

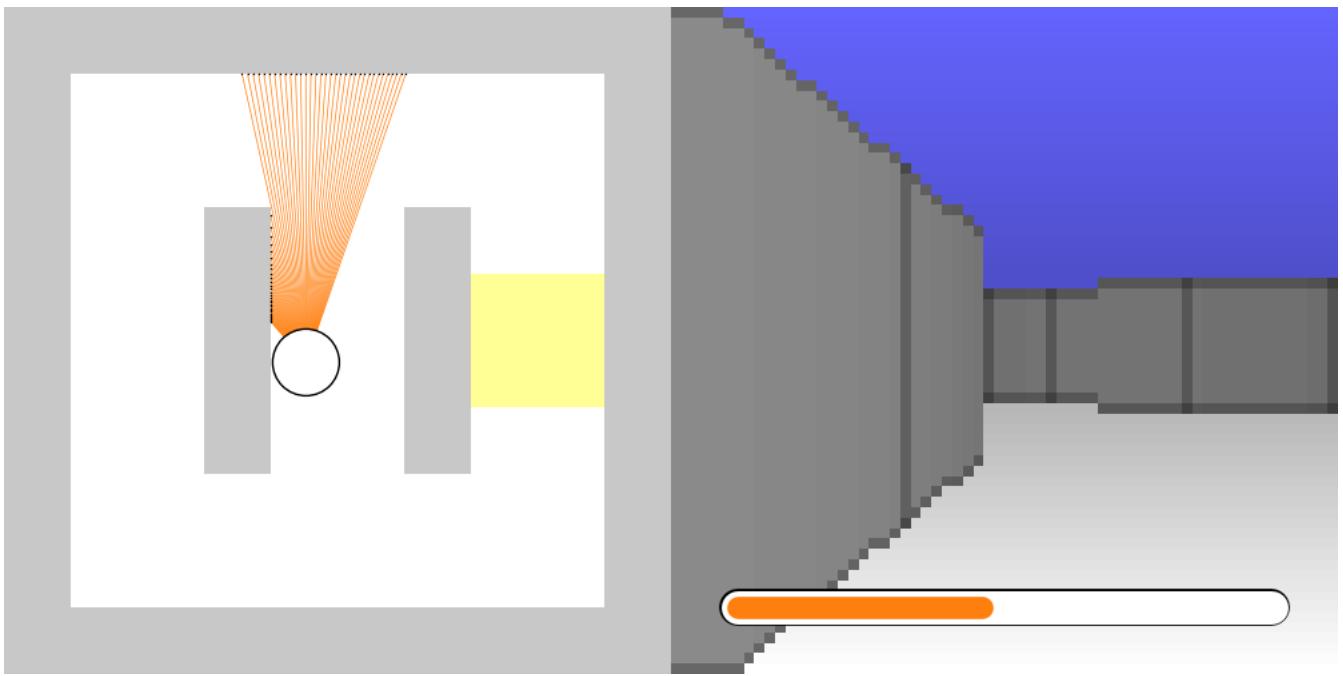
# Contents

<b>1</b>	<b>The braincraft challenge</b>	<b>2</b>
1.1	Simplified problem statement . . . . .	2
1.2	Input preprocessing . . . . .	2
1.2.1	Proximity sensors . . . . .	2
1.2.2	Color sensors . . . . .	2
1.2.3	Energy measurement . . . . .	3
1.3	A putative controller . . . . .	3
1.3.1	Navigation . . . . .	3
1.3.2	Task 1: Simple decision . . . . .	4
1.3.3	Task 1b: Simple decision but varying environment . . . . .	4
1.3.4	Task 2: Cued environment decision . . . . .	5
1.3.5	Task 3: Valued environment decision . . . . .	5
<b>2</b>	<b>Programmatoid computation</b>	<b>5</b>
2.1	Comparison implementation . . . . .	6
2.2	Boolean expression implementation . . . . .	6
2.3	Conditional expression implementation . . . . .	6
<b>3</b>	<b>Neuronoid computation</b>	<b>7</b>
3.1	Neuronoid unit . . . . .	7
3.2	Step-function mollification . . . . .	8
3.3	Approximation of the identify function . . . . .	8
3.4	Approximation of switch mechanisms . . . . .	8
<b>4</b>	<b>Examples of neuronoid computation</b>	<b>9</b>
4.1	Memory gate . . . . .	9
4.2	Multistable mechanisms . . . . .	10
4.2.1	Bistable mechanisms . . . . .	10
4.2.2	Spike generation . . . . .	10
4.2.3	Delay . . . . .	10
4.2.4	Oscillation . . . . .	10
4.3	The explog function . . . . .	11
4.4	Neuronoid approximation of <code>exp</code> and <code>log</code> . . . . .	11
<b>5</b>	<b>Comparing <code>tanh</code> with other non linearities</b>	<b>12</b>
<b>6</b>	<b>Using the braincraft challenge setup</b>	<b>14</b>
6.1	Installation of the setup . . . . .	14
6.2	Running at the programmatic level . . . . .	15
6.3	Running at the programmatoid level . . . . .	15

# 1 The braincraft challenge

Let us consider the following digital experimental setup, as described in braincraft challenge presentation<sup>1</sup>.

## 1.1 Simplified problem statement



The braincraft challenge<sup>2</sup> bot moves at a constant speed with sensor inputs and one orientation output, it uses some energy and refills this energy on a given yellow location. The 2D space size is  $[0, 1] \times [0, 1]$ . The bot starts in the middle and oriented at 90, i.e., upward.

Input variables	
$p_l \in [0_{\text{wall-hit}}, 1_{\text{no-wall}}[$	Leftward proximity, max value of the $[0, +30^\circ]$ range 32 left sensors.
$p_r \in [0_{\text{wall-hit}}, 1_{\text{no-wall}}[$	Rightward proximity, max value of the $[-30^\circ, 0]$ range 32 right sensors.
$c_{l\bullet} \in \{0, 1\}, \bullet \in \{b_{\text{blue}}, r_{\text{red}}\}$	Leftward binary red and blue color detectors.
$c_{r\bullet} \in \{0, 1\}, \bullet \in \{b_{\text{blue}}, r_{\text{red}}\}$	Rightward binary red and blue color detectors.
$g_e \in [0_{\text{death}}, 1_{\text{full}}]$	Energy gauge value.
Output variable	
$d_o \in [-5, 5]$	Orientation difference, saturated at $\pm 5^\circ$ .

With respect to the original braincraft challenge:

- we consider two leftward and rightward “average” sensors only,
- color is input as binary variables, and always available,
- the wall hit indicator is not used.

## 1.2 Input preprocessing

### 1.2.1 Proximity sensors

**Motivation** Simplifies the left-right navigation by compacting the leftward and rightward sensors as a simple pair of input.

**Implementation** A simple sum or average could be used, thus using directly a linear combination of the input in afferent units.

We also can use the explog function as derived in Appendix 8, enjoying a neuronoid approximation, and allowing to balance between averaging and computing the maximum.

### 1.2.2 Color sensors

**Motivations** Again, color blob detection is to perform either on the left or on the right, allowing the color input to be compacted.

For each color, a channel is specified, simplifying the programmatoid implementation and providing an input closer to

<sup>1</sup><https://github.com/rougier/braincraft/blob/master/README.md#introduction>

<sup>2</sup><https://github.com/rougier/braincraft>

biological colored vision. Since the setup color input is a discrete color index, the channel value is binary, accounting for the presence, or not, of a least one related color index.

**Implementation** For the distributed implementation, each camera color index value is mapped on each color channel input with the 0 value if the color is different and the 1 value if equal, e.g., using step unit:

$$\begin{aligned} c_{\dagger \bullet} &\leftarrow H(\sum_{k \in K} (1 - D(i_{\bullet} - c[k])), c[k] \in \mathcal{N} \\ \dagger &\in \{l_{\text{left}}, r_{\text{right}}\}, \bullet \in \{b_{\text{blue}}, r_{\text{red}}\} \\ D(x) &\stackrel{\text{def}}{=} H(x - 1/2) + H(-x - 1/2) = \begin{cases} \text{if } |x| < 1/2 \text{ then } 0 \text{ else } 1 \end{cases} \end{aligned}$$

where  $K$  stands for the left or right sensor related indexes,  $i_{\bullet}$  stands for the color index, and  $c[k]$  stands for the sensor color index value.

