RatInABox: A toolkit for modelling locomotion and neuronal activity in continuous environments

Tom M George
Sainsbury Wellcome Centre
University College London
London, UK
tom.george.20@ucl.ac.uk

William de Cothi
Dept. of Cell and Developmental Biology
University College London
London, UK

Claudia Clopath*
Bioengineering Department
Imperial College London
London, UK

Kimberly Stachenfeld*
DeepMind
London, UK

Caswell Barry*
Dept. of Cell and Developmental Biology
University College London
London, UK

November 24, 2022

Abstract

Generating synthetic locomotory and electrophysiological data is a useful yet cumbersome step commonly required to study theoretical models of the brain's role in spatial navigation. This process can be time consuming and, without a common framework, makes it difficult to reproduce or compare studies which each generate test data in different ways. In response we present RatInABox, an open-source Python toolkit designed to model realistic rodent locomotion and generate synthetic electrophysiological data from spatially modulated cell types. This software provides users with (i) the ability to construct one- or two-dimensional environments with configurable barriers and rewards, (ii) a realistic motion model for random foraging fitted to experimental data, (iii) rapid online calculation of neural data for many of the known self-location or velocity selective cell types in the hippocampal formation (including place cells, grid cells, boundary vector cells, head direction cells) and (iv) a framework for constructing custom cell types as well as multi-layer network models. Cell activity and the motion model are spatially and temporally continuous and topographically sensitive to boundary conditions and walls. We demonstrate that out-of-the-box parameter settings replicate many aspects of rodent foraging behaviour such as velocity statistics and the tendency of rodents to over-explore walls. Numerous tutorial scripts are provided, including examples where RatInABox is used for decoding position from neural data or to solve a navigational reinforcement learning task. We hope this tool significantly streamlines the process of theory-driven research into the brain's role in navigation.

*Equal contribution

1 Introduction

Spurred on by the proposition of a "cognitive map"[1], and the later discovery of neurons with position-[2], velocity-[3, 4] and head direction-[5]selective receptive fields in the hippocampal formation, understanding the brain's role in navigation has been a key goal of the neuroscience, cognitive science, and psychology communities. It is commonplace for theoretical or computational models to rely on artificially generated datasets. This could be for direct testing of a normative model, or to feed a learning algorithm with training data (for example a motion model may be used to generate a time series of states, or feature-vectors). Not only is this data cheaper, faster and easier to obtain than experimental data (no rats required), but it can be flexibly hand-designed to rapidly prove or disprove theoretical propositions. Many past[6, 7] and recent[8, 9, 10, 11, 12, 13, 14] models have relied on artificially generated movement trajectories and neural data.

Artificially generating data, although faster than collecting from real experiments, can still be a bottleneck in the scientific process. We observe a number of issues: First and foremost, the lack of a universal standard for trajectory and cell activity modelling hinders apples-to-apples comparisons between theoretical models whose conclusions may differ depending on the specifics of the models being used. Secondly, researchers must begin each project by spending time writing software capable of generating pseudo-realistic trajectories and neural data before the more interesting theoretical work can begin. Thirdly, inefficiently written software can significantly slow down simulation time or, worse, push users to seek solutions which are more complex and power-intensive (multithreading, GPUs, etc.) than the underlying task requires, decreasing reproducibility. Finally, even the relatively modest complexities of motion modelling in continuous environments can impel researchers towards studying only one-dimensional environments or biologically unrealistic "gridworlds" with tabularised (or "one-hot") state spaces. Not only can gridworld models scale poorly in large environments but they typically disregard aspects of motion which can be non-trivial, for example speed variability and inertia. Of course there are good reasons why gridworld and/or tabularised state-space models may be preferred – and good open source packages for modelling this [15] – however, we suspect that coding simplicity, rather than theory-based justifications, remain a common reason these are used over continuous analogs.

To overcome these issues we built RatInABox (`https://github.com/TomGeorge1234/RatInABox`): an open source Python toolkit for efficient and realistic motion modelling in complex continuous environments *and* concurrent simulation of neuronal activity data for cell types typically found in the hippocampal formation (Fig. 1).

2 RatInABox

RatInABox is an open source software package comprising three component classes:

- `Environment`: The environment (or "box") that the `Agent` exists in. An `Environment` can be 1- or 2-dimensional, contain walls/barriers and have periodic or solid boundary conditions (Fig. 1a, b, d & e).
- `Agent`: The agent (or "rat") moving around the `Environment` (Fig. 1a, d & e).
- `Neurons`: A population of neurons whose firing rates update to encode the state of the `Agent`. Specific subclasses are provided corresponding to commonly studied cell-types (including, but not limited to, `PlaceCells`, `GridCells`, `BoundaryVectorCells` and `HeadDirectionCells`, Fig. 1b & c). Users can also write their own `Neurons` subclasses.

A typical workflow would be as follows: Firstly, an `Environment` is initialised with parameters specifying its dimensionality, size, shape and boundary conditions. Walls (or "barriers") can be added to make the `Environment` more complex. Secondly, an `Agent` is initialised with parameters specifying the characteristics of its motion (mean/standard deviation of its speed and rotational velocity, as well as behaviour near boundaries). Thirdly, populations of `Neurons` are initialised with parameters specifying their characteristics (number of cells, receptive field parameters, maximum firing rates etc.). Next, a period of simulated motion occurs: on each step the `Agent` updates its position and velocity within the `Environment`, given the duration of the step, and `Neurons` update their firing rates to reflect the new state of the `Agent`. After each step, data (timestamps, position, velocity, firing rates and spikes sampled according to an inhomogenous Poisson process) are saved into their respective classes for later analysis, Fig. 1.

RatInABox is fundamentally continuous in space and time. Position and velocity are never discretised but are instead stored as continuous values and used to determine cell activity online, as exploration occurs. This differs from other models which are either discrete (e.g. "gridworld")[15] or approximate continuous rate maps using a cached list of rates precalculated on a discretised grid of locations[9].

By default the `Agent` follows a temporally continuous smooth random motion model, closely matched to the statistics of rodent foraging in an open field[3] (Fig. 2); however, functionality is also provided for non-random velocity control (via a user provided control signal) or for the `Agent` to follow an imported trajectory (Fig. 3a). Once generated, data can be plotted using in-built plotting functions (which cover most of the figures in this manuscript) or extracted to be used in the theoretical model being constructed by the user.

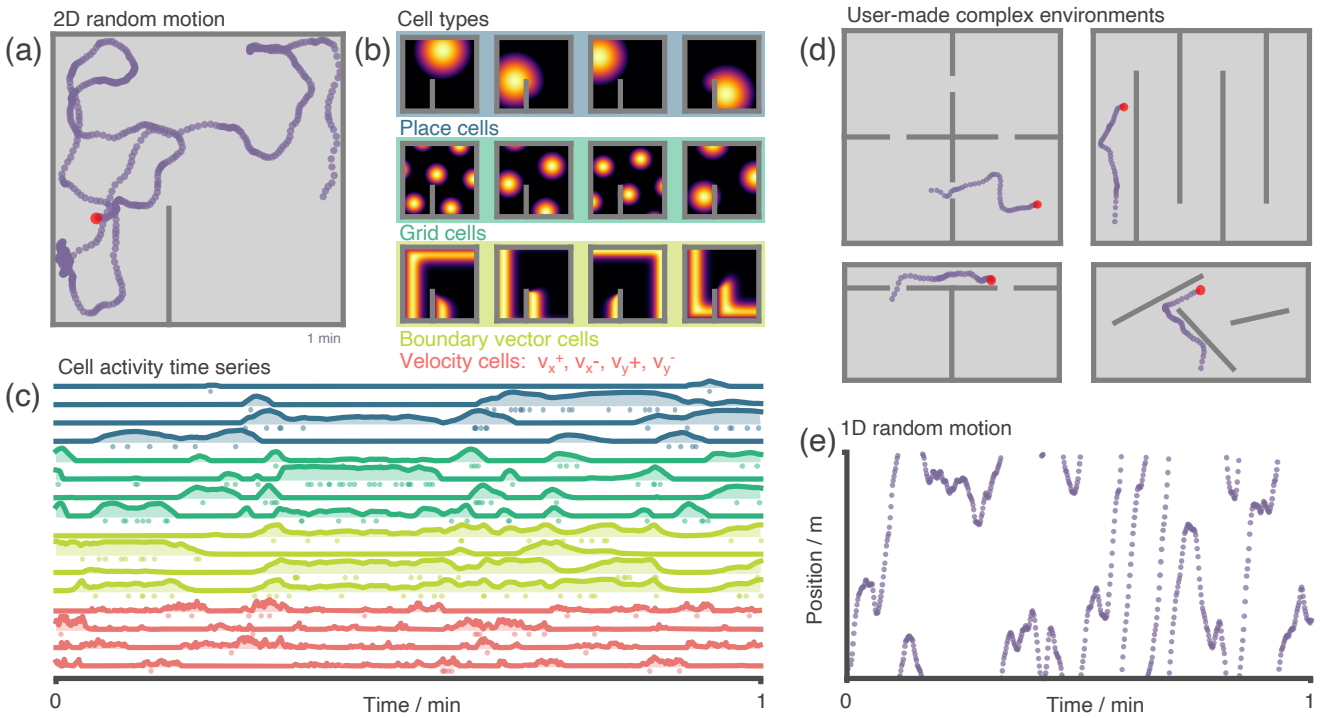

Figure 1: RatInABox is a flexible toolkit for simulating locomotion and electrophysiological data in complex continuous environments. **(a)** One minute of random motion in a 2D room with a wall. The red dot shows the current location of the `Agent`, whilst the purple dots show past locations. **(b)** Receptive fields of cells artificially generated using RatInABox. Cells interact appropriately with walls and boundary conditions. **(c)** Time series of cell activity (solid lines show firing rates, and dots show sampled spikes) for the trajectory shown in (a). **(d)** Users can easily construct complex environments by placing walls. **(e)** One minute of random motion in a 1D environment with periodic boundary conditions. Motion- and cell-model parameters used in this figure are package defaults, described in the Methods.

2.1 Intended use-cases

RatInABox can be used when locomotion and/or populations of cells need to be modelled in complex 1- or 2-dimensional continuous environments. These functionalities are coupled (locomotion directly adjusts the cell firing rates) but can also be used independently (for example an `Environment` and `Agent` can be modelled without any `Neurons` if users only need to sample a trajectory, or alternatively users can calculate cell activity on an imported trajectory without using the random motion model).

Below we briefly describe the most important details and features of RatInABox, divided into their respective classes. We leave all mathematical details to the Methods. Additional details (including example scripts and figures) can be found in the supplementary material and on the GitHub repository. The codebase itself is comprehensively commented and can be referenced for additional understanding where necessary.

2.2 The `Environment`

Unlike discretised models, where environments are stored as sets of nodes ('states') connected by edges ('actions')[15], here `Environments` are continuous domains defined by their `extent` (bounding coordinates) and a list of `walls` (line segments through which locomotion is not allowed). An `Environment` can have `'periodic'` or `'solid'` boundary conditions and can be `'1D'` or `'2D'`.

The ability to add walls to the `Environment` allows more complex mazes to be constructed to the users requirements (Fig. 1d). Boundaries and walls are thought to provide an important source of sensory data into the hippocampus[16, 17, 18, 19]. Their sensory inputs, and effect on the transition structure of the environment, may play an important role in determining cell activity during navigation[8, 9].

2.3 The `Agent`

Smooth and temporally continuous random motion Smooth and temporally continuous "random" motion can be difficult to model. To be smooth (and therefore physically plausible) a trajectory must be continuous in both position *and* velocity. To be temporally continuous, the statistics of the motion must be independent of the integration timestep being used. To be "random"

the autocorrelations of the mean-subtracted position and velocity must be expected to decay over behaviourally-appropriate timescales. Implementations of random motion models typically fail to satisfy one, or sometimes two, of these principles [20, 21].

