Heron: A Knowledge Graph editor for intuitive implementation of python based experimental pipelines

George Dimitriadis[1,✉], Ella Svahn[1,2], Andrew MacAskill[2], and Athena Akrami[1]

[1]Sainsbury Wellcome Centre, University College London, London, UK
[2]Department of Neuroscience, Physiology and Pharmacology, University College London, London, UK

To realise a research project idea, an experimenter faces a series of conflicting design and implementation considerations, regarding both its hardware and software components. For instance, the ease of implementation, in time and expertise, should be balanced against the ease of future reconfigurability and number of 'black box' components. Other, often conflicting, considerations include the level of documentation and ease of reproducibility, resource availability as well as access to online communities. To alleviate this balancing act between opposing requirements we present Heron, a new Python-based platform to construct and run experimental and data analysis pipelines. Heron's main principle is to allow researchers to design and implement the experimental flow as close as possible to their mental schemata of the experiment, in the form of a Knowledge Graph. Heron is designed to increase the implementation speed of experiments (and their subsequent updates), while minimising the number of incorporated black box components. It enhances the readability and reproducibility of the final implementation and allows the use of combinations of hardware and software otherwise impossible or too costly to achieve. Given this, Heron offers itself to sciences whose needs involve experiments with a large number of interconnected hardware and software components like robotics, neuroscience, behavioural sciences, physics, chemistry, environmental science, etc.. It is designed with those experimentalists in mind which: i) Demand full control of their setup. ii) Prefer not to have to choose between hardware and software that run only on a specific chip/operating system combination. iii) Appreciate the ease and speed that high-level languages (e.g. Python) and Graphical User Interfaces (GUIs) offer them. It assumes an intermediate knowledge of the Python language and ecosystem, offering a flexible and powerful way to construct experimental setups. It removes any inaccessible corners, yet keeps implementation costs significantly reduced compared to using lower level languages. Finally, its use results in a much cleaner and easier to understand code base, amicable to documentation and reproducibility efforts.

Open source, experiment, software
Correspondence: ***g.dimitriadis@ucl.ac.uk***

Introduction

One can divide an experiment's life circle, from concept to a running system, into a number of transformations. At first, the scientific question is mapped into an abstract schema of experimental steps (i.e. what needs to happen in order to answer the question). Subsequently these conceptual steps are transformed into a schema of hardware connectivity and software logic. At this stage the experimenter thinks in terms of high level objects like cameras and other types of sensors, time lines, triggers and events, agents and rewards, inputs and outputs. The final challenging step, which is rarely addressed (or even cognitively acknowledged) is to map the schema of the hardware and software logic to the actual hardware connectivity and operational code bases. At this level the experimenter has to work with much lower level objects like voltage differences, light intensities, TTL pulses, GPU shaders and information flow loops. This last and most time consuming step, can limit the number of iterations for ideas to be piloted and tested. Once the mental schema has been translated into code, it is this code that is usually addressed by other experimentalists in reproducibility efforts. These efforts usually face a high barrier in understanding the original experimental schema starting from its code implementation. This is because efficiently translating many lines of code (even if well documented) back to what the code actually accomplishes is in most cases a difficult task that requires years of coding experience, as well as familiarity with experimental designs. This barrier hinders efforts of reproducibility and quality control and is one that cannot be addressed solely by open sourcing one's work. Finally, the complexity of the translation from mental schema to running code-base makes design iteration efforts practically impossible. The prohibitively large iteration time, on one hand, and the inter-dependency of engineering decisions throughout the implementation cycle, on the other, makes updates of the experimental flow extremely cumbersome. This often results in practically one of the most serious experiment design and implementation hurdles. The need to create radically new implementations for only small changes in the underlying mental schema. These arguments can be seen as the main driver for the development and the rapid acceptance of high-level languages in software engineering and of micro-controller kits (e.g. Arduino) in electrical engineering and robotics. Concepts like object oriented programming (1) or actor based programming (2) for example, have been nothing else but an effort to take away the low level concepts that one needs to ultimately manipulate, and replace them with bundles of higher level ones, easier to cognise with.

In order to address this discrepancy between the mental schemata and implementation outcomes in experimental con-

struction, we developed Heron, an open source software (MIT license) platform for the construction of data flow pipelines (e.g. experiments, data analysis, robotics, etc.). Heron comes with a series of distinguishing features. The primary one is that it creates experimental pipelines that visually and structurally, bear a very significant resemblance to the original mental schema of the experimental pipeline. So what one gets as the final experiment implementation is both semantically and syntactically very close to how one originally envisions the experiment should work. Because of that, Heron creates final implementations that are easy to understand, construct, communicate and change. In this way it often makes it fairly easy to put together helpful diagrams and documentation by following Heron's visual representation of the experiment and its underlying code. It also allows for accurate inference of the real time complexity of any proposed change even before any new code is written.

A second distinguishing feature of Heron is that it targets the experimenter without expertise in arcane subjects like networking, hardware connectivity or low level software - hardware interactions. By abstracting away these low level features in its Graphical User Interface (GUI) it allows construction of experiments with multiple, diverse hardware components, even using networks of computers running different operating systems, without bothering the experimenter with having to deal with all the low level connectivity issues that arise. It achieves this without limiting its users to preconceived ideas of how any single specific piece of hardware or code should be used. This is possible as Heron offers users an Application Programming Interface - API - letting them write the code that implements their own ideas at the optimal level of abstraction given the situation. Offering Python as the main (but not only) language for this user-centric code implementation, Heron makes it even easier for code novices to achieve highly complex experimental setups that are easy to both construct and reconfigure. In the following section, we will focus on Heron as a general purpose tool for constructing pipelines used to conduct different types of experiments. First, We will describe the specific meaning of an experimental pipeline implemented as a Knowledge Graph (7). We then catalogue the design benefits and distinguishing features offered by Heron, in comparison to other efforts targeting the construction of experiments. Subsequently, we will describe the internal architecture of Heron with enough details to allow any developer to quickly get up to speed with Heron's code and contribute to its open source. Finally, we will illustrate a number of Heron experimental implementations, currently in use in the lab, each showcasing one of Heron's special features.

Concepts and Background

Mental schemata, Knowledge Graphs and Heron processes. The philosophy behind Heron's design. A mental schema is a psychological concept (5), (6) that is meant to define the way humans cognise. According to it, when people think, they categorise sensory experiences and abstract notions into groups. They then utilize the relations between these groups to draw conclusions about some hypotheses. Upon inputs from the environment and prior cognitive outcomes, the categories and their relations can update fluidly (6). An example, for the case of interest here, is the mental schema of an experiment. In order for an experiment to be developed, the experimenter brings together a set of categories, both based on their sensory experiences (e.g. a laser, a data acquisition board, or a camera) and on abstract notions (e.g. time concurrency, or subject's choices). Then a set of relationships is generated between them. The sets of these concepts with their interactions defines the mental schema of the specific experiment. For example, a camera frame must be captured immediately after a specific event has taken place. Yet, today's implementations of experimental pipelines are written (with visual or text-based code) such that they obfuscate the mental schemata they derive from. A receiver of such a code base, irrespective of their understanding of the underlying language, always needs a significant amount of time and mental effort to map back the initial mental schema. A Knowledge Graph (KG) (7) is a mathematical structure designed to capture the unstructured human knowledge on a subject (i.e. the mental schemata of different individuals relating to the same knowledge corpus) in such a way that a machine could use it to test propositions against the knowledge and also create novel propositions that a human with the knowledge would find to be true. It is practically an effort to implement the fuzzy notion of a mental schema into a concrete structure of objects and relations that is machine implementable. The fundamental structures of a KG are its nodes and their attributes. Nodes are meant to represent a group of objects at a desired level of abstraction, which does not have to be uniform amongst the different nodes of the same KG. Nodes' attributes define the state of each node and the edges that connect different nodes representing their relationships. Nodes and their attributes usually have semantic labels.

