropagation diagram (red dashed line) on the gradient evaluation of $f_\psi(\hat{S}(\theta,\phi))$ with respect to the parameters θ,ϕ in the simulator neural network μ_θ,Σ_ϕ is shown in Figure A.1.

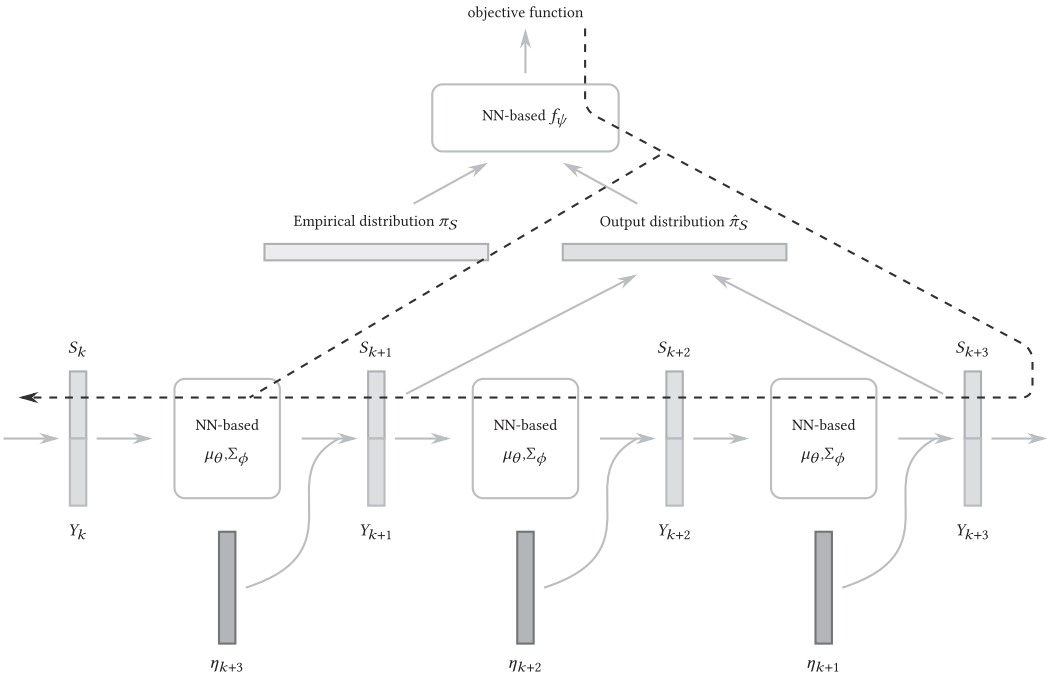


Fig. A.1. Backpropagation diagram (red dashed line) on the gradient evaluation of $f_\psi(\hat{S}(\theta,\phi))$ with respect to the parameters θ,ϕ in the simulator neural network μ_θ,Σ_ϕ.

B PROOF OF THEOREM 3.4

In the following subsections, we present a proof of Theorem 3.4 consisting of two main steps: Approximating Lipschitz functions with interpolation polynomials and approximating the polynomials with neural networks.

B.1 Polynomial Approximation of Lipschitz Functions

We first perform a linear transformation on f, namely, let

$$\Psi : [-B,B]^d \to [0,1]^d \quad \Psi(\boldsymbol{z}) = \frac{1}{2B}(\boldsymbol{z} + B \cdot \mathbf{1}). \tag{44}$$

Let $f^\Psi = f \circ \Psi^{-1}$, we have

$$f^\Psi \in [0,1]^d, \quad \|f^\Psi(\boldsymbol{x}) - f^\Psi(\boldsymbol{y})\| \le 2BC_L\|\boldsymbol{x}-\boldsymbol{y}\|. \tag{45}$$

Without loss of generality, suppose that $\|f^\Psi\|_{L^\infty([0,1]^d)} \le BC_L$. Note that if the $2BC_L$-Lipschitz function f^Ψ can be approximated by a neural network Φ in the sense that $\|\Phi(\boldsymbol{x}) - f^\Psi(\boldsymbol{x})\| \le \delta, \forall \boldsymbol{x} \in [0,1]^d$, we have

- $\Phi \circ \Psi$ can also be expressed in the form of a neural network, where $\mathcal{L}(\Phi \circ \Psi) = \mathcal{L}(\Phi)$
- $\|\Phi \circ \Psi(\boldsymbol{x}) - f^\Psi \circ \Psi(\boldsymbol{x})\| = \|\Phi \circ \Psi(\boldsymbol{x}) - f(\boldsymbol{x})\| \le \delta, \forall \boldsymbol{x} \in [-B,B]^d$

The next step is to approximate an arbitrary $2BC_L$-Lipschitz function f^Ψ on $[0,1]^d$ with a explicitly constructed interpolation polynomial. Our construction and proof are a simplified version of those in [11].

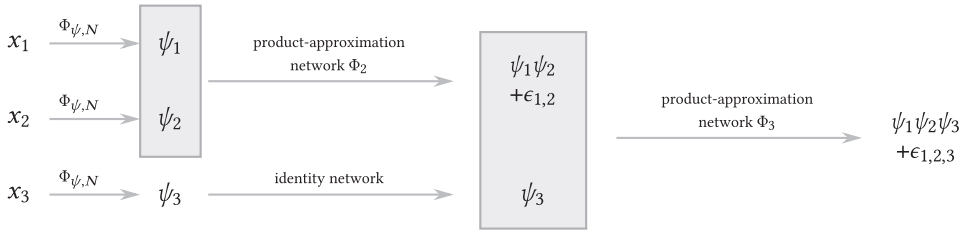


Fig. B.1. Construction of $\Phi_{\zeta_{\boldsymbol{m}}}$.

Define the trapezoid function

$$\psi(x) = \begin{cases} 1 & |x| < 1, \\ 2 - |x| & 1 \le |x| \le 2, \\ 0 & |x| > 2, \end{cases} \qquad x \in \mathbb{R}, \tag{46}$$

and let

$$\zeta_{N,\boldsymbol{m}}(\boldsymbol{x}) = \prod_{k=1}^{d} \psi\left(3N(x_k - \frac{m_k}{N})\right) := \prod_{k=1}^{d} \psi_N(x_k), \tag{47}$$

where $\boldsymbol{m} = (m_1, m_2, \ldots, m_d)$, $m_i \in \{0, 1, \ldots, N\}$. Note that $0 \le \zeta_{N,\boldsymbol{m}}(\boldsymbol{x}) \le 1$ and $\sum_{\boldsymbol{m}} \zeta_{N,\boldsymbol{m}}(\boldsymbol{x}) = 1$, $\forall \boldsymbol{x} \in [0,1]^d$. Further, let

$$P_{N,\boldsymbol{m}} = f^{\Psi}\left(\frac{1}{N} \cdot \boldsymbol{m}\right), \quad \bar{f}_N = \sum_{\boldsymbol{m}} P_{N,\boldsymbol{m}} \zeta_{N,\boldsymbol{m}}(\boldsymbol{x}). \tag{48}$$

We show that $\|\bar{f}_N - f^{\Psi}\|_\infty$ can be bounded as follows:

$$\begin{aligned} \max_{\boldsymbol{x}\in[0,1]^d} |\bar{f}_N(\boldsymbol{x}) - f^{\Psi}(\boldsymbol{x})| &= \max_{\boldsymbol{x}\in[0,1]^d} \left| \sum_{\boldsymbol{m}} \zeta_{N,\boldsymbol{m}}(\boldsymbol{x})(P_{N,\boldsymbol{m}} - f^{\Psi}(\boldsymbol{x})) \right| \\ &\overset{(i)}{\le} \max_{\boldsymbol{x}\in[0,1]^d} \sum_{\boldsymbol{m}:\left|x_k - \frac{m_k}{N}\right| < \frac{1}{N}} \left| P_{N,\boldsymbol{m}} - f^{\Psi}(\boldsymbol{x}) \right| \\ &\le \max_{\boldsymbol{x}\in[0,1]^d} 2^d \max_{\boldsymbol{m}:\left|x_k - \frac{m_k}{N}\right| < \frac{1}{N}} \left| f^{\Psi}\left(\frac{1}{N} \cdot \boldsymbol{m}\right) - f^{\Psi}(\boldsymbol{x}) \right| \\ &\le \max_{\boldsymbol{x}\in[0,1]^d} 2^d \max_{\boldsymbol{m}:\left|x_k - \frac{m_k}{N}\right| < \frac{1}{N}} 2BC_L \left\| \frac{1}{N} \cdot \boldsymbol{m} - \boldsymbol{x} \right\| \\ &\le \frac{2^{d+1}\sqrt{d}BC_L}{N}. \end{aligned} \tag{49}$$

Note that

$$\zeta_{N,\boldsymbol{m}}(\boldsymbol{x}) = 0 \quad \text{if } \exists k \in \{1, 2, \ldots, d\}, \text{ s.t. } \left| x_k - \frac{m_k}{N} \right| \ge \frac{1}{N}, \tag{50}$$

which is due to the definition (47) of $\zeta_{N,\boldsymbol{m}}(\boldsymbol{x})$ and is the reason for (i). Therefore, for $N \ge 2^{d+2}\sqrt{d}BC_L/\delta$, we have $\|\bar{f}_N - f^{\Psi}\|_\infty \le \delta/2$.

B.2 Neural Network Approximation of Polynomials

In this section, we use neural networks to approximate $\bar{f}_N$ constructed in Section B.1. We first introduce the following Theorem B.1 which constructs a neural network to approximate multiplication of two constants, i.e., $\Phi(x, y) \approx xy$.

Theorem B.1 (Neural Network Approximation of the Product Operator, Proposition 3.3 of [15]). *There exists a constant $C > 0$ such that, for all $D \in \mathbb{R}^+$ and $\epsilon \in (0, 1/2)$, there is a network $\Phi \in \mathcal{N}_{2,1}$, with $\mathcal{L}(\Phi) \le C(\log(\lceil D \rceil) + \log(\epsilon^{-1}))$, $\mathcal{W}(\Phi) \le 5$, $\mathcal{B}(\Phi) \le 1$, $\mathcal{M}(\Phi) = O(\mathcal{L}(\Phi))$, $\Phi(0, x) = \Phi(x, 0) = 0$, for all $x \in \mathbb{R}$, and*

$$\|\Phi(x, y) - xy\|_{L^\infty([D,D])} \le \epsilon. \tag{51}$$

Now, using Theorem B.1 as a building block, we approximate $\zeta_{N,\boldsymbol{m}}(\boldsymbol{x})$ defined in Section B.1, which is the product of $\psi_N(x_k) : k = 1, 2, \ldots, d$. One useful way to analyze such approximation is to view the neural network as not just a composition of linear and activation functions, but also a combination of layers with operational connections between the elements of every two adjacent layers.

First, the mapping $x_k \to \psi_N(x_k)$ can be expressed by a neural network. Consider the hat function $h : \mathbb{R} \to [0, 1]$,

$$h(x) = \begin{cases} 2x, & \text{if } 0 \le x \le \frac{1}{2}, \\ 2(1-x), & \text{if } \frac{1}{2} \le x \le 1, \\ 0, & \text{else,} \end{cases} \tag{52}$$

we have

(1) According to [15], $h(x)$ can be expressed by a neural network $\Phi_h(x)$, where $\mathcal{L}(\Phi_h) = 2$, $\mathcal{W}(\Phi_h) = 3$, $\mathcal{B}(\Phi_h) = 4$ and $\mathcal{M}(\Phi_h) = 8$. Also note that the 2-Layer network is given as $W_1 \circ \sigma \circ W_2$, and if a ReLU activation is not involved in the middle, any composition of linear transformations can be compressed into a single layer.
(2) $\psi(x)$ given as (46) can be expressed as follows:

$$\psi(x) = h\left(\frac{1}{2}x\right) + h\left(\frac{1}{2}(x+1)\right) + h\left(\frac{1}{2}(x+2)\right). \tag{53}$$

Therefore, $\psi_N(x_k)$ can be expressed by a neural network $\Phi_{\psi,N}(x_k) \in \mathcal{N}_{1,1}$, where $\mathcal{L}(\Phi)_{\psi,N} = 2$, $\mathcal{W}(\Phi_{\psi,N}) = 9$, and $\mathcal{B}(\Phi_{\psi,N}) = O(N)$. We denote by $\mathcal{N}_{d_1,d_2}$ the set of all ReLU networks with input dimension d_1 and output dimension d_2. Note that N will be balanced with ϵ and δ later on.

The next step is to iteratively approximate the product $\prod_{k=1}^d \psi_k$ with neural network approximators of multiplication. Observe that $\psi_k, k = 1, 2, \ldots, d$ and their products are all bounded by $[0, 1]$. Specifically, by Theorem B.1, we have a network $\Phi_2 \in \mathcal{N}_{2,1}$, with $\mathcal{L}(\Phi_2) = O(\log(d\epsilon^{-1}))$, $\mathcal{W}(\Phi_2) \le 5$, $\mathcal{B}(\Phi_2) \le 1$ and $\mathcal{M}(\Phi_2) = O(\mathcal{L}(\Phi_2))$ satisfying

$$\|\Phi_2(\psi_1, \psi_2) - \psi_1\psi_2\|_{L^\infty([0,1])} \le \frac{\epsilon}{d}. \tag{54}$$

Iteratively, we have a network $\Phi_3 \in \mathcal{N}_{2,1}$, with $\mathcal{L}(\Phi_3) = O(\log(d\epsilon^{-1}))$, $\mathcal{W}(\Phi_3) \le 5$, $\mathcal{B}(\Phi_3) \le 1$ and $\mathcal{M}(\Phi_3) = O(\mathcal{L}(\Phi_3))$ satisfying

$$\begin{aligned} &\|\Phi_3(\Phi_2(\psi_1, \psi_2), \phi_3) - \psi_1\psi_2\psi_3\|_{L^\infty([0,1])} \\ \le &\|\Phi_3(\Phi_2(\psi_1, \psi_2), \psi_3) - \Psi_2(\psi_1, \psi_2)\psi_3\|_{L^\infty([0,1])} + \|\Phi_2(\psi_1, \psi_2)\psi_3 - \psi_1\psi_2\psi_3\|_{L^\infty([0,1])} \le \frac{2\epsilon}{d} + O(\epsilon^2). \end{aligned} \tag{55}$$

In total, we have a network $\Phi_{\zeta_{N,\boldsymbol{m}}} \in \mathcal{N}_{d,1}$ composite of $\Phi_i, i = 2, 3, \ldots d$ and $\Phi_{\psi,N}$s, with $\mathcal{L}(\Phi_{\zeta_{N,\boldsymbol{m}}}) = O(d \log(d\epsilon^{-1}))$, $\mathcal{W}(\Phi_{\zeta_{N,\boldsymbol{m}}}) = O(d)$, $\mathcal{B}(\Phi_{\zeta_{N,\boldsymbol{m}}}) = O(N)$ and $\mathcal{M}(\Phi_{\zeta_{N,\boldsymbol{m}}}) = O(\mathcal{L}(\Phi_{\zeta_{N,\boldsymbol{m}}})$ $\mathcal{W}(\Phi_{\zeta_{N,\boldsymbol{m}}}))$ where

$$\|\Phi_{\zeta_{N,\boldsymbol{m}}}(\boldsymbol{x}) - \zeta_{N,\boldsymbol{m}}(\boldsymbol{x})\|_{L^\infty([0,1]^d)} \le 2\epsilon. \tag{56}$$

The following Figure B.1 illustrates how $\Phi_{\zeta_{N,\boldsymbol{m}}}$ is constructed through combining and paralleling small networks.

Finally, the network $\Phi_{\bar{f}_N} = \sum_{\boldsymbol{m}} P_{N,\boldsymbol{m}} \Phi_{\zeta_{N,\boldsymbol{m}}}$ with $\mathcal{L}(\Phi_{\bar{f}_N}) = O(d\log(d\epsilon^{-1}))$, $\mathcal{W}(\Phi_{\bar{f}_N}) = O(d(N+1)^d)$, $\mathcal{B}(\Phi_{\bar{f}_N}) = O(N + BC_L)$ and $\mathcal{M}(\Phi_{\bar{f}_N}) + O(\mathcal{L}(\Phi_{\bar{f}})\mathcal{W}(\Phi_{\bar{f}_N}))$ satisfies

$$\|\Phi_{\bar{f}_N}(\boldsymbol{x}) - \bar{f}_N(\boldsymbol{x})\|_{L^\infty([0,1]^d)} \leq 2^{d+1} BC_L \epsilon. \tag{57}$$

Note that $\forall \boldsymbol{x} \in [0,1]^d$, only 2^d out of the $(N+1)^d$ elements of $\{\zeta_{N,\boldsymbol{m}}\}, \forall \boldsymbol{m}$ are non-zero, and according to Theorem B.1, the approximation error is exactly 0 when zero elements are contained in the multipliers. This explains the term 2^{d+1} on R.H.S. of (57). The term BC_L on R.H.S. of (57) comes from the fact that $|P_{N,\boldsymbol{m}}| = |f^\Psi(\frac{1}{N} \cdot M)| \leq BC_L$.

