neurotune Documentation

Release 0.0.1b

Mike Vella

June 30, 2014

Contents

CHAPTER 1

A tool for automated parameter tuning of neuronal models

1.1 Installing Neurotune

```
python setup.py --install
```

1.1.1 Dependencies

Neurotune has the following hard dependencies:

- Inspyred
- Numpy

And the following soft dependencies:

- Scipy
- neuronpy
- NEURON

1.1.2 Requirements

Neurotune has so far been tested on Ubuntu 12.04. It should however also work on OSX and Windows.

1.2 Neurotune Architecture

Neurotune provides three important classes: Optimizers, Evaluators and Controllers. Any optimization of a model must utilise (usually) one instance of each of these classes.

1.2.1 Optimizer Class

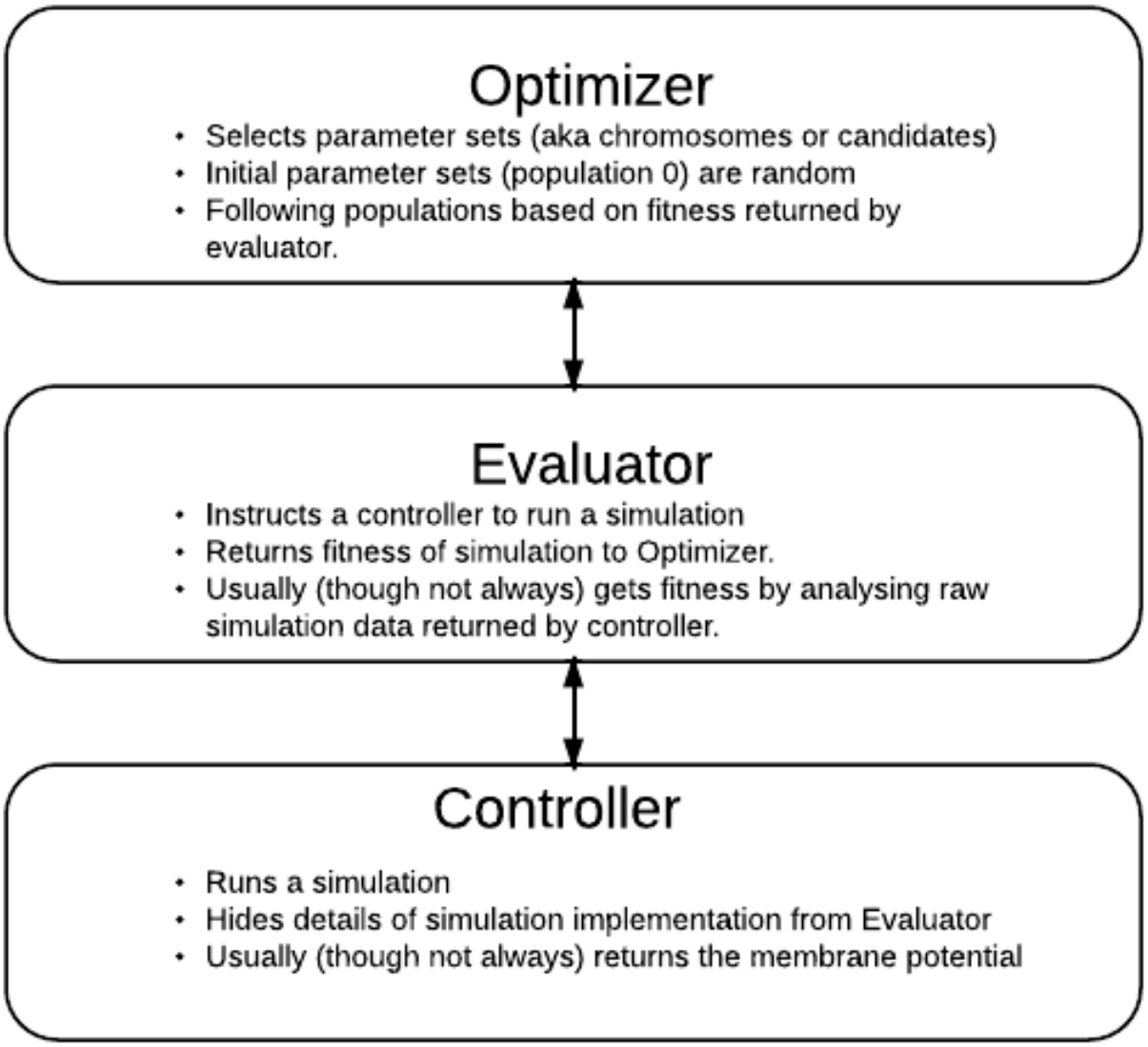
The optimizer controls the progress of the optimization - the mating, survival and death of individuals (each individaul in this case being a simulation). The optimizer generally acheives this by starting with an initial population chosen randomly (Say a population of 100, with 3 free paramaters x,y,z, the population is therefore 100 members [x1,y1,z1]. . . [x100,y100,z100]). Seeds chosen by the user may also be present in the initial population. In order to evaluate this population, the optimizer passes every member to the evaluator which assigns each indivudal a fitness value based on the simulation output.