Ornstein-Uhlenbeck processes, which sit at the heart of the RatInABox random motion model, are continuous-in-time random walks with a tendency to return to a central drift value. The decorrelation timescale can be also be controlled. We use these to update the velocity vector (linear and rotational velocities updated independently) on each update step. Position is then updated by taking a step along the velocity vector with some additional considerations to avoid walls. This method ensures both position and velocity are continuous, yet evolve "randomly" (Fig. 1a & e), and the statistics of the motion is independent of the size of the discretisation timestep being used.

Reanalysing rat locomotion data from Sargolini et al. [3] (as has been done before, by Raudies and Hasselmo [20]) we found that the histograms of linear speeds are well fit by a Rayleigh distributions whereas rotational velocities are approximately fit by normal distributions (Fig. 2a, top). Unlike Raudies and Hasselmo [20], we also extract the decorrelation timescale of these variables and observe that rotational velocity in real locomotion data decorrelates nearly an order of magnitude faster than linear velocity (0.08 s vs. 0.7 s). We set the default parameters of our Ornstein-Uhlenbeck processes (including applying a transform on the linear velocity so its long-run distribution also follows a Rayleigh distribution, see Methods) to those measured from the Sargolini et al. [3] dataset (Fig. 2b).

Motion near walls Animals rarely charge head-first into a wall, turn around, then continue in the opposite direction, as might be predicted by some naive motion models. Instead, they slow down smoothly and turn to avoid a collision. Additionally, during random foraging, rodents are observed to show a bias towards following walls, a behaviour known as thigmotaxis[22]. To replicate these observations, walls in the `Environment` lightly repel the `Agent` when it is close. Coupled with the finite turning speed this creates (somewhat counter-intuitively) a thigmotaxis-like effect where the agent over-explores walls and corners. A user-defined parameter called `''thigmotaxis''` can be used to control the strength of this emergent effect (Fig. 2d).

Imported trajectories RatInABox supports importing trajectory data which can be used instead of the inbuilt random motion model. Imported trajectory data points (which may be of low temporal-resolution) are interpolated (`scipy.interpolate`) using cubic splines and smoothly upsampled to user-define temporal precision (Fig. 3a). This upsampling is essential if one wishes to use low temporal resolution trajectory data to generate high frequency firing rate or spike data.

Trajectory control RatInABox supports active velocity control. At each integration step a target drift velocity can be specified, towards which the `Agent` accelerates (details in Methods). We anticipate this feature being used by models studying online control algorithms or the role of hippocampal representations for reinforcement learning (as we demonstrate in Fig. 3b).

2.4 `Neurons`

RatInABox provides multiple premade `Neurons` subclasses (some of which are shown in Fig. 1b). See Methods for additional mathematical details. These are:

- `PlaceCells`: A set of locations is sampled uniformly at random from across the `Environment` or provided manually, each defining the centre of a place field. The place cell firing rate is determined by the some function of the distance of the `Agent` from the centre of its place field. Provided functions are `'gaussian'`, `'gaussian_threshold'` (a gaussian cropped and levelled at 1 standard deviation), `'diff_of_gaussians'` (a wide Gaussian substracted from a narrower Gaussian with zero total volume, also known as a Ricker Wavelet or "Mexican Hat wavelet"), `'top_hat'` (fires uniformly only within a circle of specific radius) or `'one_hot'` (only the closest place cell to a given position will fire). This last item is useful for replicating tabular state spaces but with continuous motion.
- `GridCells`: Grid cells are modelled using a method originally proposed by Burgess et al. [23]. The receptive fields are given by the thresholded sum of three cosine waves at 60° with randomised or user-defined grid scales, phase offsets and orientations.
- `BoundaryVectorCells` (BVCs): BVCs are modelled similar to the double-Gaussian model used by Hartley et al. [17]. Each BVC has a preferred distance and orientation at which a boundary segment will maximally stimulate it. Users can decide if BVCs are "allocentric" (default) or "egocentric"[24]. In the past, modelling BVCs has been a significant bottleneck to modelling studies. RatInABox uses fully-vectorised calculations to determine BVC firing rates rapidly. Calculating the firing rate of 1000 BVCs takes $\mathcal{O}$(10 ms) on a consumer laptop in 2022.
- `VelocityCells`: In 1D (2D), 2 (4) velocity cells encode the positive and negative velocities v_x^+ and v_x^- (and v_y^+ and v_y^-) with positive firing rates.
- `SpeedCells`: A single cell fires proportional to the scalar speed of the `Agent`.
- `HeadDirectionCells`: Like `VelocityCells` but the velocity is normalised so that the firing rate encodes head direction information only.

- `FeedForwardLayer`: Given a set of inputs (other `Neurons`) the firing rate is given by an activated weighted sum of over all inputs. This layer forms the building block for creating more complex multi-layer networks of `Neurons`.
- `PhasePrecessingPlaceCells:` A subclass of `PlaceCells` which display phase precession[25] with respect to a background LFP theta-oscillation.

A dedicated space containing additional cell classes not described here, is made available for community contributions to this list.

Geometry and boundary conditions In RatInABox, `PlaceCells` are sensitive to walls within the `Environment`. Three distance geometries are supported: `'euclidean'` geometry calculates the Euclidean distance to a place field centre and so cell activity will 'bleed' through walls and boundaries as if they weren't there. `'line_of_sight'` geometry allows a place cell to fire only if there is direct line-of-sight to the place field centre from the current location. Finally `'geodesic'` geometry (default) calculates distance according to the shortest available walk to the cell centre, respecting walls and boundaries (notice smooth wrapping of the second and fourth place cells around the wall in Fig. 1b). The latter two geometries respect the observation that place fields don't typical pass through walls. This geometry distinction only applies to `PlaceCells` and not `GridCells` or `BoundaryVectorCells` (see Methods).

Boundary conditions can be `periodic` or `solid`. In the former case, place field corresponding to cells near the boundaries of the environment will wrap around.

Rate maps It is possible to query the firing rate of a population of `Neurons` at an array of locations simultaneously. This computation is vectorised (also referred to as "array programming") and extremely fast, allowing ground truth rate maps to be calculated and plotted with ease (as shown in Fig. 1b). Alternatively, rate maps can be 'inferred' after a period of exploration by generating a histogram over past positions weighted by the firing rate observed at each position. This technique is more robust when determining rate maps for `Neurons` classes which don't necessarily have analytically well-defined spatial receptive fields (for example neurons with recurrent connectivity, or velocity neurons, figures shown on the GitHub readme). It also more closely matches how rate maps are plotted from biological neuronal data by smoothing spike raster plots (which can also be plotted using RatInABox).

Make your own neurons and networks of neurons `PlaceCells`, `GridCells`, `BoundaryVectorCells` etc. are all subclasses of the more general `Neurons` class. We make it easy for users to write, and contribute, their own cell types in this manner (instructions and examples are provided). We highlight `FeedForwardLayer` neurons as a particularly notable subclass for those users interested in using RatInABox to build multilayer networks (demonstrated in Fig. S2). Unlike others subclasses, the firing rate of `FeedForwardLayer` neurons is not determined explicitly by the state of the `Agent` but rather by a weighted sum of the firing rates from neurons in a provided list of input layers. Users can write their own cell classes as a subclass of `FeedForwardLayer` or use it as a guide for how to make their own from scratch.

3 Demonstration and Results

The default parameters of the random motion model in RatInABox are matched to observed statistics of rodent locomotion, extracted by reanalysing data from Sargolini et al. [3][2]. Trajectories and statistics from the real data (Fig. 2a) can then be directly compared to the artificially generated trajectories from RatInABox (Fig. 2b). Further, rodents have a tendency to over-explore walls and corners, a bias that is often linked to neophobia as it is particularly pronounced when the animal is new to the environment (Fig. 2c, data from Satoh et al. [22]). This bias is also present in the artificial trajectories generated by RatInABox - the strength of which can be set by the parameter `Agent.thigmotaxis` (Fig. 2d).

RatInABox can import and smoothly interpolate user-provided trajectory data. For example, when a 50 Hz trajectory (Fig. 3a (left) data from Sargolini et al. [3]) is downsampled to 1.66 Hz, the `Agent` will move along a trajectory that is automatically upsampled using cubic spline interpolation to the user-specified sampling rate of the `Agent`, often allowing a smooth replica of the ground truth trajectory (Fig. 3a, right). Note that without upsampling, this data (1.66 Hz) would be far too low in temporal-resolution to usefully simulate neural activity. For convenience, the exact Sargolini et al. [3] datafile used in Figure 3a is uploaded with permission to the GitHub repository and can be imported using `Agent.import_trajectory(dataset="sargolini")`. An additional trajectory dataset from a much larger environment is also supplied with permission from Tanni et al. [26].

Finally, RatInABox is computationally efficient. We compare the time taken for a consumer-grade CPU (MacBook Pro, Apple M1 Max) to update the random motion model and to calculate the firing rates of large ($n = 1000$) populations of cells (Fig. 3c, purple) to how long it takes to perform typical vector calculus operations (on vectors/matrices also of size 1000) evaluated using the Python package `numpy` (Fig. 3c, red). The red bars represent the probable 'bottlenecking' operation in a downstream analysis or model-training procedure for which RatInABox is providing data. The time taken for a motion model update step , as well as place cell and grid cell updates, is very close to how long it takes to multiply a vector by a matrix and two orders of magnitude faster than inverting a matrix. Boundary vector cells are significantly slower to simulate (because they require integrating around

[2] data freely available at: `https://www.ntnu.edu/kavli/research/grid-cell-data`,
exact filename used: `8F6BE356-3277-475C-87B1-C7A977632DA7_1/11084-03020501_t2c1.mat`

