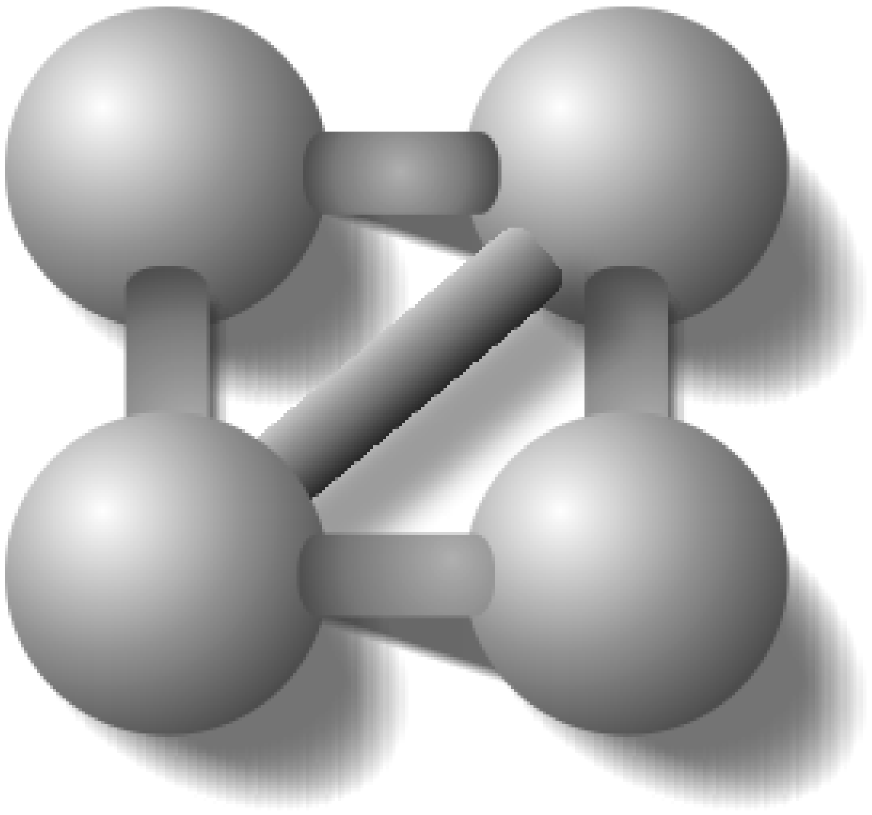
NetClamp Software User Manual

September 22, 2013



Aplysia Research Laboratory
Fishberg Department of Neuroscience
Friedman Brain Institute
Mount Sinai School of Medicine
New York, NY 10029

E. Brady Trexler, Ph.D.
ebtrexler@gothamsci.com
with
Klaudiusz R. Weiss, Ph.D.
klaudiusz.weiss@gmail.com
and
Elizabeth C. Cropper, Ph.D.
elizabeth.cropper@mssm.edu

Contents

Part I
Goals of Software

NetClamp is a computer program that addresses the interrelated yet distinct purposes of modeling neural networks and performing dynamic clamp experiments. Its intuitive and powerful interface facilitates designing, building and testing a network of cells with a variety of synaptic connections and intrinsic conductances. The networks can be hybrid in nature, containing both model cells and biological cells. The program operates in a pure simulation mode if the network contains no biological cells. Inclusion of one or more biological cells results in activation of the DAQ hardware, sampling of the membrane potential, and injection of the mathematically modeled currents to "dynamically clamp" the cells. Combining any combination of model cells and biological cells in NetClamp provides unprecedented flexibility for experimentation.

For users who wish to simulate neural network activity apart from dynamic clamp experiments, NetSim is provided as an alternative to NetClamp. NetSim does not utilize or require data acquisition hardware or drivers to be installed and thus can run on any computer.

NetClamp was inspired by and borrows heavily from DynamicClamp (v. 1.58) software from Farzan Nadim's laboratory at Rutgers University, NJ. This software could not have been written without the insights and lessons learned with Dr. Nadim's instruction and support.

Part II
Getting Started

Hardware requirements for NetClamp are E Series or M Series PCI boards from National Instruments.

Description of Software:

NetClamp and NetSim allow users to build networks of synaptically connected neurons with a variety of different current types. Models also include stimulating electrodes for injecting currents. The available network components are:

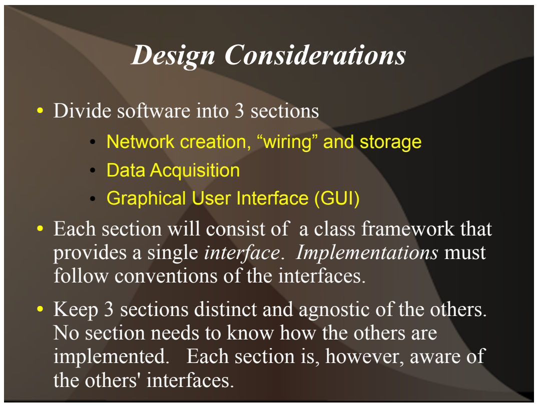
- **Cell Types**
  - *Model Cells* compute the membrane potential in software, whereas
  - *Biological Cells* sample the membrane potential from electrophysiologically recorded neurons.
  - *PlayBack Cells* are used to regenerate a previously recorded cell membrane potential.
- **Synapse Types**
  - *Bi-Directional Synapses* allow unidirectional as well as reciprocal communication between cells. Reciprocal synapses are commonplace in the retina, and this synapse configuration eases the creation of networks. Multiple currents can be added to either direction, producing compound synapses (e.g. AMPA and NMDA).
  - Gap Junction Synapses are simple resistive connections between cells.
- **Current Types**
  - *Hodgkin-Huxley Currents* are voltage dependent currents with 3 HH style gates. They can be inserted in cells or synapses.
  - *HH Synaptic Currents* for cases when the postsynaptic conductance is voltage dependent.
  - *Gap Junction Currents* are simple ohmic currents that only can be inserted into synapses.