Heron implements an experiment in the form of a Knowledge Graph and does so at two separate levels. One is the graphical level, where a series of Nodes and their in-between links (Edges) are defined. The second is the code that defines each Node's functionality. The graphical level is used to construct a Knowledge Graph representing the experiment. The text-based code level is used to implement the dynamics of each Node in the Graph and define its level of abstraction. The Knowledge Graph's Nodes are labelled and have human readable attributes partially defining their state. They are also connected with directed Edges (links) through their named input and output points. The Edges represent message passing between Nodes (see Figure 1). In this way a graphical representation of an experimenter's mental schema is created with each Node at the appropriate level of abstraction for the specific experiment. This individualised level of abstraction is achieved through Heron's second level of implementation, i.e. a text based code that defines each Node's functionality. This is an important distinction between Heron and other node-based software like LabView and Bonsai (see Background section). Heron expects that

each Node's behaviour and connectivity is defined by the experimenter, in normal, text-based code, and comes with an appropriate API to facilitate this. In this way Heron does not offer a long list of predefined Nodes which would make very strong assumptions about the structure of the experiment's mental schema. Instead, it offers the tools for designing and implementing one's own Nodes in a case by case approach at a level of abstraction required at any time. For instance, if in one case a Node representing a camera is required, while in another a Node should represent a large group of synchronised cameras acting as one, Heron provides the tooling to create either of these Nodes with minimum effort. Heron's code base is implemented in Python. This choice was informed by the need to keep the code simple to implement while at the same time be able to interact with a very diverse list of hardware drivers and software analysis libraries (see Design benefits section). It also means that Python is the go to language for the user's creation of new Node behaviours. Yet the language that Heron itself is implemented in (Python) does not enforce the language new Nodes can be implemented in by Heron users. Heron offers users a comfortable starting point to implement their own Nodes (in Python), while allowing for expert users to utilise lower level languages that offer other advantages like faster run times and lower level control of a machine's components (e.g. CPUs, GPUs, RAM, etc.). Heron poses very few (if any) limitations in what types of behaviour can be implemented in any Node. Breaking up the implementation into two levels of Knowledge Graph and running codes, as mentioned above, confers a much required "break complex problems into simpler ones" functionality at the heart of Heron's operation.

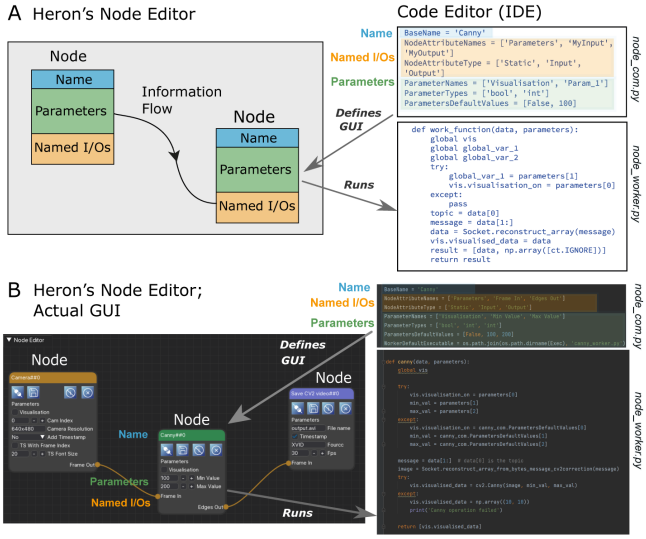

Fig. 1. A. Schematic of Heron's two levels of operation. Left shows a Knowledge Graph made of Nodes with Names, named Parameters and named Input/Output points. Nodes are connected with links (Edges) that define the flow of information. Right shows a piece of textual code that fully defines both the Node's GUI appearance and the processes run by the machine. B. Heron's implementation of A, where the Knowledge Graph is generated in Heron's Node Editor window while the code that each Node needs to be defined and the code that each Node is running is written in a separate Integrated Development Environment (IDE, in this case PyCharm).

Background.

Computational graphs. Heron's Knowledge Graph approach has its root in a number of software frameworks where experimental and data analysis pipelines are constructed as a computational graph. MATLAB Simulink (18) and LabVIEW (17) are two of the more well-known frameworks where experiments can be designed as computational graphs using pre-defined elements (i.e. nodes provided by the developers of these systems). Bonsai (4) is a new entry to this field originally designed to ease the implementation of neuroscience and behavioural science experiments. The use of directed acyclic graphs (DAGs) (19) has increased dramatically in the Big Data analytics tools where frameworks like Apache Airflow (20) and Dask (21) allow parallelisation of algorithms and data queries over large clusters of machines. Two efforts that are very similar to Heron in the way they structure computational graphs to define experimental pipelines are EPypes (8) and the Robot Operating System (ROS) (9). EPypes is (according to the developers own description) a "Python-based software framework for developing vision algorithms in a form of computational graphs and their integration with distributed systems based on publish-subscribe communication". The basic idea of message passing between individual processes, each responsible for its own algorithm, running on different machines is identical to Heron (even at the level of using the publish-subscribe communication protocol), although EPypes's focus is on computer vision algorithms. The exact same idea is utilised by ROS where (in their own words again) "The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server."

Hybrid programming. Heron's approach is based on an existing programming idea, i.e. combining both visual and textual programming in a hybrid manner. One of the most widely used examples of such realisations is VVVV (10), a framework utilising visual programming with the C# or HLSL programming languages for textual programming. This hybrid approach was found to allow for a better retention of computer software university students, when comparing only textual or only visual styles in the learning of programming (11), showing that it better suits beginner level programmers (a category into which a large percentage of experimentalists fall in).

Behavioural sciences toolboxes. Heron was originally conceptualised to be a framework for creating experiments in the fields of behavioural sciences (e.g. neuroscience, experimental psychology, etc.) and although its philosophy and use cases span a much wider spectrum, its current usage derives from experiments in this field. Other frameworks

that specifically target the same fields are pyControl (12), Bpod (Sanworks LLC, USA) (building on the central design concept of B-control (22)) and Autopilot (13). Bpod and pyControl are software-dedicated hardware efforts while Autopilot is a software framework that in the same spirit of EPypes, ROS and Heron allows a distributed experimental pipeline albeit restricting the machines to Raspberry Pi computers. All three efforts pay special attention to offering their users tools for creating state machines to define their experiments (each utilising its own way of doing so). Heron currently allows the users to decide if their pipeline would benefit from a state machine design or not and being Python based allows for the use of a plethora of state machine tools in the Python ecosystem (23, 24). This includes the capability to script Nodes that can wrap the Python APIs of pyControl and Bpod (through the Champalimaud Foundation's pybpod API). This can be of interest to those experimentalists who have invested in the respective hardware modules but would like to expand the capabilities of their pipelines beyond the reach of pyControl or Bpod.

Design benefits.

Self-documentation. The Knowledge Graph of Heron immediately offers a succinct overview of the experimental workflow and the dynamics it implements, thus acting also as the primary documentation of the experiment. Armed with a coherent picture of the experiment's information flow over time, one can accesses the code of individual Nodes, for a deeper understanding of its details. Grasping the meaning of a few hundred of lines of Python code that most Nodes require to be implemented, one Node at a time, is a much more appealing proposition than opening up a whole code base of a non-Heron experiment and be faced with thousands of lines of obscurely interconnected code arranged in a file system that only makes sense to the developer (and only for a short while after the code's implementation). Moreover, Python code, in comparison to other lower level languages, helps with readability in a self-documenting fashion (notwithstanding the plethora of in code documentation tools in the Python ecosystem). This self-documentation capability of Heron's experimental implementations confers obvious benefits to the exchange and reproducibility of experiments and minimises the possibility of misunderstandings when researchers from different groups try to interact with the experiment.

Multiple machines, operating systems and environments. In Heron, each Node runs its own process (practically its own little program, separate from all the programs of the other Nodes). This multi-process approach offers an important competence; running different Nodes on different machines (albeit by taking a hit on system resources vs a multi-threaded approach). This is important since experiments should not be constrained by the Operating System (OS) or the chip architecture that a small part of the experiment might require to run. For example, a fast, high resolution camera might have drivers only for Windows, while raspberry pi cameras can be advantageous since they are easy to multiplex (due to the pi's GPIO and low cost of a raspberry pi with a camera) while online, million parameter, deep learning algorithms will definitely not run on anything other than high spec Linux machines. Heron removes the need to choose between these capabilities since its Nodes (i.e. processes) can run on any machine connected to the main one that runs Heron, through a Secure Shell Protocol (SSH) accessible network connection. When a Knowledge Graph is initiated (i.e. a task is launched), Heron will connect to all the defined machines in the network and will initialise whatever processes it has been directed to start at each of its predefined machines. While the experiment is running, it will take care of message passing between machines and when the Graph is terminated it will make sure all processes are also gracefully terminated. Since Heron uses standard Python to implement most of its Nodes (something that users, as we mentioned above, do not have to adhere to since functionality to work with other code bases exists – see the Rats playing computer games: State machines and non-Python code Nodes example) a Heron experiment can be easily defined on machines with different chip architectures, different OSes and different levels of virtualisation. The general rule is that if a machine with a certain configuration can run the scripts of Python (or other non-Python code) that define the Nodes that need to run on that machine then that script can be part of a Heron experiment and have its Node's inputs and outputs interact with Nodes running on other machines, all set up through Heron's GUI and with minimal user effort. Heron itself (the GUI and underlying communication functionality) has been tested to be operational under Windows (10 and 11, both x64), Linux PC (Ubuntu 20.04.6, x64) MacOS and Raspberry Pi 4 (Debian GNU/Linux 12 -bookworm-, aarch64).