1.2.2 Evaluator Class

But how does the Evaluator do this? The evaluator does the above by looking at simulation output and targets and assigning a fitness (via its cost function). The evaluator needs a way to run each simulation - this is why it needs a controller. Hence there are things like ClampEvalutor - This evaluator specifially evaluates the finess of a current clamp simulation compared to the exeperimental data. - It passes those finess results back to the optimizer, the optimizer doesn't need to "know""what kind of simulation is being run to obtain these finesses.

1.2.3 Controller Class

The controller class implements the actual running of the simulation given the parameters. A controller will provide the evaluator with a run() method and hide the details of the simulation implementation from the evaluator.



1.2.4 Why this architecture?

This architecture decouples the optimization from the actual assigining of fitness and the actual control of a simulation. The ability easily mix and match optimizers, controllers and evaluators gives great exibility. For in- stance, the CLI Controller used in the musclemodel doesn't need to be a Python script- it just needs to be a program which can accept a command line argument and run a simulation based on it.

1.2.5 What is a dumb evaluator?

Most evaluators work by receiving a trace back from the simulation and assessing its fitness. A dumb evalutor is a bit special - The simulation itself reports a fitness and writes it to a file. The evaluator is dumb because all it needs to do is read this finess value and report that back to the Optimizer. This allows a lot of flexibility (If a user finds that the given evaluators don't match their needs but doesn't want the hassle of writing their own evaluator (e.g doesn't feel comfortable with OOP) they don't have to).

1.3 Examples

Full code for these examples are located [here](

1.3.1 Example 1 - Custom controller class, automatic targets

In this this example a model of a CA1 Basket cell will be optimized using automatic target extraction.

Note: This example is design such that the in-line (code) documentation should be read as well as the explanation between the code snippets.

This means that we will provide our model with a csv file containing time vs voltage data and Neurotune will analyse this file and extract features automatically. This convenience makes it a good place to start.

These examples are located in the examples module of Neurotune, you should also use some raw data (or the csv file provided) to test out your own model.

For this example we will be writing and optimizing a model written in the Python programming language and the NEURON library.

The first thing will do is define a simulation class, this can also be accomplished by a simple function, but the class defined here gives us some powerful functionality:

Note: This simulation class is not necessary when writing a model for optimization, however it is good practice because it makes our code more modular and easier to understand.

```
class Simulation(object):

    """
    Simulation class - inspired by example of Philipp Rautenberg

        Objects of this class control a current clamp simulation. Example of use:

            >>> cell = Cell() #some kind of NEURON section
            >>> sim = Simulation(cell)
            >>> sim.go()
            >>> sim.show()
```

```
    """

    def __init__(self, recording_section, sim_time=1000, dt=0.05, v_init=-60):

        self.recording_section = recording_section
        self.sim_time = sim_time
        self.dt = dt
        self.go_already = False
        self.v_init=v_init

    def set_IClamp(self, delay=5, amp=0.1, dur=1000):
        """
        Initializes values for current clamp.

        Default values:

          delay = 5 [ms]
          amp   = 0.1 [nA]
          dur   = 1000 [ms]

        """
        stim = h.IClamp(self.recording_section(0.5))
        stim.delay = delay
        stim.amp = amp
        stim.dur = dur
        self.stim = stim

    def set_recording(self):
        # Record Time
        self.rec_t = neuron.h.Vector()
        self.rec_t.record(h._ref_t)
        # Record Voltage
        self.rec_v = h.Vector()
        self.rec_v.record(self.recording_section(0.5)._ref_v)

    def show(self):
        """
        Plot the result of the simulation once it's been intialized
        """

        from matplotlib import pyplot as plt

        if self.go_already:
            x = np.array(self.rec_t)
            y = np.array(self.rec_v)

            plt.plot(x, y)
            plt.title("Simulation voltage vs time")
            plt.xlabel("Time [ms]")
            plt.ylabel("Voltage [mV]")

        else:
            print("""First you have to `go()` the simulation.""")
        plt.show()

    def go(self, sim_time=None):
        """
        Start the simulation once it's been intialized
```

```
        """

        self.set_recording()
        h.dt = self.dt

        h.finitialize(self.v_init)
        neuron.init()
        if sim_time:
            neuron.run(sim_time)
        else:
            neuron.run(self.sim_time)
        self.go_already = True
```