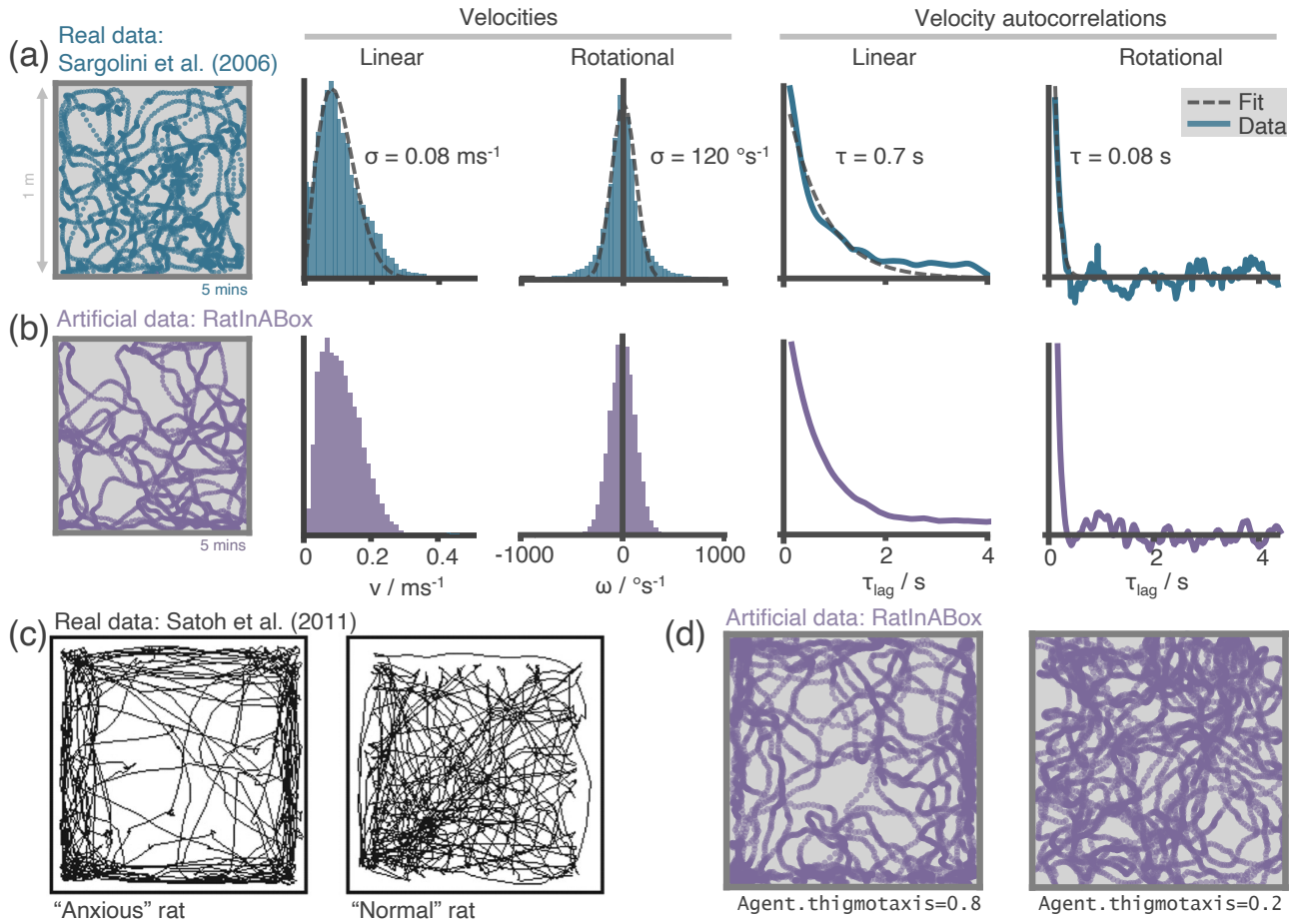

Figure 2: The RatInABox random motion model closely matches features of real rat locomotion. **(a)** An example 5 minute trajectory from the Sargolini et al. [3] dataset. Linear velocity (Rayleigh fit) and rotational velocity (Gaussian fit) histograms and the temporal autocorrelations (exponential fit) of their time series'. **(b)** A sampled 5 minute trajectory from the RatInABox motion model with parameters matched to Sargolini data. **(c)** Data from Satoh et al. [22], 10 minutes of open-field exploration. "Thigmotaxis" is the tendency of rodents to over-explore near boundaries/walls and has been linked to anxiety. **(d)** RatInABox replicates the tendency of agents to over-explore walls and corners, flexibly controlled with a `'thigmotaxis'` parameter. 10 minutes of trajectory are shown in both panels of (c) and (d).

a 360° field-of-view, see Methods) but still outpace the matrix inverse. These results imply that, depending on the details of the use-case, RatInABox will likely not be a significant bottleneck.

Our testing (not shown) reveals that `Neuron` update times scale approximately linearly with n for large populations $n >> 100$ and sublinearly for small populations where other computations (querying positions, saving data etc.) dominate. `PlaceCells`, `BoundaryVectorCells` and the `Agent` motion model update times will be additionally affected by the number of walls/barriers in the `Environment`. 1D simulations are significantly quicker than 2D simulations (those shown in Fig. 3c are 2D) due to the reduced computational load of the 1D geometry.

3.1 Case studies

We envisage RatInABox being used to support a range of theoretical studies by providing data and, if necessary, infrastructure for building models powered by this data. This 'Bring-Your-Own-Algorithm' approach makes the toolkit generally applicable, not specialised to one specific field. Two example use-cases are provided in the supplement and are briefly described below. The intention is (i) to demonstrate the capacity of RatInABox for use in varied types of computational studies and (ii) to provide tutorials as a tool for learning how to use the package. Accompanying notebooks are provide on the Github repository.

In our first example we perform a simple experiment where location is decoded from neural firing rates (Fig. S1). Data – the location and firing rate trajectories of an `Agent` randomly exploring a 2D `Environment` – are generated using RatInABox. Gaussian process regression is used to predict position from firing rates on a held-out testing dataset. We compare the accuracy of decoding using different cell types; place cells, grid cells and boundary vector cells.

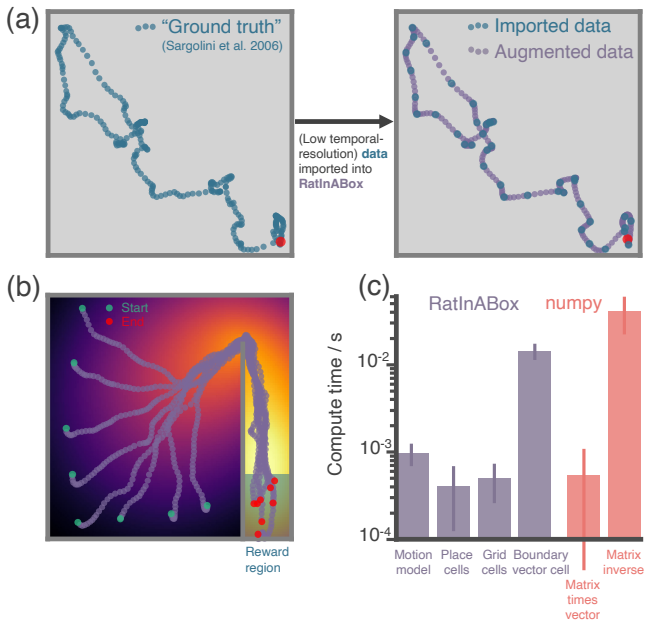

Figure 3: Data and computational efficiency of RatInABox. **(a)** Low temporal-resolution trajectory data (downsampled from 50 Hz to 1.66 Hz) imported into RatInABox is upsampled ("augmented") using cubic spline interpolation. **(b)** RatInABox used in a simple reinforcement learning example. A policy iteration technique converges onto an optimal value function (heatmap) and policy (trajectories) for an `Environment` where a reward is hidden behind a wall. Continuous state encoding, continuous policy control and the `Environment` are handled naturally by RatInABox. **(c)** Comparison of compute times for a consumer grade CPU-only laptop. (Left) Typical RatInABox operations; motion model update and cell firing rate updates for cell populations ($N = 1000$). (Right) Typical vector calculus operations for vectors/matrices of size 1000. Updating boundary vector cells is significantly slower than place or grid cells (note log-scale) but still outpaces a size-matched matrix inverse.

Next, we demonstrate the application of RatInABox to a simple reinforcement learning (RL) task (Fig. 3b and Fig. S2). A small network capable of model-free RL is constructed and trained using RatInABox. First a neuron calculates and learns – using a variant of temporal difference learning – the value function as a linear combination of `PlaceCell` basis features.Then a new 'improved' policy is defined by setting a drift velocity – which biases the `Agents` motion – proportional to the gradient of the value function. The `Agent` is therefore encouraged to move towards the region with higher value. Iterating between these stages over many episodes ("policy iteration") results in convergence towards near optimal behaviour where the `Agent` takes the shortest route to the reward, avoiding the wall (Fig. 3b).

Finally, in an additional tutorial not described in the supplement but available online we demonstrate how RatInABox can be used to build a circuit capable of learning to path integrate position from velocity signals using Hebbian-style learning rules.

Despite including these examples we stress that neural decoding, reinforcement learning and path integrationare only some of the many fields where RatInABox might prove useful.

4 Discussion

RatInABox is a lightweight, open-source toolkit for generating realistic trajectory and neural data in complex continuous environments. It should be particularly useful to those studying spatial navigation and the role of the hippocampal formation. It remains purposefully small in scope - intended primarily as a means for generating data. We do not provide, nor intend to provide, a set of benchmark learning algorithms to use on the data it generates. Specifically because RatInABox remains agnostic to how its data is used we think it is flexible enough to empower a wide variety of users to more rapidly build, train and validate models of hippocampal function and spatial navigation, accelerating progress in the field.

We intend to maintain this repository and welcome community contributions, for example new cell types and datasets, and provide a dedicated `contribs` directory for this purpose. If there is a small contribution you would like to make, please open a pull request. If there is a larger contribution you are considering, please contact the corresponding author[3].

[3] `tomgeorge1@btinternet.com`

5 Acknowledgements

We would like to thank Peter Vincent and Gastón Sivori for helpful discussions and comments on the manuscript.

References

[1] Edward C. Tolman and C.H. Honzik. Introduction and removal of reward, and maze performance in rats. *University of California, Publications in Psychology*, 4:257–275, 1930.

[2] J. O'Keefe and J. Dostrovsky. The hippocampus as a spatial map. Preliminary evidence from unit activity in the freely-moving rat. *Brain Research*, 34(1):171–175, 1971. ISSN 0006-8993. doi: 10.1016/0006-8993(71)90358-1.

[3] Francesca Sargolini, Marianne Fyhn, Torkel Hafting, Bruce L. McNaughton, Menno P. Witter, May-Britt Moser, and Edvard I. Moser. Conjunctive Representation of Position, Direction, and Velocity in Entorhinal Cortex. *Science*, 312(5774): 758–762, 2006. ISSN 0036-8075. doi: 10.1126/science.1125572.

[4] Emilio Kropff, James E. Carmichael, May-Britt Moser, and Edvard I. Moser. Speed cells in the medial entorhinal cortex. *Nature*, 523(7561):419–424, 2015. ISSN 0028-0836. doi: 10.1038/nature14622.

[5] JS Taube, RU Muller, and JB Ranck. Head-direction cells recorded from the postsubiculum in freely moving rats. I. Description and quantitative analysis. *The Journal of Neuroscience*, 10(2):420–435, 1990. ISSN 0270-6474. doi: 10.1523/jneurosci.10-02-00420.1990.

[6] Mayank R. Mehta, Michael C. Quirk, and Matthew A. Wilson. Experience-dependent asymmetric shape of hippocampal receptive fields. *Neuron*, 25(3):707–715, March 2000. doi: 10.1016/s0896-6273(00)81072-7. URL `https://doi.org/10.1016/s0896-6273(00)81072-7`.

[7] Yoram Burak and Ila R. Fiete. Accurate Path Integration in Continuous Attractor Network Models of Grid Cells. *PLoS Computational Biology*, 5(2):e1000291, 2009. ISSN 1553-734X. doi: 10.1371/journal.pcbi.1000291.

[8] Kimberly L Stachenfeld, Matthew M Botvinick, and Samuel J Gershman. The hippocampus as a predictive map. *Nat Neurosci*, 20(11):1643–1653, 2017. ISSN 1097-6256. doi: 10.1038/nn.4650. URL `http://www.nature.com/articles/nn.4650`.

[9] William Cothi and Caswell Barry. Neurobiological successor features for spatial navigation. *Hippocampus*, 30(12): 1347–1355, 2020. ISSN 1050-9631. doi: 10.1002/hipo.23246. URL `https://onlinelibrary.wiley.com/doi/10.1002/hipo.23246`.

[10] Jacopo Bono, Sara Zannone, Victor Pedrosa, and Claudia Clopath. Learning predictive cognitive maps with spiking neurons during behaviour and replays. *bioarxiv*, page 2021.08.16.456545, 2021. URL `https://www.biorxiv.org/content/10.1101/2021.08.16.456545v2`. Company: Cold Spring Harbor Laboratory Distributor: Cold Spring Harbor Laboratory Label: Cold Spring Harbor Laboratory Section: New Results Type: article.

[11] Tom M George, William de Cothi, Kimberly Stachenfeld, and Caswell Barry. Rapid learning of predictive maps with STDP and theta phase precession. *bioRxiv*, page 2022.04.20.488882, 2022. doi: 10.1101/2022.04.20.488882.

[12] Andrea Banino, Caswell Barry, Benigno Uria, Charles Blundell, Timothy Lillicrap, Piotr Mirowski, Alexander Pritzel, Martin J. Chadwick, Thomas Degris, Joseph Modayil, Greg Wayne, Hubert Soyer, Fabio Viola, Brian Zhang, Ross Goroshin, Neil Rabinowitz, Razvan Pascanu, Charlie Beattie, Stig Petersen, Amir Sadik, Stephen Gaffney, Helen King, Koray Kavukcuoglu, Demis Hassabis, Raia Hadsell, and Dharshan Kumaran. Vector-based navigation using grid-like representations in artificial agents. *Nature*, 557(7705):429–433, 2018. ISSN 0028-0836. doi: 10.1038/s41586-018-0102-6. URL `http://www.nature.com/articles/s41586-018-0102-6`.

