The Journal of Open Source Software

DE-Sim: an object-oriented, discrete-event simulation tool for data-intensive modeling of complex systems in Python

Arthur P. Goldberg[1] **and Jonathan R. Karr**[1]

1 Icahn Institute for Data Science and Genomic Technology and Department of Genetics and Genomic Sciences, Icahn School of Medicine at Mount Sinai, New York, NY 10029, USA

DOI: 10.21105/joss.0XXXX

Software

- Review
- Repository
- Archive

Editor: Editor Name

Submitted: 01 January XXXX
Published: 01 January XXXX

Summary

Recent advances in data collection, storage, and sharing have created unprecedented opportunities to gain insights into complex systems such as the biochemical networks that generate cellular behavior. Understanding the behavior of such systems will likely require larger and more comprehensive dynamical models that are based on a combination of first principles and empirical data. These models will likely represent each component and interaction using mechanistic approximations that are derived from first principles and calibrated with data. For example, dynamical models of biochemical networks often represent the interactions among molecules as chemical reactions whose rates are determined by combining approximations of chemical kinetics and empirically-observed reaction rates. Furthermore, complex models that represent multiple types of components and their interactions will require diverse approximations and large, heterogeneous datasets. New tools are needed to build and simulate such data-intensive models.

One of the most promising methods for building and simulating data-intensive models is discrete-event simulation (DES). DES represents the dynamics of a system as a sequence of instantaneous events (Fishman, 2013). DES is used for a wide range of research, such as studying the dynamics of biochemical networks, characterizing the performance of computer networks, evaluating potential war strategies, and forecasting epidemics (Banks, Carson II, Nelson, & Nicol, 2009). Although multiple DES tools exist, it remains difficult to build and simulate data-intensive models. First, it is cumbersome to create complex models with the low-level languages supported by many of the existing tools. Second, most of the existing tools are siloed from the ecosystems of data science tools that are exploding around Python and R.

To address this problem, we developed DE-Sim (https://github.com/KarrLab/de_sim), an open-source, object-oriented (OO), Python-based DES tool. DE-Sim helps researchers model complex systems by enabling them to use Python's powerful OO features to manage multiple types of components and multiple types of interactions. By building upon Python, DE-Sim also makes it easy for researchers to use Python's powerful data science tools, such as pandas (McKinney & others, 2010) and SciPy (Virtanen et al., 2020), to incorporate large, heterogeneous datasets into comprehensive and detailed models. We anticipate that DE-Sim will enable a new generation of models that capture systems with unprecedented breadth and depth. For example, we are using DE-Sim to develop a multi-algorithmic simulation tool for whole-cell models (Goldberg et al., 2018; Karr et al., 2012; Karr, Takahashi, & Funahashi, 2015) that predict phenotype from genotype by capturing all of the biochemical activity in a cell.

Here, we describe the need for new tools for building and simulating more comprehensive and more detailed models, outline how DE-Sim addresses this need, and present a brief tutorial

that describes how to build and simulate models with DE-Sim. In addition, we summarize the strengths of DE-Sim over existing DES tools, we report the simulation performance of DE-Sim, and we present a case study of how we are using DE-Sim to develop a tool for simulating whole-cell models. Finally, we outline our plans to increase the performance of simulations executed by DE-Sim. Additional examples, tutorials, and documentation are available online, as described in the 'Availability of DE-Sim' section below.

Need for tools for building and simulating data-intensive models

Many scientific fields can now collect detailed data about the components of complex systems and their interactions. For example, deep sequencing has dramatically increased the availability of molecular data about biochemical networks. Combined with advances in computing, we believe that it is now possible to use this data and first principles to create comprehensive and detailed models that can provide new insights into complex systems. For example, deep sequencing and other molecular data can be used to build whole-cell models.

