*Proceedings of the 2021 Winter Simulation Conference*
*S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, eds.*

# A TUTORIAL ON HOW TO CONNECT PYTHON WITH DIFFERENT SIMULATION SOFTWARE TO DEVELOP RICH SIMHEURISTICS

Mohammad Peyman
Pedro Copado
Javier Panadero
Angel A. Juan

Universitat Oberta de Catalunya – IN3
Euncet Business School
156 Rambla del Poblenou
Barcelona, 08018, SPAIN

Mohammad Dehghanimohammadabadi

Northeastern University
Department of Mechanical and Industrial Engineering
360 Huntington Avenue
Boston, Massachusetts, 02115, USA

## ABSTRACT

Simulation is an excellent tool to study real-life systems with uncertainty. Discrete-event simulation (DES) is a common simulation approach to model time-dependent and complex systems. Therefore, there are a variety of commercial (Simio, AnyLogic, Simul8, Arena, etc.) and non-commercial (SimPy, Salabim, etc.) software packages that enable users to take advantage of DES modeling. Although these tools are capable of modeling real-life systems with a high accuracy, they generally fail to conduct advanced analytical analysis (i.e., machine learning, interactive visualizations) or complicated optimization (i.e., simheuristics). Therefore, coupling these DES platforms with external programming languages like Python offers additional mathematical operations and algorithmic flexibility. This integration makes the simulation modeling more intelligent and extends its applicability to a broader range of problems. This study aims to provide a step-wise tutorial for helping simulation users to create intelligent DES models by integrating them with Python. Multiple demo examples are discussed to provide insights and making this connection based on commercial and non-commercial DES packages.

## 1 INTRODUCTION AND RELATED WORK

Simulation packages are a perfect tool to model complex and dynamic systems with non-linear interactions. Several review papers and tutorials are written to enable users building simulation models using a simulation software or languages. Based on a subjective evaluation of parameters, Dias et al. (2016) made three clusters of simulation tools. The first cluster includes ProModel, FlexSim, Simul8, and WITNESS. The second cluster discusses ExtendSim, Simio, PlantSimulation, and AnyLogic. Finally, the third cluster covers SimProcess, AutoMod, Micro Saint, QUEST, Enterprise Dynamics, and Process Model. Arena appears in a different cluster. In another work, Schriber et al. (2012) provided several examples in AutoMod, SLX, ExtendSim, and Simio to guide user with simulation modeling. Besides, Dagkakis and Heavey (2016) reviewed an open-source DES with application in manufacturing, services, supply chain management, and logistics. In addition, authors evaluated and identified the best fitted simulation tools for different modeling needs. Later, Guimarães et al. (2018) developed a new method for selecting the right DES software to help and improve the efficiency of processes.

In certain scenarios, simulation models need to be empowered with external environments to support advanced calculation, optimization, or evaluation. This highlights the importance of *intelligent simulation* modeling, where a DES platform is connected with a powerful programming language such as R, Python, or MATLAB. Due to the ongoing growth of machine learning techniques and optimization algorithms,

978-1-6654-3311-2/21/$31.00 ©2021 IEEE

this combination becomes even more necessary than ever. However, there is a lack of tutorials on how to efficiently combine simulation software with data science programming languages such as Python.

Yuriy and Vayenas (2008) connected AutoMod and Simul8 with an genetic algorithm engine to optimize equipment reliability assessment. Dehghanimohammadabadi and Keyser (2017) developed an intelligent simulation model by integrating MATLAB and Simio to perform multiple optimization analysis. In a similar integration, Dehghanimohammadabadi et al. (2017) created a simheuristic model for a patient scheduling optimization. Also, van der Ham (2018) introduced a new open-source and object-oriented package called Salabim. This package is developed in Python, and it includes an animation engine. Salabim has been used, for instance, to simulate a complex control system in logistics and production environments. Liu (2020) introduced Simulus, which supports event-driven and process-oriented simulation by utilizing Python packages and available modules.

