Bug Algorithms using Neural Networks

Andreas Stöckel

***Abstract*— In robotic and biological agents alike, path planning is often restricted to local, sensor-based information. In cases where self-localisation and mapping techniques are infeasible, a resource-efficient alternative relying on few sensors and minimal memory is the family of so called *bug algorithms*. Despite their name, bug algorithms are not biologically motivated and relatively abstract. Here, I present a prototypical implementation of the *Bug 0* and *Bug 2* algorithms as a biologically constrained spiking neural network built using the Neural Engineering Framework (NEF). Simulations show that the resulting neurally controlled agents successfully navigate simple environments. Failed navigation attempts in complex environments illustrate challenges arising when implementing relatively abstract algorithms as neural networks.**

I. Introduction

Navigation—travelling from a current location in space to a possibly hidden goal—is fundamental to both animal and mobile robot behaviour. Animals rely on navigation to forage food sources, while analogously mobile robots may need to return to a charging station whenever their battery runs low. Especially in small animals such as insects path planning is restricted to local, sensor-based information and supposedly little memory. Likewise, autonomous robots may be restricted to local information whenever it is desirable to design energy- and cost-efficient systems with few computational resources, rendering accumulation of global information over time, e.g. simultaneous localization and mapping (SLAM, [1]), infeasible.

Bug algorithms are a class of sensor-based navigation algorithms proposed by Lumelsky and Stepanov in 1987 [2]. Extending basic obstacle-following and goal-directed behaviours, these algorithms guarantee to guide an agent along a path from start to goal exactly if such a path exists. However, being of an abstract nature, it is non-trivial to implement these algorithms on a physical robot platform, and even more so when using a biologically constrained computing substrate. While the original authors were probably inspired by biology, they make no claim whatsoever of biological plausibility. Still, the simplicity of the algorithms poses the question whether they actually could be a viable strategy for animals or, in a broader sense, agents performing biologically constrained computations using spiking neural networks.

In the context of path planning with incomplete information, bug algorithms were originally proposed to show that provable strategies exist, i.e. one can prove that the strategy will actually find a solution exactly if a solution exists. Only heuristics—algorithms without this property—had been previously proposed for this problem. Improved versions of the algorithms have been published, such as *VisBug* [3], *DistBug* [4], and *TangentBug* [5] which incorporate range sensor information in order to decide when to switch between behaviours. A comprehensive overview of bug algorithms is given in [6]. For the sake of simplicity I focus on the original bug algorithms, yet techniques similar to those presented here should be applicable to more advanced models as well.

Building biologically constrained models of neural systems and cognitive tasks is a central aim of computational neuroscience and can be traced back to models of spiking behaviour of single neurons presented in the 1950s [7]. The central challenge is to combine individual model neurons in such a way that the network fulfills a desired task, that is to build *functional* networks. In recent years multiple modelling frameworks have been proposed, aiming at simplifying the construction of functional neural networks, while at the same time accounting for empirical neurobiological data. According to [8], approaches include a technique by Denève et al. for modelling dynamical systems as neural networks [9], detailed reproduction of biological networks as strived for in the Human Brian Project [10], and the Neural Engineering Framework (NEF), which constructs neural networks from mathematical descriptions [11].

The ability to construct functional spiking neural networks is not only interesting from the perspective of neuroscience alone. For engineers, providing spiking neural network implementations of mobile robot control algorithms—a field also called *neurorobotics*—enables their execution on analogue neuromorphic hardware systems, which promise magnitudes improved power-efficiency compared to traditional digital computers [12].

The contribution of this paper is to provide a prototypical implementation of two bug algorithms as spiking neural network using the NEF. In particular, I explore an implementation of a network which provides discrete state transitions, which is an important building block in the bug algorithm implementation. For evaluation purposes, I employ a custom robot simulator to compare the behaviour of the neural implementation to a traditionally implemented reference.

The remainder of this paper is structured as follows. In Section II I give an overview of both bug algorithms and the NEF, followed by a detailed description of the implementation in Section III. In Section IV I briefly present a set of experiments and their results, followed by a short discussion. Section V concludes this work by summarizing the main results and discussing future research directions.

The author is with the Cheriton School of Computer Science and the Centre for Theoretical Neuroscience, University of Waterloo, Waterloo ON, N2L 3G1 Canada (`astoecke@uwaterloo.ca`)

II. Background Material

Here, I review the two bug algorithms implemented in this paper and summarize the basic ideas underlying the NEF. Note that the description of Bug algorithms is mainly adapted from [2], [6], [13], whereas detailed treatments of the NEF and its applications can be found in [11], [14]–[16].

A. Bug Algorithms

Bug algorithms solve the task of navigating from a start location $\boldsymbol{p}_{\mathrm{start}}$ to a goal location $\boldsymbol{p}_{\mathrm{goal}}$ in a two-dimensional environment featuring polygonal obstacles. It is assumed that the robot is holonomic, point-shaped (i.e. it has no physical extent), and senses its current location $\boldsymbol{p}$, as well as the goal location—or equivalently, and assuming the robot has an orientation, the robot knows the direct-line distance, as well as the absolute (with respect to the world reference frame) and relative (with respect to its own orientation) angle to the goal. Furthermore, the algorithms assume the robot has a contact sensor detecting obstacle collisions. The robot supports two basic behaviours, namely driving in a straight line towards the goal and closely following an obstacle outline. As a major limitation, its memory system is limited and capable of storing few distances and angles only.

The basic algorithm in the bug family is Bug 0 [13]:

```
1: while not at goal do
2:     move towards the goal
3:     if hit an obstacle then
4:         while not able to move towards the goal do
5:             follow obstacle boundary counter clock wise
6:         end while
7:     end if
8: end while
```

Note that this algorithm does not guarantee finding a path from start to goal. Some obstacle topologies will send the agent around in circles, and since it has no memory, the robot has no chance of ever exiting one of these cycles.

