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

**Tom M George**[1,✉]
**Mehul Rastogi**[1] **William de Cothi**[2*]
**Claudia Clopath**[3,1*] **Kimberly Stachenfeld**[4,5*] **Caswell Barry**[2*]

[1]Sainsbury Wellcome Centre, UCL, UK
[2]Dept. of Cell and Developmental Biology, UCL, UK [3] Bioengineering Dept., Imperial College, UK
[4]Google DeepMind, London, UK [5]Columbia University, New York, NY
✉tom.george.20@ucl.ac.uk

November 8, 2023

## Abstract

Generating synthetic locomotory and electrophysiological data is a necessary yet cumbersome step 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 some 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 in which RatInABox is used for decoding position from neural data, reinforcement learning, or to set up an example circuit capable of learning path integration. We hope this tool significantly streamlines the process of theory-driven research into the brain's role in navigation.

# 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 do gridworld models scale poorly in large environments but they typically disregard aspects of motion which can be non-trivial (e.g. 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. The `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: First, 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 and 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 update 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. 3). Once generated data can be plotted using in-built plotting functions or extracted to be used in the theoretical model being constructed by the user.

## 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

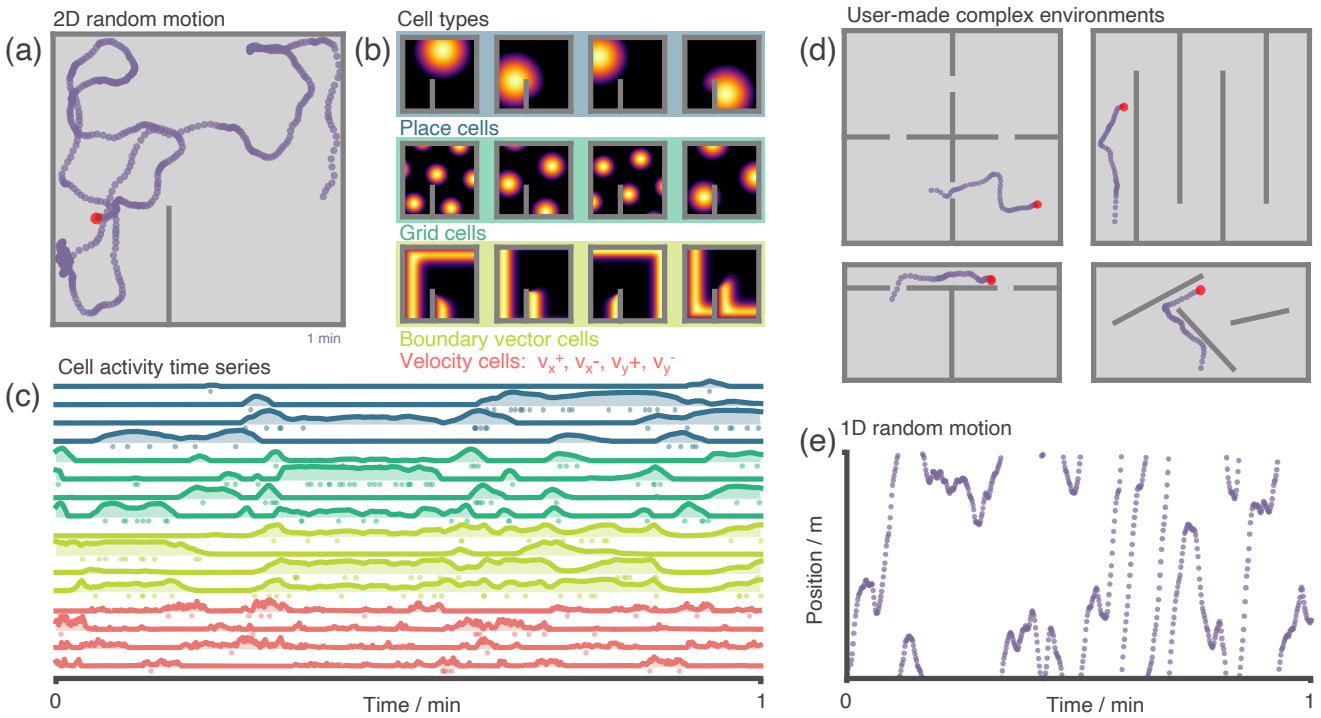


Figure 1: RatInABox is a flexible toolkit for simulating locomotion and electrophysiological data in complex continuous environments. **(a)** One minute of random motion in a 2D room with a wall. The red dot shows the current location of the `Agent`, whilst the purple dots show past locations. **(b)** 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.

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 Appendix. Additional details (including example scripts and figures) can be found in the Appendix or 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 a 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 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. With this method 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 Appendix) 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`. 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 Appendix A). 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 sigma), `'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"), `'tophat'` (fires uniformly only within a circle of specific radius) or `'onehot'` (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. The definition of 'distance' is described below.
- `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 cells fires proportional to the scalar speed of the `Agent`.