Python is a popular language in the field of data science and artificial intelligence. It is widely used to solve statistical and scientific problems. Combining simulation software with Python will provide enormous advantages for experimentation, optimization, results analysis, and professional data visualization. Additionally, it is an open-source language supplemented with multiple libraries for scientific computing, machine learning, optimization, data science, and big data (Raschka and Mirjalili 2019). Therefore, this paper aims to promote the development of *intelligent* simulation modeling by combining different simulation environments with Python. Some of our main goals are:

- To provide some insights on the use of an open-source simulation package such as SimPy.
- To show how it is possible to integrate commercial simulation packages (i.e., AnyLogic, Simula8, etc.) with Python.
- To show how the combined approaches can be used in practice to develop powerful simheuristics (Guimarans et al. 2018; Chica et al. 2020; Rabe et al. 2020).

It has to be noticed that the examples provided in this paper are demo models with simplified details. Therefore, the applied search or optimization algorithms are simple in order to ensure that the tutorial goals are met. Users can benefit from the discussed examples and learn how to integrate their simulation models with Python based on their intelligent simulation needs.

## 2 SIMPY AND PYTHON

SimPy is an object-oriented and method-based open-source library. It is a powerful tool for developing DES models. Since it is built in the Python environment, interfacing SimPy models with other Python algorithms is seamless and trivial. Using SimPy, a user can create simulation models by including methods, messages, supplies, and vehicles as active components. Additionally, users can get access to other Python libraries to extend their modeling needs. Hence, SimPy is a non-commercial DES platform that can be used to simulate different models, such as production planning, hospital operations, and vehicle routing (Sanguesa et al. 2019). This section shows how to build a DES model using SimPy, and how to perform a simple execution. To install SimPy, the *pip install simpy* command has to be prompted.

This example models several electric vehicles (EV) sharing a unique charging station. This model considers 3 types of EV, each one with a different charging time (12, 24, and 36 hours), whilst the charging station provides only 2 charging points. The simulation works as follows: each vehicle starts to request a charging point, if any is available, then it gets parked and proceeds to be charged during its charging time. Otherwise, it waits until one becomes available. When an EV releases the charging point, it completes the cycle carrying out a 5 hours trip. This cycle repeats over and over for a specified time horizon of 3 days. The Figure 1 shows the implementation of this model using Python-SimPy. This code defines a class (*ElectricVehicle*) with two methods, *run()* and *charging()*. SimPy is initialized by calling *simpy.Environment* to create an object environment *env*, which represents the simulation environment. Then, *env* is passed to

```python
class ElectricVehicle:
    car_id = 1
    def __init__(self,env,charging,station,dtrip):
        self.env=env
        self.car_id = ElectricVehicle.car_id
        self.charging_duration=charging
        self.charging_station = station
        self.action=env.process(self.run())
        self.duration_trip = dtrip
        ElectricVehicle.car_id += 1
    def run(self):
        while True:
            yield self.env.process(self.charging(self.charging_duration))
            print(f'car {self.car_id} starts trip at {self.env.now}')
            yield self.env.timeout(self.duration_trip)
    def charging(self,duration):
        with self.charging_station.request() as req:
            yield req
            print(f'car {self.car_id} is going to charge at {self.env.now} duration {self.charging_duration}')
            yield self.env.timeout(duration)
            print(f'car {self.car_id} is charged at {self.env.now}')
import simpy
nb_stations = 2
duration_trip = 5
nb_cars_types = 3
charging_times = [12,24,36]
time_horizon = 3*24
env=simpy.Environment()
charging_stations = simpy.Resource(env, capacity=nb_stations)
for i in range(nb_cars_types):
    ElectricVehicle(env,charging_times[i],charging_stations,duration_trip)
env.run(until=time_horizon) #3 days
```

Figure 1: Simpy model.

the *ElectricVehicle* constructor to create an instance of a *car*. In this example, 3 cars are created. Each car starts its trip for 5 hours and stops at the charging station for recharging. There are only 2 charging stations, so one car needs to wait until another car gets charged. The waiting time defined in three different hours (*charging times = 12, 24, 36*). In addition, *simpy.Resource()* is used to limit the number of simultaneously used processes. As there is a limitation in the number of charging stations, these can be modeled as a resource. Hence, EV arrive at the station and place a request to get a full charge. After recharging, cars start their trips. Figure 2 shows the experimental results with driving and parking starting times. This model can be extended by including additional cars, charging stations, and key performance indicators, such as waiting times.