Python and the ease of implementing code. Finally, Python as an implementation language offers Heron another set of desirable (and some not so) consequences. These include the standard pros and cons of Python versus other computer languages. Apart from this, batteries included, Python approach to problems, the main advantage for experiment implementation is Python's extensive community of developers that have contributed to the open source ecosystem of Python libraries for practically any computation imaginable. This extends to also drivers and control APIs for most hardware that an experimenter might require to use. From standard data crunching algorithms to state of the art machine learning ones and from serial communication to drivers controlling high spec equipment there is very little that has not been covered by a Python library. This wealth of ready to go solutions makes the two-tier approach of Heron (design the Knowledge Graph in a GUI and implement the Nodes' behaviour in Python) not just a viable but the preferred approach for any experimental designer. Especially for those experimenters who may not be versed in the latest nuances of low level computer code, but still would like to be fully in control of the behaviour of their experiment. For the cases where a user with deeper knowledge of software engineering has a specific need to use other languages, Heron offers one last benefit arising from

the use of its message passing library, 0MQ (14). 0MQ is a versatile and easy to use library for passing messages across different processes running on different machines. Most importantly, it includes bindings for almost all commonly used languages. Utilising this library (with minimal effort) a user can create an application in any language that communicates with a Heron Node (practically implementing a small part of Heron's protocol in another application). Then a Python wrapper Heron Node can be made responsible for the executable's life time. In this way, one has just created, with very little effort, a Node that runs an executable written in some other language but acts just like any other Heron Node passing data to and from any other Nodes.

Methods

Heron's architecture.

Node types. Heron defines three different types of Nodes, each implementing a different basic functionality of message passing. Those are the Sources, the Transforms and the Sinks (see the blue, green and red Nodes respectively in the example KG of Figure 1B). The Sources are Nodes that generate data (either computationally or by reading them from a device) and thus can only transmit data through the Nodes' outputs. The Transform Nodes can both receive and transmit data through both input and output points and are meant to allow data manipulation. Finally, the Sink Nodes can only receive data and only feature input points. The Sink Nodes are designed to either save data or talk to devices that require only computer input and not input from the external world (e.g. a motor). The Nodes' types only exist to generate a cleaner code base by separating the three types of message manipulation (output only, input then output and input only). There is nothing though (except Python's rule No 7: Readability counts), stopping a user to create side effects of functions implemented by these Nodes other than message passing, thus interacting with machines in ways different to how the Node's type would suggest.

Heron's Actor-based model. Most of Heron's advantages over similar software tools stem from the way it structures the communication between the different Nodes (and thus processes underlying those). Heron's processes do not allow each other to take control of each other and change each other's state. Each process has full control of its state and will only allow another process to influence it through the passing of messages. This is known as the Actor-based model. In contrast, the most commonly used Object Oriented Programming (OOP) model will allow an object to directly change the state of another. For example let's consider the situation where the result of an online analysis on incoming data should be used to change a camera's gain. In an OOP world the object responsible for the analysis would also have to carry a pointer to the memory that represent the object responsible for controlling the camera and directly change the gain (by changing the value of the instance's gain variable) when required. But what happens when another object is introduced latter on that also needs to control the gain of the same camera? What if the change of gain is also dependent on the gain's history? Who is responsible for correctly changing this parameter when there are more than one objects vying for control and will the introduction of the new object require changing the code on both the camera and the analysis object code bases? The Actor-based model solves these problems by allowing the camera's gain to be changed only by the object that controls the camera itself. All other object can only request such a change by sending request messages to the camera object (in Heron's case the camera Node). In this way, when a user composes a Node they have to think only about what that Node does and how it communicates with other Nodes, and never about the way code outside it might change its behaviour (which Heron with its Actor-based model will never allow).

Heron GUI's multiple uses. To understand Heron's code structure, one must initially appreciate its dual role in designing and running an experiment. When a Graph (short for Knowledge Graph) is not running, Heron acts as a Graph designer, offering a GUI where a user can create and delete Nodes, connect them with Links and assign values to their parameters. During the design period, only one process is active, the one running the Heron GUI (Editor, see Fig2). When, on the other hand, a Knowledge Graph is running, the Heron process stops being a Graph design application and assumes the role of a director in an actor based model (2). It can then concurrently compute and run a GUI for the experiment where the user can update the parameters of the different Nodes on the fly (as an experimental Control Panel). In this actor-based model each Node is represented by two processes (Worker and Communication, see Figure 2) while there are three more processes acting as message forwarders between all other processes (Proof of life, Parameters and Data). That means, a running experiment is constituted by (Number of Nodes) x 2 + 4 processes (see Figure 2). Each process is an actor that can receive and transmit messages, make local decisions (i.e. decisions that can effect only itself) and determine how to respond to incoming messages. In the (most common) cases where the Nodes running are all implemented by Python code then the Heron process is responsible for initiating the three forwarders and the com processes for all the Nodes. Each com process will then initiate the corresponding worker process. In the special case (see Rats playing computer games: State machines and non-Python code Nodes example) where a Node will call an executable instead of a Python script then the worker process can also be responsible for initialising (and terminating) the executable's process. As mentioned above, each Node is represented by two processes. In the code those are called the com and worker process. The worker process is the one that runs the Node's script defined by the user. The com process is responsible for i) grabbing messages that come out of other Nodes and are meant to reach the Node (as defined by a Link between two Nodes in the Node Editor), ii) passing those messages to the worker process, iii) receiving any messages the worker process has to pass to other Nodes and iv) passing those messages to the com processes of all the

Nodes that should receive them. The passing of messages between com processes of different Nodes is facilitated by the Data Forwarder process. The worker processes also communicate directly with the Heron process through two separate forwarders. The Parameters forwarder is responsible for passing to the worker processes the parameter values assigned by the user to the processes' respective Nodes on their GUI. This allows the user to also manipulate the state of each Node while an experiment is running. Through this functionality the Heron GUI becomes (while a Graph is running) also a control center through which an experimenter can interact with the experiment by changing live the Nodes' parameters. The Proof-of-life forwarder is responsible for receiving an initial message from the worker process which is then passed to the GUI process letting it know the worker is up and running.

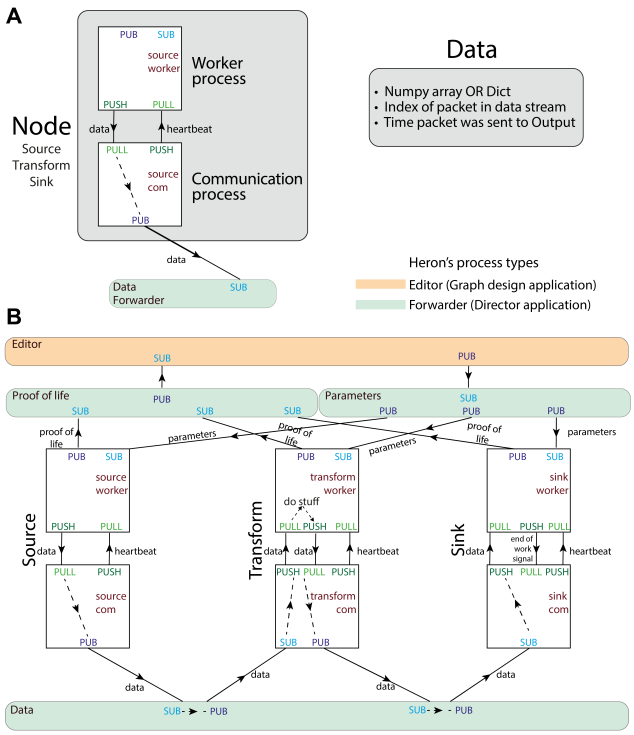

Fig. 2. Heron's design principles. A. On the left, Heron's Node structure showing the two processes that define a running Node and how messages are passed between them. The design of the special message defined as 'data' is shown on the right. B. Heron's processes and message passing diagram. Each rectangle represents a process. The top orange rectangle is the Heron's Editor process, while the other three green rectangles are the three Forwarders. The Proof of life forwarder deals with messages relating with whether a process has initialised correctly. The parameters forwarder deals with messages that pass the different parameter values from the editor process (the GUI) to the individual Nodes. Finally, the data forwarder deals with the passing of the data messages between the com processes of the Nodes. The squares represent the worker (top) and com (bottom) process of three Nodes (a Source, a Transform and a Sink from left to right). Solid arrows represent message passing between processes using either the PUB SUB or the PUSH PULL type of sockets as defined in the 0MQ protocol. Dashed line arrows represent data passing within a single process.