- `HeadDirectionCells`: Like `VelocityCells` but the velocity is first normalised so 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`.

**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 (Fig. 1b, notice smooth wrapping of place cells 2 and 4 around the wall). 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 of the list of 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 a 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 Figs. S1 & S2). Unlike others subclasses, the firing rate of `FeedForwardLayer` neurons is not determined directly 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 parameters of the random motion model in RatInABox are matched to the observed statistics of rodent locomotion, extracted by reanalysing data from Sargolini et al. [3]. Figure 2 shows a 5 minute trajectory (and its statistics) from the Sargolini et al. [3] dataset (Fig. 2a) alongside a 5 minute trajectory artificially generated using RatInABox (Fig. 2b).

Figure 2c shows a trajectory from an open-field rodent locomotion study (figure from Satoh et al. [22]) plotted next an equivalent heat map (Fig. 2d) for 60 minutes of rodent motion generated with RatInABox. Both the real and artificially generated datasets display biases to over-explore walls and corners. This bias is particularly pronounced for the animal when it is new to the environment, as often linked to neophobia in real rodents. The parameter `Agent.thigmotaxis` can be set to control the strength of this effect.

RatInABox can import and smoothly interpolate user-provided trajectory data. Figure 3a (left) shows a snippet of a trajectory from the Sargolini dataset[1]. For demonstrative purposes we downsample this trajectory (from 50 Hz to 1.66 Hz) and 'import' it into a new `Agent`. During exploration the `Agent` moves along the the imported trajectory which is automatically 'upsampled' using cubic spline interpolation. This creates 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 for usefully simulating cell activity. For convenience, the exact Sargolini et al. [3] datafile used in this figure is uploaded with permission to the GitHub repository and can be imported using `Agent.import_trajectory(dataset='sargolini')`.

Finally, RatInABox is computationally efficient. How long it takes a consumer-grade CPU (MacBook Pro, Apple M1 Max) to update the random motion model and firing rates of large ($N = 1000$) populations place cells, grid cells and boundary vector cells (Fig. 3c, purple) is closely matched 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 (2D environment with solid boundary conditions and no additional walls), 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 a 360° field-of-view, see Appendix B) but still outpace the matrix inverse. These results imply that, depending on the details of the model, RatInABox will likely not be a significant bottleneck relative to the downstream use case for the data it is generating.

