Bioinformatics, 2021, 1–3
doi: 10.1093/bioinformatics/btab530
Advance Access Publication Date: 22 July 2021
Applications Note

Systems biology

Python interfaces for the Smoldyn simulator

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

[1]Subconscious Compute Pvt. Ltd., Bangalore, Karnataka 560064, India and [2]Department of Bioengineering, University of Washington, Seattle, WA 98105, USA

*To whom correspondence should be addressed.

Associate Editor: Pier Luigi Martel

Received on December 15, 2020; revised on June 21, 2021; editorial decision on July 13, 2021; accepted on July 21, 2021

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++ application 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 and implementation: 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

Documentation is available at http://www.smoldyn.org/SmoldynManual.pdf and https://smoldyn.readthedocs.io/en/latest/python/api.html.

1 Introduction

Smoldyn is a biochemical simulator that represents proteins and other molecules of interest as individual spherical particles (Andrews *et al.*, 2010). 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 (Andrews, 2018). Users typically interact with Smoldyn through a text-based interface (Andrews, 2012) but Smoldyn can also be run through a C/C++ application programming interface (API) (Andrews, 2017) or as a module within the Virtual Cell (Cowan *et al.*, 2012) or MOOSE simulators (Ray *et al.*, 2008).

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 multiscale 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 (Andrews, 2017), 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 (Jakob *et al.*, 2017) 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

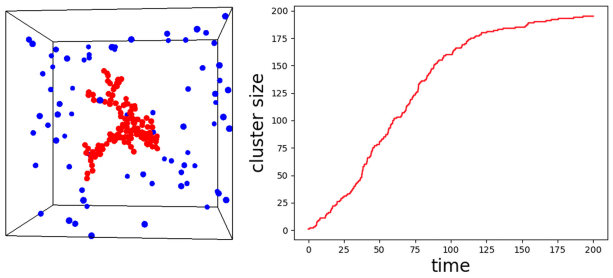

Fig. 1. (Top) Complete Python code for a simple model of molecule clustering in which blue molecules ('B') diffuse freely, but then convert to immobile red molecules ('R') upon collision with a red molecule. (Left) A snapshot of a simulation from this model. (Right) The number of red molecules over time, from the same script (Color version of this figure is available at *Bioinformatics* online.)

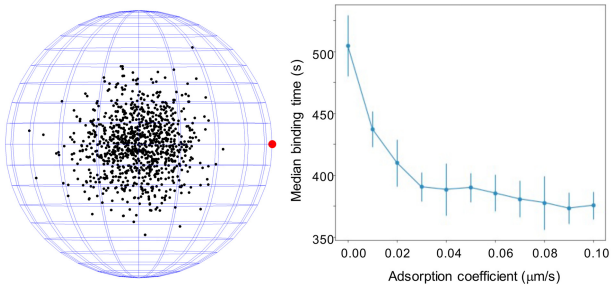

Fig. 2. (Left) Model to investigate reduction of dimensionality effects. 1000 black molecules start at the center of a 10 μm radius sphere and diffuse in the cytoplasm or on the membrane with diffusion coefficient 1 $\mu m^2 s^{-1}$ until reaching the 1 μm radius target, in red. (Right) Median target binding time as a function of membrane adsorption coefficient; error bars represent 1 SD, for 10 runs

and then adding components to it, such as species, surfaces and reactions. 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. Figure 1 illustrates this design with a molecule clustering simulation, showing the Python source code, graphical output and quantitative output.

Figure 2 shows a study of how reduction of dimensionality reduces ligand binding times (Adam and Delbrück, 1968). The Python API simplified this study because we could run many simulations in a row, with different adsorption coefficients and then process and graph the data, all with a single script.

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. 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 compute a wide range of quantitative data during simulations, such as molecule counts in specific regions, radial distribution functions and tracks of individual molecules. Previously, these data could only be written to text files, which could then be loaded into other software and analysed. Now, the C/C++ and Python APIs also allow these data to be exported directly to other software. In both figures, for example, we transferred simulation results directly to the matplotlib graphing software.

