RNS++

Real-time **N**eural **S**imulator

Documentation
v. 0.50
Feb. 5, 2003

Introduction

RNS++ is an addition to the PDP++ family of simulators and supports integrated simulation of neural systems, including behavioral (response time, error rate), fMRI, and neurophysiological data. RNS++ makes time, in real units of seconds or milliseconds, the backbone of a simulation, with continuity between trials[1]. Essentially, RNS++ is an environment for simulating neural networks as *dynamical system* models.

Features

- **Simulation**
    - Simulates biologically-realistic, systems-level networks
    - Simulates Hebbian learning, reinforcement learning, supervised learning
    - Simulate effects of multiple neurotransmitter and receptor types
    - Includes pre-defined activation and weight governing equations
    - Independent activation and weight governing equations – can mix and match
    - Built-in scripting language allows user-defined custom equations
    - Rate coding or spiking cells (integrate and fire)
    - Weights and activations update continuously
- **Data Fitting**
    - Fit multiple data types in one simulation – behavior, fMRI, neurophysiology, and more
    - Powerful gradient descent algorithm automatically fits model to data, regardless of governing equation choice or complexity of data set
    - Dissociation of learning and parameter fitting allows learning-related data to be fit
- **Ease of Use**
    - PDP++ graphical user interface
    - Parallel processing support for data fitting

RNS++ combines real-time simulation with a flexible time- and contingency-sensitive environment manager, neurobiologically-realistic activation equations (Grossberg, 1982; Usher and McClelland, 2001), and a variety of optional biophysical specializations in a menu-driven graphical interface. It is most appropriate for simulations where the timing of simulated behavioral response is important, where neurophysiological cell types or the fine time course of imaging data are to be modeled, or where biophysical specializations are simulated to satisfy functional constraints.

Timing

RNS++ simulates neural networks as dynamical systems. It uses finite difference equations to model neural activity as a rate code.

[1] The "real-time" designation means that the simulation will calculate time directly in units of seconds rather than abstract cycles. The simulator may not update in real time, however, depending on processor speed and other factors.

Complex time and task contingencies
RNS++ handles complex time and task contingencies. A given task may have multiple time-sensitive events and contingencies, both across and within trials. Certain behavioral tasks specify the timing between trials or within sub-events of a trial, such as the delay period in a memory-guided task. Experimental designs may also specify complex contingencies. For example, a memory-guided task may require a trial to be aborted if the subject fails to assume a readiness position within a certain interval. If a movement occurs before the cue is given, then the trial is also aborted. The duration of stimuli may be carefully timed. Reward contingencies depend on response timing.

RNS++ Triggers
RNS++ provides a flexible environment framework called Triggers for specifying arbitrarily complex timed events and contingencies such as these. Triggers monitor the model and environment, and they become "active" when a set of time and model response conditions are met, such as that the model responded within a given time limit. Model events become active and are shut off by specified triggers. In this way, the model and environment can interact with precise timing and arbitrarily complex contingencies.

Neurophysiological cell types
Real-time simulations are important to model certain kinds of data. At the cell level, physiological cell types have complicated signatures often with non-monotonic time courses of activity. These data provide unparalleled temporal and spatial resolution, which in turn can provide helpful constraints for neural models. The key to these cell types is their time courses of activity. Visual cells in the frontal eye field, for example, show a phasic or biphasic response to a visual stimulus (Schall, 1991); their activities die away well before the end of the trial. Movement-related cells show phasic activity around the time of movement. Anterior cingulate cells in the monkey (Shima and Tanji, 1998) show several varieties of movement-related cells (selective for task switching), differentiated by their time course (rate of activity increase) just prior to movement. RNS++ can simulate arbitrarily complex dynamical systems of interacting neurons, including the complex, non-monotonic signatures of the above cell types.

Learning vs. Fitting
RNS++ models neural networks as dynamical systems. Consequently, RNS++ distinguishes between model parameters and synaptic weights. Model parameters define the instance of the formal dynamical system model, while synaptic weights as well as neural activities are state variables of the dynamical system. Parameters determine how the model both learns and performs, namely how its activity and synaptic weights change. Learning in RNS++ occurs online, at each time step. Model parameters can be fit with gradient descent to the actual time courses of cell firing rates, to constraints on the model at equilibrium, and to behavioral statistics such as response time.

Neurobiology
Many cells have unique biophysical properties that provide significant functionality to a neural system. RNS++ provides a menu-driven framework for incorporating relevant

biophysical mechanisms into a model, to simulate the effects of known or predicted neurobiological features on the time course of cell activity and behavior.

