frontiers in Neuroinformatics

TECHNOLOGY REPORT
published: 07 June 2016
doi: 10.3389/fninf.2016.00017

BluePyOpt: Leveraging Open Source Software and Cloud Infrastructure to Optimise Model Parameters in Neuroscience

Werner Van Geit[1], Michael Gevaert[1], Giuseppe Chindemi[1], Christian Rössert[1], Jean-Denis Courcol[1], Eilif B. Muller[1], Felix Schürmann[1], Idan Segev[2,3] and Henry Markram[1,4]**

[1] Blue Brain Project, École Polytechnique Fédérale de Lausanne, Geneva, Switzerland, [2] Department of Neurobiology, Alexander Silberman Institute of Life Sciences, The Hebrew University of Jerusalem, Jerusalem, Israel, [3] The Edmond and Lily Safra Centre for Brain Sciences, The Hebrew University of Jerusalem, Jerusalem, Israel, [4] Laboratory of Neural Microcircuitry, Brain Mind Institute, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

At many scales in neuroscience, appropriate mathematical models take the form of complex dynamical systems. Parameterizing such models to conform to the multitude of available experimental constraints is a global non-linear optimisation problem with a complex fitness landscape, requiring numerical techniques to find suitable approximate solutions. Stochastic optimisation approaches, such as evolutionary algorithms, have been shown to be effective, but often the setting up of such optimisations and the choice of a specific search algorithm and its parameters is non-trivial, requiring domain-specific expertise. Here we describe BluePyOpt, a Python package targeted at the broad neuroscience community to simplify this task. BluePyOpt is an extensible framework for data-driven model parameter optimisation that wraps and standardizes several existing open-source tools. It simplifies the task of creating and sharing these optimisations, and the associated techniques and knowledge. This is achieved by abstracting the optimisation and evaluation tasks into various reusable and flexible discrete elements according to established best-practices. Further, BluePyOpt provides methods for setting up both small- and large-scale optimisations on a variety of platforms, ranging from laptops to Linux clusters and cloud-based compute infrastructures. The versatility of the BluePyOpt framework is demonstrated by working through three representative neuroscience specific use cases.

Keywords: neuron models, optimisation, bluepyopt, open-source, python, multi-objective, evolutionary algorithm, synaptic plasticity

OPEN ACCESS

Edited by:
Arjen Van Ooyen, Vrije Universiteit Amsterdam, Netherlands

Reviewed by:
Alexander K. Kozlov, Royal Institute of Technology (KTH), Sweden
Mikael Djurfeldt, Royal Institute of Technology (KTH), Sweden
Henrik Lindén, University of Copenhagen, Denmark

***Correspondence:**
Werner Van Geit
werner.vangeit@epfl.ch;
Henry Markram
henry.markram@epfl.ch

Received: *01 March 2016*
Accepted: *06 May 2016*
Published: *07 June 2016*

Citation:
Van Geit W, Gevaert M, Chindemi G, Rössert C, Courcol J-D, Muller EB, Schürmann F, Segev I and Markram H (2016) BluePyOpt: Leveraging Open Source Software and Cloud Infrastructure to Optimise Model Parameters in Neuroscience. Front. Neuroinform. 10:17. doi: 10.3389/fninf.2016.00017

1. INTRODUCTION

Advances in experimental neuroscience are bringing an increasing volume and variety of data, and inspiring the development of larger and more detailed models (Izhikevich and Edelman, 2008; Merolla et al., 2014; Markram et al., 2015; Eliasmith et al., 2016). While experimental constraints are usually available for the emergent behaviors of such models, it is unfortunately commonplace that many model parameters remain inaccessible to experimental techniques. The problem of inferring

or searching for model parameters that match model behaviors to experimental constraints constitutes an inverse problem (Tarantola, 2016), for which analytical solutions rarely exist for complex dynamical systems, i.e., most mathematical models in neuroscience. Historically, such parameter searches were done by hand tuning, but the advent of increasingly powerful computing resources has brought automated search algorithms that can find suitable parameters (Bhalla and Bower, 1993; Vanier and Bower, 1999; Achard and De Schutter, 2006; Druckmann et al., 2007; Gurkiewicz and Korngreen, 2007; Van Geit et al., 2007, 2008; Huys and Paninski, 2009; Taylor et al., 2009; Hay et al., 2011; Bahl et al., 2012; Svensson et al., 2012; Friedrich et al., 2014; Pozzorini et al., 2015; Stefanou et al., 2016). While many varieties of search algorithms have been described and explored in the literature (Vanier and Bower, 1999; Van Geit et al., 2008; Svensson et al., 2012), stochastic optimisation approaches, such as simulated annealing and evolutionary algorithms, have been shown to be particularly effective strategies for such parameter searches (Vanier and Bower, 1999; Druckmann et al., 2007; Gurkiewicz and Korngreen, 2007; Svensson et al., 2012). Nevertheless, picking the right type of stochastic algorithm and setting it up correctly remains a non-trivial task requiring domain-specific expertise, and could be model and constraint specific (Van Geit et al., 2008).

With the aim of bringing widely applicable and state-of-the-art automated parameter search algorithms and techniques to the broad neuroscience community, we describe here a Python-based open-source optimisation framework, BluePyOpt, which is available on Github (see Blue Brain Project, 2016), and is designed taking into account model optimisation experience accumulated during the Blue Brain Project (Druckmann et al., 2007; Hay et al., 2011; Markram et al., 2015; Ramaswamy et al., 2015) and the ramp-up phase of the Human Brain Project. The general purpose high-level programming language Python was chosen for developing BluePyOpt, so as to contribute to, and also leverage from the growing scientific and neuroscientific software ecosystem (Oliphant, 2007; Muller et al., 2015), including state-of-the-art search algorithm implementations, modeling and data access tools.

Of course BluePyOpt is not the only tool available to perform parameter optimisations in neuroscience (Druckmann et al., 2007; Van Geit et al., 2007; Bahl et al., 2012; Friedrich et al., 2014; Carlson et al., 2015; Pozzorini et al., 2015). Some tools provide a Graphical User Interface (GUI), other tools are written in other languages, or use different types of evaluation functions or search algorithms. We explicitly didn't make a detailed comparison between BluePyOpt and other tools because many of these tools are developed for specific and non-overlapping applications, making a systematic comparison difficult. This suggests perhaps BluePyOpt's greatest strength, its broad applicability relative to previous approaches.

At its core, BluePyOpt is a framework providing a conceptual scaffolding in the form of an object-oriented application programming interface or API for constructing optimisation problems according to established best-practices, while leveraging existing search algorithms and modeling simulators transparently "under the hood." For common optimisation tasks, the user configures the optimisation by writing a short Python script using the BluePyOpt API. For more advanced use cases, the user is free to extend the API for their own needs, potentially contributing these extensions back to the core library. The latter is important for BluePyOpt APIs to remain broadly applicable and state-of-the-art, as best-practices develop for specific problem domains, mirroring the evolution that has occured for neuron model optimisation strategies (Bhalla and Bower, 1993; Hay et al., 2011).

Depending on the complexity of the model to be optimised, BluePyOpt optimisations can require significant computing resources. The systems available to neuroscientists in the community can be very heterogeneous, and it is often difficult for users to set up the required software. BluePyOpt therefore also provides a novel cloud configuration mechanism to automate setting up the required environment on a local machine, cluster system, or cloud service such as Amazon Web Services.

To begin, this technology report provides an overview of the conceptual framework and open-source technologies used by BluePyOpt, followed by a presentation of the software architecture and API of BluePyOpt. Next, three concrete use cases are elaborated in detail, showing how the BluePyOpt APIs, concepts and techniques can be put to use by potential users. The first use case is an introductory example demonstrating the optimisation of a single compartmental neuron model with two Hodgkin-Huxley ion channels. The second use case shows a BluePyOpt-based state-of-the-art optimisation of a morphologically detailed thick-tufted layer 5 pyramidal cell model of the type used in a recent *in silico* reconstruction of a neocortical microcircuit (Markram et al., 2015). The third use case demonstrates the broad applicability of BluePyOpt, showing how it can also be used to optimise parameters of synaptic plasticity models.

2. CONCEPTS

The BluePyOpt framework provides a powerful tool to optimise models in the field of neuroscience, by combining several established Python-based open-source software technologies. In particular, BluePyOpt leverages libraries providing optimisation algorithms, parallelization, compute environment setup, and experimental data analysis. For numerical evaluation of neuroscientific models, many open-source simulators with Python bindings are available for the user to choose from (see Section 2.2). The common bridge allowing BluePyOpt to integrate these various softwares is the Python programming language, which has seen considerable uptake and a rapidly growing domain-specific software ecosystem in the neuroscience modeling community in recent years (Muller et al., 2015). Python is recognized as a programming language which is fun and easy to learn, yet also attractive to experts, meaning that novice and advanced programmers alike can easily use BluePyOpt, and contribute solutions to neuroscientific optimisation problems back to the community.

BluePyOpt was developed using an object oriented programming model. **Figure 1** shows an overview of the

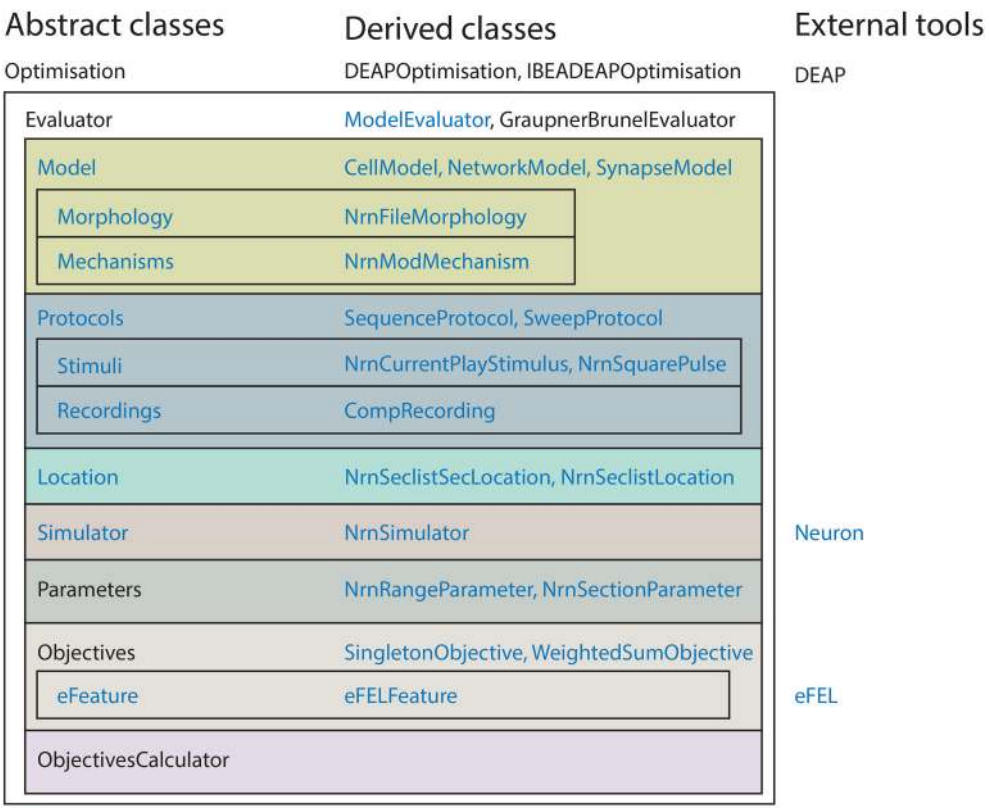

FIGURE 1 | Hierarchy of the most important classes in BluePyOpt. Ephys abstraction layer in blue.

class hierarchy of BluePyOpt. In its essence, the BluePyOpt object model defines the *Optimisation* class which applies a search algorithm to an *Evaluator* class. Both are *abstract classes*, meaning they define the object model, but not the implementation. Taking advantage of Pythonic *duck typing*, the user can then choose from a menu of implementations, *derived classes*, or easily define their own implementations to meet their specific needs. This design makes BluePyOpt highly versatile, while keeping the API complexity to a minimum. The choice of algorithm and evaluator is up to the user, but many are already provided for various use cases (see Section 2.1). For many common use cases, these are the only classes users are required to instantiate.

