Programmatoid and neuronoid computations

February 16, 2026

Contents

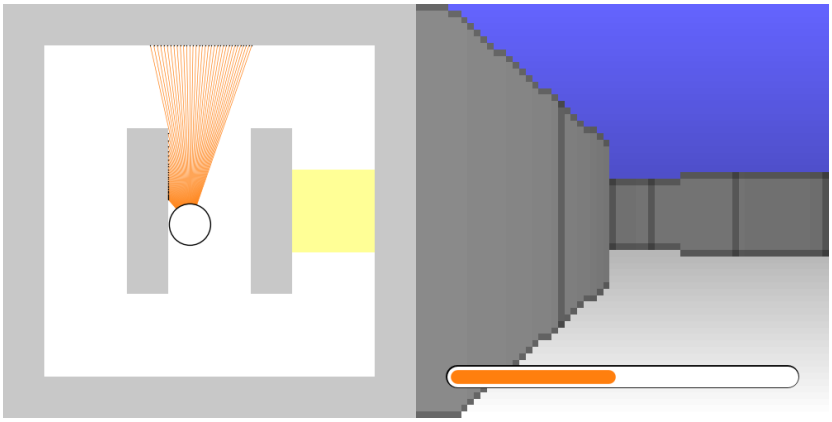
In progress

- neuronoid approximation of the explog
- Commener le compilateur qui
  - liste les variables (y a la fonction indets)
  - ajoute recursivement des variables pour applatir un ensemble d equations
  - substitue les fonctions booleennes etc... en progid puis neuroid
- Creer section sur more mechanisms aec R/S memory etc.....

1 The braincraft challenge

Let us consider the following digital experimental setup, as described in braincraft challenge presentation[1].

1.1 Simplified problem statement



The braincraft challenge[2] bot moves at a constant with sensor inputs and one orientation output, it uses some energy and refill 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.

[1] `https://github.com/rougier/braincraft/blob/master/README.md#introduction`
[2] `https://github.com/rougier/braincraft`

Input variables	
$p_l \in [0_{\text{wall-hit}}, 1_{\text{no-wall}}[$	Leftward proximity, max value of the [0, +30°] range 32 left sensors.
$p_r \in [0_{\text{wall-hit}}, 1_{\text{no-wall}}[$	Rightward proximity, max value of the [-30°, 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 ± 5°.

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 3, 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 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:

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

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

$g_{c\flat} \in [0,1], \flat \in \{1,2\}$ $\quad g_{c\flat}|_{t=0} = 0$ Last and last-before-last cumulative energy increases.
$g_{e\flat} \in [0,1], \flat \in \{1,2\}$ $\quad g_{e\flat}|_{t=0} = 0$ Last and last-before-last energy value.

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

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

Then

$$\begin{aligned} g_{e2} &\leftarrow g_{e1} \\ g_{e1} &\leftarrow g_e \end{aligned}$$

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 ω and ω' are sufficiently large numbers.

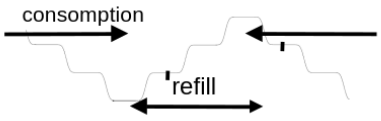


Figure 1: Representation of energy profile when cumulating energy increase starts (first tick), with $g_e > g_{e1} < g_{e2}$, and stops (last tick), with $g_e < g_{e1}$, while during refill $g_e > g_{e1} > g_{e2}$.

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

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

Programmatoid solution :

$$d_o \leftarrow \underbrace{\gamma\,(p_l - p_r)}_{\substack{\text{linear correction to}\\ \text{maintain direction ahead}}} + \underbrace{\alpha\,(t_l - t_r)}_{\substack{\text{quarter- turn}\\ \text{left or right}}}$$

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

where:

w	$\simeq$	$1/4$	Rough estimation of the path-width.
γ	$=$	$\frac{5}{w/2}$	Saturates the correction at 5° if the depth difference is half of the path-width.
α	$=$	90	Saturates the correction at ± 90° to make the quarter-turn, since the linear $\lvert\gamma\,(p_l - p_r)\rvert < 40$ is lower than the quarter-turn term, the latter submut the former.
β	$=$	w	Triggers the quarter-turn if the depth is higher than the path-width.
ω	$=$	10	Transform a boolean product to a step-function threshold.

while:

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

in words: we execute the quater-turn until another wall is detected[3].

