Neurosim: A Fast Simulator for Neuromorphic Robot Perception

Anonymous CVPR submission

Paper ID *****

Abstract

Neurosim is a fast, real-time, high-performance library for simulating sensors such as dynamic vision sensors, RGB cameras, depth sensors, optical flow, and inertial sensors. It can simulate agile dynamics of multi-rotor vehicles in complex and dynamic environments. Neurosim achieves frame rates as high as 2700 FPS on a desktop GPU, with ∼10× faster event simulation than existing methods. Neurosim integrates with a ZeroMQ-based communication library called Cortex to facilitate seamless integration with machine learning and robotics workflows. Cortex provides a high-throughput, low-latency message-passing system for Python and C++ applications, with native support for NumPy arrays and PyTorch tensors. This paper discusses the design philosophy behind Neurosim and Cortex. It demonstrates how they can be used to (i) train neuromorphic perception and control algorithms, e.g., using self-supervised and reinforcement learning on time-synchronized multi-modal data, and (ii) test real-time implementations of these algorithms in closed-loop. [1]

1. Introduction

Beginning with the work of Carver Mead and Misha Mahowald in the late 1980s, researchers in neuromorphic engineering have sought to emulate the retina and the rest of the brain in silicon. Event-based cameras are the latest hardware that this research has produced. They emulate rods and cones in the retina and record changes in the intensity of light (increases or decreases beyond a threshold) at each pixel. Event cameras, as illustrated in Fig. 1, have a high temporal resolution ($<$ 100 µs), high dynamic range (80-120 dB) and operate asynchronously with high temporal precision (∼1 ns). These properties make them ideal for fast, robust perception in difficult lighting, high speeds, and power-constrained settings.

Data from an event camera consists of a stream of 4-tuples (t_i, u_i, p_i) for $i \in \mathbb{N}$ where $t_i \in \mathbb{Z}_+$ denotes the timestamp (typically, in microseconds) of an "event", $u_i \in \mathbb{Z}^2$ are the pixel coordinates where the event occurred and the polarity $p_i \in \{-1, 1\}$ indicates whether the intensity at pixel u_i increased or decreased. So the first challenge in developing algorithms for event data comes from the fact that we need to rethink computer vision from the ground up—unlike a standard RGB camera, an event camera does not register an image. Even annotating event data for supervised learning on downstream tasks is challenging, due to its unusual nature and because it is quite noisy. It is tempting to use cross-modal supervision, e.g., either by transferring annotations from RGB and LiDAR data or by using pseudo-labels from systems trained on these other modalities. This is a viable approach, but provided appropriate care is taken to synchronize timestamps and obtain accurate extrinsic calibration. The largest event camera dataset containing multi-sensor data, M3ED, has less than four hours of event data [1].

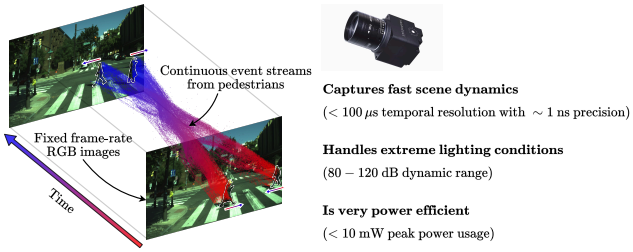

Figure 1. **Event cameras capture the scene without temporal aliasing, with high dynamic range, and consume very little power.** We illustrate the temporal non-aliasing property of event cameras in the left image. Unlike standard RGB cameras, event cameras are asynchronous sensors that respond to scene intensity changes and produce a continuous stream of information at high temporal and spatial precision.

Simulators enable us to work around these issues. Our desiderata for a simulator are as follows.

1. **Efficient, high-fidelity event simulation.** Events sense only the changes in the intensity of light, so we should expect that the sim-to-real gap for systems trained on simulated event data is smaller than that trained on simulated RGB data. RGB simulators need to capture the rich

[1] Neurosim and Cortex will be made open-source upon acceptance.

color information, lighting effects (shadows, reflections, etc.), and motion blur to be effective in the real world. Works like [9] have obtained remarkable results by training entirely on simulated event datasets. We need to render the scene at very high frame rates (a good thumb-rule is 1000+ FPS) for high-fidelity event simulation. Ideally, event data should be created on-the-fly without storing the corresponding rendered RGB images.[2]

2. **High-throughput multi-modal data from sensors for RGB images, depth, optical flow, and inertial motion, along with ground-truth from the scene and intrinsic and extrinsic calibration.** For example, a typical robot receives LiDAR data at ∼10 Hz and in total about a half a billion points per second, event data from a monocular high-definition camera at ∼50 M events per second, an HD RGB camera provides about ∼1 M pixels at ∼100 Hz, while an inertial measurement unit can register angular velocities, linear accelerations and magnetometer orientations upwards of ∼500 Hz. Sensors typically operate using their individual clocks and are synchronized externally. The quality of synchronization depends on the kind of hardware used for this purpose, and also upon the estimated motion of the robot. A simulator should simulate these realities of real-world robots that will affect downstream algorithms. And do so extremely quickly in a fraction of real-time for simulation to be a viable alternative to data curation.
3. **Feeding multi-modal robot perception data to deep learning training pipelines.** The sensors above would result in about 1 GB s^{-1}. It is important to design mechanisms to feed such low-latency, high-throughput data to deep learning training pipelines in Python-native formats without having to store the data on a hard disk, or excessive buffering in the data loaders, or artificially slowing down the simulator.
4. **Ability to run closed-loop perception and control experiments at the extremes of the performance envelope of the hardware.** Event cameras lead to interesting perception-control loops for agile robots. While RGB images would be hopelessly blurred when a quadrotor undergoes a flip (angular velocity of ∼700 $^\circ s^{-1}$), an event camera would see key-points without temporal aliasing. It is difficult to set up experiments that push the hardware platform to the limits of its agility, e.g., flips, avoiding moving obstacles, or a humanoid falling, it risks damage to the platform and the researcher.