Neurotransmitters and neuromodulators can have very different effects, depending on which receptor type they activate. RNS++ separately models excitation, inhibition, serotonin, dopamine, norepinephrine, and acetylcholine signals. Furthermore, the effect of each of these signals can be customized.

Certain models (Barto et al., 1983)use synaptic eligibility traces to model the temporal windows and time course of eligibility for learning. RNS++ includes synaptic eligibility trace options as well as other neuronal state variables, such as transmitter depletion.

Governing Equations

Unit Activity

RNS++ can simulate both rate-coded and spiking neurons. For rate-coded neurons, the activation level of a model neuron reflects the rate of spiking of a neuron. The rule that determines the current firing rate and its rate of change is a *governing equation*. There are currently currently four pre-defined governing equations, in addition to fully-customizable script-based equations.

Shunting

Shunting equations (Grossberg, 1982) are of the form

$$\frac{dx}{dt} = \tau\left[(1-x)I_e - (x+H)I_i\right]. \tag{1}$$

Here I_e is excitatory input (which may include feedforward and recurrent excitation), and I_i is inhibitory input. The continuous differential equation (1) can be converted to a finite difference equation (using the first-order Euler approximation) as:

$$x_{t+1} = x_t + \tau\left[(1-x_t)I_e - (x_t+H)I_i\right]. \tag{2}$$

where

I_e = net_excit + recurrent excitation + neuromodulator excitation + bias
I_i = net_inhib + neuromodulator inhibition + passive decay

Equations (1) and (2) show that the value of *x* at time t+1 is a function of its activity at time *t* and its excitatory and inhibitory inputs. Equilibrium occurs in equation (2) when the activity *x* of the model neuron stops changing with time. In this case, dx/dt = 0 in equation (1), and equation (1) becomes

$$x = \frac{I_e - HI_i}{I_i + I_e}. \tag{3}$$

Equation (3) tends towards an upper limit of (1) as the excitatory input I_e becomes large relative to I_i, and equation (3) tends towards a lower limit of -*H* when the inhibitory input I_i becomes large relative to the excitatory input. In this way, activity bounds are imposed by the form of the equation itself.

Usher-McClelland
(Usher and McClelland, 2001) extensively studied a differential equation model of cell activity, of the form

$$dx_i = \left[\rho_i - kx_i - \beta \sum_{i' \neq i} x_{i'} \right] \frac{dt}{\tau} + \xi_i \sqrt{\frac{dt}{\tau}} \qquad (4)$$

where
ρ_i is afferent excitation (in this case, net_excit + neuromodulator excitation + recurrent excitation)
kx_i is passive decay
$\beta \sum_{i' \neq i} x_{i'}$ is surround inhibition (net_inhib)*
and *x* is bounded below at zero.

*When using this governing equation, the signal-function threshold of the unit and afferents to the unit should be set to zero, and a recurrent inhibitory projection should be instantiated for the layer. These settings are described in more detail below.

Additive
Perhaps the simplest equation is:

$$\mathrm{dx} = I_e + I_i \qquad (5)$$

where
I_e and I_i are as defined above for equation (2). This equation is especially useful if an integrate-and-fire process is applied to the cell.

Tracking
A tracking equation is similar to (5) above but causes the activity to track the sum of excitatory and inhibitory inputs:

$$\mathrm{dx} = I_e + I_i - \mathrm{x} \qquad (6)$$

Custom
The user may define an arbitrary governing equation for cell activity, defined by the built-in scripting language.

Synaptic weights
RNS++ features several pre-defined learning equations.

Hebbian learning with decay:

The simplest learning law increases synaptic weight as a function of pre-synaptic (x) and post-synaptic (y) activity, with passive decay:

$$dw = xy + A(\text{pass\_wt\_decay.baseline} - w) \qquad (7)$$

where
A is the rate of passive decay toward its baseline.

Presynaptically-gated learning:
Learning may be dependent on pre-synaptic activity, in which case the learning law is:

$$dw = x(y-w) \qquad (8)$$

Postsynaptically-gated learning:
Conversely, learning may be dependent on post-synaptic activity, in which case the learning law is:

$$dw = y(x-y) \qquad (9)$$

Custom learning laws
The modular learning law allows more flexibility without the need for a script. LTP and LTD components can be defined independently. The rate and asymptotic value of each LTP and LTD process can be adjusted independently, and the learning can be restricted to only those times when a neuromodulator signal exceeds (or falls below) a specified threshold.