B.3 Balance and Conclusion

To have $\|\bar{f}_N - f^\Psi\|_\infty$ for Section B.1 and $\|\Phi_{\bar{f}_N}(\boldsymbol{x}) - \bar{f}_N(\boldsymbol{x})\|_{L^\infty([0,1]^d)} \leq 2^{d+1} BC_L \epsilon \leq \frac{\delta}{2}$ for Section B.2, let

$$N = 2^{d+2}\sqrt{d} BC_L \frac{1}{\delta}, \quad \epsilon = \frac{\delta}{2^{d+2} BC_L}, \tag{58}$$

we can replace the orders of the network approximator Φ_f with $\mathcal{L}(\Phi_f) = O(\log B + \log \delta^{-1})$, $\mathcal{W}(\Phi_f) = O(B^d \delta^{-d})$, $\mathcal{B}(\Phi_f) = O(B\delta^{-1})$ and $\mathcal{M}(\Phi_f) = O((\log B + \log \delta^{-1}) \cdot B^d \delta^{-d})$, and Φ_f satisfies

$$\|\Phi_f(\boldsymbol{x}) - f(\boldsymbol{x})\|_{L^\infty([-B,B]^d)} \leq \|\bar{f}_N - f^\Psi\|_\infty + \|\Phi_{\bar{f}_N}(\boldsymbol{x}) - \bar{f}_N(\boldsymbol{x})\|_{L^\infty([0,1]^d)} \leq \delta. \tag{59}$$

C CONTROLLING BOUNDING ERROR AND GENERATOR APPROXIMATION ERROR

C.1 Bounding Error

In this section, we control the bounding error term $W(\pi, \pi^B)$. With the assumption that

$$P_{\eta_k}(x) \leq O(x^{-(\alpha+2)}), \quad \alpha > 0, \tag{60}$$

the bounding error term can be bounded as follows:

$$W(\pi, \pi^B) \leq W(\pi, \pi^B) = \inf_{\gamma \in \Gamma(\pi^B, \pi)} \mathbb{E}_\gamma \left(\|\mathbf{X} - \mathbf{Y}\|\right) \leq \mathbb{E}\left(\|\mathbf{X} - \mathbf{X}^B\|\right) \leq O(B^{-\alpha}). \tag{61}$$

C.2 Generator Approximation Error

In this section, we control the generator approximation error $W(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi))$. Note that

$$W(\pi^B, \hat{\pi}^B(\mu_\theta, \Sigma_\phi)) \leq \underbrace{W(\hat{\pi}^B(\mu, \Sigma), \hat{\pi}^B(\mu_\theta, \Sigma_\phi))}_{\text{identically bounded generator approximation error}} + \underbrace{W(\pi, \hat{\pi}^B(\mu, \Sigma)) + W(\pi, \pi^B)}_{\text{bounding error}}. \tag{62}$$

As in Section C.1, the bounding error can be controlled by

$$W(\pi, \hat{\pi}^B(\mu, \Sigma)) + W(\pi, \pi^B) \leq O(B^{-\alpha}). \tag{63}$$

Next, denote the network approximation error of μ and Σ on $[-B, B]^d$ as ϵ_μ and ϵ_Σ, respectively. We have

$$\begin{aligned} W(\hat{\pi}^B(\mu, \Sigma), \hat{\pi}^B(\mu_\theta, \Sigma_\phi)) &= \inf_{\gamma \in S(\hat{\pi}^B(\mu_\theta, \Sigma_\phi), \hat{\pi}^B(\mu, \Sigma))} \mathbb{E}_{(\hat{\mathbf{X}}^B, \hat{\mathbf{Y}}^B) \sim \gamma} \left[\|\hat{\mathbf{X}}^B - \hat{\mathbf{Y}}^B\|\right] \\ &\leq \mathbb{E}_{(\hat{\mathbf{X}}^B, \hat{\mathbf{Y}}^B) \sim \gamma_e} \left[\|\hat{\mathbf{X}}^B - \hat{\mathbf{Y}}^B\|\right], \end{aligned} \tag{64}$$

where, denoting the elementary randomness of the simulator of $\hat{\mathbf{X}}^B$ and $\hat{\mathbf{Y}}^B$ as $\{\xi_i\}_{i=1}^p$ and $\{\eta_i\}_{i=1}^p$ respectively, we have

$$\mathbb{E}_{\gamma_e} \left[\|\hat{\mathbf{X}}^B - \hat{\mathbf{Y}}^B\|\right] := \mathbb{E}\left[\|\hat{\mathbf{X}}^B - \hat{\mathbf{Y}}^B\| \middle| \xi_i \equiv \eta_i, \forall i\right]. \tag{65}$$

Since

$$\mathbb{E}_{\gamma_e}\left[\|\hat{\mathbf{X}}^B - \hat{\mathbf{Y}}^B\|\right] \le \sum_{k=1}^{p} \mathbb{E}_{\gamma_e}\left[\|\hat{X}_i^B - \hat{Y}_i^B\|\right] := \sum_{i=1}^{p} \mathbb{E}_{\gamma_e}\Delta_i, \tag{66}$$

and by law of total expectation, we have

$$\mathbb{E}_{\gamma_e}\Delta_i = \mathbb{E}_{\gamma_e}\left[\mathbb{E}_{\gamma_e}(\Delta_i|\Delta_{i-1})\right], \tag{67}$$

further,

$$\begin{aligned}
&\mathbb{E}_{\gamma_e}\left[\Delta_i \middle| \Delta_{i-1}, \hat{X}_{i-1}^B, \hat{Y}_{i-1}^B\right] \\
\le&\mathbb{E}_{\gamma_e}\left[\Delta_{i-1} + |\mu(\hat{Y}_{i-1}^B) - \mu_\theta(\hat{X}_{i-1}^B)| + \|\Sigma(\hat{Y}_{i-1}^B) - \Sigma_\phi(\hat{X}_{i-1}^B)\eta_i\| \middle| \Delta_{i-1}, \hat{X}_{i-1}^B, \hat{Y}_{i-1}^B\right] \\
\le&\mathbb{E}_{\gamma_e}\left[\Delta_{i-1} + (\epsilon_\mu + \Delta_{i-1}) + (\epsilon_\Sigma + \Delta_{i-1})\|\eta_i\| \middle| \Delta_{i-1}\right] \\
\le&(2+d)\Delta_{i-1} + (\epsilon_\mu + d\epsilon_\Sigma).
\end{aligned} \tag{68}$$

Therefore,

$$\mathbb{E}_{\gamma_e}\Delta_i \le \frac{(2+d)^i - 1}{1+d}(d\epsilon_\Sigma + \epsilon_\mu), \tag{69}$$

and

$$\sum_{i=1}^{p} \mathbb{E}_{\gamma_e}\Delta_i \le \left(\frac{1}{1+d}\sum_{i=1}^{p}(2+d)^i\right)\cdot(d\epsilon_\Sigma + \epsilon_\mu) = O(\epsilon_\Sigma + \epsilon_\mu). \tag{70}$$

D PROOF OF LEMMA D.1

In this section, we first prove that the two function classes $\mathcal{F}_{NN}(\kappa, L, P, K, \epsilon_f)$ and $\mathcal{F}_{\text{Lip}}$ can approximate each other, namely,

(1) $\forall f \in \mathcal{F}_{\text{Lip}}, \exists f_\psi \in \mathcal{F}_{NN}$, such that $\|f - f_\psi\|_{L^\infty[-B,B]^d} \le \epsilon_f$,
(2) $\forall f \in \mathcal{F}_{NN}, \exists f_\psi \in \mathcal{F}_{\text{Lip}}$, such that $\|f - f_\psi\|_{L^\infty[-B,B]^d} \le 3\epsilon_f$.

For 2, we have the following lemma:

Lemma D.1. *Suppose that $f : \mathbb{R}^d \to \mathbb{R}$, $f \in C([a,b])$ satisfies $|f(x) - f(y)| \le \|x - y\| + 2\epsilon$, $\forall x, y \in [a,b]$. Then there is a 1-Lipschitz function $g : \mathbb{R}^d \to \mathbb{R}$, $g \in C([a,b])$, such that*

$$|f(x) - g(x)| \le 3\epsilon, \quad \forall x \in [a,b]. \tag{71}$$

Proof: Without loss of generality, we assume that $d = 1, a = 0, b = 1$ and $f(0) = 0$. We prove by contradiction. Let

$$x(g) := \sup\{x \in [0,1] : |f(x') - g(x')| \le 2\epsilon, \forall x' < x\}, \tag{72}$$

and

$$g^* = \arg\sup_{\|g\|_L \le 1} x(g). \tag{73}$$

The contradiction assumption suggests that $x^* := x(g^*) < 1$. We first show that $x^* > 0$ and that g^* exists. Note that for $x' < \epsilon$, we have, by definition of the functional class $\mathcal{F}_{\text{NN}}(\kappa, L, P, K, \epsilon)$,

$$|f(x') - f(0)| \le |x'| + \epsilon < 3\epsilon.$$

Therefore, taking $g(x) \equiv f(0)$ yields an approximation of f with precision of 3ϵ on $[0,\epsilon]$, implying that x^* is at least ϵ. To demonstrate the existence of g^*, suppose that $|f(x)| < C$ on $[0,1]$ for some constant C. The 1-Lipschitz functional class bounded by $2C$ on $[0,1]$, denoted as $\mathcal{F}_{\text{Lip}}^C$ is uniformly bounded and equicontinuous. Arzel*à*-Ascoli theorem suggests that $\mathcal{F}_{\text{Lip}}^C$ is sequentially compact.

Further, for any convergent sequence in $\mathcal{F}_{\text{Lip}}^C$, it can be verified that the limit also lies in $\mathcal{F}_{\text{Lip}}^C$. Therefore, $\mathcal{F}_{\text{Lip}}^C$ is a compact set. It remains to be proved that $x(g)$ is upper semi-continuous on $\mathcal{F}_{\text{Lip}}$, so that the supremum point g^* exists. In fact, for a fixed element $g_0 \in \mathcal{F}_{\text{Lip}}$, for all small enough $\epsilon > 0$, let

$$\delta = \sup_{x(g_0) \le x \le x(g_0)+\epsilon} \left[|f(x) - g_0(x)| - 3\epsilon \right].$$

By definition of $x(g)$, we have $\delta > 0$, and for all $g \in \mathcal{F}_{\text{Lip}}$ such that $\|g - g_0\|_{L^\infty[0,B]^d} < \delta$, we have $x(g) < x(g_0) + \epsilon$. The existence of g^* is proved.

We next return to the contradiction assumption, which suggests that $x^* < 1$. Note that both f and g are continuous. Therefore, by the definition of sup, we have

$$|g^*(x^*) - f(x^*)| = 3\epsilon. \tag{74}$$

Without loss of generality, let $f(x^*) = g^*(x^*) + 3\epsilon$. Also,

$$\forall \Delta > 0, \exists x^* < x_\Delta < \min\{x^* + \delta, 1\}, \text{ s.t. } f(x_\Delta) > g^*(x^*) + (x_\Delta - x^*) + 3\epsilon. \tag{75}$$

Now, we claim that $\exists \delta_0 > 0$,

$$\frac{g^*(x^*) - g^*(x)}{x^* - x} = 1, \quad \forall x \in [x^* - \delta_0, x^*).$$

If this is contradicted, we have some $\delta < \epsilon$, and

$$g^*(x^*) - \delta < g^*(x^* - \delta) < g^*(x^*) + 3\epsilon - \delta.$$

In this case, we can move g^* upward on $[x^* - \delta, x^*]$ by modifying it into

$$\tilde{g}^*(x) = \begin{cases} g^*(x), & x \in [0, x^* - \delta]; \\ g^*(x^* - \delta) + x - (x^* - \delta), & x \in (x^* - \delta, x^*], \end{cases}$$

so that $\tilde{g}^*(x^*) > g^*(x^*)$, and $\|\tilde{g}^*(x^*)\|_L \le 1$ still holds. Also, note that $\forall x \in [0,1]$,

$$\begin{aligned} f(x) &\ge f(x^*) + x - x^* - 3\epsilon = g^*(x^*) + x - x^* > \tilde{g}^*(x) - 3\epsilon, \\ f(x) &\le g^*(x) + 3\epsilon \le \tilde{g}^*(x) + 3\epsilon_f. \end{aligned}$$

so $x(\tilde{g}^*) > x(g^*)$, which contradicts (73). Therefore, the claim is valid.

Let

$$x_1 = \inf_{x \in [0, x^* - \delta_0]} \frac{g^*(x^*) - g^*(x)}{x^* - x} = 1$$

Note that $f(x_\Delta) > g^*(x_1) + (x_\Delta - x_1) + 3\epsilon$, therefore, $f(x_1) > g^*(x_1)$, which implies that $x_1 > 0$.

In this case, we can find $\delta' > 0$, such that

$$g^*(x_1) - \delta' \le g^*(x_1 - \delta') \le g^*(x_1) + 3\epsilon - \delta'.$$

We can similarly define

$$\tilde{g}^*(x) = \begin{cases} g^*(x), & x \in [0, x_1 - \delta']; \\ g^*(x_1 - \delta') + x - (x_1 - \delta'), & x \in (x_1 - \delta', x^*], \end{cases}$$

and verify that $\|\tilde{g}^*(x^*)\|_L \le 1$ and $x(\tilde{g}^*) > x(g^*)$, which also contradicts (73). The proof is complete.

REFERENCES

[1] Yacine Aı, Robert Kimmel, et al. 2007. Maximum likelihood estimation of stochastic volatility models. *Journal of Financial Economics* 83, 2 (2007), 413–452.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein generative adversarial networks. In *International Conference on Machine Learning*. PMLR, 214–223.

[3] Sanjeev Arora, Rong Ge, Yingyu Liang, Tengyu Ma, and Yi Zhang. 2017. Generalization and equilibrium in generative adversarial nets (GANs). In *International Conference on Machine Learning*. PMLR, 224–232.

[4] Manabu Asai, Michael McAleer, and Jun Yu. 2006. Multivariate stochastic volatility: A review. *Econometric Reviews* 25, 2-3 (2006), 145–175.

[5] Yu Bai, Tengyu Ma, and Andrej Risteski. 2018. Approximability of discriminators implies diversity in GANs. *arXiv preprint arXiv:1806.10586* (2018).

[6] Ravi Bansal, A. Ronald Gallant, Robert Hussey, and George Tauchen. 1994. Computational aspects of nonparametric simulation estimation. In *Computational Techniques for Econometrics and Economic Analysis*. Springer, 3–22.

[7] Eoin Brophy, Zhengwei Wang, Qi She, and Tomas Ward. 2021. Generative adversarial networks in time series: A survey and taxonomy. *arXiv preprint arXiv:2107.11098* (2021).

[8] Carmen Broto and Esther Ruiz. 2004. Estimation methods for stochastic volatility models: A survey. *Journal of Economic Surveys* 18, 5 (2004), 613–649.

[9] Wang Cen, Emily A. Herbert, and Peter J. Haas. 2020. NIM: Modeling and generation of simulation inputs via generative neural networks. In *Proceedings of the 2020 Winter Simulation Conference,* Bae, K., Feng, B., Kim, S., Lazarova-Molnar, S., Zheng, Z., Roeder, T., and Thiesing, R. (Ed.). Institute of Electrical and Electronic Engineers, Inc., Piscataway, New Jersey, 584–595.

[10] Alexei Chekhlov, Stanislav Uryasev, and Michael Zabarankin. 2005. Drawdown measure in portfolio optimization. *International Journal of Theoretical and Applied Finance* 8, 01 (2005), 13–58.

[11] Minshuo Chen, Haoming Jiang, Wenjing Liao, and Tuo Zhao. 2022. Nonparametric regression on low-dimensional manifolds using deep ReLU networks: Function approximation and statistical recovery. *Information and Inference: A Journal of the IMA* 11, 4 (2022), 1203–1253.

[12] Minshuo Chen, Wenjing Liao, Hongyuan Zha, and Tuo Zhao. 2020. Statistical guarantees of generative adversarial networks for distribution estimation. *arXiv preprint arXiv:2002.03938* (2020).

[13] Marco Cuturi. 2013. Sinkhorn distances: Lightspeed computation of optimal transport. *Advances in Neural Information Processing Systems* 26 (2013), 2292–2300.

[14] Florian Eckerli. 2021. Generative adversarial networks in finance: An overview. *Available at SSRN 3864965* (2021).

[15] Dennis Elbrächter, Dmytro Perekrestenko, Philipp Grohs, and Helmut Bölcskei. 2021. Deep neural network approximation theory. *IEEE Transactions on Information Theory* 67, 5 (2021), 2581–2623.

[16] Cristóbal Esteban, Stephanie L. Hyland, and Gunnar Rätsch. 2017. Real-valued (medical) time series generation with recurrent conditional GANs. *arXiv preprint arXiv:1706.02633* (2017).

[17] Marco Fraccaro, Søren Kaae Sønderby, Ulrich Paquet, and Ole Winther. 2016. Sequential neural models with stochastic layers. *Advances in Neural Information Processing Systems* 29 (2016).

[18] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[19] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C. Courville. 2017. Improved training of Wasserstein GANs. *Advances in Neural Information Processing Systems* 30 (2017).

[20] Linyun He and Eunhye Song. 2021. Nonparametric Kullback-Liebler divergence estimation using M-spacing. In *2021 Winter Simulation Conference (WSC)*. IEEE.

[21] Steven L. Heston. 1993. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *The Review of Financial Studies* 6, 2 (1993), 327–343.

[22] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[23] Diederik P. Kingma and Max Welling. 2013. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114* (2013).