### 1.2.3 Energy measurement

**Motivation** The energy increase accumulation is to be pre-processed, since used in some task.

#### Processed Variables

$$\begin{aligned} g_{cb} &\in [0, 1], b \in \{1, 2\} & g_{cb}|_{t=0} = 0 & \text{Last and last-before-last cumulative energy increases.} \\ g_{eb} &\in [0, 1], b \in \{1, 2\} & g_{eb}|_{t=0} = 0 & \text{Last and last-before-last energy value.} \end{aligned}$$

Here instantaneous energy increase is  $(g_e - g_{e1})$ , and we assume that the energy always changes so that  $g_e \neq g_{e1}$ .

#### Implementation

Cumulating energy increase starts, saving last increase in  $g_{c2}$ .

if  $\underbrace{g_e > g_{e1}}$  and  $\underbrace{g_{e1} < g_{e2}}$  then  $g_{c2} \leftarrow g_{c1}$ ,  $g_{c1} \leftarrow (g_e - g_{e1})$   
increase after a decrease

Cumulating energy increase continues.

if  $\underbrace{g_e > g_{e1}}$  and  $\underbrace{g_{e1} > g_{e2}}$  then  $g_{c1} \leftarrow g_{c1} + (g_e - g_{e1})$   
increase after an increase

Otherwise  $g_e < g_{e1}$ , thus cumulating energy increase stops, and  $g_{c1}$  is memorized.

$g_s \stackrel{\text{def}}{=} g_e < g_{e1}$  and  $g_{e1} > g_{e2}$  indicates that cumulating energy increase has just stopped.

$g_n \stackrel{\text{def}}{=} g_e < g_{e1}$  and  $g_{e1} < g_{e2}$  indicates that cumulating energy increase did stop before.

#### Pseudo programmatoid solution

$$\begin{aligned} g_{c1} &\leftarrow g_{c1} - H(g_{e2} - g_{e1}) \\ g_{c2} &\leftarrow g_{c2} + H(H(g_e - g_{e1}) + H(g_{e2} - g_{e1}) - 3/2)(g_{c1} - g_{c2}) \\ \text{Then} \\ g_{e2} &\leftarrow g_{e1} \\ g_{e1} &\leftarrow g_e \end{aligned}$$

which is not a pure programmatoid solution because, because of term of the form  $H(u)v$ , thus with a product, but is implementable in the neuronoid framework discussed in the sequel. In brief:

$$H(x)y \simeq \omega' h(y/\omega' + \omega h(\omega x) - \omega)$$

where  $h(\cdot)$  is the sigmoid function, and  $\omega$  and  $\omega'$  are sufficiently large numbers.

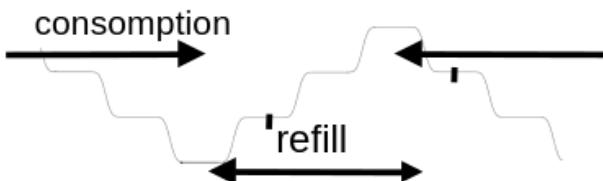


Figure 1: Representation of energy profile when cumulating energy increase starts (first tick), with  $g_e > g_{e1} < g_{e2}$ , and stops (last tick), with  $g_e < g_{e1}$ , while during refill  $g_e > g_{e2}$ .

### 1.3 A putative controller

#### 1.3.1 Navigation

**Heuristic :** The bot runs at constant velocity,

- (i) ahead by default, thanks to a linear correction, high enough to correct the direction, small enough to avoid

oscillations, and

- (ii) attempts to perform a quarter-turn in the preferred orientation as soon as passing is detected.
- With this navigation mechanism the bot performs either leftward or rightward half-loops, traversing the central corridor.

### Internal variable

$$q_p \in \{0_{\text{leftward}}, 1_{\text{rightward}}\}, \quad q_p|_{t=0} = 0 \quad \text{Preferred quarter-turn direction.}$$

### Programmatoid solution :

$$\begin{aligned} d_o &\leftarrow \underbrace{\gamma(p_l - p_r)}_{\substack{\text{linear correction to} \\ \text{maintain direction ahead}}} + \underbrace{\alpha(t_l - t_r)}_{\substack{\text{quarter- turn} \\ \text{left or right}}} \\ t_l &\leftarrow (1 - q_p) H(\beta - p_l) = H(H(\beta - p_l) - q_p - 1/2) \\ t_r &\leftarrow q_p H(\beta - p_r) = H(H(\beta - p_r) + q_p - 3/2) \end{aligned}$$

where:

$w \simeq 1/4$	Rough estimation of the path-width.
$\gamma = \frac{5}{w/2}$	Saturates the correction at $5^\circ$ if the depth difference is half of the path-width.
$\alpha = 90$	Saturates the correction at $\pm 90^\circ$ to make the quarter-turn, since the linear $ \gamma(p_l - p_r)  < 40$ is lower than the quarter-turn term, the latter subsumes the former.
$\beta = w$	Triggers the quarter-turn if the depth is higher than the path-width.
$\omega = 10$	Transform a boolean product to a step-function threshold.

while:

$$\begin{aligned} t_l &= 1 \text{ iff } q_p = 0 \text{ and while } \beta < p_l \\ t_r &= 1 \text{ iff } q_p = 1 \text{ and while } \beta < p_r \end{aligned}$$

in words: we execute the quarter-turn until another wall is detected<sup>3</sup>.

We thus have a linear output unit for  $d_o$  and two step-unit for  $t_l$  and  $t_r$ .

Given this controller the design reduces to navigation direction choice, i.e., control the  $q_p$  variable.

### 1.3.2 Task 1: Simple decision

**Strategy** Restrict navigation to the half-loop that contains the energy source, while the other does not.

**Heuristic** If the energy is too low, thus looping in the wrong direction, the direction is changed once.

At start  $q_p = 0$ . Then, if the energy is too low it changes once to  $q_p = 1$ .

$$q_p = \text{if } q_p = 1 \text{ or } \eta > g_e \text{ then } 1 \text{ else } 0$$

### Programmatoid solution

$$q_p \leftarrow H(\omega q_p + (\eta - g_e))$$

where:

$c = 1/1000$	Energy consumption at each step.
$s = 1/100$	Speed: location increment at each steps.
$b = 3/2$	Distance bound between the starting point and the putative energy sources.
$\eta \simeq b c/s = 3/20$	Energy consumption threshold if no source on the path.

### 1.3.3 Task 1b: Simple decision but varying environment

**Strategy** Restrict navigation to the half-loop that contains the energy source, while the other does not, this may change with time.

---

<sup>3</sup>The indentities:

$$\begin{aligned} (1 - q_p) H(\beta - p_l) &= H(H(\beta - p_l) - q_p - 1/2) \\ q_p H(\beta - p_r) &= H(H(\beta - p_r) + q_p - 3/2) \end{aligned}$$

are easy to verify using a simple truth table.

## Heuristic

- If the energy is too low, the direction is inverted.
- This is registered, avoiding multiple changes at low energy.
- When the energy is high enough, change registration is reset.

## Internal variable

$$g_c \in \{0, 1\} \quad g_c|_{t=0} = 0 \quad \text{Registers if the low energy has been detected.}$$

## Programmatoid solution

Inverts the direction if to be changed.

$$q_p \leftarrow \text{if } \eta > g_e \text{ and } g_c = 0 \text{ then } 1 - q_p \text{ else } q_p$$

Registers the inversion until the energy is high enough.

$$g_c \leftarrow \text{if } g_c = 0 \text{ and } \eta > g_e \text{ then } 1 \text{ elif } g_c = 1 \text{ and } 2\eta < g_e \text{ then } 0 \text{ else } g_c$$

In the sequel, we are going to derive a generic way to compile such an expression with binary values as a programmatoid .