- *Voltage Clamp (PID) Currents* that use a Proportional-Integral-Derivative algorithm to clamp a cell's voltage to a command value (essentially a software voltage clamp).

- **Electrode Types**
  - *Square Pulse Injection Electrodes* inject current periodically according to user settings.
  - *Playback Electrodes* inject current determined by a user supplied waveform. Users set the scale and offset to be applied to the data in the file.

Part III
Software Design and Implementation

NetClamp source code is divided into three interconnected components. First, a class framework provides the rules for setting up a network of neurons with their respective synapses and intrinsic currents. A second set of classes provides the interface to data acquisition and experimental control hardware. A third set of classes builds a highly flexible user interface that, in conjunction with the network classes, can accommodate new ideas easily and robustly.



Design Considerations

- Divide software into 3 sections
  - Network creation, "wiring" and storage
  - Data Acquisition
  - Graphical User Interface (GUI)
- Each section will consist of a class framework that provides a single *interface*. *Implementations* must follow conventions of the interfaces.
- Keep 3 sections distinct and agnostic of the others. No section needs to know how the others are implemented. Each section is, however, aware of the others' interfaces.

The software was coded in the C++ language using the Embarcadero RAD Studio development environment for Microsoft Windows(R) operating systems. Thus, the present implementations of the GUI and DAQ portions of the software are dependent on the Windows(R) environment, but one goal of the development process was to create the *network creation, wiring and storage* portion without this dependency. In realization of this goal, the collection of abstract classes that forms the network logic respects the required interactions with the GUI and DAQ components but is agnostic of how they are implemented. In the next section we describe the network classes.

1 Real Time Network *Base* Classes

At present, there are 4 types of objects that can exist in a NetClamp network: cells, synapses, currents and electrodes. While the first three are obvious and usual, electrodes are objects that can inject arbitrary currents into cells. With electrodes, a network of model cells can be stimulated similar to a network of biological cells, and resulting behavior examined. A fifth element not yet implemented is a mechanism for inter-current communications, such as would occur with a calcium dependent potassium current.

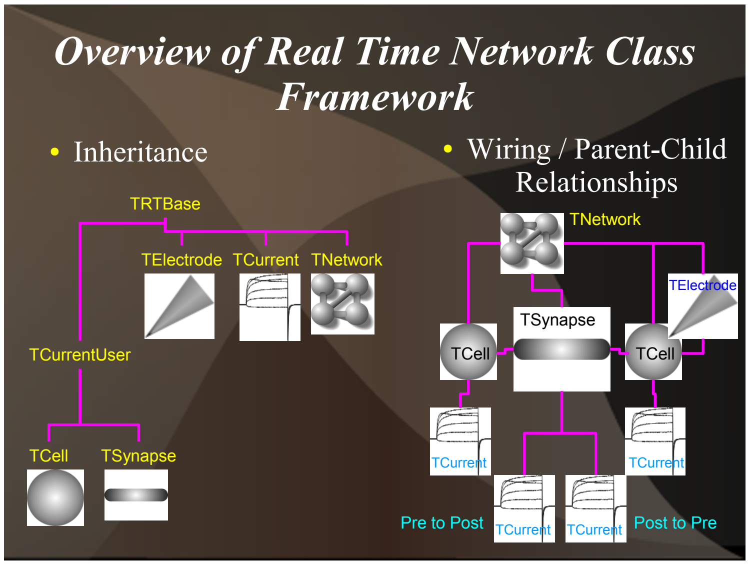


Figure 1: Diagrams of the Inheritance and Wiring relationships among the framework classes

To orient the reader, a bit of C++ and object-oriented programming jargon must be introduced. Classes begin with a capital T, to denote a *Type*. Classes may be derived from a base class, and in doing so, inherit the data and methods (functions) associated with that class. Pure virtual methods are placeholders for derived classes to add behaviors. A class containing pure virtual functions is considered *abstract*. Abstract classes provide an *interface* that is common to all derived classes. When creating a class framework such as pictured in Figure 1, the programmer considers which behaviors or methods should be common among some or all classes in that framework, and introduces an abstract or base class to capture that commonality. By doing so, other parts of the program can interact with classes derived from an abstract class without direct knowledge of the specific kind of derived object they are dealing with. This mechanism for generalization or abstraction is what makes object-oriented programming so appealing.

1.1 TRTBase

TRTBase is the pure virtual base class for all other classes in the network. It sets up streaming of network components using the boost::serialization[1] library. In addition, TRTBase provides an interface for communication between the GUI and the network classes. Each derived network component should own a GUI "form" that allows manipulation of the parameters that define that component. For example, a model cell should have parameters for capacitance and the potential at which the simulation/calculations begin, and the form that provides the interface for the cell should have edit fields for capacitance and starting potential.

As most programming environments have GUI building blocks that are based on the idea of forms, TRTBase provides a generic interface that can be used in numerous environments. To that end, there are 3 pure virtual functions that derived classes must override to implement GUI behaviors.

- **virtual void __fastcall** PopulateEditForm() = 0;
  Interface for derived classes, called by GUI to update form edit components.
- **virtual bool __fastcall** ValidateEditForm() = 0;
  Interface for derived classes, called by GUI to read form values and check for appropriate input.
- **virtual void** $*$**const __fastcall** GetEditForm() = 0;
  Returns a pointer to the form, usually called to display the form at the appropriate time.

[1]see boost.org for more information

A 4th pure virtual function is **virtual bool __fastcall** Initialize () = 0;, which provides an interface for resetting before running the simulation or dynamic clamp experiment.