Achieving such comprehensive and detailed models will likely require integrating disparate principles and diverse data. While there are several DES tools, such as SimEvents (Clune, Mosterman, & Cassandras, 2006) and SimPy (Matloff, 2008), and numerous tools for working with large, heterogeneous datasets, such as pandas and SQLAlchemy (Bayer, 2020), it is difficult to use these tools in combination. As a result, despite having all of the major ingredients, it remains difficult to build and simulate data-intensive models.

DE-Sim provides critical features for building and simulating data-intensive models

DE-Sim simplifies the construction and simulation of *discrete-event models* through several features. First, DE-Sim structures discrete-event models as OO programs (Zeigler, 1987). This structure enables researchers to use classes of *simulation objects* to encapsulate the complex logic required to represent each *model component*, and use classes of *event messages* to encapsulate the logic required to describe their *interactions*. With DE-Sim, users define classes of simulation objects by creating subclasses of DE-Sim's simulation object class. DE-Sim simulation object classes can exploit all the features of Python classes. For example, users can encode relationships between the types of components in a model into hierarchies of subclasses of simulation objects. As a concrete example, a model of the biochemistry of RNA transcription and protein translation could be implemented using a superclass that captures the behavior of polymers and three subclasses that represent the specific properties of DNAs, RNAs, and proteins. DE-Sim makes it easy to model complex systems that contain multiple types of components through multiple classes of simulation objects. Users can model arbitrarily many instances of each type of component by creating multiple instances of the corresponding simulation object class.

Second, by building on top of Python, DE-Sim makes it easy for researchers to use Python's extensive suite of data science tools to build models from heterogeneous, multidimensional datasets. For example, researchers can use tools such as H5py, ObjTables (Karr, Liebermeister, Goldberg, Sekar, & Shaikh, 2020), pandas, requests, and SQLAlchemy to retrieve diverse data from spreadsheets, HDF5 files, REST APIs, databases, and other sources; use tools such as NumPy (Oliphant, 2015) to integrate this data into a unified model; and use tools such as LMFIT, Pyomo (Hart, Watson, & Woodruff, 2011), and SciPy to calibrate models. DE-Sim also makes it easy to use many of these same tools to analyze simulation results.

DE-Sim also provides several features to help users execute, analyze, and debug simulations:

- **Stop conditions:** DE-Sim makes it easy to terminate simulations when specific criteria are reached. Researchers can specify stop conditions as functions that return true when the simulation should conclude.
- **Results checkpointing:** DE-Sim makes it easy to record the results of simulations through an easily-configurable checkpointing module.
- **Space-time visualizations:** DE-Sim can generate space-time visualizations of simulation trajectories (Figure 1). These diagrams can help researchers understand and debug simulations.
- **Reproducible simulations:** To help researchers debug simulations, repeated executions of the same simulation with the same configuration and same random number generator seed produce the same results.