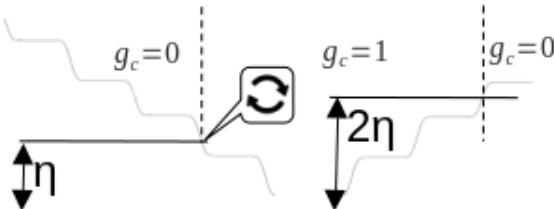


Figure 2: Representation of  $g_c$  value at different level of energy with the indication of the direction change.

### 1.3.4 Task 2: Cued environment decision

**Strategy** Restrict navigation to the half-loop without a closed path, as indicated by a color that has already been seen once.

**Heuristic** Detect and store the first color blob, and choose to turn in the direction it appears again.

The detection is reset when the energy decreases.

## Internal variable

$$c_{c\bullet} \in \{0, 1\}, \bullet \in \{b_{\text{blue}}, r_{\text{red}}\} \quad c_{p\bullet}|_{t=0} = 0 \quad \text{Detected the color cue code, if any.}$$

## Programmatoid solution

Detect the cue, if not yet done, and reset below an energy threshold

$$c_{cb} \leftarrow \text{if } c_{cb} = 0 \text{ and } c_{cr} = 0 \text{ and } (c_{lb} = 1 \text{ or } c_{rb} = 1) \text{ then } 1 \text{ elif } g_e < \eta/2 \text{ then } 0 \text{ else } c_{cb}$$

$$c_{cr} \leftarrow \text{if } c_{cb} = 0 \text{ and } c_{cr} = 0 \text{ and } (c_{lr} = 1 \text{ or } c_{rr} = 1) \text{ then } 1 \text{ elif } g_e < \eta/2 \text{ then } 0 \text{ else } c_{cr}$$

Set the direction according to the cue

$$q_p \leftarrow \text{if } c_{lb} = c_{cb} \text{ or } c_{lr} = c_{cr} \text{ then } 0 \text{ elif } c_{rb} = c_{cb} \text{ or } c_{rr} = c_{cr} \text{ then } 1 \text{ else } q_p$$

### 1.3.5 Task 3: Valued environment decision

**Strategy and Heuristic** Test energy sources and change direction if the latter yields less increase than the former.

## Programmatoid solution

$$q_p \leftarrow \text{if } \underbrace{g_e < g_{e1} \text{ and } g_{e1} > g_{e2}}_{\text{energy increase just stopped.}} \text{ and } \underbrace{g_{c2} > g_{c1}}_{\text{previous energy increase is higher.}} \text{ then } 1 - q_p \text{ else } q_p$$

## 2 Programmatoid computation

We name “programmatoid” computation the conception of an input-output straight-line program<sup>4</sup> implementing an operator on numerical value expressions using an LN-system with the step-function as non-linearity, this writing:

<sup>4</sup>[https://en.wikipedia.org/wiki/Straight-line\\_program](https://en.wikipedia.org/wiki/Straight-line_program)

$$o_n[t] \leftarrow H(\sum_{m \in \{1, M\}} w_{nm} i_m[t-1] + w_{n0}), n \in \{1, N\}$$

where  $i_m[t]$  is the  $m$ -th input at a discrete  $t$  and  $o_n[t]$  is the  $n$ -th output, including recurrent system with some output corresponding to inputs, while  $w_{nm}, n \in \{1, N\}, m \in \{0, M\}$  are the systems parameters, or, weights for  $m > 0$  and bias for  $m = 0$ .

The step-function, also called Heaviside function, implements the computation of the sign of a value:

$$\begin{aligned} H(x) &\stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x > 0 \\ 1/2 & \text{if } x = 0 \\ 0 & \text{if } x < 0 \end{cases} \\ &= \text{if } x > 0 \text{ then } 1 \text{ elif } x = 0 \text{ then } 1/2 \text{ else } 0. \end{aligned}$$

The design choice of  $H(0) = 1/2$ , instead for instance  $H(0) = 0$ , this latter simplifying some formula, is due the neuronoid approximation developed in the sequel.

By extension, we use, for any property  $\mathcal{P}$ , the notation:

$$H(\mathcal{P}) \stackrel{\text{def}}{=} \text{if } \mathcal{P} \text{ then } 1 \text{ else } 0$$

## 2.1 Comparison implementation

Given two numerical variables  $v_1$  and  $v_2$ , using the notation , we have the equivalence<sup>5</sup>, considering *true* as 1 and *false* as 0:

$H(v_1 > v_2)$	$H(v_1 \leq v_2)$	$H(v_1 = v_2)$	
$H(2H(v_1 - v_2) - 3/2)$	$H(H(v_1 - v_2))$	$\frac{H(H(v_1 - v_2)) + H(H(v_2 - v_1))}{2} - 1$	with $H(0) = 1/2$
$H(v_1 - v_2)$	$1 - H(v_1 - v_2)$	$\frac{1 - H(v_1 - v_2) + H(v_2 - v_1)}{2}$	with $H(0) = 0$

while  $H(v_1 \neq v_2) = 1 - H(v_1 = v_2)$ , we thus can implement any numerical comparison as a programmatoid.

Let us notice that all arguments  $x$  of the step function  $H(\cdot)$ , except the term  $H(v_1 - v_2)$ , verify  $|x| \geq 1/2$ , this will be reused.

## 2.2 Boolean expression implementation

Given binary variables  $b_n \in \{0_{\text{false}}, 1_{\text{true}}\}, n \in \{1, N\}$  we have the obvious<sup>6</sup> correspondence:

$b_1$	$b_1$ and $b_2$	$b_1$ or $b_2$	not $b_1$
$H(b_1 - 1/2)$	$b_1 b_2 = H(b_1 + b_2 - 3/2)$	$H(b_1 + b_2 - 1/2)$	$1 - b_1 = H(1/2 - b_1)$

and more generally:

$$\begin{aligned} \text{and}_{n \in \{1, N\}} &= H\left(\sum_{n \in \{1, N\}} b_n - N + 1/2\right) = \prod_{n \in \{1, N\}} H(b_n - 1/2) = \prod_{n \in \{1, N\}} b_n \\ \text{or}_{n \in \{1, N\}} &= H\left(\sum_{n \in \{1, N\}} b_n - 1/2\right) \end{aligned}$$

An interesting consequence is that binary variable products can be translated to a programmatoid.

Let us also notice that all arguments  $x$  of the step function  $H(\cdot)$  still verify  $|x| \geq 1/2$ .

## 2.3 Conditional expression implementation

A conditional expression on variables on any numerical type  $v_n, n \in \{0, 1\}$  with a binary variable  $b_1 \in \{0, 1\}$  writes:

$$\begin{aligned} v &\rightarrow \text{if } b_1 \text{ then } v_1 \text{ else } v_0 \\ &= (1 - b_1) v_0 + b_1 v_1 \\ \text{while, for binary variable, i.e., if and only if } v_n &\in \{0, 1\}, n \in \{0, 1\}: \\ &= H(v_0 + (1 - b_1) - 3/2) + H(v_1 + b_1 - 3/2) \\ &= H(v_0 - b_1 - 1/2) + H(v_1 + b_1 - 3/2) \\ &= H(H(v_0 - b_1 - 1/2) + H(v_1 + b_1 - 3/2)) \end{aligned}$$

as easy to verify, using for instance a truth table. Here:

- products with  $(1 - b_1)$  and  $b_1$  behaves as switches between  $v_0$  and  $v_1$

<sup>5</sup>Considering the pseudo truth table, with  $H(0) = 1/2$ :

	$H(v_1 > v_2)$	$H(v_1 = v_2)$	$H(v_1 < v_2)$
$H(v_1 - v_2)$	1	$1/2$	0
$2H(v_1 - v_2) - 3/2$	$1/2$	$-1/2$	$-3/2$
$H(2H(v_1 - v_2) - 3/2)$	1	0	0
$H(H(v_1 - v_2))$	1	1	0
$H(H(v_2 - v_1))$	0	1	1
$H(H(v_1 - v_2)) + H(H(v_2 - v_1)) - 1$	0	1	0

we obtain the expected results.

<sup>6</sup>Easy to verify with, e.g., a truth table, and by induction for the generalized formula.

- when  $v_i$  are binary, we are left with a two layer computation, the first layer being built from two programmatoid, and the second layer from either another programmatoid or a simple linear unit, i.e., a linear combination of values.

