Spiking Spielwiese: an Izhikevich spiking neuron C++ library version 0.1

Zachary Hutchinson
email: zachary.s.hutchinson@maine.edu

November 2018

Abstract

The following document describes the Spiking Spielwiese [SPSP] library. The *Introduction* gives a brief overview of the library and its reason for existence. Next, *Getting Started* demonstrates how to use the library through short examples. *Classes and Methods* describes SPSP in more detail. *Examples* discusses the larger examples included with the library.

1 Introduction

1.1 Background

Spiking neuron models encode information through the timing of their action potentials (spikes) and for this reason are referred to as the third generation of artificial neuron models [4]. Since behavior depends on the strong integration of time and an input signal, spiking neurons are an attractive choice for handing time-dependent responses to an environment [1, 2, 5, 6]. In spite of their allure, real-world applications utilizing spiking neurons are relatively scarce compared to those powered by their second generation cousins. The reasons for this discrepancy are many and include the difficulty in training networks, the dearth of general purpose tools, their computational cost and a lack of understanding how complex biological networks encode information in spike trains.

1.2 Description

Spiking Spielwiese [SPSP] is a small C++ library implementation of the Izhikevich spiking neuron model designed for projects outside spiking neural network [SNN] or computational neuroscience research. It is hoped that SPSP's shallow learning curve will open the spiking neuron model to a wider audience. Unlike other SNN tools (NEST or Brian), SPSP can be dropped into nontraditional

projects allowing neural networks to exist alongside or within non-network components, providing input to and receiving output from the rest of the project. SPSP focuses on small components, neurons and synapses, and makes as few assumptions as possible about desired network topology.

The majority of Izhikevich, single compartment neuron templates are included as given in [3]. Synapse types provide a range of connection properties from the simple weight-scaling of output to supervised spike-time dependent plasticity.

More than a library for actual projects, SPSP is an introductory tool for grasping how spiking neurons behave. More than the plug-and-code benefits of any implementation or library, SPSP exists to make it easier to understand how neuron models behave. Understanding behavior involves a lot of play, hence the playground reference in the name. Spiking neural networks, being a dynamical system, suffer from the Goldilocks dilemma evident everywhere in nature. Weak input and connections and the network will be too cold (no activity). Strong input and connections and it will be too hot (runaway activity). But there is a narrow zone in which the network designer can find an infinite number of patterns and behavior.

2 Getting Started

2.1 Overview

Spiking Spielwiese consists of two primary elements: Neurons and Synapses. The Neuron class is an implementation of the Izhikevich neuron model. Izhikevich neurons depend on nine constants for their behavior. These constants and several others unique to this implementation are provided at instantiation via neuron templates. SPSP provides the constants to neuron types common to neuroscience research; however, these are included to get users started along lines of known behavior. Since SPSP was designed to bring spiking neurons to the larger world, users are encouraged to experiment and create new templates.

Neuron objects receive input and pass output through Synapse objects. As stated earlier, Synapses can connect Neurons to other Neurons or to non-neuronal components of a larger program. The Synapse class is a non-virtual interface whose client is the Neuron object. Derived Synapse classes provide sylized functionality .

2.2 One-neuron Network

First, we must instantiate a *NeuronTemplates* object and we require only one. NeuronTemplates stores NT objects, which are the actual neuron templates, and loads a default set in its constructor.

```
1       // Holds all NTs.
2       spsp::NeuronTemplates nt;
```

To create Neuron objects, we give a pointer to an NT object as argument to the Neuron constructor. In the example below we ask the NeuronTemplates

object for a pointer to the *RegularSpiking* NT object (which models a typical excitatory cortical neuron) and use it to instantiate a new Neuron object. We do this by passing in the template's name ("RegularSpiking") to the *GetNeuronTemplate* method.

```
1       // Create neuron
2       spsp::Neuron neuron(nt.GetNeuronTemplate("RegularSpiking"));
```