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`,

[1] 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`

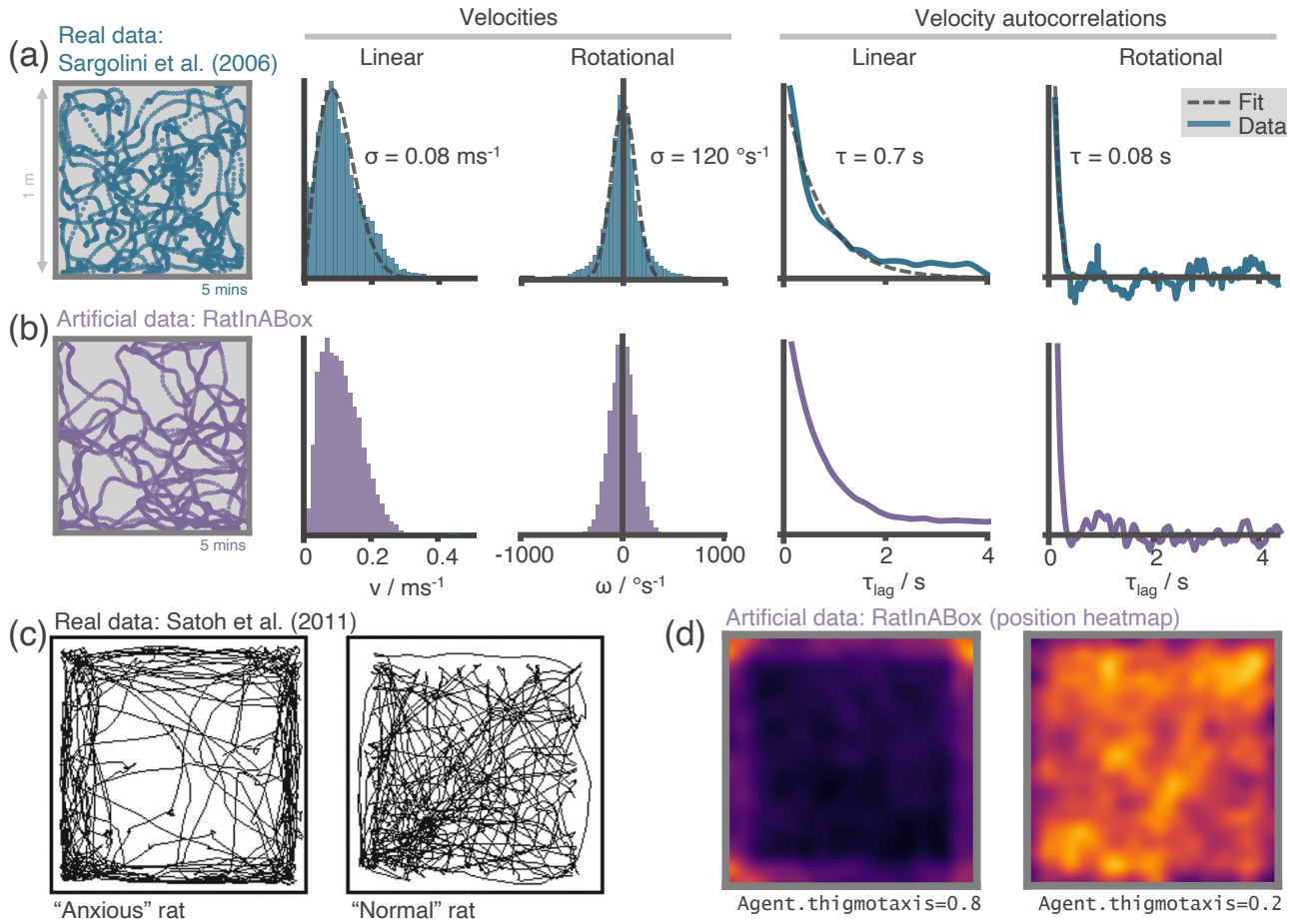


Figure 2: The RatInABox random motion model closely matches features of real rat locomotion **(a)** Trajectory data from Sargolini et al. [3]. Linear velocity (Rayleigh fit) and rotational velocity (Gaussian fit) histograms and the temporal autocorrelations (exponential fit) of their time series'. **(b)** A sampled trajectory from the RatInABox motion model with parameters matched to Sargolini data. **(c)** Data from Satoh et al. [22]. "Thigmotaxis" is the tendency of rodents to over-explore near boundaries/walls. **(d)** These smoothed position histograms (1 hour) show how RatInABox replicates the tendency of agents to over-explore walls and corners, flexibly controlled with a `'thigmotaxis'` parameter.