Two of the originally proposed solutions to this problem are the Bug 1 and Bug 2 algorithms. Bug 1 circumnavigates the entire obstacle outline, tracks the point with the minimum distance to the goal, and continues to drive towards the goal at this point. Bug 2 avoids circumnavigating the entire obstacle by driving towards the goal whenever the robot crosses the start-goal line and the current distance to the goal is smaller than the distance to the goal upon hitting the obstacle, or, expressed in terms of the absolute direction sensor:

```
 1: $\alpha \leftarrow$ goal direction
 2: while not at goal do
 3:     move towards the goal
 4:     if hit an obstacle then
 5:         $d_{\mathrm{hit}} \leftarrow$ distance to goal
 6:         while distance to goal $\geq d_{\mathrm{hit}}$ or
 7:                 goal direction $\neq \alpha$ do
 8:             follow obstacle boundary
 9:         end while
10:     end if
11: end while
```

Note that Bug 2 does not always perform better than Bug 1 in terms of the total path length, but in general the trajectory produced by Bug 2 seems more sensible to the human observer. In the following I focus on a neural implementation of the Bug 0 and Bug 2 algorithms. Due to time constraints, I will not implement Bug 1—however, since both Bug 1 and 2 make use of the same basic sensor, state transition, and memory system, the techniques for the realization of Bug 2 could be transferred to an implementation of Bug 1 .

B. The Neural Engineering Framework (NEF)

The NEF [11] is a systematic approach to modelling spiking neural networks from a mathematical description. To this end, the NEF relies on three key principles: *(i)* ensembles of neurons represent vectors $\boldsymbol{x} \in \mathbb{R}^d$, *(ii)* connections between ensembles compute functions $f(\boldsymbol{x})$, and *(iii)* recurrent connections realize time-dynamics $\dot{\boldsymbol{x}} = f(\boldsymbol{x}) + g(\boldsymbol{y})$. While the NEF itself imposes relatively few restrictions on the neuron model, I model each neuron in the network as *leaky integrate and fire* (LIF) neuron

$$\begin{aligned}\tau_{\mathrm{RC}} \cdot \dot{u}(t) &= J - u(t)\,,\\ u(t') &\leftarrow 0 \,\forall t' \in [t, t + \tau_{\mathrm{ref}}] \text{ if } u(t) \geq 1\,,\end{aligned}$$

where τ_{RC} is the membrane time constant, u is the membrane potential, and J is the input current injected into the membrane. Once the u reaches a certain threshold potential, an output spike is issued and u is clamped to zero for the duration of the so called refractory period τ_{ref}.

Assume an ensemble of n neurons receives input from m pre-synaptic neurons. Then, the input current vector $\boldsymbol{J} \in \mathbb{R}^n$ is defined over time as

$$\boldsymbol{J}(t) = \mathrm{diag}(\boldsymbol{\alpha}) W \boldsymbol{a}(t) + \boldsymbol{J}_{\mathrm{bias}} \tag{1}$$

where $\boldsymbol{\alpha}, \boldsymbol{J}_{\mathrm{bias}} \in \mathbb{R}^n$ are vectors of scalar gain factors and offsets, $W \in \mathbb{R}^{m \times n}$ is a weight matrix wherein $(W)_{ij}$ describes the connection weight between the ith pre-synaptic and the jth post-synaptic neuron, and $\boldsymbol{a}(t) \in \mathbb{R}^m$ is a function describing the activity of the pre-synaptic population

$$(\boldsymbol{a}(t))_i = \int_{-\infty}^{\infty} \sum_k \delta(t - t' - t_{ik}^{\mathrm{spike}}) \cdot H(t') \,\mathrm{d}t'\,.$$

Here, $\delta(t)$ is a Dirac-pulse, H is a convolution kernel describing synaptic dynamics, and t_{ik}^{spike} is the time of the kth output spike of the ith presynaptic neuron.

The core idea of the NEF is to factorise the weights W into of encoding and decoding vectors, turning Equation (1) into

$$\boldsymbol{J}(t) = \mathrm{diag}(\boldsymbol{\alpha}) E D \boldsymbol{a}(t) + \boldsymbol{J}_{\mathrm{bias}}\,.$$

The d-dimensional encoding vectors in $E \in \mathbb{R}^{n \times d}$ are randomly selected for each of the n neurons in an ensemble. Given a low-dimensional vector $\boldsymbol{x} \in \mathbb{R}^d$, the operation $E\boldsymbol{x}$ projects $\boldsymbol{x}$ into an $n \gg d$ dimensional neural activity space, while the decoding matrix $D \in \mathbb{R}^{d \times m}$ decodes d-dimensional representations from the m-dimensional pre-synaptic activity. Measuring steady-state pre-synaptic neural activities for a range of represented values $\boldsymbol{x}$, arranging them in a matrix A, and placing the desired decoded values $f(\boldsymbol{x})$ in a matrix Y, we can find the decoding matrix D^f calculating f using simple least-squares regression

$$Y \overset{!}{=} D^f A \Rightarrow D^f \approx Y A^T (A A^T + \sigma I)^{-1} \,,$$

where σ is a regularization factor. This method is similar to the kernel method [17]. Low-dimensional functions g are randomly and non-linearly (with the neurons being the non-linearity) projected into a high-dimensional function space (defined by neural activity $\boldsymbol{a}$ and the encoding matrix E), which in turn is linearly—and approximately—projected onto our desired function f by D. Empirical analysis shows that this approximation works best for functions which can be represented with low-order Legendre polynomials. The approximation error decreases as a function in $\mathcal{O}(1/\sqrt{n})$.