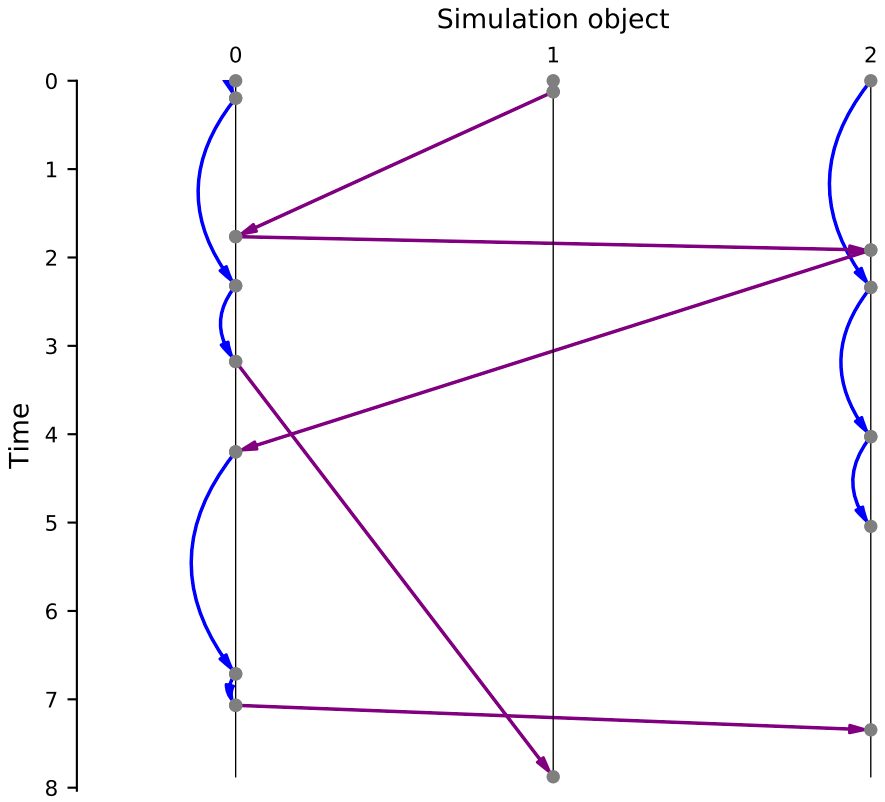

Figure 1: DE-Sim can generate space-time visualizations of simulation trajectories. This figure illustrates a space-time visualization of all of the events and messages in a simulation of the parallel hold (PHOLD) DES benchmark model (Fujimoto, 1990) with three simulation objects. The timeline (black line) for each object shows its events (grey dots). The blue and purple arrows illustrate events scheduled by simulation objects for themselves and other objects, respectively. The code for this simulation is available in the DE-Sim Git repository.

Together, we believe that these features can simplify and accelerate the development of complex, data-intensive models.

Comparison of DE-Sim with existing discrete-event simulation tools

Although there are several other DES tools, we believe that DE-Sim uniquely facilitates data-intensive modeling through a novel combination of OO modeling and support for numerous high-level data science tools. Figure 2 compares the features and characteristics of DE-Sim with some of the most popular DES tools.

Simulation tool	Application domain	Modeling language	Object-oriented models	GUI model builder	Open-source	Latest update
DE-Sim	Complex, data-intensive models	Python	✓		✓	2020
SimEvents	Communications networks and process flows	MATLAB	✓	✓		2020
SimPy	General purpose process-based framework	Python			✓	2018
SIMSCRIPT III	Object-oriented simulation of engineered systems	SIMSCRIPT III	✓			2020
SIMUL8	Business processes	Visual Logic		✓		2020
SystemC	Digital hardware	C++	✓		✓	2018

Figure 2: Comparison of DE-Sim with some of the most popular DES tools. DE-Sim is the only open-source, OO DES tool based on Python. This combination of features makes DE-Sim uniquely suitable for creating and simulating complex, data-intensive models.

SimPy is an open-source DES tool that enables users to use functions to create simulation processes (SimPy's analog to DE-Sim's simulation objects). As another Python-based tool, SymPy also makes it easy for researchers to leverage the Python ecosystem to build models. However, we believe that DE-Sim makes it easier for researchers to build complex models by enabling them to implement models as collections of classes rather than collections of functions. In addition, we believe that DE-Sim is simpler to use because DE-Sim supports a uniform approach for scheduling events, whereas SimPy simulation processes must use two different approaches: one to schedule events for themselves, and another to schedule events for other processes.

SimEvents is a library for DES within the MATLAB/Simulink environment. While SimEvents' graphical interface makes it easy to create simple models, we believe that DE-Sim makes it easier to implement more complex models. By making it easy to use a variety of Python-based data science tools, DE-Sim makes it easier to use data to create models than SimEvents, which builds on a more limited ecosystem of data science tools.

SystemC is a `C++`-based OO DES tool that is frequently used to model digital systems (Mueller et al., 2001). While SystemC provides many of the same core features as DE-Sim, we believe that DE-Sim is more accessible to researchers than SystemC because DE-Sim builds upon Python, which is more popular than `C++` in many fields of research.