[13] Rylan Schaeffer, Mikail Khona, and Ila R Fiete. No free lunch from deep learning in neuroscience: A case study through models of the entorhinal-hippocampal circuit. In *ICML 2022 2nd AI for Science Workshop*, 2022. URL `https://openreview.net/forum?id=mxi1xKzNFrb`.

[14] Marcus K. Benna and Stefano Fusi. Place cells may simply be memory cells: Memory compression leads to spatial tuning and history dependence. *Proceedings of the National Academy of Sciences*, 118(51):e2018422118, 2021. doi: 10.1073/pnas.2018422118. URL `https://www.pnas.org/doi/abs/10.1073/pnas.2018422118`.

[15] Arthur Juliani, Samuel Barnett, Brandon Davis, Margaret Sereno, and Ida Momennejad. Neuro-Nav: A Library for Neurally-Plausible Reinforcement Learning. *arXiv*, 2022.

[16] John O'Keefe and Neil Burgess. Geometric determinants of the place fields of hippocampal neurons. *Nature*, 381(6581): 425–428, 1996. ISSN 0028-0836. doi: 10.1038/381425a0.

[17] T. Hartley, N. Burgess, C. Lever, F. Cacucci, and J. O'Keefe. Modeling place fields in terms of the cortical inputs to the hippocampus. *Hippocampus*, 10(4):369–379, 2000. ISSN 1098-1063. doi: 10.1002/1098-1063(2000)10:4<369::aid-hipo3>3.0.co;2-0.

[18] C. Barry, C. Lever, R. Hayman, T. Hartley, S. Burton, J. O'Keefe, K. Jeffery, and N. Burgess. The Boundary Vector Cell Model of Place Cell Firing and Spatial Memory. *Reviews in the Neurosciences*, 17(1-2):71–98, 2006. ISSN 0334-1763. doi: 10.1515/revneuro.2006.17.1-2.71.

[19] Trygve Solstad, Charlotte N. Boccara, Emilio Kropff, May-Britt Moser, and Edvard I. Moser. Representation of Geometric Borders in the Entorhinal Cortex. *Science*, 322(5909):1865–1868, 2008. ISSN 0036-8075. doi: 10.1126/science.1166466.

[20] Florian Raudies and Michael E. Hasselmo. Modeling Boundary Vector Cell Firing Given Optic Flow as a Cue. *PLoS Comput Biol*, 8(6):e1002553, 2012. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1002553. URL `https://dx.plos.org/10.1371/journal.pcbi.1002553`.

[21] Marcus K. Benna and Stefano Fusi. Place cells may simply be memory cells: Memory compression leads to spatial tuning and history dependence. *Proceedings of the National Academy of Sciences*, 118(51):e2018422118, 2021. ISSN 0027-8424. doi: 10.1073/pnas.2018422118.

[22] Y. Satoh, S. Endo, T. Nakata, Y. Kobayashi, K. Yamada, T. Ikeda, A. Takeuchi, T. Hiramoto, Y. Watanabe, and T. Kazama. ERK2 contributes to the control of social behaviors in mice. *Journal of Neuroscience*, 31(33):11953–11967, August 2011. doi: 10.1523/jneurosci.2349-11.2011. URL `https://doi.org/10.1523/jneurosci.2349-11.2011`.

[23] Neil Burgess, Caswell Barry, and John O'Keefe. An oscillatory interference model of grid cell firing. *Hippocampus*, 17(9): 801–812, 2007. ISSN 1098-1063. doi: 10.1002/hipo.20327.

[24] Patrick Byrne, Suzanna Becker, and Neil Burgess. Remembering the past and imagining the future: A neural model of spatial memory and imagery. *Psychological Review*, 114(2):340–375, apr 2007. doi: 10.1037/0033-295x.114.2.340. URL `https://doi.org/10.1037%2F0033-295x.114.2.340`.

[25] John O'Keefe and Michael L. Recce. Phase relationship between hippocampal place units and the EEG theta rhythm. *Hippocampus*, 3(3):317–330, 1993. ISSN 1098-1063. doi: 10.1002/hipo.450030307.

[26] Sander Tanni, William de Cothi, and Caswell Barry. State transitions in the statistically stable place cell population correspond to rate of perceptual change. *Current Biology*, 2022. ISSN 0960-9822. doi: 10.1016/j.cub.2022.06.046.

[27] A. Jeewajee, C. Barry, V. Douchamps, D. Manson, C. Lever, and N. Burgess. Theta phase precession of grid and place cell firing in open environments. *Phil. Trans. R. Soc. B*, 369(1635):20120532, 2014. ISSN 0962-8436. doi: 10.1098/rstb.2012.0532. URL `https://royalsocietypublishing.org/doi/10.1098/rstb.2012.0532`.

[28] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Copyright Cambridge University Press, 2003.

[29] Kenji Doya. Reinforcement Learning in Continuous Time and Space. *Neural Computation*, 12(1):219–245, 2000. ISSN 0899-7667. doi: 10.1162/089976600300015961.

6 Methods and mathematical details

The following section describes in mathematical detail the models used within RatInABox. Table S1 compiles a list of all important parameters along with their default values, allowed ranges and how they can be adjusted.

6.1 Motion model

6.1.1 Temporally continuous random motion

Our random motion model is based on the Ornstein Uhlenbeck process, $\mathsf{X}_{\theta,\lambda,\mu}(t)$ which is a stochastic process satisfying the Langevin differential equation

$$\begin{aligned} \mathsf{X}_{\theta,\lambda,\mu}(t+dt) &= \mathsf{X}_{\theta,\lambda,\mu}(t) + d\mathsf{X}_{\theta,\lambda,\mu}(t) \\ d\mathsf{X}_{\theta,\lambda,\mu}(t) &= \theta(\mu - \mathsf{X}_{\theta,\lambda,\mu}(t))dt + \lambda\eta(t)\sqrt{dt} \end{aligned} \tag{1}$$

where $\eta(t) \sim \mathcal{N}(0,1)$ is Gaussian white noise and θ, λ and μ are constants . The first term in the update equation drives decay of $\mathsf{X}_{\theta,\lambda,\mu}(t)$ towards the mean μ. The second term is a stochastic forcing term, driving "randomness". These stochastic processes are well studied; their unconditioned covariance is

$$\langle \mathsf{X}_{\theta,\lambda,\mu}(t)\mathsf{X}_{\theta,\lambda,\mu}(t')\rangle = \frac{\lambda^2}{2\theta}e^{-\theta|t-t'|}. \tag{2}$$

Thus $\mathsf{X}_{\theta,\lambda,\mu}(t)$ decorrelates smoothly over a timescale of $\tau = 1/\theta$. Over long periods $\mathsf{X}_{\theta,\lambda,\mu}(t)$ is stochastic and therefore unpredictable. Its long-run stationary probability distribution is a Gaussian with mean μ and standard deviation $\sigma = \sqrt{\lambda^2/2\theta}$. We can re-parameterise the Ornstein Uhlenbeck process in terms of these more "intuitive" parameters (the decoherence timescale τ and the long-run standard deviation σ) using the transformations

$$\theta = \frac{1}{\tau} \quad , \quad \lambda = \sqrt{\frac{2\sigma^2}{\tau}}, \tag{3}$$

to give

$$\mathsf{X}_{\tau,\sigma,\mu}(t+dt) = \mathsf{X}_{\tau,\sigma,\mu}(t) + d\mathsf{X}_{\tau,\sigma,\mu}(t),$$
$$d\mathsf{X}_{\tau,\sigma,\mu}(t) = \frac{1}{\tau}(\mu - \mathsf{X}_{\tau,\sigma,\mu}(t))dt + \sqrt{\frac{2\sigma^2}{\tau}}\eta(t)\sqrt{dt}. \tag{4}$$

Ornstein Uhlenbeck processes have the appealing property that they are temporally continuous (their statistics are independent of dt) and allow for easy control of the long-run standard deviation and the decoherence timescale of the stochastic variable. For these reasons we use use them to model rotational and linear velocities.

For 2D locomotion we sample rotational velocity $\omega(t) = \dot{\phi}(t)$ and linear speed, $\mathsf{v}_{2D}(t) = \|\mathbf{v}(t)\|$, independently, where the total velocity vector is given by

$$\mathbf{v}(t) = \dot{\mathbf{x}}(t) = \mathsf{v}_{2D}(t)\begin{bmatrix}\cos\phi(t)\\ \sin\phi(t)\end{bmatrix}. \tag{5}$$

We sample these independently because, as shown in section 3, they have decoherence timescales differing by an order of magnitude. Rotational velocity is sampled from a standard Ornstein Uhlenbeck process with zero mean. Linear speed is also derived from an Ornstein Uhlenbeck process with one additional transform applied in order to match the observation that linear speeds have a Rayleigh, not normal, distribution. For 1D locomotion velocity is also modelled as an Ornstein Uhlenbeck process without the Rayleigh transform. In this case a non-zero mean, corresponding to directional bias in the motion, can be provided. In summary:

$$\omega(t) \sim \mathsf{X}_{\tau_\omega,\sigma_\omega,0}(t), \tag{6}$$
$$\mathsf{v}_{2D}(t) = R_{\sigma_\mathsf{v}}(z(t)) \quad \text{where} \quad z(t) \sim \mathsf{X}_{\tau_\mathsf{v},1,0}(t), \tag{7}$$
$$\mathsf{v}_{1D}(t) \sim \mathsf{X}_{\tau_\mathsf{v},\sigma_\mathsf{v},\mu_\mathsf{v}}(t) \tag{8}$$

where $R_\sigma(x)$ is a monotonic transformation which maps a normally distributed random variable $x \sim \mathcal{N}(0,1)$ to one with a Rayleigh distribution of scale parameter σ ($R_\sigma(x) \sim \text{Rayleigh}(\sigma)$). Note σ corresponds to the mode, or ≈ 0.8 times the mean, of the Rayleigh distribution).

$$R_\sigma(x) = \sigma\sqrt{-2\ln\Big(1 - \frac{1}{2}\Big[1 + \text{erf}\Big(\frac{x}{\sqrt{2}}\Big)\Big]\Big)}. \tag{9}$$

The parameters $\{\tau_\omega, \sigma_\omega, \tau_\mathsf{v}, \sigma_\mathsf{v}\}$ are fitted from real open field 2D locomotion data in Fig. 2 or can be set by the user (see table S1).

Finally, these velocities are integrated using a timestep dt (see table S1) to give continuous and smooth, but over long time periods random, motion.

6.1.2 External velocity control

It is possible to provide an external velocity signal controlling the `Agents` motion. After the random motion update (as described above) is applied, if an external velocity $\mathbf{v}_{\text{drift}}(t)$ is provided by the user, an additional update to the velocity vector is performed

$$d\mathbf{v}(t+dt) = \frac{1}{\tau_{\text{drift}}}(\mathbf{v}_{\text{drift}}(t) - \mathbf{v}(t))dt. \tag{10}$$

In cases where $\tau_{\text{drift}} >> \tau_\mathsf{v}$ the net update to the velocity (random update and drift update) is dominated by the random component. When $\tau_{\text{drift}} << \tau_\mathsf{v}$ the update is dominated by the drift component. We define $\tau_{\text{drift}} := \tau_\mathsf{v}/k$ where k is an argument also provided by the user. To good approximation for large $k >> 1$ the `Agent` velocity closely tracks the drift velocity at all times and is not random whilst for $k << 1$ the drift velocity is ignored and the motion is entirely random.

6.1.3 Motion near walls in 2D

An important feature is the ability to generate `Environments` with arbitrary arrangements of walls (aka 'barriers'). Walls are meaningful only if they appropriately constrain the motion of the `Agent`. For biological agents this means three things: (i) The agent cannot travel through a wall, (ii) the agent slows down upon approaching a wall to avoid a full-speed collision and (iii) there is a bias (not always observed) called "thigmotaxis" to stay near walls. Our motion model replicates these three effects as follows:

To avoid travelling through walls, if a collision is detected the velocity is elastically reflected off the wall (normal component is flipped). The speed is then scaled to one half the average motion speed, $\mathsf{v}_{2D}(t) = 0.5\sigma_\mathsf{v}$.