As mentioned above, Heron allows any of the Nodes in a Graph to be initiated and executed in machines different to the one running the main Heron process. At the level of the processes, that means that Heron, if instructed to run a Node on a different machine, will run only the worker process of that Node on the different machine while its com process will run on the same machine as Heron. That has as a drawback that a user cannot put multiple Nodes on a separate machine and expect them to interact (through messages) within that machine since all message passing happens through the com processes which will always run on the Heron running machine. Future version of Heron will address this limitation by allowing Heron to run Graphs headless (without the GUI process being active) which will allow sub-Graphs to fully run on one machine and communicate their result to Nodes in the machine running Heron.

Code architecture. Heron's code is separated into three main folders each pertaining to one of the three aspects of its basic functionality (see Supplementary Figure 1 for all of Heron's folder structures). The Communications folder includes scripts that deal with the low-level communication between all Nodes that make up a running Graph (experiment). The GUI folder holds scripts that deal with Heron's Graphical User Interface (GUI). The Operations folder keeps all the code that defines all the Nodes that Heron recognises and can use to create a Graph with. The Operations folder is further subdivided into the Source, Transform and Sink folders, each holding codes according to the type of the Node it implements. The Operations folder also holds symbolic links to code repositories separate from Heron. Those, assuming a specific folder structure, are recognised by Heron as valid Node describing codes. The scripts in the Communications folder are class definitions for 8 objects: 6 for the worker and com objects for each Node type which implement Heron's communication protocol, one for the object that deals with the network connectivity (SSHCom) and one re-implementing pyzmq's (Python's 0MQ bindings) Socket object, adding to it the ability to pass numpy arrays and dictionaries as messages. This is required because in Heron all messages are either numpy arrays or python dictionaries. That means that the worker functions of the worker scripts of the Source and Transform Nodes will always return either a list of numpy arrays or a list of dictionaries. Each element of the list corresponds to one output of the Node. The Operations folder has three levels of subdivision. Immediately bellow it are the Source, Transform and Sink folders and inside those are folders representing groups of Nodes in each Node type (e.g. Vision for Nodes that have to do with computer vision). Inside those subcategory folders are the folders that hold the scripts for each Node (e.g. Camera which holds the scripts that read a web camera into Heron). Inside each Node's folder there are a minimum of two scripts with name suffixes $_com$ and $_worker$. The $_com$ script allows the user to define a Node's characteristics (parameters, inputs and outputs) with a few lines of code and without requiring any GUI relevant code (Heron takes care of that). The $_worker$ script is responsible for the functionality of the Node (being the script that is run by the Node's worker process through the $node_type_worker$ object) and implements a minimum of three functions. These are the initialisation, the worker and the end-of-life functions

(the names are arbitrary and the user can define them as they please). The initialisation function is run when a Node is first started by Heron (i.e. its com process is up and running and its worker process has just started but is being tested before it starts receiving and transmitting data). The worker function is the main function that implements what the Node is supposed to do. The worker functions of the three types of Nodes are implemented differently. In the case of a Source Node, the worker function needs to be an infinite loop that somehow generates data and passes them on the Node's com process at the end of every loop. The Transform and Sink Nodes need a worker function implemented as a callback since their worker processes will call the worker function every time there is any data arriving at the input of the Nodes (i.e. any time their com process has received a message from another com process and has passed this to its worker process). Both the Transform and the Sink Nodes will stop accepting messages until their worker functions have returned and Heron is designed to have no message buffering, thus automatically dropping any messages that come into a Node's inputs while the Node's worker function is still running. Finally, the end-of-life function will be called when a worker process hasn't received a heartbeat signal from the source process for a pre-determined amount of time and its role is to gracefully terminate the process.

Usage. There are two skills that a user should possess in order to aptly use Heron. Firstly, one requires a familiarity with Heron's GUI which allows i) downloading and installing new Nodes from existing repositories, ii) defining a local network of computers on which the different Nodes can run, iii) setting up a pipeline using the existing Nodes and iv) running the pipeline all the while being comfortable in debugging it as problems arise. The second skill is the implementation of new Nodes based on the user's individual needs. In this section we will provide a basic description of both the GUI usage and the development of new Nodes.

Using Heron's GUI.

Adding new Nodes from pre-existing code Heron comes pre-packaged with a small set of Nodes that have a generic enough usage that most users would find useful. An important point though about Heron is that every user will be developing their own Nodes which in most cases will take the form of code shared in some online repository. Heron is designed to easily access repos that have been developed following a specific file structure to represent a set of Heron Nodes and integrate them into its GUI and workflow without the user needing to do anything else other than create/download the repository and point Heron to it. This also simplifies the further development of Nodes by the community of users since a new Node repository does not have to interact with the main repository of Heron and thus avoids all the pitfalls of pushing, pulling and merging code repositories at different levels of maturity.

Local Network Heron's GUI allows an easy definition of the local access network (LAN) of machines that will run Nodes forming a single pipeline. A user has only to provide the IP, port, user name and password of a machine in the LAN and Heron will communicate between machines using an SSH protocol, taking care of issues like process lifecycle on different machines, opening and closing ports and proper passing of messaging between processes over the network.

Setting up a pipeline Once all the Nodes' repositories have been made known to Heron and the LAN of all machines has been set up, a user needs to implement the experiment's pipeline. This is achieved again graphically by introducing the required Nodes in the Node Editor (main window of Heron), setting up their parameter values and finally connecting the Nodes together by creating links between outputs and inputs. Heron allows many to many connectivity, meaning a Node's output can connect to any number of inputs and an input can receive any number of outputs.

Running a pipeline Once a pipeline has been defined (generating the Knowledge Graph of the experiment) then running it is achieved by pressing the Start Graph button of Heron. Heron will go through each Node (in order of addition to the Node Editor) and will start the processes that the Node represents (see Heron's Architecture for more details). It will then connect all the processes with 0MQ sockets as defined by the links between the Nodes and pass the Nodes' parameters to the worker process of each Node. The pipeline of data being generated by the Source Nodes, being transformed by the Transform Nodes and finally saved or turned into control of machines by the Sink Nodes will keep on running until the user pressed the End Graph button. At this point Heron will gracefully terminate all processes (including the ones running on separate machines) and close down all communication sockets.

Creating a new Node Heron users will develop their own Nodes for their specific experiments. To facilitate this, Heron provides a series of tools. First there is a set of templates that offer a scaffold on top of which a user can build their own code. The templates have the required code elements that all valid Heron Nodes must possess and are fully documented to help a user quickly build functioning Nodes. An abbreviated and annotated Transform template for the com and worker scripts can be seen in Supplementary Box 1A and 1B. On top of Node templates, Heron provides a GUI based system for generating new Nodes. This will create the required files and all the common between Nodes, skeleton code, appropriate to the type of Node under development and using names (for functions and variables) chosen by the user during the GUI based creation process. Finally, in order to expedite the correct placement of new Nodes in pipelines Heron offers a Node (User Defined Function 1I 1O) designed to allow python functions to be run as Nodes in Heron (without the user having to follow Heron's API). This Node can be used to simulate inputs and outputs to an under development Node in order to test and debug its functionality before incorporating it into a full pipeline.

Node Repositories As described above, Heron offers the tools to integrate any new code (designed with the correct file structure) into its collection of Nodes and make it available in its GUI. Although not necessary, good practice would be to develop any new Node (or closely related group of Nodes) as part of a separate repository so that the Node can be easily shared with the rest of the community. Currently a public GitHub organisation called (rather unimaginatively) Heron-Repositories is hosting both the main Heron Git repository and all other Git repositories of Nodes developed to cover the developers' experimental needs. Any of the individual Heron Nodes repositories can serve as an easy to follow example on the file structure expected by Heron for successful integration of new code. All of the Nodes presented in the Results paragraphs examples can be found in this repository.