Next we need a way to send input to the neuron and get its output. This is accomplished via implementations of the Synapse interface. For this example we will use SimpleSynapse which scales the *signal* by a *weight*. Let's create one for the input and and one for the output of our neuron. It is necessary to create them wrapped in *std::shared_ptr*s because this is how they are stored by the Neuron object. And we will use the SimpleSynapse constructor which accepts one argument, the synaptic weight.

```
1       // Input and output synapses
2       std::shared_ptr<spsp::Synapse> syn_input =
3           std::make_shared<spsp::SimpleSynapse>(10.0);
4
5       std::shared_ptr<spsp::Synapse> syn_output =
6           std::make_shared<spsp::SimpleSynapse>(10.0);
```

Note that our SimpleSynapses are stored in a base class pointer for the convenience of storing them in Neuron objects. In a network environment, most internal Synapses will never need to be downcasted. Other situations might require getting at specialized implementation details. I suggest using dynamic_cast-ing to test for derived type.

Now that we have two synapses, let's connect them to the Neuron.

```
1       // Connect Synapses to Neuron
2       neuron.AddInputSynapse(syn_input);
3       neuron.AddOutputSynapse(syn_output);
```

As you can see, it is the Neuron object that store pointers to the Synapse objects. All communication is unidirectional; Synapses know nothing about the outside world. This is necessary for Synapses to facilitate communication between neuronal and non-neuronal components.

Thus far, we have connected the three components, but we have not provided any input. Let us assume input is a constant value of 10.0. Set the signal of the input synapse. Given that we set the weight of the input synapse to 10.0, when the neuron calls for input from this synapse it will get $signal * weight$, or 100.0. A constant input of 100 will cause the neuron to spike.

```
1       // Constant input of 10.0
2       syn_input->SetSignal(10.0);
```

All looks good, but nothing will happen unless we tell the Neuron object to update its state. A call to a neuron's *Update* method causes it to sum its inputs, update the values for v and u, check whether the current values cause an action potential, and send its current output to all outgoing synapses.

Lastly, we create a simple update loop that runs for 1000 milliseconds. In each iteration, the Neuron is updated and its current output is printed to the console followed by the signal of the outgoing synapse. Note, a program using SPSP requires a *time* variable. Several of the Synapse types maintain a history of recent activity which is necessary for ablities such as spike-time dependent plasticity [STDP].

```
    // Time in milliseconds
    uint64_t time = 0;

    for(int i = 0; i < 1000; i++) {

        // Update the neuron
        neuron.Update(time);

        // Output
        std::cout << "Time: " << time << "   "
            << std::setw(12) << neuron.GetCurrentOutput() << "   "
            << std::setw(12) << syn_output->GetSignal()
            << std::endl;

        // Time forward.
        time++;
    }
```

Putting it all together. NOTE: All code examples in this document can be found on the github page under *documentation/code_examples*. To try them out, copy them to a directory of your choice, copy the libspsp files to the same directory and build with your compiler of choice.

```

#include<iostream>
#include<memory>
#include<cstdint>
#include<iomanip>

#include"NTemplate.hpp"
#include"Synapse.hpp"
#include"Neuron.hpp"

int main(int argc, char**argv) {

    // Holds all NTs.
    spsp::NeuronTemplates nt;

    // Create neuron
    spsp::Neuron neuron(nt.GetNeuronTemplate("RegularSpiking"));

    // Input and output synapses
    std::shared_ptr<spsp::Synapse> syn_input =
        std::make_shared<spsp::SimpleSynapse>(10.0);

    std::shared_ptr<spsp::Synapse> syn_output =
        std::make_shared<spsp::SimpleSynapse>(10.0);

    // Connect Synapses to Neuron
    neuron.AddInputSynapse(syn_input);
    neuron.AddOutputSynapse(syn_output);

    // Constant input of 10.0
    syn_input->SetSignal(10.0);

    // Time in milliseconds
    uint64_t time = 0;

    for(int i = 0; i < 1000; i++) {

        // Update the neuron
        neuron.Update(time);

        // Output
        std::cout << "Time: "
            << std::setw(4) << time << "   "
            << std::setw(12) << neuron.GetCurrentOutput() << "   "
            << std::setw(12) << syn_output->GetSignal()
            << std::endl;

        // Time forward.
        time++;
    }

    return 0;
}
```