The next thing we will do is define our custom controller. The controller in Neurotune is what actually runs the simulation. Neurotune provides off-the-shelf controllers for common needs, however this one is customised to make the purpose of the controller more clear. This is a "canonical controller" because it takes as an input an array of candidates (candidate solutions - strings of numbers corresponding to the parameter set of a solution proposed by the optimizer) and returns a corresponding array of voltage traces. It is also considered canonical because it provides a run method.

```
class BasketCellController():

    """
    This is a canonical example of a controller class

    It provides a run() method, this run method must accept at least two parameters:
        1. candidates (list of list of numbers)
        2. The corresponding parameters.
    """

    def run(self,candidates,parameters):
        """
        Run simulation for each candidate

        This run method will loop through each candidate and run the simulation
        corresponding to it's parameter values. It will populate an array called
        traces with the resulting voltage traces for the simulation and return it.
        """

        traces = []
        for candidate in candidates:
            sim_var = dict(zip(parameters,candidate))
            t,v = self.run_individual(sim_var)
            traces.append([t,v])

        return traces

    def set_section_mechanism(self, sec, mech, mech_attribute, mech_value):
        """
        Set the value of an attribute of a NEURON section
        """
        for seg in sec:
            setattr(getattr(seg, mech), mech_attribute, mech_value)

    def run_individual(self,sim_var):
        """
        Run an individual simulation.
```

```
        The candidate data has been flattened into the sim_var dict. The
        sim_var dict contains parameter:value key value pairs, which are
        applied to the model before it is simulated.

        The simulation itself is carried out via the instantiation of a
        Simulation object (see Simulation class above).

        """

        #make compartments and connect them
        soma=h.Section()
        axon=h.Section()
        soma.connect(axon)

        axon.insert('na')
        axon.insert('kv')
        axon.insert('kv_3')
        soma.insert('na')
        soma.insert('kv')
        soma.insert('kv_3')

        soma.diam=10
        soma.L=10
        axon.diam=2
        axon.L=100

        #soma.insert('canrgc')
        #soma.insert('cad2')

        self.set_section_mechanism(axon,'na','gbar',sim_var['axon_gbar_na'])
        self.set_section_mechanism(axon,'kv','gbar',sim_var['axon_gbar_kv'])
        self.set_section_mechanism(axon,'kv_3','gbar',sim_var['axon_gbar_kv3'])
        self.set_section_mechanism(soma,'na','gbar',sim_var['soma_gbar_na'])
        self.set_section_mechanism(soma,'kv','gbar',sim_var['soma_gbar_kv'])
        self.set_section_mechanism(soma,'kv_3','gbar',sim_var['soma_gbar_kv3'])

        for sec in h.allsec():
            sec.insert('pas')
            sec.Ra=300
            sec.cm=0.75
            self.set_section_mechanism(sec,'pas','g',1.0/30000)
            self.set_section_mechanism(sec,'pas','e',-70)

        h.vshift_na=-5.0
        sim=Simulation(soma,sim_time=1000,v_init=-70.0)
        sim.set_IClamp(150, 0.1, 750)
        sim.go()

        sim.show()

        return np.array(sim.rec_t), np.array(sim.rec_v)
```

The function **main()** is where the actual optimization takes place - the evaluator, controller and optimizer classes are instantiated into objects and the optimizer **optimize()** method is invoked:

```
def main():
    """
    The optimization runs in this main method
```

```
"""

#make a controller
my_controller= BasketCellController()

#parameters to be modified in each simulation
parameters = ['axon_gbar_na',
              'axon_gbar_kv',
              'axon_gbar_kv3',
              'soma_gbar_na',
              'soma_gbar_kv',
              'soma_gbar_kv3']

#above parameters will not be modified outside these bounds:
min_constraints = [0,0,0,0,0,0]
max_constraints = [10000,30,1,300,20,2]


# EXAMPLE - how to set a seed
#manual_vals=[50,50,2000,70,70,5,0.1,28.0,49.0,-73.0,23.0]

#analysis variables, these default values will do:
analysis_var={'peak_delta':0,
              'baseline':0,
              'dvdt_threshold':2}

weights={'average_minimum': 1.0,
         'spike_frequency_adaptation': 1.0,
         'trough_phase_adaptation': 1.0,
         'mean_spike_frequency': 1.0,
         'average_maximum': 1.0,
         'trough_decay_exponent': 1.0,
         'interspike_time_covar': 1.0,
         'min_peak_no': 1.0,
         'spike_broadening': 1.0,
         'spike_width_adaptation': 1.0,
         'max_peak_no': 1.0,
         'first_spike_time': 1.0,
         'peak_decay_exponent': 1.0,
         'pptd_error':1.0}



#make an evaluator, using automatic target evaluation:
my_evaluator=evaluators.IClampEvaluator(controller=my_controller,
                                        analysis_start_time=1,
                                        analysis_end_time=500,
                                        target_data_path='100pA_1.csv',
                                        parameters=parameters,
                                        analysis_var=analysis_var,
                                        weights=weights,
                                        targets=None, # because we're using automatic
                                        automatic=True)

#make an optimizer
my_optimizer=optimizers.CustomOptimizerA(max_constraints,min_constraints,my_evaluator,
                                    population_size=3,
                                    max_evaluations=100,
```

```
                                        num_selected=3,
                                        num_offspring=3,
                                        num_elites=1,
                                        seeds=None)

    #run the optimizer
    my_optimizer.optimize()

main()
```