In order to slow down *before* colliding with a wall the `Agent` feels an acceleration, perpendicular to the wall, whenever it is within a small distance, d_{wall}, of the wall.

$$\dot{\mathbf{v}}(t) = k_1 \sum_{\text{walls},j} \mathbf{n}_j \begin{cases} \frac{\sigma_\mathsf{v}^2}{d_{\text{wall}}^2} \cdot (d_{\text{wall}} - d_{\perp,j}(t)) & \text{if } d_{\perp,j}(t) \le d_{\text{wall}}, \\ 0 & \text{if } d_{\perp,j}(t) > d_{\text{wall}}. \end{cases} \tag{11}$$

$d_{\perp,j}(t)$ is the perpendicular distance from the `Agent` to the jth wall, $\mathbf{n}_j$ is the perpendicular norm of the jth wall (the norm pointing towards the `Agent`) and k_1 is a constant (explained later). Note that this acceleration is identical to that of an oscillating spring-mass where the base of the spring is attached a distance d_{wall} from the wall on a perpendicular passing through the `Agent`. The spring constant is tuned such that a mass starting with initial velocity towards the wall of $-\sigma_{\text{v}}\mathbf{n}_j$ would stop *just* before the wall. In summary, for $k_1 = 1$, if the `Agent` approaches the wall head-on at speed of σ_{v} (its mean speed) this deceleration will just be enough to avoid a collision.

A second way to slow down motion near a wall is to consider a hypothetical conveyor belt near the wall. This conveyor belt has a non-uniform velocity pointing away from the wall of

$$\dot{\mathbf{x}}(t) = k_2 \sum_{\text{walls},j} \mathbf{n}_j \begin{cases} \sigma_{\text{v}}\left(1 - \sqrt{1 - \frac{(d_{\text{wall}} - d_{\perp,j}(t))^2}{d_{\text{wall}}^2}}\right) & \text{if } d_{\perp,j}(t) \leq d_{\text{wall}}, \\ 0 & \text{if } d_{\perp,j}(t) > d_{\text{wall}}. \end{cases} \tag{12}$$

Note that this velocity is identical to that of a spring-mass attached to the wall with initial velocity $\sigma_{\text{v}}\mathbf{n}_j$ away from the wall and spring constant tuned to stop the mass just before it reaches a distance d_{wall}. In summary, for $k_2 = 1$, if the `Agent` approaches the wall head-on at speed of σ_{v} the conveyor belt will just be fast enough to bring it to a halt at the location of the wall.

Although similar, there is one crucial and exploitable difference between the 'spring-deceleration' and 'conveyor-belt' models: the 'conveyor belt' changes the `Agents` position $\mathbf{x}(t)$ on each step but not its internal velocity variable $\mathbf{v}(t)$. As as result (and as the conveyor-belt intuition suggests) it will slow down the `Agent` without causing it to turn around. This creates a "lingering" or "thigmotaxis" effect whereby whenever the `Agent` heads towards a wall it may carry on doing so, without collision, for some time until the stochastic processes governing its motion (section 6.1.1) cause it to turn. Conversely the 'spring-deceleration' model has no "thigmotaxis" effect since it actively changes the velocity causing the `Agent` to bounce off the walls.

k_1 and k_2 are controlled by a thigmotaxis parameter $\lambda_{\text{thig}} \in [0, 1]$ to govern the trade-off between these two models.

$$k_1 = 3(1 - \lambda_{\text{thig}})^2, \qquad k_2 = 6\lambda_{\text{thig}}^2. \tag{13}$$

When $\lambda_{\text{thig}} = 1$ only the conveyor belt model is active giving strong thigmotaxis effects. When $\lambda_{\text{thig}} = 0$ only the spring-deceleration model is active giving no thigmotaxis effect. By default $\lambda_{\text{thig}} = 0.5$. The constants 3 and 6 are tuning parameters chosen by hand in order that direct collisions with the walls are rare but not impossible.

Although this procedure, merely intended to smoothly slow the `Agent` near a wall, may seem complex, it has a two advantages: Firstly, deceleration near walls is smooth, becoming stronger as the `Agent` gets nearer and so induces no physically implausible discontinuities in the velocity. Secondly, it provides a tunable way by which to control the amount of thigmotaxis (evidenced in Fig. 2c & d). Recall that these equations only apply to motion very near the wall and they can be turned off entirely (equivalent to setting $k_1 = k_2 = 0$, see table S1).

6.2 Cell models

In the following section we list mathematical models for each of the default provided `Neurons` subclasses. Readers will note that, oftentimes, parameters are set randomly at the point of initialisation (e.g. where the place cells are located, the orientation of grid cells, the angular preference of boundary vector cells etc.). These random parameters are all set as class attributes and so can be redefined after initialisation if necessary. For simplicity here we describe default behaviour only – the default values for all parameters and how to change them are given in table S1.

For most of the below-listed cell classes it is also possible to set their maximum and minimum firing rates (f_{max}, f_{min}). For simplicity the formulae provided below are written such that they have a maximum of 1 and minimum of 0 but reader should be aware that after evaluation these are then linearly scaled according to

$$F(t) \leftarrow (f_{\text{max}} - f_{\text{min}})F(t) + f_{\text{min}}. \tag{14}$$

6.2.1 `PlaceCells`

A set of locations (the centre of the place fields), $\{\mathbf{x}_i\}$, is randomly sampled from the `Environment`. By default these locations sit on a grid uniformly spanning the `Environment` to which a small amount of random jitter (half the scale of the sampled grid) is added. Thus place cell locations appear 'random' but initialising in this way ensures all parts of the `Environment` are approximately evenly covered with the same density of place cells.

The distance from the `Agent` to the place field centres, respecting walls and the desired type of distance geometry (`euclidean`, `line_of_sight`, `geodesic`), is calculated $d_i(t) = \|\mathbf{x}_i - \mathbf{x}(t)\|$. The firing rate is then determined by one of the following

functions (defaulting to $F^{\texttt{gaussian}}$):

$$F_i^{\texttt{gaussian}}(t) = e^{-d_i^2/2w_i^2} \tag{15}$$

$$F_i^{\texttt{gaussian_threshold}}(t) = \max\left(0, \frac{e^{-d_i^2/2w_i^2} - e^{-1/2}}{1 - e^{-1/2}}\right) \tag{16}$$

$$F_i^{\texttt{diff_of_gaussians}}(t; r = 1.5) = \frac{e^{-d_i^2/2w_i^2} - (1/r^2)e^{-d_i^2/2(rw_i)^2}}{1 - 1/r^2} \tag{17}$$

$$F_i^{\texttt{top_hat}}(t) = \begin{cases} 1 & \text{if } d_i \leq w_i \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

$$F_i^{\texttt{one_hot}}(t) = \delta(i == \text{argmin}_j(d_j)). \tag{19}$$

Where used, w_i is the user-provided radius (aka. width) of the place cells (defaulting to 0.2 m).

6.2.2 `GridCells`

Each grid cell is assigned a random wave direction $\theta_i \sim \mathcal{U}_{[0,2\pi]}$, gridscale $\lambda_i \sim \text{Rayleigh}(\lambda_0)$ and phase offset $\phi_i \sim \mathcal{U}_{[0,2\pi]}$. The firing rate of each grid cell is given by the thresholded sum of three cosines

$$F_i(t) = \frac{1}{3}\max\left(0, \cos\left(2\pi\frac{\mathbf{x}(t)\cdot\mathbf{e}_{\theta_i}}{\lambda_i} + \phi_i\right) + \cos\left(2\pi\frac{\mathbf{x}(t)\cdot\mathbf{e}_{\theta_i+\pi/3}}{\lambda_i} + \phi_i\right) + \cos\left(2\pi\frac{\mathbf{x}(t)\cdot\mathbf{e}_{\theta_i+2\pi/3}}{\lambda_i} + \phi_i\right)\right) \tag{20}$$

where $\mathbf{e}_\theta$ is the unit vector pointing in the direction θ.

6.2.3 `BoundaryVectorCells`

Each boundary vector cell (BVC) has a random preferred tuning distance $d_i \sim \text{Rayleigh}(d_0)$ and tuning angle $\phi_i \sim \mathcal{U}_{[0,2\pi]}$. These determine the contribution to its firing rate from a boundary segment at distance r and angle θ. The firing rate is given by integrating (computationally we use a default value of $d\theta = 2^\circ$ to numerically approximate this integral) the contributions from the nearest line-of-sight boundary segments around the full 2π field-of-view;

$$F_i(t) = K_i \int_0^{2\pi} g_i(r,\theta)d\theta, \tag{21}$$

where $g_i(r,\theta)$ is the boundary response function, modelled by

$$g_i(r,\theta) \propto \exp\left(-\frac{(d_i - r)^2}{2\sigma_{\text{rad}}^2(d_i)}\right) \cdot f_{\text{VM}}(\theta|\phi_i,\kappa). \tag{22}$$

$f_{\text{VM}}(x|\mu,\kappa)$ is the radial von Mises distribution (a generalisation of a Gaussian for periodic variables)

$$f_{\text{VM}}(x|\mu,\kappa) \propto \exp(\kappa\cos(x-\mu)). \tag{23}$$

The asymptotic equivalence between a Gaussian and a von Mises (for small angular tuning, a condition satisfied by the default parameters used here, the von Mises distribution approaches a Gaussian of variance $\sigma^2 = 1/\kappa$) means this model is effectively identical to the original model proposed by Hartley et al. [17] but with the difference being that our BVCs will not show discontinuities if they have wide angular tunings of order π.

In the model, the angular tuning width σ_{ang} is constant (from which we define $\kappa := 1/\sqrt{\sigma_{\text{ang}}}$) and the radial tuning width increases linearly with the preferred tuning distance $\sigma_{\text{red}} = d_i/\beta + \xi$ for constants β and ξ. See table S1 for the values which are chosen to match those used by Cothi and Barry [9].

K_i is a normalisation constant calculated empirically at initialisation such that each BVC has a maximum firing rate (before scaling) of 1. This factor is `Environment`-dependent and will not be recalculated if the `Environment` changes. Therefore any additional walls added to the `Environment` *after* initialisation have the potential to change the maximum firing rate.

Finally, for allocentric BVCs (default) θ is the bearing angle of a wall segment from due-North. For egocentric BVCs θ is the bearing angle of a wall segment relative to the heading direction of the `Agent`.

6.2.4 `VelocityCells`

In 2D `Environments` there are four velocity cells encoding, respectively, the positive-x, negative-x, positive-y and negative-y velocities of the `Agent`,

$$\begin{aligned} F_1(t) &= \max(0, \mathsf{v}_x(t)/\sigma) \\ F_2(t) &= \max(0, -\mathsf{v}_x(t)/\sigma) \\ F_3(t) &= \max(0, \mathsf{v}_y(t)/\sigma) \\ F_4(t) &= \max(0, -\mathsf{v}_y(t)/\sigma) \end{aligned} \tag{24}$$

where $\sigma = \sigma_{\mathsf{v}} + \mu_{\mathsf{v}}$ is the likely speed scale of the `Agent`. Hence, for typical motion and ignoring firing rate scaling from f_{max} and f_{min}, the firing rates of these cells is of order 1. In 1D only the two cells $F_1(t)$ and $F_2(t)$ are defined.

6.2.5 `SpeedCells`

A single cell encodes the scaled speed of the `Agent`

$$F(t) = \|\mathbf{v}\|/\sigma \tag{25}$$

where, same as with the `VelocityCells`, $\sigma = \sigma_{\mathsf{v}} + \mu_{\mathsf{v}}$ is the typical speed scale of the `Agent` moving under random motion. These (and head direction cells) are execptions to the rule that firing rates go from 0 to 1 and are then scaled, as there is no limit to the firing rate of a speed cell (assuming there is no limit to the speed of the `Agent`).