If you scroll back through the output, you should see that once the neuron begins to spike (92 ms), the neuron spikes every 77 ms. Here's mine:

```
Time:    92            1            5
Time:    93     0.735759      4.23883
Time:    94     0.406006      2.88765
Time:    95     0.199148      1.66075
```

```
Time:    96      0.0915782       0.838952
Time:    97      0.0404277       0.388568
Time:    98      0.0173513       0.170553
Time:    99     0.00729506      0.0724222
Time:   100     0.00301916      0.0301008
Time:   101      0.0012341      0.0123258
```

The third column is the neuron's output. Notice that effects of a spike at time 92 are smeared. The fourth column is the neuron's output as scaled by the outgoing synapse. Curious! If you recall, we set the weight of the output synapse to 10.0, so why is it half that? I will explain this in detail later, but suffice now to say that neuron's normalize their output before sending it to the efferent synapses. Above, an action potential is smeared over 10 ms. It is possible to smear action potentials further and drive neurons harder so that spikes overlap. Their effects are additive. This is an arbitrary choice for default behavior to lower the threshold for runaway behavior.

Normalizing happens in the Neuron object. Setting a Synapse's signal directly as we do for the input is direct. Also, it is important to stress that only Neuron's have a dynamic state (or Update method). Synapse signals do not change unless they are tied to a Neuron object whose Update method gets called, or if the signal is set manually using SetSignal.

2.3 Two-neuron Network

Now let's build on the previous example to show how two neurons can be connected together. How to accomplish might be rather evident from the previous example alone; however, what is not evident (and where most SNN tribulations are unearthed) is behavior. For the serious approach, this subsection will have more insight into the latter than the former.

I will layout the snippets that differ and offer the full code at the end.

Obviously, we need another neuron. I've renamed the old one *neuronA* and given it a partner *neuronB*. Take a moment to consider how they will be connected as SNN information flows in one direction. For this example, *neuronA* will drive *neuronB*. Or,

$$Input -> NeuronA -> NeuronB -> Output \tag{1}$$

```
// Create neurons
spsp::Neuron neuronA(nt.GetNeuronTemplate("RegularSpiking"));
spsp::Neuron neuronB(nt.GetNeuronTemplate("RegularSpiking"));
```

In the crude graph above, there are three connecting (or synaptic) arrows. Each requires a Synapse object. We have two from the previous example, but we need a third to connect the two neurons.

```
std::shared_ptr<spsp::Synapse> syn_inter =
    std::make_shared<spsp::SimpleSynapse>(1000.0);
```

What?! Wait! A weigth of 1000? Is that necessary? That's a 10^2 increase!

In the earlier example all we needed was 10 for the input synapse because the signal is continuous. This synapse connects A to B. If you compiled and ran the one-neuron example, you saw that most of the time the output synapse's signal

is 0, separated by bursts of 10 ms activity. Activity can be sparse. For demonstration purposes we'll leave the other two synapses unchanged and increase the input signal strength to get a response from the second neuron, B.

In the snipped below, connectivity becomes more complicated. And we raise the input signal strength to 30. Again, since the weight of *syn_input* is 10, the real value driving *neuronA* is 300. This is enough to cause a RegularSpiking neuron to spiking once every 10 ms.

```
    // Connect Synapses to Neuron
    neuronA.AddInputSynapse(syn_input);
    neuronA.AddOutputSynapse(syn_inter);
    neuronB.AddInputSynapse(syn_inter);
    neuronB.AddOutputSynapse(syn_output);

    // Constant input of 20.0
    syn_input->SetSignal(30.0);
```