[2]Existing simulators like TartanAir [21] work around this problem by generating high frame rate (1000 FPS) RGB data, saving them to disk, and then post-processing them to generate event data using [16]—a prohibitively slow process that consumes roughly 2-3 terabytes of storage for an hour of high-definition (HD) simulation. Alternatives like CARLA [4] and AirSim [20] suffer from temporal aliasing. They rely on low frame rate (∼30 FPS) RGB rendering to approximate high frequency events, and as a consequence have significant artifacts in the generated event data.

2. The Design of Neurosim

Neurosim is an asynchronous, modular, and high-performance simulator designed with a focus on neuromorphic perception on multirotors. Its design philosophy hinges on four key principles: (1) modularity – each component (rendering, dynamics, controllers, communication) operates independently and can be extended, replaced, and fault-isolated; (2) low-latency – the visual and dynamics simulators are chosen to be fast. The full simulation stack (with an event camera) achieves ∼2700 FPS on a single Nvidia 4090 GPU; (2) real-time streaming – data flows directly to training pipelines without the need for large intermediate disk storage. Reproducibility is ensured through randomization seeds rather than traditional static datasets; (4) Pythonic accessibility – a simple API that is Python and deep learning friendly. As we will show in Sec. 4, these design choices make Neurosim easy to use for a variety of applications in embodied perception. In this section, we discuss the key components of Neurosim as illustrated in Fig. 2.

Neurosim simulates event cameras at multi-kilohertz rates. We choose Habitat-Sim [18] to render the scene because of its exceptional rendering speed and flexibility. It supports diverse 3D asset formats, is compatible with multiple open-source 3D scene datasets, and provides built-in rigid-body physics integration through PyBullet [2]. Habitat-Sim is equipped with a variety of sensors and camera models, including RGB-D, semantic, and egomotion cameras – each configurable with independent intrinsics and sensor parameters. Habitat-Sim can render a typical indoor scene with multiple RGB-D sensors at VGA (640 × 480) resolutions at ∼3000 FPS on a single GPU.[3]

Neurosim uses the rendered intensity images to simulate an event camera by keeping track of a state for each pixel that records the intensity corresponding to the last triggered event at that pixel. The high frame rate in Habitat-Sim reduces the likelihood of large intensity changes between consecutive frames that could lead to missed events. Real-world event sensors exhibit non-idealities such as noise, bandwidth limitations, and sensor saturation effects. Neurosim simulates these artifacts, although they can be disabled by the user if so desired. The resultant rendering speed, more than 2700 FPS, corresponds to a temporal resolution of ∼0.37 ms for simulated events. Most event-based algorithms working on real data utilize event rates far lower than our simulated resolution [3], [7].

Current CPU implementations of the above contrast-threshold-based event generation model in the literature

[3]Our modular design allows the visual renderer in Neurosim to support diverse 3D assets and sensors. For Habitat-Sim, this includes Matterport3D, Gibson, Replica, MP3D+Gibson, etc., and sensors such as RGB, Depth, Semantic, Egomotion, etc.

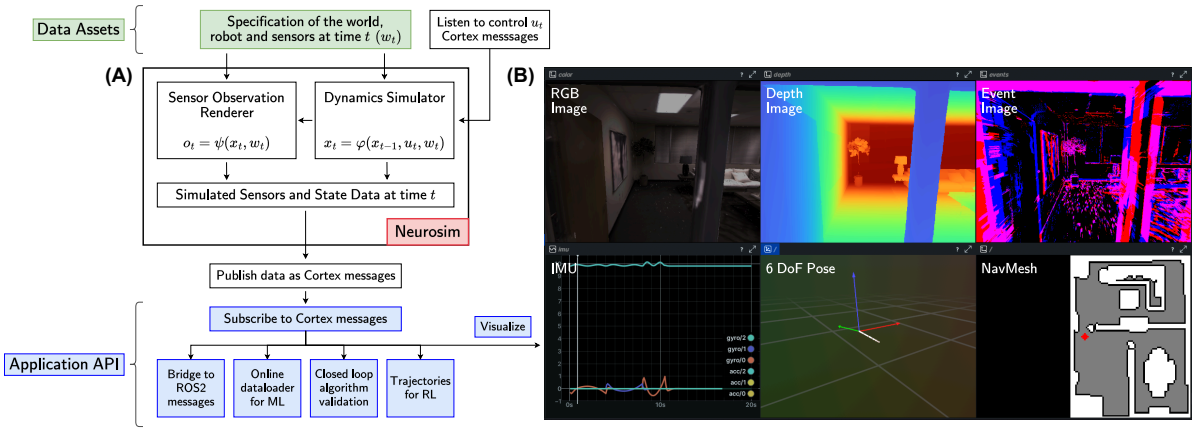

Figure 2. **Overview of the design of Neurosim.** Neurosim is designed to be modular, high-performance, and easy to use for a variety of applications in embodied perception on multirotors. As shown in **(A)**, it consists of four main components: (1) A data stack in the form of 3D scene assets and the specifications for sensors and multirotors, (2) A real-time rendering engine for high-fidelity vision sensor simulation, (3) A fast and accurate multirotor dynamics model for aerodynamics and physics, and (4) A communication interface to interact with the simulator – receive sensor data and send control commands. In **(B)**, we illustrate how we can use the communication interface to connect Neurosim to a visualizer – here Rerun [17] – and draw the simulated data in real-time. It displays the rendered RGB image, depth map, IMU readings, 6-DoF pose, navigation mesh, and the event stream. For visualization, events are accumulated over short time windows (20 ms) and colored by polarity.

