Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Application Notes

OXFORD

System Biology

Python Interfaces for the Smoldyn Simulator

Dilawar Singh [1], Steven S. Andrews [2*]

[1] Subconscious Compute Pvt. Ltd. Bangalore, India, and
[2] Department of Bioengineering, University of Washington, Seattle, WA, US.

* To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Smoldyn is a particle-based biochemical simulator that is frequently used for systems biology and biophysics research. Previously, users could only define models using text-based input or a C/C++ applicaton programming interface (API), which were convenient, but limited extensibility.
Results: We added a Python API to Smoldyn to improve integration with other software tools such as Jupyter notebooks, other Python code libraries, and other simulators. It includes low-level functions that closely mimic the existing C/C++ API and higher-level functions that are more convenient to use. These latter functions follow modern object-oriented Python conventions.
Availability: Smoldyn is open source and free, available at `http://www.smoldyn.org`, and can be installed with the Python package manager `pip`. It runs on Mac, Windows, and Linux.
Contact: steven.s.andrews@gmail.com
Supplementary information: Documentation is available at `http://www.smoldyn.org` and `https://smoldyn.readthedocs.io`.

1 Introduction

Smoldyn is a biochemical simulator that represents proteins and other molecules of interest as individual spherical particles [4]. These particles diffuse, exclude volume, undergo reactions with each other, and interact with surfaces, much as real molecules do. Smoldyn is notable for its high accuracy, fast performance, and wide range of features [3]. Users typically interact with Smoldyn through a text-based interface [1] but Smoldyn can also be run through a C/C++ application programming interface (API) [2] or as a module within the Virtual Cell [6] or MOOSE simulators [10].

Smoldyn's text-based input method is relatively easy to use, but has the drawbacks of not being a complete programming language and being difficult to interface with other tools. To address these issues, we developed a Python scripting interface for Smoldyn. Python is widely used in science and engineering because it is simple, powerful, and supported by a wide range of software libraries. Also, Python code is interpreted rather than compiled, which allows for interactive use and generally reduces time between development and application. Our Python API enables Smoldyn to be run as a physics engine with other user interfaces, to be linked to complementary simulators to support multi-scale modeling, or to be run within a notebook environment such as Jupyter.

2 Implementation and features

Smoldyn's Python API is assembled in three levels. At the bottom, the previously existing C/C++ API [2], which is written in C, provides access to most of Smoldyn's internal data structures and functions. This API is primarily composed of functions for getting and setting Smoldyn model components, setting simulation parameters, and running simulations. The middle level, written in C++, uses the PyBind11 library [9] to create a Python wrapper for the C/C++ API, thus making all of the C/C++ functions and data accessible from Python. PyBind11 is a simple open-source header-only C++ library that was designed primarily for this task of adding Python bindings to existing C++ code; it takes care of reference counting, object lifetime management, and other basic utilities. The top level or "user API," which is written in Python, exposes a set of Python classes to the user. They offer functions for building and simulating models using object-oriented programming, including standard Python features such as error handling and default parameters. They work by calling the low-level Python API, which calls the C/C++ API.

The classes in the user API represent key model elements. These include a "simulation" class for the entire simulated system, a "species" class for chemical species, a "reaction" class for chemical reactions, a "surface" class for biological membranes or other physical surfaces, a "compartment" class for defined volumes of space, and others. A user creates a model by creating a simulation class first and then adding

© The Author 2020. All rights reserved.

```
# Smoldyn simulation of molecule clustering
import smoldyn
s=smoldyn.Simulation(low=[-10,-10,-10],high=[10,10,10])
B=s.addSpecies('B',color='blue',difc=1,display_size=0.3)
R=s.addSpecies('R',color='red',difc=0,display_size=0.3)
B.addToSolution(200)
R.addToSolution(1,pos=[0,0,0])
rxn=s.addReaction(name='r',subs=[B,R],prds=[R,R],rate=20)
rxn.productPlacement(method='bounce',param=0.6)
smoldyn.addOutputData('data')
s.addCommand(cmd='molcount data',cmd_type='E')
s.setGraphics('opengl_good',1)
s.run(200,dt=0.1)
```

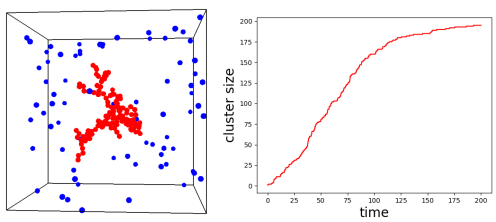

Fig. 1. (Top) Complete Python code for a simple model of molecule clustering in which blue molecules ('B') diffuse freely, but then convert to immobile red molecules ('R') upon collision with a red molecule. (Left) A snapshot of a simulation from this model. (Right) The number of red molecules over time, from the same simulation plus a few lines of graphing code using matplotlib.

components to it, such as species, surfaces, and reactions (Figure 1). Once it's complete, the user tells the simulation class to run the model, typically displaying the results to a graphical output window in the process. An entire simulation is encapsulated by its own object, so users can define multiple simulations at once and even have them interact with each other.