6.2.6 `HeadDirectionCells`

The heading direction, $\vec{\text{HD}}(t) = \mathbf{v}(t)/\|\mathbf{v}(t)\|$ of the `Agent` determines the firing rates of four head direction cells (or two in the case of 1D motion)

$$\begin{aligned} F_1(t) &= \max(0, \vec{\text{HD}}_x(t)) \\ F_2(t) &= \max(0, -\vec{\text{HD}}_x(t)) \\ F_3(t) &= \max(0, \vec{\text{HD}}_y(t)) \\ F_4(t) &= \max(0, -\vec{\text{HD}}_y(t)). \end{aligned} \tag{26}$$

6.2.7 `PhasePrecessingPlaceCells`

`PhasePrecessingPlaceCells` (a subclass of `PlaceCells`) display a phenomena known as phase precession with respect to an underlying theta oscillation; within each theta cycle the firing rate of a place cell peaks at a phase dependent on how far through the place field the `Agent` has travelled. Specifically, as the `Agent` enters the receptive field the firing rate peaks at a late phase in the cycle and as the `Agent` leaves the cell the firing rate peaks at an early phase in the cycle, hence the name phase precession. Phase precession is implemented by modulating the spatial firing rate with a phase precession factor, $F_i^\theta(t)$,

$$F_i(t) \leftarrow F_i(t) \cdot F_i^\theta(t), \tag{27}$$

which rises and falls each theta cycle, according to:

$$F_i^\theta(t) = 2\pi f_{\text{VM}}\left(\phi_\theta(t) \middle| \phi_i^*\big(\mathbf{x}(t), \dot{\mathbf{x}}(t)\big), \kappa_\theta\right). \tag{28}$$

This is a von Mises factor where $\phi_\theta(t) = 2\pi\nu_\theta t$ is the current theta phase and $\phi_i^*\big(\mathbf{x}(t), \hat{\dot{\mathbf{x}}}(t)\big)$ is the 'preferred' theta phase of a cell; a function of it's position $\mathbf{x}(t)$ and direction of motion $\hat{\dot{\mathbf{x}}}(t)$. This preferred phase is calculated by first establishing how far through a cells spatial receptive field the `Agent` has travelled along its current direction of motion;

$$d_i(\mathbf{x}(t), \hat{\dot{\mathbf{x}}}(t)) = (\mathbf{x}(t) - \mathbf{x}_i) \cdot \hat{\dot{\mathbf{x}}}(t), \tag{29}$$

and then mapping this to a uniform fraction β_θ of the range $[0, 2\pi]$;

$$\phi_i^*(t) = \pi - \beta_\theta \pi \frac{d_i(t)}{\sigma_i}. \tag{30}$$

σ_i is the width of the cell at its boundary, typically defined as $\sigma_i = w_i$, except for `gaussian` place cells where the boundary is arbitrarily drawn at two standard deviations $\sigma_i = 2w_i$.

The intuition for this formula is can be found by considering an `Agent` travelling straight through the midline of a circular 2D place field. As the `Agent` enters into the receptive field (at which point $(\mathbf{x}(t) - \mathbf{x}_i) \cdot \hat{\dot{\mathbf{x}}}(t) = -\sigma_i$) the firing rate will peak at a theta phase of $\pi + \beta\pi$. This then precesses backwards as it passes through the field until the moment it leaves ($(\mathbf{x}(t) - \mathbf{x}_i) \cdot \hat{\dot{\mathbf{x}}}(t) = \sigma_i$) when the firing rate peaks at a phase of $\pi - \beta\pi$. This generalises to arbitrary paths through 2D receptive fields. This model has been used and validated before by Jeewajee et al. [27]. κ_θ determines the spread of the von Mises, i.e. how far from the preferred phase the cell is likely to fire.

6.2.8 `FeedForwardLayer`

Each `FeedForwardLayer` has a list of input layers $\{\mathsf{L}_j\}_{j=1}^N$ which feed into it. Each input layer must be a `Neurons` subclass. For input layer j which has n_j neurons with firing rates $F_k^{\mathsf{L}_j}(t)$ for $k \in [1, n_j]$, a weight matrix is initialised by drawing weights

randomly $\mathsf{w}_{ik}^{\mathsf{L}_j} \sim \mathcal{N}(0, g/\sqrt{n_j})$ (for default weight intialisation scale $g = 1$). The firing rate of the ith `FeedForwardLayer` neuron is given by

$$r_i(t) = \sum_{j=1}^{N} \sum_{k=1}^{n_j} \mathsf{w}_{ik}^{\mathsf{L}_j} F_k^{\mathsf{L}_j}(t) \tag{31}$$

$$F_i(t) = \phi(r_i(t)) \tag{32}$$

where $\phi(x)$ is a non-linear activation function defaulting to a linear identity function of unit gain. A full list of available activations and their defining parameters can be found in the `utils.py` file; these include ReLU, sigmoid, Retanh and linear (the default) functions.

Alongside $\phi(r_i(t))$ this layer also calculates and saves $\phi'(r_i(t))$ where ϕ' is the derivative of the activation function, a necessary quantity for many classical learning rules.

6.3 Table of default parameters

This table lists the RatInABox parameters and their default values. The 'Key' column give the key in a parameters dictionary which can be passed to each class upon initialisation. Any variables not present in the parameters dictionary at initialisation will be taken as default. For example, initialising an `Environment` of size 2 m (which is *not* the default size) would be done as follows:

```
import ratinabox
from ratinabox.Environment import Environment

params = {"scale":2.0}
Env = Environment(params)
```

Parameter	Key	Description (unit)	Default	Acceptable range
		`Environment()`		
D	`dimensionality`	Dimensionality of `Environment`.	"2D"	["1D","2D"]
Boundary conditions	`boundary_conditions`	Determines behaviour of `Agent` and `PlaceCells` at the room boundaries.	"solid"	["solid", "periodic"]
Scale	`scale`	Size of the environment (m).	1.0	$\mathbb{R}^+$
Aspect	`aspect`	Aspect ratio for a 2D environment; width = sa, height = s.	1.0	$\mathbb{R}^+$
dx	`dx`	Discretisation length used for plotting rate maps (m).	0.01	$\mathbb{R}^+$
		`Agent()`		
dt	`dt`	Time discretisation step size (s).	0.01	$\mathbb{R}^+$
τ_{v}	`speed_coherence_time`	Timescale over which speed (1D or 2D) decoheres under random motion (s).	0.7	$\mathbb{R}^+$
σ_{v} (2D) μ_{v} (1D)	`speed_mean`	2D: Scale parameter and mode of the speed Rayleigh distribution. 1D: Mean of the speed Normal distribution in 1D (ms^{-1}).	0.08	2D: $\mathbb{R}^+$ 1D: $\mathbb{R}$
σ_{v}	`speed_std`	Standard deviation of speed Normal distribution in 1D (ms^{-1}).	0.08	$\mathbb{R}^+$
τ_ω	`rotational_velocity_coherence_time`	Timescale over which rotational velocity decoheres under random motion (s).	0.08	$\mathbb{R}^+$
σ_ω	`rotational_velocity_std`	std of rotational velocity Normal distribution (rad s^{-1}).	$2\pi/3$	$\mathbb{R}^+$
λ_{thig}	`thigmotaxis`	Thigmotaxis parameter.	0.5	$\mathbb{R}^+$
d_{wall}	`wall_repel_distance`	Wall range of influence (m).	0.1	$\mathbb{R}^+$
Wall repulsion	`walls_repel`	Whether walls repel the `Agent` when it is near.	`True`	[`True`,`False`]
k	`drift_to_random_strength_ratio`*	How much motion is dominated by the drift velocity relative to random motion.	1.0	$\mathbb{R}^+$
		`Neurons()`		
n	`n`	Number of neurons.	10	$\mathbb{Z}^+$
f_{max}	`max_fr`	Maximum firing rate, see source code for applicable cell types (Hz).	1.0	$\mathbb{R}^+$
f_{min}	`min_fr`	Minimum firing rate, see source code for applicable cell types (Hz).	0.0	$0 < f_{\text{min}} < f_{\text{min}}$
Name	`name`	A name which can be used to identify a `Neurons` class.	"Neurons"	Any string
		`PlaceCells()`		
Type	`description`	Place cell firing function.	"gaussian"	["gaussian", "gaussian_threshold", "diff_of_gaussians", "top_hat", "one_hot"]
w_i	`widths`	Place cell width parameter; can be specified by a single number repeated for all cells, or an array, one for each cell (m).	0.2	$\mathbb{R}^+$
Centres	`place_cell_centres`	Place cell locations. If `None`, place cells are randomly scattered.	`None`	`None` or array of positions (length n)
Wall geometry	`wall_geometry`	How place cells interact with walls.	"geodesic"	["geodesic", "line_of_sight", "euclidean"]

		`GridCells()`		
λ_0	`gridscale`	The grid scale. (m)	0.45	$\mathbb{R}^+$
Random orientation	`random_orientations`	Whether to randomise grid orientations, if `True` $\phi_i \sim \mathcal{U}_{[0,2\pi]}$.	`True`	[`True`,`False`]
Random grid-scale	`random_gridscales`	Whether to randomise grid cells, if `True` $\lambda_i \sim$ Rayleigh(λ_0).	`True`	[`True`,`False`]
		`BoundaryVectorCells()`		
Reference	`reference_frame`	Determines type of BVCs.	"`allocentric`"	["`allocentric`", "`egocentric`"]
$d\theta$	`dtheta`	Size of angular integration step ($^\circ$).	2.0	$0 < d\theta << 360$
d_0	`pref_wall_dist`	Wall distance parameter (m).	0.15	$\mathbb{R}^+$
σ_{ang}	`angle_spread_degrees`	BVC angular spread parameter ($^\circ$).	11.25	$0 < \sigma_{\text{ang}} < 360$
ξ	`xi`	BVC radial tuning offset (m).	0.08	$\mathbb{R}^+$
β	`beta`	BVC radial tuning gradient inverse.	12.0	$\mathbb{R}^+$
		`FeedForwardLayer()`		
$\{\mathsf{L}_j\}$	`input_layers`	A list of `Neurons` classes which are upstream inputs to this layer.	[] (empty list)	A list of `Neurons`
Activation function	`activation_params`	A dictionary of parameters passed to `utils.activate()` defining the activation function. Contains parameterisations of "`linear`", "`sigmoid`" and "`relu`".	{"`activation`": "`linear`"}	See `utils.activate` for full list
g	`w_init_scale`**	Scale of random weight initialisation	1.0	$\mathbb{R}^+$
		`PhasePrecessingPlaceCells()`		
ν_θ	`theta_freq`	The theta frequency (Hz).	10.0	$\mathbb{R}^+$
κ_θ	`kappa`	The phase precession breadth parameter.	1.0	$\mathbb{R}^+$
β_θ	`beta`	The phase precession fraction.	0.5	$0.0 < \beta < 1.0$

Table S1: Default values, keys and allowed ranges for RatInABox parameters. * This parameter is passed as a `kwarg` to `Agent.update()` function, not in the input dictionary. ** This parameter is passed as a `kwarg` to `FeedForwardLayer.add_input()` when an input layer is being attached, not in the input dictionary.

6.4 Tutorials and demonstrations

We provide numerous resources, listed here, to streamline the process of learning RatInABox. Next to each we describe the key features – which you may be interested in learning – covered by the resource.

- **Github readme:** Installing and importing RatInABox. Descriptions and diagrams of key features.
- **Simple script:** A minimal example of using RatInABox to generate and display data. Code duplicated below for convenience.
- **Extensive script:** A more detailed tutorial showing advanced data generation, and advanced plotting.
- **Decoding position example:** Data collection. Firing rate to position decoding. Data plotting.
- **Reinforcement Learning Example**: A bespoke `ValueNeuron` subclass is defined. Implementation of TD learning. External 'non-random' control of `Agent` velocity.
- **Path Integration Example**: Extensive use of `FeedForwardLayer` to build a deep multilayer network. Implementation of a local Hebbian learning rule.
- **List of plotting functions:** Lists and describes all available plotting functions.