This generalizes to conditional expressions on variables on any numerical type  $v_n, n \in \{0, N\}$ : <sup>7</sup>:

$$\begin{aligned}
v &\rightarrow \text{if } b_1 \text{ then } v_1 [\text{elif } b_n \text{ then } v_n]_{n \in \{2, N\}} \text{ else } v_0 \\
&= \prod_{n' \in \{1, N\}} (1 - b_{n'}) v_0 \\
&+ \sum_{n \in \{1, N\}} b_n \prod_{\substack{n' \in \{1, N\}, \\ n' \neq n}} (1 - b_{n'}) v_n \\
&= \sum_{n \in \{0, N\}} h_n v_n \\
\text{with } h_0 &\stackrel{\text{def}}{=} H(1/2 - \sum_{n' \in \{1, N\}} b_{n'}) \in \{0, 1\} \\
\text{and } h_n &\stackrel{\text{def}}{=} H(b_n - \sum_{n' \in \{1, N\}, n' \neq n} b_{n'} - 1/2) \in \{0, 1\}, n \in \{1, N\}
\end{aligned}$$

while, for binary variable, i.e., if and only if  $v_n \in \{0, 1\}, n \in \{0, N\}$ :

$$\begin{aligned}
&= H(v_0 - \sum_{n' \in \{1, N\}} b_{n'} - 1/2) \\
&+ \sum_{n \in \{1, N\}} H(v_n + b_n - \sum_{n' \in \{1, N\}, n' \neq n} b_{n'} - 3/2).
\end{aligned}$$

Let us also notice that all arguments  $x$  of the step function  $H(\cdot)$  again verify  $|x| \geq 1/2$ .

As a consequence,

- A  $N$ -term conditional expression on binary variables reduces to a two layers programmatoid of  $N$  and 1 unit, the former unit being either linear or with a step-wise function.

- A  $N$ -term conditional expression on numerical variables reduces to a two layers system of  $N$  programmatoid and an output unit with  $h_n$  exclusive switches.

### 3 Neuronoid computation

#### 3.1 Neuronoid unit

By “neuronoid” we name the not very biologically plausible<sup>8</sup> simplest biological neuron or neuron small ensemble inspired by mean-field modelisation<sup>9</sup> of the Hodgkin–Huxley neuronal axon model<sup>10</sup>.

We define the equation<sup>11</sup>:

$$\begin{aligned}
\tau \frac{\partial v_i}{\partial t}(t) + v_i(t) &= z_i(t), \quad z_i(t) \stackrel{\text{def}}{=} h \left( \sum_j w_{ij} v_j(t) + w_{i0} \right), \\
h(x) \stackrel{\text{def}}{=} \frac{1}{1 + \exp(-4x)} &= h(x_0) + \operatorname{sech}(2x_0)^2 (x - x_0) + O((x - x_0)^2), \\
&= 1 - \exp(-4x) + O(\exp(-8x))
\end{aligned}$$

where  $v_i$  is the membrane potential, so that:

$$\begin{aligned}
v(t) &= 1/\tau \int_0^t e^{-(t-s)/\tau} z(s) ds + v(0) e^{-t/\tau} \\
&= z(0) + e^{-t/\tau} (v(0) - z(0))|_{z(t)=z(0)} \quad (\text{constant input}) \\
&= z(t)|_{\tau=0} \quad (\text{no leak}),
\end{aligned}$$

and the corresponding discrete approximation using an Euler schema writes:

$$\begin{aligned}
&\simeq (1 - \gamma) v(t - \Delta t) + \gamma z(t - \Delta t)) \\
&= \sum_{s=0}^{t-1} \gamma (1 - \gamma)^{t-s+1} z(s) + v(0) (1 - \gamma)^t \\
&= z(0) + (1 - \gamma)^t (v(0) - z(0))|_{z(t)=z(0)} \quad (\text{constant input}) \\
&= z(t)|_{\gamma=1} \quad (\text{no leak}).
\end{aligned}$$

with  $0 < \gamma < 1$  and  $0 < \tau$  in correspondence:

<sup>7</sup>By induction, considering the second line, one one hand, due to the products, if  $b_n = 0, n \in \{1, N\}$  we obtain  $v_0$ . On the other hand, if  $b_k = 0, k < K$  and  $b_K = 1$ , due the products, we obtain  $v_K$ , which is precisely the semantic of the conditional expression of the first line.

The 3rd line is deduced from the second line, transforming the binary products on the corresponding programmatoid while:

$$\begin{aligned}
h_0 &\stackrel{\text{def}}{=} H(\sum_{n' \in \{1, N\}} (1 - b_{n'}) - N + 1/2) \\
&= H(1/2 - \sum_{n' \in \{1, N\}} b_{n'}) \\
h_n &\stackrel{\text{def}}{=} H(b_n + \sum_{n' \in \{1, N\}, n' \neq n} (1 - b_{n'}) - N + 1/2) \\
&= H(b_n - \sum_{n' \in \{1, N\}, n' \neq n} b_{n'} - 1/2)
\end{aligned}$$

The 4rd line integrates the  $v_n$  variables in the programmatoid sum, since they are binary, and provides the same obvious algebraic reduction as for the 3rd line.

<sup>8</sup>[https://en.wikipedia.org/wiki/Biological\\_neuron\\_model#Relation\\_between\\_artificial\\_and\\_biological\\_neuron\\_models](https://en.wikipedia.org/wiki/Biological_neuron_model#Relation_between_artificial_and_biological_neuron_models)

<sup>9</sup><https://inria.hal.science/cel-01095603v1>

<sup>10</sup>[https://en.wikipedia.org/wiki/Hodgkin–Huxley\\_model](https://en.wikipedia.org/wiki/Hodgkin–Huxley_model)

<sup>11</sup>We also have:

$$h(x) \stackrel{\text{def}}{=} \frac{1}{1 + \exp(-4x)} = \frac{1 + \tanh(2x)}{2} = \operatorname{sech}(2x) e^{2x}/2,$$

$$\begin{aligned}\gamma &\stackrel{\text{def}}{=} 1 - \exp(-1/\tau) \Leftrightarrow \tau = 1/\log(1/(1-\gamma)), \\ &\text{while} \\ &\lim_{\gamma \rightarrow 0} \tau = +\infty, \lim_{\gamma \rightarrow 1} \tau = 0.\end{aligned}$$

Here,  $h(\cdot)$  is the normalized sigmoid with

$$h(-\infty) = 0, h(0) = 1/2, h'(0) = 1, h(+\infty) = 1, h(x) = 1 - h(-x)$$

All this is just very standard derivations, available as `Maple` code<sup>12</sup>, of the vanilla neuron model, as already proposed by [Lapicque, 1907].

We call “neuronoid computation” the conception of an input-output transform based on feed-forward and recurrent combination of neuronoids, as defined previously.

### 3.2 Step-function mollification

The step-function approximates sigmoid with a huge slope at zero, i.e.:

$$\forall x \neq 0, H(x) = \lim_{\omega \rightarrow +\infty} h(\omega x), h'(\omega x)|_{x=0} = \omega$$

while the convergence is also obtained for  $v = 0$  in the distribution sense with  $H(0) = h(0) = 1/2$ . More precisely, the  $\mathcal{L}_1$  error magnitude, on  $]-\infty, +\infty[$ , writes:

$$|\epsilon_{H,\omega}(x)|_{\mathcal{L}_1} = \frac{\log(2)}{2} \frac{1}{\omega}, \epsilon_{H,\omega}(x) \stackrel{\text{def}}{=} H(x) - h(\omega x)$$

while:

$$|\epsilon_{H,\omega}(x)| = e^{-4\omega} + O(e^{-8\omega})$$

It has been noticed that, except for numerical comparisons, all arguments  $x$  of the step function  $H(\cdot)$  verify  $|x| \geq 1/2$ , and the related error is negligible as soon as, say,  $\omega \geq 10$ :

$\omega$	1	2	5	10	20	50
$\epsilon_{\omega}(\pm 1/2)$	0.119	0.0180	$0.455 \cdot 10^{-4}$	$0.206 \cdot 10^{-10}$	$0.426 \cdot 10^{-19}$	$0.372 \cdot 10^{-45}$

*Any programmatoid computation involving the  $H(\cdot)$  function can thus be approximated by a neuronoid, with  $\tau = 0$ , and sufficiently large  $\omega$ .*

Derivations are available as `Maple` code<sup>13</sup>.