For neuron model optimisations in particular, BluePyOpt provides further classes to support feature-based multi-objective optimisations using NEURON, as shown in **Figure 1**. Classes *Model*, *Morphology*, *Mechanisms*, *Protocol*, *Stimuli*, *Recordings*, *Location* are specific to setting up neuron models and assessing their input-output properties. In the examples in this paper we focus on single cell models by using the *CellModel* class. There is also the possibility to define classes like e.g., *NetworkModel* or *SynapseModel* to optimise other types of models. Other classes *Objectives* and *eFeature* are more generally applicable, with *derived classes* for specific use cases, e.g., *eFELFeature* provides features extracted from voltage traces using the open-source eFEL library discussed below. They define features and objectives for feature-based multi-objective optimisation, a stochastic optimisation strategy (Druckmann et al., 2007, 2011; Hay et al., 2011). We generally recommend it as the first algorithm to try for a given problem domain. For example, the third example for the optimisation of synaptic plasticity models also employs this strategy.

In the sub-sections to follow, an overview is provided for the various software components and the manner in which BluePyOpt integrates them.

2.1. Optimisation Algorithms

Multiobjective evolutionary algorithms have been shown to perform well to optimise parameters of biophysically detailed multicompartmental neuron models (Druckmann et al., 2007; Hay et al., 2011). To provide optimisation algorithms, BluePyOpt relies on a mature Python library, Distributed Evolutionary Algorithms in Python (DEAP), which implements a range of such algorithms (Fortin et al., 2012). The advantage of using this library is that it provides many useful features out of the box, and it is mature, actively maintained and well documented. DEAP provides many popular algorithms, such as Non-dominated Sorting Genetic Algorithm-II (Deb et al., 2002), Covariance Matrix Adaptation Evolution Strategy (Hansen and Ostermeier, 2001), and Particle Swarm Optimisation (Kenny and Eberhart, 1995). Moreover, due to its extensible design, implementing new search algorithms in DEAP is straight-forward. Historically, the

Blue Brain Project has used a *C* implementation of the Indicator Based Evolutionary Algorithm *IBEA* to optimise the parameters of biophysically detailed neuron models (Bleuler et al., 2003; Zitzler and Künzli, 2004; Markram et al., 2015), as this has been shown to have excellent convergence properties for these problems (Schmücker, 2010). Case in point, we implemented a version of IBEA for the DEAP framework, so this algorithm is consequently available to be used in BluePyOpt.

Moreover, DEAP is highly versatile, whereby most central members of its class hierarchy, such as individuals and operators, are fully customizable with user defined implementations. Classes are provided to keep track of the *Pareto Front* or the *Hall-of-Fame* of individuals during evolution. Population statistics can be recorded in a logbook, and the genealogy between individuals can be saved, analyzed and visualized. In addition, *checkpointing* can be implemented in DEAP by storing the algorithm's state in a Python pickle file for any generation, as described in DEAP's documentation (DEAP Project, 2016).

Although the use cases below use DEAP as a library to implement the search algorithm, it is worth noting that BluePyOpt abstracts the concept of a search algorithm. As such, it is entirely possible to implement algorithms that are independent of DEAP, or that use other third-party libraries.

2.2. Simulators

To define a BluePyOpt optimisation, the user must provide an evaluation function which maps model parameters to a fitness score. It can be a single Python function that maps the parameters to objectives by solving a set of equations, or a function that uses an external simulator to evaluate a complex model under multiple scenarios. For the latter, the only requirement BluePyOpt imposes is that it can interact with the external simulator from within Python. Often, this interaction is implemented through Python modules provided by the user's neuroscientific simulator of choice, as is the case for many simulators in common use, including NEURON (Hines et al., 2009), NEST (Eppler et al., 2009; Zaytsev and Morrison, 2014), PyNN (Davison et al., 2009), BRIAN (Goodman and Brette, 2009), STEPS (Wils and De Schutter, 2009), and MOOSE (Ray and Bhalla, 2008). Otherwise, communication through shell commands and input/output files is also possible, so long as an interface can be provided as a Python class.

2.3. Feature Extraction

For an evaluation function to compute a fitness score from simulator output, the resulting traces must be compared against experimental constraints. Voltage recordings obtained from patch clamp experiments are an example of experimental data that can be used as a constraint for neuron models. From such recordings the neuroscientist can deduce many interesting values, like the input resistance of the neuron, the action potential characteristics, firing frequency etc. To standardize the way these values are measured, the Blue Brain Project has released the Electrophysiology Feature Extract Library (eFEL) (Blue Brain Project, 2015), also as open-source software. The core of this library is written in C++, and a Python wrapper is provided. BluePyOpt can interact with eFEL to compute a variety of features of the voltage response of neuron models. A fitness score can then be computed by some distance metric comparing the resulting model features to their experimental counterparts. As we will see for the last example in this article, a similar approach can also be taken for other optimisation problem domains.

2.4. Parallelization

Optimisations of the parameters of an evaluation function typically require the execution of this function repeatedly. For a given iteration of the optimisation, such executions are often in the hundreds (scaling e.g., with evolutionary algorithm population size), are compute bound, and are essentially independent, making them ripe for parallelization. Parallelization of the optimisation can be performed in several ways. DEAP provides an easy way to evaluate individuals in a population on several cores in parallel. The user need merely provide an implementation of a *map* function. In its simplest form, this function can be the Python serial *map* in the standard library, or the parallel *map* function in the multiprocessing module to leverage local hardware threads. To parallelize over a large cluster machine, the DEAP developers encourage the use of the SCOOP (Hold-Geoffroy et al., 2014) map function. SCOOP is a library that builds on top of ZeroMQ (ZeroMQ Project, 2007), which provides a socket communication layer to distribute the computation over several computers. Other map functions and technologies can be used like MPI4Py (Dalcín et al., 2005) or iPython ipyparallel package (Pérez and Granger, 2007). Moreover, parallelization does not necessarily have to happen at the population level. Inside the evaluation of individuals, map functions can also be used to parallelize over stimulus protocols, feature types, etc., however for the problem examples presented here, such an approach would not make good use of anything more than 10 to 20 cores.

2.5. Cloud

To increase the throughput of optimisations, multiple computers can be used to parallelize the work. Such a group of computers can be composed of machines in a cluster, or they can be obtained from a cloud provider like Amazon Web Services, Rackspace Public Cloud, Microsoft Azure, Google Compute Engine, or the Neuroscience Gateway portal (Sivagnanam et al., 2013).

These and other cloud providers allow for precise allocation of numbers of machines and their storage, compute power and memory. Depending on the needs and resources of an individual or organization, trade-offs can be made on how much to spend vs. how fast the results are needed.

Setting up a cluster or cloud environment with the correct software requirements is often complicated and error prone: Each environment has to be exactly the same, and scripts and data need to be available in the same locations. To ease the burden of this configuration, BluePyOpt includes Ansible (Red Hat, Inc., 2012) configuration scripts for setting up a test environment on one local computer (using Vagrant HashiCorp, 2010), for setting up a cluster with a shared file system, or for provisioning and setting up an Amazon Web Services cluster.

Ansible is open-source software that allows for reproducible environments to be created and configured from simple textual descriptions called "Playbooks." These Playbooks encapsulate the discrete steps needed to create an environment, and offer extra tools to simplify things like package management, user creation and key distribution. Furthermore, when a Playbook is changed and run against an already existing environment, only the changes necessary will be applied. Finally, Ansible has the advantage over other systems, like Puppet Labs (2005) and Chef (2009), that nothing except a Python interpreter needs to be installed on the target machine and all environment discovery and configuration is performed through SSH from the machine on which Ansible is run. This decentralized system means that a user can use Ansible to setup an environment in their home directory on a cluster, without intervention from the system administrators.

3. SOFTWARE ARCHITECTURE

The BluePyOpt software architecture follows an object oriented programming model, whereby the various concepts of the software are modularized into cleanly separated and well defined classes which interact as shown in diagrams of the class hierarchy (**Figure 1**), object model (**Figure 2**) and program control flow (**Figure 3**). In what follows, the role of each class and how it relates to and interacts with other classes in the hierarchy is described.

3.1. Optimisation Abstraction Layer

At the highest level of abstraction, the BluePyOpt *API* contains the classes *Optimisation* and *Evaluator* (**Figure 2**). An *Evaluator* object defines an evaluation function that maps *Parameters* to *Objectives*. The *Optimisation* object accepts the *Evaluator* as input, and runs a search algorithm on it to find the parameter values that generate the best objectives. At the moment, the main optimisation subclass is *IBEADEAPOptimisation*, but this will be extended in the future with other classes for specific search algorithms.

The task of the search algorithm is to find the parameter values that correspond to the best objective values. Defining "best" is left to the specific implementation, but the optimisation algorithm and evaluator should follow consistent conventions. As in the use cases below, the goal of the algorithm could be minimizing a weighted sum of the objectives or a multiobjective front in a multidimensional space.

The *Optimisation* class allows the user to control the settings of the search algorithm. In case of IBEA, this could be the number of individuals in the population, the mutation probabilities, etc.