Results

Here we showcase a number of experiments implemented in Heron. Each example has been chosen to highlight one of Heron's competences as described in the Design benefits. As mentioned above, all the Nodes used to construct the examples presented here can be found in the Heron-Repositories GitHub organisation. We are not making public the specific experiment files since these are hardware specific and would need large changes to be made compatible to any other hardware. But Heron's graphical nature makes it easy to go from an image capture of an experiment's Heron GUI (see Figs. 3 to 5) to a working experiment by simply combining the required Nodes.

Probabilistic reversal learning. Implementation as self-documentation. The first experimental example is provided here to showcase how Heron's implementation of an experiment becomes the easiest way for non-developers to acquaint themselves with the experiment and its logic. Thus, here we describe the implementation from the point of view of someone who sees it for the first time and is trying to understand what the experiment is (without accessing any other publication or written explanation). As seen in Figure 3, this experimental pipeline is made up of 4 Nodes. One is called a "KeyPress" which since it connects to an input named 'Start / Previous Trial Result' seems to play the role of the start button of the experiment. The last Node is a "Save Pandas DF" which suggests saving the output of the 3rd Node (the "Trial Controller") in a row of a pandas DataFrame. The main experiment seems to be defined by 2 Nodes, one named "Trial Generator" and one named "Trial Controller". We can immediately conceptualise the pipeline as a two part one, where the first part generates some kind of trial state which it then outputs as its 'Trial Definition' output to the second Node which inputs that 'Trial Definition' and runs (Controls) that trial given its state. We notice that these two Nodes are reciprocally connected, meaning that the "Trial Generator" requires the output of the "Trial Controller" (named 'Trial History') to generate the definition of the next trial. Looking a bit more closely at the names of the parameters of the Nodes we can deduce a number of things about the experiment's structure and function. From the Trial Generator parameters, we see that trials seem to fall into two blocks. We can also see that the experiment has trials with 4 types of Stim (maybe short for Stimulation, or Stimuli). We can assume (but would need to verify from the code) that the Reward Block Contingencies mentioned for the two Blocks are the probabilities of a reward given the type of Stimulation. So, we have surmised that the trials come into blocks of specific length (probably a random variable drawn from some distribution from the user provided Blocks Length Range parameter) and of specific trial type (one of four) and each trial type in each block has a user defined reward probability. So far so intelligible. By looking at the Trial Controller Node's parameters we see first of all that the Node requires a COM Port and Baud Rate to be defined showing that it is controlling some device through a serial port. The 'Reward Only After Lick' parameter tells us that this is an experiment where the subject needs to lick (and in some cases this is the only way to get a reward). The names of the rest of the parameters indicate that the experiment is an olfactory one (see 'Odour Window') where the subject gets to experience an odour (maybe one of the four stimulations described in the Trial Generator), then gets a pre-response delay, then a response window time and finally a reward window. In order to get a better picture of what the experiment is actually doing we next look at the code. But most importantly, we know exactly where in the code base we should be looking without needing to search through tens of files to follow an obscure logic to figure out where the bits of the code that implement the actual logic are. There are two Nodes of interest so we should, as a first step, look at the worker scripts of those two Nodes. The Trial Generator is an undaunting 127 lines of code while the Trial Controller is 251 (empty lines included). Yet we can zoom even more in where we should initially look by simply ignoring the initialisation and end_of_life functions and just concentrate on the $worker_function$ of each script. Those two functions are shown in the $Worker_function$ Code in Supplementary Boxes 2A and 2B. We need only two pieces of prior knowledge to understand the code. One is that the worker functions of the Transform (and Sink) Nodes are callbacks that are called every time the Node receives a message to any of its inputs. Both the Trial Generator and the Trial Controller are Transforms (they have both inputs and outputs and are marked as green on the GUI). The second is that each Node outputs through the return statement of its worker function a list of numpy arrays with each element of the list being the array that is outputted by a specific output of the Node (in order from left to right in the list corresponding from top to bottom in the Node). So, a Node with two outputs will have a worker function that returns a list with two numpy arrays / dictionaries. In this case, for example, the Trial Controller's worker function should return a list with two arrays i.e. the Trial Result output and the Trial History one. With this in mind and looking at the Trial Generator we can see that it takes in the previous trial's stimulation number and if the correct port was licked, use this information to keep a running track of the correct licks for each port, decide which

block we currently are in, decide which stimulation to generate next and offer or not a reward. By following the Trial Controller $worker_function's$ code we can easily validate our prior speculation about the temporal structure of a single trial following a path of delays interspersed with stimulation and potential reward. An overview of the actual experiment running on Heron, also showcasing how Heron communicates with the Arduino board that does the hardware control, can also be seen in Supplementary Figure 2, together with a schematic of the experimental timeline.

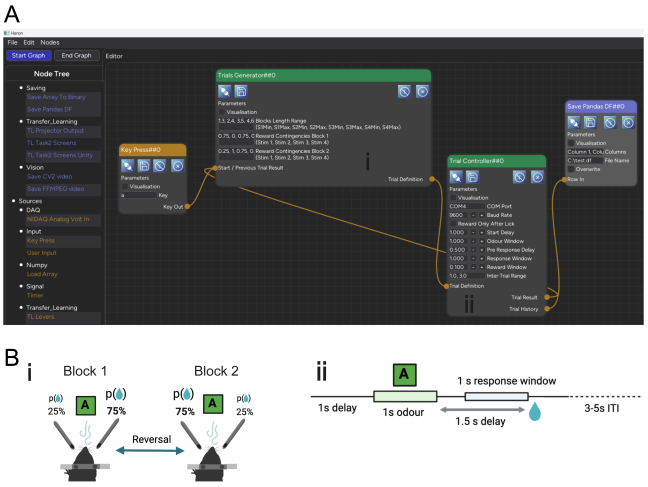

Fig. 3. A. The Knowledge Graph of the Probabilistic Reversal Learning experiment showing 4 Nodes comprising the task, two of which (Trial Generator and Trial Controller) are connected in a loop (the output of one is fed into the input of the other). B. The schematic or mental schema of the experiment. Notice the direct correspondence (i and ii) between the Heron Nodes and the two main components of the experiment's mental schema as well as the Node's parameters and the schema components' basic variables.

Fetching. Four environments, three Operating Systems, two machines, one pipeline . This next experiment showcases Heron's ability to run Nodes whose worker process runs on machines separate from the one running the Heron GUI. The experiment itself requires the monitoring of a rodent as it tries to fetch a cotton bud from a random point in a large arena and drop it into a shallow pot under a hole at one of its corners. The hardware setup has a Grasshopper3 USB 3.0 colour camera (GS3-U3-23S6C-C, FLIR) which records the whole of the arena from the top at 120 frames per second with HD resolution, while 4 smaller, black and white, 30 frames per second cameras (OV9281, ArduCam) are used to capture the animal from 3 different angles and to also monitor the target pot from underneath (since its base is transparent plastic) using the fourth camera. The experiment needs to also know when the animal has deposited the cotton bud in the hole and also record at which angle this has happened. When the animal has deposited the cotton bud it is rewarded with a treat from a nearby reward port. The pot rotates removing the cotton bud from the arena (whilst another, empty pot takes its place) and the cotton bud is thrown back into the arena for the animal to fetch again. The Grasshopper camera at the time of the experiment's development could run only on Python 3.8. The 4 smaller cameras are part of a system that synchronises them, providing a stitched up single frame of 5120x800 pixels resolution, (ArduCam 1MP*4 Quadrascopic Monochrome Camera Kit) which can run only on either a Raspberry Pi (>=3) (Raspberry Pi Foundation), a Jetson Nano (Nvidia) or a Xavier (Nvidia) single board computers. The angle detection of the cotton bud when it is in the pot is done using Meta's Detectron 2 (15), a deep learning algorithm which we trained with a few hundred samples to detect the cotton bud's edges when in the pot using as an input the part of the OV9281 camera frames that come from the camera underneath the pot. Detectron 2 requires an Intel or AMD CPU based computer running Linux. Heron itself and all the other Nodes for this experiment run on a Windows PC with Python 3.9. The choice to run the Heron GUI on the Windows machine was made to keep the latency of the 120fps camera to a minimum. So, the experimental pipeline needs the following Machine/OS/Python configurations: Intel/Windows/Python 3.9+, Intel/Windows/Python 3.8, Intel/Linux/Python 3.9+ and ARM/Linux/Python 3.9+. To create all the above configurations, we first made in the main Windows 11 machine (that Heron runs on in a Python 3.9 environment) a separate (conda) environment with Python 3.8 and the required Spinnaker (for the Flir camera) python package. Then we set up on the same machine a Windows Linux Subsystem (WLS 2) virtual machine, running Ubuntu and Python 3.9 with all the required packages to run Detectron 2. Finally, we connected the Windows machine through a LAN to an NVIDIA Jetson Nano with the Arducam Quadrascopic system running Ubuntu and Python 3.9 with all the packages for the Arducam system to operate. To summarise, the pipeline (which can be seen in Figure 4) when run, is utilising two physically separate machines, three operating systems (one Windows 11 and two Linux, one on a virtual machine) and four different Python environments. Once each machine/OS/Python environment is up and running and each one can run the Nodes(s) that it is supposed to run by itself (something that can be tested and debugged at the level of an individual Node without requiring the whole pipeline to be up and running), then assembling the pipeline is as simple as connecting the Nodes appropriately on Heron's Node Editor and telling each one (through the Node's secondary parameters window) which computer it should run on and which python executable it should call to run the worker script. Heron hides from the user all of the work required for the different processes in all the machines to start at the right time, connect correctly to each other, exchange data while the pipeline is running and finally gracefully stop when the pipeline stops without leaving hanging processes and inaccessible bits of memory all over the place. An overview of which machine runs what Node can be seen in Figure 4 while Supplementary Figure 3 shows a snapshot of an animal having fetched a cotton bud while its angle is being live detected by one of the Detectron 2 algorithms.