### 3.3 Approximation of the identify function

We also have to use a sigmoid to approximate the identity function defining:

$$\begin{aligned}l_{\omega'}(x) &\stackrel{\text{def}}{=} \omega' (h(x/\omega') - 1/2) \\ &= x - \frac{4}{3} \frac{1}{\omega'^2} x^3 + O(x^5)\end{aligned}$$

and writing  $\epsilon_{l,\omega'}(x) \stackrel{\text{def}}{=} x - l_{\omega'}(x)$ , the  $\mathcal{L}_1$  error magnitude on  $[-M, +M]$  writes:

$$\int_{x=-M}^{x=+M} |\epsilon_{l,\omega'}(x)| dx = \frac{2}{3} \frac{M^4}{\omega'^2} + O\left(\frac{1}{\omega'^4}\right)$$

and the related  $\mathcal{L}_0$  error magnitude:

$$\max(|\epsilon_{l,\omega'}(x)|), x \in [-M, M] = \epsilon_{l,\omega'}(M) = \frac{4}{3} \frac{M^3}{\omega'^2} + O\left(\frac{1}{\omega'^4}\right)$$

The identity function is thus impaired by a bias  $\pm \frac{4}{3} \frac{M^3}{\omega'^2}$  reducing the output value with respect to the input value. Assuming values are normalized, in the  $[-1, +1]$  we obtain, for the  $\mathcal{L}_0$  error magnitude:

$\omega'$	10	50	100	200	500	1000
$\epsilon_{l,\omega'}(1)$	0.0131	$0.533 \cdot 10^{-3}$	$0.133 \cdot 10^{-3}$	$0.333 \cdot 10^{-4}$	$0.54 \cdot 10^{-7}$	$0.12 \cdot 10^{-7}$

with a negligible error for, say,  $\omega' \leq 100$ .

Derivations are available as `Maple` code<sup>14</sup>.

### 3.4 Approximation of switch mechanisms

Given normalized floating point variables  $v_n \in [-1, 1], n \in \{1, N\}$  and binary variables  $b_n \in \{0, 1\}, n \in \{1, N\}$ , conditional expressions and related mechanisms require the implementation of formula of the form<sup>15</sup>:

<sup>12</sup><https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/neuronoid mpl.out.txt>

<sup>13</sup><https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/mollification mpl.out.txt>

<sup>14</sup><https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/linearapproximation mpl.out.txt>

<sup>15</sup>We easily verifies that:

- if  $b_n = 0$ , while  $|v_n/\omega'| \leq 1/\omega' \ll 1$ , the term  $\omega' h(v_n/\omega' + \omega b_n - \omega) \simeq h(-\omega) \simeq 0$  up to  $O(e^{-4\omega})$  as derived previously, while  
- if  $b_n = 1$ , the term  $\omega' h(v_n/\omega' + \omega b_n - \omega) = \omega' h(v_n/\omega')$  corresponds to the approximation of the identity function, up to  $O(\frac{1}{\omega'})$  as derived previously.

$$\begin{aligned} v &= \sum_{n \in \{0, N\}} b_n v_n \\ &= \sum_{n \in \{0, N\}} \omega' h(v_n / \omega' + \omega b_n - \omega) + O\left(\frac{1}{\omega'^2}\right) + O(e^{-4w}) \end{aligned}$$

The implementation as neuronoid, thus allows to implement more formula than only using programmatoid.

## 4 Examples of neuronoid computation

A step ahead, we considering neuronoid with  $\tau > 0$  it seems obvious that we can designed temporizing mechanisms, oscillators and sequence generator, sleep sort mechanism, etc.

### 4.1 Memory gate

For a data input  $i[t]$ , a control  $i_l[t] \in \{0, 1\}$ , and an output  $o[t]$ , the equation:

$$o[t] \leftarrow \text{if } i_l[t] = 1 \text{ then } o[t-1] \text{ else } i[t]$$

implements, if  $i[t] \in \{0, 1\}$ ,  $o[t] \in \{0, 1\}$ , a 1 bit memory, i.e., also called a RS gate, and reusing the previous conditional instruction parameters.

The key point is that it is now a recurrent system, which stability is obvious at the programmatoid level, but not necessarily at the neuronoid level, since we have an recurrent equation for  $i_l[t] = 1$ .

$$\begin{aligned} \text{For } i[t] \in \{0, 1\}, o[t] \in \{0, 1\} \text{ at the programmatoid level:} \\ o[t] \leftarrow H(i[t] - i_l[t] - 1/2) + H(o[t-1] + i_l[t] - 3/2) \end{aligned}$$

$$\begin{aligned} \text{For } i[t] \in \{0, 1\}, o[t] \in \{0, 1\}: \\ o[t] \leftarrow h(\omega(i[t] - i_l[t] - 1/2)) + h(\omega(o[t-1] + i_l[t] - 3/2)) \\ = O(e^{-4\omega}) + h(\omega(o[t-1] - 1/2))|_{i_l[t]=1} \end{aligned}$$

$$\begin{aligned} \text{For } i[t] \in [-1, 1], o[t] \in [-1, 1]: \\ o[t] \leftarrow \omega'(h(i[t]/\omega' - \omega i_l[t]) + h(o[t-1]/\omega' - \omega(1 - i_l[t]))) \\ = O(\omega' e^{-4\omega}) + \omega' h(o[t-1]/\omega')|_{i_l[t]=1} \end{aligned}$$

- The former equation is exact, so entirely stable during iterations.

- For the second equation, memorizing  $o[0] \in \{0, 1\}$ , thus with  $i_l[t] = 1$ , for

$$\epsilon[t] \stackrel{\text{def}}{=} \begin{cases} o[t] & |_{o[0]=0} \\ 1 - o[t] & |_{o[0]=1} \end{cases}$$

we obtain<sup>16</sup>:

$$\begin{aligned} \epsilon[t] &= (1 - 2o[0])h(-(3/2 - i[t])\omega) + h(\omega(\epsilon[t-1] - 1/2)) \\ &= \begin{cases} 0 & \epsilon[t-1] = 0 \quad \text{if } o[0] = 1 \\ e^{-2\omega} + O(e^{-4\omega}) & \epsilon[t-1] = \varepsilon e^{-2\omega} \quad \text{otherwise} \end{cases} \end{aligned}$$

In words : the iteration remains stable, and the error at the order of magnitude of  $O(e^{-2\omega})$ . In particular the previous bias  $\varepsilon$  only impacts the  $O(e^{-4\omega})$  terms. Furthermore, if  $o[0] = 1$ , the positive bias yields a convergence towards 1 without any bias if  $i[t] = 1$ , and with a exponentially decreasing bias if  $i[t] = 0$ .

- For the third equation, memorizing  $o[0] \in [-1, 1]$ , thus with  $i_l[t] = 1$ , while  $i[t] \in [-1, 1]$ , we obtain:

$$\begin{aligned} o[t] &\leftarrow \omega'(h(i[t]/\omega' - \omega i_l[t]) + h(o[t-1]/\omega' - \omega(1 - i_l[t]))) \\ \text{while, since } i_l[t] = 1: \end{aligned}$$

$$= \omega'(h(i[t]/\omega' - \omega) + h(o[t-1]/\omega'))$$

while, when neglecting  $|i[t]/\omega'| < 1/\omega' \ll \omega$ :

$$= \omega'(h(-\omega) + h(o[t-1]/\omega'))$$

while, using previous series formula:

$$= o[t-1] \pm \frac{4}{3} \frac{1}{\omega'^2} + e^{-4\omega} + O\left(\frac{1}{\omega'^4}\right) + O(e^{-8\omega})$$

while, considering only the larger term:

$$\simeq o[t-1] \pm \frac{4}{3} \frac{1}{\omega'^2} = o[0] \pm t \frac{4}{3} \frac{1}{\omega'^2}$$

The memorization is thus impaired by a linear bias only, thanks to the fact that linear unit has been normalized. Because

---

<sup>16</sup>For  $o[0] = 0$  the derivation is straightforward. For  $o[0] = 1$  and  $i[t] = 0$ :

$$\begin{aligned} \epsilon[t] &= 1 - o[t] \\ &= 1 - h(-3/2\omega) - h(\omega(1 - \epsilon(t-1) + 1 - 3/2)) \\ &= 1 - h(-3/2\omega) - h(\omega(-\epsilon(t-1) + 1/2)) \\ &= -h(-3/2\omega) + h(\omega(\epsilon(t-1) - 1/2)) \quad \text{since } h(x) = 1 - h(-x) \end{aligned}$$

with a similar derivation for  $o[0] = 1$  and  $i[t] = 1$ .