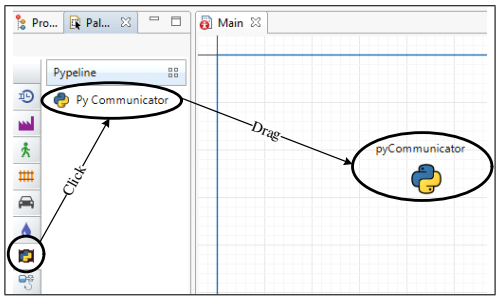
```
car 1 is going to charge at 0 duration 12
car 2 is going to charge at 0 duration 24
car 1 is charged at 12
car 1 starts trip at 12
car 3 is going to charge at 12 duration 36
car 2 is charged at 24
car 2 starts trip at 24
car 1 is going to charge at 24 duration 12
car 1 is charged at 36
car 1 starts trip at 36
car 2 is going to charge at 36 duration 24
car 3 is charged at 48
car 3 starts trip at 48
car 1 is going to charge at 48 duration 12
car 2 is charged at 60
car 1 is charged at 60
car 2 starts trip at 60
car 1 starts trip at 60
car 3 is going to charge at 60 duration 36
car 2 is going to charge at 65 duration 24
```

Figure 2: SimPy result.

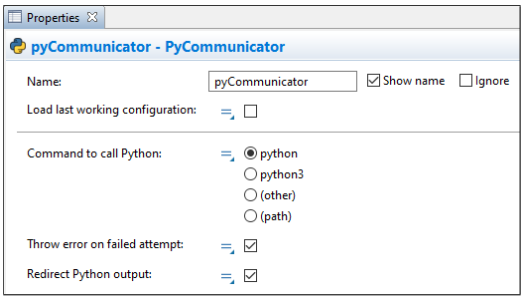
## 3 CALLING PYTHON FROM ANYLOGIC

The AnyLogic (AL) simulation platform is a modern simulation software based on the object-oriented paradigm. Borshchev (2013) introduced AL as an efficient software that can support different modeling methods, such as system dynamics, discrete-event, and agent-based. AL supports multiple types of simulation experiments. For instance, optimization and parameter variation. Besides, different application problems, such as transportation systems, supply chain management, industrial development, and business process evaluation can be modeled in AL (Merkuryeva and Bolshakovs 2010; Muravev et al. 2021). A simulation model in AL includes a set of active objects that work simultaneously and interact with each other. The active object is the main structure of the model, which enables users to develop their customized modification. For more details about features and tutorials refer to this website How to learn AnyLogic.

A key advantage of AL is its capability to integrate with Python packages. This enables the simulation model to interact with external codes written in Python. This connection is supported by a Pypeline library written in Java. Using this library, AL users can link the simulation model with any customized functions and algorithms developed in Python environment to support experimentation. Pypeline (*pypeline.jar*) can be downloaded from github.com and added to the AL palette. After installation, the Pypeline library will be added to the AL environment (Figure 3a). At this point, the AL model is ready to connect with Python by adding an object called *pyCommunicator* to the main model. As shown in (Figure 3b), Pypeline is compatible with any Python installation, and it uses the same version as it is installed on the user's machine. Therefore, to complete the pyCommunicator settings, a user needs to select an appropriate Python version and add the Python path (the directory in which the Python is installed.)




(a) Pypeline icon (pyCommunicator) on AL palette.




(b) Linking Pypeline to Python local installation path.

Figure 3: AL Pypeline library

Pypeline provides two methods of communication. The first function is *run*, which is the one-directional statement to send commands to Python. This function can be used in many instances such as import statements, variable assignments, and function calls. The second function is *runResult*, which is two-directional. This function not only sends statements to Python, but it also can receive returns based on Python calculations. This functionality allows users to change the model configuration, conduct experiments, and even perform optimization from a Python environment. Both of these functions take a string as an input to represent execution codes. *runResults* returns a custom object, *Attempt*, which includes two outputs: *(i) isSuccessful*, a Boolean indicating whether the Python command is successful or not; and *(ii) getFeedback*, which shows the model error if it exists.

To illustrate how AL calls Python, a simple simulation model is created based on the the flowchart depicted in Figure 4. This model contains a *source* that is the main agent, a *queue* (a placeholder for agent to wait for the next process), and a *delay* that calls the Pypeline file. The delay time directly depends on the Python file execution time or could vary based on the computation size and complexity. There are two text elements in the model, *calls* and *result*, which represent the *number of Python calls* and the *returned value*, correspondingly. After finishing the process (e.g., the Python computation in this case), the agent leaves the system from the *sink*.