[4, 20] are exceedingly slow. GPU implementations such as [16] are faster, but they still require multiple CUDA-kernel launches and do not effectively utilize the shared memory and warp parallelism features of modern GPUs. Neurosim uses a CUDA kernel to compute events across pixels in parallel. It then aggregates events efficiently at warp level to avoid atomic contention. Each thread first processes a single pixel and determines whether an event is triggered. Threads within a warp then cooperatively compute a bit-mask of active (event-generating) threads, and count the number of events in that warp. GPU SRAM (shared memory, which is roughly 20× faster than global memory) is used to store intermediate results and perform warp-level reductions. A single atomic operation reserves a contiguous block in the global event buffers for all events in that warp. The base index of this block is broadcast to all threads in the warp. Each active thread then computes its own offset within the warp by counting preceding set bits in the warp mask, and writes its event (pixel coordinates, timestamp, and polarity) into the pre-allocated segment of the output arrays. This warp-synchronous design minimizes the number of global atomics from one per event to only about one per warp – roughly a 32× (warp size) reduction. When aggregated across all resident warps on a streaming multiprocessor (e.g., 64 warps per SM on recent Nvidia architectures), the peak number of concurrent event writes can scale to thousands of threads per SM while still using only tens of atomics. This is important because each rendering step produces a variable number of events. Naive pre-allocation requires two kernel launches, and per-event atomics are prohibitively expensive.

Our design significantly increases throughput and sustains event generation from multi-kilohertz intensity images on modern GPUs. Neurosim can process a new intensity frame, trigger events based on the contrast threshold model, update the internal pixel states, and output the event stream at over 31 kHz for VGA and 23 kHz for HD frames. We benchmark our implementation against existing event simulators in Fig. 3.

Running closed-loop perception and control experiments at the extremes of the performance envelope of the hardware Neurosim, in its first iteration, includes support for a fast and accurate model of multirotor dynamics and inertial sensors named RotorPy [6]. Control commands (rotor RPMs, or thrust and angular velocities in body frame) can be provided to change the vehicle pose. We include parameters like aerodynamic drag specifications and multirotor configurations like mass, inertia, rotor positions and motor response characteristics for accurate simulation. Neurosim also supports swapping out the dynamics simulator for different libraries, custom implementations, or other platforms. RotorPy can simulate the dynamics at more than 20 kHz on a single CPU core. It includes procedures for motion planning for multirotor platforms using differential flatness-based trajectory generation. Dynamics and sensor simulation in Neurosim are coupled tightly with very little latency to ensure that sensor simulation faithfully captures the vehicle pose. Sec. 4 shows an example of how this can be used to sample agile trajectories in congested indoor environments to simulate sensory data that would be much too risky to obtain using real hardware.

The Neurosim architecture mirrors a typical autonomy

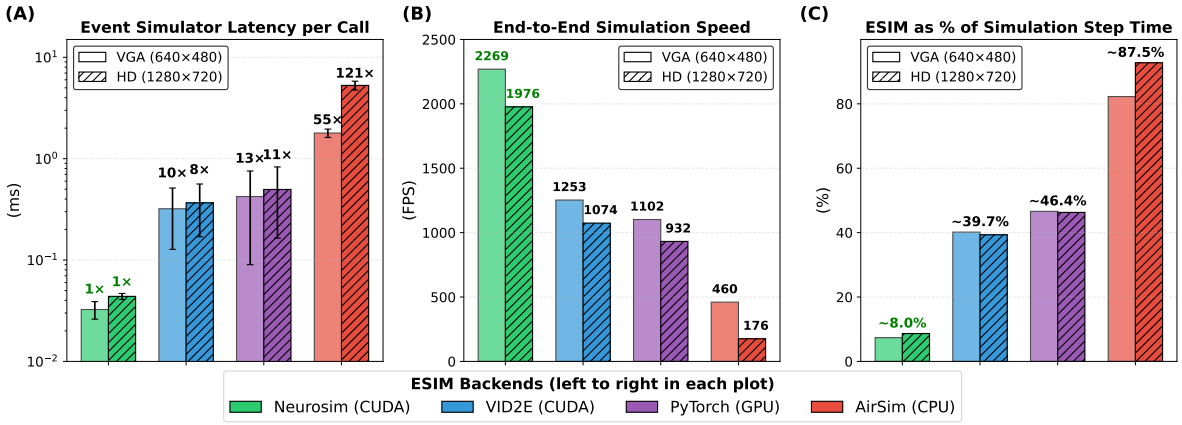

Figure 3. **Neurosim event simulator backend is optimized to prevent bottlenecking of the full simulation pipeline.** We benchmark Neurosim's event camera simulator against existing simulators: a CUDA-based simulator [16], a GPU-based PyTorch implementation, and a CPU-based implementation [20]. **A.** We measure the average latency to simulate events from a single intensity image update – the time taken to process a new intensity frame, trigger events based on the contrast threshold model, update the internal pixel states, and output the event stream. Neurosim achieves over 31 kHz for VGA and 23 kHz for HD frame sizes. It is roughly 8–13 × faster than other GPU-based implementations, and 55–121 × faster than CPU-based implementations. **B.** Neurosim using the custom event simulator backend achieves ∼2300 FPS for a full simulation step (including rendering, dynamics, and event simulation) with VGA vision sensors. Other event simulation backends bottleneck the full simulation pipeline, achieving only 200–1200 FPS. **C.** Neurosim's event simulator only takes up ∼ 8 % of the total simulation step time compared to 40–90 % for other event simulators. This high performance is attributed to the warp-synchronous CUDA kernel design that minimizes atomic operations during event aggregation and allows our event generation model to run in a single kernel launch.