For maximum flexibility, a script can also be used to control the learning laws.

RNS++ objects
In the following documentation, the reader is assumed to be familiar with PDP++ and PDP++ objects. Therefore, only the additions to and modifications of the PDP++ base will be described.

RtUnit

RtUnit is the basic neural unit, derived from the PDP++ class *Unit*. It is a data structure which keeps track of afferent signals as well as current activity. Most of the management for RtUnit objects is performed by RtUnitSpec objects, described below.

The "net" field is not used; instead, afferent excitation and inhibition are summed separately as "net_excit" and "net_inhib" respectively. The bias field is not used, either.

Afferent neuromodulator signals are also summed individually, including dopamine ("net_dopa"), norepinephrine ("net_ne"), acetylcholine ("net_ach"), and serotonin (“net_sero”). Receptor properties for these neuromodulators are controlled by RtUnitSpec. Note that signals for dopamine d1 and d2 signals are computed; however, these are deprecated; net_dopa should be used instead with appropriate receptor properties specified by the RtUnitSpec.

Some networks may require a persistent trace of activity; this is stored as “act_trace”, which rises to current network activity level and decays slowly, according to:

$$\text{d\_act\_trace} = \text{act\_trace.rise\_rate}*[\text{act} - \text{act\_trace}]^{+} - \text{act\_trace.decay\_rate}*\text{act\_trace} \quad (10)$$

where $[x]^{+}$ denotes max(x, 0).

Some networks may also use random sampling of the input for learning. The Boolean “isLearnEvent” is true if the postsynaptic cell is enabled to randomly sample the input for learning purposes. Learning occurs in certain cases when postsynaptic activity exceeds a specified threshold (dpost_l). A *sampling event* occurs when dpost_l is temporarily lowered, so that the post-synaptic cell can learn to respond to the pattern of afferent activity despite weak postsynaptic activity.

The voltage clamp boolean, if true, maintains the unit at its current activity level, regardless of inputs.

The max_axon_delay allows models to incorporate a synaptic delay in the RtUnit efferents up to the specified number of time steps.

RtUnitSpec

RtUnitSpec provides a set of rules that govern the state of RtUnit objects. It is derived from the PDP++ class *UnitSpec*

The bias_con_type and bias_spec fields are not used.

The governing equation is determined by the field “govern_eq” and determines how the RtUnit activity is updated. It can be one of Rt_Shunting, Rt_Tracking, Rt_Additive, or Rt_Usher_Mc, described above.

The governing equation parameter field (“gov”) controls the behavior of the activation equations. The time_const determines the *rate* of activity change for each kind of activation equation. The passive decay (“pass_decay”) and bias excitation (“bias_excit”) members define constant inhibition and excitation, respectively. The hyperpolarization field (“hyperpol”) defines the lower bound toward which inhibition drives activity (“H” in equations (1) – (3)), although activity may be bounded below at zero by the “act_range”field.

The time_const is the rate at which the activation equations change.

NMDA channels are typically blocked by Mg2+ until a certain level of depolarization is reached, at which point the Mg2+ is expelled, and the NMDA channel can be activated by glutamate to allow the influx of both Na and Ca. The model framework specifies NMDA_thresh as the activity level above which the NMDA channels are deemed to unblock. The NMDA_gain specifies the increased effect of net_excit on cell activity when the RtUnit activity exceeds NMDA_thresh; a gain of unity means no effect. Essentially, net_excit is multiplied by NMDA_gain only when RtUnit's act exceeds NMDA_thresh.

The neuromodulator signals representing dopamine, norepinephrine, acetylcholine, and serotonin are maintained as net inputs to a given RtUnit. However, their effect depends on the receptor molecules they activate. The effect of these receptor molecules on *activity* can be modeled as follows (the effect on learning is modeled separately). Each neuromodulator signal can be independently configured to directly excite or directly inhibit cells, with strengths defined by the ex_wt and inh_wt parameters, respectively. Likewise, each neuromodulator signal can multiply the strength of other afferent excitation or inhibition with strengths defined by the ex_mult and inh_mult parameters, respectively. Specifically, afferent excitation and inhibition are multiplied by (1+sum(ex_mult*net_<nmod>)) and (1+sum(inh_mult*net_<nmod>)), respectively, where <nmod> represents a given neuromodulator signal (net_ne, net_ach, net_dopa, or net_sero).

