DANA: Distributed (asynchronous) Numerical and Adaptive modelling framework

Nicolas P. Rougier, Jérémy Fix

DANA: Distributed (asynchronous) Numerical and Adaptive modelling framework

Nicolas Rougier*[1] and Jérémy Fix[2]

[1]INRIA Bordeaux - Sud Ouest, 351, Cours de la Libération, 33405 Talence Cedex, France
[2]IMS, SUPELEC, 2 rue Edouard Belin, F-57070 Metz, France

July 18, 2012

Abstract

DANA is a python framework (http://dana.loria.fr) whose computational paradigm is grounded on the notion of a unit that is essentially a set of time dependent values varying under the influence of other units via adaptive weighted connections. The evolution of a unit's value are defined by a set of differential equations expressed in standard mathematical notation which greatly ease their definition. The units are organized into groups that form a model. Each unit can be connected to any other unit (including itself) using a weighted connection. The DANA framework offers a set of core objects needed to design and run such models. The modeler only has to define the equations of a unit as well as the equations governing the training of the connections. The simulation is completely transparent to the modeler and is handled by DANA. This allows DANA to be used for a wide range of numerical and distributed models as long as they fit the proposed framework (e.g. cellular automata, reaction-diffusion system, decentralized neural networks, recurrent neural networks, kernel-based image processing, etc.).

Keywords: python, software, computational neuroscience, distributed, numerical, adaptive, mean-rate, simulation, computation

Introduction

The original artificial neural network (ANN) paradigm describes a fine grain computational system, that is distributed among a population of elementary processing units analogous to the processing taking place within the brain. Those units, the so called artificial neurons, communicate with each other through weighted connections. Such a system can be described by a graph in which the vertices are the processing components and the edges are the communication channels allowing the transfer of information within the system. The different ANN architectures proposed by the community can be distinguished by the type of units (logical, rate, spikes, etc.) and topology (feed-forward, feedback, recurrent, etc.) used. They can be ordered graphs where some units are systematically evaluated before others (e.g. perceptron or multi-layer perceptron (MLP)) or they can be recurrent networks where no *a priori* evaluation order can be defined. The system may be endowed with a central supervisor able to *look* at the whole graph in order to take some global decision depending on meta-conditions (e.g. winner takes all in the Kohonen self organizing maps (SOM) or back-propagation learning in the MLP) or can be fully distributed, relying only on local computations. This description is indeed loose enough to allow virtually any kind of network, from a very simple network of two neurons that is able to ride a bike[1] up to the blue brain project[2] gathering thousands of highly detailed models of the neocortical column. Obviously, if these two models fall into the generic neural network paradigm, they are quite different both in their essence and their respective goals. The dual neuron network is built around a concept of a neuron that is very

*Corresponding author: Nicolas.Rougier@inria.fr
[1]http://www.paradise.caltech.edu/~cook/
[2]http://bluebrain.epfl.ch/

similar to a logical unit able to carry out quite complex computations while in the case of the blue brain project, the concept of a neuron is directly related to precise anatomical, electrical and morphological data on the composition, density and distribution of the different cortical cells.

Motivations

Between these two extremes, there exist several mid-level approaches which promote population models for the study of complex functions and structures and there are generally dedicated software such as Emergent (Aisa et al., 2008) (previously PDP++ (O'Reilly and Y.Munakata, 2000)), Miind (de Kamps et al., 2008), Topographica (Bednar, 2008), NEST initiative (Gewaltig and Diesmann, 2007), etc. These simulators are mostly based on high-level interfaces where a user can build a model by using a set of predefined objects such as neurons, connections, training algorithms, etc. and also offer the possibility of deriving new models from these base objects. While those high-level approaches yield several well-known advantages, the main problem in our view is the lack of control of what the model is actually computing since the underlying computations may be buried very deep in the software. Such high level interfaces can be thus delicate to use because they offer no guarantee of strongly enforcing an actual parallel, distributed, numerical and adaptive approach. Let us for example consider the winner-takes-all (WTA) algorithm which is usually implemented as the $maximum()$ function over the activities of a given layer such that, only the most active unit remains active at the end of the simulation. This implementation requires an examination of all units at once and modifying parameters, such as synpatic efficacy according to the result of the function. There is then a question of who is examining and acting; the model or the modelling software?

This means that part of the properties of the system may indeed be due to the supervisor, without specifying how the supervisor itself is controlled. This also means that one would not be able to reproduce this network using, for instance, only hardware artificial neurons linked together without introducing an ad-hoc surrounding architecture for the coordination of the network. A question arises here as to what extent such artifacts could impact results and claims. We will not answer in the general case; however, in the context of the study of emergence where we seek to understand properties of a population of units whose behavior depend only on local interactions between the units, we think it is critical to both strongly constrain the modeling framework, in order to be able to guarantee results, and to still offer the modeler the freedom to design its own model as is the case, for example, in the Brian simulator.