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

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

[3] The indenties:

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

are easy to verify using a simple truth table.

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 $q_p = 1$.

$$q_p = \text{if } q_p = 1 \text{ or } \eta > g_e \text{then1 else 0}$$

Programmatoid solution

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

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.
η	$\simeq$	$b\,c/s = 3/20$	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.

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

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

Programmatoid solution

Inverts the direction if to be changed.

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

Registers the inversion until the energy is high enough.

$g_c \quad \leftarrow \quad$ if $g_c = 0$ and $\eta > g_e$ then 1 elif $g_c = 1$ and $2\,\eta < g_e$ then 0 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 .

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.

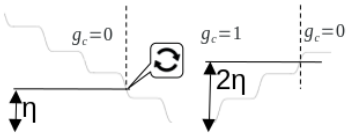


Figure 2: Representation of g_c value at different level of energy with the indication of the direction change.

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

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

Programmatoid solution

Detect the cue, if not yet done, and reset below an energy threshold
$c_{cb} \leftarrow$ if $c_{cb} = 0$ and $c_{cr} = 0$ and ($c_{lb} = 1$ or $c_{rb} = 1$) then1 elif $g_e < \eta/2$ then 0 else c_{cb}
$c_{cr} \leftarrow$ if $c_{cb} = 0$ and $c_{cr} = 0$ and ($c_{lr} = 1$ or $c_{lr} = 1$) then1 elif $g_e < \eta/2$ then 0 else c_{cr}
Set the direction according to the cue
$q_p \leftarrow$ if $c_{lb} = c_{cb}$ or $c_{lr} = c_{cr}$ then 0 elif $c_{rb} = c_{cb}$ or $c_{rr} = c_{cr}$ then 1 else q_c

1.3.5 Task 3: Valued environment decision

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

Programmatoid solution

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

2 Programmatoid computation

We name "programmatoid" computation the conception of an input-output straight-line program[4] implementing an operator on numerical value expressions using an LN-system with the step-function as non-linearity, this writing:

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

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 outpout corresponding to inputs, while

[4] https://en.wikipedia.org/wiki/Straight-line_program

$w_{nm}, n \in \{1, N\}, m \in \{0, M\}$ 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:

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

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[5], considering $true$ as 1 and $false$ as 0:

$H(v_1 > v_2)$	$H(v_1 \leq v_2)$	$H(v_1 = v_2)$	
$H(2\,H(v_1 - v_2) - 3/2)$	$H(H(v_1 - v_2))$	$H(H(v_1 - v_2))+$ $H(H(v_2 - v_1)) - 1$	with $H(0) = 1/2$
$H(v_1 - v_2)$	$1 - H(v_1 - v_2)$	$1 - H(v_1 - v_2)$ $+H(v_2 - v_1)$	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 $b_n \in \{0_{false}, 1_{true}\}, n \in \{1, N\}$ we have the obvious[6] correspondence:

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

[5] 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
$2\,H(v_1 - v_2) - 3/2$	1/2	$-1/2$	$-3/2$
$H(2\,H(v_1 - v_2) - 3/2)$	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.

[6] Easy to verify with, e..g., a truth table, and by induction for the generalized formula.