Habituation refers to the gradual reduction of an internal state variable with continued cell activity. When act exceeds habit_thresh, the RtUnit "habit" variable begins to decay toward zero at rate habit_rate. The rates of decay and recovery are both multiplied by habit_time_const.

Similarly, the RtUnit act_trace increases towards RtUnit act at rate at_rise_rate, and it decays to zero at rate at_decay_rate.

Each RtUnit can receive recurrent (self) excitation, which is multiplied by recur_excit_wt, and which can be of the form LINEAR, STEP, SQUELCH, XSQUARED, or SIGMOID. The parameters are level (a) recur_excit_level_a, and gain (n) recur_excit_gain_n. LINEAR recurrent excitation is of the form n[act-a]+, where []+ denotes positive rectification. STEP provides recurrent excitation value n if act > a, and 0 otherwise. SQUELCH provides recurrent excitation of n*act if act > 0, and 0 otherwise. XSQUARED provides excitation n*act*act. SIGMOID provides recurrent excitation act^n/(a^n+act^n).

Inputs can be either hard clamped or soft clamped, if the soft_clamp Boolean is true. Hard clamping summarily sets the value of the unit to the value of the input. Soft clamping uses the value of the input, multiplied by soft_clamp_gain, as a forcing function to the governing equations.

In some cases, it may be desirable to normalize the input weights or the output weights of a unit to 1, for example with competitive learning. This can be accomplished by setting norm_input_wt and norm_output_wt, respectively, to true.

RtUnit activities can be initialized with a random variable, specified by initial_act

Random fluctuations can be added to the RtUnit activities act at each time step, specified by the noise field.

Learning events can occur that allow weights to be updated more easily. Learning only occurs when postsynaptic activity exceeds a specified threshold activity. The samp_learn field allows this learning threshold to be lowered, so that learning occurs even with lower postsynaptic cell activity. A learning event occurs in a given cell with probability "prob" per second. If a learning event occurs, it lasts for "dur" seconds, and it lowers the postsynaptic learning threshold by dpost_l.

In addition to rate coding, cells can be modeled as spiking with an integrate-and-fire (IAF) feature. If the Boolean do_int_and_fire is true, then whenever activity exceeds iaf_Thresh, the RtUnit act is set to 1.0 immediately, and then to zero at the subsequent time step.

In some applications, it may be desirable to recode the magnitude of a cell's activity as a map of activity. For example, cells may respond only to a certain level of intensity. If the VectorMap field is active, then cells will respond maximally if net_excit == peak, and the response tapers off linearly to zero at peak +/- hwidth. The resulting signal is multiplied by VectorMap.gain. Essentially, the cell will only respond to input within a certain magnitude range.

RtCon

RtCon is derived from the PDP++ class *Connection.* It maintains the current weight "wt" or connection strength by which one RtUnit signals another. RtCon also maintains an eligibility variable ("elig"), which determines the learning eligibility of individual synapses. The weight may be clamped at a fixed level if the Boolean isClamp is set to true, so that no weight updates occur, regardless of the choice of learning rule.

RtConSpec

RtConSpec determines how RtCon objects are used and updated. It is derived from the PDP++ class *ConSpec.* RtConSpec defines the governing equations by which RtCon weights are updated.

The "rnd" field defines the initial values of weights governed by the RtConSpec instance.

The "wt_limits" field governs the minimum and maximum values that weights can have, regardless of the learning rule.

The “con_type” field determines how the activity is processed by the receiving RtUnit. UNUSED means that no activity will be sent. Otherwise, activity will be summed ain the receiving unit at each time step as follows:

Option	Recipient RtUnit field
EXCIT	net_excit
INHIB	net_inhib
NE	net_ne
DOPA	net_dopa
SERO	net_sero
ACH	net_ach

The “signal_func.thresh” variable defines the activity level (“act”) which the sending unit must exceed in order to signal the receiving unit. The threshold signal function (“signal_func.sig_type”) can be either rectifying (RECT): [act-thresh]+ or squelching (SQUELCH): (act > thresh ? act : 0).

The signal at each synapse is multiplied by wt_gain, which serves to modulate the strength of an entire projection from one layer to another.

The signal from any given sending unit to a receiving unit can by capped at a maximum value of synapse_sat. This saturating function is applied **prior to** wt_gain.

The learning laws are described in the section on Governing Equations above. All learning equations (except Custom script-based instances) are governed by the learning rate parameter lrate.