Brian limitations

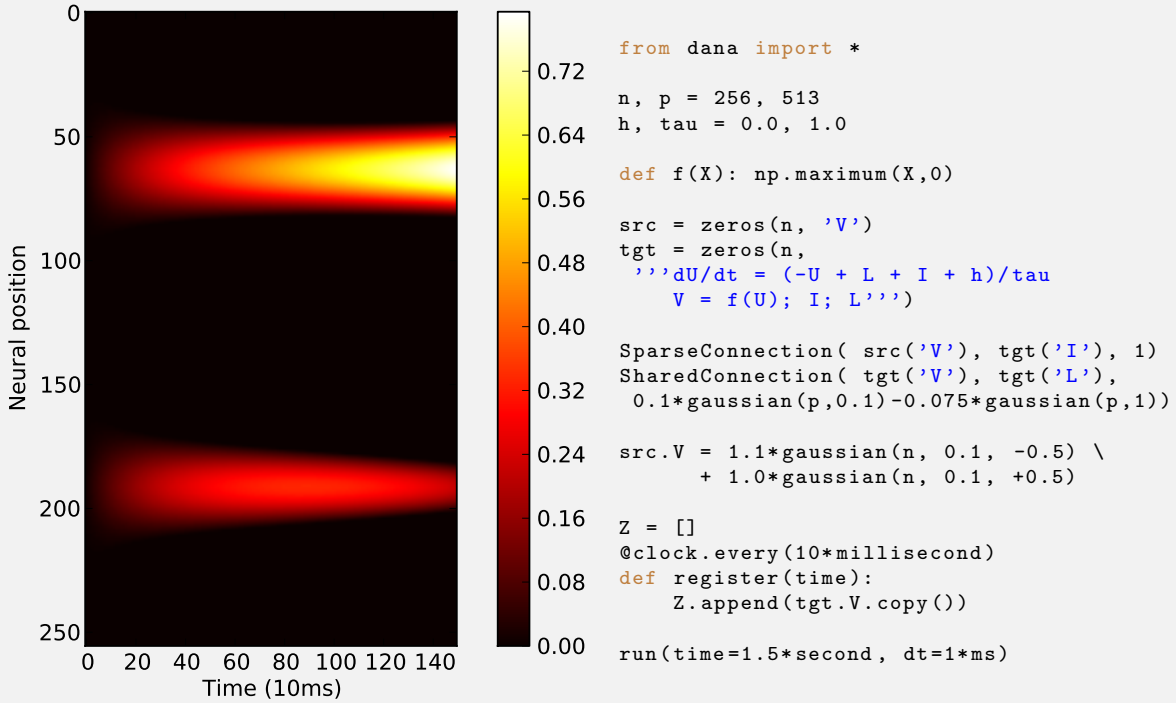
The Brian simulator (Goodman and Brette, 2008) has become very popular among neuroscientists in just a few years and it has been ranked 2^{nd} most popular neural simulator in a recent survey (Hanke and Halchenko, 2011), just after the well known Neuron software (Carnevale and Hines, 2006). The reason behind this success is certainly the intuitive and highly flexible interface offered by Brian as well as efficient vectorized computations that make Brian quite competitive with regular C code for large networks. Designing a non-standard spiking neuron model using Brian is also straightforward as you just need to provide the model's equations in the usual mathematical notation. In just a few lines of codes, you can easily and very quickly simulate a whole model. However, Brian has been designed specifically for spiking neuron models with dedicated optimization for the sparse (in time) transfer of information from one neuron to the other. If one wants to model a mean-rate neuron (or any model that relies on continuous quantities), it is still possible to tweak Brian to simulate it; however, doing this prevents the model from benefiting from Brian's optimizations and makes the overall simulation quite slow since it requires a dedicated network operation (see appendix).

Reclaiming control

One of the goals of the DANA framework is to give the user full control over its model such that he knows what is actually computed at every timestep. We think it is critical to frame what is precisely computed by the model in order to avoid any modeling artifact such as a central supervisor or homunculus. One way to achieve this is to use a constrained computational framework that guarantees the model to be consistent regarding a definition of what we think are distributed, asynchronous, numerical and adaptive

computations. The DANA framework (http://dana.loria.fr) aims to provide such a definition in the form of a set of computational principles (namely, distributed, asynchronous, numerical and adaptive) and to allow for the rapid design of models in an easy and intuitive way. These constraints may change radically the way models are designed. For example, figure 1 shows the implementation of a dynamic neural field, which under some conditions leads to the emergence of a localized bubble of activity. This can be interpreted as a k-WTA. However, this implementation does not require an implicit supervisor and the final equilibirum state is the result of unit computations only.

$$\tau \frac{\partial U(\mathbf{x},t)}{\partial t} = -U(\mathbf{x},t) + \underbrace{\int_{-\infty}^{+\infty} W(|\mathbf{x}-\mathbf{y}|)\overbrace{f(U(\mathbf{y},t))}^{V(\mathbf{y},t)}\,d\mathbf{y}}_{L(\mathbf{x},t)} + I(\mathbf{x},t) + h \quad (1)$$




```
from dana import *

n, p = 256, 513
h, tau = 0.0, 1.0

def f(X): np.maximum(X,0)

src = zeros(n, 'V')
tgt = zeros(n,
 '''dU/dt = (-U + L + I + h)/tau
    V = f(U); I; L''')

SparseConnection( src('V'), tgt('I'), 1)
SharedConnection( tgt('V'), tgt('L'),
 0.1*gaussian(p,0.1)-0.075*gaussian(p,1))

src.V = 1.1*gaussian(n, 0.1, -0.5) \
      + 1.0*gaussian(n, 0.1, +0.5)

Z = []
@clock.every(10*millisecond)
def register(time):
    Z.append(tgt.V.copy())

run(time=1.5*second, dt=1*ms)
```

Figure 1: A one dimensional neural field in DANA, with code on the right, model equation on the top and field activity evolution over time at the left. The differential equation describes the spatio-temporal evolution of a coarse grained variable (such as synaptic or firing rate value) for a spatially continuous population of neurons. V is the membrane potential, W a neighborhood function, f the transfer function of a single neuron (mapping the membrane potentials to the firing rates), τ a time constant, I the input and h the resting potential. In the simulation script, L gathers the lateral contributions through the weights W. The model has been simulated for 1.5 second using a time-step of 1 millisecond and activity has been registered every 10 milliseconds.

Distributed, Numerical and Adaptive computing

The computational paradigm supporting the DANA framework is grounded on the notion of a unit that is essentially a set of time dependent values varying under the influence of other units via trainable weighted connections (figure 2). The evolution of the units value are defined by a set of differential equations expressed in standard mathematical notation. The units are organized into groups that form a network. Each unit can be connected to any other unit (including itself) using a weighted connection. The DANA framework offers a set of core objects needed to design and run such networks. The modeler

who uses DANA only has to specify the equations of his model (computations, training), in usual mathematical notation, and does not have to handle how the model is simulated which greatly facilitates the definition and testing of a model.