Finally, we need to make sure we call Update on both neurons. This is included in the full code below.

```
#include<iostream>
#include<memory>
#include<cstdint>
#include<iomanip>

#include"NTemplate.hpp"
#include"Synapse.hpp"
#include"Neuron.hpp"

int main(int argc, char**argv) {

    // Holds all NTs.
    spsp::NeuronTemplates nt;

    // Create neurons
    spsp::Neuron neuronA(nt.GetNeuronTemplate("RegularSpiking"));
    spsp::Neuron neuronB(nt.GetNeuronTemplate("RegularSpiking"));

    // Input, intermediate, and output synapses
    std::shared_ptr<spsp::Synapse> syn_input =
        std::make_shared<spsp::SimpleSynapse>(10.0);

    std::shared_ptr<spsp::Synapse> syn_inter =
        std::make_shared<spsp::SimpleSynapse>(1000.0);

    std::shared_ptr<spsp::Synapse> syn_output =
        std::make_shared<spsp::SimpleSynapse>(10.0);

    // Connect Synapses to Neuron
    neuronA.AddInputSynapse(syn_input);
    neuronA.AddOutputSynapse(syn_inter);
    neuronB.AddInputSynapse(syn_inter);
    neuronB.AddOutputSynapse(syn_output);

    // Constant input of 30.0
    syn_input->SetSignal(30.0);

    // Time in milliseconds
    uint64_t time = 0;

    for(int i = 0; i < 1000; i++) {

        // Update the neuron
        neuronA.Update(time);
        neuronB.Update(time);

        // Output
        std::cout << "Time: "
            << std::setw(4) << time << " --- A: "
            << std::setw(12) << neuronA.GetCurrentOutput() << " "
            << std::setw(12) << syn_inter->GetSignal() << " --- B: "
            << std::setw(12) << neuronB.GetCurrentOutput() << " "
            << std::setw(12) << syn_output->GetSignal()
            << std::endl;

        // Time forward.
        time++;
    }

    return 0;
}
```

As I mentioned above, *neuronA* will spike every 10 ms. *neuronB* on the other hand will spike every 100 ms. Below is partial output from the two-neuron example showing the first spike of *neuronB*.

```
Time:  111 --- A:          0        0 --- B:          0          0
Time:  112 --- A:          1      500 --- B:          0          0
Time:  113 --- A:   0.735759  423.883 --- B:          0          0
Time:  114 --- A:   0.406006  288.765 --- B:          0          0
Time:  115 --- A:   0.199148  166.075 --- B:          1          5
Time:  116 --- A:  0.0915782  83.8952 --- B:   0.735759    4.23883
Time:  117 --- A:  0.0404277  38.8568 --- B:   0.406006    2.88765
Time:  118 --- A:  0.0173513  17.0553 --- B:   0.199148    1.66075
Time:  119 --- A: 0.00729506  7.24222 --- B:  0.0915782   0.838952
Time:  120 --- A: 0.00301916  3.01008 --- B:  0.0404277   0.388568
Time:  121 --- A:  0.0012341  1.23258 --- B:  0.0173513   0.170553
Time:  122 --- A:          0        0 --- B: 0.00729506  0.0724222
Time:  123 --- A:          0        0 --- B: 0.00301916  0.0301008
Time:  124 --- A:          0        0 --- B:  0.0012341  0.0123258
Time:  125 --- A:          0        0 --- B:          0          0
```

More than just an example of how to wire two neurons together, this example demonstrates a single Neuron has to work hard to cause a few spikes in another Neuron downstream. Without incoming current, the voltage variable of an Izhikevich neuron decays toward a resting value. A spike happens only if the voltage reaches a prescibed value (see *vpeak* in the NeuronTemplates class). Therefore, the time of incoming spikes is far more important than an overall rate. Three spikes in close succession might drive a neuron to spike when 50 spread out of 1000 ms do not.