SIMSCRIPT III (Rice, Marjanski, Markowitz, & Bailey, 2005) and SIMUL8 (Concannon, Hunter, & Tremble, 2003) are commercial DES tools that enable researchers to implement models using proprietary languages. SIMSCRIPT III is well-suited to modeling decision support systems for domains such as war-gaming, communications networks, transportation, and

manufacturing, and SIMUL8 is well-suited to modeling business processes. However, we believe that DE-Sim is more powerful for most scientific and engineering fields because DE-Sim can leverage Python's robust ecosystem of data science packages.

Tutorial: Building and simulating models with DE-Sim

Users can define and execute a DE-Sim model of a system in three steps: (1) implement event message classes that represent the interactions between the components of the system, (2) implement simulation object classes that represent the states of the components of the system and their actions that initiate and respond to interactions, and (3) use instances of these classes to build and simulate the model. Here, we illustrate these steps with a model of a random walk on the integer number line. The random walk steps forward or backward with equal probability, and takes steps every one or two time units with equal likelihood. This tutorial and additional tutorials are available as interactive Jupyter notebooks at https://sandbox.karrlab.org/tree/de_sim).

1: Create an event message class to represent steps in a random walk.

Create a subclass of `de_sim.EventMessage` to represent steps. Use the instance attribute `step_value` to capture the magnitude and direction of a step. Declare that the `step_value` instance attribute represents the data carried by the message by setting the class attribute `msg_field_names` to a list with the single element `step_value`. The simulation objects which execute steps will use this `step_value` attribute to change the position of the random walk.

```
import de_sim

class RandomStepMessage(de_sim.EventMessage):
    " An event message storing the value of a step of a random walk "
    msg_field_names = ['step_value']
```

The `msg_field_names` attribute is similar to the `field_names` parameter used by Python's `namedtuple` factory function. The parameters to an `EventMessage`'s constructor correspond, in order, to the attributes listed in `msg_field_names`.

2: Define a simulation object class that represents the position of the random walk and schedules and executes random steps.

Simulation objects are like threads, in that a simulation's scheduler decides when to execute them, and their execution is suspended when they have no work to do. But DES simulation objects and threads are scheduled by different algorithms. Whereas a traditional scheduler executes threads whenever they have work to do, a DES scheduler executes simulation objects to ensure that events occur in simulation time order, as stated by the fundamental invariant of discrete-event simulation: all events in a simulation are executed in non-decreasing time order.

By guaranteeing this behavior, the DE-Sim scheduler ensures that causality relationships between events are respected. (The invariant says *non-decreasing* time order, and not *increasing* time order, because events can occur simultaneously, as discussed in DE-Sim's Jupyter notebooks.)

This invariant has two consequences:

- All synchronization between simulation objects is controlled by the simulation times of events, and the execution order rules for simultaneous events.

- Each simulation object executes its events in non-decreasing time order.

The Python classes that generate and handle simulation events are simulation object classes, which are defined as subclasses of `de_sim.SimulationObject`. DE-Sim provides a custom class creation method for `SimulationObject` that gives special meaning to specific methods and attributes.

Below, we define a simulation object class that models a random walk and illustrates the key features of `SimulationObject`.

```
import random

class RandomWalkSimulationObject(de_sim.SimulationObject):
    " A 1D random walk model, with random delays between steps "

    def __init__(self, name):
        super().__init__(name)

    def init_before_run(self):
        " Initialize before a simulation run; called by the simulator "
        self.position = 0
        self.history = {'times': [0],
                        'positions': [0]}
        self.schedule_next_step()

    def schedule_next_step(self):
        " Schedule the next event, which is a step "
        # A step moves -1 or +1 with equal probability
        step_value = random.choice([-1, +1])
        # The time between steps is 1 or 2, with equal probability
        delay = random.choice([1, 2])
        # Schedule an event `delay` time units in the future
        # Schedule the event for this object
        # The event stores a `RandomStepMessage` containing `step_value`
        self.send_event(delay, self, RandomStepMessage(step_value))

    def handle_step_event(self, event):
        " Handle a step event "
        # Update the position and history
        self.position += event.message.step_value
        self.history['times'].append(self.time)
        self.history['positions'].append(self.position)
        self.schedule_next_step()

    # `event_handlers` contains pairs that map each event message class
    # received by this simulation object to the method that handles
    # the event message class
    event_handlers = [(RandomStepMessage, handle_step_event)]

    # messages_sent registers all message types sent by this object
    messages_sent = [RandomStepMessage]
```