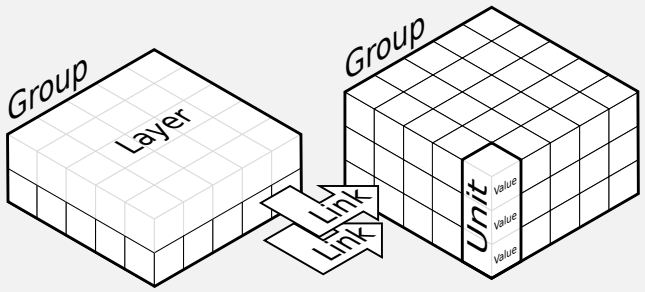


Figure 2: DANA framework. A unit is a set of one to several values (V_i). A group is a set of one to several homogeneous units. A layer is a subset of a group restricted to a unique value V_i. A layer is a group. A link is a weighted connection between a source group to a target group. A group can be linked to any other group including itself.

The framework has been implemented as a Python[3] library using fast vectorized computation brought by the NumPy[4] library. In the next paragraphs, we illustrate the main concepts of DANA through the simulation of the Bienenstock-Cooper-Munro (BCM, (Bienenstock et al., 1982)) learning rule. This learning rule describes a model of synaptic plasticity in the visual cortex that is able to account for the selectivity of the visual neurons. It relies on a sliding threshold ensuring the stability of the rule as shown in equations (2) to (4) where $\mathbf{d}$ is the input, $\mathbf{m}$ the synaptic connections with the input, $\mathbf{c}$ the post-synaptic activity, $\boldsymbol{\theta}$ the adaptive modification threshold and η the learning rate. τ and $\bar{\tau}$ are time constants. Defining and simulating this learning rule is straightforward as shown on figure 3.

Computing

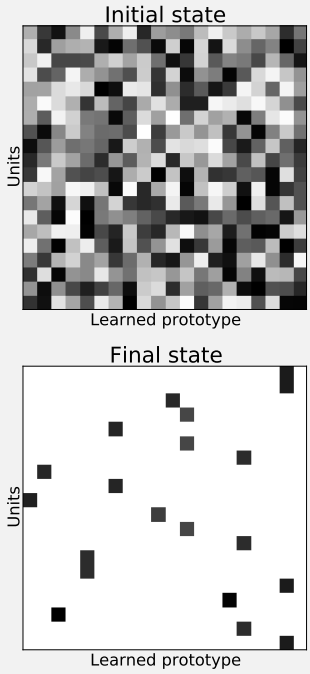
We explained earlier that a unit is the elementary processing element and a group is a set of one to several homogeneous units. However, there is no such explicit concept of a unit within the proposed implementation. In DANA, we define groups as a set of units sharing the same update rules. For the BCM example, we have two groups; one that holds the current stimulus (`src`) and one that computes the actual activity and the sliding threshold (`tgt`):

```
src = np.zeros( n )
tgt = zeros( n, '''dC/dt = (I-C)*(1/tau)
                   dT/dt = (C**2-T)*(1/tau_)
                   I''' )
```

The `src` group is defined as a regular numpy array while the `tgt` group is defined as a DANA group with three values. The evolution of the values `C` and `T` is governed by differential equations 2 and 3. The value `I` holds the output of the weighted connections from the `src` group. In DANA, group values can be of three different types:

- A *declaration* specifies the existence of a value in a group. It can be used to receive the output of projections from one group to another or the result of some local computations,
- An *equation* of the form $y = f(y, x, ...)$ describes how a given variable is computed when the equation is called. The name of the variable is given on the left side of the equation and the update function is represented on the right side of the equation,
- A *differential equation* represents a first order ordinary differential equation of the form $dy/dt = f(y, x, t, ...)$ and describes how a given variable is updated when the equation is called (using Euler,

[3] http://python.org/
[4] http://scipy.numpy.org/



$$\tau \dot{\mathbf{c}} = \mathbf{m.d} - \mathbf{c} \tag{2}$$

$$\bar{\tau} \dot{\boldsymbol{\theta}} = \mathbf{c}^2 - \boldsymbol{\theta}, \tag{3}$$

$$\dot{\mathbf{w}} = \eta \mathbf{c}(\mathbf{c} - \boldsymbol{\theta})\mathbf{d} \tag{4}$$

```
from random import choice
from dana import *

n = 10
tau, tau_, eta = 1.0, 0.1, 0.01

stims = np.identity( n )
src = np.zeros( n )
tgt = zeros( n, '''dC/dt = (I-C)*(1/tau)
                   dT/dt = (C**2-T)*(1/tau_)
                   I '''')
DenseConnection( src, tgt('I'),
      rnd.uniform( 0, 1, (n,n) )
      'dW/dt = eta*post.C*(post.C-post.T)*pre')

@before(clock.tick)
def set_stimulus(*args):
    src[:] = choice(stims)

run(n=10000)
```

Figure 3: Implementation of the BCM learning rule. Starting from an initial random weight matrix, units learn to become responsive to a unique stimulus and to remain silent for the others.

Runge-Kutta second and fourth order or exponential Euler). The name of the variable is given by the left side of the equation and the update function is represented by the right side of the equation. Considering only first order differential equations is not restrictive as any higher order equation can be expressed by a system of first order differential equations.

Connecting

The connections are directly specified between groups. Considering a source group of size n and a target group of size m, a connection is a matrix $(L_{ij})_{1 \leq i \leq n, 1 \leq j \leq m}$ describing individual connections between a source unit i and a target unit j. The connection output is computed as the matrix product of the source group and the connection matrix which corresponds to the classic weighted sum. This output is made available to the target group for the definition of value equations. The definition of a connection involves the involved source and target group values.

A group does not have a topology *per se* and the shape is mainly used to access group elements or display it in a terminal or a figure. If the given weight matrix shape is different from $m \times n$, the kernel is interpreted as a prototype kernel for a single target unit that is to be used for every other unit, as illustrated on figure 4. In this case, DANA internally builds a new matrix such that any target unit receives the output of the prototype kernel multiplied by the source unit and its immediate neighborhood (that spans the kernel size) as illustrated on figure 4.