[24] Rui Luo, Weinan Zhang, Xiaojun Xu, and Jun Wang. 2018. A neural stochastic volatility model. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[25] Angelo Melino and Stuart M. Turnbull. 1990. Pricing foreign currency options with stochastic volatility. *Journal of Econometrics* 45, 1-2 (1990), 239–265.

[26] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. *Foundations of Machine Learning*. MIT Press.

[27] Eckhard Platen and Nicola Bruti-Liberati. 2010. *Numerical Solution of Stochastic Differential Equations with Jumps in Finance*. Vol. 64. Springer Science & Business Media.

[28] Neil Shephard. 1993. Fitting nonlinear time-series models with applications to stochastic variance models. *Journal of Applied Econometrics* 8, S1 (1993), S135–S152.

[29] Neil Shephard and Torben G. Andersen. 2009. Stochastic volatility: Origins and overview. In *Handbook of Financial Time Series*. Springer, 233–254.
[30] Umut Şimşekli. 2017. Fractional Langevin Monte Carlo: Exploring Lévy driven stochastic differential equations for Markov chain Monte Carlo. In *International Conference on Machine Learning*. PMLR, 3200–3209.
[31] Shuntaro Takahashi, Yu Chen, and Kumiko Tanaka-Ishii. 2019. Modeling financial time-series with generative adversarial networks. *Physica A: Statistical Mechanics and its Applications* 527 (2019), 121261.
[32] Stephen John Taylor. 1982. Financial returns modelled by the product of two stochastic processes-a study of the daily sugar prices 1961-75. *Time Series Analysis: Theory and Practice* 1 (1982), 203–226.
[33] Zhu Tingyu and Zheng Zeyu. 2021. Learning to simulate sequentially generated data via neural networks and Wasserstein training. In *2021 Winter Simulation Conference (WSC)*. IEEE.
[34] Tan Wan and L. Jeff Hong. 2022. Large-scale inventory optimization: A recurrent-neural-networks-inspired simulation approach. *INFORMS Journal on Computing* (2022).
[35] Ruixin Wang, Prateek Jaiswal, and Harsha Honnappa. 2020. Estimating stochastic Poisson intensities using deep latent models. In *2020 Winter Simulation Conference (WSC)*. IEEE, 596–607.
[36] Magnus Wiese, Robert Knobloch, Ralf Korn, and Peter Kretschmer. 2020. Quant GANs: Deep generation of financial time series. *Quantitative Finance* 20, 9 (2020), 1419–1440.
[37] Jinsung Yoon, Daniel Jarrett, and Mihaela van der Schaar. 2019. Time-series generative adversarial networks. (2019).
[38] Yufeng Zheng, Zeyu Zheng, and Tingyu Zhu. 2020. A doubly stochastic simulator with applications in arrivals modeling and simulation. *arXiv preprint arXiv:2012.13940* (2020).

Received 14 January 2022; revised 4 January 2023; accepted 10 January 2023

NIM: Generative Neural Networks for Automated Modeling and Generation of Simulation Inputs

WANG CEN and PETER J. HAAS, University of Massachusetts Amherst

Fitting stochastic input-process models to data and then sampling from them are key steps in a simulation study but highly challenging to non-experts. We present Neural Input Modeling (NIM), a Generative Neural Network (GNN) framework that exploits modern data-rich environments to automatically capture simulation input processes and then generate samples from them. The basic GNN that we develop, called *NIM-VL*, comprises (i) a variational autoencoder architecture that learns the probability distribution of the input data while avoiding overfitting and (ii) long short-term memory components that concisely capture statistical dependencies across time. We show how the basic GNN architecture can be modified to exploit known distributional properties—such as independent and identically distributed structure, nonnegativity, and multimodality—to increase accuracy and speed, as well as to handle multivariate processes, categorical-valued processes, and extrapolation beyond the training data for certain nonstationary processes. We also introduce an extension to NIM called *Conditional Neural Input Modeling* (CNIM), which can learn from training data obtained under various realizations of a (possibly time series valued) stochastic "condition," such as temperature or inflation rate, and then generate sample paths given a value of the condition not seen in the training data. This enables users to simulate a system under a specific working condition by customizing a pre-trained model; CNIM also facilitates what-if analysis. Extensive experiments show the efficacy of our approach. NIM can thus help overcome one of the key barriers to simulation for non-experts.

CCS Concepts: • **Computing methodologies → Modeling methodologies;**

Additional Key Words and Phrases: Input modeling, neural networks, distribution fitting

ACM Reference format:
Wang Cen and Peter J. Haas. 2023. NIM: Generative Neural Networks for Automated Modeling and Generation of Simulation Inputs. *ACM Trans. Model. Comput. Simul.* 33, 3, Article 10 (August 2023), 26 pages.
https://doi.org/10.1145/3592790

1 INTRODUCTION

Stochastic discrete-event simulation is a time-honored technology for improving the design and operation of complex engineered systems under uncertainty, but the barriers to entry are high. Traditionally, a domain expert examines the existing system and specifies the simulation model structure (system elements and their interrelations), along with probability distributions for simulation inputs, which are often fitted to empirical data gleaned from the existing system. For example, a

Authors' address: W. Cen and P. J. Haas, University of Massachusetts Amherst, 140 Governors Drive, Amherst, MA 01003; emails: cenwang@umass.edu, phaas@cs.umass.edu.

1049-3301/2023/08-ART10 $15.00
https://doi.org/10.1145/3592790

manufacturing expert might decide that a robot station in an automated manufacturing system can be modeled as a FIFO queue. Based on a small number of observed arrival times for raw parts and observed processing times for the manufacturing robot, the expert will fit probability distributions for these quantities. Finally, the validated simulation model can be used to explore potential improvements to the station via changes in buffer sizes, robot capabilities, job scheduling, and so on. Although modern simulation software tools such as AnyLogic and Arena provide graphical interfaces that can greatly ease the task of specifying the simulation model structure, modeling the simulation inputs remains one of the most challenging tasks for a non-expert. Our goal is to facilitate this process via automation.

Traditional Input Modeling. Traditionally, data for fitting input distributions has been expensive and painful to collect (e.g., a human would have to stand on a factory floor, stopwatch in hand) and hence has been in short supply. With little data available, a modeler typically imposes strong simplifying assumptions, for example, by assuming that the interarrival times to a system are i.i.d. according to one of a set of supported distribution functions from which the simulation engine can efficiently generate samples. The specific distribution function is selected as the one that best fits the observed data, as measured by an appropriate goodness-of-fit statistic; state-of-the-art tools such as ExpertFit [21, Ch. 6] or Stat::Fit [12] can automate this selection task.

Unfortunately, distributions with complex features such as multimodality or phase-type structure [24] are typically hard to capture—for example, one of our experiments in Section 6.1 (see Figure 14) shows that ExpertFit fails to model a Gamma-Uniform mixture distribution. Even for "simple" distributions, handcrafted variate-generation code has traditionally been required. Recently, Jiang and Nelson [18] proposed fitting multiple distributions and averaging them to achieve better input model quality, but the candidate distributions must be prespecified and the method is limited to the i.i.d. case. With respect to i.i.d. random number generation, techniques have been developed to automatically produce code to generate from a wide range of distribution functions [16], but again these distributions must be prespecified.

The situation becomes even more challenging when inputs are not well modeled as a sequence of i.i.d. random variables. A system such as ExpertFit will sometimes detect the lack of i.i.d. structure, but then, with little guidance or software support, the user faces a bewildering array of possible models for autocorrelated and possibly nonstationary sequences, including matrix-geometric processes [24], time series models such as ARIMA, GARCH, and SETAR, with various choices for the innovations distribution [5], or, for arrival processes, direct point process models of arrival times such as nonhomogeneous, compound, clustered, or doubly stochastic Poisson processes [10].

The other option for non-expert input modeling is to fit an empirical distribution for i.i.d. data and use input traces for more general stochastic processes, perhaps combined with some sort of bootstrap resampling [13]. Both approaches usually suffer from the fact that the data values produced during a simulation run are generally limited to those in the available data or, in the case of empirical distributions, require ad hoc tail modeling [6, Section 4.6]. This issue is one aspect of a general overfitting problem: the simulation model captures the training data precisely but does not generalize well beyond it. Use of input traces has the additional drawback that, if the simulation model is to be deployed widely, moving potentially large amounts of data around is cumbersome and raises potential privacy issues. Schruben and Singham [29] have proposed an interesting approach toward resampling trace data that is inspired by agent-based flocking algorithms. Their technique can avoid the overfitting problem but is primarily designed for situations in which there is relatively little trace data—indeed, only one trace is needed—and the goal is to generate qualitatively "similar" sample paths via careful perturbation, where the similarity to the real data is manually controlled by an "affinity" parameter together with added (typically Gaussian)

noise. Our method, in contrast, exploits large amounts of data to learn the underlying generative distribution while avoiding overfitting.

Neural Input Modeling. We aim to exploit the fact that data is becoming ever more abundant due to the increasing use of sensors, the emergence of the Internet of Things, and the retention of log data in formally defined process management systems [31]. Other potential sources of structured log data include information extraction from text [25], as well as from images and video [35]. Our key observation is that, in data-rich environments, neural networks are a powerful and flexible tool for learning complex and subtle patterns from data; if designed carefully, they can potentially automate the tasks of learning simulation input distributions and of generating samples from these distributions during simulation runs.

We present **Neural Input Modeling (NIM)**, a framework for automated modeling and generation of simulation input distributions. NIM uses **Generative Neural Networks (GNNs)**, which not only learn a potentially complex statistical distribution but also provide a means of sampling from the distribution. NIM is designed to avoid overfitting problems and, unlike with input traces, can generate sample values outside the original training range. GNNs have been used in a variety of domains, including synthetic generation of music [33], faces [27], text [4], and more. To the best of our knowledge, NIM is the first system to use GNNs to automate simulation input modeling.

NIM takes inspiration from the inversion method. Suppose we want to generate samples of a continuous random variable X having cumulative distribution function (cdf) F. If we generate $Z \sim N(0,1)$, then it is well known that $X = G(Z)$ is distributed according to F, where $G(Z) = F^{-1}(\Phi(Z))$ and Φ is the cdf of a standard normal random variable. We can extend this idea to a stochastic process $X = (X_1, X_2, \ldots, X_t)$ having joint cdf F by factorizing $F(x_1, x_2, \ldots, x_t)$ as $F_1(x_1)F_2(x_2|x_1)\cdots F_t(x_t|x_1, x_2, \ldots, x_{t-1})$. We then generate i.i.d. normal variates $Z_1, \ldots, Z_t$ and set $X_1 = G_1(Z_1), X_2 = G_2(Z_2|X_1), \ldots, X_t = G_t(Z_t|X_1, X_2, \ldots, X_{t-1})$, where $G_i(z_i|x_1, \ldots, x_{i-1}) = F_i^{-1}(\Phi(z_i)|x_1, \ldots, x_{i-1})$. In other words, we have specified a function G that transforms $Z = (Z_1, \ldots, Z_t)$ into a sample path $X = (X_1, \ldots, X_t)$ having joint distribution F. NIM can be viewed as a system for automatically learning the complex transformation function G from i.i.d. samples of X. (In contrast, the classic NORTA method [8] and its generalizations require a user to manually select the transformation G to achieve a prespecified target distribution for X.)

Our NIM model derives from a particular form of GNN called a **Variational Autoencoder (VAE)** [11, 20]. A VAE uses a pair of neural networks to learn an internal representation of a stochastic process from data (the "encoder") and then transform a sequence of i.i.d. Gaussian input variables into a realization of the modeled process (the "decoder"). A VAE does not need to make any prior assumptions about the features of the training data and appears easier to work with than other types of GNNs, such as generative adversarial networks. In addition, a VAE automatically avoids overfitting by inherently using a regularization term in its goodness-of-fit objective function. Doersch [11, Appendix A] formalizes the "inversion method" reasoning in the prior paragraph for VAEs in a special case involving one-dimensional i.i.d. random samples.

There has been some very recent work on using **Wasserstein Generative Adversarial Networks (WGANs)** to model some specific types of simulation inputs. WGANs are composed of a "generator network" that generates synthetic data and a "discriminator network" that attempts to discern whether input data is real or synthetic; the two networks are jointly trained, with the overall goal of minimizing the Wasserstein distance between the actual and synthetic data distributions. Zheng and Zheng [34] proposed the use of WGANs to model doubly stochastic Poisson processes, and Zhu and Zheng [36], motivated by financial applications, model random sequences having the recursive form $X_{k+1} = \mu(l_k, X_k) + \Sigma(l_k, X_k)\eta_{k+1}$, where η_k is a manually specified domain-specific sequence of random variables, l_k is a deterministic covariate (similar to a special case of a NIM

"condition" as described in the following), and μ and Σ are functions modeled via WGANs. An advantage of the WGAN formulation is the ability to derive statistical properties (although currently under fairly strong assumptions). In contrast, our approach is completely automated and imposes minimal assumptions on the underlying input sequence, allowing modeling of a broad range of stochastic input processes. The downside is that data requirements are typically heavier, since no prior assumptions can be leveraged. Moreover, formal statistical guarantees are currently unavailable and a topic of future work; we therefore provide an extensive empirical study.

Article Organization. In Section 2, we start by reviewing the standard VAE model of Kingma and Welling [20], which uses **Multilayer Perceptrons (MLPs)** (e.g., see [14]) for both the encoder and decoder. Direct use of this VAE leads immediately to a simple neural network, called *NIM-VM*, that is restricted to modeling and generation of i.i.d. random variables. In Section 3, we describe how the basic NIM-VM architecture can be modified by using **Long Short-Term Memory (LSTM)** components [15] for the encoder and decoder. The resulting network, called *NIM-VL*, can then compactly represent complex stochastic processes. NIM-VL can be further adapted to exploit known properties of the stochastic process of interest to increase speed and accuracy. Section 4 describes various extensions of the basic NIM-VL architecture to allow modeling of multivariate, categorical, and nonstationary input processes. In Section 5, we introduce an extension to NIM called **Conditional Neural Input Modeling (CNIM)**, which can learn from training data obtained under various realizations of a (possibly time series valued) influencing stochastic "condition," such as temperature or inflation rate, and then generate sample paths given a value of the condition not seen in the training data. This enables users to simulate a system under a specific working condition by customizing a pretrained model; CNIM also facilitates what-if analysis. In Section 6, we provide empirical evidence showing that NIM and CNIM can accurately and automatically capture a range of complex stochastic input processes and then efficiently generate synthetic sample paths. We also examine the effectiveness of NIM in the context of a queueing simulation model with complex inputs. We conclude in Section 7. An online supplement contains some additional results, including some details about the model-training process and an ablation study confirming that *both* VAE and LSTM components are needed to accurately capture complex simulation inputs. Section and figure numbers prefaced with an "S" refer to the online supplemental materials.

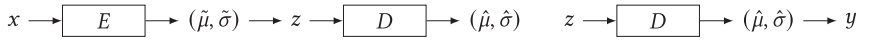
An earlier conference version of this article was presented at the 2020 Winter Simulation Conference. The current work contains significant extensions of the original work, including (i) CNIM, (ii) extensions to multivariate, categorical-valued, and nonstationary input processes, (iii) concomitant experiments for all of these extensions, (iv) the ablation study mentioned previously, and (v) some additional experiments examining the effect of training data and neural network size.

2 VAES AND NIM-VM

In this section, we describe the standard VAE framework, which directly yields the NIM-VM neural network for modeling and generating sequences of i.i.d. random variables.

In general, a GNN can, to a good approximation, generate new data instances from a data distribution $P(x)$ after being shown i.i.d. training samples from P; here, x might be a real-valued random variable, an image, a piece of digital music, or a sample path of a stochastic process. In the setting of simulation input modeling, the new data instances might represent i.i.d. interarrival or service times in a queueing network, blood pressure measurements for arriving patients, and so on. We primarily focus on real-valued stochastic processes throughout but discuss multivariate and categorical processes in Section 4.

VAE Overview. A VAE is a specific type of GNN that accomplishes the learning and generation tasks via a pair of neural networks, an *encoder* E and a *decoder* D. The generative model for the



(a) Training (b) Generation

Fig. 1. NIM-VM training and generation architectures.

observed data assumes that a data sample is created by (i) sampling a *latent variable* from some prior distribution, (ii) feeding that latent variable into a function that outputs a *data-generation distribution*, and (iii) drawing a sample from the data-generation distribution. The encoder E in the VAE learns to infer the latent-variable distribution that likely produced the observed data samples. Thus a trained encoder stochastically maps an observed data value x into a latent value z that serves as the internal representation of x. The decoder D learns the function from (ii), taking a latent-variable sample z and outputting the data-generation distribution from which the final sample is drawn. We use historical data to train both E and D such that the foregoing process will, to a good approximation, generate samples from the underlying data distribution P.