Specific excitatory projections can be excluded from weight normalization by setting exclude_norm.rwt or swt, to exclude the projection from weight normalization with the receiving or sending unit, respectively.

In some cases, it may be desirable to limit postsynaptic eligibility to the time immediately following onset of postsynaptic activity. This can be accomplished by setting “habit_post_learn” to true, which reduces the learning ability of postsynaptic cells with continued postsynaptic activity.

Model Inputs

RNS++ uses inputs from RtEvent objects in the Environment object. These RtEvents are controlled by RtEventSpec objects.

Events and Triggers

RtEvent presentation is controlled by Triggers, which are members of RtEventSpecs. These triggers monitor the environment for certain conditions, such as a length of time elapsing or a response from the model. When triggers become active, they can cause events to be presented to the network as input, or they can cause existing inputs to be shut off. They can also cause the trial to be ended.

Triggers are maintained in the "triggers" field of RtEventSpecs. Each trigger has a set of TriggerCondition objects in the "trigCondGroup" field. The trigger becomes active according to the trigger_logic field, which specifies whether all of the trigger conditions need to be satisfied, or whether any one condition is sufficient to activate the trigger. The trig_action field specifies what happens when a trigger is activated. The PASSIVE setting is used for presenting events at a specific time, because the RtPatternSpecs (components of the RtEvent) monitor the triggers and act accordingly, without the trigger having to actively do anything. The TRIAL_END trig_action specifies that the trial should immediately end. The TRIAL_END_NEXT_SPEC field does not yet do anything except end the trial.

Each TriggerCondition monitors for several types of events:

Condition	**Description**
ELAPSED_TRIAL_START	The condition is satisfied when "elapsed_time" (in seconds) passes since the start of the trial.
ELAPSED_LAST_TRIGGER	The condition is satisfied when "elapsed_time" (in seconds) passes since the "other_trigger" became active
LAYER_OVER_THRESH	Satisfied when the specified layer activity ("aLayer") exceeds the specified "thresh"
CORRECT	Satisfied when a valid ResponseStat object for the current process running the network specifies that a **correct** response was made by the network
INCORRECT	Satisfied when a valid ResponseStat object for the current process running the network specifies that an **incorrect** response was made by the network
NOT TRIGGER	Satisfied if and only if the specified "other_trigger" has NOT been activated

Triggers are monitored by RtPatternSpec objects, which can be edited from the environment RtEventSpec view by selecting a layer represention in the event and then pressing "Edit Pat(s)" in the Enviro view. The RtPatternSpec "needsTrigger" field specifies whether a specified trigger ("on Trigger") needs to be active for the event to be presented. Likewise, the "needsVetoTrigger" specifies whether the activity of the specified "vetoTrigger" can shut off the event presentation, regardless of the "needsVetoTrigger" trigger state.

With these mechanisms, the Trigger system allows arbitrarily complex and time-sensitive contingencies to be monitored and acted upon.

Running Models

SchedProcess is the base class for Objects that control how the network simulations are run. RtTrial processes run the network through one trial. RtEpoch processes run the network through a single epoch, defined as a pass (in some order) through all events in the environment. TrainProcess objects are used unmodified from the main PDP++ distribution.

RtTrial processes have an associated ResponseStat object, derived from class Stat. ResponseStat objects monitor the network for responses, defined as when the specified Layer ("layer") exceeds a specified "threshold", or when the maximum change in activation falls below a specified "settle_thresh". These mutually-exclusive criteria are "THRESHOLD" and "SETTLE", respectively. The relevant criterion to use is set by the "response_type" field. When a response occurs, the response time is registered in the "resp_time" variable. The response time can be relative to the trigger time of a specified trigger object, set by the "responseSinceWhen" object. Responses occurring within "minResponseTime" of "responseSinceWhen" will be ignored. A response is deemed correct according to the RtPatternSpec corresponding to the layer of the "layer" field. Specifically, the response is deemed correct if the maximally-active unit of "layer" has a corresponding non-zero pattern unit in the RtPatternSpec. Otherwise, the response is deemed incorrect. The "resp" field is set to 0 if no response has yet been made, -1 if the response was incorrect, and 1 if the response was correct. The "tr" field points to the parent RtTrial object. Model activity can be logged with MonitorStat objects, which are a standard feature of PDP++.

Data Fitting

RNS++ features a powerful set of data fitting tools for fitting models to arbitrarily complex data sets.