Multiple *Objectives* can be accumulated using e.g., *MaxObjective* or *WeightedSumObjective* objects before being passed on to the search algorithm. When running multiobjective optimisations, the algorithm will treat the objectives as separate dimensions, and will not provide a single score to every individual. At the end of an optimisation, the hall of fame ranking is based on the sum of the objectives. Depending on the preferences of the user, this can easily be extended to

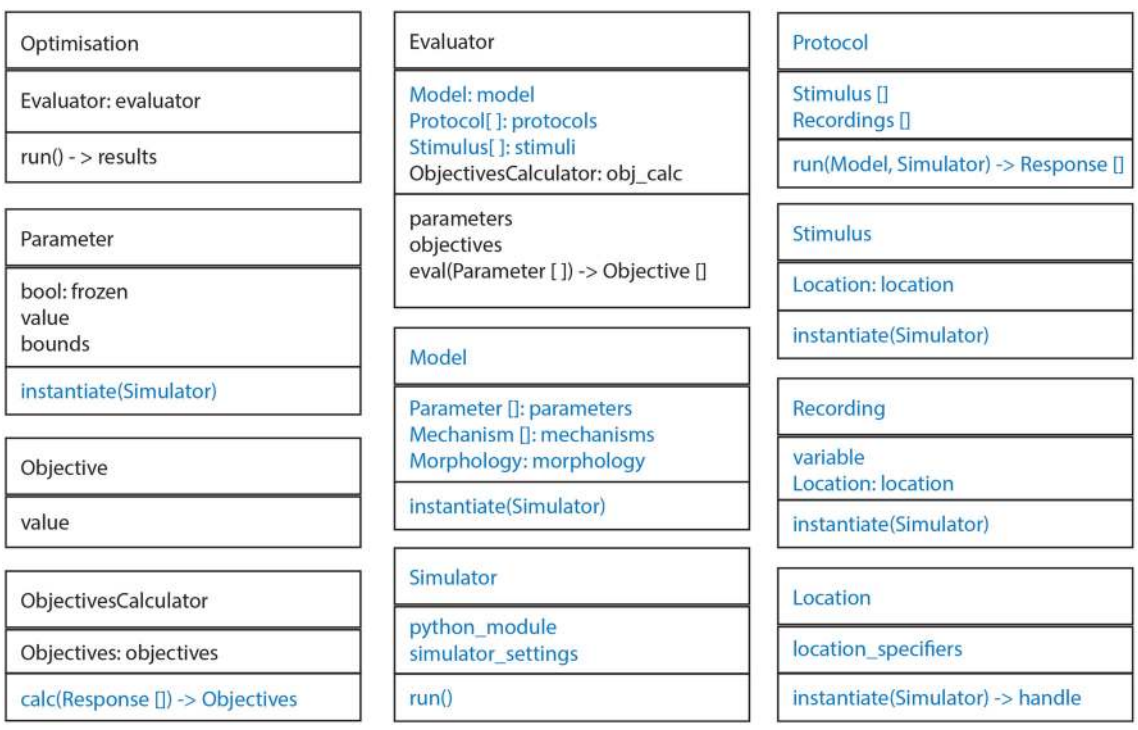

FIGURE 2 | General structure of most important classes. Every box represents a class. In every box the top panel is the name, the middle panel the most important fields and the bottom panel the most important methods. Ephys abstraction layer in blue.

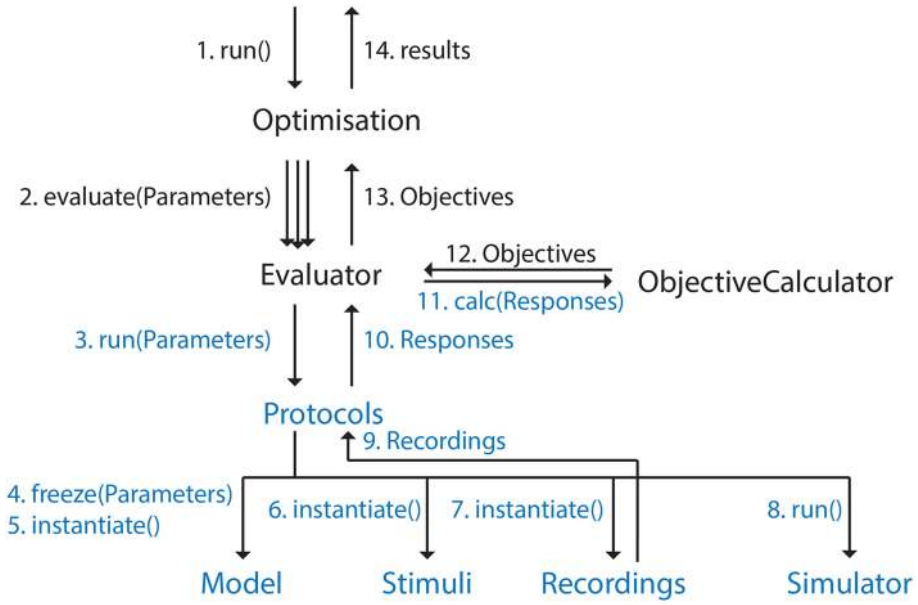

FIGURE 3 | Graph representing control flow in BluePyOpt. Ordering is clarified by the numbers. Arrow labels that contain parentheses represent function calls, the other labels data being returned. This figure is meant to give a high level description of the control flow, not all function calls and intermediate objects are included. Ephys abstraction layer in blue.

include other combinations like root mean square, weighted sum, etc.

3.2. EPhys Model Abstraction Layer

On a different level of abstraction, we have classes that are tailored for electrophysiology (ephys) experiments and can be used inside the *Evaluator*. The ephys model layer provides an abstraction of the simulator, so that the person performing the optimisation is not required to have knowledge of the intricate details of the simulator. This layer is provided as a convenience, and is entirely optional. Users can choose to implement an electrophysiological model in any way they want, as long as they construct their own *Evaluator*.

A *Protocol* is applied to a *Model* for a certain set of *Parameters*, generating a *Response*. An *ObjectivesCalculator* is then used to calculate the *Objectives* based on the *Response* of the *Model*. All these classes are part of the `bluepyopt.ephys` package.

3.2.1. Model

By making a *Model* an abstract class, we give users the ability to use our software for a broad range of use cases. A *Protocol* can attach *Stimuli* and *Recordings* to a *Model*. When the *Simulator* is then run, a *Response* is generated for each of the *Stimuli* for a given set of *Model* parameter values.

Examples of broad subclasses are a *NetworkModel*, *CellModel* and *SynapseModel*. Specific subclasses can be made for different simulators, or assuming some level of similarity, the same model object can know how to instantiate itself in different simulators. In the future, functionality could be added to import/export the model configuration from/to standard description languages like NeuroML (Cannon et al., 2014) or NineML (Raikov et al., 2011).

Particular parameters of a *Model* can be in a frozen state. This means that their value is fixed, and will not be considered for optimisation. This concept can be useful in multi-stage optimisation in which subgroups of a model are optimised in a sequential fashion.

Another advantage of this abstraction is that a *Model* is a standalone entity that can be run outside of the *Optimisation* and have exactly the same *Protocols* applied to it, generating exactly the same *Response*. One can also apply extra *Protocols* to assess how well the model generalizes, or to perfom a sensitivity analysis.

3.2.2. Simulator

Every model simulator should have a subclass of *Simulator*. Objects of this type will be passed on to objects that are simulator aware, like the *Model* and *Stimuli* when their *instantiate* method is invoked. This architecture allows e.g., the same model object to be run in different simulators. Examples of functionality this class can provide are links to the Python module related to the simulator, the enabling of variable time step integration, etc. *Simulators* also have *run()* method that starts the simulation.

3.2.3. Protocol

A *Protocol* is an object that elicits a *Response* from a model. In its simplest form it represents, for example, a step current clamp stimulus, but more complicated versions are possible, such as stimulating a set of cells in a network with an elaborate protocol and recording the response. A *Protocol* can also contain sub-protocols, providing a powerful mechanism to reuse components.

3.2.4. Stimulus, Recording and Response

The *Stimulus* and *Recording* objects, which are part of a *Protocol* are applied to a model and are aware of the simulator used. Subclasses of *Stimulus* are concepts like current/voltage clamp, synaptic activation, etc. Both of these classes accept a *Location* specifier. Several *Recording* objects can be combined in a *Response* which can be analyzed by an *ObjectiveCalculator*. At the moment the *Recording* and *Response* objects store all their data in memory, but if the need arises, the underlying implementation could also use data models that write to disk.

3.2.5. Location

Specifying the location on a neuron morphology of a recording, stimulus or parameter in a simulator can be complicated. Therefore we created an abstract class *Location*. As arguments the constructor accepts the location specification, e.g., in NEURON this could be a *sectionlist* name and an index of the section, or it could point to a section at a certain distance from the soma. Upon request, the object will return a reference to the object at the specified location, this could e.g., be a NEURON section or compartment. At a location, a variable can be set or recorded by a *Parameter* or *Recording*, respectively.

3.2.6. ObjectivesCalculator, eFeature

The *ObjectivesCalculator* takes the *Response* of a *Model* and calculates the objective values from it. When using ephys recordings, one can use the eFEL library to extract *eFeatures*. Examples of these eFeatures are spike amplitudes, steady state voltages, etc. The values of these eFeatures can then be compared with values from experimental data, and a score can be calculated based on the difference between model and experiment (for an example of such a score, see Section 4.2.3. Features are not limited to voltage traces or the eFEL library, but can also be computed on concentrations or any other variables which can be recorded using the *Recording* class during the experiment.

4. EXAMPLE USE CASES

To provide hands on experience how real-world optimisations can be developed using the BluePyOpt API, this section provides step-by-step guides for three use-cases. The first is a single compartmental neuron model optimisation, the second is an optimisation of a state-of-the-art morphologically detailed neuron model, and the third is an optimisation of a synaptic plasticity model. All examples to follow assume NEURON default units, i.e., ms, mV, nA, $\mu F\,cm^{-2}$, etc. (Carnevale and Hines, 2008).

4.1. Single Compartmental Model

The first use case shows how to set up an optimisation of single compartmental neuron model with two free parameters: The maximal conductances of the sodium and potassium Hodgkin-Huxley ion channels. This example serves as an introduction for the user to the programming concepts in BluePyOpt. It uses the NEURON simulator as backend.

First we need to import the top-level `bluepyopt` module. This example will also use BluePyOpt's electrophysiology features, so we also need to import the `bluepyopt.ephys` subpackage.

```
import bluepyopt as bpop
import bluepyopt.ephys as ephys
```

Next we load a morphology from a file. By default a morphology in NEURON has the following *sectionlists*: somatic, axonal, apical and basal. We create a *Location* object (specifically, a *NrnSecListLocation* object) that points to the somatic sectionlist. This object will be used later to specify where mechanisms are to be added etc.

```
morph = ephys.morphologies.NrnFileMorphology
      ('simple.swc')
somatic_loc = ephys.locations.NrnSeclistLocation
           ('somatic', seclist_name= 'somatic')
```

Now we can add ion channels to this morphology. First we add the default NEURON Hodgkin-Huxley mechanism to the soma, as follows.

```
hh_mech = ephys.mechanisms.NrnMODMechanism(
  name='hh',
  prefix='hh',
  locations=[somatic_loc])
```

The *name* argument can be chosen by the user, and should be unique across mechanisms. The *prefix* argument string should correspond to the SUFFIX field in the NEURON NMODL description file (Carnevale and Hines, 2006) of the channel. The *locations* argument specifies which sections the mechanism are to be added to.