Recurrent connections on a single neuron ensemble allow to implement arbitrary dynamics $\dot{\boldsymbol{x}} = f(\boldsymbol{x}) + g(\boldsymbol{y})$, where $\boldsymbol{x}$ is the value represented by the neuron ensemble and $\boldsymbol{y}$ is an input value represented by another ensemble. Assuming H is a first-order exponential low-pass filter with time-constant τ_{syn}, one can show that the functions f' and g' that must be decoded to realize this system are

$$f'(\boldsymbol{x}) = \tau_{\mathrm{syn}} \, f(\boldsymbol{x}) + \boldsymbol{x} \,, \qquad g'(\boldsymbol{y}) = \tau_{\mathrm{syn}} \, g(\boldsymbol{y}) \,.$$

To summarize, given the NEF, building functional spiking neural networks can be distilled into defining vectorial quantities, functions transforming these quantities, and dynamics evolving over time. Given the number of neurons representing each quantity, as well as neuron parameters E, $\boldsymbol{\alpha}$, and $\boldsymbol{J}_{\mathrm{bias}}$, we can compute ensemble activities over $\boldsymbol{x}$ and optimize for decoders D, resulting in connection weights $W = ED$. Conveniently, this process is implemented in a Python package called Nengo[1] [18], which is used in the following for both neural network description and simulation.

III. Methods

This section focuses on the bug algorithm implementation. I first introduce the robot simulator and a mathematical description of the basic robot behaviours, leading to the network topology implementing the Bug 0 behaviour. I then use a neural state transition sub-network to extend this basic variant to the Bug 2 algorithm

A. Robot Simulator

The robot simulator has been specifically implemented for this project. In contrast to the assumptions in the algorithm description, the virtual robot is disc-shaped with radius $r = 0.15\,\mathrm{cm}$ and has a non-holonomic drive controlled by a steering vector $\boldsymbol{v} = (v_x, v_y)$ encoding both the desired change in orientation $\dot{\alpha} \propto \operatorname{atan2}(x_y, v_x)$, as well as the velocity $\|\dot{\boldsymbol{p}}\| = \|\boldsymbol{v}\|$. In addition to the sensors listed in the theoretical description of the bug algorithms, the robot is equipped with a low-dimensional omnidirectional range sensor for obstacle

[1]Software available at `https://www.nengo.ai/`

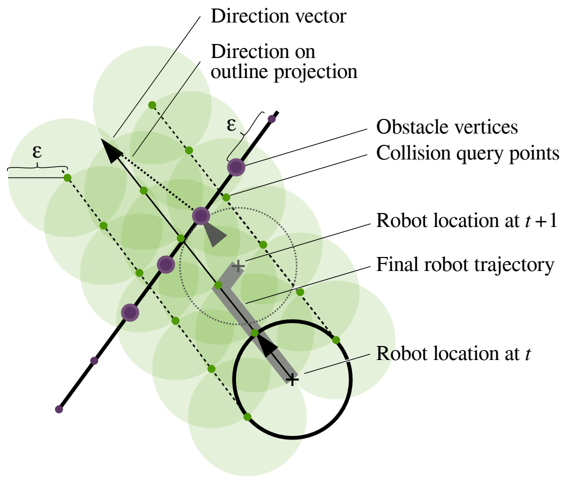

Fig. 1. Illustration of the robot movement model and collision queries. Obstacle points (plum) are queried (green) along three rays parallel to the direction vector. On collision, the direction vector is projected onto the obstacle outline orientation and movement continues in that direction.

outline following. Absolute and relative goal directions are represented as two-dimensional unit vectors.

The environment is read from a low-resolution ASCII bitmap. Obstacle outlines are vectorized into polygons, simplified, and subdivided with a maximum distance ϵ between vertices. Vertices are stored in a k-d tree [19] for logarithmic lookup time in collision queries. Ray-polygon intersections are performed by subdividing the ray into vertices with ϵ distance, querying obstacle vertices within an ϵ-ball centred at each ray vertex, and calculating ray-line intersection between the ray and the polygon segments associated to each obstacle vertex. Robot motion is simulated in discrete $10\,\mathrm{ms}$ timesteps.

As sketched in Figure 1, collision detection is performed by sending three parallel rays along the desired robot path and performing corresponding ray-polygon intersection queries. Upon collision, the robot centre is set to the closest possible distance d to the obstacle outline, as calculated by

$$d = r \cdot \left(1 - \frac{|\langle \boldsymbol{d}, \boldsymbol{o} \rangle|}{\|\boldsymbol{d}\| \cdot \|\boldsymbol{o}\|} \right)^{-\frac{1}{2}} \,,$$

where $\boldsymbol{d}$ and $\boldsymbol{o}$ are the current movement direction and obstacle orientation. The robot then continues along the obstacle, by projecting the direction onto the outline

$$\boldsymbol{d}' = \frac{\langle \boldsymbol{d}, \boldsymbol{o} \rangle \cdot \boldsymbol{o}}{\|\boldsymbol{o}\|} \,.$$

Physically this corresponds to having no friction between the robot and its environment, and allows sliding along obstacles.

In the simulator software, individual behavioural patterns are implemented as part of callback function invoked at each timestep, which transforms current sensor data and into a motor command $\boldsymbol{v}$. In the case of neural network controlled agents, the callback function feeds sensor data into the network, advances the Nengo simulation, and decodes the motor output from the network.