`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 (e.g. those shown in Fig. 3c) 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.

We demonstrate the application of this package to a simple reinforcement learning task 3b (additionally described in Fig. S1). A small network capable of reinforcement learning is constructed and trained using RatInABox. First, temporal difference learning approximates the value function over a feature basis of `PlaceCells`. A new 'improved' policy is then 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.

We also use RatInABox to model how path integration – where a position encoding is maintained in the absence of of allothetic sensory drive, and updated by integrating a velocity signal – can be learnt with local Hebbian learning rules. A population of `Neurons` receives afferent input from internal recurrent connections as well as a population of `VelocityNeurons`. A local Hebbian learning rule adjusts the synaptic weights in order to minimise the discrepancy against an allothetic target signal coming from a population of `PlaceCells`[25]. The system develops a connectivity structure matching that of a class of 1D ring attractor

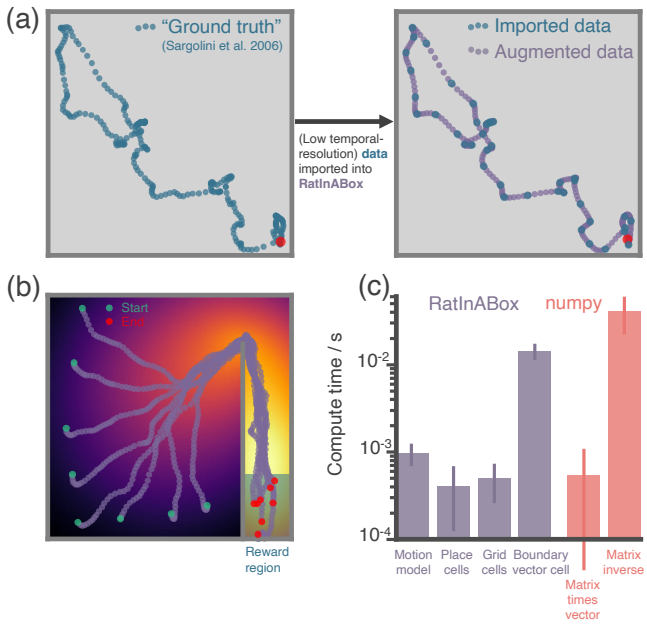


Figure 3: Data and computational efficiency of RatInABox. (a) Low temporal-resolution trajectory data imported into RatInABox is upsampled ("augmented") using cubic spline interpolation. (b) RatInABox used in a simple reinforcement learning example. A policy iteration technique converges onto an optimal 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 inherently by RatInABox. (c) Comparison of compute times (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.

networks commonly used as models of angular path integration[26]. The cells are able to maintain an accurate position estimate under random motion in the absence of the target signal, Fig. S2.

As a final demonstration, we proved a script where neural data generated from RatInABox is used to decode position. We compare the accuracy with which self-location can be decoding from different cell types (place cells, grid cells and boundary vector cells) using two decoding algorithms (linear regression and Gaussian process regression).

In the GitHub repository we provide scripts which explain these case studies in much more detail and can be run and referenced as tutorials for how to use RatInABox in more complex projects like these. Despite including these examples we stress that reinforcement learning and path integration and neural decoding are but two of 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, accelerating progress in the field.

We intend to maintain this repository and welcome community contributions, for example new cell types, 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[2].

## 5 Acknowledgements

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

[2] `tomgeorge1@btinternet.com`

## 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. Technical report, 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] Tom 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] Pantelis Vafidis, David Owald, Tiziano D'Albis, and Richard Kempter. Learning accurate path integration in a ring attractor model of the head direction system. *bioRxiv*, page 2021.03.12.435035, 2021. doi: 10.1101/2021.03.12.435035.