As part of improving extensibility, we included callback functions in the Python user's API. They can be used to update the Smoldyn simulation environment to states that are generated by other software, perhaps while using prior Smoldyn output to help determine that state. For example, we combined Smoldyn with the MOOSE software to simulate a simple model of pre-synaptic vesicle release. Here, MOOSE computes the membrane potential, V_m, which Smoldyn then imports at every 1 ms to update the reaction rates (see the "integrate_with_moose.py" example file in the Smoldyn download package). Internally, callbacks get registered with the Smoldyn code before a simulation starts and are then called by Smoldyn at every n'th cycle through the main simulation loop.

Smoldyn can also compute a wide range of quantitative data during simulations, such as molecule counts in specific regions, radial distribution functions, and tracks of individual molecules. These data are typically written to text files, which can then be loaded into other software and analyzed. Alternatively, the C/C++ and Python APIs also allow quantitative data to be exported to data tables that can then be transferred to other software. For example, simulation results can transferred directly to the matplotlib graphing software (Figure 1).

3 Discussion

The PyBind11 library was an excellent tool for this task. Its small size and headers-only design meant that it could be included with the main code rather than being linked, which improved software robustness and simplified cross-platform compatibility. Also, the fact that it doesn't automate the C++ wrapping process, in contrast to SWIG [5], meant that we needed to define each Python function ourselves; this took more work, but led to better customization in the end.

Smoldyn is written in a combination of C, C++, and Python, which is partly a result of its development history, but also represents good design. The vast majority of Smoldyn's computational effort typically occurs in core algorithms that check for molecule collisions with each other or with surfaces, and that then address those collisions. We were able to make these routines fast and memory-efficient by writing them in C, which has very low computational overhead costs. The C++ code in the C/C++ API works well because of its compatibility with other C++ code, including PyBind11, and is a good intermediate between C and Python. Finally, the Python code in the user's API is easy to use, fast to write and test, and an ideal interface to many other software libraries. Python runs slowly, but this doesn't matter because only a tiny fraction of the total computational effort is spent here.

In some ways, Python has become the universal language of systems biology modeling because it is widely supported by a wide range of simulators, along with many high quality numerical and scientific code libraries. As a result, a single Python script can easily run multiple simulations that interact with each other. However, Python compatibility does not solve the problem of how to run a single model with different simulators because each one requires different Python code. The only viable solution is that simulators need to support systems biology standards for describing models, such as the Systems Biology Markup Language [8] for general systems biology problems, and the MUSIC language [7] for neuroscience modeling. Smoldyn does not support these standards yet but, when we add this capability, the new Python API will simplify the task.

Acknowledgements

SSA thanks Herbert Sauro for helpful discussions and Shawn Garbett for prior work on a Python API for Smoldyn.

References

[1]S. S. Andrews. Spatial and stochastic cellular modeling with the Smoldyn simulator. In *Bacterial Molecular Networks*, pages 519–542. Springer, 2012.

[2]S. S. Andrews. Smoldyn: particle-based simulation with rule-based modeling, improved molecular interaction and a library interface. *Bioinformatics*, 33(5):710–717, 2017.

[3]S. S. Andrews. Particle-based stochastic simulators. *Encyclopedia of Computational Neuroscience*, 10:978–1, 2018.

[4]S. S. Andrews, N. J. Addy, R. Brent, and A. P. Arkin. Detailed simulations of cell biology with smoldyn 2.1. *PLoS Comput Biol*, 6(3):e1000705, 2010.

[5]D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. *Tcl/Tk Workshop*, 43:74, 1996.

[6]A. E. Cowan, I. I. Moraru, J. C. Schaff, B. M. Slepchenko, and L. M. Loew. Spatial modeling of cell signaling networks. In *Methods in cell biology*, volume 110, pages 195–221. Elsevier, 2012.

[7]M. Djurfeldt, J. Hjorth, J. M. Eppler, N. Dudani, M. Helias, T. C. Potjans, U. S. Bhalla, M. Diesmann, J. H. Kotaleski, and Ö. Ekeberg. Run-time interoperability between neuronal network simulators based on the music framework. *Neuroinformatics*, 8(1):43–60, 2010.

[8]M. Hucka, A. Finney, H. M. Sauro, and others. The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[9]W. Jakob, J. Rhinelander, and D. Moldovan. pybind11–seamless operability between c++ 11 and python, 2017.

[10]S. Ray, R. Deshpande, N. Dudani, and U. S. Bhalla. A general biological simulator: the multiscale object oriented simulation environment, MOOSE. *BMC Neuroscience*, 9(Suppl 1):P93, 2008.