NIM-VM is a direct implementation of such a standard VAE (Figure 1). The prior distribution $P(z)$ of the latent variable z is $N(0, 1)$, a standard normal distribution. The data-generation distribution is of the form $P(x \mid z) = N(\hat{\mu}, \hat{\sigma}^2)$. The decoder D thus learns the mapping from z to $(\hat{\mu}, \hat{\sigma}) = (\hat{\mu}(z), \hat{\sigma}(z))$, and the final sample y is generated from the corresponding normal distribution. This generation process is illustrated in Figure 1(b). We feed a sequence of i.i.d. $N(0, 1)$ z-values into the trained decoder to produce the desired sequence of i.i.d. y-values having distribution P. However, given a data example x, the encoder infers the posterior probability $P(z \mid x)$—the posterior distribution of the latent representation z given the observed data x. This distribution is complex and, in general, expensive to compute: an application of Bayes' theorem requires evaluation of $\int P(x \mid z)P(z)\,dz$ over all configurations of latent variables, and z can be high dimensional for multivariate input processes (see Section 4). The VAE therefore approximates the posterior distribution by a simpler distribution $Q(z \mid x)$, which we take to be a normal distribution $N(\tilde{\mu}, \tilde{\sigma}^2)$ because of its analytical tractability. Thus, encoder E learns the mapping from x to $(\tilde{\mu}, \tilde{\sigma}) = (\tilde{\mu}(x), \tilde{\sigma}(x))$, and the latent variable z is generated from the corresponding normal distribution. This latent-variable generation process is illustrated in the leftmost portion of Figure 1(a).

Note that the input z to the decoder depends on whether we are in the training or generation phase. During training, a sample from the posterior distribution $N(\tilde{\mu}, \tilde{\sigma}^2)$ will be input to the decoder function; this is done by setting $z = \tilde{\mu} + \tilde{\sigma}\xi$, where $\xi \sim N(0, 1)$. During generation, z is a sample from $N(0, 1)$.

Network Architecture. Both the encoder and the decoder employ an MLP structure, which takes inspiration from biology. A standard MLP consists of three layers of neurons: an input layer, a hidden layer, and an output layer. In the hidden layer, each artificial neuron receives a signal from one or more neurons in the input layer, applies a nonlinear "activation function," and then sends the result to the output layer, where the loss is computed. In our setting, the input layer comprises one neuron, the output layer comprises two neurons (one each for the normal mean and variance), and the hidden layer comprises $m \geq 1$ neurons; Figure 2 shows an MLP with $m = 3$. In NIM-VM, the encoder computes $\tilde{\mu}$ and $\tilde{\sigma}^2$ from an observation x via the sequence of computations

$$\tilde{h} = \max(0_m, \tilde{W}_1 x + \tilde{b}_1), \quad \tilde{\mu} = \tilde{W}_2\tilde{h} + \tilde{b}_2, \quad \log\tilde{\sigma}^2 = \tilde{W}_3\tilde{h} + \tilde{b}_3,$$

where 0_m is a column vector of m zeros, the maximum is taken component-wise, $\tilde{W}_1, \tilde{b}_1 \in \mathfrak{R}^{m\times 1}$, $\tilde{W}_2, \tilde{W}_3 \in \mathfrak{R}^{1\times m}$, and $b_2, b_3 \in \mathfrak{R}$. Similarly, the decoder computes $\hat{\mu}$ and $\hat{\sigma}^2$ from a latent variable z via

$$\hat{h} = \max(0_m, \hat{W}_1 z + \hat{b}_1), \quad \hat{\mu} = \hat{W}_2\hat{h} + \hat{b}_2, \quad \log\hat{\sigma}^2 = \hat{W}_3\hat{h} + \hat{b}_3.$$

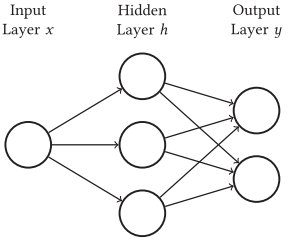


Fig. 2. Multilayer perceptron.

We denote by $\theta = (\tilde{W}_1, \tilde{W}_2, \tilde{W}_3, \tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \hat{W}_1, \hat{W}_2, \hat{W}_3, \hat{b}_1, \hat{b}_2, \hat{b}_3)$ the parameters of the network. The W's and b's are called *weights* and *biases*; they are learned during the training phase, which we now discuss.

Training the VAE. The training phase seeks parameter values θ that minimize a carefully chosen loss function. Specifically, the loss for a training point x is given by

$$L(x;\theta) = D_{\mathrm{KL}}\Big(Q_\theta(z \mid x) \,\Big\|\, P(z)\Big) - E_{z\sim Q_\theta(z|x)}[\log P_\theta(x \mid z)], \tag{1}$$

where D_{KL} denotes **Kullback-Liebler (KL)** divergence and we have explicitly indicated the dependence on θ; see the work of Doersch [11] for a formal derivation. Under the VAE assumptions (i.e., $P(z) = N(0,1)$, $Q_\theta(z \mid x) = N(\tilde{\mu}, \tilde{\sigma}^2)$ and $P_\theta(x \mid z) = N(\hat{\mu}, \hat{\sigma})$), the loss takes the specific form

$$L(x;\theta) = -\frac{1}{2}(\log\tilde{\sigma}^2 - \tilde{\mu}^2 - \tilde{\sigma}^2 + 1) + \frac{1}{2}\left(\log 2\pi + \log\hat{\sigma}^2 + \frac{(x-\hat{\mu})^2}{\hat{\sigma}^2}\right). \tag{2}$$

Note that $\tilde{\mu} = \tilde{\mu}(x;\theta)$, $\tilde{\sigma}^2 = \tilde{\sigma}^2(x;\theta)$, $\hat{\mu} = \hat{\mu}(z;\theta)$, and $\hat{\sigma}^2 = \hat{\sigma}^2(z;\theta)$; we often suppress these dependencies for readability. Also note that the second term is a method-of-moments estimator of the second term in Equation (1): given a training point x, we approximate $E_{z\sim Q_\theta(z|x)}\left[\log P_\theta(x \mid z)\right]$ by $\log P_\theta(x \mid z)$, where z is the value output by the encoder.

The key ideas motivating the form of the loss function are that (i) given i.i.d. $N(0,1)$ z-values, the decoder will produce $\hat{\mu}(z)$ and $\hat{\sigma}^2(z)$ values such that the resulting y-values will jointly be distributed as an i.i.d. sample from the target data distribution, and (ii) a set of z-values produced by the encoder, taken together, look like i.i.d. samples from a standard normal distribution $N(0,1)$, since this is what is needed during generation. In the loss function, the first term represents the KL divergence between $N(\tilde{\mu}, \tilde{\sigma}^2)$ and $N(0,1)$; minimizing this term helps achieve goal (ii) presented earlier. The second term is the negative expected log-likelihood of x under the $N(\hat{\mu}, \hat{\sigma}^2)$ distribution for z, also called the *reconstruction loss*; minimizing this term (i.e., maximizing the expected log-likelihood), helps achieve goal (i), which is to make the synthetic data look like the training data. Importantly, the KL divergence term acts as a regularizer and helps prevent overfitting to the training data (see the following). As further motivation for our loss function, it follows from Doersch [11, Equation (5)] that the loss can be written as

$$L(x;\theta) = -\log P(x) + D_{\mathrm{KL}}\Big(Q_\theta(z \mid x) \,\Big\|\, P(z \mid x)\Big).$$

Since KL divergence is always nonnegative, we have that $L(x;\theta) \geq -\log P(x)$ so that minimizing the loss tends to maximize the log-likelihood of the training data; the better the approximation of $P(z \mid x)$ by $Q_\theta(z \mid x)$, the smaller the KL divergence, and the tighter the relation between loss minimization and likelihood maximization. From a neural network point of view, the KL divergence term, acting as a regularizer, serves to keep the z-values (i.e., internal representations) for the data observations "sufficiently diverse." Without the regularizer, the encoder could learn to "cheat" and

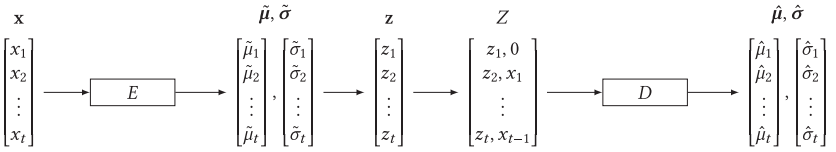


Fig. 3. NIM-VL training architecture.

give each x value a representation in a different region of Euclidean space. This means that similar x-values could be given vastly different representations, undermining the meaningfulness of the latent space of z-values and causing the network to output only the exact training data values. The regularization term has the effect of keeping similar x's representations close together and avoiding the preceding overfitting problem [1]. Some VAE training details are given in Section S1.

3 LSTM COMPONENTS AND NIM-VL

NIM-VM can be used to model and generate i.i.d. univariate random variables. In this section, we describe how to extend our methodology to model and generate i.i.d. sample paths of a univariate stochastic process, given a training set of i.i.d. sample paths.

NIM-VL Overview. A straightforward solution would directly modify NIM-VM so that a training point x is now a sample path: $x = (x_1, x_2, \ldots, x_t)$ for some $t > 1$. Then the input layer of the MLP for encoder E would comprise t neurons to hold x and the output layer would consist of $2t$ neurons to hold $\tilde{\mu} = (\tilde{\mu}_1, \ldots, \tilde{\mu}_t)$ and $\tilde{\sigma}^2 = (\tilde{\sigma}_1^2, \ldots, \tilde{\sigma}_t^2)$. During training, we would generate $z = (z_1, \ldots, z_t)$ by setting $z_i = \tilde{\mu}_i + \tilde{\sigma}_i \xi_i$ for $i \in [1..t]$, where the ξ_i's are i.i.d. $N(0, 1)$. Similarly, the input layer of the MLP for decoder D would now comprise t neurons to hold z and the output layer would contain $2t$ neurons to hold $\hat{\mu}$ and $\hat{\sigma}^2$. During the generation phase, we would feed a vector $z = (z_1, \ldots, z_t)$ of i.i.d. $N(0, 1)$ random variables into the decoder and generate $y = (y_1, \ldots, y_t)$ by setting $y_i = \hat{\mu}_i + \hat{\sigma}_i \zeta_i$ for $i \in [1..t]$, where the ζ_i's are i.i.d. $N(0, 1)$. Correspondingly, we would have $\tilde{W}_1 \in \mathfrak{R}^{m \times t}$, $\tilde{W}_2, \tilde{W}_3 \in \mathfrak{R}^{t \times m}$, and $b_2, b_3 \in \mathfrak{R}^t$.

This approach has three serious problems. First, the number of neurons—or, equivalently, the sizes of the weight matrices and bias vectors—grows linearly with the length of the stochastic process, leading to a network that is slow and cumbersome when modeling long sequences. Second, the model can only handle a fixed input and output size, namely t. For example, if each training sample path has length $t = 100$, then the network can only generate sample paths of length 100. Ideally, we would like a model that learns on length-100 sample paths but can generate, for example, length-1,000 or even longer sample paths. Finally, MLPs are not good at capturing long-range dependencies, which is key to modeling complex stochastic processes [23].

To overcome these problems, we modify the NIM-VM architecture by incorporating LSTM components, as introduced in the work of Hochreiter and Schmidhuber [15]. We refer to the resulting neural architecture as NIM-VL. Specifically, we replace the MLP networks in both the encoder and decoder with LSTM layers. Moreover, instead of feeding a latent variable $z = (z_1, z_2, \ldots, z_t)$ directly to the decoder during training or generation, we pass pairs $(z_1, 0), (z_2, x_1), \ldots, (z_t, x_{t-1})$ to the decoder during training, and pairs $(z_1, 0), (z_2, y_1), \ldots, (z_t, y_{t-1})$ during generation, where $x = (x_1, \ldots, x_t)$ is an input (training) sample path and $y = (y_1, \ldots, y_t)$ is an output (generated) sample path. Figures 3 and 4 show the training and generation architectures.

The LSTM layers allow the VAE to compactly capture temporal dependencies within a sample path. In addition, the change to the decoder's input increases NIM's flexibility. This can be seen clearly in Figure 4. Instead of generating sample paths whose length is fixed and limited by the length of the training data, NIM-VL can generate values one at a time, thereby enabling it to

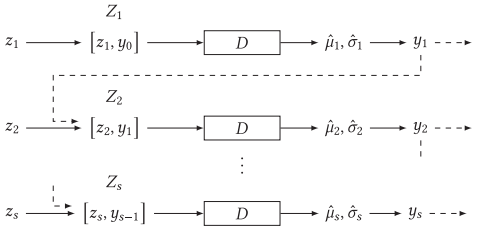


Fig. 4. NIM-VL generation architecture. (Note that we can have $s > t$.)

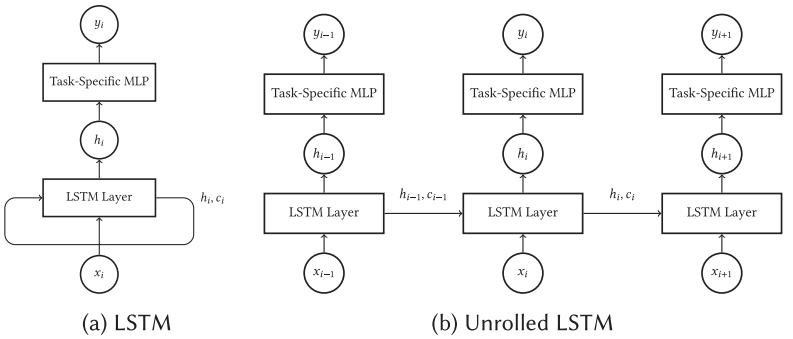


Fig. 5. LSTM architecture and dataflow.

generate sample paths of arbitrary length. We note that, even though the form of the preceding pairs might create the impression that an LSTM is Markovian in nature, the LSTM architecture in fact explicitly models temporal correlations over multiple timesteps [23, Section 1.2]. We note that several authors have recently suggested combining VAEs and LSTMs for use in fields such as robotics [26]—where the LSTM-VAE is used to learn the baseline distribution of dynamic robot behavior to detect behavioral anomalies—and epidemiology [17]—where the neural network is used to forecast disease occurrence at 1 step or 10 steps in the future; these applications are quite different from simulation input modeling, however.

LSTM Details. LSTM networks belong to the family of **Recurrent Neural Networks (RNNs)** (see [23]). RNNs are widely employed in sequence-modeling tasks such as machine translation, image captioning, text-to-speech synthesis, handwriting recognition, and game playing. Whereas an MLP treats a sequence $x = (x_1, x_2, \ldots, x_t)$ as merely a point in a high-dimensional space, RNNs explicitly model the current value x_i as a function of the previous value x_{i-1} and a hidden state vector. This hidden state vector allows RNNs to capture long-range dependencies. For example, an RNN that models arrivals to a restaurant might learn that a high arrival rate in the morning predicts a high arrival rate later that evening. LSTM networks improve upon standard RNNs by allowing easier and more stable training. We give a high-level functional overview of LSTMs in the following, and refer the reader to the work of Hochreiter and Schmidhuber [15] for further details about the low-level architecture.

At a (discrete) timestep i, the LSTM unit receives the *hidden state* and *cell state* from the previous timestep, along with the current input. The unit computes a new hidden and cell state through a series of nonlinear transformations, and passes this data on to the next timestep. The hidden state is also fed into the input layer of a task-specific MLP to compute the output y_i. For our specific NIM-VL encoder, the MLP's hidden layer transforms the input hidden state $\tilde{h}_i$ into an intermediate state $\tilde{g}_i$, which is then used to compute $\tilde{\mu}_i$ and $\tilde{\sigma}_i^2$; the decoder behaves analogously. Figure 5(a)

depicts this process. We also show an "unrolled" version of the LSTM network to clearly illustrate the dataflow (Figure 5(b)).

Formally, the network structure is as follows. Let $h_i, c_i \in \mathfrak{R}^m$ and $g_i \in \mathfrak{R}^l$ be the hidden state, cell state, and task-specific hidden state (i.e., the state of the hidden layer of the task-specific MLP) at time i, where m and l are user-defined sizes. Let x_i be the input at time step i and θ_{LSTM} be the collection of trainable weights inside the LSTM network. For simplicity, we denote the entire LSTM transformation at time i as $(h_i, c_i) = f_{\text{LSTM}}(h_{i-1}, c_{i-1}, x_i; \theta_{\text{LSTM}})$. Then the encoder computes $\tilde{\mu}$ and $\tilde{\sigma}^2$ from an observation x via the following computations, where $\tilde{W}_1 \in \mathfrak{R}^{l\times m}$, $\tilde{b}_1 \in \mathfrak{R}^{l\times 1}$, $\tilde{W}_2, \tilde{W}_3 \in \mathfrak{R}^{1\times l}$, and $\tilde{b}_2, \tilde{b}_3 \in \mathfrak{R}$:

$$(\tilde{h}_i, \tilde{c}_i) = f_{\text{LSTM}}(\tilde{h}_{i-1}, \tilde{c}_{i-1}, x_i; \tilde{\theta}_{\text{LSTM}}),\ \tilde{g}_i = \max(0_l, \tilde{W}_1\tilde{h}_i + \tilde{b}_1),\ \tilde{\mu}_i = \tilde{W}_2\tilde{g}_i + \tilde{b}_2,\ \log\tilde{\sigma}_i^2 = \tilde{W}_3\tilde{g}_i + \tilde{b}_3. \tag{3}$$

The decoder similarly computes $\hat{\mu}$ and $\hat{\sigma}^2$ from z via

$$(\hat{h}_i, \hat{c}_i) = f_{\text{LSTM}}(\hat{h}_{i-1}, \hat{c}_{i-1}, x_{i-1}, z_i; \hat{\theta}_{\text{LSTM}}),\ \hat{g}_i = \max(0_l, \hat{W}_1\hat{h}_i + \hat{b}_1),\ \hat{\mu}_i = \hat{W}_2\hat{g}_i + \hat{b}_2,\ \log\hat{\sigma}_i^2 = \hat{W}_3\hat{g}_i + \hat{b}_3. \tag{4}$$

In our experiments, we set $l = m$ in both the encoder and decoder LSTMs and refer to m as the *size parameter* of the NIM-VL network. (Similarly, the size parameter m of a NIM-VM network is the number of neurons in the hidden layer of the encoder and decoder MLPs.) The training procedure is similar to NIM-VM—see Section S1—in that we use the Adam algorithm for mini-batch gradient descent to minimize an appropriate loss function. With training points now t-dimensional (i.e., $x = (x_1, \ldots, x_t)$), the loss function in Equation (1) now takes the specific form

$$L(x;\theta) = -\frac{1}{2}\sum_{i=1}^{t}(\log\tilde{\sigma}_i^2 - \tilde{\mu}_i^2 - \tilde{\sigma}_i^2 + 1) + \frac{1}{2}\sum_{i=1}^{t}\left(\log 2\pi + \log\hat{\sigma}_i^2 + \frac{(x_i - \hat{\mu}_i)^2}{\hat{\sigma}_i^2}\right). \tag{5}$$

Exploiting Domain Knowledge. When building a simulation, the modeler usually has some domain knowledge about the input processes to the system. For example, when studying the behavior of a queueing system, we know that the interarrival times and the processing times must always be positive. In other cases, we might know that the blood sugar levels of successive patients are i.i.d., or that the distribution of transportation times is multimodal. Exploiting these types of knowledge lets us train a more accurate NIM model, and can also accelerate training and generation. We outline some specific techniques in the following. Although we describe our methods in the context of NIM-VL, they can also be applied to NIM-VM essentially without change.