Depending on the type of connections, DANA offers various optimizations:

- *Shared connection*: when the same kernel is shared by all the units from the target group, the output can be computed using a convolution or Fast Fourier Transform (FFT),
- *Sparse connection*: when the connections are sparse, the connection matrix is stored in a sparse array,

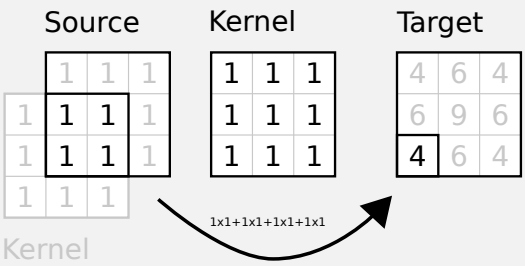


Figure 4: The figure shows how the output of target unit at position (2,0) is computed according to the prototype kernel. First, the shape of the target is used to assign a set of normalized coordinates to the unit and these normalized coordinates are used to find the corresponding source unit. The prototype kernel is then centered on this unit and an element wise multiplication is performed where kernel and source overlaps. They are then summed up and stored at the target unit position within the target group.

- *Dense connection*: the most generic non optimized connection.

User can also decide to define its own type of connection by sub-classing the `Connection` class and providing an `output()` method. For the BCM example, the connection is made dense because each output unit is connected to every input unit with their own (i.e. not shared) weights.

Learning

A differential equation can be associated to *sparse* and *dense* connections. It defines the temporal evolution of the weights as a function of any of the target group values (or post-synaptic group, identified as `post`), source group values (or pre-synaptic group, identified as `pre`) and connection output value identified by its name. For example, in case of the BCM learning rule, the connection reads:

```
DenseConnection( src, tgt('I'),
                 rnd.uniform( 0, 1, (n,n) )
                 'dW/dt = eta*post.C*(post.C-post.T)*pre' )
```

This differential form of the learning rule allows to write various standard learning rules such as the Hebbian learning rule (`pre*post`). A broader range of learning rules can also be implemented. As an example, one can consider the general definition of Hebb like learning rules given in Gerstner (2002) and expressed by equation 5.

$$\frac{dw_{ij}}{dt} = F(w_{ij}, \nu_i, \nu_j) \tag{5}$$

where ν_j and ν_j are respectively the pre- and post- synaptic activities and w_{ij} is the connection strength between these two neurons. A Taylor expansion at $\nu_i = \nu_j = 0$ of equation (5) leads to a general expression for Hebb like learning rules (see eq. (6)).

$$\frac{dw_{ij}}{dt} = \sum_{k=0}^{\infty}\sum_{l=0}^{\infty} c_{kl}\nu_i^k\nu_j^l \tag{6}$$

Various classical learning rules, such as Hebb, Oja (Oja, 1982) or BCM (Bienenstock et al., 1982) can all be expressed within such a formulation and easily defined within DANA. Finally, following the guidelines that no supervisor should coordinate and define when learning or computation takes place, the learning rules are evaluated unconditionally at each time step. Learning could certainly be modulated by some signals gating when some synapses should be potentiated or depressed but these modulatory signals must be explicitly specified within the model. An example of these in Neuroscience is the modulatory influence of dopamine on the plasticity of corticostriatal synapses in the basal ganglia (Reynolds and Wickens, 2002).

Simulating

Once the model has been fully specified, it is possible to run it for a specified period of time using:

```
run( time=t, dt=Δt),
```

This running loop will evaluate the connections, update the group values through their equations and update the weights through their equations. As the model is running, it is also possible to trigger some events based on a clock (fig. 5). The events can be triggered once, after a certain time has elapsed, on a regular basis for the entire duration of the simulation or starting and ending at specific times, and finally before or after the tick of the internal clock. This way, it is possible to record the values within a model or to provide a new input at regular time intervals.

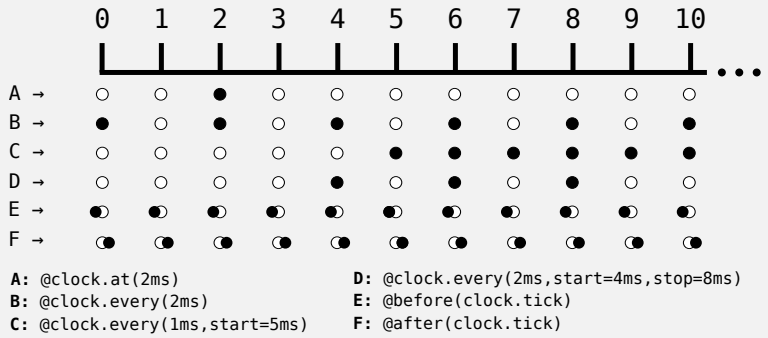


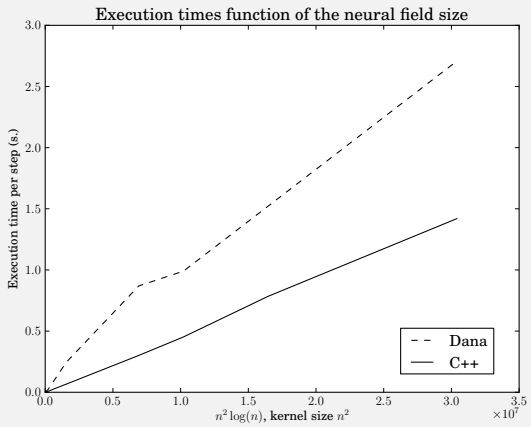
Figure 5: Time management. Unique, regular and time-bounded events can be plugged into the simulation through the use of decorated functions relative to the clock. **A** Unique event at 2ms. **B** Regular event every 2ms. **C** Regular event every 1ms, starts at 5ms and never stops. **D** Regular event every 2ms, starts at 4ms, stops at 8ms. **E** Regular event just before clock tick. **E** Regular event just after clock tick.