$O(e^{-4\omega}) \ll O(\frac{1}{\omega'^2})$ , from the chosen values, the bias due the use of a neuronoid approximation of a programmatoid is negligible with respect to the bias due to the use of a neuronoid approximation of a linear unit.

Derivations are available as **Maple** code<sup>17</sup>.

## 4.2 Multistable mechanisms

The previous developments leads to solutions for several temporal systems, as briefly given now.

### 4.2.1 Bistable mechanisms

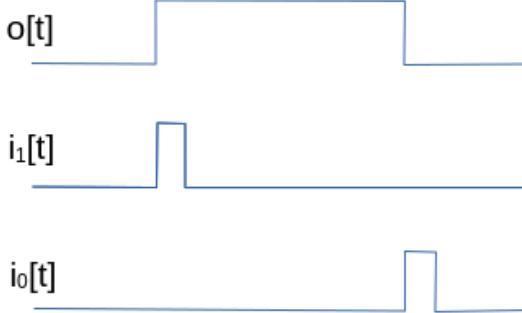


Figure 3: With:

$$\begin{aligned} o[t] &\leftarrow \text{if } o[t] = 0 \text{ and } i_1[t] = 1 \text{ then } 1 \\ &\quad \text{elif } o[t] = 1 \text{ and } i_0[t] = 1 \text{ then } 0 \\ &\quad \text{else } o[t - 1] \end{aligned}$$

a bistable system with two 1 and 0 inputs, is implemented.

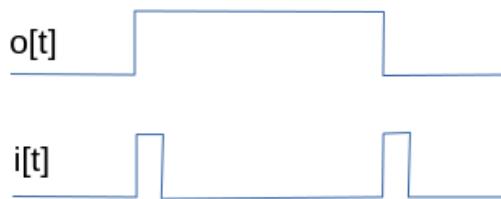


Figure 4: With:

$$\begin{aligned} o[t] &\leftarrow \text{if } o[t] = 0 \text{ and } i_1[t] = 1 \text{ then } 1 \\ &\quad \text{elif } o[t] = 1 \text{ and } i_0[t] = 1 \text{ then } 0 \\ &\quad \text{else } o[t - 1] \\ i_1[t] &\leftarrow \text{if } i_1[t] = 0 \text{ and } i[t] = 1 \text{ then } 1 \\ &\quad \text{elif } o[t] = 0 \text{ then } 0 \\ &\quad \text{else } i_1[t - 1] \\ i_0[t] &\leftarrow \text{if } i_0[t] = 0 \text{ and } i[t] = 0 \text{ then } 1 \\ &\quad \text{elif } o[t] = 1 \text{ then } 0 \\ &\quad \text{else } i_0[t - 1] \end{aligned}$$

a bistable system with one two-way switch is implemented, using internal variables  $i_0[t]$  and  $i_1[t]$ . This corresponds to a frequency divider by 2.

### 4.2.2 Spike generation

### 4.2.3 Delay

Derivations are available as **Maple** code<sup>18</sup>.

### 4.2.4 Oscillation

Derivations are available as **Maple** code<sup>19</sup>.

<sup>17</sup><https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/memorygate.mpl.out.txt>

<sup>18</sup><https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/delayastable.mpl.out.txt>

<sup>19</sup><https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/delayastable.mpl.out.txt>

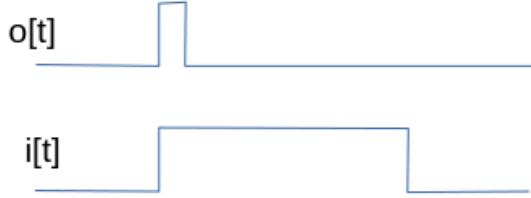


Figure 5: With:

$$\begin{aligned} o[t] &\leftarrow \begin{cases} 1 & \text{if } r[t] = 0 \text{ and } i[t] = 1 \\ 0 & \text{else} \end{cases} \\ r[t] &\leftarrow \begin{cases} 1 & \text{if } o[t] = 1 \\ 0 & \text{elif } i[t] = 0 \\ r[t-1] & \text{else} \end{cases} \end{aligned}$$

a spike generation detecting signal rising, with an internal variable  $r[t]$  registering if already detected.

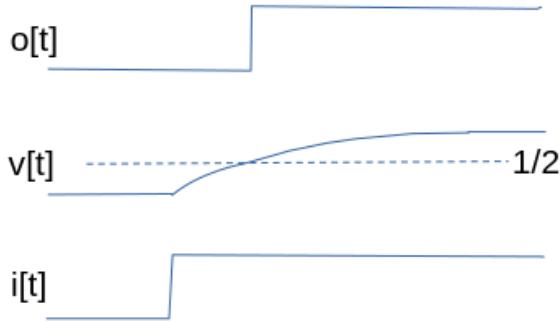


Figure 6: With:

$$\begin{aligned} o[t] &\leftarrow \begin{cases} 1 & \text{if } v[t] > 1/2 \\ 0 & \text{else} \end{cases} \\ v[t] &\leftarrow (1 - \gamma) v[t-1] + \gamma i[t] \end{aligned}$$

a delayed output is implemented. Assuming that  $v[t] = 0$ , when  $i[t] = 1$  and remains at this value, we obtain for the delay  $T$ :

$$T = -\frac{\log(2)}{\log(1-\gamma)} > 0 \Leftrightarrow \gamma = 1 - e^{-\frac{\log(2)}{T}}.$$

Here we assume that  $i[t] = 1, t \in [0, T]$  but it is very easy to avoid this constraint with a bistable mechanism as input.

### 4.3 The explog function

The maximal and the average operators can be combined, e.g.<sup>20</sup>:

$$\begin{aligned} p_{\dagger} &\leftarrow \log \left( \sum_{k \in K} \exp(\mu p[k]) \right) / \mu \\ &= \frac{1}{K} \sum_k p_k + \log(K)/\mu + O(\mu) \\ &= \max_k(p[k]) + o\left(\frac{1}{\mu}\right) \end{aligned}$$

where  $K$  stands for the left or right sensor related indexes,  $p[k]$  stands for the sensor proximity value, and  $\mu > 0$  parameterizes the balance between the max (for large  $\mu$ ) and the average (for small  $\mu$ ) operators, as shown in Fig. 8.

### 4.4 Neuronoid approximation of exp and log

Taking benefit of the fact that sigmoid is convex ad the exponential function and concave as the logarithm function in some range, we can easily design the following approximations, obtained by some manual intuitive choice of the sigmoid combination and numerical adjustment of the parameters.

Combining these two approximations leads to a neuronoid implementation of the explog function.