Bounded Random Variables. If we know *a priori* that there exist lower and/or upper bounds on the range of the input stochastic process of interest, then we first transform each training point $x = (x_1, \ldots, x_t)$ by applying a nonlinear transformation ϕ that maps each value into $(-\infty, +\infty)$, thereby creating a modified training point $x' = (\phi(x_1), \ldots, \phi(x_t))$. We then apply the normal training procedure to the x'-values. During the subsequent generation phase, we apply the inverse transform to each generated point $y' = (y_1, \ldots, y_t)$ to create the final output $y = (\phi^{-1}(y_1'), \ldots, \phi^{-1}(y_t'))$. For example, if the original range is $(\alpha, +\infty)$, we can choose $\phi(v) = \log(v - \alpha)$ and $\phi^{-1}(v) = e^v + \alpha$. Similarly, if the range is $(-\infty, \alpha)$, then we can negate each value so the range becomes $(-\alpha, +\infty)$ and apply the same method. For a range (α, β), we can choose $\phi(v) = \psi^{-1}((v - \alpha)/(\beta - \alpha))$ and $\phi^{-1}(v) = (\beta - \alpha)\psi(v) + \alpha$, where $\psi(v) = 1/(1 + e^{-v})$ is the sigmoid function.

I.i.d. Random Variables. If the target variables are known to be i.i.d., then we can simply use the NIM-VM architecture discussed in Section 2. This can increase both accuracy and generation speed, since NIM-VM will not learn any erroneous intertemporal correlations, and MLP components are simpler and faster than LSTM components.

Multimodal Distributions. If we know that the marginal distributions of the x_i's are multimodal, we can replace the usual Gaussian data-generation distribution $N(\hat{\mu}, \hat{\sigma}^2)$ in the decoder by a Gaussian mixture distribution with M mixture components, where M is a user-specified parameter. Specifically, given an input z_i, the decoder first executes the following computations:

$$(\hat{h}_i, \hat{c}_i) = f_{\text{LSTM}}(\hat{h}_{i-1}, \hat{c}_{i-1}, x_{i-1}, z_i; \hat{\theta}_{\text{LSTM}}), \quad \hat{g}_i = \max(0_l, \hat{W}_1\hat{h}_i + \hat{b}_1), \quad \alpha_i = \text{softmax}(\hat{W}_2\hat{g}_i + \hat{b}_2). \tag{6}$$

For $v = (v_1, \ldots, v_M)$, the ith component of softmax(v) is defined as $e^{v_i} / \sum_{j=1}^{M} e^{v_j}$. The components of the vector $\alpha_i = (\alpha_{i1}, \ldots, \alpha_{iM})$ computed in Equation (6) thus sum to 1 and are interpreted as mixture coefficients. The decoder computes the individual means and variances for the mixture components via

$$\hat{\mu}_{ij} = \hat{W}_{3j}\hat{g}_i + \hat{b}_3, \quad \log \hat{\sigma}_{ij}^2 = \hat{W}_{4j}\hat{g}_i + \hat{b}_4$$

for $j \in [1..M]$. We also change the second term in the loss function accordingly to

$$\sum_{i=1}^{t} \log\left(\sum_{j=1}^{M} \alpha_{ij}(2\pi\hat{\sigma}^2)^{-1/2} \exp\left((x - \hat{\mu}_{ij})^2 / 2\hat{\sigma}_{ij}^2\right)\right).$$

To generate an output y_i, we first choose a mixture component by sampling $j \in [1..M]$ according to the probabilities $\alpha_{i1}, \ldots, \alpha_{iM}$, and then sample y_i from the jth Gaussian distribution $N(\hat{\mu}_{ij}, \hat{\sigma}_{ij}^2)$. When in doubt about the true number M' of modes, it is best for accuracy's sake to err on the side of too many mixture components; if $M > M'$, we found empirically that NIM will automatically make some of the $\hat{\mu}_{ij}$'s and $\hat{\sigma}_{ij}$'s approximately equal to each other so that the effective number of modes is M'.

Necessity of VAE. Is it possible to model stochastic input processes simply by training an LSTM to use the previous value x_{i-1} to compute the probability distribution of the current value x_i? This would lead to a simpler and faster network. In Section S2, we therefore consider an ablated version of the NIM-VL network in which the VAE component is removed and only the LSTM network is kept. In our experiments, this ablated architecture usually failed to generate sample paths having the required statistical characteristics. These results confirm that it is indeed the combination of VAE and LSTM techniques that makes it possible for NIM-VL to model and generate complex stochastic processes.

4 NIM-VL EXTENSIONS

In this section, we discuss three extensions to NIM-VL that enable it to be used when the modeled stochastic sequence is multivariate, when it takes on discrete categorical values, and when it is nonstationary.

Multivariate Sequences. Multivariate stochastic input sequences have traditionally been modeled using classical linear **Vector Autoregressive (VAR)** processes [5, Section 14.2] whose marginal distributions are Gaussian. More recent models that allow arbitrary marginal distributions include the VARTA model of Biller and Nelson [3], which was subsequently extended to allow a broader variety of dependence structures via copula theory [2]. Such models, and many more, can be captured by NIM with minimal changes to the basic architecture. In detail, to model a process where $x_i \in \Re^d$ for $i \geq 0$, we use essentially the NIM-VL (or NIM-VM) architecture as described previously, but with modifications to change the dimensions of $\tilde{\mu}_i$, $\tilde{\sigma}_i$, $\hat{\mu}_i$, $\hat{\mu}_i$, and z_i from 1 to to d. For the encoder, the weights and biases inside the LSTM are now of respective dimensions $m \times d$ and $m \times 1$ so that the hidden and cell states are $m \times 1$ as before, and the weights and biases in

Equation (3) are modified so that $\tilde{W}_2, \tilde{W}_3 \in \mathfrak{R}^{d \times l}$ and $b_2, b_3 \in \mathfrak{R}^d$. The decoder is modified similarly. During training, we compute the d-dimensional variate $z_i = \tilde{\mu}_i + \tilde{\sigma}_i \xi$, where $\xi \sim N(0, I_d)$ is a d-dimensional standard normal random variable; during generation, z_i is sampled from $N(0, I_d)$. In Section 6.1, we demonstrate how NIM can capture processes having statistical dependencies both over time and between state components using a simple VAR(1) sequence.

Categorical-Valued Sequences. To model a stochastic sequence having finite state space $S = \{1, 2, \ldots, C\}$, we modify the NIM architecture so that the decoder produces not the parameters $\hat{\mu}_i$ and $\hat{\sigma}_i^2$ of a normal distribution for generating a real-valued output state y_i but rather the parameters $\hat{p} = (\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_C)$ of a discrete distribution for producing a categorical-valued y_i. Specifically, during the training phase and for each $i \in [1..t]$, we first transform each observed categorical value x_i via one-hot encoding, as is standard for dealing with categorical data. In other words, we transform x_i into a C-dimensional vector whose x_ith element equals 1 and whose jth element equals 0 for $j \neq x_i$. Next, the decoder computations of NIM-VL are changed to

$$(\hat{h}_i, \hat{c}_i) = f_{\text{LSTM}}(\hat{h}_{i-1}, \hat{c}_{i-1}, x_{i-1}, z_i; \hat{\theta}_{\text{LSTM}}),\ \hat{g}_i = \max(0_l, \hat{W}_1 \hat{h}_i + \hat{b}_1),\ \hat{s}_i = \hat{W}_2 \hat{g}_i + \hat{b}_2.$$

Here, $\hat{s}_i = (\hat{s}_{i,1}, \ldots, \hat{s}_{i,C})$ is a "score vector" that is normalized to a vector of probabilities $\hat{p}_i$ using the softmax function defined in Equation (6)—that is,

$$\hat{p}_{i,j} = \frac{e^{\hat{s}_{i,j}}}{\sum_{l=1}^{C} e^{\hat{s}_{i,l}}}$$

for $j \in [1..C]$. The loss function is defined by modifying the reconstruction-loss term in Equation (5) to obtain

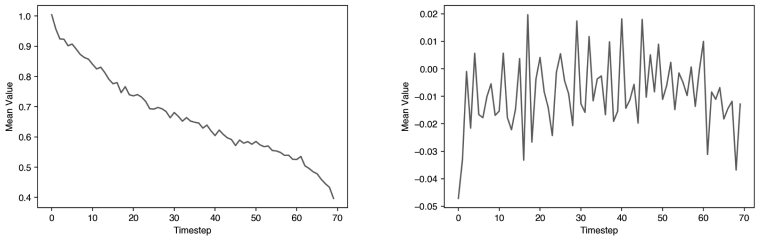
$$L(x; \theta) = -\frac{1}{2} \sum_{i=1}^{t} (\log \tilde{\sigma}_i^2 - \tilde{\mu}_i^2 - \tilde{\sigma}_i^2 + 1) - \sum_{i=1}^{t} \log(\hat{p}_{i,x_i}).$$

During the generation phase, we first sample $z_i \sim N(0, 1)$ as usual. Then we feed z_i into the decoder to compute $\hat{s}_i$ and normalize it via softmax to obtain $\hat{p}_i$. Finally, we generate y_i as a sample from the discrete distribution $\hat{p}_i$. In Section 6.1, we show how our technique can accurately capture a second-order finite-state Markov chain.

Extrapolating Nonstationary Sequences. As exemplified later by Figure 12 in Section 6.1, in which NIM-VL is used to model a **Nonhomogeneous Poisson Process (NHPP)** with periodic rate function, we can sometimes extrapolate beyond the training data. In other words, given training points comprising stochastic sequences of the form $x = (x_1, x_2, \ldots, x_t)$, we can train NIM-VL to generate synthetic sequences of length greater than t. In the case of the Poisson process, where each x_i is an interarrival time, we can extrapolate beyond the continuous-time interval $[0, t]$ on which the training data was observed.

For certain classes of nonstationary input processes, we also can extrapolate beyond the training data by applying differencing techniques developed for classical time series analysis [5]. Specifically, denote by ∇ the time series differencing operator: for $x = (x_1, x_2, \ldots, x_t)$, define $\nabla x = (x_2 - x_1, x_3 - x_2, \ldots, x_t - x_{t-1})$ and recursively define the dth-order differencing operator by $\nabla^d x = \nabla(\nabla^{d-1} x)$. A nonstationary time series x is *homogeneous* if the process $\nabla^d x$ is stationary or periodic for some $d \geq 1$; in practice, small values of d usually suffice.[1] Typically, such processes can be represented as the sum of a stationary or periodic process plus a polynomial trend. For a homogeneous nonstationary sequence, we can simply train NIM-VL on the differenced training

[1]This usage differs slightly from the standard time series terminology in which a time series x is called *homogeneous* if $\nabla^d x$ is stationary for some value of d.



(a) Mean interarrival times (original) (b) Mean interarrival times (differenced)

Fig. 6. The mean values of successive NHPP interarrival times before and after differencing. After differencing, the stochastic process is stationary. (Means are computed over 2,000 interarrival-time sequences.)

sequences. Then, at generation time, we first generate the dth-order differences using NIM-VL and then recover the desired stochastic sequence by inverting the difference operator via cumulative summation. This procedure is analogous to how an ARIMA(p, d, q) model is fit to a nonstationary sequence by first taking dth-order differences and then fitting a stationary ARMA(p, q) model to the differenced observations.

For example, when modeling the interarrival time sequence from a nonstationary NHPP whose rate function is $\lambda(t) = 1 + 0.02t$, the mean of the successive interarrival times is nonstationary and decreasing over time. However, after taking the first-order difference, the mean becomes stationary. Figure 6 shows the effect of the differencing. In general, the degree of differencing required can be informally determined by visual inspection of the time series and its **Autocorrelation Functions (ACFs)** or by various formal statistical tests (e.g., See Section 10.1 in the book by Box et al. [5]). Similarly, periodicity can be detected visually or via spectral density or periodogram [32] methods. We illustrate the main ideas via an example (Section 6.1) but leave a full treatment of these issues to future work.

5 CONDITIONAL NEURAL INPUT MODELING

We now introduce CNIM. This extension to NIM allows us to model and generate stochastic input sequences given a vector C of static ("global") or time-varying ("local") external *conditions* that can influence the characteristics of the sequence. For example, global daily weather conditions (total precipitation, average wind speed, average temperature) can greatly affect the arrival rate of calls to an emergency call center during the day. CNIM trains a single joint model on both interarrival sequences and global weather conditions. During simulation, we then can specify particular weather conditions (total precipitation = 5.2 inches, average wind speed = 20.6 mph, average temperature = 31.7°) and generate call arrival patterns specific to these conditions. For more detailed modeling, we might want to condition the hourly call arrival rates on the hourly weather conditions. Categorical conditions can be handled via one-hot encoding.

Whereas traditional input modeling helps practitioners simulate how the system of interest will behave in general, CNIM can further answer what-if questions about the system under a particular set of working conditions and thus help improve the quality of decision making for this specific scenario. Additionally, CNIM can help us customize our general operational knowledge to novel settings. Examples of the possible CNIM applications include the following:

(1) A company has several factories operating under different working conditions. For example, at one factory, the workers are highly skilled but the machines they use are 6.5 years old. At another site, the workers are less experienced but the machines are only 3 years old, and so on. Suppose that we have collected processing-time datasets $x^{(1)}, \ldots x^{(n)}$ for the n factories.

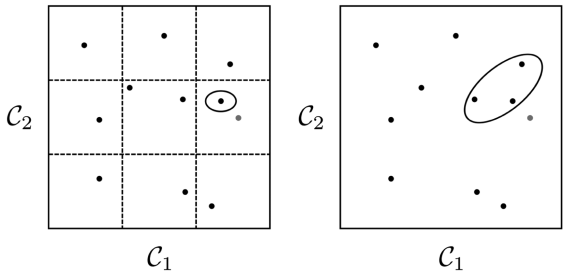


Fig. 7. Two alternative methods for generating sample paths based on an external condition $C = (C_1, C_2)$. On the left, the condition space is partitioned and the data examples in the same grid are selected as training data. On the right, three nearest neighbors in the condition space are selected to train the NIM model.

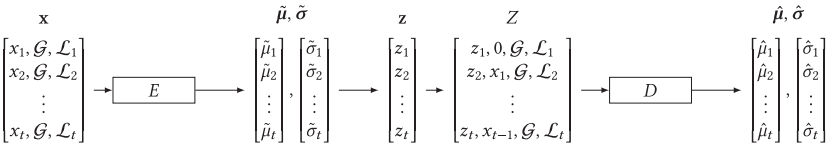
Now the company is opening a new factory where the workers are highly skilled and the machines are 4 years old. We are interested in producing processing time samples under these new conditions to simulate operations at this new factory.

(2) The orders placed at an ice cream shop correlate closely with real-time temperature readings throughout the day. We would like to generate sample paths of the form $(x_1, x_2, \ldots, x_{12})$—where x_i is the amount of ice cream ordered during the ith hour of operation—given hourly temperature observations. This application is similar to the factory example, except that the conditions are local and take the form of a stochastic process rather than a static global vector of parameter values.

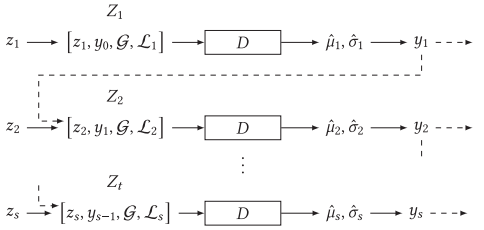
Importantly, CNIM interpolates from the data it has seen during training and does not try to extrapolate to the regions in the condition-value space where there is no training data available. For example, if the training data only contains observations under sunny and rainy weather conditions, then the user might be able to generate input sequences under intermittent showers but will not be able to generate sequences under snowy conditions.