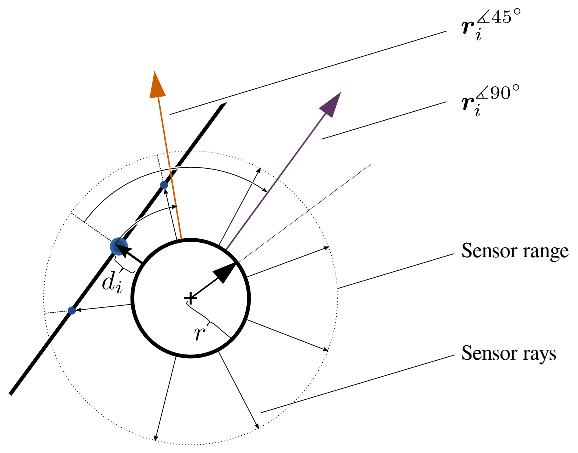

Fig. 2. Illustration of the range sensor used for wall following. Wall following is achieved by calculating a weighted sum of rotated sensor rays (depicted: sensor ray $\boldsymbol{r}_i$ with the shortest distance d_i rotated by $45°$ and $90°$). The depicted set of non-symmetric sensor orientations was chosen to break symmetries causing the robot to get stuck at corners.

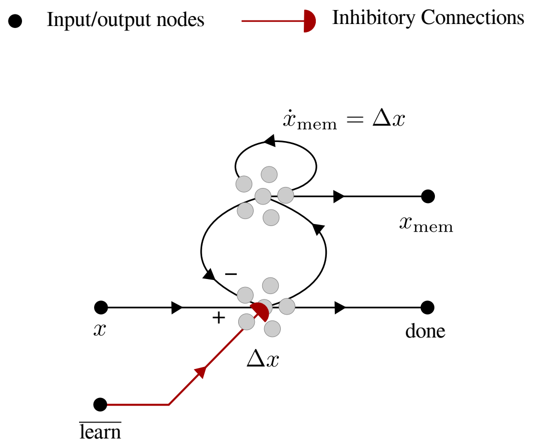

Fig. 3. Single memory sub-network. An integrator ensemble integrates the difference between the input signal x and the currently stored value x_{mem}. For small difference values Δx an output indicates that the memory is *done* learning. The input $\overline{\text{learn}}$ allows to stop the learning process and to maintain the currently stored value. See next page for a description.

For evaluation purposes the simulator calculates the ground-truth shortest path (assuming a point robot) by constructing a visibility graph of the obstacle polygon corner points and start and goal points as outlined in [13]. The set of shortest paths between all start and goal points defined in the environment is obtained by repeated execution of Dijkstra's algorithm [20].

B. *Basic Robot Behaviours*

As discussed above, the bug algorithms rely on the robot being capable of two basic behaviours: moving towards the goal in a straight line and closely following an obstacle outline. In the simulation framework presented above, the first behaviour can be trivially implemented by setting the motor command $\boldsymbol{v}$ to a vector proportional to the agent's relative goal direction vector. Note that in contrast to the algorithmic assumptions, the robot's trajectory will describe a curve instead of a straight line in most circumstances.

As outlined in [21], the omnidirectional range sensor present on the virtual robot can be used to implement a wall following behaviour by moving perpendicular to the sensor ray with the shortest distance to the wall. Since the function $\arg\min_i d_i$ is only representable with high-order terms of the Legendre function basis, this algorithm is surprisingly hard to implement with few neurons in the NEF. To this end, I simplify the algorithm by calculating a weighted sum of the rotated sensor direction vectors, where rays corresponding to a smaller distance have quadratically greater weight. Furthermore, I rotate the ray vectors by $45°$ instead of $90°$, which forces the robot to move towards the obstacle and to slide along the wall upon collision. The final motor command $\boldsymbol{v}$ for the obstacle-following behaviour is given as

$$\boldsymbol{v} = \sum_{i=1}^{n} \left(1 - \frac{d_i - r}{1 - r}\right)^2 \cdot \boldsymbol{r}_i^{\measuredangle 45°} \,,$$

where n is the number of sensor rays, d_i is the measured distance for the ith ray, r is the robot radius, the one in the denominator corresponds to the maximum sensor range, and $\boldsymbol{r}_i^{\measuredangle 45°}$ is the direction vector of the ith ray rotated by $45°$ (see also Figure 2). Importantly, due to finite rotation speed of the simulated robot, the robot will not always stay in contact with the obstacle outline. Especially when passing sharp corners, the presented strategy will cause the robot to overshoot before it moves back to the wall. Additionally, note that this algorithm implicitly causes the agent to move along the obstacle outline with a fixed rotational direction, as required for the Bug 0 strategy.

C. *Implementation of Bug 0*

Given the above primitives, a central challenge concerning the implementation of Bug 0 on a physical (albeit simulated) robot is the "able to move towards goal" condition—after all, the robot does not possess a sensor which provides this information, and the theoretical algorithm does not specify how it can be deduced from the present sensor input. As an approximation, I periodically interrupt the "follow obstacle" behaviour, causing the robot to try moving towards the goal. In case this is not possible, it will still hit the obstacle outline and in consequence return to the wall following behaviour.

Figure 4 sketches the network architecture used in the neural implementation of the Bug 0 algorithm. The network executes the two basic behaviours—following the obstacle outline and driving in the relative compass direction towards the goal—at the same time. The network uses a *Basal Ganglia* sub-network—a circuit modelled after the Basal Ganglia in vertebrates [22]—to select one of the behaviours based on a low-passed filtered version of the contact sensor signal. This signal is periodically suppressed by a $1\,\mathrm{Hz}$ oscillator, forcing the robot to test whether it can move towards the goal. The actual behaviour selection is performed by inhibiting