<sup>20</sup>Let us derive the formula:

- The series at  $\mu \rightarrow 0^+$  is easily obtained from any symbolic calculator writing, e.g.:

```
series(log(sum(exp(mu * p[k])), k = 1..K)) / mu, mu = 0, 2);
```

- There is no obvious series development at  $\mu \rightarrow +\infty$  but, considering  $p[1] \leq p[2] \leq \dots \leq p[K]$ , without loss of generality, i.e., that -for the notation- index's order correspond to decreasing values, we obtain from straightforward algebra:

$$\begin{aligned} \log \left( \sum_{k \in \{1..K\}} \exp(\mu p[k]) \right) / \mu &= p[1] + \rho / \mu \\ \rho &\stackrel{\text{def}}{=} \log \left( 1 + \sum_{k \in \{2..K\}} \exp(-\mu(p[1] - p[k])) \right) \\ \text{with } 0 \leq \rho &\leq \log(1 + (K-1) \exp(-\mu(p[1] - p[2]))) \leq \log(K) \end{aligned}$$

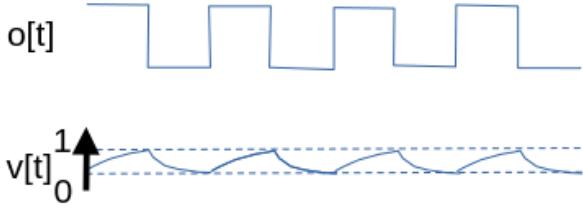


Figure 7: With:

```


$$o[t] \leftarrow \begin{cases} 0 & \text{if } v[t] < 1/3 \\ 1 & \text{if } v[t] > 2/3 \\ o[t-1] & \text{else} \end{cases}$$


$$v[t] \leftarrow (1-\gamma)v[t-1] + \gamma(1-o[t])i[t]$$


```

an binary oscillator is implemented. It is stopped if  $i[t] = 0$  and running for  $i[t] = 1$ . We obtain for the period  $T$ :

$$T = -\frac{2 \log(2)}{\log(1-\gamma)} > 0 \Leftrightarrow \gamma = 1 - 2e^{-\frac{\log(2)}{T}}.$$

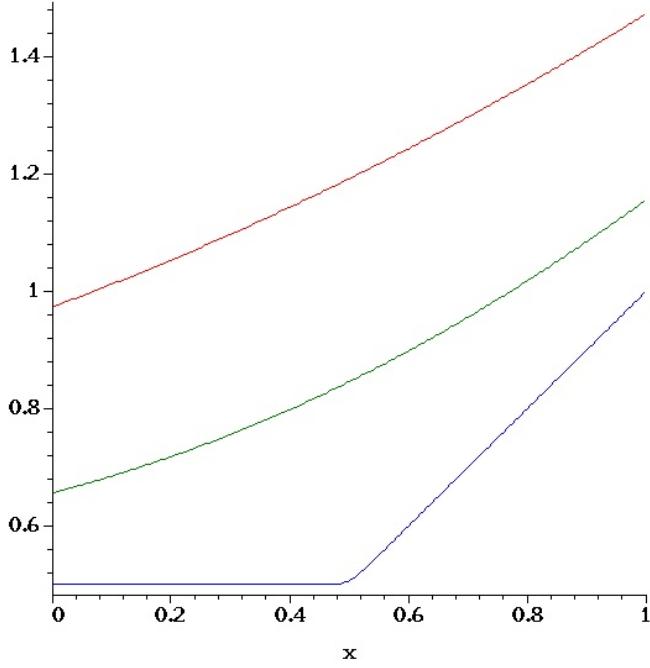


Figure 8: Representation of the exp-log function for  $K = 2$ ,  $p[2] = 1/2$ , with  $p[1] \in [0, 1]$  and  $\mu \in \{1, 2, 100\}$ , from top to bottom. With small  $\mu$  it is closed to linear average, whereas for  $\mu = 100$  it is close to  $\max(x, 1/2)$ .

Derivations of this section are available as **Maple** code<sup>21</sup>.

## 5 Comparing tanh with other non linearities

We have  $h(x) = \frac{1+\tanh(2x)}{2}$  thus based on the **tanh** non linearity. This corresponds most common activation function when deriving a rate-based reduced neural network from bio-physical elements [Cessac and Samuelides, 2007]. This is far from being the only choice, as reviewed in [Szandala, 2020], and we could have considered, for instance  $\bar{h}$  functions of the form:

Name:	Tanh	Erf	Arctan	Softsign
Formula:	$\frac{1}{2} + \frac{\tanh(2x)}{2}$	$\frac{1}{2} + \frac{\arctan(\pi x)}{\pi}$	$\frac{1}{2} + \frac{\operatorname{erf}(\sqrt{\pi}x)}{2}$	$\frac{1}{2} + \frac{x}{1+ 2x }$
Sharpness:	$ h(x)  <  h(x) $	$ h(x)  >  h(x) $	$ h(x)  >  h(x) $	$ h(x)  >  h(x) $
$\mathcal{L}_1 =$	0	$-\frac{\pi \log(2)-2}{2\pi}$	$+\infty$	$+\infty$
$\mathcal{L}_0 \simeq$	0	0.018	0.082	0.81
$\kappa_{max}$	1.53	1.52	2.04	4

while all profiles are normalized and symmetric, i.e.:

so that  $\lim_{\mu \rightarrow +\infty} \rho = 0$  and  $\rho = o(\mu)$ , yielding the expected result.

<sup>21</sup><https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/explog mpl.out.txt>

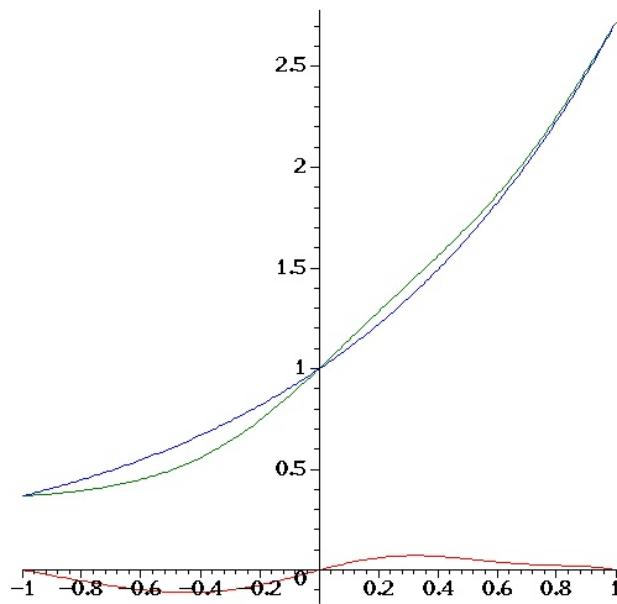


Figure 9: Approximation in the  $[-1, 1]$  interval of the exponential function, in blue, by a sigmoid combination, in green:

$0.345 + 2.34 h(x - 1) + 1.226 h(x) :$   
yielding a  $\mathcal{L}_1$  error of 0.05, the error being drawn in red.

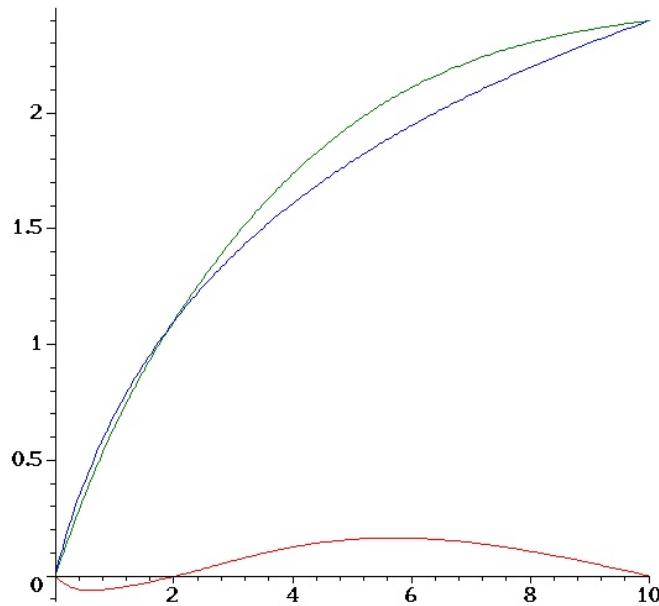


Figure 10: Approximation in the  $[0, 10]$  interval of the logarithm  $\log(1+x)$  function, in blue, by a sigmoid combination, in green:

$-2.48 + 4.42 h(x/10) + 0.54 h(x/2) :$   
yielding a  $\mathcal{L}_1$  error of 0.05, the error being drawn in red.

$$\bar{h}(-\infty) = 0, \bar{h}(0) = 1/2, \bar{h}'(0) = 1, \bar{h}(+\infty) = 1, \bar{h}(x) = 1 - \bar{h}(-x)$$

and are drawn in the Figure 11.