RNS++ explicitly dissociates learning or training from model fitting. This dissociation allows RNS++ to fit existing data on learning to the way in which a model learns. Specifically, a model can be represented as a *dynamical* system **x**:

$$\mathbf{x} = [\mathbf{a}\ \mathbf{w}\ \mathbf{s}], \tag{11}$$

where

a is a vector representing the activation state of the model
w is a vector representing the synaptic weights
s is a vector representing other state variables of the system, such as learning eligibility and activity traces

The system of governing equations defines a dynamical system, in which each component of **x** is updated at each time step by numerical integration, specifically a first-order Euler finite difference approximation. The system of governing equations is defined by:

$$\frac{d}{dt}\mathbf{x} = \mathbf{f}(\mathbf{x}, t; \mathbf{p}) \qquad (12)$$

where

p is a vector of parameters, which determine the particular instance of the formal model **f** in (12).

It is important to note that (12) includes all governing equations for the model, including weight update equations as well as activation update equations. The model behavior **M** is found by numerically integrating **x** over the relevant time period:

$$\mathbf{M}(\mathrm{t}; \mathbf{p}) = \int_{t0}^{tf} \mathbf{f}(\mathbf{x}, t; p)dt$$

(13)

Data

A model **f(x,** t; **p)** is designed to fit a data set Ω. This data set may consist of several data types:

1) **Behavior:** Desired reaction times to certain events or aggregate statistics such as behavioral error rate

2) **Electrophysiology**: Cells recorded in awake, behaving animals have specific time courses associated with events and responses. These time courses may correspond to model cell activity time courses.

3) **Static constraints**: It may be desirable to have model cells show a certain equilibrium level of activity under specific sets of input conditions. For example, it might be desirable to have a cell with an equilibrium activity of 0.3 when net input is zero and 0.8 when net_excit is 1.0.

These types of data provide constraints on the parameters **p** for a given model. In general, the best fit model is defined by the best least-squares fit between the model output and the data to be fit. An objective function formalizes this criterion:

$$\mathrm{E} = (\Omega - \mathbf{M(p)})^2, \qquad (13)$$

where

M is the model output given parameter set **p** and corresponding to the desired output Ω. The best fit model is thereby found by minimizing E. A large class of gradient descent algorithms has been developed to minimize E. However, a complicating factor is the presence of noise in many simulations. While noise is sometimes necessary, for example to fit error rate data, it means that E may vary from one evaluation to the next, . R. Bogacz (http://www.math.princeton.edu/~rbogacz/autofit/) has shown that the Subplex algorithm (Rowan, 1990), which is a variant of the Nelder-Mead (Nelder and Mead, 1965) Simplex method, reliably minimizes (13).

Once **p** has been found such that (13) is minimized, the goodness-of-fit may be evaluated by two methods. First, the chi-squared test evaluates the null hypothesis that the differences between best-fit model output and data are due to random noise, and that the model accurately fits the data. The degrees of freedom (DOF) in the model are the number of data points in the vector Ω, less the number of parameters **p**. The chi-squared statistic is:

$$X^2 = \sum_i \left(\frac{\Omega_i - M_i}{\sigma_i} \right) \tag{14}$$

where

i is a data point to be fit

σ_i is the standard deviation associated with point i

Another somewhat weaker method to evaluate goodness-of-fit involves testing the significance of the Pearson correlation between the model and the data.

Implementation

Parameters: The ModelParam Object

The parameter set **p** is maintained by the ModelParam object, which is derived from class BaseSpec. Once a ModelParam object is created, the updateParamList button will create a list of paramMembers, each of which references a floating point field in all available spec objects (e.g., RtUnitSpec, RtConSpec, etc.). The parameter is identified by the name of the spec object that owns it, by the field name, and by the subfield name, if the parameter is part of an inline object in the spec. Each paramMember contains a boolean field "fit", which if checked, will include the value as a parameter to be fit. Upper and lower limits can also be specified, so that the fitting algorithm with not find inappropriate values for the parameter.