stack that interfaces with a hardware platform. Different processes consume sensor observations, compute upon them using their respective perception, planning, or control pipelines, and publish control commands. The simulator ingests these commands just like a real multirotor would. There are two modes of operation. In the "closed-loop mode", Neurosim runs in lockstep with other processes that subscribe to sensor streams, perform computation, and publish control commands that immediately affect the simulated vehicle. Example applications of such synchronous execution include evaluating event-based perception and control algorithms or reinforcement learning event-based control policies. In the "streaming mode", Neurosim behaves as if a platform were transmitting real-time data for online training. Sec. 4 demonstrates an example application of asynchronous simulation. The simulator publishes observations that are used directly for online training of multi-modal foundation models without any intermediate disk storage. Sec. 3 discusses how high-frequency and high-throughput perception and control data (e.g., Python data structures, NumPy, and PyTorch arrays) for both synchronous and asynchronous operation are communicated using a publish-subscribe framework called Cortex.

3. The Cortex Communication Interface

Cortex is a high-performance communication library for robotics and machine learning applications that is designed to work together with Neurosim. It connects the simulator to different kinds of downstream data consumers and connects the controllers back to the simulator. Cortex is designed to be a lightweight, low-latency and compatible alternative to middleware like ROS [11], while providing a more robotics-friendly abstraction than raw messaging libraries like ZeroMQ [14] or plain Unix sockets.

The core philosophy behind Cortex is as follows. First, modern CPUs and memory subsystems have evolved dramatically in recent years. For example, consumer-grade desktop AMD processors have more than 128 MB L3 cache memory, clock speeds upwards of 4 GHz, and more than 16 cores. This makes low-latency, zero-copy inter-process communication (IPC) extremely efficient, without having to perform frequent memory accesses. Second, there are a number of performant low-level C/C++ based libraries that utilize these hardware advancements, e.g., ZeroMQ or NanoMsg. They also come with efficient Python bindings. Python 3.14 introduces free-threading support by removing the Global Interpreter Lock (GIL) that has historically hindered multi-threaded parallelism. So Python interpretation is no longer the bottleneck for low-latency, high-throughput inter-process communication. Coupled with new techniques like just-in-time (JIT) compilation, Python is becoming viable for high-performance scripting.

Feeding high-throughput multi-modal robot sensory data to deep learning training pipelines. As its transport layer, Cortex uses ZeroMQ [14], a high-performance asynchronous messaging library. This is a robust and efficient

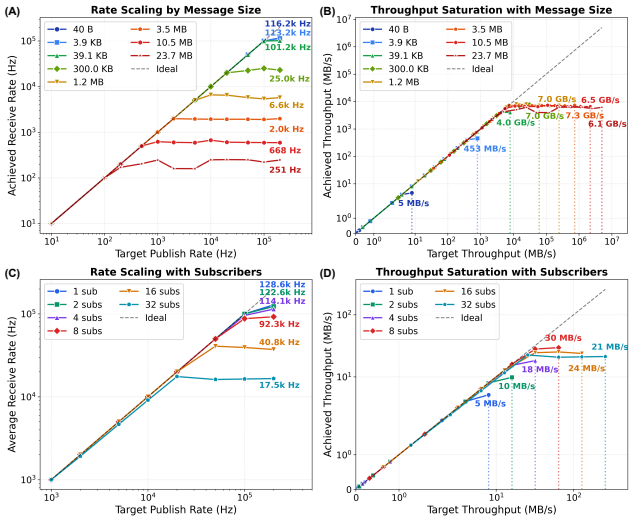

Figure 4. **Cortex provides high-throughput, low-latency, and scalable communication in Python.** We benchmark different aspects of Cortex's performance on a modern desktop CPU (AMD Ryzen 9 7950X). **A.** We use NumPy float (32 bit) arrays of different sizes ranging from 10 elements (40 B) to 1080×1920 RGB images (23.7 MB) as our message payloads. We measure the max achieved successful publish-subscribe rate (in Hz) for each payload size over a 15 s interval. Cortex achieves 100+ kHz for small (< 39 kB) messages and maintains sufficiently high rates (250 Hz) for large 1080p RGB images. **B.** We measure the achieved throughput (in MB s^{-1}) for the same payload sizes. Cortex achieves up to 7 GB s^{-1} throughput for large messages, saturating the underlying hardware limits. For smaller messages, the throughput is limited by per-message overheads – the messaging rate caps at ∼100 kHz. **C.** We test the scalability of Cortex by varying the number of subscribers from 1 to 32, each receiving 40 B messages from a single publisher. The achieved message rate per subscriber remains constant at around 100 kHz as the number of subscribers increases up to 8, after which it degrades due to CPU contention from many busy-waiting subscribers on the same machine. **D.** We measure the achieved throughput (in MB s^{-1}) for the same setup as in C. As deduced from the message rates, the throughput scales roughly linearly with the number of subscribers until CPU contention effects kick in beyond 8 subscribers.

communication substrate, supporting various patterns (e.g., publish-subscribe, request-reply) and protocols (e.g., TCP, IPC, PGM). A key feature of Cortex is dynamic node discovery. A lightweight discovery daemon runs in the background and maintains a registry of all active nodes and their advertised topics. Nodes register themselves with the daemon upon startup and query it to find other nodes and topics they wish to communicate with. This allows for dynamic and flexible network topologies without a centralized broker or hardcoded addresses. Communication between processes remains peer-to-peer in nature; this is critical to minimizing latency and single points of failure. Processes can use different transports depending on their use cases, and multiple subscribers can listen to the same topic seamlessly.