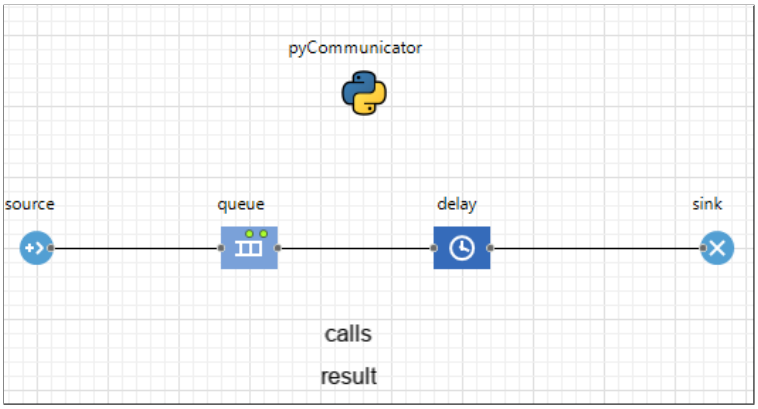



Figure 4: AL simple model.

The Python script that AL calls in this example is a continuous optimization problem to minimize the *basinfunction* function defined as follows: $f(x) = \sum_{i=1}^{n} a(x_i - h)^2 + k$, where $x \in [-0.5, 0.5]$ is the decision variable, with constant parameters $n = 2$, $a = 0.5$, $h = 2$, and $k = -5$. To solve this problem, a *randomSearch* algorithm is coded in Python (Figure 5).

```
import math, random, time
## Basin function code
def basinfunction(vector):
    a,h,k = 0.5, 2, -5
    # based on formulate we have some constant value a=0.5 ,k=-5, h=2
    sum = 0
    for item in vector:
        sum = sum + a * pow((item - h),2) + k
    return sum

def randomSearch(searchSpace, problemSize):
    #search space is min and max range of cordinate
    min = searchSpace[0] #min value in searchspace
    max =  searchSpace[1] #max value in searchspace
    inputValues = [] #list of values
    for i in range(0, problemSize):
        inputValues.append(min + (max - min) * random.random())
    return inputValues

##Solver Framework##
def Solver(maxIterations=1000,problemSize=2):
  bestSol = None
  searchSpace = [-5, 5]
  bestCost = float('inf')
  while maxIterations > 0 :
            newsol = randomSolution(searchSpace, problemSize)
        newCost = basinfunction(newsol)
        if newCost < bestCost:
                bestSol = newsol
              bestCost = newCost
        maxIterations -= 1
    return(bestCost)
```

Figure 5: Random search algorithm.

This code includes a function called *Solver*. This function runs the *randomSearch* to minimize the *basinfunction* objective function. This algorithm creates a new solution (*newsol*) by randomly selecting a value for $x$ in each iteration. Once the search is complete, the best ever found solution (*bestSol*) and its corresponding cost (*bestCost*) are returned.

To run the Python code from the AL model, the model properties need to be edited properly. This change can be done as follows:

1. From the *main* tab, select the *main-agent type*.

2. Modify the *Agent action* by changing the *on startup* settings (Figure 6):
   - Syntax 1: *pyCommunicator.run("import filename")*, which imports the Python file.
   - Syntax 2: *pyCommunicator.run("result, calls=0,0")*, which calls the Python function and receives its results.

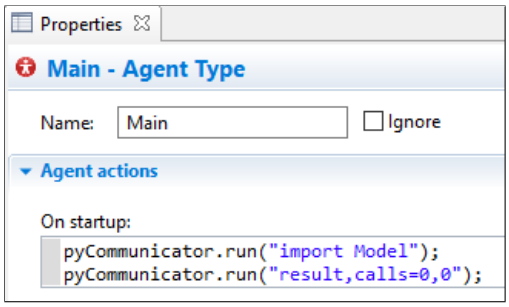



Figure 6: Call Python file.

3. From the *delay* object, modify the *on exit* event setting to enable the AL model to trigger the Python code (Figure 7). These changes are as follows:
   - Syntax 1: *pyCommunicator.run("calls += 1")* to increment the number of times the Python files are called.
   - Syntax 2: *pyCommunicator.run("results = round(Model.Solver(), 4)")* deploys the random search algorithm written in the *Solver* function. The number 4 determines how many decimal points of accuracy have to be included in the results.
   - Syntax 3: *Attempt attempt1 = pyCommunicator.runResults("calls")* to enable the AL model to trigger Python.
   - Syntax 4: *Attempt attempt2 = pyCommunicator.runResults("results")* to enable the AL model to receive Python results.