Performances

In order to evaluate the performances of DANA, we compared the execution times of a script written in DANA and its dedicated C++ counterpart. As a first experiment, we consider a 2D neural field with difference of Gaussian lateral connections. With appropriately defined lateral connections and a random input exciting the field, a localized bump of activity emerges. In this case, as the weights are fixed and not subject to learning, we use *shared* connections involving the FFT to update the lateral contribution. The DANA script to simulate this example and the C++ and DANA execution times are shown in figure 6. With no surprise, the dedicated C++ script is faster than the DANA implementation. However, DANA is only two times slower than the dedicated script while offering much more flexibility and being more generic than the C++ counterpart (please note that due to the use of the FFT, the execution times are in $O(n^2 \log(n))$ with fields of size $n \times n$). In this simulation, all the units of the group were connected. As a second example, we consider the reaction diffusion Gray-Scott model Pearson (1993) (see fig. 12). In this model, the kernel size is 3×3. Therefore, we make use of the *Sparse Connections* of DANA since using the FFT is not optimal for such small kernels. The DANA simulation script and an illustration of the model are shown in figure 12. The execution time of a single step in both DANA and a dedicated C++ script is given on figure 7. Again, the C++ dedicated script is faster than its DANA translation. In summary, in both situations a C++ dedicated script runs faster than the equivalent model written in DANA. However, this cost remains reasonable when we consider the flexibility that DANA offers for defining the models (and the overhead it induces).

Case studies

We report here several examples of models taken from the literature in order to illustrate the versatility of the DANA framework. Each of these models fit the proposed framework at various degrees and the




```
from dana import *

n, p = 256, 513
h, tau = 0.0, 1.0/0.9

def f(X): np.minimum(np.maximum(X,0),1)
focus = Group((n,n),
              '''dV/dt = (-V + L + I)/tau
                     U = f(V); I; L''')
SharedConnection( focus('U'), focus('L'),
                  0.062*gaussian((p,p),0.15)
                 -0.059*gaussian((p,p),1),
                  fft=True, toric=True)
focus['I'] = 1
focus['I'] += np.random.random((n,n))
run(time=100*second, dt=100*ms)
```

Figure 6: Execution time comparison between a DANA and C++ implementation of a two dimensional neural field. The execution time is averaged over 1000 steps.


```
Du, Dv, F, k = 0.16, 0.08, 0.020, 0.055
Z = Group( (128,128),
    """du/dt = Du*Lu - Z + F*(1-U) : float
       dv/dt = Dv*Lv + Z - (F+k)*V : float
       U = np.maximum(u,0)         : float
       V = np.maximum(v,0)         : float
       Z = U*V*V                   : float
       Lu; Lv """ )
K = np.array([[np.NaN,  1., np.NaN],
              [  1.,   -4.,   1.  ],
              [np.NaN,  1., np.NaN]])
SparseConnection(Z('U'),Z('Lu'),K,toric=True)
SparseConnection(Z('V'),Z('Lv'),K,toric=True)

run(n=1000)
```

Figure 7: Execution time comparison between a DANA and C++ implementation for the Gray-Scott reaction diffusion model. The time to execute a single step is averaged over 1000 steps.

resulting script is generally straightforward from the mathematical description.

Saccadic exploration of a visual environment Fix et al. (2011, 2006)

In (Fix et al., 2011, 2006) we proposed a distributed mechanism for exploring a visual environment with saccadic eye movements (see fig. 8). In this section and in the simulation script provided online, we detail how this mechanism can be implemented within the DANA framework. The experimental setup consists of three identical targets that have to be scanned successively with saccadic eye movements. The targets are identical (same shape, same visual features), a target is only characterized by its spatial position.

The experimental studies on the monkey brain areas involved in the control of saccadic eye movements indicate that most of these areas encode a saccadic target in an eye-centered frame of reference (such as in the Lateral Intraparietal area (LIP) and the Frontal Eye Fields (FEF)). In addition, some areas such as the dorsolateral prefrontal cortex (dlPFC) exhibit sustained activities when a spatial position, relevant for the task at hand, has to be memorized. These areas are part of the saccadic system. They project on the basal ganglia with which they form partially segregated loops. The loop with FEF is proposed to be involved in action selection. The loop involving dlPFC may be involved in spatial memory updating in particular because dlPFC has sustained activities when a spatial information has to be memorized and because it interacts with the mediodorsal nucleus of thalamus which has been shown recently to be the target of a corollary discharge signal sent from the superior colliculus (SC).

In agreement with the above mentioned biological data, the model presented in (Fix et al., 2011, 2006) considers that the spatial position of the targets are encoded in an eye centered frame of reference. In order to successfully perform the saccadic scanpath, we further proposed that the position of the previously focused targets are stored in an eye-centered frame of reference, and updated by anticipation with each saccadic eye movement. This update involves sigma-pi connections (*i.e.* weighed sum of products). These connections are not provided by DANA. However, as shown on figure 8, it is sufficient to subclass the *Connection* type and to overload the *output* method in order to define custom connectivity patterns.

A computational model of action selection Gurney et al. (2001b)

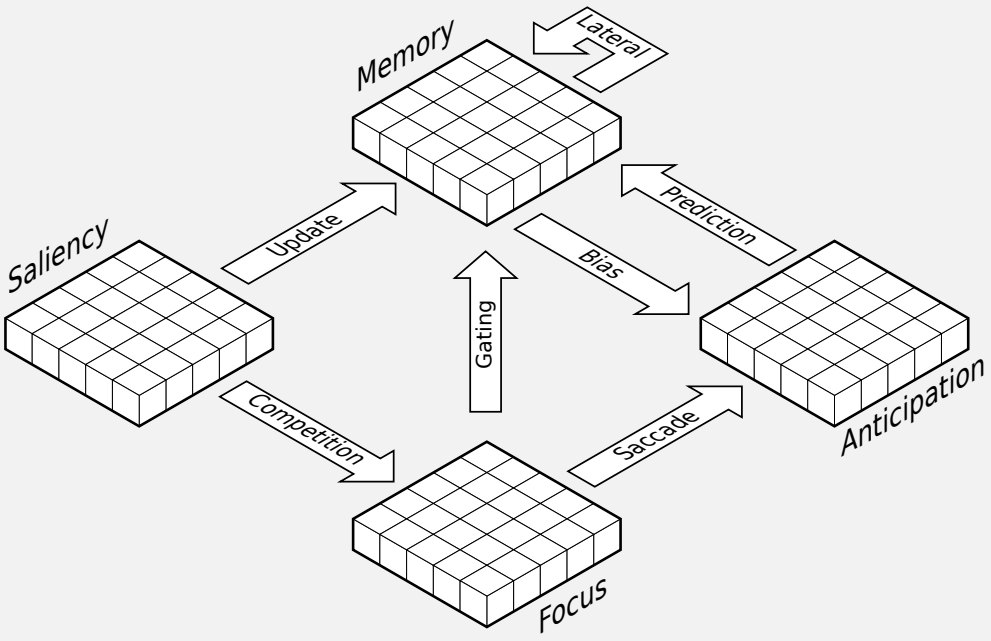
Gurney and his colleagues proposed in (Gurney et al., 2001a,b) a biologically plausible model of the basal ganglia (BG) complex based on the hypothesis that action selection is the primary role of the BG. The model is made of 9 different structures: Posterior Cortex (PC), Striatum D1 (St1), Striatum D2 (St2), Sub-Thalamic Nucleus (STN), External Globus Pallidus (GPe), Internal Globus Pallidus (GPi), Ventro-Lateral Thalamus (VLT), Prefrontal Cortex (PFC) and the Thalamic Reticular Nucleus (TRN). Each of these structures possesses its own equation for computing the activity as well as a specific set of connections with other structures.