We benchmarked Cortex on a modern desktop CPU (AMD Ryzen 9 7950X) in Fig. 4 to find that it achieves message rates, throughput, and scalability that is suitable for high-bandwidth robotics and machine learning applications.

Cortex is designed with native support for libraries like NumPy and PyTorch. This allows efficient serialization and deserialization of complex data structures such as multi-dimensional arrays and tensors. Cortex employs zero-copy techniques for these messages with exposed memory buffers, as long as either the producer or consumer ensures memory safety. Inspired by the philosophy of LCM [8], Cortex provides a simple messaging API with strong type checking. Users define custom message types using Python dataclasses with NumPy and PyTorch support, rather than being restricted to slower alternatives like Python lists. The library handles the details of serialization, deserialization, and transport. For type safety, Cortex computes a unique 64-bit hash for each message type based on its structure, fields, and data types. This hash is included in the message header, allowing receivers to verify that they are receiving the expected message type, catch mismatches early, and dynamically load/cache deserialization logic for new message types at runtime. This ensures strong type safety and helps prevent runtime errors due to incompatible message formats.

A majority of robotic autonomy code is implemented within the ROS ecosystem today. Cortex provides a performant bidirectional bridge to ROS 2. The bridge allows researchers and developers to use Cortex as a lightweight communication layer for high-bandwidth machine learning data streams while still enjoying the extensive support of the ROS 2 ecosystem. In the context of Neurosim, this bridge allows users to use ROS 2 workflows, say for controllers and state estimators, seamlessly, while benefiting from Cortex-based communication in Neurosim behind the scenes. For example, using this Cortex-based interface, one can use Rerun [17] while simultaneously using RViz in ROS 2 [11] for visualizing some user-specific code artifacts– both receiving data and synchronized against the same underlying Neurosim instance.

Cortex also facilitates efficient data loading for downstream applications by removing disk I/O bottlenecks. Neurosim behaves like a robot platform that sends real-time rendered sensory data over Cortex – broadcasting multi-modal sensor streams like RGB, depth, events, IMU, and state estimates. Potential consumers of this data stream are deep learning training pipelines that ingest the data for training. Data subscribers operate in separate processes, utilizing zero-copy deserialization to receive data using Cortex and push it into multiprocessing queues. Online data loaders consume these queues and assemble the data into pre-allocated PyTorch tensors. This architecture handles modality-specific batching (such as aggregating variable-

length event streams), prefetching of data batches, and multi-GPU distributed training. Cortex-based online data loading is significantly more scalable than disk-based loading. It allows a single simulation node to saturate the ingestion bandwidth of multiple training GPUs and removes the logistical burden of managing terabytes of static simulated data.

4. Applications

The combination of Neurosim and Cortex enables a range of applications that were previously impractical due to limitations of existing tools. We next present three kinds of experiments that demonstrate the capabilities of our framework.

Real-time, closed-loop control. Neurosim facilitates the development of end-to-end control algorithms in a real-time and closed-loop setting using asynchronous simulator and controller processes connected via Cortex. As illustrated in Fig. 2, the simulator broadcasts multi-modal sensor data, which are consumed by a controller to generate actuation commands, such as individual rotor speeds. In this mode of operation, the latency of control algorithms plays an important role in closed-loop performance. Slow controllers and communication bottlenecks can be simulated, and the effect of such limitations can be studied precisely. By default, the simulator performs a zeroth-order hold on the most recent control command if new commands are not received in time. This is useful to evaluate algorithms on different hardware platforms, e.g., a simulator node can run on a powerful desktop GPU, while the controller can be tested on actual robot hardware such as NVIDIA Jetson Orin. This setup is particularly valuable for debugging and testing reactive algorithms, such as event-based collision avoidance or high-speed visual servoing, where controllers must respond to environmental changes with millisecond precision. In Fig. 6 (A), we analyze the control-to-simulation latency and achievable sensor rates for a closed-loop controller receiving RGB images from Neurosim at different controller rates up to 5 kHz. In this analysis, both the controller and simulator nodes are running on the same machine and communicate over Cortex using IPC. These results indicate that Cortex can sustain low-latency communication (< 0.7 ms) and high sensor rates (~ 2.3 kHz), making it suitable for real-time closed-loop control applications. In effect, this analysis shows that Neurosim supports evaluation and development of real-time closed-loop control at more than 1500 Hz.

The modular design of Neurosim, enabled by Cortex, allows for flexibility in the control stack. The simulator is agnostic to the controller's implementation. One can prototype in Python, deploy high-performance C++ implementations, or interface with existing ROS 2 navigation stacks. This extensibility can support a wide range of use cases, from benchmarking end-to-end learning-based policies to performing safety analysis and regression testing of control algorithms.