[26] K Zhang. Representation of spatial orientation by the intrinsic dynamics of the head-direction cell ensemble: a theory. *The Journal of Neuroscience*, 16(6):2112–2126, 1996. ISSN 0270-6474. doi: 10.1523/jneurosci.16-06-02112.1996.

# A Motion model

**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 $\theta$, $\lambda$ and $\mu$ are constants (although in principle they could vary in time). The first term in the update equation is drives decay of $\mathsf{X}_{\theta,\lambda,\mu}(t)$ towards the mean $\mu$ (typically zero). 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 $\mu$ 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 $\tau$ and the long-run standard deviation $\sigma$) using the transformations

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

to give

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

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

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 (but 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 $\sigma$ ($R_\sigma(x) \sim \text{Rayleigh}(\sigma)$). Note $\sigma$ corresponds to the mode, or $\approx 0.8$ 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 decoherence timescale and standard deviation parameters can be set by the user but otherwise default to the values fitted from real open field 2D locomotion data in Fig. 2: $\{\tau_\omega, \sigma_\omega, \tau_\mathsf{v}, \sigma_\mathsf{v}\} = \{0.08\text{ s}, 120\text{ °s}^{-1}, 0.7\text{ s}, 0.08\text{ ms}^{-1}\}$.

Finally, these velocities are integrated to give continuous and smooth, but over long time periods random, motion.

**External velocity control** It is possible to provide an external velocity signal controlling the `Agent`s motion. After the random motion update (as described above) is applied, if an external velocity $\mathbf{v}_{\text{drift}}(t)$ is passed 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. $k$ defaults to 1.

## B Cell models

In the following section we list mathematical models for each of the default provided `Neurons` subclasses. Reader 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 parameters are all set as class attributes and so can be redefined, after initialisation, if necessary. For simplicity here we describe default behaviour only.

For most of the below-listed cell classes it is also possible to set their maximum and minimum firing rates ($f_{\text{max}}$, $f_{\text{min}}$). 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 linear scaled according to

$$F(t) \leftarrow (f_{\text{max}} - f_{\text{min}})F(t) + f_{\text{min}}. \tag{11}$$

### B.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 (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 $F^{\text{Gaussian}}$):

$$F_i^{\text{Gaussian}}(t) = e^{-d_i^2/2w_i^2} \tag{12}$$

$$F_i^{\text{Gaussian threshold}}(t) = \max\left(0, \frac{e^{-d_i^2/2w_i^2} - e^{-1/2}}{1 - e^{-1/2}}\right) \tag{13}$$

$$F_i^{\text{Difference of two Gaussians}}(t; k = 1.5) = \frac{e^{-d_i^2/2w_i^2} - (1/k^2)e^{-d_i^2/2(kw_i)^2}}{1 - 1/k^2} \tag{14}$$

$$F_i^{\text{Top hat}}(t) = \begin{cases} 1 & \text{if } d_i \leq w_i \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

$$F_i^{\text{One hot}}(t) = \delta(i == \text{argmin}_j(d_j)). \tag{16}$$

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

### B.2 `GridCells`

Each grid cell is assigned a wave direction $\theta_i$, a random gridscale $\lambda_i$ drawn from a Rayleigh distribution of scale parameter $\lambda_0$ (supplied by the user, defaulting to 0.45 m) and a uniform random phase offset $\phi_i \in [0, 2\pi]$. The firing rate of each grid cell is given by the thresholded sum of three cosines

$$F_i(t) = \frac{1}{3}\max\left(0, \cos\left(2\pi\frac{\mathbf{x}(t)\cdot\mathbf{e}_{\theta_i}}{\lambda_i} + \phi_i\right) + \cos\left(2\pi\frac{\mathbf{x}(t)\cdot\mathbf{e}_{\theta_i+\pi/3}}{\lambda_i} + \phi_i\right) + \cos\left(2\pi\frac{\mathbf{x}(t)\cdot\mathbf{e}_{\theta_i+2\pi/3}}{\lambda_i} + \phi_i\right)\right) \tag{17}$$