One potential alternative to CNIM is the *partition method*: to generate sample paths under condition vector C, we first partition the space of condition values (the "condition space") into a grid and then train an (unconditional) NIM model on the sequences whose condition value falls into the same grid cell as C (see Figure 7). Another alternative is the *nearest neighbor method*, where we train a NIM model on the k nearest data points to C. As we demonstrate in Section 6.4, however, neither method works very well, for several reasons. First, to generate sample paths under multiple conditions, one must train multiple NIM models, one for each condition, which often translates into relatively long training times. Second, it can be difficult to find a good scheme selecting the appropriate data points on which to train. For example, we must determine the appropriate width of the grids or how many neighboring condition values we want to include. The situation becomes especially difficult when the condition is represented as a high-dimensional real-valued vector. Most critically, after partitioning or selecting the nearest neighbors, there may not be enough data in a given region of the condition space to train a high-quality model. In contrast, for CNIM, we only need to train a single model, which can then generate sample paths under different conditions. By training on the entire dataset, CNIM can learn the common features across all data examples and fine-tune its response by automatically interpolating the points near C.

Technical Details. Mathematically, CNIM aims to generate a stochastic process sample path $x = (x_1, \ldots, x_t)$, given an external condition in the form of a sequence $C = (C_1, \ldots, C_t)$ of the same length as x. In general, each element of C_i is of the form $C_i = [\mathcal{G}, \mathcal{L}_i]$, where $\mathcal{G} = [\mathcal{G}_1, \ldots, \mathcal{G}_q]$ is a static vector of $q \geq 1$ "global" conditions that hold over the entire duration of the input process



(a) CNIM training architecture



(b) CNIM generation architecture

Fig. 8. CNIM architecture.

and $\mathcal{L} = (\mathcal{L}_1, \ldots, \mathcal{L}_t)$ is a time series of "local" conditions that vary over the simulation run; each $\mathcal{L}_i$ is itself a vector of length $n \geq 1$. For instance, in the factory example, we have $\mathcal{G} = [\mathcal{G}_1, \mathcal{G}_2]$, where $\mathcal{G}_1 = 1$ if the workers are highly skilled and 0 otherwise, and $\mathcal{G}_2 =$ the age of the machines. Thus, $\mathcal{G} = (1, 6.5)$ for the first factory and $\mathcal{G} = (0, 3.0)$ for the second. The condition $\mathcal{G}$ impacts the entire stochastic sequence of processing times. There is no local condition, so $\mathcal{L}_i = []$ for $i \in [1..t]$. In the example of the ice cream shop, there is no global condition, so that $\mathcal{G} = []$, and the local condition $\mathcal{L}_i$ is the temperature at the ith timestep.

From a technical perspective, the extension from NIM to CNIM does not require changing the NIM architecture; it only requires changing the training data provided to NIM. In other words, both global and local CNIM models lead to the same neural network architecture and loss function as before, with the difference being how we concatenate the stochastic process x and the condition c and feed the combined data to NIM. Specifically, the input to the encoder during both training and generation becomes $x' = (x'_1, \ldots, x'_t)$, where $x'_i = [x_i, C_i]$ for $i \in [1..t]$, and the input to the decoder becomes $Z'_i = [z_i, x_{i-1}, C_i]$ for $i \in [1..t]$. Figure 8 shows the resulting architectures of both models.

Practicalities. One natural question is whether the global condition $\mathcal{G}$ must be concatenated with each x_i when forming the input to NIM. A possible alternative is to introduce $\mathcal{G}$ into the computation by using it to initialize the hidden state of the LSTM component and letting this component handle the propagation of $\mathcal{G}$ over all of the timesteps. Unfortunately, this type of approach fails because, due to the finite capacity of the LSTM component, initial conditions eventually fade out, so that the global condition $\mathcal{G}$ is not correctly propagated to the ith time step as i becomes large. We therefore must "manually" propagate $\mathcal{G}$ by appending it to each x_i. We note, however, that we do not need to store multiple copies of $\mathcal{G}$ on disk; condition $\mathcal{G}$ can be stored once and concatenated with each x_i as the input-process sample paths are fed into NIM. Moreover, using, for example, the PyTorch expand operator, we need only store one copy of $\mathcal{G}$ in memory during training.

There are some subtle considerations when determining whether a given condition should be treated as local or global. In particular, our network architecture (see Figure 8) implicitly assumes that

$$P(x_i \mid x_1, \ldots, x_{i-1}; \mathcal{L}_1, \ldots, \mathcal{L}_t) = P(x_i \mid x_1, \ldots, x_{i-1}; \mathcal{L}_1, \ldots, \mathcal{L}_i) \tag{7}$$

so that $P(x_i)$ cannot depend on any information that will only be revealed after timestep i. As an example, suppose we have a dataset consisting of stochastic process $x = (x_1, \ldots, x_t)$ and the local condition $\mathcal{L} = (\mathcal{L}_1, \ldots, \mathcal{L}_t)$ where x_i are the arrival times for an NHPP and each $\mathcal{L}_i$ is the instantaneous arrival rate at time x_i (i.e., $\mathcal{L}_i = \lambda(x_i)$). At generation time, suppose we have a specific local condition $\mathcal{L}' = (\mathcal{L}'_1, \ldots, \mathcal{L}'_t)$ so that our goal is to generate sample paths $y = (y_1, \ldots, y_t)$ such that the instantaneous rate at time y_i is $\mathcal{L}'_i$. In this case, CNIM will not work well because, for an NHPP, we have

$$P(x_{i+1} > t \mid x_i = t_i) = \exp\Big(-\int_{t_i}^{t} \lambda(u)\, du\Big).$$

In other words, to calculate the probability distribution of next arrival time x_{i+1}, we need to know the entire shape of $\lambda(u)$ for $u \in [x_i, t)$, in violation of (7). We therefore need to use a global condition $\mathcal{G} = (\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_t)$. We show, via an experiment in Section 6.4, that the global approach can be used successfully for this example.

The additional functionality of CNIM comes at a computational cost. Specifically, assuming that the dimension of the encoder's hidden layer $\hat{h}$ is $|\hat{h}|$ and that each x_i is real valued, the weight matrix to map the encoder input $[x_i, \mathcal{G}, \mathcal{L}_i]$ to the hidden vector $\hat{h}$ is of size $(|\mathcal{G}| + |\mathcal{L}| + 1) \times |\hat{h}|$, where $|\mathcal{G}|$ and $|\mathcal{L}|$ are the dimensions of $\mathcal{G}$ and $\mathcal{L}_i$. For ordinary NIM, the weight matrix is only of size $1 \times |\hat{h}|$. A similar analysis applies for the decoder. In the foregoing NHPP example, $|\mathcal{G}| = t$ so that CNIM becomes relatively expensive when t is large.

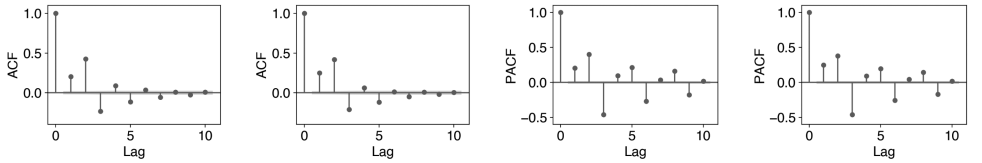
6 EXPERIMENTS

We conducted a series of experiments to assess NIM's accuracy and performance. Our results indicate that NIM can faithfully model both complex i.i.d. probability distributions and complex, possibly nonstationary stochastic processes. Moreover, when interarrival and service times learned by NIM are used to drive a GI/G/1 queueing simulation, the outputs are close to those of the same simulation driven by the "ground truth" input processes. NIM was also able to accurately model the arrival process to a real-world call center. For CNIM, we found that the sample paths it generates have the required characteristics specified by the conditional parameters. Finally, NIM exhibited fast training and generation speeds, enabling practical deployment, and the required training set sizes were modest compared to other VAE applications such as images, music, and text.

In general, we selected the network sizes essentially by cross validation. One simple approach for doing this is as follows: (i) partition the overall training data into training and testing subsets, (ii) determine an appropriate error metric such as the integrated difference between ground truth and empirical arrival rates, (iii) initially fit a small NIM model on the training subset, and (iv) iteratively increase the model size until the error (as measured using the testing subset) reaches an acceptable value. See also the discussion on AutoML in Section 6.5.

6.1 Synthetic Complex Input Processes

We tested NIM-VL on seven synthetic stochastic processes of varying complexity: an ARMA(3,3) process, a nonstationary ARMA/ARMA mixture process, an interarrival-time sequence for a nonhomogenous Poisson process, the waiting-time sequence for a GI/G/1 queue, a sequence of i.i.d. multimodal random variables, a multivariate VAR(1) process, and a Markov chain with finite state space. In each case, NIM-VL was trained with 1,000 ground truth sample paths, each of length 100 unless otherwise noted. (For the i.i.d. example, this amounts to 100,000 i.i.d. training samples.) Sample paths generated traditionally are called *ground truth*, and those generated by our VAE are called *NIM-VL*. Unless specified otherwise, we used size parameter $m = 32$ for NIM-VL and NIM-VM, as defined previously.



(a) ACF GT (b) ACF NIM-VL (c) PACF GT (d) PACF NIM-VL

Fig. 9. ACF and PACF for the ARMA(3,3) process.

Table 1. Mean Log-Likelihood of Sample Paths Generated by NIM-VL, Ground Truth, and ExpertFit

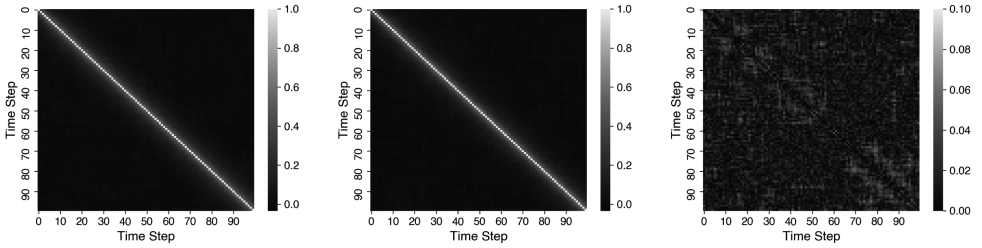
Stochastic Process	NIM-VL	Ground Truth	ExpertFit	ExpertFit Dist'n/Quality
ARMA(3, 3)	−198.56 ± 1.18	−197.98 ± 1.07	−5,931.81 ± 240.43	Johnson SU/Good
NHPP	43.00 ± 0.17	43.53 ± 0.17	37.73 ± 0.18	Pearson Type VI/Good
I.i.d. Gamma-unif. mix.	−236.26 ± 0.39	−226.95 ± 0.19	−415.42 ± 0.64	Johnson SB/Bad

ARMA(3,3). We first modeled an ARMA(3,3) process using NIM-VL. The autoregressive coefficients were $(0.3, 0.4, -0.1)$, and the moving average coefficients were $(0.2, 0.3, -0.7)$. Distribution of each error term (for this and all other ARMA and ARIMA processes discussed in the article unless otherwise noted) is $N(0, 1)$. At test time, we generated one NIM-VL sample path and compared it to a ground truth ARMA(3, 3) sample path, both of length 10,000. Figure 9 shows the empirical ACF and **Partial Autocorrelation Function (PACF)** of the ground truth ARMA time series and NIM-generated sample path. Notice that the ACF and PACF of the NIM-VL generated sample path have damped sine wave shapes almost identical to those of the ground truth ARMA process.

We also numerically evaluated the fidelity of NIM-VL by computing the mean log-likelihood, under the ARMA(3,3) distribution, over 1,000 sample paths of length $t = 100$ generated (i) by NIM-VL after training, (ii) as sequences of i.i.d. samples from a fitted distribution found by ExpertFit, and (iii) directly from a true ARMA(3,3) distribution (and different from the paths used to train NIM-VL).

The results are given in the first row of Table 1, where precision is indicated by 95% confidence interval bounds. It can be seen that the mean log-likelihood for NIM-VL-generated sample paths is close to that for the ground truth sample paths. ExpertFit was unable to detect the inter-temporal correlations of ARMA(3,3), and mistakenly identified the process as a sequence of i.i.d. samples from a Johnson SU distribution, rating the fit as "good." Not surprisingly, the log-likelihood score was extremely poor in this case.

ARMA/ARMA Mixture. Next, we considered a mixture $\{X_i\}_{i\geq 1}$ of two nonstationary ARMA(2,2) processes $\{A_i\}_{i\geq 1}$ and $\{B_i\}_{i\geq 1}$. To generate a sample path, we ran both processes in parallel; at timestep i, we set $X_i = A_i$ with probability 0.5 and otherwise set $X_i = B_i$. The parameters of the two processes are $(0.95, -0.1; 0.2, 0.95)$ and $(0.8, -0.3; 0.3, 0.7)$. At test time, the trained model was used to generate 10,000 NIM-VL sample paths of length 100, and these NIM-VL sample paths were compared against 10,000 validation ground truth sample paths (again distinct from the sample paths used for training). As a simple way to compare these complex, nonstationary stochastic processes, we took the validation ground truth sample paths $x_n = (x_{n,1}, \ldots, x_{n,100})$ for $n \in [1..10{,}000]$ and computed empirical correlation coefficients $\hat{\rho}_{ij}^{\text{GT}} = \widehat{\text{Corr}}[X_i, X_j]$ for $1 \leq i, j \leq 100$. We similarly computed $\hat{\rho}_{ij}^{\text{NIM}}$ for the NIM-VL sample paths and plotted the absolute differences $d_{ij} = |\hat{\rho}_{ij}^{\text{NIM}} - \hat{\rho}_{ij}^{\text{GT}}|$ in a heat map. Figure 10(a) and 10(b) show the empirical correlations $\hat{\rho}_{ij}^{\text{GT}}$ and $\hat{\rho}_{ij}^{\text{NIM}}$, respectively. Figure 10(c) gives the correlation difference plot. Note that Figure 10(a) and (b) both have scale



(a) Ground truth (b) NIM-VL (c) Difference

Fig. 10. Empirical correlation coefficients for the ARMA/ARMA mixture stochastic process.

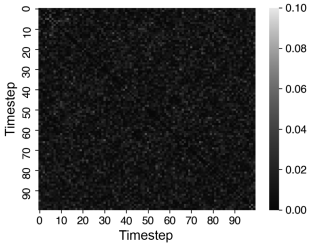


Fig. 11. Empirical correlation coefficient differences for NHPP.

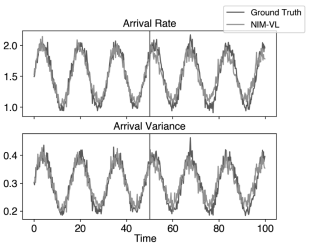


Fig. 12. Empirical arrival rate and arrival variance for NHPP.

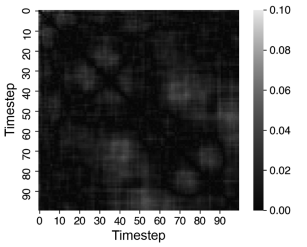


Fig. 13. Empirical correlation coefficient differences for GI/G/1 waiting-time sequence.

$[0, 1]$, whereas Figure 10(c) has scale $[0, 0.1]$. The largest absolute correlation difference value is 0.060, indicating good agreement.

Nonhomogeneous Poisson Process. We next tested our approach on an NHPP interarrival time process with a rate function $\lambda(t) = \frac{1}{2}\sin(\frac{\pi}{8}t) + \frac{3}{2}$. Because we know *a priori* that interarrival times for an NHPP are positive with probability 1, we applied the log-transformation technique described in Section 3 when using NIM-VL. Ground truth data was generated via thinning (see [22]). Figure 11 shows the correlation differences. The largest absolute correlation difference is 0.069, again indicating good agreement.

We also compared the empirical arrival-rate function to the ground truth function $\lambda(\cdot)$ given previously. We trained NIM-VL on 1,000 sample paths of arrival times in the time interval $[0, 50]$ and tested on 10,000 sample paths with arrivals in $[0, 100]$. In detail, we computed the empirical arrival-rate function by first computing the sequence of arrival times (by taking partial sums of interarrival times), then dividing the interval $[0, 100]$ into small subintervals of length 0.2, and finally computing the average number of arrivals in each subinterval over the 10,000 sample paths. We also computed the empirical and theoretical variance of the number of arrivals in each subinterval.

Figure 12 shows the empirical and ground truth arrival-rate functions, as well as the empirical and ground truth variance in the number of arrivals over the small subintervals. As can be seen, the agreement is quite good, not just during the interval $[0, 50]$, the period on which NIM-VL was trained, but also on the interval $[50, 100]$, where NIM-VL never saw data values during training. NIM-VL thus learned the underlying statistical structure of the NHPP—without knowing that it *was* an NHPP—and thus was able to extrapolate beyond the training data.