2.4 Many-to-One Example

Next we'll demonstrate connecting two *presynaptic* neurons to one *postsynaptic* neuron. In other words, two neurons, each driven by a constant input, will provide separate input to a third. I will give you the whole code and then highlight a few points.

```
#include<iostream>
#include<memory>
#include<cstdint>
#include<iomanip>

#include"NTemplate.hpp"
#include"Synapse.hpp"
#include"Neuron.hpp"

int main(int argc, char**argv) {

    // Holds all NTs.
    spsp::NeuronTemplates nt;

    // Create neurons
    spsp::Neuron neuronA(nt.GetNeuronTemplate("RegularSpiking"));
    spsp::Neuron neuronB(nt.GetNeuronTemplate("RegularSpiking"));
    spsp::Neuron neuronC(nt.GetNeuronTemplate("RegularSpiking"));

    // Input, intermediate, and output synapses
    std::shared_ptr<spsp::Synapse> syn_inputA =
        std::make_shared<spsp::SimpleSynapse>(10.0);
    std::shared_ptr<spsp::Synapse> syn_inputB =
        std::make_shared<spsp::SimpleSynapse>(10.0);

    std::shared_ptr<spsp::Synapse> syn_interA =
        std::make_shared<spsp::SimpleSynapse>(500.0);
    std::shared_ptr<spsp::Synapse> syn_interB =
        std::make_shared<spsp::SimpleSynapse>(500.0);

    std::shared_ptr<spsp::Synapse> syn_output =
        std::make_shared<spsp::SimpleSynapse>(10.0);

    // Connect Synapses to Neuron
    neuronA.AddInputSynapse(syn_inputA);
    neuronA.AddOutputSynapse(syn_interA);
    neuronB.AddInputSynapse(syn_inputB);
    neuronB.AddOutputSynapse(syn_interB);
```

```
39      neuronC.AddInputSynapse(syn_interA);
40      neuronC.AddInputSynapse(syn_interB);
41      neuronC.AddOutputSynapse(syn_output);
42
43      // Constant input of 30.0
44      syn_inputA->SetSignal(30.0);
45      syn_inputB->SetSignal(30.0);
46
47      // Time in milliseconds
48      uint64_t time = 0;
49
50      for(int i = 0; i < 1000; i++) {
51
52          // Update the neuron
53          neuronA.Update(time);
54          neuronB.Update(time);
55          neuronC.Update(time);
56
57          // Output
58          std::cout << "Time: "
59              << std::setw(4) << time << " --- A: "
60              << std::setw(12) << syn_interA->GetSignal() << " --- B: "
61              << std::setw(12) << syn_interB->GetSignal() << " --- C: "
62              << std::setw(12) << syn_output->GetSignal()
63              << std::endl;
64
65          // Time forward.
66          time++;
67      }
68
69      return 0;
70  }
```

neuronA and *neuronB* provide input to *neuronC*. Notice that the constant signal to A and B is the same and we halved the weight of each *inter* synapse. This will produce an identical spiking pattern to the one above for the two-neuron network. I suspect, if you've made it this far, you'll find this example boring. More code, more obejcts, doing the same work. So let's give it a peculiarity by setting one input signal slightly different. How does a signal strength of 31 for *syn_inputB*? Excellent. I'll provide this adjusted code in the examples.