Subclasses of `SimulationObject` use the following methods and attributes to initialize themselves, schedule and handle events, and access the simulation's time.

- Methods:
    1. **`init_before_run`** (optional): immediately before a simulation run, after all of the simulation objects have been added to a simulator, the simulator calls each simulation object's `init_before_run` method. In this method, simulation objects can send initial events and perform other initializations. For example, in `RandomWalkSimulationObject`, `init_before_run` schedules the object's first event and initializes its position and history attributes.
    2. **`send_event`**: `send_event(delay, receiving_object, event_message)` schedules an event to occur `delay` time units in the future at simulation object `receiving_object`. `event_message` must be an `EventMessage` instance. An event can be scheduled for any simulation object in a simulation, including the object scheduling the event, as `RandomWalkSimulationObject` does. The event will be executed at its scheduled simulation time by an event handler in the simulation object `receiving_object`. The value of the handler's `event` parameter will be the scheduled event, which contains `event_message` in its `message` attribute. Object-oriented DES terminology often describes the event message as being sent by the sending object at the message's send time (the simulation time when the sending object schedules the event) and being received by the receiving object at the event's receive time (the simulation time when the event is executed). An event message can thus be viewed as a directed edge in simulation space-time from the pair (sending object, send time) to (receiving object, receive time), as illustrated in Figure 1.
    3. **event handlers**: an event handler is a method that handles and executes a simulation event. Event handlers have the signature `event_handler(self, event)`, where `self` is the simulation object that handles (receives) the event, and `event` is a simulation event. A subclass of `SimulationObject` must define at least one event handler, as illustrated by `handle_step_event` in the example above.
- Attributes:
    1. **`event_handlers`**: a simulation object can receive arbitrarily many types of event messages, and implement arbitrarily many event handlers. The attribute `event_handlers` maps event message classes to event handlers. It must contain an iterator over pairs that map each event message class received by a `SimulationObject` subclass to the event handler that handles the event message class. In the example above, `event_handlers` associates `RandomStepMessage` event messages with the `handle_step_event` event handler.
    2. **`messages_sent`**: the types of messages sent by a subclass of `SimulationObject` must be listed in `messages_sent`. It ensures that a simulation object doesn't send messages from the wrong `EventMessage` class.
    3. **`time`**: `time` is a read-only attribute that always equals the current simulation time in a simulation object. For example, a `RandomWalkSimulationObject` saves the value of `time` when recording its history.

3: Use the classes created above to simulate a random walk.

The `de_sim.Simulator` class simulates models. Its `add_object` method adds a simulation object to the simulator. The `initialize` method, which calls each simulation object's `init_before_run` method, must be executed before a simulation starts. At least one simulation object in a simulation must schedule an initial event—otherwise, the simulation cannot start. More generally, a simulation with no events to execute will terminate. A simulator's `run` method simulates a model. It takes the maximum time of a simulation run, and several optional configuration arguments. More information is available in the DE-Sim API documentation at https://docs.karrlab.org/de_sim.