Next we need to specify the parameters of the model. A parameter can be in two states: frozen and not-frozen. When a parameter is frozen it has an exact known value, otherwise it has well-defined bounds but the exact value is not known yet. The parameter for the specific capacitance of the soma will be a frozen value.

```
cm_param = ephys.parameters.NrnSectionParameter(
  name='cm',
  param_name='cm',
  value=1.0,
  locations=[somatic_loc],
  frozen=True)
```

Here *param_name* refers to the name of the parameter in the NEURON simulator namespace, whereas *name* is a user-specified alias used in BluePyOpt.

The two parameters that represent the maximal conductance of the sodium and potassium channels are to be optimised, and are therefore specified as `frozen=False`, i.e., not-frozen, and bounds for each are provided with the *bounds* argument.

```
gnabar_param = ephys.parameters.
             NrnSectionParameter(
  name='gnabar_hh',
  param_name='gnabar_hh',
  locations=[somatic_loc],
  bounds=[0.05, 0.125],
  frozen=False)
```

```
gkbar_param = ephys.parameters.NrnSectionParameter(
  name='gkbar_hh',
  param_name='gkbar_hh',
  bounds=[0.01, 0.075],
  locations=[somatic_loc],
  frozen=False)
```

To create the cell template, we pass all these objects to the constructor of the model.

```
simple_cell = ephys.cellmodels.NrnCellModel(
  name='simple_cell',
  morph=morph,
  mechs=[hh_mech],
  params=[cm_param, gnabar_param, gkbar_param])
```

To optimise the parameters of the cell, we further need to create a *CellEvaluator* object. This object needs to know which protocols to inject, which parameters to optimise, and how to compute a score, so we'll first create objects that define these aspects.

A protocol consists of a set of stimuli and a set of responses (i.e., recordings). These responses will later be used to calculate the score of the specific model parameter values. In this example, we will specify two stimuli, two square current pulses delivered at the soma with different amplitudes. To this end, we first need to create a location object for the soma. This object is required in addition to the previously defined *somatic_loc*, because it points to the compartment in the middle of the soma, whereas the former refers to the entire section list defining the soma.

```
soma_loc = ephys.locations.NrnSeclistLocation(
  name='soma',
  seclist_name='somatic',
  sec_index=0,
  comp_x=0.5)
```

For each step in the protocol, we add a stimulus (*NrnSquarePulse*) and a recording (*CompRecording*) in the soma.

```
sweep_protocols = {}
for protocol_name, amplitude in [('step1', 0.01),
    ('step2', 0.05)]:
  stim = ephys.stimuli.NrnSquarePulse(
    step_amplitude=amplitude,
    step_delay=100,
    step_duration=50,
    location=soma_loc,
    total_duration=200)
  rec = ephys.recordings.CompRecording(
  name='%s.soma.v' % protocol_name,
  location=soma_loc,
  variable='v')
  protocol = ephys.protocols.SweepProtocol
            (protocol_name, [stim], [rec])
  sweep_protocols[protocol.name] = protocol
```

The `step_amplitude` argument of the NrnSquarePulse specifies the amplitude of the current pulse, and `step_delay`, `step_duration`, and `total_duration` specify the start time, length and total simulation time. Finally, we create a combined protocol that encapsulates both current pulse protocols.

```
twostep_protocol = ephys.protocols.SequenceProtocol
          ('twostep', protocols=sweep_protocols)
```

Now to compute the model score that will be used by the optimisation algorithm, we define objective objects. For this example, our objective is to match the eFEL "Spikecount" feature to specified values for both current injection amplitudes. In this case, we will create one objective per feature (see Section 4.2.3 for more information about this objective). The eFEL allows for features requiring multiple traces, such as simultaneous somatic and dendritic voltage recordings. The *recording_names* argument therefore takes a dictionary where the keys are the recording locations. The empty string key denotes the main location, in this case the soma.

```
efel_feature_means = {'step1': {'Spikecount': 1},
    'step2': {'Spikecount': 5}}

objectives = []

for protocol_name, protocol in
    protocols.iteritems():
  stim_start = protocol.stimuli[0].step_delay
  stim_end = stim_start +
      protocol.stimuli[0].step_duration
  for efel_feature_name, mean in
  efel_feature_means[protocol_name].iteritems():
    feature_name = '%s.%s' % (protocol_name,
        efel_feature_name)
    feature = ephys.efeatures.eFELFeature(
      feature_name,
      efel_feature_name=efel_feature_name,
      recording_names={'': '%s.soma.v' %
          protocol_name},
      stim_start=stim_start,
      stim_end=stim_end,
      exp_mean=mean,
      exp_std=0.05 * mean)
    objective = ephys.objectives.SingletonObjective(
      feature_name,
      feature)
    objectives.append(objective)
```

We then pass these objective definitions to a *ObjectivesCalculator* object, calculate the total scores from a protocol response.

```
obj_calc = ephys.scorecalculators.
        ObjectivesCalculator(objectives)
```

Finally, we can combine everything together into a *CellEvaluator*. The CellEvaluator constructor has a field *param_names* which contains the (ordered) list of names of the parameters that are used as input (and will be fitted later on).

```
cell_evaluator = ephys.evaluators.CellEvaluator(
  cell_model=simple_cell,
  param_names=['gnabar_hh', 'gkbar_hh'],
  fitness_protocols=protocols,
  fitness_calculator=obj_calc)
```

Now that we have a cell template and an evaluator for this cell, the *IBEADEAPOptimisation* object can be created and run. As an evolutionary algorithm, the IBEA algorithm evolves a

population through consecutive generations. For each generation or iteration, a set of offspring individuals are generated from selected parents from the previous generation. When setting up the algorithm, we can specify the size of the offspring population and the maximum number of generations.

```
optimisation = bpop.deapext.optimisations.
            IBEADEAPOptimisation(
  evaluator=cell_evaluator,
  offspring_size = 100)

final_pop, hall_of_fame, logs, hist =
    optimisation.run(max_ngen=10)
```

After a short time (approximately 4 min on a single 2.9 GHz Intel Core i5 core), the optimisation returns the final population, the hall of fame (sorted by the sum of the objectives), a logbook, and an object containing the history of the population during the execution of the algorithm. **Figure 4** shows the results in a graphical form. As expected, the solution to the optimisation problem is not unique. Several distinct individuals (**Figure 4A**) have a perfect score of 0. We can visualize the region of the solution space explored by the algorithm using a triangular grid plot (**Figure 4B**).

4.2. Neocortical Pyramidal Cell

Our second use case is a more complex example demonstrating the optimisation of a morphologically detailed model of a thick-tufted layer 5 pyramidal cell (L5PC) from the neocortex (**Figure 5A**). This example uses a BluePyOpt port of the state-of-the-art methods for the optimisation of the L5PC model described in Markram et al. (2015). The original model is available online from the Neocortical Microcircuit Collaboration Portal (Ramaswamy et al., 2015). Due to its complexity, we will not describe the complete optimisation script here. The full code is available from the BluePyOpt website. What we will do here is highlight the particularities of this model compared to the introductory single compartmental model optimisation. As a first validation and point of reference, we ran the BluePyOpt model with its original parameter values from Ramaswamy et al. (2015), as shown in **Figure 5B**.

For clarity, the code for setting the parameters, objective and optimisation algorithm is partitioned into separate modules. Configuration values are stored and read from JavaScript Object Notation *JSON* files.

4.2.1. Parameters

Evidently, the parameters of this model, as shown in **Table 1**, far exceed in number those of the single compartmental use case. The parameters marked as frozen are kept constant throughout the optimisation. The parameters to be optimised are the maximal conductances of the ion channels and two values related to the calcium dynamics. The location of the parameters is based on sectionlist names, whereby sections are automatically assigned to the somatic, axonal, apical and basal sectionlists by NEURON when it loads a morphology.

An important aspect of this neuron model is the non-uniform distribution of certain ion channel conductances. For example,

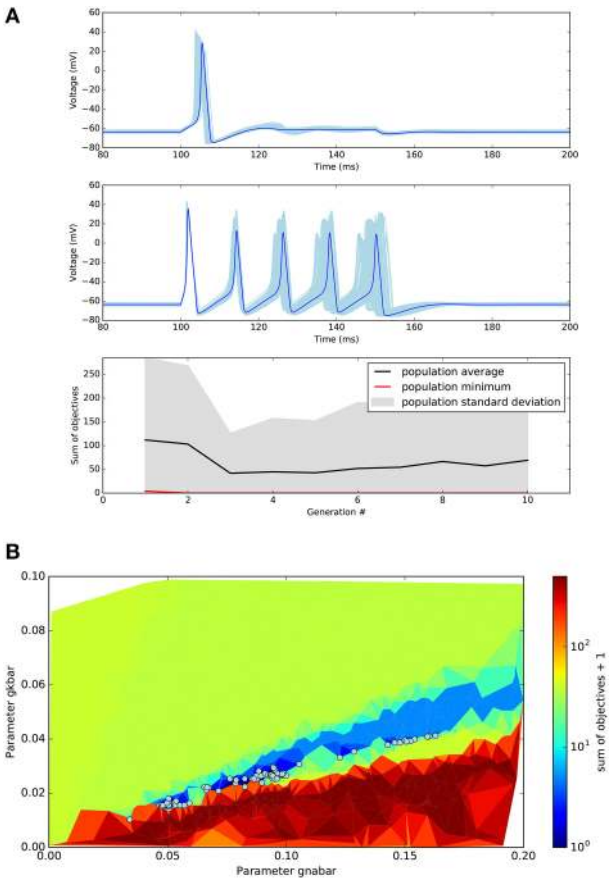

FIGURE 4 | Results of the single compartmental model optimisation. **(A)** Top plots: In light blue, voltage traces recorded during the two different step current injections for all the individuals found that have objectives sum equal to zero. In dark blue, an example of one of these individuals. The target objectives of Step1 and Step2 were 1 and 5 action potentials respectively. Bottom plot: Evolution of minimal objectives sum during the 10 generations of the evolutionary algorithm. **(B)** Triangular grid plot of the parameter space. Every point of the grid is a point where the algorithm evaluated an individual. X- and Y-axis represent the values of the sodium and potassium maximal conductance respectively (units $\mathrm{S\,cm^{-2}}$). The color represents the average of the objectives sum of every triangle's three points. An offset of 1 was added to the logarithmic color bar scale for visualization purposes. Circles represent the solutions with an objectives sum of 0.

the h-channel conductance is specified to increase exponentially with distance from the soma (Kole et al., 2006), as follows.

```
soma_loc = ephys.locations.NrnSeclistLocation(
  seclist_name='somatic',
  seclist_index=0,
  seg_x=0.5)

exponential_scaler =
    ephys.parameterscalers.NrnDistanceScaler(
  origin=soma_loc,
  distribution='(-0.8696 +
      2.087*math.exp(({distance})*0.0031))*{value}')

parameter = ephys.parameters.NrnRangeParameter(
  name='gIhbar_Ih.apical',
```