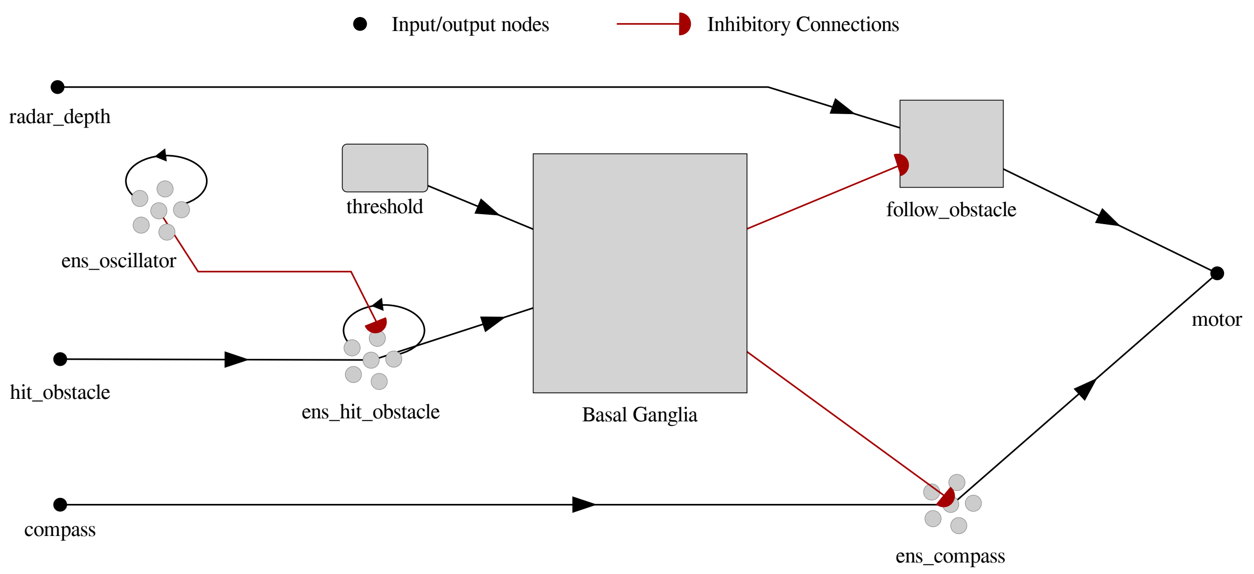

Fig. 4. Sketch of the neural network implementation of Bug 0. See text on previous page for a description.

inactive portions of the network. An inhibitory connection is established by selecting connection weights $(W)_{ij}$ in such a way that negative currents are injected into the post-synaptic neuron ensemble, suppressing all neural activity. Due to the linear decoders in the NEF, inactive neuron populations represent the value zero.

D. *Implementation of Bug 2*

From the pseudo-code description of bug algorithms in Section II we can deduce four possible states: *(i)* memorizing the absolute orientation towards the goal at the start location, *(ii)* moving towards the goal, *(iii)* storing the distance to the goal upon hitting an obstacle, and *(iv)* following the obstacle outline. As with the Bug 0 implementation described above, all these behaviours can be easily implemented and executed in parallel as a neural network. In comparison to Bug 0, the only new behaviour is storing values in memory. Here, I implement a simple working memory which stores values in neural activity, as opposed to long-term memory which adapts synaptic weights [16]. The corresponding network is depicted in Figure 3. It allows to interrupt the learning process—i.e. to retain a value in memory—and provides a signal indicating whether learning is done—i.e. the value stored in memory is close to the target value.

From an implementation point of view, a complication in contrast to Bug 0 is the fact that Bug 2 needs to take its current state into account when switching between behaviours. The first behaviour that needs to be executed is *(i)* and following behaviours must cycle through states *(ii)*-*(iv)*. The present implementation solves this using a state transition network. Each behaviour provides a signal indicating whenever it is "done" (e.g. the "move towards goal behaviour" hits a wall, the "follow obstacle outline" behaviour abortion condition is met, one of the two memories has converged to the target value), and the state transition network disinhibits the behaviour corresponding to the next state in a transition table.

The state transition network is depicted in Figure 5. In technical terms, it resembles an edge-triggered flip-flop. Whenever the input signal (the sum of all "done" signals described above) has a rising edge, the network transitions to the next state. While the input has a small value, the network transfers the current state to an "old state" ensemble. Whenever the input is high, the connection form "current state" to "old state" is inhibited and the next state is decoded from "old state" and transferred to the "current state". States are represented as two-dimensional points on the unit-circle. The "old state" and "current state" ensembles implement attractor dynamics [23], [24] to cleanly converge to one of the possible states. The four-dimensional output value is cleaned up through a low-pass filter adding additional delay, a hysteresis network preventing unwanted transient state transitions, and finally inverted, facilitating inhibition of inactive behaviours.

IV. Experiments and Results

I am comparing the neural bug algorithm implementation to a pure Python reference which—on a conceptual level—uses the same basic behaviours. In terms of resource utilization, the neural Bug 0 implementation uses a total of 2050 LIF neurons, whereas Bug 2 requires 4010 neurons.[2]

Figure 6 shows the performance of Bug 0 and Bug 2 for an obstacle topology in which Bug 0 should fail from a theoretical perspective. Indeed, both reference and neural implementation exhibit a highly similar behaviour and do not reach the goal. Instead, and as elaborated in Section II, they continue to repeat the same trajectory over and over again. While the neural and reference Bug 2 implementations clearly do not find the shortest path from start to goal, both reach the goal and as with Bug 0, their trajectories are highly

[2]For reference, Wikipedia lists between 250 thousand and one million (significantly more complex) neurons for the entire insect brain, see *List of animals by number of neurons* (2017, July 26).