The example below simulates a random walk for `max_time` time units and uses Matplotlib to visualize its trajectory (Figure 3).

```
# Create a simulator
simulator = de_sim.Simulator()

# Create a random walk simulation object and add it to the simulation
random_walk_sim_obj = RandomWalkSimulationObject('rand_walk')
simulator.add_object(random_walk_sim_obj)

# Initialize the simulation
# This executes `init_before_run` in `random_walk_sim_obj`
simulator.initialize()

# Run the simulation until time 10
max_time = 10
simulator.run(max_time)

# Plot the random walk
import matplotlib.pyplot as plt
plt.step(random_walk_sim_obj.history['times'],
         random_walk_sim_obj.history['positions']),
         where='post')
plt.xlabel('Time')
plt.ylabel('Position')
plt.show()
```

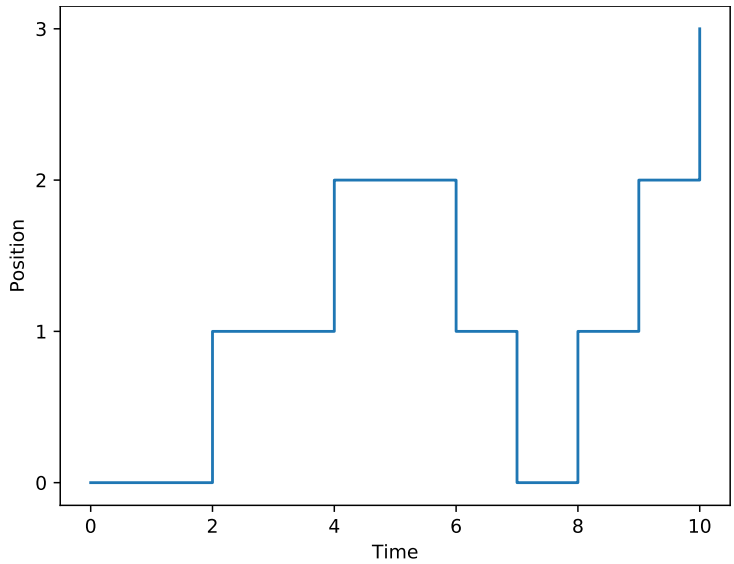

Figure 3: Trajectory of a simulated random walk on the integer number line. The source code for this simulation is available in the DE-Sim Git repository.

Performance of DE-Sim

Figure 4 illustrates the performance of DE-Sim simulating cyclic messaging networks over a range of network sizes. Each messaging network consists of a ring of nodes. When a node handles an event, it schedules the same type of event for its forward neighbor with a one time-unit delay. Each simulation is initialized by sending a message to each node at the first time-unit. The code for this performance test is available in the DE-Sim Git repository, and a Jupyter notebook that runs the test is available at https://sandbox.karrlab.org/tree/de_sim.

		Run time (s)		Event rate (events s^{-1})	
Nodes	Events	Mean	Std dev	Mean	Std dev
4	400	0.05	0.01	8,930	1,930
16	1,600	0.10	0.02	16,531	3,052
64	6,400	0.31	0.04	20,986	2,734
256	25,600	1.22	0.16	21,139	2,597
1,024	102,400	5.34	0.96	19,573	3,465
4,096	409,600	20.06	1.53	20,491	1,506

Figure 4: Performance of DE-Sim simulating a range of sizes of a cyclic messaging network. We executed each simulation for 100 time-units. Each statistic represents the average of three simulation runs in a Docker container running on a 2.9 GHz Intel Core i5 processor.

Case study: a multi-algorithmic simulator for whole-cell models implemented using DE-Sim

We are using DE-Sim to develop WC-Sim (Goldberg & Karr, 2020), a multi-algorithmic simulator for whole-cell models (Goldberg et al., 2018; Karr et al., 2012, 2015). Whole-cell models that predict phenotype from genotype by representing all of the biochemical activity in a cell have great potential to help scientists elucidate the basis of cellular behavior, help bioengineers rationally design biosensors and biomachines, and help physicians personalize medicine.