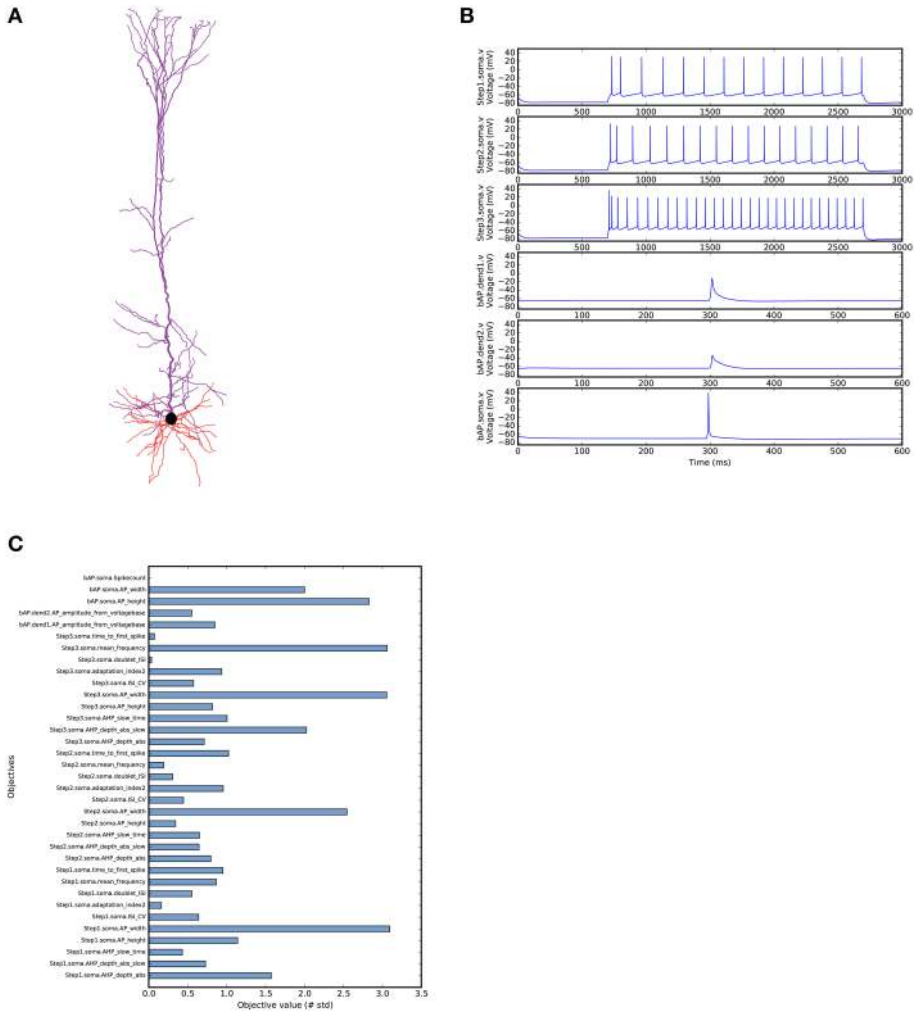

FIGURE 5 | L5PC model as simulated by BluePyOpt with parameter values from Markram et al. (2015). (A) Morphological reconstruction of L5PC used in the model obtained from the NMC portal (Ramaswamy et al., 2015). **(B)** Voltage traces recorded in soma and dendrites (dend1 660 μm, dend2 800 μm from soma in apical trunk). **(C)** Objective scores for the model calculated based on experimental mean and standard deviation.

```
param_name='gIhbar_Ih'
value_scaler=exponential_scaler,
value=8e-5,
frozen=True,
locations=[apical_loc]))
```

4.2.2. Protocols

During the optimisation, the model is evaluated using three square current step stimuli applied and recorded at the soma. For these protocols, a holding current is also applied during the entire stimulus, the amplitude of which is the same as was used in the *in vitro* experiments to keep the cell at a standardized membrane voltage before the step current injection.

Another stimulus protocol checks for a backpropagating action potential (*bAP*) by stimulating the soma with a very short pulse, and measuring the height and width of the bAP at a location of 660 and 800 μm from the soma in the apical dendrite. It is specified as follows.

```
for loc_name, loc_distance in [('dendloc1', 660),
    ('dendloc2', 800)]:
 loc = ephys.locations.
       NrnSomaDistanceCompLocation(
           name=loc_name,
           soma_distance=loc_distance)
 recording = nrpel.recordings.CompRecording(
      name='bAP.%s.v' % (loc_name),
      location=loc)
```

4.2.3. Objectives

For each of the four stimuli defined above, a set of eFeatures is calculated (**Table 2**). These are then compared with the same features extracted from experimental data. As described in Markram et al. (2015), experiments were performed that applied

TABLE 1 | List of parameters for L5PC example.

Location	Mechanism	Parameter name	Distribution	Units	Lower bound	Upper bound
Apical	NaTs2_t	gNaTs2_tbar	Uniform	$S\,cm^{-2}$	0	0.04
Apical	SKv3_1	gSKv3_1bar	Uniform	$S\,cm^{-2}$	0	0.04
Apical	Im	gImbar	Uniform	$S\,cm^{-2}$	0	0.001
Axonal	NaTa_t	gNaTa_tbar	Uniform	$S\,cm^{-2}$	0	4
Axonal	Nap_Et2	gNap_Et2bar	Uniform	$S\,cm^{-2}$	0	4
Axonal	K_Pst	gK_Pstbar	Uniform	$S\,cm^{-2}$	0	1
Axonal	K_Tst	gK_Tstbar	Uniform	$S\,cm^{-2}$	0	0.1
Axonal	SK_E2	gSK_E2bar	Uniform	$S\,cm^{-2}$	0	0.1
Axonal	SKv3_1	gSKv3_1bar	Uniform	$S\,cm^{-2}$	0	2
Axonal	Ca_HVA	gCa_HVAbar	Uniform	$S\,cm^{-2}$	0	0.001
Axonal	Ca_LVAst	gCa_LVAstbar	Uniform	$S\,cm^{-2}$	0	0.01
Axonal	CaDynamics_E2	gamma	Uniform		0.0005	0.05
Axonal	CaDynamics_E2	decay	Uniform	ms	20	1000
Somatic	NaTs2_t	gNaTs2_tbar	Uniform	$S\,cm^{-2}$	0	1
Somatic	SKv3_1	gSKv3_1bar	Uniform	$S\,cm^{-2}$	0	1
Somatic	SK_E2	gSK_E2bar	Uniform	$S\,cm^{-2}$	0	0.1
Somatic	Ca_HVA	gCa_HVAbar	Uniform	$S\,cm^{-2}$	0	0.001
Somatic	Ca_LVAst	gCa_LVAstbar	Uniform	$S\,cm^{-2}$	0	0.01
Somatic	CaDynamics_E2	gamma	Uniform		0.0005	0.05
Somatic	CaDynamics_E2	decay	Uniform	ms	20	1000
Location	**Mechanism**	**Parameter name**	**Distribution**	**Units**	**Value**	
Global		v_init		mV	−65	
Global		celsius		°C	34	
All		g_pas	Uniform	$S\,cm^{-2}$	3e-05	
All		e_pas	Uniform	mV	−75	
All		cm	Uniform	$\mu F\,cm^{-2}$	1	
All		Ra	Uniform	$\Omega\,cm$	100	
Apical		ena	Uniform	mV	50	
Apical		ek	Uniform	mV	−85	
Apical		cm	Uniform	$\mu F\,cm^{-2}$	2	
Somatic		ena	Uniform	mV	50	
Somatic		ek	Uniform	mV	−85	
Basal		cm	Uniform	$\mu F\,cm^{-2}$	2	
Axonal		ena	Uniform	mV	50	
Axonal		ek	Uniform	mV	−85	
Basal	Ih	gIhbar	Uniform	$S\,cm^{-2}$	8e-05	
Apical	Ih	gIhbar	Exp	$S\,cm^{-2}$	8e-05	
Somatic	Ih	gIhbar	Uniform	$S\,cm^{-2}$	8e-05	

Optimised parameters with bounds are in upper part of the table, lower part lists the frozen parameters with their value. The parameter with exp *distribution is scaled for every morphological segment with the equation* $-0.8696 + 2.087 \cdot e^{0.0031 \cdot d}$ *with d the distance of the segment to the soma.*

these and other protocols to L5PCs *in vitro*. For these cells, the same eFeatures were extracted, and the mean μ_{exp} and standard deviation σ_{exp} calculated. The bAP target values are extracted from Larkum et al. (2001).

For every feature value f_{model}, one objective score is calculated:

$$objective = \left| \frac{\mu_{exp} - f_{model}}{\sigma_{exp}} \right|$$

following Druckmann et al. (2007). This approach normalizes all objective scores to a common scale, allowing them to be combined regardless of units, and weights specific objectives according to feature variability.

4.2.4. Optimisation

For the optimisation of this cell model we needed significantly more computing resources. The goal was to find a solution that has objective values that are within approximately 3 standard

TABLE 2 | List of eFeatures for the L5PC example, obtained from experiments.

Stimulus	Location	eFeature	Mean	Std
Step1	soma	AHP_depth_abs	−60.3636	2.3018
		AHP_depth_abs_slow	−61.1513	2.3385
		AHP_slow_time	0.1599	0.0483
		AP_height	25.0141	3.1463
		AP_width	3.5312	0.8592
		ISI_CV	0.109	0.1217
		adaptation_index2	0.0047	0.0514
		doublet_ISI	62.75	9.6667
		mean_frequency	6	1.2222
		time_to_first_spike	27.25	5.7222
Step2	soma	AHP_depth_abs	−59.9055	1.8329
		AHP_depth_abs_slow	−60.2471	1.8972
		AHP_slow_time	0.1676	0.0339
		AP_height	27.1003	3.1463
		AP_width	2.7917	0.7499
		ISI_CV	0.0674	0.075
		adaptation_index2	0.005	0.0067
		doublet_ISI	44.0	7.1327
		mean_frequency	8.5	0.9796
		time_to_first_spike	19.75	2.8776
Step3	soma	AHP_depth_abs	−57.0905	2.3427
		AHP_depth_abs_slow	−61.1513	2.3385
		AHP_slow_time	0.1968	0.0112
		AP_height	19.7207	3.7204
		AP_width	3.5347	0.8788
		ISI_CV	0.0737	0.0292
		adaptation_index2	0.0055	0.0015
		doublet_ISI	22.75	4.14
		mean_frequency	17.5	0.8
		time_to_first_spike	10.5	1.36
bAP	dend1	AP_amplitude_from_voltagebase	45	10
	dend2	AP_amplitude_from_voltagebase	36	9.33
	soma	AP_height	25.0	5.0
		AP_width	2.0	0.5
		Spikecount	1.0	0.01

Locations dend1 and dend2 are respectively 660 and 800 μm from the soma in the apical trunk. Depending on the eFeature type the units can be mV or ms.

deviations from the experimental mean. For this we ran 100 generations with an offspring size of the genetic algorithm of 100 individuals (**Figure 6D**). The evaluation of these 100 individuals was parallelized over 50 CPU cores (Intel Xeon 2.60 GHz) using SCOOP, and took about 4 h to run.

A diverse set of acceptable solutions was found (**Figure 6C**). **Figure 6A** shows the 10 best solutions in the hall of fame (based on the sum of the objectives). The best solution has objective values that are in the same range as the reference model (**Figures 5C**, **6B**). The profile of the objective scores is not the same for these two models, showing that there are multiple solutions that match the experimental data to a similar degree. **Figure 7** shows a comparison of the optimised model to its reference under Gaussian noise current injection (not used during the optimisation).