Rats playing computer games.. The rats playing computer games experiment teaches a rat to rotate a line on a screen using a left and a right lever press to hit a target line and avoid a trap line. It presents a rat with a nose poke hole,

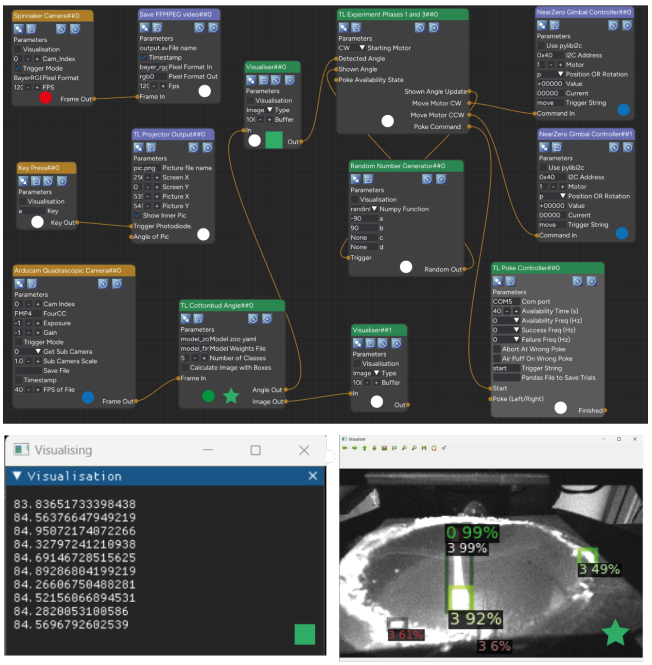

Fig. 4. The Knowledge Graph of the Fetching the cotton bud experiment. The coloured circles at the bottom middle of each Node and the coloured star and square (where shown) are not part of the Heron GUI but are used in the Figure to indicate in which Machine/OS/Python configuration each Node is running its worker script (circles) and which visualisation image at the bottom corresponds to which Node. For the circle the colour code is: White = PC/Windows 11/Python 3.9, Red = PC/Windows 11/ Python 3.8, Blue = Nvidia Jetson Nano/Ubuntu Linux/Python 3.9, Green = PC/Ubuntu Linux in WSL/Python 3.9. The two smaller windows bellow Heron's GUI are visualisations created by the two Visualiser Nodes. The right visualisation (coming from Node Visualiser##1) is the output of the Detectron2 algorithm showing how well it is detecting the whole cotton bud (detection box 0) and the cotton bud's tips (detection boxes 3). The left visualisation box (output of the Visualiser##0 Node) is showing the angle of the cotton bud (in an arbitrary frame of reference where the 0 degrees would be almost horizontal on the screen and the 90 almost vertical). This angle is calculated in the TL Cottonbud Angle Node which is responsible for running the deep learning algorithm and using its inference results calculating the angle of the cotton bud. As shown the TL Cottonbud Angle Node is running on a Linux virtual machine (since Detectron 2 cannot run on Windows machines).

two levers (left and right to the hole) and two screens to the animal's front and right, when it is poking. At the final stage of training the animal should be able to first nose poke, look at the screen at a set of jagged lines and press one of the two levers to make one of the lines rotate toward the correct line (target) and away from the wrong one (trap). Once this has been achieved the jagged lines disappear and a separate visual cue appears (usually animated) letting the animal know that there is reward in the reward port (see Supplementary Figure 4). This is a very challenging behaviour for a rat and the experiment needs several stages of shaping to teach it to the animals. Each stage of shaping toward the final behaviour requires a set of different visual stimuli and a different set of states in a rather large state machine. Also, at the conception of the experiment there was no prior experience on the ideal path to the final behaviour, so a very malleable stimulus generation technique and state machine development was required in order for a large number of ideas to be tested in a small amount of time. Because of the above this experiment is ideal in showcasing a series of Heron's capabilities. These are Heron's ability to work with non-Python code, Heron's workload management, the tools Heron provides to keep track of complicated experiments and record all required information and finally Heron GUI's capability to act not just as a Graph designer but also as a control panel while a Graph is running.

State machines and non-Python code Nodes. As mentioned above, although Heron's code base for Node development is in Python, it is relatively easy to create Nodes with code bases on different languages. One way to do this, (which is the way most scientific computing Python libraries are using), is to use Python interop technologies which allow functions written in other languages (most commonly languages of the C family) to be called by Python. Although this is very powerful it is a painstaking task, can be very time consuming, and usually requires significant experience with both Python and whatever other language one is working with. Another, not as a low-level approach, offering less control but faster implementation times, is the use of the 0MQ library to pass messages between a Heron Node and an executable written in another language (what Heron is doing to connect all of the Nodes of a pipeline but with a significantly toned-down communication protocol). Here we took the second route. In the Unity game engine (Unity Technologies, US) we made a simple 2D game using C# code, that covered all the possible visual stimuli we needed to show to the animals at any stage (see Figure 5D for a snapshot of the Unity development environment). Instead of using the standard game inputs (e.g. keyboard or game controller) to control what and when was played on the screen, we used a string of commands that was received by the game executable through a 0MQ SUB socket (using NetMQ, one of many C# 0MQ bindings). The Unity generated executable was also designed to do a handshake through a PUSH PULL socket with whatever process initiated it. This made sure the initialising process knew if anything was wrong and also was able to send some initialisation messages. Once the game executable was ready we created a standard Heron Sink Node whose initialisation function would start the executable, handshake with it and send it the Node's parameters (e.g. if the game was meant to show its stimuli on one or two screens). Then its worker function would just pass to the correct socket any string that would be received at its input from the other Heron Nodes. Properly formatted strings would be understood by the game and update its state accordingly. The Node's end_of_life function would finally close down the executable when Heron's Knowledge Graph was terminated. This use of process control done not directly from the main Heron process but from a Node's worker process is compatible with the concept of Heron as an actor-based framework where an actor (a Node's worker process) can also initialise and end other actors (in this case the Unity executable). Regarding the state machines used in this experiment, we found that no matter how large and complicated a state machine gets, the difficulty lies in its initial design and not in its implementation. Here we used statemachine (16), a python library that allows the definition of a state machine and its individual states with easily attachable callbacks at state transitions.