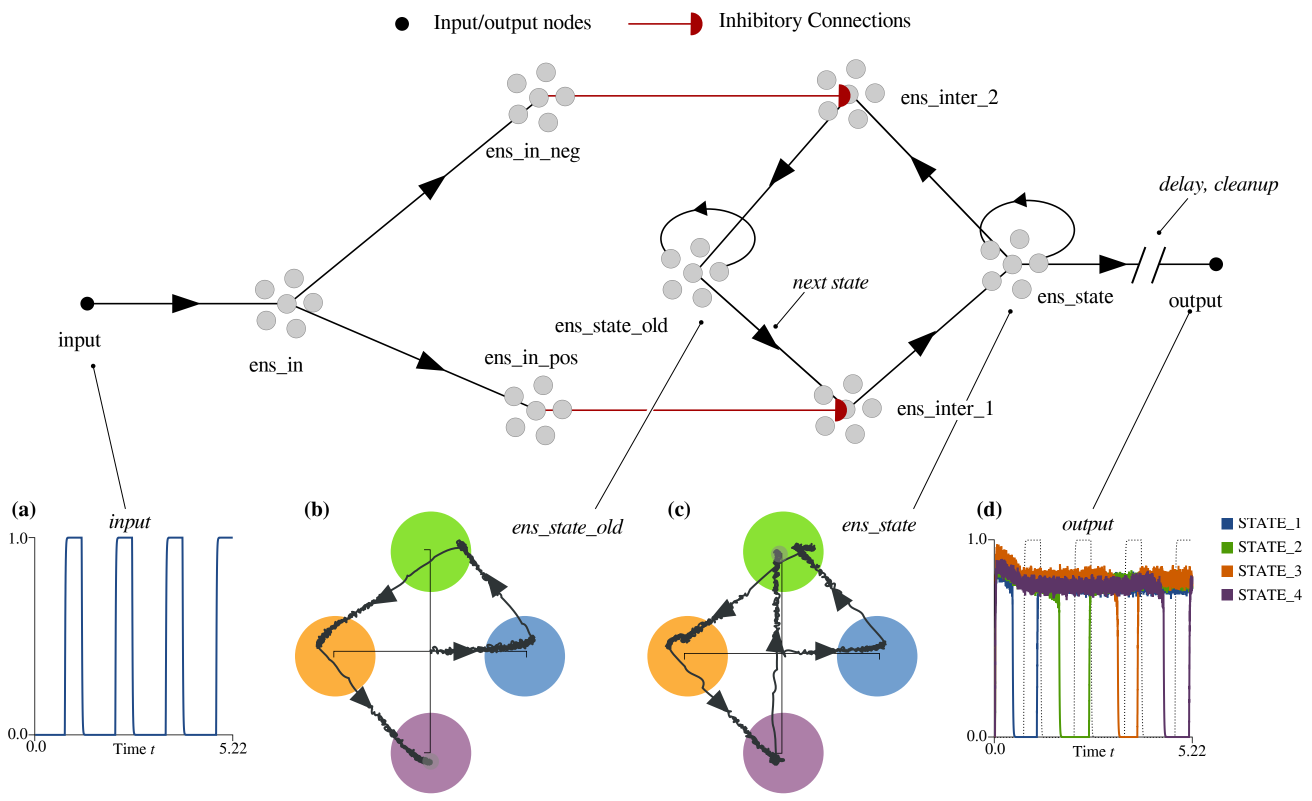

Fig. 5. Sketch of the state transition network used in the implementation of Bug 2. **(a)** exemplary input signal, **(b)** old state trajectory over time in 2D state space; coloured regions indicate state space attractors, **(c)** current state over time in 2D state space, **(d)** inverted output after cleanup and delay; high output values correspond to inactive states being inhibited. See text for a description.

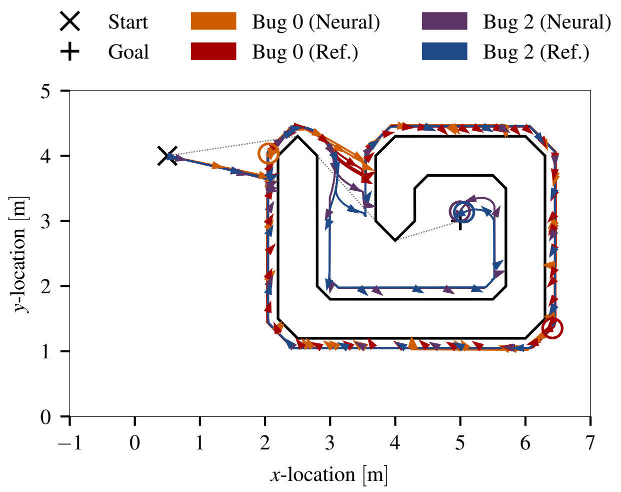

Fig. 6. Robot trajectories in an environment not solvable for Bug 0. Trajectories are similar for the corresponding reference and neural variants.

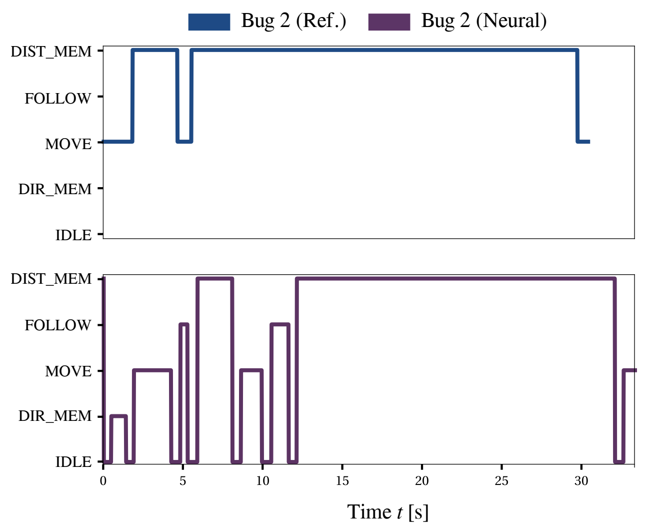

Fig. 7. State over time for the Bug 2 trajectories shown in Figure 6. The reference implementation is faster due to the lack of a memorization phase.

similar, even though slight differences in velocity exist. As shown in the state-over time plot in Figure 7, the neural implementation requires significant time to memorize the angles and directions, whereas this process is practically instant on a traditional computing substrate.