The overall dynamic of the model is illustrated in figure 9 which reproduces faithfully results from (Gurney et al., 2001b). The DANA framework allowed us to transcript the model in a straightforward way. First, we defined all the structures according to their respective equations with `U` and `V` being respectively the core activity and the bound activity and the connections are made between individual structures, enforcing their respective topologies (one to one or one to all). Before running the model, we also plugged some timed events that change the input (activity in `PC`) on a regular time basis. Figure 9 shows the temporal evolution of the activity of the main structures.

Discussion and Future Work

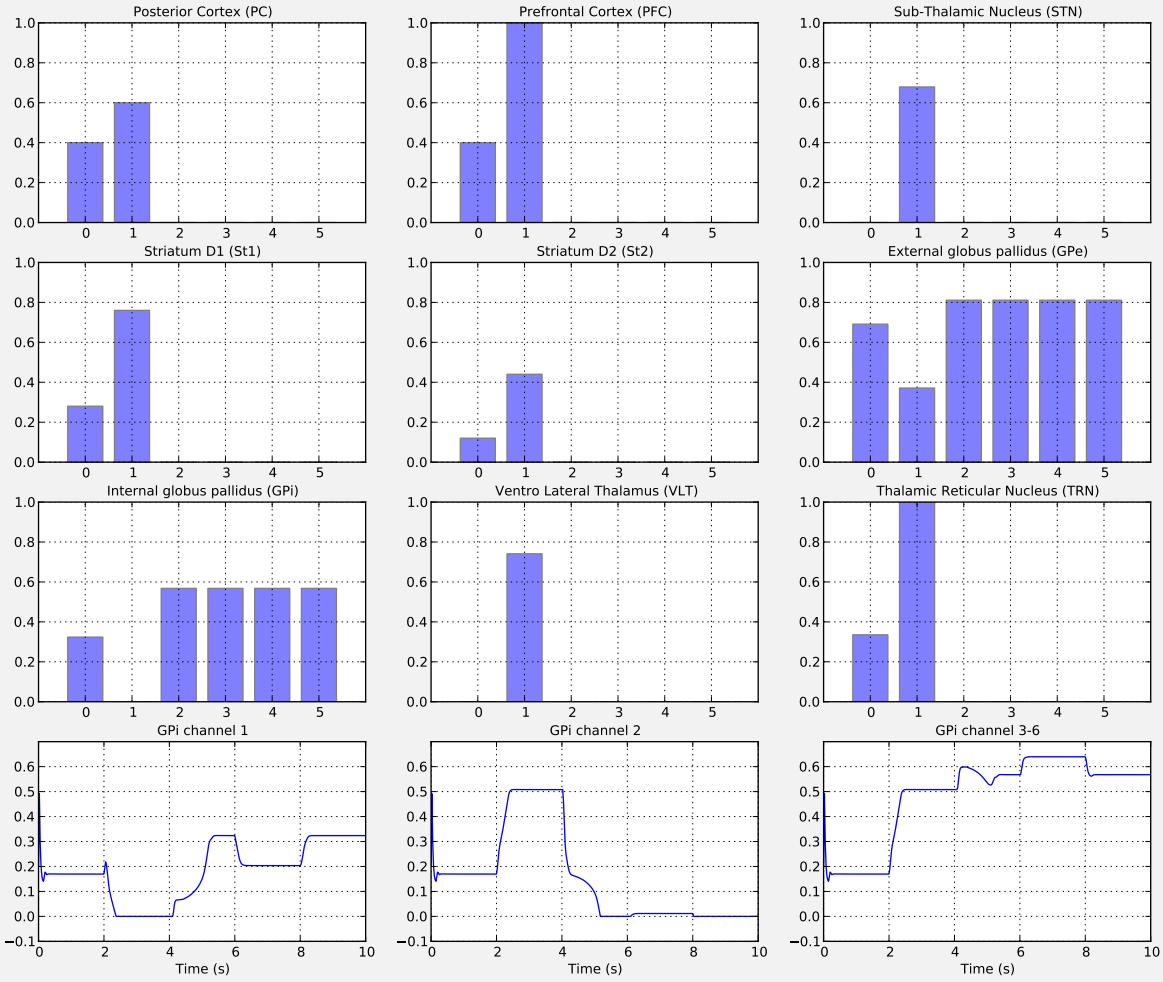
Asynchrony

The original implementation of the DANA framework offered the posibility to evaluate models asynchronously where units were updated in random order. The motivation for such numerical scheme was to reject the idea of a central clock that would signal any unit when to update and to avoid any implicit synchronization effect. The standard procedure to evaluate a time-dependent differential system is is to synchronously evaluate the values of each variables: the state at time $t + \Delta t$ is exclusively computed




```
class SigmaPiConnection(Connection):

    def __init__(self, source=None, modulator=None, target=None, scale=1, direction=1):
        Connection.__init__(self, source, target)
        self._scale = scale
        self._direction = +1
        names = modulator.dtype.names
        self._actual_modulator = modulator

    def output(self):
        src = self._actual_source
        mod = self._actual_modulator
        tgt = self._actual_target
        R = np.zeros(tgt.shape)
        if len(tgt.shape) == len(src.shape) == len(mod.shape) == 1:
            R = convolve1d(src,mod[::self._direction])
        elif len(tgt.shape) == len(src.shape) == len(mod.shape) == 2:
            R = convolve2d(src,mod[::self._direction,::self._direction])
        else:
            raise NotImplemented
        return R*self._scale
```