Online training of multi-modal (including event-based) perception models. We demonstrate the utility of Neurosim by training an event-based monocular depth estimation network using the synchronized data generated in simulation. This network uses Fast Feature Field (F^3) as the input representation that was developed by Das et al. [3]. In short, given a set $e^- = \{(s,u) : e(s,u) \in \{+1,-1\}$ for $s \in [t-\Delta t, t), u \in \Omega\}$ that consists of events of positive or negative polarity within a time horizon $[t-\Delta t, t)$ and the image plane $\Omega \in \mathbb{Z}^2$, the representation of these events $F^3(t,u) \in \mathbb{R}^p$ is sufficient to reconstruct future events e^+ that correspond to times $s \in [t, t+\Delta t)$. Das et al. developed a neural architecture to compute F^3 efficiently (at 120 Hz on HD frames and 440 Hz on VGA frames). They showed state-of-the-art performance on downstream tasks ranging from optical flow, stereo depth, monocular depth estimation, and semantic segmentation being performed at 25–75 Hz on HD frames using this representation.

We procedurally generated diverse minimum-snap [12] trajectories through complex indoor environments, incorporating a wide variety of yaw profiles and flight speeds. These trajectories are dynamically feasible for the quadrotor platform of choice—accounting for mass, inertia, motor profiles, and aerodynamics—tracked by an SE(3) controller which is provided state estimates at 100+ Hz. The training pipeline consumes this stream via the online data loader discussed in Sec. 3. It receives time-synchronized event streams from a simulated event camera along with ground-truth depth maps. In addition to the benefits mentioned earlier, this decoupled approach allows us to easily modify the training inputs—scenes, trajectories, sensor parameters—on-the-fly without having to re-generate and store large datasets on the hard disk. Using a visualizer node from Cortex, one can also monitor the simulation in real-time while training is ongoing. We show some example trajectories and the corresponding sensor data in Fig. 5. These trajectories demonstrate fast flights across multiple floors, corridors, and interconnected rooms, with angular and linear velocities often exceeding 4 rad s^{-1} and 2 m s^{-1}, respectively.

Following [3], we train an RGB-based monocular depth prediction network (Depth Anything V2 [22]) that is modified to take F^3 as input. Training details are provided in the Supp. Sec. 6. Fig. 6 (B) shows a qualitative example of the data (trajectory, events, RGB, and depth) generated from Neurosim and the depth predictions from the trained network. The network is trained completely online. Each training step consumes a batch of data generated from Neurosim on-the-fly without any intermediate disk storage.

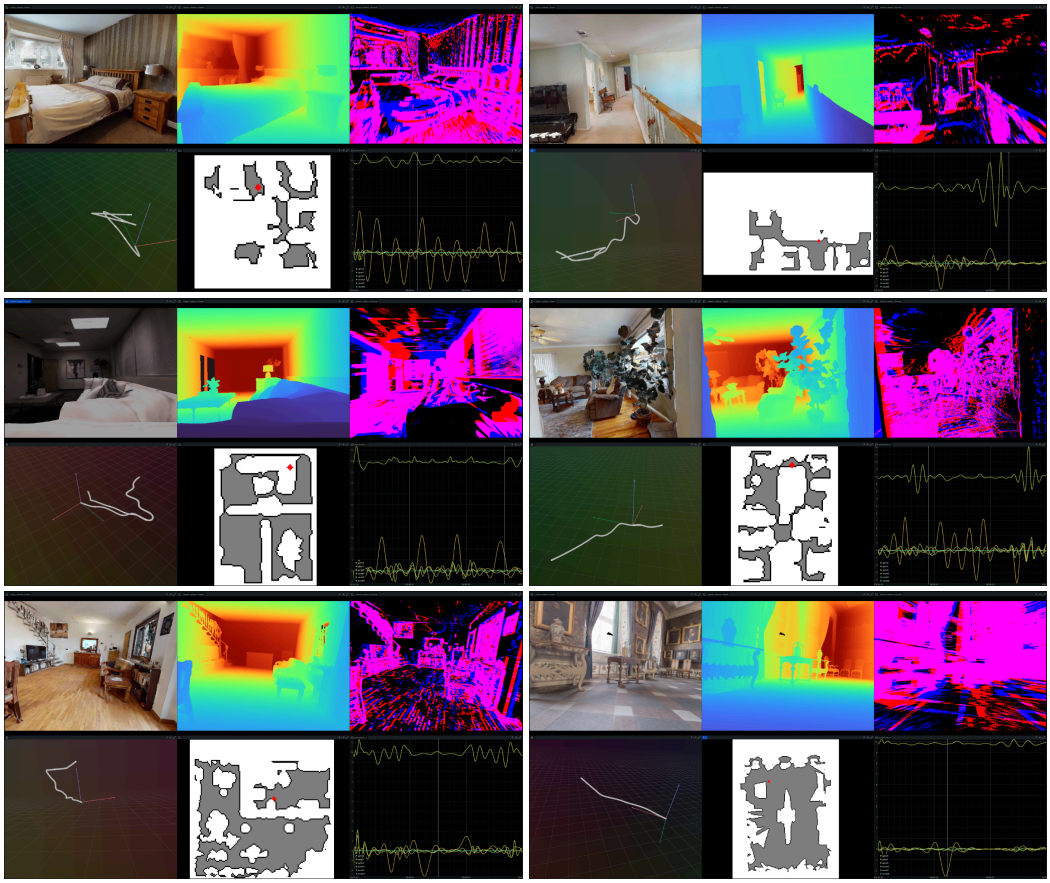

Figure 5. **Examples of a quadrotor tracking randomly generated MinSnap trajectories in a variety of indoor scenes.** In each example, from left to right, we plot the color image, depth image (near to far is colored as dark blue to red), and events aggregated with polarity over 20 ms, on the top. 3D trajectory with current pose, navigation mesh with current position, and IMU readings are plotted at the bottom. The navigation mesh marks free space and non-reachable locations at the current flying height with gray and white, respectively. These scenes demonstrate fast flights across multiple floors, corridors, and interconnected rooms, with angular and linear velocities often exceeding 4 rad s^{-1} and 2 m s^{-1}, respectively.