4.3. Spike-Timing Dependent Plasticity (STDP) Model

The BluePyOpt framework was designed to be versatile and broadly applicable to a wide range of neuroscientific optimisation problems. In this use case, we demonstrate this versatility by using BluePyOpt to optimise the parameters of a calcium-based STDP model (Graupner and Brunel, 2012) to summary statistics

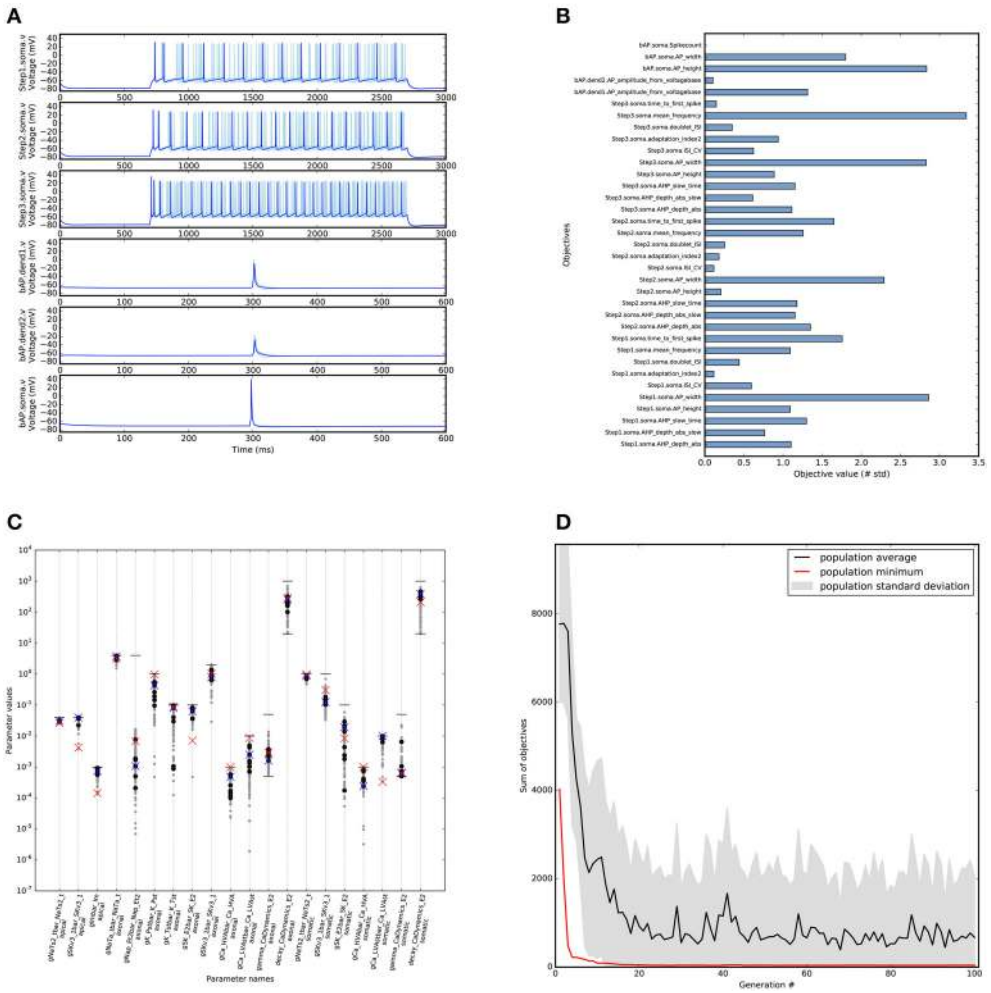

FIGURE 6 | Results of optimising L5PC model using BluePyOpt. Similar to **Figure 5B**, with the top ten objective values found by BluePyOpt, and the best one plotted darker **(B)** Objective scores for the best objective values found by BluePyOpt. The goal of the algorithm is to minimize these values. **(C)** Parameter diversity in the solution space. Parameter values shown for reference model (red crosses), best (blue crosses), 10 best individuals (black dots) and all individuals with all objectives below 5 (gray dots). Horizontal lines (black) represent the bounds of the parameters, when lower line is missing the bound is 0. **(D)** Evolution of the L5PC optimisation that found the model in **(A)**. Plot shows the minimal, maximal and average scores found in the consecutive generations of the evolutionary algorithm.

from *in vitro* experiments (Nevian and Sakmann, 2006). That is, we show how to fit the model to literature data, commonly reported just as mean and SEM of the amount of potentiation (depression) induced by one or more stimulation protocols.

In the set of experiments performed by Nevian and Sakmann (2006), a presynaptic action potential (AP) is paired with a burst of three post-synaptic APs to induce either long-term potentiation (LTP) or long-term depression (LTD) of the postsynaptic neuron response. The time difference Δt between the presynaptic AP and the postsynaptic burst determines the direction of change: A burst shortly preceding the presynaptic AP causes LTD, with a peak at $\Delta t = -50$ ms; conversely, a burst shortly after the presynaptic AP results in LTP, with a peak at $\Delta t = +10$ ms (Nevian and Sakmann, 2006).

The model proposed by Graupner and Brunel (2012) assumes bistable synapses, with plasticity of their absolute efficacies governed by post-synaptic calcium dynamics. That is, each synapse is either in an high-conductance state or a low-conductance state; potentiation and depression translate then into driving a certain fraction of synapses from the low-conductance state to the high-conductance state and vice versa; synapses switch from one state to another depending on the time spent by post-synaptic calcium transients above a potentiation (depression) threshold. Following Graupner and Brunel (2012), the model is described as

$$\tau \frac{d\rho}{dt} = -\rho(1-\rho)(\rho_\star - \rho) + \gamma_p(1-\rho)\Theta[c(t)-\theta_p] - \gamma_d \rho \Theta[c(t)-\theta_d] + \text{Noise}(t) \quad (1)$$

$$\frac{dc}{dt} = -\frac{c}{\tau_{\text{Ca}}} + C_{\text{pre}} \sum_i \delta(t - t_i - D) + C_{\text{post}} \sum_j \delta(t - t_j) \quad (2)$$

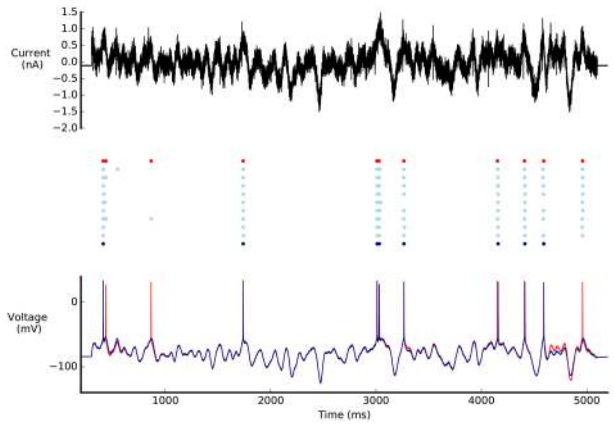

FIGURE 7 | Comparison of L5PC model solutions found by BluePyOpt to reference model. Top: Gaussian noise current injected in the models. Middle: Raster plot of model responses to noise current injection. Bottom: Voltage response of the models to noise current injection. In red, model parameters from Markram et al. (2015) model, in light blue the best 10 individuals found by BluePyOpt, in dark blue the best individual. Figure as in Pozzorini et al. (2015).

where ρ is the absolute synaptic efficacy, $\rho_\star$ delimits the basins of attraction of the potentiated and depressed state, γ_p (γ_d) is the potentiation (depression) rate, Θ is the Heaviside function, θ_p (θ_d) is the potentiation (depression) threshold, Noise(t) is an activity dependent noise. The postsynaptic calcium concentration is described by the process c, with time constant τ_{Ca}. C_{pre} is the calcium transient caused by a presynaptic spike occurring at time t_i, with a delay D to account for the slow activation of NMDARs, while C_{post} is the calcium transient caused by a postsynaptic spike occurring at time t_j.

For periodic stimulation protocols, such as in Nevian and Sakmann (2006), the synaptic transition probability can be easily calculated analytically (Graupner and Brunel, 2012), allowing estimation of the amount of potentiation (depression) induced by the stimulation protocol without actually running any neuron simulations. The amount of potentiation (depression) obtained with different protocols *in vitro* become the objectives of the optimisation.

A small Python module *stdputil* calculating this model is available in the example section on the BluePyOpt website. To optimise this model, only an *Evaluator* class has to be defined that implements an evaluation function:

```
class GraupnerBrunelEvaluator(bpop.evaluators.
                    Evaluator):
    def __init__(self):
        super(GraupnerBrunelEvaluator,
            self).__init__()
        # Graupner-Brunel model parameters and
            boundaries
        # From Graupner and Brunel (2012)
        self.graup_params = [('tau_ca', 1e-3,
            100e-3),
                             ('C_pre', 0.1, 20.0),
                             ('C_post', 0.1, 50.0),
                             ('gamma_d', 5.0, 5000.0),
                             ('gamma_p', 5.0, 2500.0),
                             ('sigma', 0.35, 70.7),
                             ('tau', 2.5, 2500.0),
                             ('D', 0.0, 50e-3),
                             ('b', 1.0, 100.0)]

        self.params = [bpop.parameters.Parameter
                       (param_name, bounds=(min_bound,
                           max_bound))
                       for param_name, min_bound,
                           max_bound in self.
                       graup_params]

        self.param_names = [param.name for param in
            self.params]

        self.protocols, self.sg, self.stdev,
            self.stderr = \
           stdputil.load_neviansakmann()

        self.objectives = [bpop.objectives.Objective
                           (protocol.prot_id)
                       for protocol in self.protocols]

    def get_param_dict(self, param_values):
        return gbParam(zip(self.param_names,
            param_values))

    def compute_synaptic_gain_with_lists(self,
         param_values):
        param_dict =
            self.get_param_dict(param_values)

        syn_gain = [stdputil.protocol_outcome
                    (protocol, param_dict) \
                   for protocol in self.protocols]

        return syn_gain

    def evaluate_with_lists(self, param_values):
        param_dict =
            self.get_param_dict(param_values)

        err = []
        for protocol, sg, stderr in
            zip(self.protocols, self.sg,
            self.stderr):
           res = stdputil.protocol_outcome(protocol,
               param_dict)

           err.append(numpy.abs(sg - res) / stderr)
        return err
```

With the evaluator defined, running the optimisation becomes as simple as:

```
evaluator = GraupnerBrunelEvaluator()

opt = bpop.deapext.optimisations.
    IBEADEAPOptimisation(GraupnerBrunelEvaluator())
results = opt.run(max_ngen=200)
```

Figure 8 shows the results of the optimisation. As in the other use cases, a large set of acceptable solutions are found by the algorithm.