Figure 8 shows a similar experimental setup with a larger map (causing, due to normalization, a reduced resolution for distances and angles) and eight instances of the neural Bug 2. Here, both Bug 0 (neural and reference implementation), as well as the reference implementation of Bug 2 reach the goal, where the reference Bug 2 trajectory is relatively close to the shortest path. Only one instance of the neural Bug 2 closely follows this path, and a total of five instances are stuck in circles around obstacles. Collision with those obstacles is

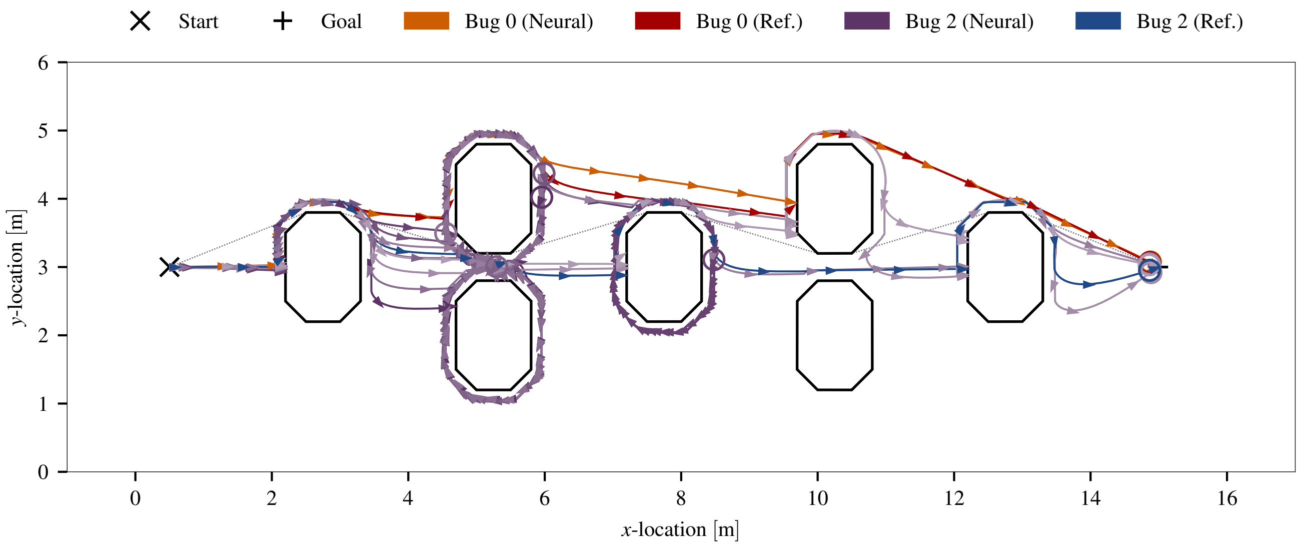

Fig. 8. Robot trajectory in a large environment. Dotted line shows the shortest path assuming a point robot. Three out of eight neural Bug 2 robots reach the goal, the rest endlessly circles obstacles. Both neural and reference Bug 0 robots reach the goal on a highly similar path, yet their trajectory is significantly longer than the reference Bug 2 trajectory.

caused by the robots leaving previous obstacles at a point not exactly on the connecting line between start and goal location due to imprecision in the direction representation. This causes a collision with an obstacle that does not have any intersection with the start-goal line at all. Correspondingly, the affected robots are—at least in theory—not allowed to leave. In practice however, noise and emergent chaotic dynamics may cause the robot to leave the obstacle outline after a few cycles. In contrast, in one case the robot never leaves an obstacle outline which clearly intersects the start-goal line. Further analysis shows that for some reason the neuron ensembles representing state in the state transition network of this instance do not fully converge to the new state, and due to attractor dynamics, the network converges back to the original obstacle-following behaviour.

To summarize, the prototypical implementations suggests that it is *in principle* possible to implement Bug algorithms as neural networks. Especially the results for Bug 0 are basically indistinguishable from the reference implementation. For Bug 2, issues remain with larger maps reducing precision, as well as a lack of fine-tuning of the network dynamics and noise causing (or preventing) switching behaviours at the correct moment in time.

V. Conclusions and Future Directions

I presented a neural version of two basic local navigation policies, namely the *Bug 0* and *Bug 2* algorithm. To this end, I implemented a robot simulator which executes behaviours specified as imperative programs or spiking neural networks. Experiments show that it is clearly possible to realize the aforementioned algorithms on a neural computing substrate with relatively few neurons, assuming the presence of a corresponding sensory system. However, with increasing size and complexity of the virtual robot arena, the performance of the neural *Bug 2* implementation breaks down due to inevitable imprecisions in memory and sensor processing, questioning the biological plausibility of both the provided implementation and the algorithm itself.

Even assuming that biological sensor systems exist which map onto the theoretical one—e.g. some evidence suggests insect antennas being capable as olfactory direction sensing if a steep chemical gradient exists [25]—the provided implementation is not a model in a strict sense, since it does not possess any predictive power concerning actual insect behaviour. With some squinting, behavioural patterns corresponding to Bug 0 can be found in insects. E.g. meticulous research by Wehner et al. shows that desert ants of genus *Cataglyphis* use an odometer for distance measurements, sky polarisation for orientation, and their visual system for landmark navigation when returning to their nest [26]–[28]. However, there are far to many seemingly random components in insect behaviour that contradict the carefully laid out sequence of actions in the Bug 1 and Bug 2 policies, and even *Cataglyphis* resorts to a randomly looking search strategy when its path is obstructed. To conclude, bug algorithms may be too restrictive on the one hand—e.g. insects are capable of using a visual system for orientation and can store more than two variables—but require too high precision on the other hand, e.g. leaving the obstacle outline on *exactly* the intersection line between the start and the goal location in the Bug 2 algorithm.