Data: The FitDataSpec Object

The data to be fit by the model are maintained in the FitDataSpec object. Each FitDataSpec contains a list of **FitDataInfo** members, each of which contains a data element to be fit. FitDataInfo can specify several different kinds of data to be fit, corresponding to the above three categories of behavior, electrophysiology, and static constraints. Alternatively, for maximum flexibility, a custom script-based objective function can be defined. Each FitDataInfo member calculates a difference between the data and the actual model output. The "fitEnergyVal" field is set to the square of this difference, multiplied by "constraint_wt". The types of data points specified by "constraint_type" are:

STATIC_STID; STATIC_SEE: These constrain a specified layer and unit in the network ("whichLayerRecorded" and "whichUnitRecorded") to maintain a specified equilibrium value "desiredAct". The error between actual and desired equilibrium can be calculated in two ways: the squared time derivative (STID), or the squared equilibrium error (See and Ryan). The two are not generally equivalent due to the nonlinearity of the governing equations. The STID method clamps the unit value to the desired value, then calculates and squares the time derivative for the unit, defined by (12). The SEE method requires more computation than STID but may work better under some circumstances. The SEE method allows the model to run to equilibrium, then finds and squares the difference between the actual and desired equilibrium values.

Dynamic constraints refer to constraints on the time course of the model activity and include reaction time as well as electrophysiological constraints. Each constraint is evaluated by running the model through one or more events, defined sequentially in the "events" list. Events are added to the list by selecting the appropriate event in the "eventToLink" field and then pressing the "LinkEvent" button.

DYN_RT: The model is run through the specified event(s), and the model response time on the *last event in the list* is recorded in the "actualRt" field.

DYN_CELL: The model is run through the specified events, and the time course of activity is recorded from the unit "whichUnitRecorded" in layer "whichLayerRecorded". The time and unit activity for the model are recorded in "model_time" and "model_act". These are compared with the corresponding "data_time" and "data_act" arrays, set by the user. The dataAlignTime specifies when in the model trial should be time zero, for alignment with the data to be fit. Also, if the timing of a model event is variable, the alignment time can be specified to coincide with the trigger "alignTrigger". Each data point will be multiplied by "data_gain" at each time step, in order to scale the data to the model.

The DYN_CELL_RT setting calculates both of the above constraints, in order to economize computation where possible.

The button CalcRSquare finds the R^2 value, which is the correlation between the model and the data. Currently, this only works for response time data, i.e., the "actualRt" and "desiredRt" fields, provided that constraint_type is DYN_RT or DYN_CELL_RT.

Fitting the data: The RtConstrain process object
Once the parameters to be fit are chosen and the data specified, an RtConstrain process performs the fit. It evaluates an objective function, which is a measure of the sum-squared error of the model/data fit. RtConstrain derives from RtEpoch, which in turn derives from EpochProcess. Thus, RtConstrain acts like an epoch process, which controls an RtTrial sub-process to actually run the network. Users can choose from three possible optimization methods in the field "fit_method": BFGS (Press, 1992), Simplex (Nelder and Mead, 1965; Press, 1992), or Subplex (Rowan, 1990). The appropriate ModelParam object must be specified in the "params" field, and the FitDataSpec must be

specified in the "fitData" field. The details of each optimization process are maintained in the "op" (BFGS), "simp" (Simplex), and "subp" (Subplex) objects, respectively. The "trial_proc" field specifies the RtTrial object that will actually run the network during the fitting procedure.

If BFGS is used, then the gradient must be explicitly calculated by perturbing each parameter and evaluating the objective function. The degree of perturbation follows an annealing schedule, so that exponentially smaller parameter perturbations are used as fitting progresses. The maximum initial perturbation is specified by max_delta_p, and the minimum by min_delta_p. These may be either absolute values or fractions of the corresponding parameter, as specified by the "dp_bounding" field.

In some cases, the user may wish to penalize the network for an incorrect response. This can be accomplished by setting the "errorPenalty" field to a positive, nonzero number.

Fitting procedures can require large computational resources. Intermediate results can be written to a file regularly if the "loopDumpParams" boolean is true. In this case, a text-based dump of the current best parameter set will be saved to the file specified by "paramDumpFile"
Fitting processes can be speeded up with parallel processing. The individual processes must be started manually, and their hosts must share a common file system. The "dFit" object controls this. It specifies whether distributed fitting will be used. A single process should be set as master; it will direct all other slave processes and collect results from them. The number of slave processes ("numSlaves") must *include* the master process, since it also doubles as a slave process. The set of FitDataInfo constraints will be divided evenly among the processors. In some cases, each constraint may depend on the state of the network set by previous constraints in the list. The "numOverlap" field allows a specified number of immediately preceding events in the sequence of FitDataInfo constraints to prime the network before model output is recorded. Thus, there is overlap between processes in actually evaluating the constraints. A custom, script-based objective function may be computed last in order to collect aggregate statistics (e.g., behavioral error rate) from the network. If the script depends on results stored in the previous constraints, then the distributed fit manager must wait until all other parallel processes have completed and reported in before running the final script. This is assured if the "waitLastScript" boolean is checked. Finally, note that the parallel processing support described here currently applies only to fitting, not to running the network conventionally. Essentially, different constraints specified by FitDataInfo objects are processed in parallel on separate processors, but each constraint can only be evaluated by an RtTrial process running on a single CPU. More sophisticated parallel processing support is being developed as part of the PDP++ package, but RNS++ does not currently use this.