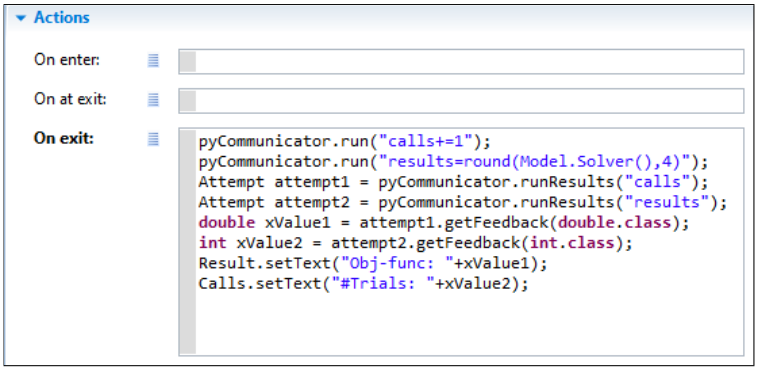



Figure 7: On exit properties of "delay".

   - Syntax 5 to 8: The rest of the codes are for matching the datatype between Python and AL, as well as for displaying the results on the AL screen (Figure 8).

## 4 CALLING ANYLOGIC FROM PYTHON

AL provides a cloud API for Python, which gives the capability of running simulation models in the cloud. Performing models in the cloud have the advantage that it may release computational and hardware requirements. To learn how to use this API, interested users can refer to AL cloud website. This section, shows how to take advantage of this API based on an existing Service System Demo model. This model contains a single *source*, a *service module*, a *checkpoint*, and one *sink*, as it is shown in Figure 9.

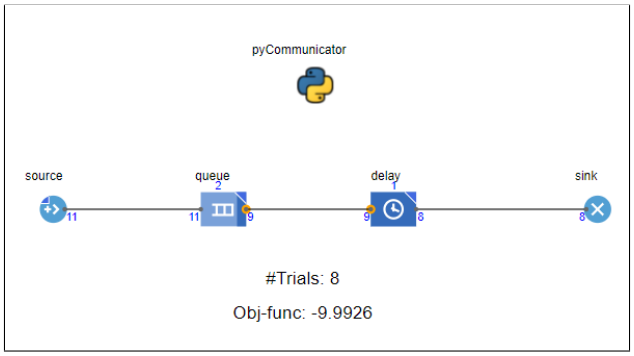



Figure 8: The structure of AL model integrated with Python.

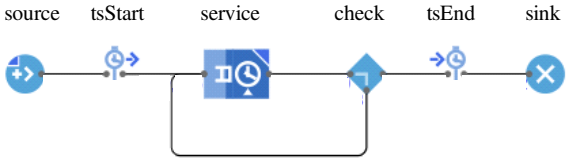



Figure 9: Service System Demo Model.

The model simulates a service process and allows a user to perform experiments by changing the model configuration (e.g., the server capacity). The model outputs are the average size of the service queue and the resource utilization. The following steps provide guidelines to develop AL models from cloud client in Python (Figure 10):

1. Install the AL cloud client library by using the *pip* package manager from the command line (line 2).
2. Import the AL cloud client library, and set the client object API key: **e05a6efa-ea5f-4adf-b090-ae0ca7d16c20** (lines 4 to 8).
3. Create a *model* object using the *client.get_latest_model_version()* function. This makes the interaction between Python and AL to be seamless for inputs, outputs, and dashboard configurations (line 11).
4. Establish the experimental settings *input* using *client.create_inputs_from_experiment()*. The *input* value can be user-defined (e.g., *Server capacity* is set to 10) (lines 14 to 17).
5. Develop the remote AL simulation model using the *client.create_simulation()* function and embedding the experimental inputs. (line 20).
6. Retrieve the simulation results by triggering the simulation experiments with *simulation.get_outputs_and_run_if_absent()* (line 23).

Figure 11 shows the experiment results of the aforementioned model with a server capacity of 10 units. These results show the mean service queue size of approximately 1 person, and the average Server utilization of 0.25.