Here is the output from the many-to-one example where *inputB* has a signal strength of 31 showing the first spike of *neuronC*.

```
 1  Time:  107 --- A:         0 --- B:         0 --- C:         0
 2  Time:  108 --- A:         0 --- B:       250 --- C:         0
 3  Time:  109 --- A:         0 --- B:   211.942 --- C:         0
 4  Time:  110 --- A:         0 --- B:   144.383 --- C:         0
 5  Time:  111 --- A:         0 --- B:   83.0374 --- C:         0
 6  Time:  112 --- A:       250 --- B:   41.9476 --- C:         0
 7  Time:  113 --- A:   211.942 --- B:   19.4284 --- C:         0
 8  Time:  114 --- A:   144.383 --- B:   8.52767 --- C:         0
 9  Time:  115 --- A:   83.0374 --- B:   3.62111 --- C:         0
10  Time:  116 --- A:   41.9476 --- B:   1.50504 --- C:         0
11  Time:  117 --- A:   19.4284 --- B: 0.616288 --- C:         0
12  Time:  118 --- A:   8.52767 --- B:         0 --- C:         5
13  Time:  119 --- A:   3.62111 --- B:         0 --- C:   4.23883
14  Time:  120 --- A:   1.50504 --- B:         0 --- C:   2.88765
15  Time:  121 --- A: 0.616288 --- B:         0 --- C:   1.66075
16  Time:  122 --- A:         0 --- B:         0 --- C:  0.838952
17  Time:  123 --- A:         0 --- B:         0 --- C:  0.388568
18  Time:  124 --- A:         0 --- B:         0 --- C:  0.170553
19  Time:  125 --- A:         0 --- B:         0 --- C: 0.0724222
20  Time:  126 --- A:         0 --- B:         0 --- C: 0.0301008
21  Time:  127 --- A:         0 --- B:       250 --- C: 0.0123258
22  Time:  128 --- A:         0 --- B:   211.942 --- C:         0
```

There are a few things to note. Given the stronger input, *neuronB* is spiking 4 ms earlier by the 100 ms mark. However, *neuronC*'s first spike comes 3 ms later when the input signals differ. So, the network is getting a stronger input overall, but the efferent neuron is spiking later. How is that possible?

As I mentioned earlier, coincidence is key. Neurons A and B are no longer spiking simultaneously; therefore, the input to C is spread out. In this second example, you can imagine there are more updates to C when input is above 0.

However, the pulses are weaker. Spikes decay quickly (under default values). At time 112, *neuronA* spikes but its peak signal of 250 is added to the down edge of B, resulting in 290, versus the previous example where spikes overlapped perfectly and peak input was 500.

Incrementing one input channel by one gives the gears of paradise a knock, but the system will still be cyclic. The system's cycle, if not sabotaged by precision or base-2 representations, will just be longer. Like many dynamical systems, spiking neuron models produce complex behavior quite easily.

3 Classes and Methods

3.1 Neuron

At the heart of the Neuron class is an implementation of the Izhikevitch spiking neuron model. It also contains other storage and functionality necessary (IMO) to use the neuron model in anything other than a toy setting.

3.1.1 Methods

Neuron() Default constructor. Better to use the parameterized one below.

Neuron(NT * nt) Parameterized constructor. **Argument list:** pointer to an NT object. **Description:** NT object provides the necessary constants to build a Neuron of a particular type.

Reset() Hard reset of the Neuron **Description:** This can be used to clear any remaining spike history from the Neuron if the program requires a type of hard reset. Also sets v to c and u to d.

Update(uint64_t time) Main update method. **Argument List:** time - a monotonically increasing value. **Description:** Updates the heart of the model. Gathers input and sets output. Registers if the model has spiked.

UpdateSpikes(bool new_spike) Updates the Neuron's spike history. If *new_spike* is true, it adds a spike to the *spike_age_history*. It ages each existing spike in the history, discarding those beyond the *max_spike_age* threshold. It updates the *current_output*.

RegisterSpike(uint64_t time) Sends the time a Neuron spikes to pre- and postsynaptic neurons. Only called if a Neuron spikes.

Output() Responsible for sending the Neuron's *current_output* to all efferent neurons. This happens every time *Update()* is called.

EnableNoise() Both *EnableNoise* functions turn on the Neuron object's ability to generate noise as part of its update routine. Noise is added just like synaptic input in the update to the voltage variable v. There are two types of noise available *Uniform* and *Normal*. If *Uniform* is chosen, the

parameters a and b become the min and max of the range. For *Normal*, a is the mean and b is the standard deviation. By default, Noise is set to None, which means Neuron objects are not subject to any pseudo-random input. The difference between the two functions is that one takes a *seed* and instantiates a new random number generator unique to the Neuron. The other takes a shared pointer to an outside generator.