In addition, scripts reproducing all figures in the GitHub readme and this paper are provided too. The code comments are nearly comprehensive and can be referenced for additional understanding where needed.

6.4.1 A simple script

See the GitHub repository for instructions on how to install RatInABox. The following is a Python script demonstrating a very basic use-case.

Import RatInABox and necessary classes. Initialise a 2D `Environment`. Initialise an `Agent` in the `Environment`. Initialise some `PlaceCells`. Simulate for 20 seconds. Print table of times, position and firing rates. Plot the trajectory and the firing rate timeseries'.

```
import ratinabox
from ratinabox.Environment import Environment
from ratinabox.Agent import Agent
from ratinabox.Neurons import PlaceCells

Env = Environment()
Ag = Agent(Env)
PCs = PlaceCells(Ag)
```

```
for i in range(int(20/Ag.dt)):
    Ag.update()
    PCs.update()

print(Ag.history["t"][:10])
print(Ag.history["pos"][:10])
print(PCs.history["firingrate"][:10])
print(PCs.history["spikes"][:10])

Ag.plot_trajectory()
PCs.plot_rate_timeseries()
```

7 Supplementary

7.1 Figure details and parameter settings

A Jupyter script replicating figures 1, 2 and 3 can be found at https://github.com/TomGeorge1234/RatInABox/blob/main/demos/paper_figures.ipynb. Unless stated below, parameters in all figures take their default values listed in table S1.

Figure 1: Panel (b): Place cells are of type `gaussian_threshold` with widths $w_i = 0.4$ m. Panel (e) $\mu_{\mathsf{v}} = 0.1$ and $\sigma_{\mathsf{v}} = 0.2$.

Figure 2: Panel (a): Curve fitting is done using `scipy.optimize.curve_fit`. Panel (d): $dt = 100$ ms.

Figure 3: Panel (a): 25 seconds of trajectory data from Sargolini et al. [3] is imported, converted into metres, mean centred and then downsampled by 30x (from 50 Hz to 1.66 Hz) before being imported into a RatInABox `Agent`. Panel (b): RatInABox for reinforcement learning experiment is described below. Panel (c): The average and standard deviation over 1000 repeats is plotted. For the motion model this is taken for motion updates of a default `Agent` in a default `Environment` (i.e. 2D with solid boundary conditions and no additional walls). For the `numpy` matrix calculations the time taken does not include the time taken to initialise the matrices.

7.2 Supplementary figures

In this section we demonstrate how RatInABox could be used in two simple experiments: neural decoding and reinforcement learning. The intention is not to present novel scientific results but rather to demonstrate the capability of RatInABox to facilitate novel scientific research in a variety of fields. Additional demos beyond these two are given in the online repository and, as with all figures in this paper, executable Jupyter scripts are provided to replicate all figures shown.

7.2.1 Supplementary figure 1: RatInABox for neural decoding

Jupyter script: https://github.com/TomGeorge1234/RatInABox/blob/main/demos/paper_figures.ipynb.

In this demonstration we study, using RatInABox, which type of spatially modulated cell type is best for decoding position.

First we generate training and testing datasets. A set of `Neurons` ($n = N_{\text{cells}} = 20$) is initialised in a 1 m square `Environment` containing a small barrier (Fig. S1b, top). A six minute trajectory is simulated using the RatInABox random motion model to produce a dataset of inputs $\{\mathbf{x}_i\}_{i=1}^{N_T}$ and targets $\{\mathbf{y}_i\}_{i=1}^{N_T}$:

$$\mathbf{x}_i = \vec{F}(\mathbf{x}(t_i)) \sim \mathcal{X} \subseteq \mathbb{R}^{N_{\text{cells}}} \tag{33}$$

$$\mathbf{y}_i = \mathbf{x}(t_i) \sim \mathcal{Y} \subseteq \mathbb{R}^2 \tag{34}$$

where $\mathbf{x}(t_i)$ is the position of the `Agent` at time t_i and $\vec{F}$ is the firing rate of the neuronal population. These data are split into training ($0 < t_i < 5$ mins, fig. S1a purple) and testing ($5 < t_i < 6$ mins, fig. S1a black) fractions. The goal of the decoder is to learn a mapping $G : \mathcal{X} \to \mathcal{Y}$, from firing rates to positions.

To do this we use Gaussian Process Regression (GPR). GPR is a form of non-parameteric regression where a prior is placed over the infinite-dimensional function space $P(G(\mathbf{x}))$ in the form of its covariance kernel $C(\mathbf{x}, \mathbf{x}')$ and mean $\mu(\mathbf{x})$ (typically zero). This defines a prior on the targets in the training set $\mathbf{Y} = (\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, ...)^{\mathsf{T}}$,

$$P(\mathbf{Y}) = \mathcal{N}(\mathbf{Y}; \mathbf{0}, \mathsf{C}), \tag{35}$$

where $\mathsf{C}_{ij} = C(\mathbf{x}_i, \mathbf{x}_j) + \sigma_\eta \delta_{ij}$ is a covariance matrix established over the data points. The second term accounts for additive noise in the data function. This can be used to make an inference on the posterior of the target for an unseen testing data point, $P(\mathbf{y}_{\text{test}} | \{\mathbf{x}_i\}^{\text{train}}, \{\mathbf{y}_i\}^{\text{train}}, \mathbf{x}_{\text{test}})$ – itself a Gaussian – the mean of which is taken as the "prediction". A more comprehensive reference/tutorial on Gaussian Process Regression is given by MacKay [28].

We use a radial basis function (aka "squared exponential") kernel with width $l = l_0\sqrt{N_{\text{cells}}}$ which scales with the expected size of the population vector ($\sim \sqrt{N_{\text{cells}}}$, we set $l_0 = \sqrt{20}$)

$$C(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2l^2}\right) \tag{36}$$

and a small amount of target noise $\sigma_\eta = 1e-10$. Note that the closest 'parameterised' analog to GPR with an RBF kernel is linear regression against Gaussian basis features of length scale l. Since the Gaussian is a non-linear function this means our regression prior is also non-linear function of firing rate (and therefore potential non-biologically plausible). We choose to optimise with the `sklearn.gaussian_process.GaussianProcessRegressor` package. Note we do not attempt to optimise the hyperparameters l_0 or σ_η which one would probably do in a more rigorous experiment. RatInABox parameters are all

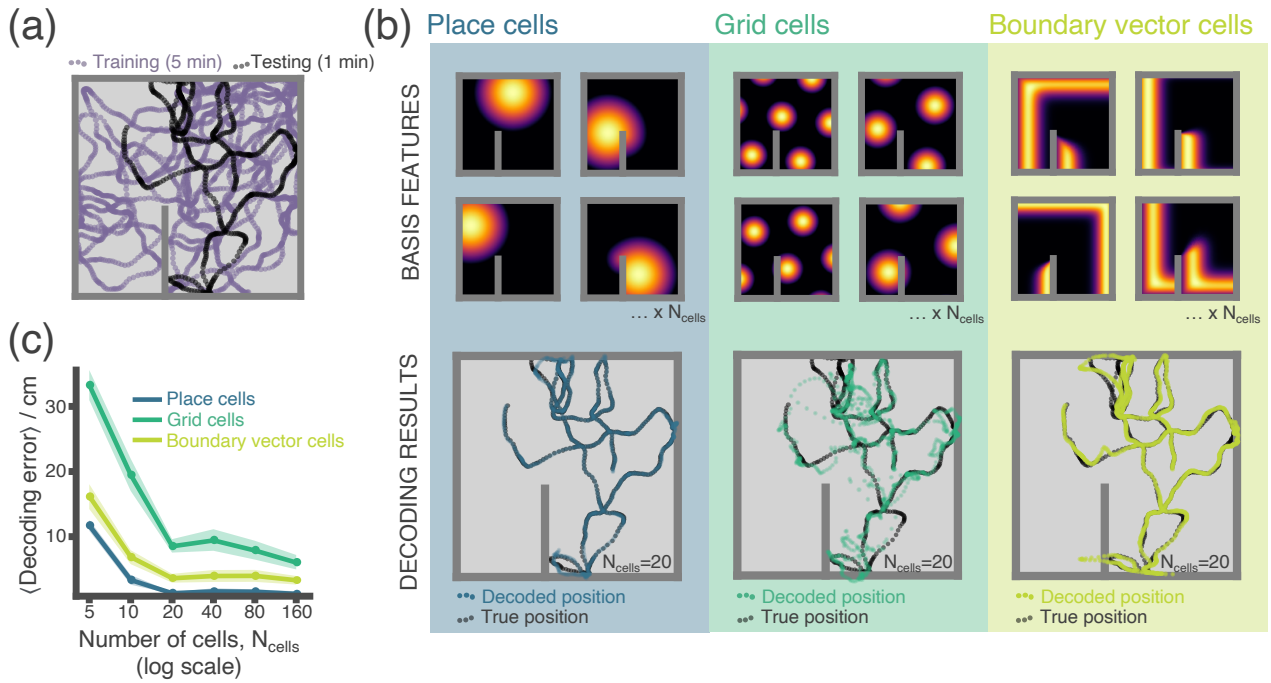

Figure S1: RatInABox used for a simple neural decoding experiment. **(a)** Training (5 min) and testing (1 min) trajectories are sampled in a 1 m square environment containing a small barrier. **(b)** The firing rates of a population of $N_{\text{cells}} = 20$ cells, taken over the training trajectory, are used to fit a Gaussian Process regressor model estimating position. This decoder is then used to decode position from firing rates on the the unseen testing dataset. Top row shows receptive field for 4 of the 20 cells, bottom row shows decoding estimate (coloured dots) against ground truth (black dots). The process is carried out independently for populations of place cells (left), grid cells (middle) and boundary vector cells (right). **(c)** Average decoding error against number of cells, note log scale. Error region shows the standard error in the mean over 15 random seeds. A jupyter script demonstrating this experiment is given in the codebase GitHub repository.

default with the exception that the place cells are of type `gaussian_threshold` and width $w_i = 0.4$ m and the timestep is set to $dt = 50$ ms.

Figure S1, panel b (lower) show the results over comparable sets of `PlaceCells`, `GridCells` and `BoundaryVectorCells`. Coloured dots show the prediction – mean of the posterior – of the GPR model "trained" on all points in the training dataset for that particular cell type. This is plotted on top of the true trajectory, shown in black. `PlaceCells` perform best achieving under 1 cm average decoding error, followed by `BoundaryVectorCells` then `GridCells` where the decoded position is visibly noisy.

Place cells outperform grid cells which outperform BVCs irrespective of how many cells are using in the basis feature set. More cells gives lower decoding error.

7.2.2 Supplementary figure 2: RatInABox for reinforcement learning

Jupyter script: https://github.com/TomGeorge1234/RatInABox/blob/main/demos/reinforcement_learning_example.ipynb.

In this example we demonstrate how RatInABox can be used in a reinforcement learning (RL) study. The goal is as follows: train an artificial `Agent` to explore a 2D `Environment` where a reward is hidden behind a wall. The `Agent` should become proficient at navigating around the wall and towards the reward from all locations within the `Environment`.

The core of our approach will rest on model-free RL where an `Agent` first learns a value function for a policy (a process known as "policy evaluation") and then uses this value function to define a new, improved policy ("policy improvement"). Iterating between these two procedures ("policy iteration") can result in convergence towards an optimal or near-optimal policy.

A core pillar of RatInABox is its continuous approach to modelling time and space. This continuity will require revising typical approaches to how the value function if defined, approximated and then learned, as well as how motion control (aka action selection, in discrete space) is performed. This is not a weakness, in fact we would argue it is one of the strengths. Once we are done, we are left with a formulation of model-free RL which bears much higher resemblance to biological navigation. Furthermore, since most of the complexities of feature encoding and motion control in continuous time and space are handled by RatInABox innately this "upgrade" comes almost for free.