It should be noted that most, if not all, derived classes will expose a method, *Update()*, that is called to calculate the current, given time and voltage parameters. The information needed for each object's *Update()* function varies; since a single function signature could not be appropriate for all object types, each type introduces its own *Update()* method. TRTBase does not have an *Update()* method. In each network object, *Update()* is a public method that calls a protected pure virtual method, *DoUpdate()*. Usually it is sufficient to override *DoUpdate()* alone, since *Update()* will simply check if a network object is set to Active, and then call *DoUpdate()* accordingly.

1.2 TCurrent

TCurrent is an abstract base class from which all currents must be derived. Users wishing to implement new behavior most often need only subclass TCurrent, providing a concrete class that overrides the pure virtual method of TCurrent:

- **virtual double __fastcall** DoUpdate(**double** step, **double** Vkin, **double** Vdrv) = 0;

DoUpdate() is called by the public method *TCurrent::Update()*, and takes three parameters. First is *step*, which is the number of milliseconds passed since last call. The second is *Vkin*, which is the voltage governing kinetics of conductance. The third is *Vdrv*, which is the voltage governing ionic flow through conductance. The two voltage parameters allow the same current class to serve as an intrinsic cellular current as well as a synaptic current whose kinetics are determined by the presynaptic voltage and postsynaptic driving force (see Section 2.1 below).

1.3 TCurrentUser

TCurrentUser is a base class for TCell and TSynapse, and provides methods common to both, such as *AddCurrent()* and *RemoveCurrent().*

1.4 TCell

TCell is an abstract base class from which all cells must be derived. TCell owns one array each of Currents, Electrodes, and Synapses. TCell's Update(**double** step) member function calls the *Update()* functions of its Currents, Electrodes, and Synapses in a loop, summing each contribution to calculate the total current and storing the value. Derived classes normally do not need to override *TCell::Update()*;

*TCell::Update(**double** step)* takes only one parameter - the time since the last call (in ms). Because a cell's voltage can be calculated (in a simulation) or sampled (in dynamic clamp), the membrane voltage must be set by one of two methods before the call to *Update()*. The three pure virtual methods that determine the appropriate function calls are:

- **virtual bool __fastcall** IsVoltageDependent () = 0;
  Used to determine if cell voltage is sampled (true) or calculated (false). If true, then the GUI must handle the network as a dynamic clamp experiment.
- **virtual double __fastcall** SetVm (**double** Vm) = 0;
  Sets the cell's membrane voltage to value sampled from DAQ part of program.
- **virtual double __fastcall** CalcVm (**double** step) = 0;
  Calculates membrane voltage according to an appropriate algorithm.

1.5 TSynapse

TSynapse is an abstract base class from which all cells must be derived. It is a base class for current containers that facilitate communication between cells. TSynapse is bidirectional by design and has two arrays of TCurrents, accessed by *PreToPostCurrents()* and *PostToPreCurrents()*. Arrays were chosen rather than single TCurrents, because it is possible that a single neurotransmitter activates multiple postsynaptic receptors or a synapse might co-release transmitters and peptides or some other effector. TSynapse owns pointers to the pre- and postsynaptic TCells, accessed through *Pre()* and *Post()*.

TSynapse's Update method is called by the each of the cells that it connects.

- **double __fastcall** Update(TCell ∗ **const** cell, **double** step);

Depending on the identity of the *cell* parameter (either *Pre()* or *Post()*), the *Update()* methods of either the *PreToPostCurrents()* or the *PostToPreCurrents()* are called and their sum returned (see Listing 1).

1.6 TNetwork

TNetwork owns and organizes all other classes and provides text based methods for creating, deleting, connecting and modifying network components. There is no need to subclass TNetwork.

TNetwork exposes an *Update()* method that takes four parameters and returns a pointer to the fourth:

- **double**∗ **__fastcall** Update(**double** step, **double**∗ Vm_in, **double**∗ Vm_out, **double**∗ I_nA);

The *step* is the time (in ms) since the last call, *Vm_in* is the array of sampled cell voltages for this time point, *Vm_out* is the array of voltages of all cells in the network, and *I_nA* is the array of scaled voltages for injecting currents. *Vm_in* and *I_nA* can be empty if the network is in simulation mode (no dynamic clamping). Within the *TNetwork::Update()* method, the *Update()* methods of all cells in the network members are called and their currents stored in *I_nA* (see Listing 1). TNetwork also overrides and exposes an *Initialize()* method, which must be called before calling *TNetwork::Update()*. *TNetwork::Initialize* calls the *Initialize()* method of all network components.