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:

$$\begin{aligned}v &\rightarrow \text{ if } b_1 \text{ then } v_1 \text{ else } v_0 \\ &= (1-b_1)\, v_0 + b_1\, v_1\end{aligned}$$

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

$$\begin{aligned}&= H(v_0 + (1-b_1) - 3/2) + H(v_1 + b_1 - 3/2) \\ &= H(v_0 - b_1 - 1/2) + H(v_1 + b_1 - 3/2) \\ &= H(H(v_0 - b_1 - 1/2) + H(v_1 + b_1 - 3/2))\end{aligned}$$

as easy to verify, using for instance a truth table. Here:
- products with $(1-b_1)$ and b_1 behaves as switches between v_o and v_1
- 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 $v_n, n \in \{0,N\}$: [7]:

[7] 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 dediced from the second line, transforming the binary products on the corresponding programmatoid while:

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

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.

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

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

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

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 of 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[8] simplest biological neuron or neuron small ensemble inspired by mean-field modelisation[9] of the Hodgkin–Huxley neuronal axon model[10].

$$
\begin{array}{c}
\tau\frac{\partial v_i}{\partial t}(t)+v_i(t)=z_i(t), \quad z_i(t) \stackrel{\text{def}}{=} h\left(\sum_j w_{ij}\,v_j(t)+w_{i0}\right), \\
h(x) \stackrel{\text{def}}{=} \frac{1}{1+\exp(-4\,x)} \quad = \frac{1+\tanh(2\,x)}{2},
\end{array}
$$

where v_i is the membrane potential, so that:

[8] https://en.wikipedia.org/wiki/Biological_neuron_model#Relation_between_artificial_and_biological_neuron_models

[9] https://inria.hal.science/cel-01095603v1

[10] https://en.wikipedia.org/wiki/Hodgkin-Huxley_model

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

and the corresponding discrete approximation using an Euler schema writes:

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

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

$$\gamma \stackrel{\text{def}}{=} 1 - \exp(-1/\tau) \Leftrightarrow \tau = 1/\log(1/(1-\gamma)),$$

while

$$\lim_{\gamma \to 0} \tau = +\infty, \lim_{\gamma \to 1} \tau = 0.$$

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[11], 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.:

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

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:

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

while:

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

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 neglictible as soon as, say, $\omega \geq 10$:

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

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

[11] `https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/neuronoid.mpl.out.txt`

Derivations are available `Maple` code[12].

3.3 Approximation of the identify function

We also use a sigmoid to approximate the indentity function defining:

$$l_{\omega'}(x) \stackrel{\text{def}}{=} \omega' \left(h(x/\omega') - 1/2\right), \text{ with an error } \epsilon_{l,\omega'}(x) \stackrel{\text{def}}{=} x - l_{\omega'}(x)$$

so that, the $\mathcal{L}_1$ error magnitude on $[-M, +M]$:

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

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

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

Assuming values are normalized, in the $[-1, +1]$ we obtain, for the $\mathcal{L}_0$ error magnitude:

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

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

Derivations are available `Maple` code[13].

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[14]:

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

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.

[12] `https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/mollificatio.mpl.out.txt`

[13] `https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/linearapproximation.mpl.out.txt`

[14] 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 $O\left(e^{-4\,w}\right)$ 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 $O\left(\frac{1}{\omega'}\right)$ as derived previously.

4.1 The explog function

The maximal and the average operators can be combined, e.g.[15]:

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

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 μ) and the average (for small μ) operators, as shown in Fig. 3.

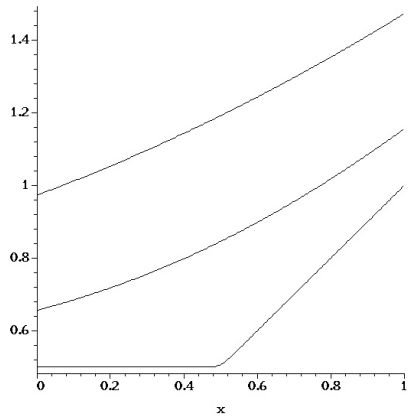


Figure 3: Representation of the explog function for $K = 2$, $p[2] = 1/2$, with $p[1] \in [0, 1]$ and $\mu \in \{1, 2, 100\}$, from top to bottom. With small μ it is closed to linear average, whereas for $\mu = 100$ it is close to $\max(x, 1/2)$.