For context, the depth prediction model roughly consumes 8.5 M events per second, which would require $\sim$ 450 GB of disk storage per hour if it was to be stored. Neurosim runs on a single Nvidia 4090 GPU roughly consuming 1.5 GB of VRAM and 120 W of power. The data generation rate is enough to saturate the training pipeline for an F^3-Depth Anything V2 "small" model running on the same machine on a second GPU. This experiment demonstrates that Neurosim and Cortex can facilitate the training of foundation models for robot perception completely online on low-latency, high-throughput data.

Event-based Reinforcement Learning. Neurosim natively supports training event-based reinforcement learning (RL) policies completely online, enabling multiple concurrent simulation instances to produce high-throughput rollout data on a single GPU. Neurosim RL lowers the barrier of using multi-modal input, custom reward functions, domain randomization and collision checking, providing a modular interface to train both on and off-policy RL methods. To demonstrate an on-policy training capability, we train a continuous-control policy via Proximal Policy Optimization (PPO) [19] for a "stabilize-hover" task. In this environment, a quadrotor is initialized with random linear and angular velocities. Relying solely on event-camera observations and state estimates, the vehicle must learn control actions to decelerate and stabilize to a hover without colliding with the scene. Both the structure and motion from event streams are critical to stabilize and not crash the quadrotor into obstables – making the task particularly challenging and interesting for dynamic vision sensors. The reward function is designed to penalize the magnitude of linear and angular velocities to encourage deceleration, while regularizing action smoothness for smoother control. Additionally, a flat-bonus survival reward is provided at each timestep to encourage the agent to avoid crashing and learn from longer trajectories. Further algorithmic setup, reward specifications, and PPO hyperparameter details are provided in the

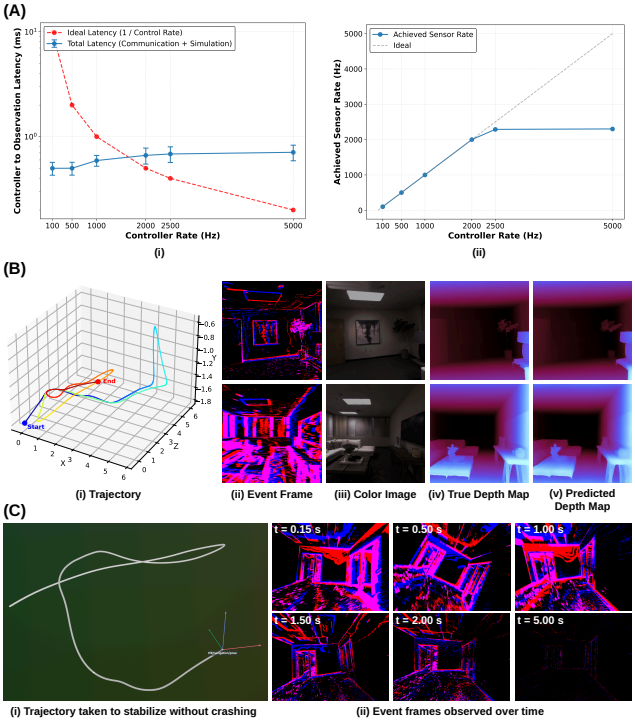

Figure 6. **Neurosim can be used for closed-loop controller testing, online data generation for learning and on-policy reinforcement learning.** **(A)** shows a latency and frequency analysis of Neurosim in typical indoor flying scenes for closed-loop control. The simulator runs at 10 kHz, performing dynamics updates. The controller and simulator nodes work separately and communicate asynchronously using Cortex. We vary the controller rate (same as the sensor rendering rate) from 100 Hz to 5 kHz to measure the effects of communication-simulation latencies and bandwidth saturation. We use a VGA (640 $\times$ 480) RGB camera as our sensor, that is, each sensor observation is $\sim$ 0.9 MB in size. **(i)** shows the average end-to-end latency experienced by the controller – the time taken from sending a control command to receiving the next sensor observation after the command has been applied. This latency captures the communication overhead as well as the simulator processing time. As the controller and target sensor rate increase, the communication latency increases slightly due to increased IPC bandwidth contention. However, the overall latency remains below 0.7 ms even at 5 kHz controller rates. We also plot the ideal latency at a particular control rate, that is, the time taken to take control in real-time. Any control rate achieving a controller-to-observation latency less than the ideal latency is guaranteed to receive the latest sensor observations at the control rate. **(ii)** shows the sensor observation rate received by the controller vs the desired sensor rate. The controller is able to receive the VGA color sensor readings at the target rate up to 2.3 kHz, after which it plateaus. This plateau is the current limit of the simulator's rendering speed on a single Nvidia 4090 GPU, combined with the communication overhead of Cortex. **(B)** shows qualitative examples of learning event-based monocular depth. **(i)** shows an example MinSnap trajectory sampled in a complex indoor environment, avoiding obstacles. For visualization, the trajectory is colored based on time, from blue (start) to red (end). Figure 6. **(B) (ii)** shows 20 ms of events plotted onto a frame and colored based on their polarity. The rendered events depend on the underlying motion of the robot – tracking randomly sampled trajectories while respecting the vehicle dynamics and controllers. **(iii)** and **(iv)** show the color and depth image captured at the last event timing, respectively. **(v)** shows the depth image predicted from the events plotted in (ii). The predictor – an F^3-based monocular depth model – is trained online using the data generated from Neurosim. **(C)** shows an example rollout using a policy learnt to stabilize and not crash a quadrotor when thrown with a random initial and angular velocity in a cluttered indoor environment in Neurosim. Here, the quadrotor is initialized with a linear velocity and yaw rate of 2 $\mathrm{m\,s^{-1}}$ and 1 $\mathrm{rad\,s^{-1}}$, respectively. **(i)** shows the trajectory taken by the trained policy to decelerate, stabilize, and avoid crashing into the walls. **(ii)** shows 20 ms event frames colored by polarity at different time instances. It is clearly visible that the event rates reduce over time. In this scenario, ego-motion of the quadrotor is the only source of intensity change in the camera frame, so as the vehicle decelerates and stabilizes, the event rates reduce significantly.