CPU load. . Using multiple processes rather than multiple threads does incur a hit in the total CPU load Heron will require in order to execute a Graph. The 'Rats playing computer games' experiment, with a 15 node Graph (the largest one we had ever had to execute over multiple experiments implemented in Heron) offers a good test bed to showcase Heron's CPU requirements. The experiment was conducted on a Windows 10 x64 machine with an 8 core i7-7700K CPU clocking at 4.20GHz. The Graph runs an acquisition and save of a 120fps 576x676 pixels video, an acquisition and save of a 20KHz, 5 channel DAQ trace updated at 10Hz, a Unity game running concurrently on two HD screens updating at 120Hz and an acquisition and save of a 30fps 5460x800 pixels video running on the Jetson Nano machine. When the computer was idle it showed a base CPU load of just around 10-15%. When Heron was up but not running a Knowledge Graph, it consumed about 2-5% of the CPU for a total load of 15-20%. With the above experiment running, the CPU load hit 100% with the Unity game requiring around 24% and the remaining 45-55% used by the Graph's 32 processes (one process was running on an NVIDIA Jetson Nano machine). The above numbers though are true if the operating system was allowed to assign CPU cores to the different Heron operations as it saw fit. Heron has a feature which allows the user to force the operating system to keep a worker process running on a specific CPU core. Rerunning the above Graph while setting 10 of the 15 worker processes to specific CPUs and allowing the system to optimise the load for only 5 of them (the Spinnaker Camera, Save FFMPEG Video, Arducam Quadrascopic Camera, Save Array To Binary and TL Taks2 Screens Unity) dropped the total CPU load to 80%, again with the Unity game consuming around 24% while all Graph python processes consuming around 25%. The above specifications show that Heron is capable of running Graphs with heavy acquisition and save loads. Also, Heron's ability to utilise multiple machines in a single Graph means that processes requiring exceptional CPU, memory and / or GPU loads can be offloaded to machines other than the one running the Heron GUI. In the above example this is the case with the acquisition and save of the Arducam video. Also, before implementing the CPU setting feature we used to run the Unity game on a 3rd machine in order to keep the 24% CPU load requirement out of the main Heron machine. We stopped using this 3rd machine once the CPU setting feature was in place, and the transition from three to two machines required only a single line change in the Node's secondary window to indicate where the worker script was to be found.

Synchronisation (time alignment). Multiple processes, running over different machines can become extremely difficult to synchronise, especially to the tight requirements of a scientific experiment. Heron offers a set of tools to allow users to time align every single packet of data generated and saved by any Node to any other packet irrespective of whether the packets were generated by Nodes on the same or separate machines. Here we use the 'Rats playing computer games' experiment to showcase how such time alignment would be achieved between a few of the experiment's Nodes.

More specifically the experiment has two separate video captures which, since they happen on different machines, they can only be synchronised through an external clock. This is similar to an electrophysiology data acquisition system whose data points can be synchronised to other data sets (e.g videos, or button presses), only by am external clock (a TTL pulse generator usually) that is simultaneously recorded by all devices. The whole setup is shown diagrammatically in Figure 5C.

We use a 120fps projector (whose output is controlled by the Node TL Projector Output##0) to act as our base clock. By pressing a key after the Graph has started running, a part of the projector's output (depicted as a screen in Figure 5C) turns blue and thus stimulates a blue sensitive photodiode. This happens at every frame thus the photodiode generates a 120Hz TTL pulse train timed to the onset of the blue pass of the projector. That train starts when the key is pressed and stops when it is pressed a second time (thus demarcating the start and stop of the recording part of the session). The photodiode's TTL pulse acts as an external trigger to the Spinnaker (FLIR) camera whose output is being captured by Heron's Spinnaker Camera##0 Node. The same pulse is also fed to a National Instruments USB Data Acquisition System (NIDAQ) and to an Arduino. The NIDAQ is being captured (at 10 packets of 2000 points each per second) by Heron's NIDAQ Analog Volt In##0 Node. The Arduino runs a small loop that generates a second TTL pulse for every 4 TTL pulses coming in from the Spinnaker camera (thus creating a 30Hz TTL train). This TTL train is fed as an external trigger to the four cameras of the Arducam system as well as to the GPIO of the Jetson Nano machine that is running the Heron Node which captures the Arducam's video frames output (Arducam Quadrascopic Camera ##0). The Arducam system generates a TTL pulse every time the global shutters of the four cameras fire and that is fed into the same NIDAQ as the Spinnaker TTL pulse out (see left column of Figure 5C.).

The Spinnaker camera at every trigger generates not only a frame but also the frame's id. This is captured by Heron's Spinnaker Node's Worker process and saved to its Substate dataframe together with the equivalent id of the Node's data packet and the timestamp of this (see Figure 5C "Spinnaker Camera Worker Process Data" table). A Substate dataframe is Heron's way to allow the designer of a Node to save information (to a pandas dataframe) every time the Worker process runs a pass through its loop. This is done through Heron's API requiring only a single line of code with the information to be saved to the dataframe.

Any failures of the Node to capture a frame get registered in this correlation between camera ids and packet ids. The Spinnaker Node sends the captured frame to the SaveFFMPEGvideo##0 where an ffmpeg pipeline saves each frame to a video stream. As mentioned before, Heron does not offer any capability to buffer packets, so if the FFMPEG Node is too late to save any frame then the next one gets lost. Heron records this in the FFMPEG Node's Com process log file. A Com processes log file (optionally set for any Node by filling

in the log file's path to the Node's secondary window appropriate entry) records the id of any packet that arrives from an upstream Node as given by that Node and the id of that packet as given by the current Node. In this way any dropped packets are registered and a one to one equivalence between packets sent and packets received can always be established. In this case, the FFMPEG Node's Com process log file indicates the id of each packet that has left the Spinnaker Node and arrived in the FFMPEG Node and the id assigned to it by the FFMPEG Node. The id of the FFMPEG Node corresponds to the frame id of the video generated by the ffmpeg pipeline in the Node. In this way we can now correlate any frame of the produced video to the id of that frame generated by the Spinnaker camera despite some dropped frames in the process (in the shown example the camera generated a total of 216300 frames and the final video had a total of 216244 frames, a drop of 56 frames dropped over the two Nodes, i.e. a 0.02%).

At the same time the Arducam system (triggered by the subsampled TTL pulse from the Spinnaker camera) generates frames at a rate of 30fps. These are captured and saved by the Arducam Node running on the Jetson Nano. The Node is also registering (for every incoming frame) the equivalent id of the TTL pulse that generated that frame by also reading the Jetson Nano's GPIO port. In this way the Arducam Node's Worker process can generate a Substate dataframe that correlates the packet's id as given by the process (and thus the id of the video's frame) with the TTL pulse id that produced that frame.

In order to now correlate the frames of the Arducam video with those of the Spinnaker video we use the saved TTL pulses from the NIDAQ instrument as saved by the Heron NIDAQ Node. The id of the frame captured by the Spinnaker camera corresponds to the number of the pulse in the Spinnaker pulse train and the id of the pulse saved in the Jetson Nano's GPIO port corresponds to the number of the pulse in the Arducam pulse train. By knowing which Spinnaker pulse has generated which Arducam pulse one can work backwards through the above described correlations all the way to knowing which frame in the Spinnaker video corresponds to which frame in the Arducam one despite those being captured on different devices and some frames getting lost by Heron's "dropping of packets" architecture. In Figure 5C we show more specifically how the last pulses of the TTL trains in the NIDAQ can be used to trace the correspondence between the last frames in the videos of the Spinnaker and Arducam systems using Heron's Substate dataframes and log files.

On the fly experimental control. As mentioned above, the "Rats playing computer games" experiment is a rather demanding task that requires slow and careful shaping on the subjects' behaviour. Certain parts of the training could not be automated (especially when we were trying ideas that we didn't not know if they were relevant to the final shaping protocol). During those sessions the experimenter had to change a number of the experiment's parameters on the fly as the subjects produced behaviour given both how well they were doing and how the experimenter perceived certain aspects of their behaviour that were too hard to quantify (and thus use in an automated system). Examples of these experimental parameters were the number of reward pellets offered to the subjects, the maximum angle between the moving line and the target line and the speed of the moving line (and thus the amount of time required for a lever to be kept pressed), the amount of time the reward was available after a successful trial and whether the levers would vibrate or not when pressed correctly, etc. (see Figure 5E). Heron allowed us to operate in this "experimenter in the loop" fashion because it allows the Worker process of any Node to update its parameters as they are changed by the experimenter through the GUI while the Graph is running. Which parameters are capable of online updating is up to the author of the Node and Heron allows both updatable and non-updatable parameters (which are easily defined in the Worker process code through Heron's API).

Discussion

We have presented Heron, a new tool for coding up experimental pipelines. We have put forward the proposition that using Heron instead of the many other frameworks that one can utilise to create software to run experiments has a series of advantages. It can practically self-document, creating Knowledge Graphs (KG) that are as close as possible to one's mental schema of the experiment. These KGs and code bases are easy to follow by researchers other than the developers of the experiment, irrespective of the complexity of the experiment. It can trivially connect processes that run on different operating systems and machines in a single, unified pipeline. For example a series of raspberry pi computers, each reading some cameras or other sensors, can connect and pass the data to a Linux based, many GPU, machine that does online machine learning analysis while these results can pass to a PC machine running a computer game controlled by those results. It is using a language (Python) for the development of experiments that is one of the easiest and most versatile computer languages with a large community of developers and rich libraries for most functionalities. Finally, it is versatile enough to allow easy integration of code bases written with languages other than the one it has been developed in.