Figure 8: Schematic view of the architecture of a model for the saccadic exploration of a visual environment. The image captured by a camera is filtered and represented in the saliency map. This information feeds two pathways : one to the memory and one to the focus map. A competition in the focus map leads to the most salient location that is the target for the next saccade. The anticipation circuit predicts the future state of the memory with its current content and the programmed saccade. This is done through the use of a sigma-pi connections whose code correspond to the bottom part of the figure.




```
...
# Striatum D2: medium spiny neurons of the striatum with D2 dopamine receptors
St2 = zeros(n, """dU/dt = 1/tau*(-U + PC_ + PFC_)
                  V     = np.minimum(np.maximum(U-eSt2,0),1)
                  PC_; PFC_""")
...
# St2 connections
SparseConnection( PC('V'),  St2('PC_'),   PC_St2 * np.ones(1) )
SparseConnection( PFC('V'), St2('PFC_'), PFC_St2 * np.ones(1) )
...

# Timed events
@clock.at(2*second)
def update_PC(t):  PC.V[0] = .4

...

# Run simulation for 10 seconds
run(time=10*second, dt=1*millisecond)
```

Figure 9: Computational model of action selection in the basal ganglia from Gurney et al. (2001b). Full script with all definitions and graphic code is 250 lines (model is made of more than 18 equations, 9 groups and 20 connections between groups).

from the state at time t. The usual way to perform such a synchronization is to explicitly implement a temporary buffer at the variable level and store the computed value at time $t + \Delta t$. Once the value of all the vatiables has been evaluated at time $t + \Delta t$, the *public* value is replaced by the content of the buffer (there exist other ways of doing this synchronous evaluation, however the idea remains to separate information between time t and time $t + \Delta t$). To perform such a synchronization, there is thus a need for a global signal indicating to the variables that the evaluation period is over and that they can replace their previous value with the newly computed one.

At the computational level, this synchronization is rather expensive and is mostly justified by the difficulty of mathematically handling asynchronous models (Rougier and Hutt, 2009). For example, cellular automata have been extensively studied during the past decades for the synchronous case and many theorems have been proved in this context. However, recent works on asynchronous cellular automata (Fates, 2009) showed that the behavior of the models and their associated properties may be of a radically different nature depending on the level of synchrony of the model (only a sub-part of all the available automata can be evaluated asynchronously). Since we want to reject any kind of centralized process, we introduced asynchronous computations relying on results from Robert (1994) (discrete systems) and Mitra (1987) (continuous systems). However, if Mitra and Robert give conditions to ensure the proper convergence of the relaxation, we do not introduce such constraints within the framework since it might be quite a complex implementation. However, our recent study (Taouali et al., 2011) explained that even if asynchronous evaluation may provide a richer behavior in some circumstances, allowing a dynamical system to reach additional fixed points, the simple addition of noise provides an almost equivalent behavior. Consequently, we **removed the asynchronous evaluation** from the DANA framework in favor of a simple noise addition when necessary. This is left to the responsibility of the modeler.

Finite transmission speed

Until now, we have considered the velocity of connection between two groups to be infinite leading to an instantaneous transmission of information. However, considering such an infinite velocity in connections does not make sense from a biological point of view and forbids *de facto* the experimental study of some phenomena directly dependent on a finite transmission speed. We have proposed in (Hutt and Rougier, 2010) a fast numerical procedure (associated with its mathematical proof) that allows us to use delayed differential equations (DDEs) instead or regular differential equations. This requires a dedicated solver that has been detailed in (Hutt and Rougier, 2010) but has not yet been implemented within DANA.

Integration into Brian

As we already mentioned, Brian is a famous python library for simulating spiking neural networks. DANA is more dedicated to time continuous systems such as mean-rate model of neurons. Integrating DANA within Brian would offer the community a unified library for simulating both types of neural networks. Indeed, we tried to ensure a maximum compatibility with Brian in order to ease this integration and we already share a lot of common objects (equations, groups and connections) but the code currently differs (see for example fig. 10 and fig. 11). This integration is the subject of the BEP (Brian Enhancement Proposal) 24. However, before proposing this integration to the Brian development team, we aim at solving the case of finite transmission speeed connections in DANA that we mentioned in the previous section.

Acknowledgment

Declaration of interest

The authors report no conflicts of interest.

A Brian/DANA comparison

A.1 Game of Life

```
from brian import *
from scipy.signal import convolve2d
N = 256
K = array([[1,1,1],
           [1,0,1],
           [1,1,1]])
G = NeuronGroup(N*N, 'alive:1')
G.alive = randint(2, size=N*N)
A = reshape(G.alive, (N,N))

@network_operation
def update():
    n = convolve2d(A, K, mode='same')
    A[:] = maximum(0,1.0 -
      (n<1.5)-(n>3.5)-(n<2.5)*(1-A))

run(500*defaultclock.dt)
```

```
from dana import *

n = 256
src = Group((n,n),
  '''V = maximum(0,1.0-(N<1.5) - \
                       (N>3.5) - \
                       (N<2.5)*(1-V)); N''')
C = SparseConnection(src('V'), src('N'),
                     np.array([[1., 1., 1.],
                               [1., 0., 1.],
                               [1., 1., 1.]]))
src.V = rnd.randint(2, src.shape)


run(n=500)
```

Figure 10: Game of life written using Brian (left) and DANA (right). The Brian version is much slower only due to the lack of dedicated connections (*SparseConnection* in this example).