Using TNetwork in a simulation or dynamic clamp experiment

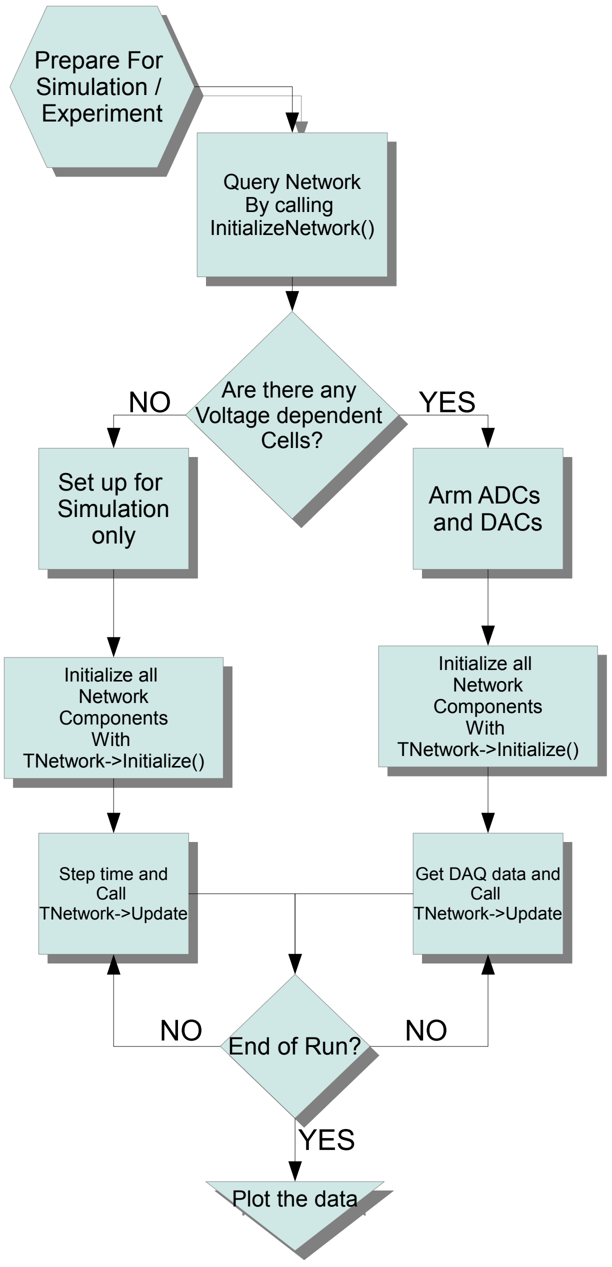
See the following flowchart which diagrams the order of function calls when running a simulation

1. *TNetwork->InitializeNetwork()* to differentiate simulation (all model cells) versus dynamic clamp (some voltage dependent cells).

2. *TNetwork->Initialize()*, to initialize all network components

3. *TNetwork->Update()*, repeatedly called to advance the network state and calculate currents if necessary

Figure 2: Flowchart for TNetwork Method Calls


Prepare For Simulation / Experiment
Query Network By calling InitializeNetwork()
Are there any Voltage dependent Cells?
NO
YES
Set up for Simulation only
Arm ADCs and DACs
Initialize all Network Components With TNetwork->Initialize()
Initialize all Network Components With TNetwork->Initialize()
Step time and Call TNetwork->Update
Get DAQ data and Call TNetwork->Update
NO
End of Run?
NO
YES
Plot the data

To promote a better understanding of the order of operations when using *TNetwork::Update()*, a pseudocode program listing is provided.

Listing 1 shows the order of operations for calculating the cell voltages and currents in a network. The *Update()* method of each component is either called directly by *TNetwork->Update()*, or within the *Update()* method of the synapse or cell.

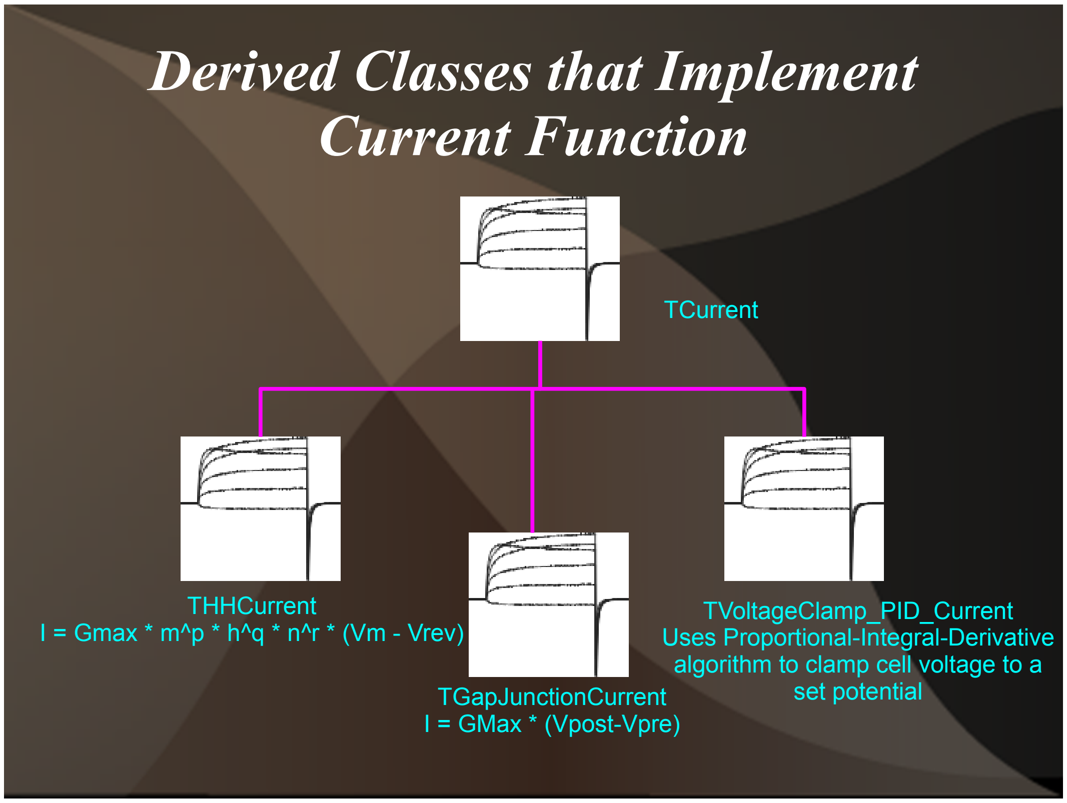
Listing 1: Pseudocode implementation of *TNetwork::Update()*

```
TNetwork->Update(step /*ms*/, *Vm_in /*Volts*/, *Vm_out /*mV*/, *I_nA /*Volts*/)
   VmArray = for each VmDepCells->SetVm // DAC Volts covert to mV in SetVm()
   VmArray = for each TimeCells->CalcVm
   For each cell:
      Cells->Update(step)
         Icell = 0;
         For each intrinsic current
            Icell = Icell - Current->Update( step, Vm, Vm );   // subtract
         For each electrode
            Icell = Icell + Electrode->Update( step );         // add
         For each synapse
            Icell = Icell - Synapse->Update( thiscell, step);   // subtract
               Isyn = 0
               For each synaptic current
                  If thiscell is Postsynaptic
                     Isyn += PreToPostCurrs-> Update( step, Pre->Vm, Post->Vm )
                  Else if thiscell is Presynaptic
                     Isyn += PostToPreCurrs-> Update( step, Post->Vm, Pre->Vm )
```