Derivations are available `Maple` code[16].

4.2 RS input/output gate

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

$$o(t) \quad \leftarrow \quad \text{if } i_l(t) = 1 \text{ then } o(t-1) \text{ else } i(t)$$

[15] Let us derive the formula:
- The series at $\mu \to 0^+$ 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 $\mu \to +\infty$ but, considering $p[1] \le p[2] \le \cdots \le p[K]$, without loss of generality, i.e., that -for the notation- index's order correspond to decreasing values, we obtain from straightforward algebra:

$$\begin{array}{c} \log\left(\sum_{k\in\{1\cdots K\}}\exp(\mu\, p[k])\right)/\mu = p[1] + \rho/\mu \\ \rho \stackrel{\text{def}}{=} \log\left(1 + \sum_{k\in\{2\cdots K\}}\exp(-\mu\,(p[1] - p[k]))\right) \\ \text{with } 0 \le \rho \le \log\left(1 + (K-1)\exp(-\mu\,(p[1] - p[2]))\right) \le \log(K) \end{array}$$

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

[16] `https://raw.githubusercontent.com/vthierry/braincraft/master/doc/tex/linearapproximation.mpl.out.txt`

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 $i_l(t) = 1$:

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

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

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

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

5 Using the braincraft challenge setup

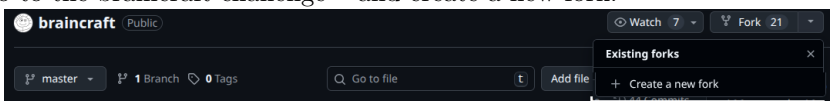
This documentation allows to use the braincraft challenge for a programmatic implementation, or for a programmatoid/neuronoid implementation using code translator (not yet available).

Here is braincraft challenge original documentation original documentation[17].

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

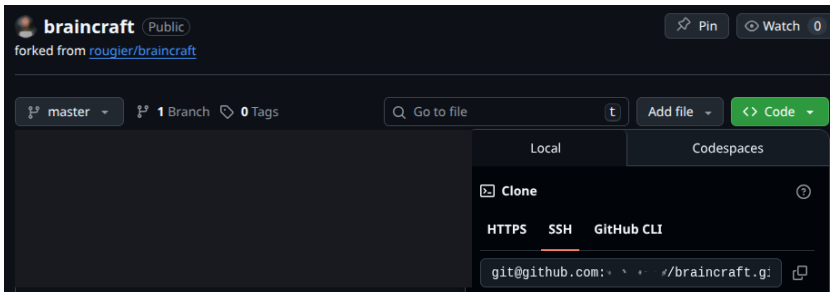
5.1 Installation of the setup

- Connect to `https://github.com` with your login.
- Go to the braincraft challenge[18] and create a new fork:



- Download the repository in SSH read/write mode:

[17] `https://github.com/rougier/braincraft/blob/master/README.md`
[18] `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`

5.2 Running at the programmatic level

When running at the programmatic level,
- the `next_output_from_network(context)`[19] callback is to be implemented, and
- to be called by the "callback" version of the `evaluate(···)`[20] method.

The `challenge_callback_1.py`[21] source file includes all documented methods.

References

[Lapicque, 1907] Lapicque, L. (1907). Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. *Journal de physiologie et de pathologie générale*, 9:620–635.

[19] `https://html-preview.github.io/?url=https://github.com/vthierry/braincraft/blob/master/doc/api/challenge_callback.html#next_output_from_network`
[20] `https://html-preview.github.io/?url=https://github.com/vthierry/braincraft/blob/master/doc/api/challenge_callback.html#evaluate`
[21] `https://github.com/vthierry/braincraft/blob/master/braincraft/challenge_callback.py`