A.2 Integrate and Fire

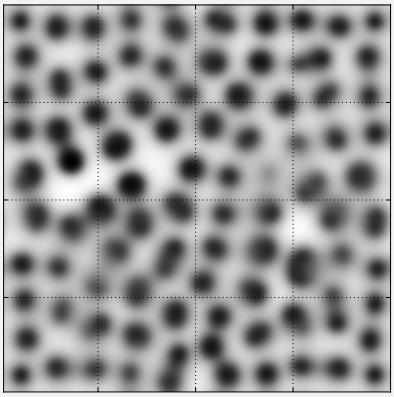
```
from brian import *
tau = 20 * msecond
Vt = -50 * mvolt
Vr = -60 * mvolt
El = -49 * mvolt
psp = 0.5 * mvolt

G = NeuronGroup(40,
    "dV/dt = -(V-El)/ tau : volt",
    threshold=Vt, reset=Vr)
C = Connection(G, G)
C.connect_random(
    sparseness=0.1, weight=psp)


G.V = Vr + rand(40) * (Vt - Vr)



run(1 * second)
```

```
from dana import *
msecond = 0.001
mvolt = 0.001
tau = 20.0 * msecond
Vt = -50.0 * mvolt
Vr =-60.0 * mvolt
El =-49.0 * mvolt
psp = 0.5 * mvolt

G = zeros(40, """dV/dt = -(V-El)/tau + I*psp
                      S = V > Vt; I""")
W = (rnd.uniform(0,1,(40,40))
      * (rnd.random((40,40)) < 0.1))
C = SparseConnection(G('S'), G('I'), W)
G.V = Vr + rnd.random(40) * (Vt - Vr)

@after(clock.tick)
def spike(t):
    G.V = np.where(G.V>Vt, Vr, G.V)
run(1 * second)
```

Figure 11: Integrate and fire model written using Brian (right) and DANA (left). While it is possible to define a spiking neuron model in DANA, the resulting simulation is highly inefficient compared to Brian performances.

B Beyond neuroscience

B.1 A reaction-diffusion system Pearson (1993)



```
Du, Dv, F, k = 0.16, 0.08, 0.020, 0.055
Z = Group( (128,128),
    """du/dt = Du*Lu - Z + F*(1-U) : float
       dv/dt = Dv*Lv + Z - (F+k)*V : float
       U = np.maximum(u,0)         : float
       V = np.maximum(v,0)         : float
       Z = U*V*V                   : float
       Lu; Lv """ )
K = np.array([[np.NaN,  1., np.NaN],
              [  1.,   -4.,   1.  ],
              [np.NaN,  1., np.NaN]])
SparseConnection(Z('U'),Z('Lu'),K,toric=True)
SparseConnection(Z('V'),Z('Lv'),K,toric=True)

run(n=10000)
```

Figure 12: Reaction Diffusion Gray-Scott model from Pearson (1993). Full script with graphic code is 110 lines.

References

Aisa, B., Mingus, B., O'Reilly, R., 2008. The emergent neural modeling system. Neural Networks 21, 1146–1152.

Bednar, J., 2008. Topographica: Building and analyzing map-level simulations from python, c/c++, matlab, nest, or neuron components. Frontiers in Neuroinformatics 3 (8).

Bienenstock, E., Cooper, L., Munro, P., 1982. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. Journal of Neuroscience 2 (1), 32–48.

Carnevale, N., Hines, M., 2006. The NEURON Book. Cambridge University Press.

de Kamps, M., Baier, V., Drever, J., Dietz, M., Mösenlechner, L., van der Velde, F., 2008. 2008 special issue: The state of miind. Neural Networks 21 (8), 1164–1181.

Fates, N., 2009. Asynchronism induces second order phase transitions in elementary cellular automata. Journal of Cellular Automata 4 (1), 21–38.

Fix, J., Rougier, N. P., Alexandre, F., 2011. A Dynamic Neural Field Approach to the Covert and Overt Deployment of Spatial Attention. Cognitive Computation 3 (1), 279–293.

Fix, J., Vitay, J., Rougier, N. P., 2006. A Distributed Computational Model of Spatial Memory Anticipation During a Visual Search Task. In: Butz, M. V., Sigaud, O., Pezzulo, G., Baldassarre, G. (Eds.), SAB ABiALS. Vol. 4520 of Lecture Notes in Computer Science. Springer, pp. 170–188.

Gerstner, W., 2002. Spiking Neuron Models. Cambridge University Press, Ch. 10.

Gewaltig, M.-O., Diesmann, M., 2007. Nest (neural simulation tool). Scholarpedia 2 (4), 1430.

Goodman, D., Brette, R., 2008. Brian: a simulator for spiking neural networks in python. Frontiers in Neuroinformatics 2 (5), 1–10.

Gurney, K., Prescott, T., Redgrave, P., 2001a. A computational model of action selection in the basal ganglia i: A new functional anatomy. Biological Cybernetics 84, 401–410.

Gurney, K., Prescott, T., Redgrave, P., 2001b. A computational model of action selection in the basal ganglia ii: Analysis and simulation of behaviour. Biological Cybernetics 84, 411–423.

Hanke, M., Halchenko, Y., 2011. Neuroscience runs on gnu/linux. Frontiers in Neuroinformatics 5 (8).

Hutt, A., Rougier, N., 2010. Activity spread and breathers induced by finite transmission speeds in two-dimensional neural fields. Physical Review E: Statistical, Nonlinear, and Soft Matter Physics 82.

Mitra, D., 1987. Asynchronous relaxations for the numerical solution of differential equations by parallel processors. SIAM journal on scientific and statistical computing 8 (1), 43–58.

Oja, E., 1982. Simplified neuron model as a principal component analyzer. Journal of Mathematical Biology 15 (3), 267–273.

O'Reilly, R., Y.Munakata, 2000. Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain. MIT Press, Cambridge, MA, USA.

Pearson, J., 1993. Complex patterns in a simple system. Science 261 (5118), 189–192.

Reynolds, J. N., Wickens, J. R., 2002. Dopamine-dependent plasticity of corticostriatal synapses. Neural Networks 15, 507–521.

Robert, F., 1994. Les systèmes dynamiques discrets. Vol. 19 of Mathématiques et Applications. Springer.

Rougier, N., Hutt, A., 2009. Synchronous and Asynchronous Evaluation of Dynamic Neural Fields. Journal of Difference Equations and Applications To appear.

Taouali, W., Vieville, T., Rougier, N., Alexandre, F., 2011. No clock to rule them all. Journal of Physiology - Paris 105 (1–3), 83–90.