Due to the diverse timescales of the subsystems inside cells, one promising way to simulate whole-cell models is to co-simulate slow subsystems with fined-grained algorithms and fast subsystems with coarse-grained algorithms. For example, slow subsystems, such as transcription, could be simulated with the Stochastic Simulation Algorithm (SSA, Gillespie (1977)), while faster subsystems, such as signal transduction, could be simulated with ordinary differential equations (ODEs). Metabolism, another fast process, could be simulated with dynamic Flux-Balance Analysis (dFBA, Mahadevan, Edwards, & Doyle III (2002)). However, there are no tools for co-simulating these algorithms beyond ad hoc implementations for specific models.

To accelerate whole-cell modeling, we are using DE-Sim to create WC-Sim. First, we implemented a Python class that tracks the populations of the molecular species represented by a model. Second, we used pandas, NumPy, and ODES scikit (Malengier et al., 2018) to implement simulation object subclasses for dFBA, ODE, and SSA simulations. Third, we used event messages to schedule the execution of these simulation algorithms and coordinate their reading and updating of the species populations.

DE-Sim's OO functionality facilitated the implementation of classes for dFBA, ODE, and SSA, and DE-Sim's event messages streamlined the co-simulation these algorithms. In addition, the ability to use NumPy made it easy to generate the random numbers needed to implement SSA, and the ability to use ODES scikit expedited the use of CVODE (Hindmarsh et al. (2005)) to implement the ODE simulation object class. We anticipate that WC-Sim will help researchers conduct unprecedented simulations of cells.

Conclusion

In summary, DE-Sim is an open-source, object-oriented, discrete-event simulation tool implemented in Python that makes it easier for modelers to create and simulate complex, data-

intensive models. First, DE-Sim enables researchers to conveniently use Python's OO features to manage multiple types of model components and their interactions. Second, DE-Sim enables researchers to directly use Python data science tools, such as pandas and SciPy, and large, heterogeneous datasets to construct models. Together, we anticipate that DE-Sim will accelerate the construction and simulation of unprecedented models of complex systems, leading to new scientific discoveries and engineering innovations.

To further advance the simulation of data-intensive models, we aim to improve the simulation performance of DE-Sim. One potential direction is to use DE-Sim as a specification language for a parallel DES system such as ROSS (Carothers, Bauer, & Pearce, 2000). This combination of DE-Sim and ROSS would enable modelers to both create models with DE-Sim's high-level model specification semantics and quickly simulate models with ROSS.

Availability of DE-Sim

DE-Sim is freely and openly available under the MIT license at the locations below.

- Python package: PyPI: de-sim
- Docker image: Docker Hub: karrlab/de_sim
- Tutorials: Jupyter notebooks at https://sandbox.karrlab.org/tree/de_sim
- Installation instructions and documentation of DE-Sim's API: docs.karrlab.org
- Issue tracker: GitHub: KarrLab/de_sim
- Source code: GitHub: KarrLab/de_sim
- Guide to contributing to DE-Sim developers: GitHub: KarrLab/de_sim
- Code of conduct for developers: GitHub: KarrLab/de_sim
- Continuous integration: CircleCI: gh/KarrLab/de_sim

DE-Sim requires Python 3.7 or higher and pip.

This article discusses version 0.1.1 of DE-Sim.

Acknowledgements

We thank Yin Hoon Chew for her helpful feedback. This work was supported by the National Science Foundation [1649014 to JRK], the National Institutes of Health [R35GM119771 to JRK], and the Icahn Institute for Data Science and Genomic Technology.

References

Banks, J., Carson II, J., Nelson, B., & Nicol, D. (2009). *Discrete-event system simulation*. Pearson. ISBN: 978-0136062127

Bayer, M. (2020). SQLAchemy-the database toolkit for Python. Retrieved from https://www.sqlalchemy.org/

Carothers, C. D., Bauer, D., & Pearce, S. (2000). ROSS: A high-performance, low memory, modular time warp system. *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, *62*, 53–60. doi:10.1109/PADS.2000.847144

Clune, M. I., Mosterman, P. J., & Cassandras, C. G. (2006). Discrete event and hybrid system simulation with simevents. In *Proceedings of the 8th international workshop on discrete event systems* (pp. 386–387).