Tutorial

Tutorial:

This tutorial is designed to point you in the right direction and give a basic overview of how RNS++ works.

Open the rt_demo.proj.gz by executing at the command line:

```
        % rns++ rns_demo.proj.gz
```

You will see the project view, network view, and environment view.

Project View
Click "View Project" in the project window if the processes (e.g., Train_0) are not visible. Right click on the top-most Train process. Press "Run" and watch the network and graph log. You will see the input come on and gradually activate the output unit. The graph log plots the time course of this activity.

To implement **axonal delays**, click the project view "View Specs" button. Right-click the RtConSpec_0 icon to bring up the edit view. Adjust the "delay" variable to a positive value. Then run the simulation again by pressing the "Run" button on the Train_0 process control panel. You will note that there is a delay between the onset of the network input and the activity of the output. See "triggers" below for info on the timing.

If you want two or more different axonal delays coming out of the same neuron, then create another RtConSpec object and change the delay for each different delay you want. Then apply the conspecs to the different projections out of the neuron.

To **fit the model** to a given reaction time, edit the FitDataSpec_0 icon in the Spec view of the project window. Then edit the "data" field of this spec. In the data edit window, set "desired Rt" to a number around 0.2, press apply, and then OK. Now press the "View Project" button in the project view to view the process objects. Right click "Constrain_0" to open the control panel. Press Run and wait a few seconds for it to stop (it will run faster if you uncheck the "update" button in the upper left corner of the network view). The model has just performed a suplex optimization of certain parameters to fit the model to the given reaction time. To see which parameters were adjusted, open the "ModelParam_0" spec in the project view, and edit "param Members". Scroll to the right until you see an entry with the "fit" box check-marked. This entry describes the parameter that was fit. It is possible to fit multiple parameters to multiple data points, although this example fits only one parameter to one data point.

Environment View
The model can implement arbitrarily complex time contingencies. In the environment view, click "Edit Specs" if the event specs are not already visible. Now right click the EventSpec_0 entry in the lower left. Edit the triggers field in the edit dialog that appears. This dialog contains a set of triggers, which are activated by certain events in the trial. Edit a few of the "trig Cond Group" fields to see what conditions specify the given triggers. Note that the timing of triggers is determined by in the TriggerCondition objects, NOT in the "when Triggered" field of the Trigger objects. The "when Triggered" simply records what actually happened rather than what was desired. Next, in

the EventSpec_0 dialog box, edit the "patterns" member to see how pattern specs rely on triggers to know when to present their inputs to the network. Note the "needs Trigger" and "Needs Veto Trigger", which turn patterns on and off at times determined by the triggers.

Spiking Neurons
If you want a spiking neuron, edit the "RtUnitSpec_0" icon in the project view, and check on the "do Int And Fire" box at the bottom. Run the network again and watch the spikes in the output neuron. They are best seen in the graph view.

References

Barto AG, Sutton RS, Anderson CW (1983) Neuronlike adaptive elements that can solve difficult learning control problems. IEEE transactions on Systems, Man, & Cybernetics 13:834-846.

Grossberg S (1982) Studies of Mind and Brain. Boston: Reidel.

Nelder J, Mead R (1965) A Simplex Method for Function Minimization. Computer Journal 7:308-313.

Press W (1992) Numerical Recipes in C. Cambridge: Cambridge University Press.

Rowan T (1990) Functional Stability Analysis of Numerical Algorithms. In: Department of Computer Sciences: University of Texas at Austin.

Schall JD (1991) Neuronal activity related to visually guided saccades in the frontal eye fields of rhesus monkeys: comparison with supplementary eye fields. J Neurophysiol 66:559-579.

See STK, Ryan EB (1995) Cognitive mediation of adult age differences in language performance. Psychology & Aging 10:458-468.

Shima K, Tanji J (1998) Role of cingulate motor area cells in voluntary movement selection based on reward. Science 282:1335-1338.

Usher M, McClelland JL (2001) The time course of perceptual choice: the leaky, competing accumulator model. Psychological Review 108:550-592.