We are arguing here that Heron's learning curve, starting from a basic capability in Python, is measured in the few hours of trying to create a couple of new Nodes and joining those together in order to create some toy experiment. Once that is understood then the limit to what can be achieved is defined by the level of Python knowledge of the developer. Heron has been conceptualised to grow into a community project. Both itself and the repositories holding extra Heron Nodes are open source under an MIT licence. The separation of repositories that hold the main Heron code and the individual Nodes' code bases allows for the growth of a Node ecosystem where users will be able to share their development using standard repository based tools. Finally, the developers welcome efforts for collaboration with the aim for Heron to eventually become a multi-developer, collaborative project expanding with capabilities covering the needs of ex-

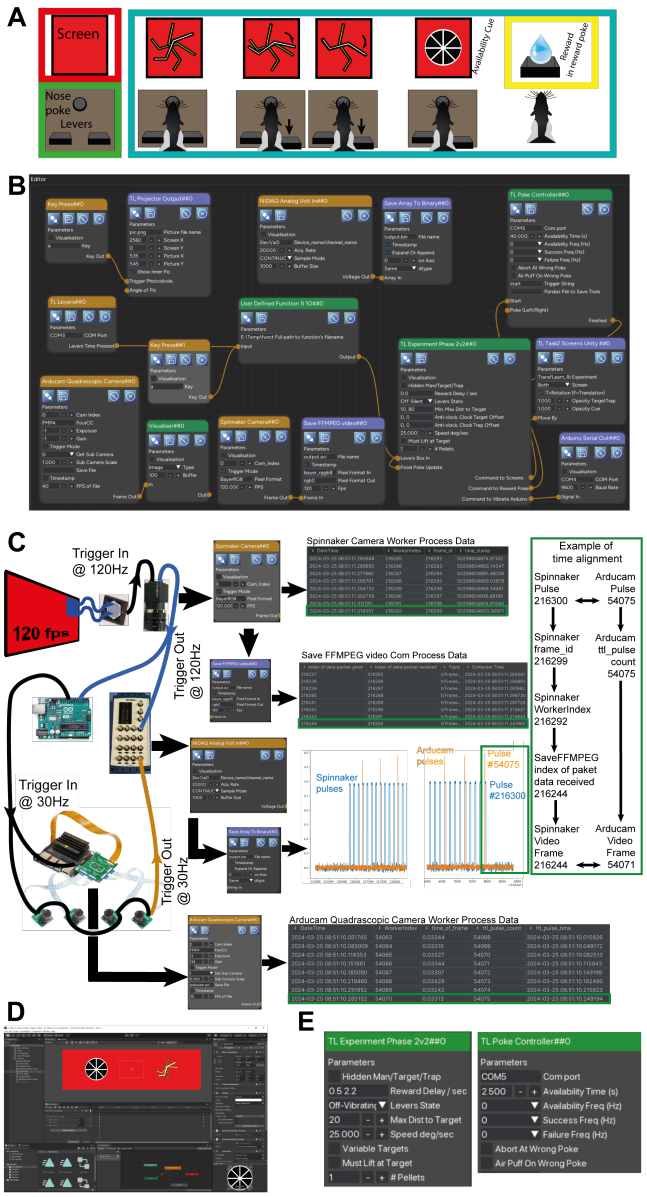

Fig. 5. Computer game for rats. A. The experimental design (see Supplementary Fig 4 for a detailed explanation) B. The Heron Knowledge Graph. C. An overview of some of the hardware connectivity of the experimental setup and the way Heron deals with the time alignment of the data packets from different Nodes. The green rectangle to the right shows how one can use Heron's data recording facilities to correlate the frames of the Spinnaker Camera saved video (the example here highlights the last frame #216244) with the video frames of the Arducam cameras (#54071 in this example) which are generated and saved by a separate Node running on a different machine. D. The Unity project used to create the visuals and their reactions to the commands send by the TL Experiment Phase 2v2##0, logic Node. E. A zoom of the parameters of the two logic Nodes in the experiment (TL Experiment Phase2v2##0 and TL Poke Controller##0). Updating those parameters during an experiment an experimenter can control the way the arena interacts with the subject and allows Heron's GUI to act as a control panel for the experiment while a Graph is running.

perimental scientists in all experimental fields and beyond.

The code for all repositories used in this report can be found at https://github.com/Heron-Repositories, while Heron's documentation is hosted in https://heron-42ad.readthedocs.io/en/latest/. Heron's name is a tribute to one of the first known creators of automata, Heron of Alexandria.

Funding and Conflicts of interest

Funding. This work was supported by Gatsby Charitable Foundation (562980, 549084 for AA, 568836, 562992 for GD) and Wellcome Trust (562763 for AA, 562776 for GD)

Conflicts of Interest. The authors declare that they have no conflicts of interest.

Bibliography

1. E. Kindler and I. Krivy, "Object-oriented simulation of systems with sophisticated control," Int. J. Gen. Syst., vol. 40, no. 3, pp. 313–343, Apr. 2011, doi: 10.1080/03081079.2010.539975.
2. G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems. 1986. doi: 10.7551/mitpress/1086.001.0001.
3. R. Bitter, T. Mohiuddin, and M. Nawrocki, LabVIEW: Advanced programming techniques. Crc Press, 2006.
4. G. Lopes et al., "Bonsai: an event-based framework for processing and controlling data streams," Front. Neuroinformatics, vol. 9, 2015, doi: 10.3389/fninf.2015.00007.
5. F. C. Bartlett and C. Burt, "Remembering: A Study in Experimental and Social Psychology," Br. J. Educ. Psychol., vol. 3, no. 2, pp. 187–192, 1933, doi: 10.1111j.2044-8279.1933.tb02913.x.
6. L. H. Williams, "Thinking through death and employment: The automatic yet temporary use of schemata in everyday reasoning," Eur. J. Cult. Stud., vol. 22, no. 1, pp. 110–127, Feb. 2019, doi: 10.1177/1367549417719061.
7. D. Fensel et al., "Introduction: What Is a Knowledge Graph?," in Knowledge Graphs: Methodology, Tools and Selected Use Cases, D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, and A. Wahler, Eds. Cham: Springer International Publishing, 2020, pp. 1–10. doi: 10.1007/978-3-030-37439-6/_1.
8. O. Semeniuta and P. Falkman, "EPypes: a framework for building event-driven data processing pipelines," PeerJ Comput. Sci., vol. 5, p. e176, Feb. 2019, doi: 10.7717/peerj-cs.176.
9. Stanford Artificial Intelligence Laboratory et al., Robotic Operating System. 2018. [Online]. Available: https://www.ros.org
10. "vvvv - a multipurpose toolkit," vvvv. https://vvvv.org/documentation/vvvv-a-multipurpose-toolkit (accessed May 17, 2022).
11. M. Noone and A. Mooney, "Visual and textual programming languages: a systematic review of the literature," J. Comput. Educ., vol. 5, no. 2, pp. 149–174, Jun. 2018, doi: 10.1007/s40692-018-0101-5.
12. T. Akam et al., "Open-source, Python-based, hardware and software for controlling behavioural neuroscience experiments," eLife, vol. 11, p. e67846, Jan. 2022, doi: 10.7554/eLife.67846.
13. J. L. Saunders and M. Wehr, "Autopilot: Automating behavioral experiments with lots of Raspberry Pis." bioRxiv, Oct. 17, 2019. doi: 10.1101/807693.
14. Pieter Hintjens, ZeroMQ Messaging for Many Applications, 1st ed. O'Reilly Media, 2013.
15. Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, "Detectron2." 2019. [Online]. Available: https://github.com/facebookresearch/detectron2
16. "python-statemachine," GitHub. https://github.com/fgmacedo/python-statemachine (accessed May 20, 2022).
17. R. Bitter, T. Mohiuddin, M. Nawrocki, LabVIEW: Advanced programming techniques. Crc Press 2006
18. MATLAB, Natick, Massachusetts: The MathWorks Inc.
19. K. Thulasiraman, M. N. S. Swamy, 5.7 Acyclic Directed Graphs. Graphs: Theory and Algorithms, John Wiley and Son, p. 118, ISBN 978-0-471-51356-8.
20. https://github.com/apache/airflow
21. M. Rocklin, Dask: Parallel computation with blocked algorithms and task scheduling. In Proceedings of the 14th python in science conference.
22. https://brodylabwiki.princeton.edu/bcontrol/index.php?title=Main_Page
23. https://github.com/fgmacedo/python-statemachine
24. https://github.com/alysivji/finite-state-machine