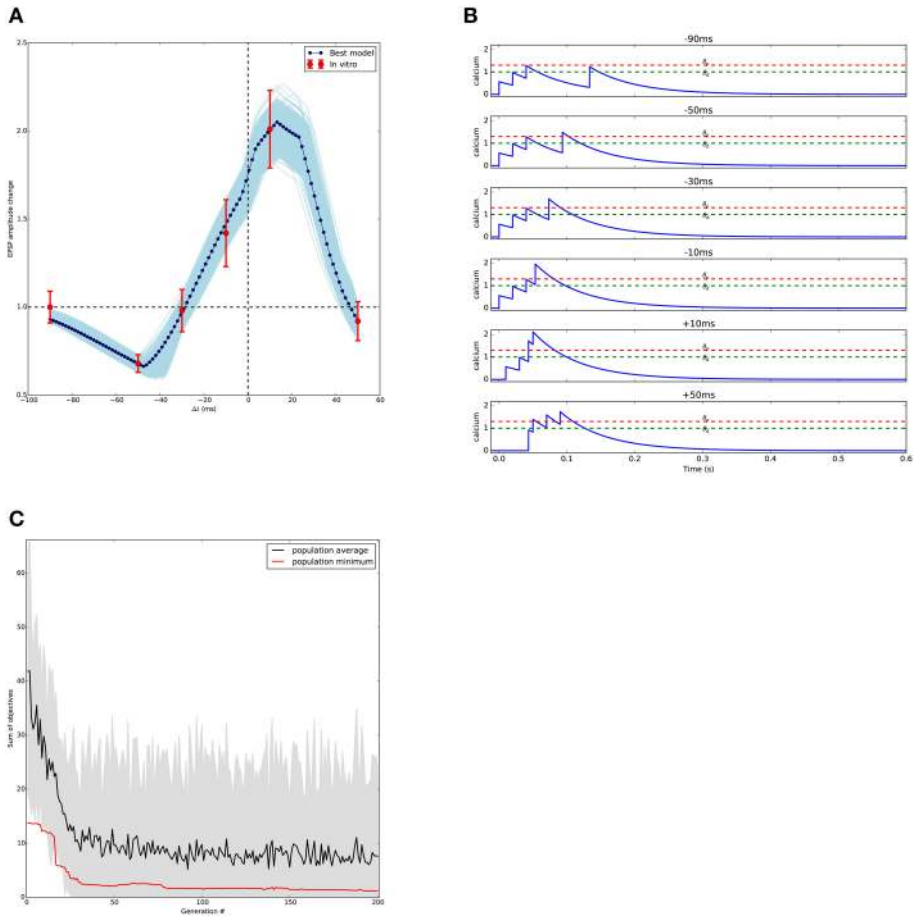

FIGURE 8 | Results of STDP fitting. (A) Comparison between model and experimental results; the models match the available *in vitro* data and predict the outcome of the missing points. In light blue, models generated by individuals having fitness values within one standard error of the mean from experimental *in vitro* data. In dark blue, best model, defined as the closest to all experimental data points. Experimental data from Nevian and Sakmann (2006) digitized using Rohatgi (2015). **(B)** Calcium transients generated by the best model **(A)** for each stimulation protocol. Potentiation and depression thresholds, θ_p and θ_d respectively, are indicated by the dashed lines. **(C)** Evolution of the STDP optimisation that found the model in **(A)**. Minimal and average scores found in the consecutive generations of the evolutionary algorithm.

5. DISCUSSION

BluePyOpt was designed to be a state-of-the-art tool for neuroscientific model parameter search problems that is both easy to use for inexperienced users, and versatile and broadly applicable for power users. Three example use cases were worked through in the text to demonstrate how BluePyOpt serves each of these user communities.

From a software point of view, this dual goal was achieved by an object oriented architecture which abstracts away the domain-specific complexities of search algorithms and simulators, while allowing extension and modification of the implementation and settings of an optimisation. Python was an ideal implementation language for such an architecture, with its very open and minimal approach to extending existing implementations. Object oriented programming allows users to define new subclasses of existing BluePyOpt API classes with different implementations. The *duck typing* of Python allows parameters and objectives to have any kind of type, e.g., they don't have to be floating point numbers. In extreme cases, function implementations can even be overwritten at run time by *monkey patching*. These features of Python give extreme flexibility to the user, which will make BluePyOpt applicable to many use cases.

A common issue arising for users of optimisation software is the configuration of computing infrastructure. The fact that BluePyOpt is coded in Python, an interpreted language, and provides Ansible scripts for its installation, makes it straightforward to run on diverse computing platforms. This will give the user the flexibility to pick the computing infrastructure which best fits their needs, be it their desktop computer, university cluster or temporarily rented cloud infrastructure, such as offered by Amazon Web Services.

This present paper focuses on the use of BluePyOpt as an optimisation tool. It is worth noting that the application domain of BluePyOpt needn't remain limited to this. The ephys model abstraction can also be used in validation, assessing

generalization, and parameter sensitivity analyses. E.g., when applying a map function to an ephys model evaluation function which takes as input a set of morphologies, one can measure how well the model generalizes when applied to different morphologies. The present paper expressly does not touch on issues of generalization power, overfitting, or uniqueness of solution. It is worth now making a few points on the latter. While BluePyOpt could successfully optimise the three examples, **Figures 4B**, **6C** show a diversity of solutions giving good fitness values. That is, for these neuron model optimisation problems, the solutions found are non-unique. This is compatible with the observation that Nature itself also utilizes various and non-unique solutions to provide the required phenotype (Schulz et al., 2007; Taylor et al., 2009). For other problems solutions could be unique, making BluePyOpt useful e.g., for extracting parameters for models of synapse dynamics (Fuhrmann et al., 2002).

While BluePyOpt significantly reduces the domain specific knowledge required to employ parameter optimisation strategies, some thought from the user in setting up their problem is still required. For example, BluePyOpt does in principle allow brute force optimisation of all parameters of the L5PC model example, including channel kinetics parameters and passive properties, but such an approach would almost certainly be unsuccessful. Moreover, when it comes to assessing fitness of models, care and experience is also required to avoid the optimisation getting caught in local minima, or cannibalizing one objective for another. For neuron models for example, feature-based approaches coupled with multi-objective optimisation strategies have proven especially effective (Druckmann et al., 2007). Indeed, even the stimuli and features themselves can be optimised on theoretical grounds to improve parameter optimisation outcomes (Druckmann et al., 2011). For these reasons, an important companion of BluePyOpt will be a growing library of working optimisation examples developed by domain experts for a variety of common use cases, to help inexperienced users quickly adopt a working strategy most closely related to their specific needs.

As this examples library grows, so too will the capabilities of BluePyOpt evolve. Some improvements planned for the future include the following:

Support for multi-stage optimisations allowing for example the passive properties of a neuron to be optimised in a first stage, prior to optimising the full-active dendritic parameters in a second phase

Embedded optimisation allowing for example an optimisation of a "current at rheobase" feature requiring threshold detection during the optimisation using e.g., a binary search. Also, for integrate-and-fire models such as the adapting exponential integrate-and-fire (Brette and Gerstner, 2005), a hybrid of a global stochastic search and local gradient descent has been shown to be a competitive approach (Jolivet et al., 2008)

Fast pre-evaluation of models to exclude clearly bad parameters before computation time is wasted on them

Support for evaluation time-outs to protect against optimisations getting stuck in long evaluations, for example when using NEURON's CVODE solver, which can occasionally get stuck at excessively high resolutions.

Support for explicit units to make optimisation scripts more readable, and sharing with others less error prone.

Although parameter optimisations can require appreciable computing resources, the ability to share the code of an optimisation through a light-weight script or ipython notebook using BluePyOpt will improve reproducibility in the field. It allows for neuroscientists to exchange code and knowledge about search algorithms that perform well for particular models. In the future, making it possible for users to read and write model descriptions from community standards (Raikov et al., 2011; Cannon et al., 2014), could further ease the process of plugging in a model into a BluePyOpt optimisation. By providing the neuroscientific community with BluePyOpt, an open source tool to optimise model parameters in Python which is powerful, easy to use and broadly applicable, we hope to catalyse community uptake of state-of-the-art model optimisation approaches, and encourage code sharing and collaboration.

DOWNLOADS

The source code of BluePyOpt, example scripts, cloud installation scripts, documentation and a list of the software dependencies are available on Github at https://github.com/BlueBrain/BluePyOpt, the former under the GNU Lesser General Public License version 3 (LGPLv3), and the latter two under a BSD license.

AUTHOR CONTRIBUTIONS

WV, MG, and JC designed the software and contributed code. WV, GC, MG, and CR designed the examples and contributed code. WV, EM, MG, and GC wrote the manuscript. All: Conception and design, drafting and revising, and final approval.

FUNDING

The work was supported by funding from the EPFL to the Laboratory of Neural Microcircuitry (LNMC) and funding from the ETH Domain for the Blue Brain Project (BBP). Additional support was provided by funding for the Human Brain Project from the European Union Seventh Framework Program (FP7/2007- 2013) under grant agreement no. 604102 (HBP). The BlueBrain IV BlueGene/Q and Linux cluster used as a development system for this work is financed by ETH Board Funding to the Blue Brain Project as a National Research Infrastructure and hosted at the Swiss National Supercomputing Center (CSCS).

ACKNOWLEDGMENTS

We wish to thank Michael Graupner for his support with the implementation of the calcium-based STDP model (Graupner and Brunel, 2012) and for fruitful discussions, and Elisabetta Iavarone for testing the cloud installation functionality.

REFERENCES

Achard, P., and De Schutter, E. (2006). Complex parameter landscape for a complex neuron model. *PLoS Comput. Biol.* 2:e94. doi: 10.1371/journal.pcbi.0020094

Bahl, A., Stemmler, M. B., Herz, A. V., and Roth, A. (2012). Automated optimization of a reduced layer 5 pyramidal cell model based on experimental data. *J. Neurosci. Methods* 210, 22–34. doi: 10.1016/j.jneumeth.2012.04.006

Bhalla, U. S., and Bower, J. M. (1993). Exploring parameter space in detailed single neuron models: simulations of the mitral and granule cells of the olfactory bulb. *J. Neurophysiol.* 69, 1948–1965.

Bleuler, S., Laumanns, M., Thiele, L., and Zitzler, E. (2003). "PISA — a platform and programming language independent interface for search algorithms," in *Evolutionary Multi-Criterion Optimization (EMO 2003), Lecture Notes in Computer Science*, eds C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele (Berlin: Springer), 494–508.

Blue Brain Project (2015). eFEL. Available online at: https://github.com/BlueBrain/eFEL (Accessed February 16, 2016).

Blue Brain Project (2016). BluePyOpt. Available online at: https://github.com/BlueBrain/BluePyOpt (Accessed February 16, 2016).

Brette, R., and Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol.* 94, 3637–3642. doi: 10.1152/jn.00686.2005

Cannon, R. C., Gleeson, P., Crook, S., Ganapathy, G., Marin, B., Piasini, E., et al. (2014). LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning neuroml 2. *Front. Neuroinform.* 8:79. doi: 10.3389/fninf.2014.00079

Carlson, K. D., Nageswaran, J. M., Dutt, N., and Krichmar, J. L. (2015). An efficient automated parameter tuning framework for spiking neural networks. *Front. Neurosci.* 8:10. doi: 10.3389/fnins.2014.00010

Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge University Press.