The principal limitation of the Python API is that it does not support rule-based modeling using wildcards (Andrews, 2019). Additionally, Python scripts on Macintosh computers that use real-time graphical output stop executing when simulations are complete. This arises from limitations with the OpenGL graphics library versions that are available for Macs.

The Smoldyn download package includes about 15 Python scripts that demonstrate most of the Python API features. They include the three examples described above, which are called 'cluster.py', 'DimensionalityEffect.py' and 'integrate_with_moose.py'.

3 Discussion

A particular benefit of our Python API is that it enables simple interoperability between Smoldyn and existing software libraries. We took advantage of this for the figures shown here, in both cases combining functions from Smoldyn, Numpy and Matplotlib, where the latter two addressed data manipulation and plotting. SciPy and Pandas are other particularly useful libraries for scientific computing. With them, it would be straightforward to, for example, perform statistical inference on biochemical models using the stochastic results from well-defined Smoldyn models and adjust parameters to achieve some optimal simulation result or simulate fluorescence microscopy images from model systems.

Several options are available for adding Python bindings to existing C/C++ APIs, including the Cython language (Behnel *et al.*, 2011), the SWIG automatic conversion tool (Beazley, 1996) and Pybind11 (Jakob *et al.*, 2017). We chose Pybind11 for several reasons. 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 cross-platform compatibility. Also, it did not require adding an additional language to the project. Additionally, it enabled us to customize our API as desired; for example, we defined the 'smoldyn.Simulation' function to accept either boundary values or an input file name, with optional arguments, and it includes a docstring with usage information.

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 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. This code is less efficient than the C code, due to Python being a high-level and interpreted language, but this has a negligible impact on total simulation times 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 (Hucka *et al.*, 2003) for general systems biology problems and the MUSIC language (Djurfeldt *et al.*, 2010) for neuroscience modeling. Smoldyn does not support these standards yet but, when we add this capability, the new Python API will simplify the task.

Financial Support: none declared.

Conflict of Interest: none declared.

Acknowledgement

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

References

Adam,G. and Delbrück,M. (1968) Reduction of dimensionality in biological diffusion processes. *Struct. Chem. Mol. Biol.*, **198**, 198–215.

Andrews,S. (2012) Spatial and stochastic cellular modeling with the Smoldyn simulator. In: *Bacterial Molecular Networks*. Springer, Berlin, Germany, pp. 519–542.

Andrews,S. (2019) Rule-based modeling using wildcards in the Smoldyn simulator. In: *Modeling Biomolecular Site Dynamics*. Springer, Berlin, Germany, pp. 179–202.

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

Andrews,S.S. (2018) Particle-based stochastic simulators. *Encyclopedia Comput. Neurosci.*, **10, 978-1**.

Andrews,S.S. *et al.* (2010) Detailed simulations of cell biology with Smoldyn 2.1. *PLoS Comput. Biol.*, **6**, e1000705.

Beazley,D.M. (1996) SWIG: an easy to use tool for integrating scripting languages with C and C++. *Tcl/Tk Workshop*, **43**, 74.

Behnel,S. *et al.* (2011) Cython: the best of both worlds. *Comput. Sci. Eng.*, **13**, 31–39.

Cowan,A.E. *et al.* (2012) Spatial modeling of cell signaling networks. In: *Methods in Cell Biology. Vol.* **110**, Elsevier, Amsterdam, Netherlands, pp. 195–221.

Djurfeldt,M. *et al.* (2010) Run-time interoperability between neuronal network simulators based on the music framework. *Neuroinformatics*, **8**, 43–60.

Hucka,M. *et al.* (2003) The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models. *Bioinformatics*, **19**, 524–531.

Jakob,W. *et al.* (2017) Pybind11–seamless operability between c++ 11 and python. https://pybind11.readthedocs.io/en/stable/index.html.

Ray,S. *et al.* (2008) A general biological simulator: the multiscale object oriented simulation environment (MOOSE). *BMC Neuroscience*, **9**, P93.