SetAlphaBase(double alphabase) Alphabase constant of a Neuron determines how much the output alpha function smears an action potential. You can approximately equate the value with how long it takes a spike to reach its max (1.0) output. This method also sets the *max_spike_age* to $10 * alpha_base$. 10 is a magic number in the code and can be changed depending on how much of the spike decay you want to include. Eventually, I will give this a variable within the Neuron class.

AddInputSynapse(std::shared_ptr¡Synapse¿ synapse

AddOutputSynapse(std::shared_ptr¡Synapse¿ synapse Adds input and output synaptic connections to the Neuron.

Input() Used by the Update() method to sum the signal from all synapses in its input vector.

V() Access to the voltage variable

U() Access to the reset variable

GetCurrentOutput() Access to the raw current output variable.

GetCurrentOutputNormalized() Normalizes the current output variable on the fly using the following sigmoid: $\frac{current\_output}{1+current\_output}$. By default, the Neuron object uses this to send output to efferent synapses. This is not always preferable. When *alphabase* is set to larger values, spike output overlaps additively, resulting in values greater than 1. By default this isn't an issue as *alphabase* is set to 1; however, I have used this function over the raw value to limit runaway behavior for those who increase *alphabase*.

3.1.2 Data

I will not cover the Izhikevitch constants here. See the NTemplate class description for an itemized list.

alphabase Alphabase controls the smearing of a Neuron's output due to a spike via the equation:

$$alphaout = \frac{spike\_age}{alphabase} e^{-\frac{(spike\_age - alphabase)}{alphabase}} \tag{2}$$

max_spike_age The age at which a spike ceases to have any impact on the Neuron's behavior.

current_output The sum of alphaout for all spikes younger than max_spike_age.

3.2 Synapse

The Synapse class is an interface providing the necessary functionality for Neuron objects. Currently, Spiking Spielwiese provides a few basic implementations: SimpleSynapse, STDPSynapse, CountingSynapse and PCSynapse.

3.2.1 SimpleSynapse

SimpleSynapse provides a basic connection between neurons, or between neurons and external components. It provides signal scaling ability but is incapable of learning.

active The active variable provides a way for Neuron's to ignore inactive Synapses. Honestly, I am not sure when or where this would be useful. Unlike computational neuroscience projects, non-traditional projects might want the ability to artificially silence certain connections. The default implementation simple ignores these Synapses on both ends, pre- and postsynaptic. Specific projects might want to turn on the presynaptic end to maintain history if the synapse ever becomes active. Or Neurons could delete inactive Synapses. There is commented-out code in the Neuron model's Input and Output methods that does just this.

weight Incoming signal (via the use of SetSignal()) is scaled by this amount. In other words, the default implementation scales the signal when it is stored. Not when it is retrieved. This is somewhat arbitrary.

delay Synapses can have a time delay. Each synapse keeps a history of its signal (see *signal_history_size*). The delay variable allows the synapse to look back into its history. For example, a Synapse with a delay of 5 will respond with its signal from 5 time steps ago when GetSignal is called. Note: *delay* is only useful if the history buffer is greater than it.

time Time is an internal counter that rolls over according to *signal_history_size*. It aids *delay* is retrieving a signal's history.

signal_history_size The size of the signal history buffer.

signal The signal history buffer.

pre_spike_time

post_spike_time The last time the pre- and postsynaptic neurons fired a spike.

3.2.2 STDPSynapse

STDPSynapse is an implementation of a synapse capable of Spike-Time Dependent Plasticity. It inherits from *SimpleSynapse* and so provides signal scaling and delay capabilities. STDP requires the addition of five new variables and some overrides to the methods of *SimpleSynapse*. Changes to the synaptic strength are Hebbian in nature. The Hebbian equation used is:

$$strength = strength + (learn\_rate * e^{\frac{-\delta t}{learn\_window}} \tag{3}$$

It overrides SetSignal in order to incorporate *strength* into the equation. It embeds STDP into overriden spike registering methods.