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

### B.3 `BoundaryVectorCells`

Each BVC has a preferred tuning distance $d_i$ and tuning angle $\phi_i$. This determines the contribution to its firing rate from a boundary segment at distance $r$ and angle $\theta$. The firing rate is given by integrating (computationally we use $d\theta = 1^\circ$ to numerically approximate this integral) the contributions from the nearest line-of-sight boundary segments around the full $2\pi$ field-of-view

$$F_i(t) = K_i \int_0^{2\pi} g_i(r, \theta) d\theta, \tag{18}$$

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

$$g_i(r, \theta) \propto \exp\left(-\frac{(d_i - r)^2}{2\sigma_{\text{rad}}^2(d_i)}\right) \cdot f_{\text{VM}}(\theta|\phi_i, \kappa). \tag{19}$$

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

$$f_{\text{VM}}(x|\mu, \kappa) \propto \exp(\kappa\cos(x - \mu)). \tag{20}$$

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 $\pi$.

In the model, the angular tuning width $\sigma_{\text{ang}}$ is constant (default 11.25°, 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 $\beta$ (defaulting to 12) and $\xi$ (defaulting to 0.08 m).

$K_i$ 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) $\theta$ is the bearing angle of a wall segment from due-North. For egocentric BVCs $\theta$ is the bearing angle of a wall segment relative to the heading direction of the `Agent`.

### B.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`,

$$\begin{aligned} F_1(t) &= \max(0, \mathbf{v}_x(t)/\sigma) \\ F_2(t) &= \max(0, -\mathbf{v}_x(t)/\sigma) \\ F_3(t) &= \max(0, \mathbf{v}_y(t)/\sigma) \\ F_4(t) &= \max(0, -\mathbf{v}_y(t)/\sigma) \end{aligned} \tag{21}$$

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 $f_{\text{max}}$ and $f_{\text{min}}$, 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.

### B.5 `SpeedCells`

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

$$F(t) = \|\mathbf{v}\|/\sigma \tag{22}$$

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.

### B.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)

$$\begin{aligned} F_1(t) &= \max(0, \vec{\text{HD}}_x(t)) \\ F_2(t) &= \max(0, -\vec{\text{HD}}_x(t)) \\ F_3(t) &= \max(0, \vec{\text{HD}}_y(t)) \\ F_4(t) &= \max(0, -\vec{\text{HD}}_y(t)). \end{aligned} \tag{23}$$

### B.7 `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 $i^{\text{th}}$ `FeedForwardLayer` neuron is given by

$$r_i(t) = \sum_{j=1}^{N} \sum_{k=1}^{n_j} \mathsf{w}_{ik}^{\mathsf{L}_j} F_k^{\mathsf{L}_j}(t) \tag{24}$$

$$F_i(t) = \phi(r_i(t)) \tag{25}$$

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 and sigmoid functions).

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

## C 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**: Bespoke `ValueNeuron` subclass 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.