2 Real Time Network *Derived* Classes

2.1 Derived from TCurrent

Figure 3: Classes derived from TCurrent


Derived Classes that Implement Current Function
TCurrent
THHCurrent
I = Gmax * m^p * h^q * n^r * (Vm - Vrev)
TGapJunctionCurrent
I = GMax * (Vpost-Vpre)
TVoltageClamp_PID_Current
Uses Proportional-Integral-Derivative algorithm to clamp cell voltage to a set potential

THHCurrent

There are a total of **7 parameters** to set. THHCurrent Uses 3 voltage-dependent kinetic parameters (or gates), ***m(V)***, ***h(V)***, and ***n(V)***, and 3 exponents, ***p***, ***q***, and ***r***, to determine the fraction of **Gmax** available to conduct current. The inclusion of a third voltage dependent gate, ***n(V)***, differs from standard Hodgkin-Huxley type implementations; however, it increases flexibility by allowing two separate activation or inactivation gates with different kinetics. Current is calculated according to

$$I = G_{MAX} * m(V)^p * h(V)^p * n(V)^r * (V_m - V_{rev}) \tag{1}$$

For each voltage dependent parameter ($m(V)$, $h(V)$, or $n(V)$) there are seven separate parameters that define its voltage dependence. The equations that define each parameter ($y(V)$) are:

$$\frac{dy(t,V)}{dt} = \frac{y_{ss}(V) - y(t,V)}{y_\tau(V)} \tag{2}$$

$$y_{ss}(V) = \frac{1 - SS_{min}}{(1 + \exp(\frac{V - V_0}{k}))^w} + SS_{min} \tag{3}$$

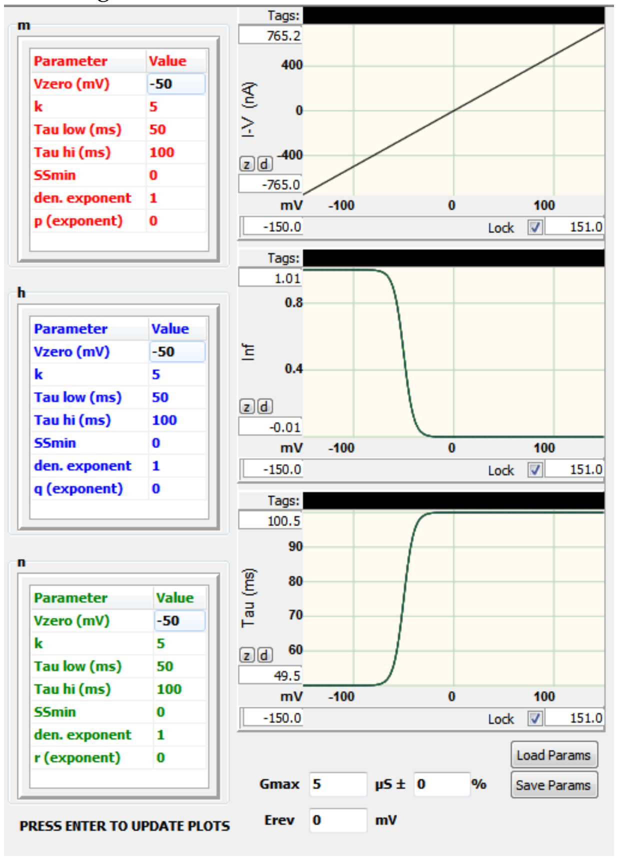
$$y_\tau(V) = \frac{\tau_{low} - \tau_{hi}}{(1 + \exp(\frac{V - V_0}{|k|}))^w} + \tau_{hi} \tag{4}$$

Equation 3 represents the steady state activity of the gates (from 0 to 1) as a function of voltage. 4 represents the time constant of the relaxation to the steady state value at that voltage. Please note that in Equation 4 the absolute value of k is used. This provides consistent interface to the $\tau_{low}$and τ_{hi} parameters, which are interpreted as the time constants to the left of and to the right of V_0, respectively.

Table 1: Parameters for THHCurrent

Parameter Name	Values	Notes
V_0(Vzero)	-150..150 mV	midpoint for sigmoid function
k	-20..20	dimensionless term for sigmoid steepness
τ_{low}	0..∞ ms	time constant for relaxation to steady state (left of V_0)
τ_{hi}	0..∞ ms	time constant for relaxation to steady state (right of V_0)
SS_{min}	0..1	minimum steady state value
w (den. exponent)	0,1,2,3...	another way to change the steepness of the sigmoid functions
p, q, r (exponent)	0,1,2,3...	Cooperativity of gates: if 0, gate is not voltage dependent

Figure 4: GUI Interface for THHCurrent



m

Parameter	Value
Vzero (mV)	-50
k	5
Tau low (ms)	50
Tau hi (ms)	100
SSmin	0
den. exponent	1
p (exponent)	0

h

Parameter	Value
Vzero (mV)	-50
k	5
Tau low (ms)	50
Tau hi (ms)	100
SSmin	0
den. exponent	1
q (exponent)	0

n

Parameter	Value
Vzero (mV)	-50
k	5
Tau low (ms)	50
Tau hi (ms)	100
SSmin	0
den. exponent	1
r (exponent)	0