Next, we computed log-likelihoods of generated sample paths, analogously to our ARMA(3,3) experiment. Looking at the second row of Table 1, it can be seen that, as with ARMA(3,3), the mean log-likelihood for NIM-VL generated sample paths is close to that for the ground truth sample

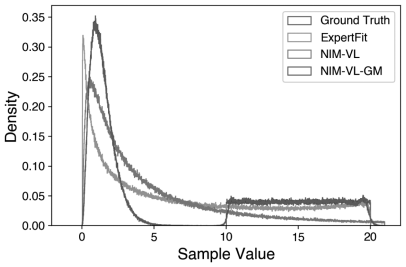


Fig. 14. Empirical densities for Gamma-Uniform mixture.

paths. ExpertFit is again unable to detect intertemporal correlations, and mistakenly identifies the process as a sequence of i.i.d. samples from a Pearson Type VI distribution, rating the fit as "good." Although the log-likelihood score is much better than for ARMA(3,3), it is still lower than for NIM-VL. We note that a more careful user of ExpertFit could generate an autocorrelation plot, and hence might visually detect the lack of independence, but then there would be no guidance on what to do next.

Waiting Time Sequence for a GI/G/1 Queue. The next complex stochastic process that we examined was the waiting-time sequence for a GI/G/1 queue, which can be generated from the well-known Lindley recursion or by simulating the queue. We used a Gamma$(0.25, 4)$ interarrival-time distribution and a Gamma$(0.5, 1)$ service-time distribution. Since the waiting time is always nonnegative, we used the log-transformation described in Section 3. We generated 1,000 sample paths, each comprising the sequence of waiting times for the first 100 jobs, and used these to train NIM-VL with size parameter $m = 128$. We then compared the NIM-VL sample paths against a validation set of 1,000 ground truth sample paths. Figure 13 shows the empirical correlation differences. The maximum difference is 0.056, again showing good agreement.

Multimodal Distribution. Finally, in the i.i.d. setting, we show how domain knowledge can be exploited to accurately capture a multimodal distribution by using the Gaussian-mixture technique of Section 3, as well as a log-transformation to enforce nonnegativity. We denote the resulting network as NIM-VL-GM. Specifically, we consider a mixture distribution having two components, a Gamma$(2.875, 0.5)$ and a Uniform$(10, 20)$ distribution, with mixing weights 0.6 and 0.4. For this challenging distribution, ExpertFit fitted a Johnson SB distribution family to the data, completely missing the multimodality, but at least rated the fit as "bad." As shown in Figure 14, the enhanced version of NIM-VL is able to capture the structure of this distribution. Moreover, the third line in Table 1 shows that the empirical mean log-likelihood of NIM-VL-GM is close to that of ground truth and is significantly higher than that of ExpertFit.

Multivariate Stochastic Process. We tested the ability of NIM to capture a multivariate stochastic process using a two-dimensional VAR(1) process of the form

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} 0.3 & -0.4 \\ -0.6 & -0.3 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} e_{1,t} \\ e_{2,t} \end{bmatrix},$$

where $e_{1,t}$ and $e_{2,t}$ are i.i.d. $N(0, 1)$ random variables. We created a dataset of 1,000 sample paths from this process comprising 50 timesteps each. After training, we used NIM-VL to generate one sample path of length 500 and estimated the coefficient matrix via least squares [5, p. 516]. The estimated coefficient matrix is

$$\begin{bmatrix} 0.330 & -0.381 \\ -0.634 & -0.228 \end{bmatrix},$$

Table 2. Mean Log-Likelihood of Multivariate Sample Paths Generated by NIM-VL, Ground Truth, and the i.i.d. Model

Stochastic Process	NIM-VL	Ground Truth	I.i.d. 2D Normal
VAR(1)	$-1,435.20$	$-1,438.10$	$-1,616.34$

Table 3. Ground Truth Transition Probability of Second-Order Markov Chain and Estimated Transition Probability of the Sequences Generated by NIM-VL

	Next State		
Current State	1	2	3
1, 1	1/10	1/2	2/5
1, 2	1/3	1/3	1/3
1, 3	1/2	1/3	1/6
2, 1	1/5	7/10	1/10
2, 2	4/5	1/10	1/10
2, 3	1/10	4/5	1/10
3, 1	3/10	3/10	2/5
3, 2	2/5	2/5	1/5
3, 3	1/10	1/3	17/30

True transition probabilities.

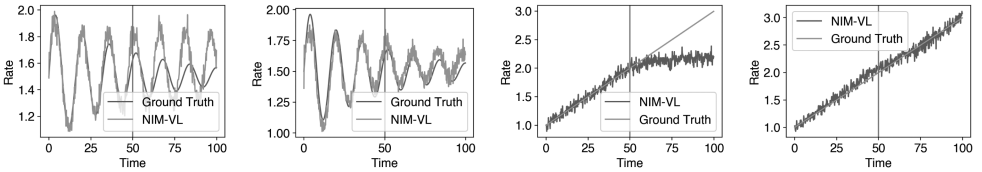
	Next State		
Current State	1	2	3
1, 1	0.095	0.504	0.401
1, 2	0.325	0.323	0.352
1, 3	0.484	0.307	0.210
2, 1	0.163	0.729	0.108
2, 2	0.839	0.068	0.093
2, 3	0.071	0.839	0.090
3, 1	0.275	0.309	0.416
3, 2	0.386	0.387	0.227
3, 3	0.107	0.345	0.547

NIM-VL empirical transition probabilities.

a close match to the ground truth coefficient matrix that generated the original process. Furthermore, we computed the mean log-likelihood of sample paths generated by NIM-VL and compared it with that of a true VAR(1) process. We also evaluated the mean log-likelihood in a hypothetical situation where correlation across timesteps is ignored and the process is erroneously modeled as an i.i.d. sequence of two-dimensional Gaussian vectors. The result is shown in Table 2. As can be seen, the mean log-likelihood for NIM-VL-generated sample paths falls in the same range as for the ground truth sample paths and is much higher than for the sample paths generated from the erroneous i.i.d. Gaussian model. This experiment indicates that the multivariate extension of NIM-VL can accurately model certain multivariate stochastic processes. If the state dimension increases and there are many complex interactions between state components at a given time or at successive times, as in a sequence of video frames, then our architecture would encounter increasing difficulty in capturing the process due to both process complexity and increasing network size. Increases in order are easier to handle.

Finite-State Categorical Stochastic Process. The next experiment models a second-order Markov chain to test NIM-VL's ability to model discrete stochastic process with finite number of categories. The transition probabilities are given on the left side of Table 3. As in previous experiments, we generated a training dataset of 1,000 sample paths, each of length 50. The neural network has size parameter $m = 32$. After training, we used NIM-VL to generate discrete sequences and estimated the empirical transition probability, shown on the right side of Table 3. It can be seen that the true and empirical transition probabilities agree closely.

Extrapolating Nonstationary Stochastic Processes. We consider two examples where NIM-VL can successfully extrapolate a nonstationary process beyond the time interval used for training. The first example is "easier," in that the ground truth values in the extrapolated time interval are similar to a subset of the values that occur during the training interval, and the evolution of the data values follows a predictable pattern (although NIM-VL must learn the pattern). In this case, we show that standard NIM-VL can successfully extrapolate values if there is enough training data



(a) Size param. $m = 32$ (b) Size param. $m = 256$ (c) Without differencing (d) With differencing

Fig. 15. Extrapolation of two nonstationary NHPPs with different rate functions. In each case, NIM-VL was trained on arrivals in the interval $[0, 50]$ and arrival times were extrapolated to the interval $[50, 100]$; empirical arrival rates are based on 2,000 i.i.d. sample paths. (a, b) The empirical rates for a damped sine wave rate function. (c, d) The empirical arrival rate of for a linearly increasing rate function without and with differencing.

and the network size parameter m is sufficiently large. For the second example, the ground truth values in the extrapolation range are different from those in the training range so that standard NIM-VL fails. Because the process is homogeneous nonstationary, we show that differencing can be applied successfully.

For the first example, we considered an NHPP process whose rate function is given by $\lambda(t) = 0.5 \exp(-0.02t) \sin(\frac{\pi}{8}t) + 1.5$ (i.e., a damped sine wave). We used a dataset comprising 1,000 training sequences to train two different NIM-VL networks having size-parameter values of $m = 32$ and $m = 256$, respectively. We then compared ground truth and empirical rate functions for each network. As seen in Figure 15(a) and (b), both networks performed well over the training interval. In the extrapolation interval, the larger network was able to model the gradually decreasing amplitude of the sine wave, whereas the smaller network was not. Thus, given enough training data, increasing the size of a NIM-VL network can improve its ability to model more complex patterns in the dataset. As shown in Table 4 in Section 6.5, a complementary result is that, for a given network size, a larger training dataset is typically needed to capture more subtle statistical features of the stochastic process.

Our second example demonstrates the use of differencing to allow extrapolation of homogeneous nonstationary input processes beyond the training interval. In detail, we considered an NHPP having linear rate function $\lambda(t) = 1 + 0.02t$. We created a training dataset comprising 2,000 sequences of interarrival times in the time interval $[0, 50]$. Although a NIM-VL model trained directly on the interarrival-time sequences captured the linear rate function within the range $[0, 50]$, the model was unable to correctly extrapolate the rate function to the range $[50, 100]$, as seen in Figure 15(c). To overcome this difficulty, we trained NIM-VL on the successive differences of arrival times, used the trained model to generate synthetic sequences of differences, and then integrated these sequences to obtain the desired synthetic sequences of interarrival times. As seen in Figure 15(d), use of the differencing technique allowed us to correctly extrapolate the rate function to the interval $[50, 100]$. In an additional experiment (Section S4) we found that NIM-VL with differencing can accurately model a stochastic sequence that is both nonstationary and periodic.

The preceding results indicate that modeling the differenced sequence can be a simple yet effective way for NIM to extrapolate homogeneous nonstationary stochastic processes. Our examples had linear trends, so first-order differencing sufficed, but as mentioned previously, techniques from the time series literature can be adapted to automatically estimate the required order of differencing for processes having a polynomial trend. Other trend removal techniques can potentially be used to handle other types of nonstationary stochastic processes (e.g., see [7, pp. 14–25]), but it becomes challenging to automate these procedures for non-experts. We leave a full treatment of extrapolation beyond the training data to future work.

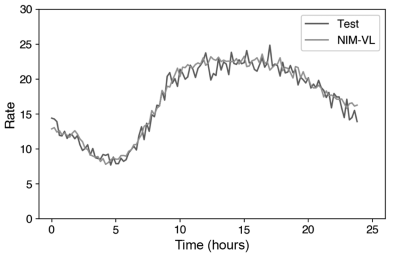


Fig. 16. Empirical arrival rates for emergency-call data.

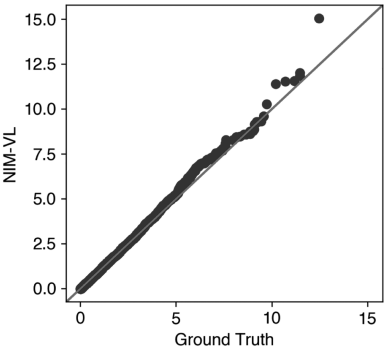


Fig. 17. Q-Q plot for W_{60} distribution in a GI/G/1 queue.

6.2 Real-World Dataset: Emergency Call Center

We applied NIM-VL (with size parameter $m = 128$) to a real-world dataset, the daily emergency-call interarrival times for the San Francisco Fire Department in 2018. We randomly selected two-thirds of the data (243 days) to train the model. We then generated sample paths and compared their empirical arrival-rate function to the remaining one-third (122 days) of the San Francisco Fire Department data using the same method discussed in the previous section.

As can be seen in Figure 16, the empirical arrival-rate function of NIM-generated data closely approximates that of the actual data. Note that the ground truth arrival-rate function is constructed from less data than was used to construct the arrival-rate function for NIM-VL, and hence is less smooth.

6.3 A Queueing Simulation

Our next experiment examines the end-to-end effect of using inputs from NIM-VL to simulate the waiting time W_{60} of the 60th job in an NHPP/Gamma/1 FIFO queue: the interarrival time process is an NHPP as before with i.i.d. Gamma(1.2, 0.4) service times. The training sets for interarrival times and for service times each comprise 1,000 sample paths with 50 observations per path. We applied our log-transformation method for interarrival and service times. Figure 17 shows a Q-Q plot for the distribution of W_{60} over 4,000 simulation replications, comparing simulations with ground truth inputs to simulations with NIM-VL inputs; there is close agreement between the simulations. Note that each sample path in the training data comprises only 50 jobs, but we simulate 60 jobs. This indicates that, if the system is not too nonstationary, we can extrapolate beyond our training set; a trace-driven simulation would not be applicable here.

6.4 Conditional Neural Input Modeling

Global Condition. Our first experiment for CNIM tests its ability to generate sample paths when given a two-dimensional global condition $\mathcal{G} = (\mathcal{G}_1, \mathcal{G}_2)$. Specifically, our training dataset comprises 2,000 sequences of interarrival times for NHPPs, where each sequence is generated according to rate function $\lambda(t; \mathcal{G}) = \mathcal{G}_1 \sin(\mathcal{G}_2 t) + 1.8$ with $\mathcal{G}$ selected randomly and uniformly from the rectangle $[0.5, 1] \times [10, 20]$, the "condition space." Thus, $\mathcal{G}_1$ and $\mathcal{G}_2$ control the amplitude and frequency of the periodic rate function. At test time, we evaluate the quality of CNIM-generated sample paths at different locations in the condition space. Figure 18 plots the empirical rate functions estimated from 2,000 sample paths for $\mathcal{G}$ values located at the four corners, midpoints of the four edges, and the center of the condition space. We emphasize that none of the training points were

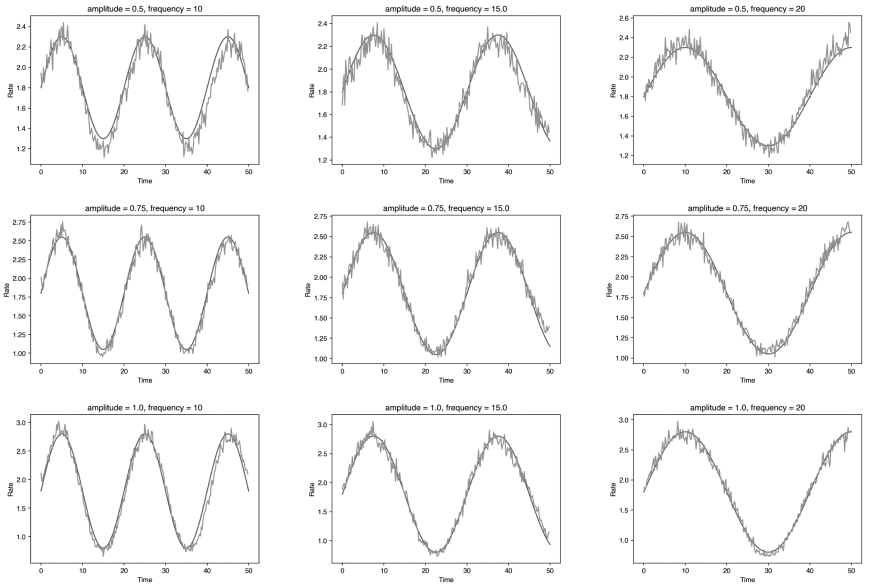


Fig. 18. Empirical arrival rate estimated at nine different points in the condition space.

generated using any of these specific nine values of $\mathcal{G}$. It can be seen that all of the empirical rate functions match the ground truth rate functions closely. We further study the mean log-likelihood throughout the condition space. In other words, for each $\mathcal{G}$ in an 11×11 grid of points that cover the condition space, we generate 2,000 NHPP sample paths using the rate function $\lambda(t; \mathcal{G})$ and compute the mean log-likelihood over the paths. For a given sample path $x = (x_1, x_2, \ldots, x_{n(x)})$ generated under a condition $\mathcal{G}$, the log-likelihood is

$$l(x_1, x_2, \ldots, x_{n(x)}) = \sum_{j=1}^{n(x)} \log \lambda(t_j; \mathcal{G}) - \int_0^T \lambda(u; \mathcal{G})\, du,$$

where $t_j = \sum_{k=1}^{j} x_k$ is the jth arrival time for $j \in [1..n]$. Besides CNIM, we also implemented the partition and nearest neighbors methods mentioned in Section 5. The absolute differences between the (estimated) mean log-likelihood for learned and ground truth sample paths at the various values of $\mathcal{G}$ are shown for the three methods in Figure 19. As we can see, the absolute difference in log-likelihood is much higher for these two alternative procedures that only use the nearby training points. Overall, this experiment shows that CNIM is able to learn how the characteristics of a stochastic process change as some external global condition changes.

Local Condition. Our next experiment tests CNIM's performance for a local condition. We consider a stochastic process $x = (x_1, \ldots, x_t)$ whose state x_i is influenced by two kinds of events: UP and DOWN. Specifically, $x_0 = 0$ and

$$x_i = \begin{cases} x_{i-1} + w_i, & \text{if } \mathcal{L}_i \text{ is UP} \\ x_{i-1} - w_i, & \text{if } \mathcal{L}_i \text{ is DOWN} \end{cases}$$

for $i \geq 1$, where $\{w_i\}_{i \geq 1}$ is a sequence of i.i.d. $N(0.05, 0.1^2)$ random variables. We collected 2,000 sample paths of this system as training data, where each training data point has 50 timesteps. For each timestep i, we encode UP event as 1 and DOWN event as -1 and randomly choose between

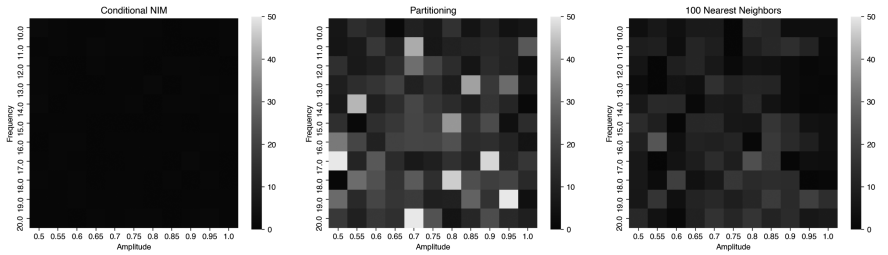


Fig. 19. Absolute differences between estimated mean log-likelihood of learned and ground truth sample paths for CNIM, partitioning, and nearest neighbors methods, evaluated at $11 \times 11 = 121$ points on a grid covering the condition space.