Carnevale, N. T., and Hines, M. L. (2008). Units used in neuron. Available online at: https://www.neuron.yale.edu/neuron/static/docs/units/unitchart.html (Accessed February 26, 2016).

Chef (2009). Open Source Chef. Available online at: http://www.chef.io (Accessed February 16, 2016).

Dalcín, L., Paz, R., and Storti, M. (2005). MPI for Python. *J. Parall. Distrib. Comput.* 65, 1108–1115. doi: 10.1016/j.jpdc.2005.03.010

Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.* 2:11. doi: 10.3389/neuro.11.011.2008

DEAP Project (2016). DEAP documentation. (accessed February 26, 2016).

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *Trans. Evol. Comp* 6, 182–197. doi: 10.1109/4235.996017

Druckmann, S., Banitt, Y., Gidon, A., Schürmann, F., Markram, H., and Segev, I. (2007). A novel multiple objective optimization framework for constraining conductance-based neuron models by experimental data. *Front. Neurosci.* 1:7–18. doi: 10.3389/neuro.01.1.1.001.2007

Druckmann, S., Berger, T. K., Schrmann, F., Hill, S., Markram, H., and Segev, I. (2011). Effective stimuli for constructing reliable neuron models. *PLoS Comput. Biol.* 7:e1002133. doi: 10.1371/journal.pcbi.1002133

Eliasmith, C., Gosmann, J., and Choo, X. (2016). Biospaun: a large-scale behaving brain model with complex neurons. *arXiv preprint arXiv:1602.05220*.

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2009). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2:12. doi: 10.3389/neuro.11.012.2008

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., and Gagné, C. (2012). DEAP: evolutionary algorithms made easy. *J. Mach. Learn. Res.* 13, 2171–2175.

Friedrich, P., Vella, M., Gulys, A. I., Freund, T. F., and Kli, S. (2014). A flexible, interactive software tool for fitting the parameters of neuronal models. *Front. Neuroinform.* 8:63. doi: 10.3389/fninf.2014.00063

Fuhrmann, G., Segev, I., Markram, H., and Tsodyks, M. (2002). Coding of temporal information by activity-dependent synapses. *J. Neurophysiol.* 87, 140–148. doi: 10.1152/jn.00258.2001

Goodman, D. F. M., and Brette, R. (2009). The brian simulator. *Front. Neurosci.* 3:192–197. doi: 10.3389/neuro.01.026.2009

Graupner, M., and Brunel, N. (2012). Calcium-based plasticity model explains sensitivity of synaptic changes to spike pattern, rate, and dendritic location. *Proc. Natl. Acad. Sci. U.S.A.* 109, 3991–3996. doi: 10.1073/pnas.11093 59109

Gurkiewicz, M., and Korngreen, A. (2007). A numerical approach to ion channel modelling using whole-cell voltage-clamp recordings and a genetic algorithm. *PLoS Comput. Biol.* 3:e169. doi: 10.1371/journal.pcbi.0030169

Hansen, N., and Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evol. Comput.* 9, 159–195. doi: 10.1162/106365601750190398

HashiCorp (2010). Vagrant. Available online at: http://www.vagrantup.com/ (Accessed February 16, 2016).

Hay, E., Hill, S., Schürmann, F., Markram, H., and Segev, I. (2011). Models of neocortical layer 5b pyramidal cells capturing a wide range of dendritic and perisomatic active properties. *PLoS Comput. Biol.* 7:e1002107. doi: 10.1371/journal.pcbi.1002107

Hines, M., Davison, A. P., and Muller, E. (2009). NEURON and Python. *Front. Neuroinform.* 3:1. doi: 10.3389/neuro.11.001.2009

Hold-Geoffroy, Y., Gagnon, O., and Parizeau, M. (2014). "Once you SCOOP, no need to fork," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment* (New York, NY: ACM), 60. doi: 10.1145/2616498.2616565

Huys, Q. J., and Paninski, L. (2009). Smoothing of, and parameter estimation from, noisy biophysical recordings. *PLoS Comput. Biol.* 5:e1000379. doi: 10.1371/journal.pcbi.1000379

Izhikevich, E. M., and Edelman, G. M. (2008). Large-scale model of mammalian thalamocortical systems. *Proc. Natl. Acad. Sci. U.S.A.* 105, 3593–3598. doi: 10.1073/pnas.0712231105

Jolivet, R., Schürmann, F., Berger, T. K., Naud, R., Gerstner, W., and Roth, A. (2008). The quantitative single-neuron modeling competition. *Biol. Cybern.* 99, 417–426. doi: 10.1007/s00422-008-0261-x

Kenny, J., and Eberhart, R. (1995). "Particle swarm optimization," in *Proceedings of IEEE International Conference Neural Networks*, Vol. 4 (Perth), 1942–1948.

Kole, M. H., Hallermann, S., and Stuart, G. J. (2006). Single Ih channels in pyramidal neuron dendrites: properties, distribution, and impact on action potential output. *J. Neurosci.* 26, 1677–1687. doi: 10.1523/JNEUROSCI.3664-05.2006

Larkum, M. E., Zhu, J. J., and Sakmann, B. (2001). Dendritic mechanisms underlying the coupling of the dendritic with the axonal action potential initiation zone of adult rat layer 5 pyramidal neurons. *J. Physiol.* 533, 447–466. doi: 10.1111/j.1469-7793.2001.0447a.x

Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 456–492. doi: 10.1016/j.cell.2015.09.029

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., et al. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 668–673. doi: 10.1126/science.1254642

Muller, E., Bednar, J. A., Diesmann, M., Gewaltig, M.-O., Hines, M., and Davison, A. P. (2015). Python in neuroscience. *Front. Neuroinform.* 9:11. doi: 10.3389/fninf.2015.00011

Nevian, T., and Sakmann, B. (2006). Spine Ca2+ signaling in spike-timing-dependent plasticity. *J. Neurosci.* 26, 11001–11013. doi: 10.1523/JNEUROSCI.1749-06.2006

Oliphant, T. E. (2007). Python for scientific computing. *Comput. Sci. Eng.* 9, 10–20. doi: 10.1109/MCSE.2007.58

Pérez, F., and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Comput. Sci. Eng.* 9, 21–29. doi: 10.1109/MCSE.2007.53

Pozzorini, C., Mensi, S., Hagens, O., Naud, R., Koch, C., and Gerstner, W. (2015). Automated high-throughput characterization of single neurons by means of simplified spiking models. *PLoS Comput. Biol.* 11:e1004275. doi: 10.1371/journal.pcbi.1004275

Puppet Labs (2005). Puppet. Availble online at: http://puppetlabs.com/ (Accessed February 16, 2016).

Raikov, I., Cannon, R., Clewley, R., Cornelis, H., Davison, A., De Schutter, E., et al. (2011). NineML: the network interchange for neuroscience modeling language. *BMC Neurosci.* 12(Suppl. 1):P330. doi: 10.1186/1471-2202-12-S1-P330

Ramaswamy, S., Courcol, J.-D., Abdellah, M., Adaszewski, S. R., Antille, N., Arsever, S., et al. (2015). The neocortical microcircuit collaboration portal: a resource for rat somatosensory cortex. *Front. Neural Circuits* 9:44. doi: 10.3389/fncir.2015.00044

Ray, S., and Bhalla, U. S. (2008). PyMOOSE: interoperable scripting in python for moose. *Front. Neuroinform.* 2:6. doi: 10.3389/neuro.11.006.2008

Red Hat, Inc. (2012). Ansible. Available online at: http://www.ansible.com/ (Accessed February 16, 2016).

Rohatgi, A. (2015). Webplotdigitizer, Version: 3.9. Available online at: http://arohatgi.info/WebPlotDigitizer (Accessed February 26, 2016).

Schmücker, N. (2010). *Advancing Automated Parameter Constraining on Parallel Architectures for Neuroscientic Applications*. Bachelor thesis, Osnabrück University.

Schulz, D. J., Goaillard, J.-M., and Marder, E. E. (2007). Quantitative expression profiling of identified neurons reveals cell-specific constraints on highly variable levels of gene expression. *Proc. Natl. Acad. Sci. U.S.A.* 104, 13187–13191. doi: 10.1073/pnas.0705827104

Sivagnanam, S., Majumdar, A., Yoshimoto, K., Carnevale, N. T., Astakhov, V., Bandrowski, A., et al. (2013). "Introducing the neuroscience gateway," in *Proceedings International Workshop on Science Gateways* (Zurich), 3–5.

Stefanou, S. S., Kastellakis, G., and Poirazi, P. (2016). "Creating and constraining compartmental models of neurons using experimental data," in *Advanced Patch-Clamp Analysis for Neuroscientists*, ed A. Korngreen (New York, NY: Springer), 325–343. doi: 10.1007/978-1-4939-3411-9_15

Svensson, C.-M., Coombes, S., and Peirce, J. W. (2012). Using evolutionary algorithms for fitting high-dimensional models to neuronal data. *Neuroinformatics* 10, 199–218. doi: 10.1007/s12021-012-9140-7

Tarantola, A. (2016). Inverse problem. From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein. (Accessed February 26, 2016).

Taylor, A. L., Goaillard, J.-M., and Marder, E. (2009). How multiple conductances determine electrophysiological properties in a multicompartment model. *J. Neurosci.* 29, 5573–5586. doi: 10.1523/JNEUROSCI.4438-08.2009

Van Geit, W., Achard, P., and De Schutter, E. (2007). Neurofitter: a parameter tuning package for a wide range of electrophysiological neuron models. *Front. Neuroinform.* 1:1. doi: 10.3389/neuro.11.001.2007

Van Geit, W., De Schutter, E., and Achard, P. (2008). Automated neuron model optimization techniques: a review. *Biol. Cybern.* 99, 241–251. doi: 10.1007/s00422-008-0257-6

Vanier, M. C., and Bower, J. M. (1999). A comparative survey of automated parameter-search methods for compartmental neural models. *J. Comput. Neurosci.* 7, 149–171. doi: 10.1023/A:1008972005316

Wils, S., and De Schutter, E. (2009). STEPS: modeling and simulating complex reaction-diffusion systems with python. *Front. Neuroinform.* 3:15. doi: 10.3389/neuro.11.015.2009

Zaytsev, Y. V., and Morrison, A. (2014). CyNEST: a maintainable Cython-based interface for the NEST simulator. *Front. Neuroinform.* 8:23. doi: 10.3389/fninf.2014.00023

ZeroMQ Project (2007). ZeroMQ. Available online at: http://zeromq.org (Accessed February, 16, 2016).

Zitzler, E., and Künzli, S. (2004). "Indicator-based selection in multiobjective search," in *Parallel Problem Solving from Nature (PPSN VIII)*, eds X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tiňo, A. Kabán, and H.-P. Schwefel (Berlin: Springer-Verlag), 832–842.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.