PRESS ENTER TO UPDATE PLOTS

Gmax 5 µS ± 0 %

Erev 0 mV

TGapJunctionCurrent

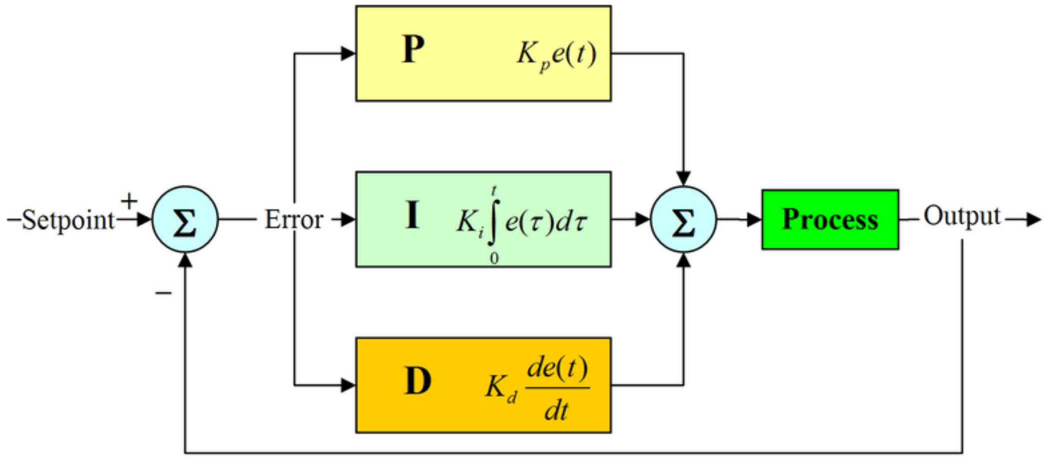
Uses the voltage difference between pre- and postsynaptic cells to calculate current. The only parameter is **Gmax**, in μS.

TVoltageClamp_PID_Current

This current is based on an article entitled "PID_without_a_PhD".

From wikipedia and a_sample_book_chapter, "The PID controller calculation (algorithm) involves three separate constant parameters, and is accordingly sometimes called three-term control: the proportional, the integral and derivative values, denoted P, I, and D. Heuristically, these values can be interpreted in terms of time: P depends on the present error, I on the accumulation of past errors, and D is a prediction of future errors, based on current rate of change."

A block diagram describing its function:



When used, this should be the only current in a cell. There are **7 parameters** to set:

Vcommand - the desired potential that the cell will hold.

Imax and **Imin** - the maximal and minimal limits on the integral term calculation.

Pgain, **Igain**, and **Dgain** - the respective gains of the **P**roportional, **I**ntegral and **D**ifferential components of the calculation.

tau - an added stability term to slow the predictive effects of the **D** term.

It is advised that tests for parameters (tuning the PID controller) be performed on resistor/capacitor model cells before attempting on live cell.

2.2 Derived from TCell

- TModelCell
  - This cell calculates the voltage after the step by integration of

$$I = C\frac{dV}{dt} \tag{5}$$

  - It takes two parameters - the membrane capacitance in nF and the starting membrane potential in mV.
- TBiologicalCell
  - This cell implements the "dynamically clamped" cell. The user interface for this cell

- TPlaybackCell
  - This cell takes a previously recorded voltage trace and "Plays back" this voltage. Ideally, this cell type is used in a network as a presynaptic input to another cell. The *Update()* method for this cell returns 0 and ignores any intrinsic, synaptic or electrode currents.

3 Data Acquisition Classes

Under construction – for now, DAQ is hardwired to use NIDAQmx functions and is integral to GUI classes.

4 GUI Classes

The GUI has two main forms: TGUINetworkForm, for setting up the network and all parameters, and TGUIRunModelForm for running the simulation/experiment and viewing the results.