### C.1 A simple script

See the GitHub repository for instructions on how to install RatInABox. The following is a Python script demonstrating a 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()
```

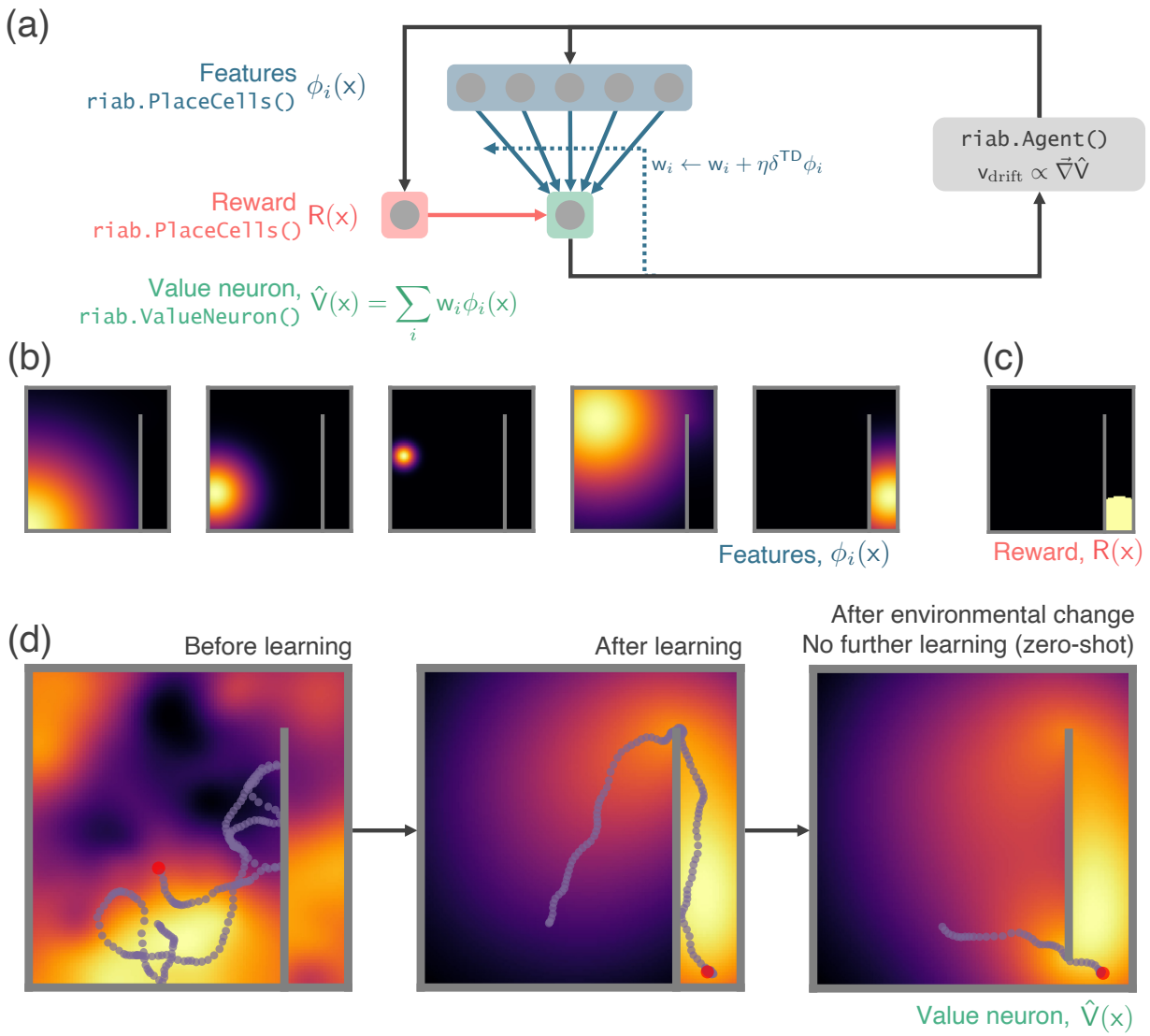


Figure S1: RatInABox used in a simple reinforcement learning project. **a** A schematic of the 1 layer linear network. Using a simple model-free policy iteration algorithm the `Agent`, initially moving under a random motion policy, learns to approach an optimal policy for finding a reward behind a wall. 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 $\epsilon$-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 detailed (and mathematical) description of this project is given in the GitHub repository.

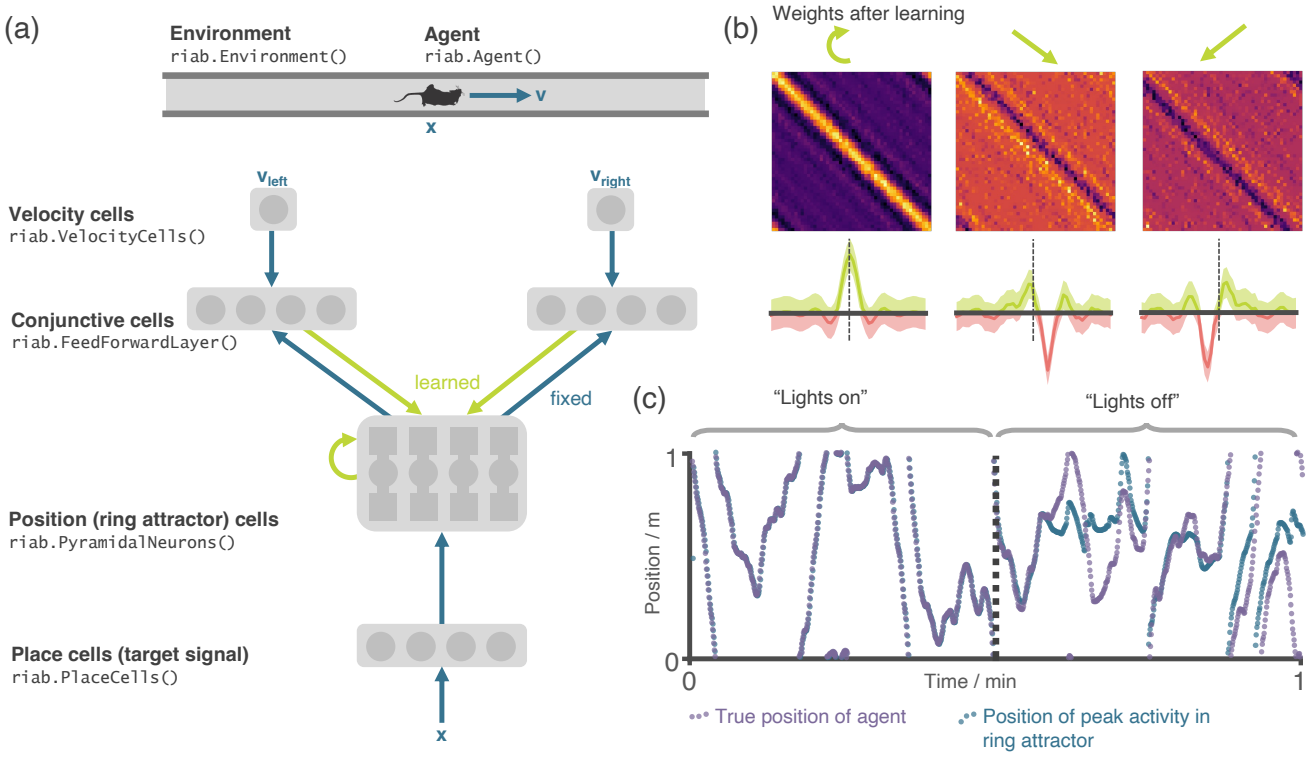


Figure S2: RatInABox used to study a bioligically plausible mechanism for learning path integration. **a** The network structure which learns, using a local Hebbian learning rule, to perform path integration. All layers are RatInAbox `Neurons` subclasses. The `Agent` moves under an unbiased 1D random motion policy. Green weights are plastic whilst blue weight fixed to sensible initial values. **b** The system develops a connectivity structure matching that of a class of 1D ring attractor networks commonly used as models of path integration. Recurrent weights show short-range excitation and long-range inhibition (like a continuous bump attractor network) whilst conjuctive velocty inputs develop an asymmetric connectivity structure which can "push" the activity bump around the attractor network proportional with the speed on the `Agent` thus "integrating" position. Bottom row shows averages and standard deviations over aligned rows. **c** After learning, an experiment is performed in "lights on" (where the allothetic target signals are still present) and "lights off" where the target signal is removed entirely and only inputs from the path integration infrastructure remain. The `Agent` is able to maintain a position estimate, accumulating only small errors, for over 30 seconds after the target signal is removed. A detailed (and mathematical) description of this project is given in the GitHub repository.