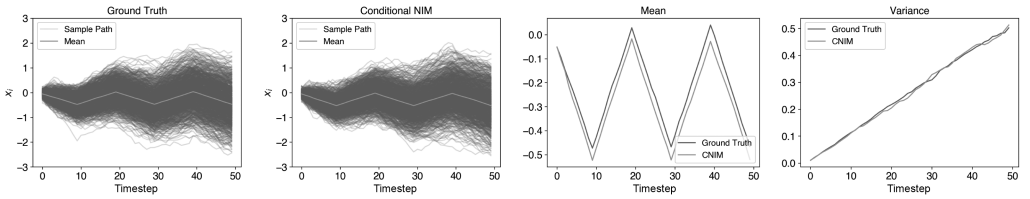


Fig. 20. CNIM for a local-condition sequence: sample paths, mean, and variance.

the two events with equal probability. After training the model, we generate sample paths under a fixed event sequence l, specifically, $\mathcal{L}_i = -1$ for $i \in \{1, \ldots, 10\} \cup \{20, \ldots, 30\} \cup \{40, \ldots, 50\}$ and $\mathcal{L}_i = 1$ otherwise. Figure 20 shows 1,000 ground truth and CNIM-generated sample paths of the stochastic process x under the specific event sequence. As can be seen, the sample-path behavior for CNIM is quite similar to the ground truth behavior. Figure 20 also shows the time-varying empirical mean and variance of the sample paths generated by CNIM and of the ground truth sample paths. As expected, the mean follows a zig-zag pattern and the variance increases linearly. Again, the behavior of CNIM closely approximates the ground truth.

Global vs. Local Condition for an NHPP. In Section 5, we asserted that, in the context of modeling the sequence of arrival times $x = (x_1, x_2, \ldots, x_t)$ for an NHPP, the ostensibly "local" condition of instantaneous arrival rates $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_t)$—where $\mathcal{L}_i = \lambda(x_i)$—actually needs to be treated as a global condition in order not to violate the "nonanticipatory" condition in Equation (7). In this example, we show how to model an NHPP conditioned on realization of rate function by using the discretized realization of the rate function as a global condition. We developed a CNIM model for the customer arrivals of an ice cream shop where we assume the arrival process follows an NHPP whose rate function is $\lambda(t) = 0.5 + 0.1T(t)$; here, $T(t)$ is the temperature at time t for $t \in [0, 24]$. Further, $T(t)$ is determined by four parameters: $a \in [15, 25]$ is the temperature at the beginning of the day, $b \in [14, 17]$ is the time when the temperatures peaks during the day, $c \in [25, 30]$ is the peak temperature at b, and $d \in [15, 25]$ is the temperature at the end of the day. $T(t)$ is linear between $[0, b]$ and $[b, 24]$. All four parameters are uniformly randomly sampled in their ranges. Note that we do not directly provide the four parameters as a condition but instead give the realization of the temperatures as a condition. In particular, for a specific sample path of the temperature process, we discretize it into a vector $\mathcal{G} = (\mathcal{G}_1, \ldots, \mathcal{G}_{50})$, which is the temperature process evaluated at 50 evenly spaced points from $t = 0$ to $t = 24$, and provide this vector as a global condition. We collected 2,000 interarrival time sequences over the time interval $[0, 24]$ as training data. For Figure 21, we used CNIM to generate 20,000 sample paths with $(21, 16, 25, 24)$

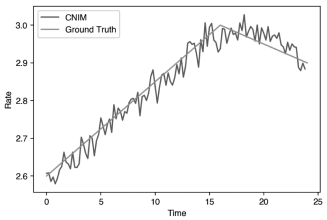


Fig. 21. Modeling an NHPP based on a discretized rate function as the global condition.

Table 4. Minimum Number of Training Sample Paths Required to Achieve Fidelity Comparable to an n = 1,000 Baseline

Stochastic Process:	ARMA(3, 3)	ARMA mix.	NHPP	GI/G/1 Waiting time	Gamma-unif. mix.
Min. No. of Sample Paths:	10	500	250	500	1,000

as the temperature process parameter—again, the temperature process is discretized into a vector $\mathcal{G}' = (\mathcal{G}'_1, \ldots, \mathcal{G}'_{50})$—and estimated the empirical rate function. As can be seen, CNIM was able to capture this NHPP by using a discretized rate function as global condition and thus bypassing the "nonanticipatory" restriction.

6.5 Training Times, Generation Speeds, and Training Set Size

In this section, we consider some practical aspects of our NIM implementation.

Training Times. We implemented NIM-VL in PyTorch 1.0 and trained our models on a workstation equipped with a 2.10-GHz Intel Xeon CPU having 376 GB of RAM, plus an NVIDIA GeForce RTX 2080 Ti GPU with 12 GB of memory and CUDA 10.1 installed. Training times ranged between 10 and 20 minutes. As discussed in Section S3, training times compared favorably to at least one state-of-the-art method for modeling NHPPs.

Generation Speeds. After training, the models were exported in ONNX format, which can then be used for sample-path generation on a wide range of computers. We tested generation speeds on a commodity 2018 MacBook Pro (2.2-GHz six-core Intel Core i7 with 32 GB of RAM). We were able to generate 1,000 sequences of 1,000 learned NHPP interarrival times in roughly 0.85 seconds. For i.i.d. data, NIM-VM (with 32 hidden nodes per encoder and decoder) is able to generate 10^6 i.i.d. learned exponential random variables in roughly 0.12 seconds. Since sample-path generation consists mainly of simple matrix multiplications, it can potentially be accelerated via GPUs. Thus, for the neural network sizes considered, NIM-VL was fast enough to be usable in practice. As mentioned previously, we selected network sizes essentially by ad hoc cross validation, and found that a NIM-VL size parameter of $m = 32$ sufficed for many of our simpler input processes (especially when extrapolation was not required), a size parameter of $m = 128$ sufficed for most of the more complex processes, and a maximum size parameter of $m = 256$ yielded satisfactory results in all experiments. In future work, we intend to investigate how AutoML techniques [19] can be used to automatically and efficiently select the smallest (and hence fastest) NIM model that will suffice to capture a given input process.

Training Set Size. We have used a baseline training set size of 1,000 sample paths throughout, with excellent results. Unlike, for example, image generation, where huge amounts of training data are needed, the data requirements for stochastic process modeling seem lighter. We tested the effect of training set size, recording for each of the five synthetic stochastic processes in Section 6.1 the smallest size for which the fidelity was comparable to the baseline (see Table 4). We observe

that, not surprisingly, the simpler the distribution or stochastic process, the less training data is needed.

7 CONCLUSION

GNNs are promising tools for automating the modeling and generation of simulation input data. NIM can automatically capture complex stochastic processes, thereby facilitating one of the hardest tasks in a simulation study and promoting wider use of simulation modeling and analysis. To this end, we have released NIM as an open source tool (https://github.com/cenwangumass/nim). We have extended the basic NIM technology in a variety of ways, allowing it to exploit *a priori* knowledge, to capture a greater variety of input processes, to sometimes extrapolate beyond the training interval, and to allow conditional generation of input-process sample paths. We emphasize, however, that NIM is not a "silver bullet." Data must be plentiful, and estimating tails (especially heavy ones) and extrapolating far beyond the training data is challenging for any input modeling scheme. Sanity checking and validation are still needed—for example, using visualization and data mining to explore the training and NIM-generated data and the resulting simulation output. In future work, we plan to further develop capabilities around extrapolation and processes with discrete state spaces, to explore other types of GNNs, such as WGANs, Wasserstein autoencoders [30], and diffusion models [28], develop statistical error bounds, and investigate automatic network sizing via AutoML methods.

ACKNOWLEDGMENTS

The authors wish to thank Justin Domke for helpful insights into GNNs and Emily A. Herbert for contributions to initial versions of this work. We thank the reviewers for their helpful comments.

REFERENCES

[1] Jaan Altosaar. 2020. Tutorial—What Is a Variational Autoencoder? Retrieved April 18, 2020 from https://jaan.io/what-is-variational-autoencoder-vae-tutorial.
[2] Bahar Biller. 2009. Copula-based multivariate input models for stochastic simulation. *Operations Research* 57, 4 (2009), 878–892.
[3] Bahar Biller and Barry L. Nelson. 2003. Modeling and generating multivariate time-series input processes using a vector autoregressive technique. *ACM Transactions on Modeling and Computer Simulation* 13, 3 (2003), 211–237.
[4] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Jozefowicz, and Samy Bengio. 2016. Generating sentences from a continuous space. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*. 10–21.
[5] George E. P. Box, Gwilym M. Jenkins, Gregory C. Reinsel, and Greta M. Ljung. 2016. *Time Series Analysis: Forecasting and Control*. Wiley.
[6] P. Bratley, B. L. Fox, and L. E. Schrage. 1987. *A Guide to Simulation*. Springer-Verlag.
[7] P. J. Brockwell and R. A. Davis. 2009. *Time Series: Theory and Methods*. Springer.
[8] M. C. Cario and B. L. Nelson. 1997. *Modeling and Generating Random Vectors with Arbitrary Marginal Distributions and Correlation Matrix*. Technical Report. Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL.
[9] Wang Cen, Emily A. Herbert, and Peter J. Haas. 2020. NIM: Modeling and generation of simulation inputs via generative neural networks. In *Proceedings of the 2020 Winter Simulation Conference (WSC'20)*. IEEE, Piscataway, NJ, 584–595.
[10] D. R. Cox and V. Isham. 1980. *Point Processes*. Chapman & Hall.
[11] Carl Doersch. 2016. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908v2* (2016).
[12] Geer Mountain Software 2020. Stat::Fit Distribution Fitting Software. Retrieved April 18, 2020 from https://www.geerms.com.
[13] Wolfgang Härdle, Joel Horowitz, and Jens-Peter Kreiss. 2003. Bootstrap methods for time series. *International Statistical Review* 71, 2 (2003), 435–459.
[14] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning* (2nd ed.). Springer.
[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[16] Wolfgang Hörmann, Josef Leydold, and Gerhard Derflinger. 2004. *Automatic Nonuniform Random Variate Generation*. Springer.
[17] M. R. Ibrahim, J. Haworth, A. Lipani, N. Aslam, T. Cheng, and N. Christie. 2021. Variational-LSTM autoencoder to forecast the spread of coronavirus across the globe. *PLoS One* 16, 1 (2021), e0246120.
[18] Wendy Xi Jiang and Barry L. Nelson. 2018. Better input modeling via model averaging. In *Proceedings of the 2018 Winter Simulation Conference (WSC'18)*. IEEE, Piscataway, NJ, 1575–1586.
[19] Shubhra Kanti Karmaker, Md. Mahadi Hassan, Micah J. Smith, Lei Xu, Chengxiang Zhai, and Kalyan Veeramachaneni. 2021. AutoML to date and beyond: Challenges and opportunities. *ACM Computing Surveys* 54, 8 (2021), 1–36.
[20] Diederik P. Kingma and Max Welling. 2013. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114v10* (2013).
[21] Averill M. Law. 2015. *Simulation Modeling and Analysis* (5th ed.). McGraw-Hill, New York, NY.
[22] P. A. W. Lewis and G. S. Shedler. 1979. Simulation of nonhomogeneous Poisson processes by thinning. *Naval Research Logistics Quarterly* 26 (1979), 403–413.
[23] Zachary C. Lipton. 2015. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019v4* (2015).
[24] Marcel Neuts. 1981. *Matrix-Geometric Solutions in Stochastic Models*. Dover.
[25] Christina Niklaus, Matthias Cetto, André Freitas, and Siegfried Handschuh. 2018. A survey on open information extraction. In *Proceedings of the 27th International Conference on Computational Linguistics (COLING'18)*. 3866–3878.
[26] Daehyung Park, Yuuna Hoshi, and Charles C. Kemp. 2018. A multimodal anomaly detector for robot-assisted feeding using an LSTM-based variational autoencoder. *IEEE Robotics and Automation Letters* 3, 3 (2018), 1544–1551.
[27] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).
[28] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125* (2022).
[29] Lee W. Schruben and Dashi I. Singham. 2014. Data-driven simulation of complex multidimensional time series. *ACM Transactions on Modeling and Computer Simulation* 24, 1 (2014), Article 5, 13 pages.
[30] Ilya Tolstikhin, Olivier Bousquet, Sylvain Gelly, and Bernhard Schoelkopf. 2017. Wasserstein auto-encoders. *arXiv preprint arXiv:1711.01558* (2017).
[31] Wil M. P. van der Aalst. 2018. Process mining and simulation: A match made in heaven! In *Proceedings of the 2018 Summer Simulation Conference (SummerSim'18)*. Article 4, 12 pages.
[32] Jacob T. VanderPlas. 2018. Understanding the Lomb-Scargle periodogram. *Astrophysical Journal Supplement Series* 236, 1 (2018), 16.
[33] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. 2017. MidiNet: A convolutional generative adversarial network for symbolic-domain music generation. *arXiv preprint arXiv:1703.10847* (2017).
[34] Yufeng Zheng and Zeyu Zheng. 2021. Doubly stochastic generative arrivals modeling. *arXiv:2012.13940* [stat.ML] (2021).
[35] Luowei Zhou, Chenliang Xu, and Jason J. Corso. 2018. Towards automatic learning of procedures from web instructional videos. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, the 30th Innovative Applications of Artificial Intelligence Conference, and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (AAAI'18/IAAI'18/EAAI'18)*. 7590–7598.
[36] Tingyu Zhu and Zeyu Zheng. 2021. Learning to simulate sequentially generated data via neural networks and Wasserstein training. In *Proceedings of the 2021 Winter Simulation Conference (WSC'21)*. IEEE, Piscataway, NJ, 1–12.

Received 12 January 2022; revised 24 October 2022; accepted 3 April 2023

ACM Transactions on Modeling and Computer Simulation

https://tomacs.acm.org

Guide to Manuscript Submission

Submission to the *ACM Transactions on Modeling and Computer Simulation* is done electronically through https://mc.manuscriptcentral.com/tomacs. Please visit https://tomacs.acm.org/authors.html for further information and guidelines on submission procedure.

You will be asked to create an abstract that will be used throughout the system as a synopsis of your paper. You will also be asked to classify your submission using the ACM Computing Classification System through a link provided at the Author Center. For completeness, please select at least one primary-level classification followed by two secondary-level classifications. To make the process easier, you may cut and paste from the list. Remember, you, the author, know best which area and sub-areas are covered by your paper; in addition to clarifying the area where your paper belongs, classification often helps in quickly identifying suitable reviewers for your paper. So it is important that you provide as thorough a classification of your paper as possible.

The ACM Production Department prefers that your manuscript be prepared in either LaTeX or MS Word format. Style files for manuscript preparation can be obtained at the following location: https://www.acm.org/publications/authors/submissions. For editorial review, the manuscript should be submitted as a PDF or Postscript file. Accompanying material can be in any number of text or image formats, as well as software/documentation bundles in zip or tar-gzipped formats.

Questions regarding editorial review process should be directed to the Editor-in-Chief. Questions regarding the post-acceptance production process should be addressed to the Associate Director of Publications, Sara Kate Heukerott, at heukerott@hq.acm.org.

Ceasing Print Publication of ACM Journals and Transactions

ACM has made the decision to cease print publication for ACM's journals and transactions as of January 2024. There were several motivations for this change: ACM wants to be as environmentally friendly as possible; print journals lack the new features and functionality of the electronic versions in the ACM Digital Library; and print subscriptions, which have been declining for years, have now reached a level where the time was right to sunset print.

Please contact acmhelp@acm.org should you have any questions.

Subscription and Membership Information.

Send orders to:

ACM Member Services Dept.
General Post Office
PO Box 30777
New York, NY 10087-0777

For information, contact:

Mail: ACM Member Services Dept.
1601 Broadway, 10th Floor
New York, NY 10019-7434
Phone: +1-212-626-0500
Fax: +1-212-944-1318
Email: acmhelp@acm.org
Catalog: https://www.acm.org/publications/alacarte

About ACM. ACM is the world's largest educational and scientific computing society, uniting educators, researchers and professionals to inspire dialogue, share resources and address the field's challenges. ACM strengthens the computing profession's collective voice through strong leadership, promotion of the highest standards, and recognition of technical excellence. ACM supports the professional growth of its members by providing opportunities for life-long learning, career development, and professional networking.

Visit ACM's Web site: https://www.acm.org.

Change of Address Notification. To notify ACM of a change of address, use the addresses above or send an email to coa@acm.org.

Please allow 6-8 weeks for new membership or change of name and address to become effective. Send your old label with your new address notification. To avoid interruption of service, notify your local post office before change of residence. For a fee, the post office will forward 2nd- and 3rd-class periodicals.