Policy evaluation The value of a motion policy, π, is defined as the decaying sum (or integral in continuous time) of expected future rewards

$$\hat{\mathsf{V}}(t) \approx \mathsf{V}^{\pi}(t) = \mathbb{E}\left[\frac{1}{\tau}\int_{t}^{\infty} e^{-\frac{t'-t}{\tau}}\mathsf{R}(t')dt'\right] \tag{37}$$

where the expectation is taken over any stochasticity present in the current policy (i.e. how the `Agent` moves) and `Environment`/reward (although in this case both will be deterministic). This definition of value is temporally continuous. The key differences compared to the more common form – where value is written as a discrete sum of rewards over future timesteps – is that it is now a continuous integral over a reward density function and temporal discounting is done by exponentially decaying future reward over a time period τ. The prefactor of $1/\tau$ is an optional constant of normalisation.

In order to learn the value function we define a new `ValueNeuron` class. The `ValueNeuron`, which is a subclass of `FeedForwardLayer`, recieves feedforward input from a set of features corresponding to `PlaceCells` scattered across the `Environment` with firing rates $\{\phi_i\}_{i=1}^{N_\phi=1000}$ where $\phi_i(\mathbf{x}) = F_i(\mathbf{x})$ is the firing rate of the ith place cell at location $\mathbf{x}$. This linear approximation to the value function can be written as

$$\hat{\mathsf{V}}(\mathsf{I}(t);\mathsf{w}) = \sum_{i=1}^{N}\mathsf{w}_i\phi_i(t). \tag{38}$$

We can take the temporal derivative of equation (37) and derive a consistency equation (analogous to the Bellman equation) satisfied by this value function. This naturally gives a temporal difference-style update rule which relies on "bootstrapping" (the current estimate of the value function is used in lieu of the true value function) to optimize the weights of the value function approximation. A good reference for continuous RL is Doya [29] if readers wish to know more about deriving this learning rule.

$$\delta\mathsf{w}_{\mathsf{i}}(t) = \eta\left(\mathsf{R}(t) + \tau\frac{d\hat{\mathsf{V}}(t)}{dt} - \hat{\mathsf{V}}(t)\right)\mathsf{e}_i(t). \tag{39}$$

For now it suffices to observe that this learning rule is very similar to the temporally discrete TD-update rule. The first term in brackets represents the continuous analog of the temporal difference error (in fact, if you rediscretise using the Euler formula $\dot{\mathsf{V}}(t) = \frac{\mathsf{V}(t+dt)-\mathsf{V}(t)}{dt}$ to replace the derivative, and set $dt = 1$, you will see they are identical). The second term is the 'eligibility trace' determining to which state – or basis feature – credit for the TD error should be assigned. Using an eligibility trace is optional, and it could just be replaced with $\phi_i(t)$, however doing so aids stability of the learning. It is defined as:

$$\mathsf{e}_i(t) = \frac{1}{\tau_{\mathsf{e}}}\int_{-\infty}^{t} e^{-\frac{t-t'}{\tau_{\mathsf{e}}}}\phi_i(t')dt'. \tag{40}$$

In total the newly defined `ValueNeuron` does three things, schematically laid out in Fig. S2a:

1. It linearly summates its `PlaceCell` inputs, eqn (38).
2. It stores and updates the eligibility traces, eqn (40).
3. It implements the learning rule, eqn (39), which requires access to the reward density function $\mathsf{R}(t)$, the eligibility traces $\mathsf{e}_i(t)$, its firing rate $\hat{\mathsf{V}}(t)$ and the temporal derivative of its firing rate $\frac{d\hat{\mathsf{V}}(t)}{dt}$.

We use a temporal discount horizon of $\tau = 10$ s and an eligibility trace timescale of $\tau_{\mathsf{e}} = 5$ s. Input features are a set of $N_\phi = 1000$ `PlaceCells` of random widths uniformly sampled from 0.04 m to 0.4 m (Fig. S2b). The reward density function is taken to be the firing rate of a single `PlaceCell` positioned behind the wall of type `top_hat` and width 0.2 m (Fig. S2c). The learning rate is set to $\eta = 1e-4$.

Policy improvement Now we have a neuron capable of learning the value function under its current policy ("policy evaluation"). We want to use this to improve the policy ("policy improvement") towards an optimal one. To do this we will exploit the "drift velocity" feature (see Sec. 6.1.2). We set the drift velocity to be 3 times the mean velocity in the direction of steepest ascent of the value function.

$$\mathbf{v}_{\text{drift}}(t) = 3\sigma_{\mathsf{v}}\hat{\nabla}_{\mathbf{x}}\hat{\mathsf{V}}(\mathbf{x}(t)). \tag{41}$$

This way the `Agent` is encouraged to move towards regions of higher and higher value. Note that calculating this gradient is a local calculation and can be done on-the-fly by the `Agent`, as it locomotes. This method of value ascent is essentially a continuous analog of a similar algorithm, "greedy policy optimization", used in discrete action spaces.

Policy iteration Learning is done in batches of 8 episodes each. An episode consists of the `Agent` being reset to a random location in the `Environment` and left to explore. The episode ends when the `Agent` gets close to the reward *or* times out (60 seconds). At the start of each batch the current value function is copied and cached - this cached version is used, but not updated, to determine the drift velocity in eqn (41) for the duration of the next batch. Varying the strength of the drift bias relative to the random motion allows us to control the trade of between exploration and exploitation. Scheduling goes as follows: initially the `drift_to_random_strength_ratio` is set to $k = 0.1$ (i.e. mostly random exploration). On each successful episode which did not end in a timeout, this is increased by 10% up to a maximum of $k = 1$ (approximately equal contributions of random and drift motions).

Results Initially the input weights to the `ValueNeuron` are drawn randomly $\mathtt{w}_i \sim \mathcal{N}(0, \frac{1}{\sqrt{N_\phi}})$ and therefore the value map (and `Agent` motion) is random (Fig. S2d, left). After 10 batches (80 episodes) the `Agent` has successfully learnt a near-optimal value function showing high-value in and near to the corridor, and low values elsewhere. This allows it to rapidly navigate towards the reward, avoiding the obstructing wall, from all locations in the `Environment` (Fig. S2d, middle, colourmap) (Fig S2d, middle and 3b).

By virtue of using continuous action control in continuous space, the trajectories of the trained `Agent` look highly realistic compared to typical gridworld RL. Since `PlaceCells` in RatInABox interact adaptively with the `Environment`, when a small gap is created at the base of the obstructing wall the receptive fields of `PlaceCells` near this gap "spill" through. This causes an instantaneous update to the percieved value function and therefore policy allowing the `Agent` to immediately find a short cut to the reward with no additional training, a form of zero-shot learning (Fig. S2d, right).

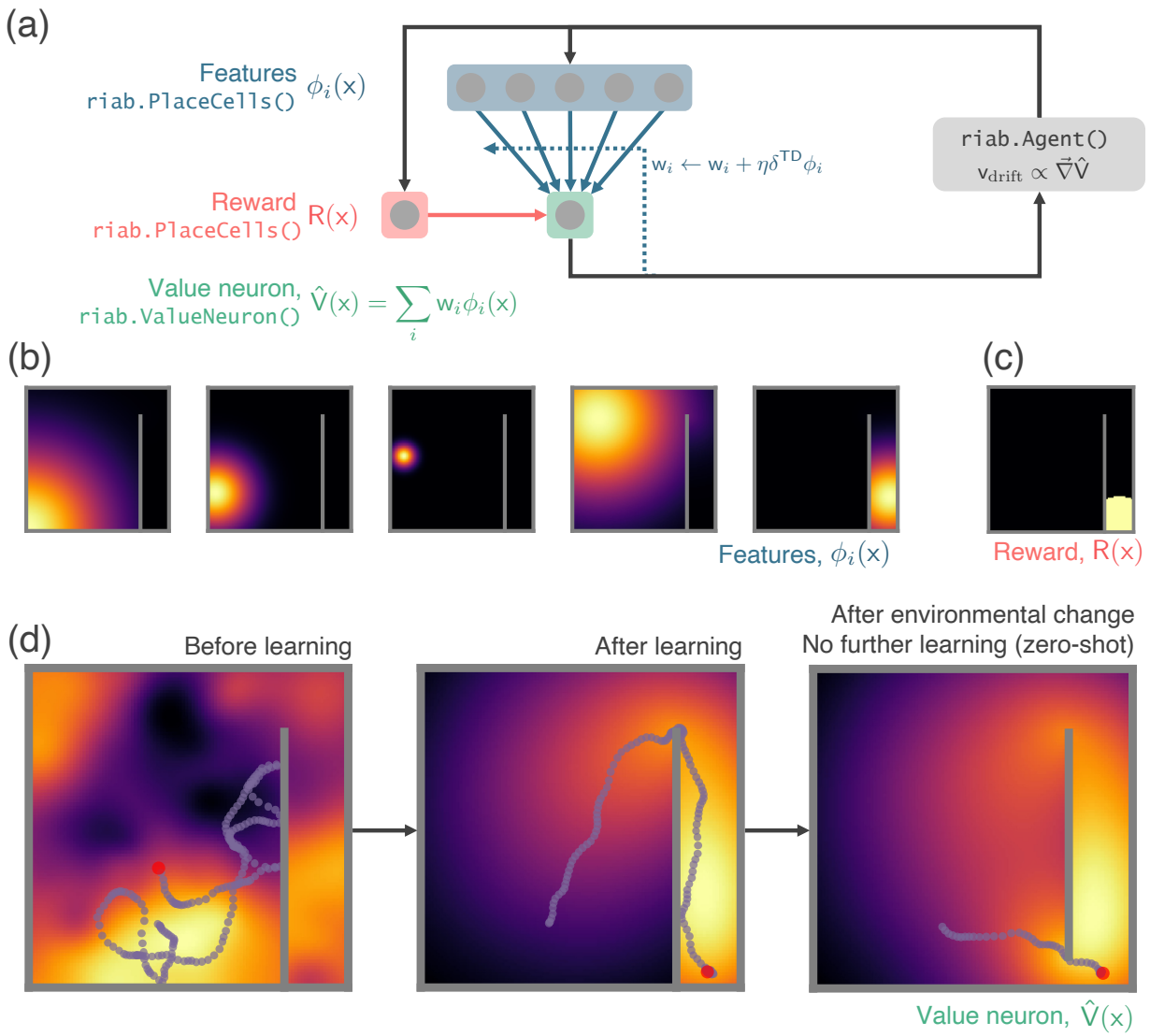

Figure S2: RatInABox used in a simple reinforcement learning project. **(a)** A schematic of the 1 layer linear network. Using a simple model-free policy iteration algorithm the `Agent`, initially moving under a random motion policy, learns to approach an optimal policy for finding a reward behind a wall. The policy iteration algorithm alternates between (left) calculating the value function using temporally continuous TD learning and (right) using this to define an improved policy by setting the drift velocity of the `Agent` to be proportional to the gradient of the value function (a roughly continuous analog for the ϵ-greedy algorithm). **(b)** 1000 `PlaceCells` act as a continuous feature basis for learning the value function. **(c)** The reward is also a (top-hat) `PlaceCell`, hidden behind the obstructing wall. **(d)** A `ValueNeuron` (a bespoke `Neurons` subclass defined for this demonstration) estimates the policy value function as a linear combination of the basis features (heatmap) and improves this using TD learning. After learning the `Agent` is able to accurately navigate around the wall towards the reward (middle). Because `PlaceCells` in RatInABox are continuous and interact adaptively with the `Environment` when a small gap is opened in the wall place fields corresponding to place cells near this gap automatically bleed through it, and therefore so does the value function. This allows the `Agent` to find a shortcut to the reward with zero additional training. A jupyter script replicating this project is given in the demos folder GitHub repository.