It should be noted that in SPSP, strength is not clamped to an initial range. Meaning, Hebbian learning can drive the strength variable below zero from above or above from below, effective switching the sign of synaptic scaling. The sign of synaptic weights is the primary method by which connections are excitatory or inhibitory. In other words, it is possible for excitatory relationships between neurons to become, via Hebbian learning, inhibitory and vice versa.

strength The plasticity variable. Rather than alter synaptic weight, *STDPSynapse* alters its strength. Strength and weight are combined into the following sigmoid function:

$$actual\_weight = \frac{(weight * strength)}{|weight| + |strength|} \tag{4}$$

learn_window (Pre and Post) See equation above for strength modification. They govern the time frame in which the behavior of pre- and postsynaptic spikes increase or decrease strength.

learn_rate (Pre and Post) See equation above for strength modification. These govern the rate at which learning happens.

3.2.3 Counting Synapse

A *CountingSynapse* is not biologically inspired; it is a weight-scaled spike counter. Every time the presynaptic neuron fires and registers a new spike, the variable stored in a shared pointer is incremented by the synapses weight. The shared pointer allows external components to track the contents of this variable without needing any interface to the synapse itself. Postsynaptic connections are possible but have no effect on the synapse. By default, *CountingSynapse* and their derivatives provide no input to postsynaptic neurons. In other words, by default the value in count is not utilized as the synapse's signal.

weight Each time the presynaptic neuron fires, the value in *count* is incremented by this.

count The counter.

3.2.4 PCSynapse

A *PCSynapse* is an extension of the *CountingSynapse*. PC stands for Producer-Consumer. Presynaptic activity adds the weight to the value stored in count, and postsynaptic activity subtracts weight from count.

3.3 NTemplate

For a discussion on the various constants that go into an Izhikevich model neuron, I suggest perusing his book [3]. The NT class contains two additional constants of my making: baseline and alphabase.

alphabase See discussion in the Neuron class for details.

baseline baseline provides a way to give a Neuron object constant, internal input. For a computational neuroscience example, some neurons in the human brain fire continuously in the absence of external input. These are called 'tonically-active'. The baseline value provides a way to model this. Another possibility would be to set baseline to a value just below what it takes to cause a neuron to spike. Any subsequent afferent activity would cause a spike. Or baseline could be set to a negative value, requiring a strong constant barrage of activity to dig it out of a well.

4 Examples

4.1 Trace Example

4.2 Follow Example

4.3 Bee Example

4.4 Target Example

4.5 GridWar Example

References

[1] Paul Ardin et al. "Using an Insect Mushroom Body Circuit to Encode Route Memory in Complex Natural Environments". In: *PLOS Computational Biology* 12.2 (Feb. 2016), pp. 1–22.

[2] A. Bouganis and M. Shanahan. "Training a spiking neural network to control a 4-DoF robotic arm based on Spike Timing-Dependent Plasticity". In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*. July 2010, pp. 1–8.

[3] Eugene M. Izhikevich. *Dynamical Systems in Neuroscience: Geometry of Excitability and Bursting, The*. English. Cambridge: MIT Press, 2007. ISBN: 9780262090438.

[4] Wolfgang Maass. "Networks of spiking neurons: The third generation of neural network models". In: *Neural Networks* 10.9 (1997), pp. 1659–1671. ISSN: 0893-6080.

[5] Jurek Müller et al. "A neural network model for familiarity and context learning during honeybee foraging flights". In: *Biological Cybernetics* 112.1 (Apr. 2018), pp. 113–126.

[6] David P.M. Northmore. "A network of spiking neurons develops sensorimotor mechanisms while guiding behavior". In: *Neurocomputing* 58-60 (June 2004). Computational Neuroscience: Trends in Research 2004, pp. 1057–1063. ISSN: 0925-2312.