Future research should focus on more extensive experiments, e.g. performing experiments with varying neuron count, adapting the bug policies to require less precision (better noise resilience), and incorporating results form biological studies on insect and animal navigation behaviour in general, as well as already published insect-inspired algorithms [29]. Not only may this lead to more biologically plausible, but also to more robust algorithms which can be employed in neurorobotics.

References

[1] S. Thrun, "Probabilistic robotics," *Communications of the ACM*, vol. 45, no. 3, pp. 52–57, 2002.

[2] V. J. Lumelsky and A. A. Stepanov, "Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape," *Algorithmica*, vol. 2, no. 1, pp. 403–430, 1987.

[3] V. J. Lumelsky and T. Skewis, "Incorporating range sensing in the robot navigation function," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 20, no. 5, pp. 1058–1069, Sep. 1990, ISSN: 0018-9472. DOI: 10.1109/21.59969.

[4] I. Kamon and E. Rivlin, "Sensory-based motion planning with global proofs," *IEEE Transactions on Robotics and Automation*, vol. 13, no. 6, pp. 814–822, Dec. 1997, ISSN: 1042-296X. DOI: 10.1109/70.650160.

[5] I. Kamon, E. Rimon, and E. Rivlin, "TangentBug: A Range-Sensor-Based Navigation Algorithm," *The International Journal of Robotics Research*, vol. 17, no. 9, pp. 934–953, 1998. DOI: 10.1177/027836499801700903.

[6] S. M. LaValle, *Planning algorithms*. Cambridge, United Kingdom: Cambridge University Press, 2006.

[7] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[8] B. Komer and C. Eliasmith, "A unified theoretical approach for biological cognition and learning," *Current Opinion in Behavioral Sciences*, 2016.

[9] B. Martin, C. K. Machens, and S. Denève, "Predictive Coding of Dynamical Variables in Balanced Spiking Networks," *PLOS Computational Biology*, vol. 9, no. 11, pp. 1–16, Nov. 2013. DOI: 10.1371/journal.pcbi.1003258.

[10] H. Markram, "The Human Brain Project," *Scientific American*, vol. 306, no. 6, pp. 50–55, 2012.

[11] C. Eliasmith and C. H. Anderson, *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. Cambridge, Massachusetts: MIT press, 2003, 380 pp., ISBN: 978-0-262-55060-4.

[12] K. Boahen, "A Neuromorph's Prospectus," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 14–28, 2017.

[13] F. Bullo and S. L. Smith, *Lectures on robotic planning and kinematics*. 2016.

[14] C. Eliasmith, T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen, "A large-scale model of the functioning brain," *Science*, vol. 338, pp. 1202–1205, 2012. DOI: 10.1126/science.1225266.

[15] T. C. Stewart, "The neural engineering framework," *AISB Quarterly*, pp. 2–7, 2012.

[16] C. Eliasmith, *How to Build a Brain: A Neural Architecture for Biological Cognition*, ser. Oxford Series on Cognitive Models and Architectures. Oxford, United Kingdom: Oxford University Press, 2013, ISBN: 978-0-19-979469-0.

[17] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel Methods in Machine Learning," *The Annals of Statistics*, vol. 36, no. 3, pp. 1171–1220, 2008.

[18] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith, "Nengo: A Python tool for building large-scale functional brain models," *Frontiers in Neuroinformatics*, vol. 7, no. 48, 2014.

[19] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. DOI: 10.1145/361002.361007.

[20] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[21] T. Yata, L. Kleeman, and S. Yuta, "Wall following using angle information measured by a single ultrasonic transducer," in *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, vol. 2, May 1998, 1590–1596 vol.2. DOI: 10.1109/ROBOT.1998.677372.

[22] T. C. Stewart, X. Choo, and C. Eliasmith, "Symbolic reasoning in spiking neurons: A model of the cortex/basal ganglia/thalamus loop," in *32nd Annual Meeting of the Cognitive Science Society*, S. Ohlsson and R. Catrambone, Eds., Portland, Oregon: Cognitive Science Society, 2010, pp. 1100–1105.

[23] C. Eliasmith, "A unified approach to building and controlling spiking attractor networks," *Neural computation*, vol. 7, no. 6, pp. 1276–1314, 2005.

[24] ——, "Attractor network," *Scholarpedia*, vol. 2, p. 1380, 2007. [Online]. Available: http://www.scholarpedia.org/article/Attractor_network.

[25] D. Schneider, "Insect antennae," *Annual review of entomology*, vol. 9, no. 1, pp. 103–122, 1964.

[26] G. Hartmann and R. Wehner, "The ant's path integration system: A neural architecture," English, *Biological Cybernetics*, vol. 73, no. 6, pp. 483–497, 1995, ISSN: 0340-1200. DOI: 10.1007/BF00199541.

[27] R. Wehner, "Desert ant navigation: How miniature brains solve complex tasks," *Journal of Comparative Physiology A*, vol. 189, no. 8, pp. 579–588, 2003.

[28] S. Sommer and R. Wehner, "The ant's estimation of distance travelled: Experiments with desert ants, cataglyphis fortis," English, *Journal of Comparative Physiology A*, vol. 190, no. 1, pp. 1–6, 2004, ISSN: 0340-7594. DOI: 10.1007/s00359-003-0465-4.

[29] R. Möller, "A model of ant navigation based on visual prediction," *Journal of theoretical biology*, vol. 305, pp. 118–130, 2012.