4.1 TGUINetworkForm

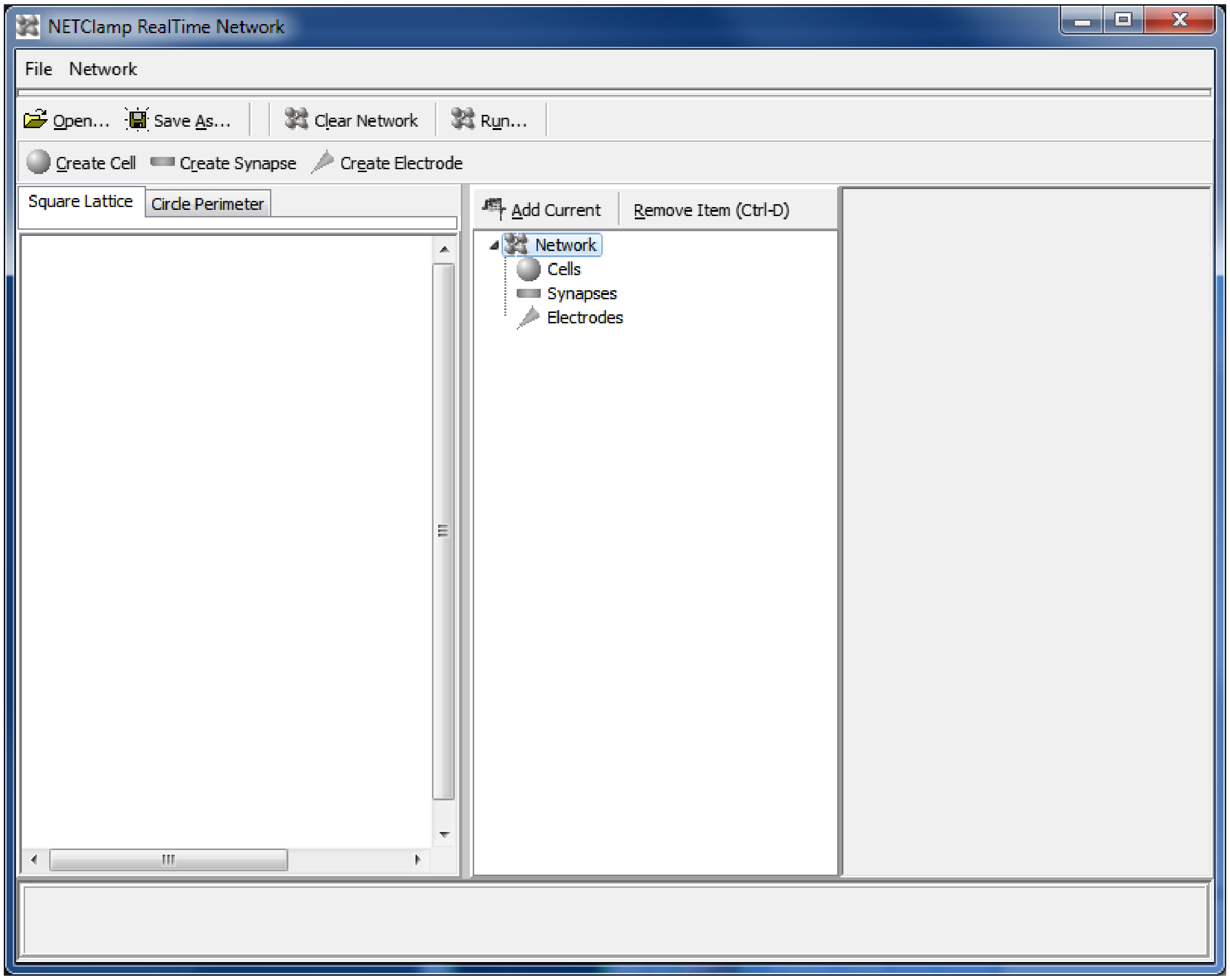
The network setup form consists of 3 vertical panels (see Figure 5). The leftmost panel is where the network is "drawn". There are two arrangements for the network: the Square Lattice (SL) or the Circle Perimeter (CP). When Create Cell is clicked, an image of a red circle appears that can be placed by the mouse pointer. In the SL configuration, the cell is constrained to be placed on a grid. In the CP configuration, the cell can be placed anywhere, and the software decides where to draw it. The software provides two different network arrangements, or views, to accommodate a variety of intercellular connections. The SL configuration is appropriate for networks in which cells are connected only to their nearest neighbors, or adjacent cells. If the cells are connected to distant cells, the SL view is insufficient. To accommodate multiple, distant connections, The CP view places all cells around the perimeter of a circle, and connections are displayed as chords that link up the cells. With this arrangement, each cell can have a connection with every other cell. The idea for the CP configuration was "borrowed" from the SNNAP program from the Neurobiology and Anatomy department at the UT Health Science Center at Houston Medical School.

Placing a cell on either the SL or CP panel opens a dialog box that asks which type of cell should be created. Once cells have been positioned, synapses can be created by clicking Create Synapse. The mouse cursor changes to a star shape. Two cells are chosen in order and a dialog box appears that asks which type of synapse should be created. Electrodes are created by choosing Create Electrode and clicking on the appropriate cell. Again, a dialog box appears that asks which type of electrode should be created. Current creation is handled similarly, but the choice of current container, either cell or synapse, is made in the middle column that holds a heirarchical tree view of the network.

The choices that appear in the dialog box are obtained from a object database. Classes derived from the base classes described in Section 1 register themselves with this database. One of the benefits of the object oriented model used is that the base classes (see Figure 1) and the main GUI classes do not need to know anything about the derived classes: neither their specific implementation nor their unique GUI. When a derived class is coded, the *GetEditForm()* method is overriden to return a pointer to a GUI object that allows editing of that class' parameters. TGUINetworkForm uses this pointer to display the GUI object in the rightmost column when a cell, synapse, current or electrode is created, or when it is chosen in the heirarchical tree view. The agnostic behavior of TGUINetworkForm with respect to the function and GUI of the derived classes means that TGUINetworkForm source code is never altered when a new derived class is added to the application.

The dialog box that allows the user to choose which type of derived class the queries the database which classes are available. Based on the user's choice, a global ***class factory*** is petitioned to create the appropriate object. The details of the class factory's implementation are too involved to be included in this document, but the reader should be aware of its presence and utility. Any new features to be added to the NetClamp software will likely be accomplished by deriving from one of the base classes, and the source code for the derived class must contain the appropriate measures to register the new class with the database and the class factory.

Figure 5: Image of application window upon start-up, displaying TGUINetworkForm.


NETClamp RealTime Network
File Network
Open... Save As... Clear Network Run...
Create Cell Create Synapse Create Electrode
Square Lattice Circle Perimeter
Add Current Remove Item (Ctrl-D)
Network
Cells
Synapses
Electrodes

4.2 TGUIRunModelForm

The TGUIRunModelForm class contains DAQ and GUI code for performing the simulation/experiment (Figure 6). Although the parameter edit labels are appropriate for dynamic clamp experiments, the parameter values also operate for simulations. There are differences between how the parameters are used, however. The parameter meanings for dynamic clamp experiments will be defined first, followed by identification of those parameters whose meaning changes for simulations.

Dynamic Clamp Experiments

- Sample Rate is a request to the DAQ system. Since dynamic clamp experiments rely on DAQ hardware, and Windows is not a real-time operating system, the time between each sample can vary. The software measures the performance of the DAQ system and provides an analysis at the bottom of the form. The histogram shows the number of samples of a given duration. Because the requested rate is not the true sample rate, it can be set as high as possible. The effective sample rate is taken from the inverse of the average time between samples. Experience has shown that with requested sample rates from 100 kHz to 500 kHz, the effective sample rate can be as high as 33 kHz. The effective sample rate is highly dependent on the National Instruments board installed, the operating system, and the CPU type and speed.