Supp. Sec. 7. Fig. 6 (C) shows an example rollout of the trained policy. The policy is able to learn a trajectory that decelerates and stabilizes the quadrotor from a random initial velocities to hover without crashing in a cluttered indoor environment.

5. Discussion

Neurosim and Cortex open up avenues to build and test new neuromorphic algorithms in simulation. Multi-kilohertz event simulation and high-throughput multi-modal sensing with accurate calibration and scene ground-truth enable key aspects of training and evaluation in embodied autonomy. This supports (i) streaming data directly into learning and control pipelines without disk I/O; (ii) privileged supervision (e.g., metric depth, pose, and time-aligned cross-modal views) at high rates for online self-supervised learning and reinforcement learning that scales with environment diversity rather than dataset storage; (iii) closed-loop experiments at the limits of hardware performance for high-speed perception and control; and (iv) controlled studies of sensor timing, from synchronized streams to realistic clock skew and asynchronous sampling.

Several directions remain open to improvements. First, we plan to add hardware-in-the-loop experiments where an edge device (e.g., an embedded controller or perception pipeline on Jetson hardware) consumes simulated sensors and returns control at realistic latencies. This will expose timing, scheduling, and bandwidth constraints that are often abstracted away in typical evaluations. Second, we aim to support Unreal Engine 5 [5] as an additional rendering backend to broaden asset compatibility and photorealism, while preserving the high-rate event generation pathway and maintaining determinism for reproducible learning.

References

[1] Kenneth Chaney, Fernando Cladera, Ziyun Wang, Anthony Bisulco, M. Ani Hsieh, Christopher Korpela, Vijay Kumar, Camillo J. Taylor, and Kostas Daniilidis. M3ed: Multi-robot, multi-sensor, multi-environment event dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 4015–4022, 2023. 1

[2] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning, 2016. 2

[3] Richeek Das, Kostas Daniilidis, and Pratik Chaudhari. Fast feature field (F^3): A predictive representation of events. *arXiv preprint*, 2025. arXiv:2509.25146. 2, 6, 1

[4] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017. 2, 3

[5] Epic Games. Unreal engine. https://www.unrealengine.com. Technical Report. 8

[6] Spencer Folk, James Paulos, and Vijay Kumar. RotorPy: A python-based multirotor simulator with aerodynamics for education and research. *arXiv preprint arXiv:2306.04485*, 2023. 3, 1

[7] Guillermo Gallego, Tobi Delbrück, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew J. Davison, Jörg Conradt, Kostas Daniilidis, and Davide Scaramuzza. Event-based vision: A survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 44(1):154–180, 2022. 2

[8] Albert S. Huang, Edwin Olson, and David C. Moore. Lcm: Lightweight communications and marshalling. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4057–4062, 2010. 5

[9] Simon Klenk, Marvin Motzet, Lukas Koestler, and Daniel Cremers. Deep event visual odometry. In *International Conference on 3D Vision, 3DV 2024, Davos, Switzerland, March 18-21, 2024*, pages 739–749. IEEE, 2024. 2

[10] Xavier Lagorce, Garrick Orchard, Francesco Galluppi, Bertram E. Shi, and Ryad B. Benosman. Hots: A hierarchy of event-based time-surfaces for pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(7):1346–1359, 2017. 1

[11] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. 4, 5

[12] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525, 2011. 6

[13] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, page 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[14] The ZeroMQ Project. Zeromq: An open-source universal messaging library. https://github.com/zeromq. 4

[15] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. 1

[16] Henri Rebecq, Daniel Gehrig, and Davide Scaramuzza. Esim: an open event camera simulator. In *Proceedings of The 2nd Conference on Robot Learning*, pages 969–982. PMLR, 2018. 2, 3, 4

[17] Rerun Development Team. Rerun: A visualization sdk for multimodal data, 2024. Available from https://www.rerun.io/ and https://github.com/rerun-io/rerun. 3, 5

[18] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019. 2

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. 7, 1

[20] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017. 2, 3, 4

[21] Wenshan Wang, Delong Zhu, Xiangwei Wang, Yaoyu Hu, Yuheng Qiu, Chen Wang, Yafei Hu, Ashish Kapoor, and Sebastian Scherer. Tartanair: A dataset to push the limits of visual slam. 2020. 2

[22] Lihe Yang, Bingyi Kang, Zilong Huang, Zhen Zhao, Xiaogang Xu, Jiashi Feng, and Hengshuang Zhao. Depth anything v2. In *Advances in Neural Information Processing Systems*, pages 21875–21911. Curran Associates, Inc., 2024. 6, 1