Concannon, K. H., Hunter, K. I., & Tremble, J. M. (2003). Dynamic scheduling II: SIMUL8-planner simulation-based planning and scheduling. In *Proceedings of the 35th Conference on Winter simulation* (pp. 1488–1493). doi:10.1145/1030818.1031019

Fishman, G. S. (2013). *Discrete-event simulation: Modeling, programming, and analysis*. Springer Science & Business Media.

Fujimoto, R. M. (1990). Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulations* (Vol. 22, pp. 23–28). Retrieved from https://gdo149.llnl.gov/attachments/20776356/24674621.pdf

Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, *81*(25), 2340–2361. doi:10.1021/j100540a008

Goldberg, A. P., & Karr, J. R. (2020). WC-Sim: A multi-algorithmic simulator for whole-cell models. Retrieved from https://github.com/KarrLab/wc_sim

Goldberg, A. P., Szigeti, B., Chew, Y. H., Sekar, J. A., Roth, Y. D., & Karr, J. R. (2018). Emerging whole-cell modeling principles and methods. *Current Opinion in Biotechnology*, *51*, 97–102. doi:10.1016/j.copbio.2017.12.013

Hart, W. E., Watson, J.-P., & Woodruff, D. L. (2011). Pyomo: Modeling and solving mathematical programs in python. *Mathematical Programming Computation*, *3*(3), 219.

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., & Woodward, C. S. (2005). SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, *31*(3), 363–396.

Karr, J. R., Liebermeister, W., Goldberg, A. P., Sekar, J. A. P., & Shaikh, B. (2020). ObjTables: Structured supplementary spreadsheets that promote data quality, reuse, and integration. *arXiv*.

Karr, J. R., Sanghvi, J. C., Macklin, D. N., Gutschow, M. V., Jacobs, J. M., Bolival Jr, B., Assad-Garcia, N., et al. (2012). A whole-cell computational model predicts phenotype from genotype. *Cell*, *150*(2), 389–401. doi:10.1016/j.cell.2012.05.044

Karr, J. R., Takahashi, K., & Funahashi, A. (2015). The principles of whole-cell modeling. *Current Opinion in Microbiology*, *27*, 18–24. doi:10.1016/j.mib.2015.06.004

Mahadevan, R., Edwards, J. S., & Doyle III, F. J. (2002). Dynamic flux balance analysis of diauxic growth in escherichia coli. *Biophysical journal*, *83*(3), 1331–1340.

Malengier, B., Kišon, P., Tocknell, J., Abert, C., Bruckner, F., & Bisotti, M.-A. (2018). ODES: A high level interface to ode and dae solvers. *Journal of Open Source Software*, *3*(22), 165.

Matloff, N. (2008). Introduction to discrete-event simulation and the SimPy language. Retrieved from https://web.cs.ucdavis.edu/~matloff/matloff/public_html/156/PLN/DESimIntro.pdf

McKinney, W., & others. (2010). Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51–56). doi:10.25080/majora-92bf1922-00a

Mueller, W., Ruf, J., Hoffmann, D., Gerlach, J., Kropf, T., & Rosenstiehl, W. (2001). The simulation semantics of SystemC. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition* (pp. 64–70). doi:10.1109/DATE.2001.915002

Oliphant, T. E. (2015). *A guide to NumPy*. CreateSpace Independent Publishing Platform. ISBN: 978-1517300074

Rice, S. V., Marjanski, A., Markowitz, H. M., & Bailey, S. M. (2005). The simscript iii programming language for modular object-oriented simulation. In *Proceedings of the winter simulation conference, 2005.* (pp. 10–pp). IEEE.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., et al. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*(3), 261–272. doi:10.1038/s41592-019-0686-2

Zeigler, B. P. (1987). Hierarchical, modular discrete-event modelling in an object-oriented environment. *Simulation*, *49*(5), 219–230. doi:10.1177/003754978704900506