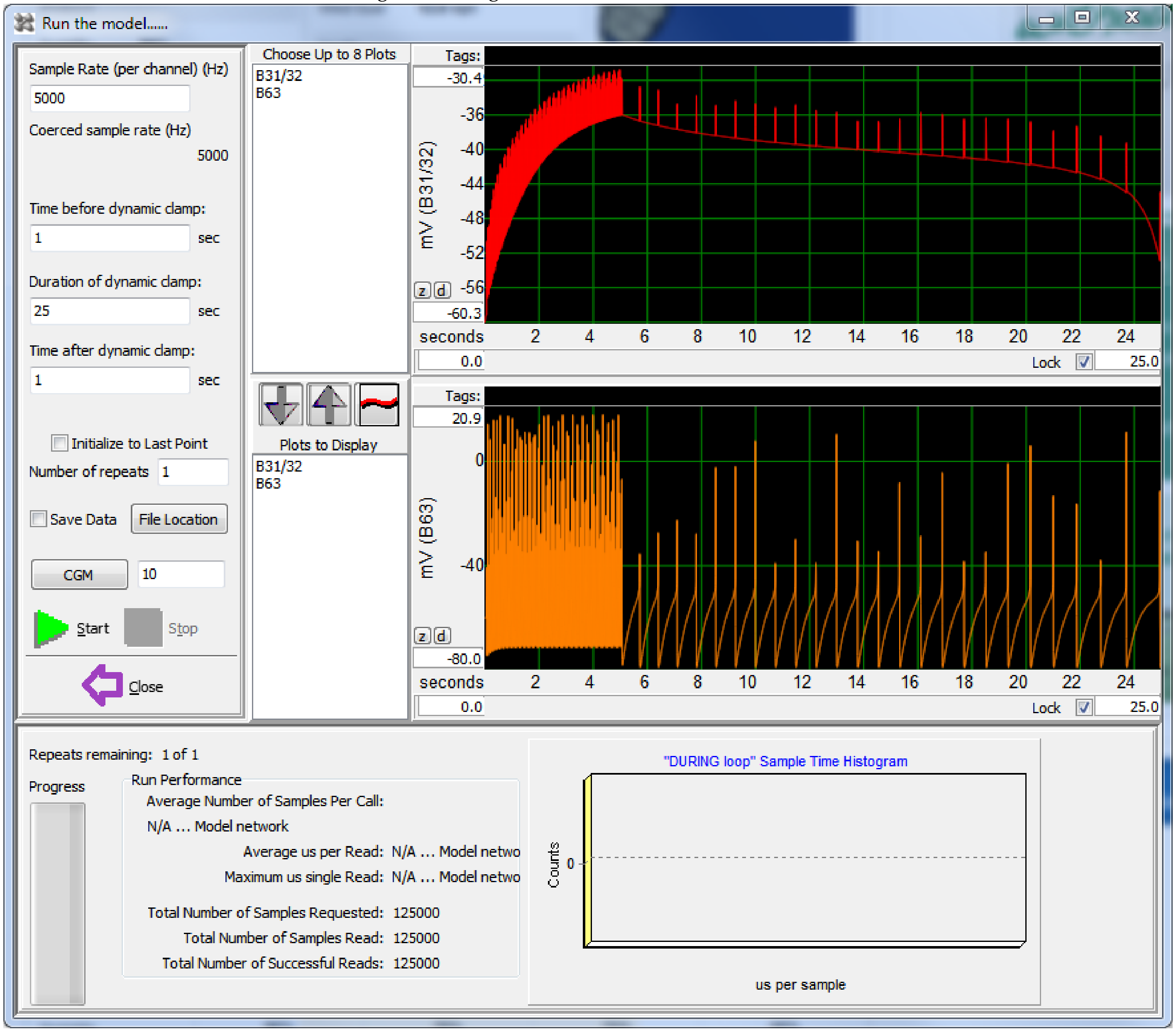
- The times before, during and after the application of dynamic clamping are set by the three edit boxes below the sample rate input. Membrane voltages are sampled and calculated during all 3 periods, but clamping currents are only injected in the middle segment.
- Initialize to Last Point works in conjunction with the Number of Repeats input. If checked, the *Network::Initialize()* is not called between repeats. This behavior allows continuity between the end of one run and the beginning of the next, without resetting all integration values to their initial state.
- Save Data allows saving of all channel voltages in a tab delimited text file for import into other analysis programs. If checked, and a suitable file name/path given, a file with columns for each cell will be written after the data is plotted to the screen.
- CGM and the adjacent input box export the voltage traces as Computer Graphics Metafiles, with scale bars included. This eases construction of figures.
- Up to 8 plots can be displayed at once. The list of all available cells is given in the top box, and the choices are displayed below. The two arrow buttons and the tilde (~) are used to move plots to and from, or clear the "displayed" list.

Simulation Experiments

For when no biological cell's membrane voltages are sampled, the:

- Sample Rate assigned for a simulation sets the precise step size for each integration step, and
- Time Before and Time After are ignored

Figure 6: Image of TGUIRunModelForm



Run the model......

Sample Rate (per channel) (Hz): 5000

Coerced sample rate (Hz): 5000

Time before dynamic clamp: 1 sec

Duration of dynamic clamp: 25 sec

Time after dynamic clamp: 1 sec

Initialize to Last Point

Number of repeats: 1

Save Data

CGM: 10

Choose Up to 8 Plots: B31/32, B63

Plots to Display: B31/32, B63

Repeats remaining: 1 of 1

Run Performance
Average Number of Samples Per Call:
N/A ... Model network
Average us per Read: N/A ... Model netwo
Maximum us single Read: N/A ... Model netwo
Total Number of Samples Requested: 125000
Total Number of Samples Read: 125000
Total Number of Successful Reads: 125000