0.1 Web Client

The web client is designed to be a simple interface through which simulations for each of the modules can be run and automated, both separately and jointly. This graphical user interface interacts with each module's API and provides the functionality needed to save and access the results of every experiment.

In order to maintain consistency with the rest of the project, we decided to use a web framework for Python that could easily connect with each of our modules. The basic requirements were that the client could be easily deployed on a local environment for testing purposes, but also able to serve multiple users concurrently when deployed in production. We chose Bottle (http://bottlepy.org) as it is a lightweight web framework with a built-in HTTP development server that would address the first requirement out-of-the-box. It can also be paired with Nginx (http://wiki.nginx.org/) through uWSGI (https://uwsgi-docs.readthedocs.org/en/latest/). By using Nginx, a high-performance HTTP server on top of uWSGI, a full stack interface between web frameworks and web servers, our web client can be deployed in production, providing load-balanced high-availability.

Given that running our modules' simulations is computationally expensive, it was imperative that the output and results of each run be saved in a persistent store. As shown in figure X, our project consists of five modules connected sequentially, with the output of the each module in the sequence serving as input to the next module. By saving the output of each module during a simulation, we would be able to perform multiple runs of the next module in the sequence without rerunning the previous modules. Additionally, by saving the data generated by each run, the web client can provide a view of the results for analysis by simply querying the database instead of generating them from scratch.

We chose MongoDB, a documented-oriented database, as a data store as it provides a fast, scalable solution that does not require strict design decisions in advance. As we developed our product, MongoDB's dynamic schemas allowed us to modify our objects without having to spend considerable time fixing compatibility issues. Additionally, MongoDB documents follow a JavaScript Object Notation (JSON)-like structure, which mirrors the structure of our Python objects, making it straight forward to understand what fields in our stored documents correspond to what properties of our Python classes.

[DOCUMENT] [OBJECT]

Animation Object as stored in MongoDB

```
{

  "width": 640,
  "height": 480,
  "num_frames": 50,
  "background_id": 552d618629750413075fde0d,
  "description": "Sample animation.",
  "frames_per_second": 5,
  "targets": [
            { 'color': 'rgb(20,97,107)',
              'velocity': '5',
              'velocity_vector': ['1', '2'],
              'type': '1',
              'start_pos': ['1', '2'],
              'frames': '50',
              'size': '1' },

            { 'color': 'rgb(45,86,55)',
              'velocity': '2',
              'velocity_vector': ['2', '5'],
              'type': '2',
              'start_pos': ['10', '3'],
              'frames': '40',
              'size': '3' }
    ]
}
```

Animation Object constructor in Python

```
def __init__(self,

width= 640,
height= 480,
num_frames= 50,
background_id= "552d618629750413075fde0d",
description= "Sample animation.",
frames_per_second= 5,
targets=[
         a.Target(color=rgb(20,97,107),
                  velocity=5,
                  velocity_vector=[1,2],
                  type=1,
                  start_pos=[1,2],
                  frames=50,
                  size=1),

         b.Target(color=rgb(45,86,55),
                  velocity=2,
                  velocity_vector=[2,5],
                  type=2,
                  start_pos=[10,3],
                  frames=40,
                  size=3)]):

self.targets = targets
self.width = width
self.height = height
self.background_id = background_id
self.num_frames = num_frames
self.frames_per_second = frames_per_second
self.description = description
```

Bottle has a built-in templating engine that enhances HTML with a thin layer of Python that can be inserted as both as inline and embedded snippets. These templates also allow the server to pass an object to the view, which can then be accessed using straightforward Python dictionary syntax.

```
<!DOCTYPE html>
<html>
% include('head.tpl', title="Pattern Recognition Simulation")
<body>
% include('header.tpl')
<div class="container">
```

Separate the templates into reusable sections/blocks of HTML code that can be included dynamically in other templates.

```
<!-- Tab Panes -->
<div class="tab-content">
    % for i in range(len(simulation['potential_plots'])):
    % p_plot = simulation['potential_plots'][i]
    <div role="tabpanel" class="tab-pane" id="p{{i + 1}}">
        {{!p_plot}}
    </div>
    %end
</div>
```

Iterate through an array passed to the view and access the relevant fields in each iteration.

```
<tr>
    <td>True Positives</td>
    <td>{{'%.2f' % neuron['spike_info'][0][0]}}%</td>
    <td>{{'%.2f' % neuron['spike_info'][1][0]}}%</td>
    <td>{{'%.2f' % neuron['spike_info'][2][0]}}%</td>
    <td>{{'%.2f' % neuron['spike_info'][3][0]}}%</td>
</tr>
```

Format and access the field of an object passed to the view.