- The Arctan and Softsign profiles are smoother than the Tanh profile, whereas we are looking here for sharper profiles in order to better approximate the step-function. They also have higher maximal curvatures  $\kappa_{max}$  which means that the curvature is more inhomogeneous than for the Tanh profile, and they do not correspond to biologically plausible activation functions [Cessac and Samuelides, 2007], despite the fact they could be of great interest in machine learning [Szandała, 2020].

- The error function<sup>22</sup>, Erf, based profile is very closed numerically from the Tanh profile, with a finite  $\mathcal{L}_1 \simeq -0.028$  small distance magnitude, and a  $\mathcal{L}_0 < 2\%$  small distance magnitude, with similar maximal curvatures  $\kappa_{max}$  up to 1%, and is also quoted as a biologically plausible activation functions [Cessac and Samuelides, 2007]. Though its sharpness is a bit better than the Tanh profile, with a better repartition of the curvature, all algebraic derivations made in this paper would not have as easy as its is now.

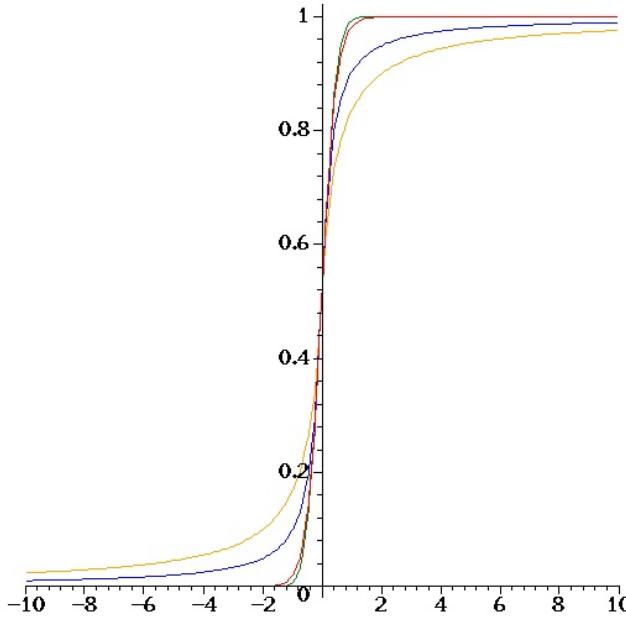


Figure 11: Comparision between the normalized Tanh profile in red, the Erf profile in green, the Arctan profile in blue and the Softsign in orange, i.e., from the sharpest to the smoothest.

Derivations are available as `Maple` code<sup>23</sup>.

## 6 Using the braincraft challenge setup

This documentation allows to use the braincraft challenge for a programmatic implementation, or for a programmatoid/neuronoid implementation using code translator (not yet available).

Here is braincraft challenge original documentation original documentation<sup>24</sup>.

You must be familiar with basic `git` usage and basic `python` programmation.

### 6.1 Installation of the setup

- Connect to <https://github.com> with your login.
- Go to the braincraft challenge<sup>25</sup> and create a new fork:



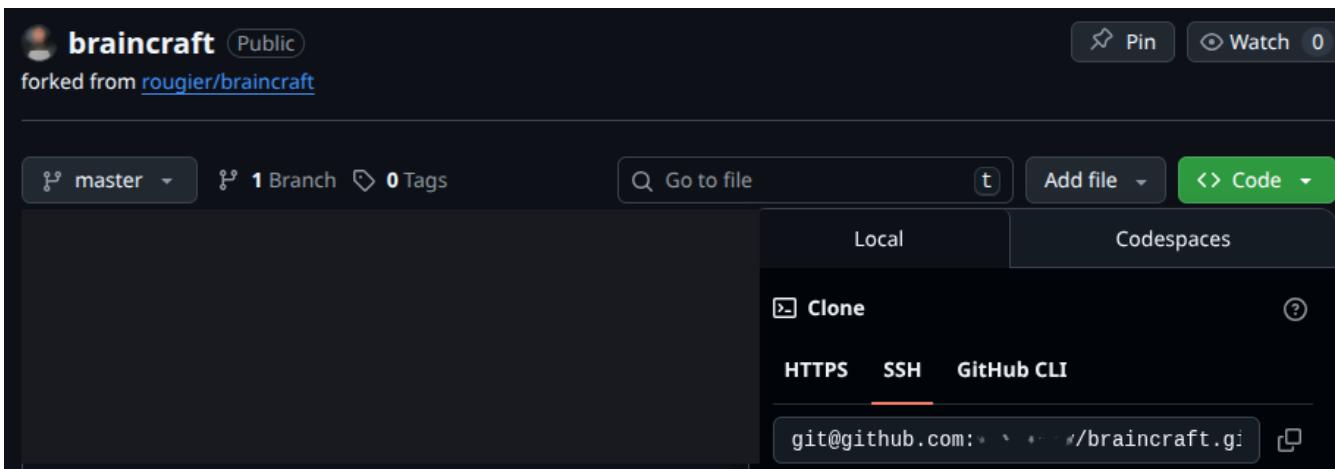
- Download the repository in SSH read/write mode:

<sup>22</sup>[https://en.wikipedia.org/wiki/Error\\_function](https://en.wikipedia.org/wiki/Error_function)

<sup>23</sup><https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/sigmoidatan mpl.out.txt>

<sup>24</sup><https://github.com/rougier/braincraft/blob/master/README.md>

<sup>25</sup><https://github.com/rougier/braincraft>



- In the braincraft local git directory, run `make test`
- + You may have to run `make install`, before.
- + You are advised to use a virtual environment, running `make venv`

## 6.2 Running at the programmatic level

When running at the programmatic level,

- the `next_output_from_network(context)`<sup>26</sup> callback is to be implemented, and
- to be called by the “callback” version of the `evaluate(...)`<sup>27</sup> method.

The `challenge_callback_1.py`<sup>28</sup> source file includes all documented methods.

## 6.3 Running at the programmatoid level

At the programmatoid level, the system is defined by a set of equations ....

For binary variables  $b_i \in \{0, 1\}$ , continuous variables  $v_i \in [-1, 1]$ , while the parameter  $T > 0$ , the following functions, defined previously, are implemented:

<code>If(b_1, v_1, ..., v_0)</code>	Conditional expression with an odd number of arguments.
<code>And(v_1, ...)</code>	Conjunction with a variable number of arguments.
<code>Or(v_1, ...)</code>	Disjunction with a variable number of arguments.
<code>Not(v_1)</code>	Negation.
<code>Bprod(b_1, v_1, ...)</code>	Sum of binary products, with an even number of arguments.
<code>Bistable(i_0, i_1)</code>	Two inputs bistable latch.
<code>Bistable(i_1)</code>	One input bistable latch.
<code>Spikeup(i_1)</code>	Spike generation on input rising front.
<code>Delay(i_1, T)</code>	Delayed output. The delay T is given in number of global clock event.
<code>Oscillator(i_1, T)</code>	Oscillatory output. The period T is given in number of global clock event.

In the present version the `explog` mechanism is still to be implemented.

## References

- [Cessac and Samuelides, 2007] Cessac, B. and Samuelides, M. (2007). From neuron to neural networks dynamics. *The European Physical Journal Special Topics*, 142(1):7–88.
- [Lapicque, 1907] Lapicque, L. (1907). Recherches quantitatives sur l’excitation électrique des nerfs traitée comme une polarisation. *Journal de physiologie et de pathologie générale*, 9:620–635.
- [Szandala, 2020] Szandala, T. (2020). *Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks*.

<sup>26</sup>[https://html-preview.github.io/?url=https://github.com/vthierry/braincraft/blob/master/doc/api/challenge\\_callback.html#next\\_output\\_from\\_network](https://html-preview.github.io/?url=https://github.com/vthierry/braincraft/blob/master/doc/api/challenge_callback.html#next_output_from_network)

<sup>27</sup>[https://html-preview.github.io/?url=https://github.com/vthierry/braincraft/blob/master/doc/api/challenge\\_callback.html#evaluate](https://html-preview.github.io/?url=https://github.com/vthierry/braincraft/blob/master/doc/api/challenge_callback.html#evaluate)

<sup>28</sup>[https://github.com/vthierry/braincraft/blob/master/braincraft/challenge\\_callback.py](https://github.com/vthierry/braincraft/blob/master/braincraft/challenge_callback.py)