1.3.2 Example 2 - Custom controller class, manual targets

This example is very similar to the one above, but the optimization is done with manual targets.

This is actually pretty easy. In the example above when the evaluator is defined, there is a line:

We now create a targets dictionary, each target must be one which is available to the specific evaluator, see the evaluator's documentation to see what analysis it provides.

```
manual_targets={'average_minimum': -38.83,
         'spike_frequency_adaptation': 0.01,
         'trough_phase_adaptation': 0.005,
         'mean_spike_frequency': 47.35,
         'average_maximum': 29.32,
         'trough_decay_exponent': 0.11,
         'interspike_time_covar': 0.04,
         'min_peak_no': 34,
         'spike_broadening': 0.81,
         'spike_width_adaptation': 0.00,
         'max_peak_no': 35,
         'first_spike_time': 164.0,
         'peak_decay_exponent': -0.045,
         'pptd_error':0}
```

And then define the evaluator such that the **automatic** key is set to false and the targets parameter is set to our targets dict:

> **Warning:** PPTD error and other such deviation functions should always be 0.0. PPTD error also does not work if a target data path is not provided and an error will result.

```
my_evaluator=evaluators.IClampEvaluator(controller=my_controller,
                                        analysis_start_time=1,
                                        analysis_end_time=500,
                                        target_data_path='100pA_1.csv',
                                        parameters=parameters,
                                        analysis_var=analysis_var,
                                        weights=weights,
                                        targets=manual_targets,
                                        automatic=False)
```

1.3.3 Example 3 - CLI controller, single-threaded

In this example the model of a C. elegans muscle cell will be optimized using manual targets, CLI controller. This work is part of the Open Worm Project.

1.3.4 Example 4 - CLI controller, multi-threaded

As above but multi-threaded.

CHAPTER 2

Source documentation

2.1 neurotune Package

2.1.1 `controllers` Module

The controllers module provides different controller classes, applicable to different simulations.

A controller object's job is to control simulations- At a high level a controller objects accepts a list of parameters and chromosomes and (usually) returns corresponding simulation data. This is implemented polymporphically in subclasses. Each controller class must therefore provide a run method, which is used by the evaluator to run a simulation.

A controller must be able to accept simulation parameters (chromosomes) from the evaluator.

The evaluator is therefore only concerned with assigining fitness to chromosomes.

On the whole this allows for deep modularization - as long as the user can provide a controller which will (for instance) reutrn sample and time arrays for arbitrary chromosome and parameter lists a range of evaluators would be able to utilise it.

class `neurotune.controllers.`**`CLIController`**(*cli_argument*)
Bases: `neurotune.controllers.__Controller`

Control simulations via command line arguments executed through the Python os module.

`run`(*candidates*, *parameters*, *fitness_filename='evaluations'*)
Run simulation

class `neurotune.controllers.`**`NrnProject`**(*nrnproject_path*, *db_path*, *exp_id=None*)
Bases: `neurotune.controllers.__Controller`

Run an nrnproject simulation based on optimizer parameters.

`run`(*candidates*, *parameters*)
Run simulations

class `neurotune.controllers.`**`NrnProjectCondor`**(*host*, *username*, *password*, *port=80*, *local_analysis=False*, *candidates_per_job=100*)
Bases: `neurotune.controllers.NrnProject`

Run NrnProject-based simulations on a Condor-managed federated system

2.1.2 `evaluators` Module

2.1.3 `optimizers` Module

2.1.4 `traceanalysis` Module

2.2 examples Package

2.2.1 Subpackages

example_1 Package

`optimization` Module

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

n