## 5 CALL SIMUL8 FROM PYTHON

Simul8 is a simulation software that is used for simulating the discrete-entities processing at the discrete time. It is useful not only in industry, but also in academia. Simul8 visualizes the processes by using 2D animation, and it is suitable for DES. Also, it operates with 6 main parts, such as work item, entry point, storage bin, work center, work exit point, and resource (Elder 2014). This section elaborates on how to interface Simul8 with Python based on the example provided in Simul8.com. It needs to be noted that,

```
#install the AnyLogic cloud client library
pip install https://cloud.anylogic.com/files/api-8.5.0/clients/anylogiccloudclient-8.5.0-py3-none-any.whl

# Load anylogiccloudclient library
from anylogiccloudclient.client.cloud_client import CloudClient

#Create a CloudClient object, given the API key
client = CloudClient("e05a6efa-ea5f-4adf-b090-ae0ca7d16c20")

#Obtain latest model version of "Service System Demo" model
model = client.get_latest_model_version("Service System Demo")

#Create an Inputs object with the default input values
inputs = client.create_inputs_from_experiment(model, "Baseline")

#Change the "Server Capacity" parameter value
inputs.set_input("Server capacity", 10)

#Create a simulation object with the inputs
simulation = client.create_simulation(inputs)

#Obtain the simulation outputs
outputs = simulation.get_outputs_and_run_if_absent()

print("For Server Capacity = " + str(inputs.get_input("Server capacity")))
print("Mean queue size = " + str(outputs.value("Mean queue size|Mean queue size")))
print("Server utilization = " + str(outputs.value("Utilization|Server utilization")))
```

Figure 10: An example of calling AL model from Python using AL cloud client.

```
For Server Capacity = 10
Mean queue size = 0.9999193295506064
Server utilization = 0.2501690832867859
```

Figure 11: Output produced by the example presented using the AL cloud API for Python.

the 32-bit version of Python is recognizable by Simul8 as a COM application (you should use *pip install pywin32*). The following steps explain the syntax codes provided in Figure 12:

```
import win32com
from win32com import client
from win32com.client import makepy
makepy.main()
class EventHandler:
    def OnS8SimulationEndRun(self):
        n=1
        while (n <= S8.ResultsCount):
            print(S8.Results(n))
            n +=1
S8 = win32com.client.Dispatch("Simul8.S8Simulation")
S8.Open(r"C:\Program Files (x86)\SIMUL8_P_T\Examples\OptQuest\Call_Center_1.s8")
events = win32com.client.WithEvents(S8, EventHandler)
S8.RunSim(300)
S8.Quit()
```

Figure 12: Python code to run a model.

1. Import the required packages (lines 1 and 2).
2. Import *makepy*, which is a specific file for *win32com* – it is required to create COM modules out of COM-enabled applications (line 3).
3. Enter into the *Makepy* file by running the Python code (line 4), and then select the *Simul8 Library* from the list (Figure 13).
4. Create a class called *EventHandler*, which counts the number of times tha Simul8 is called, and prints the simulation results (lines 5 to 10).

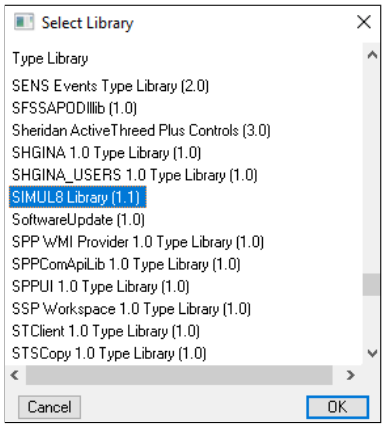
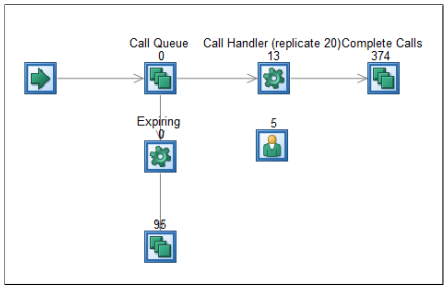



Figure 13: Simul8 library.

5. Use the dispatch method to open the Simul8 software (line 11).
6. Open the developed simulation model (Figure 14a) (line 12).
7. Connect the *EventHolder* class with Simul8 (line 13).
8. Run the simulation experiment for 300 minutes (line 14) and then terminate the the execution (line 15). Finally, Figure 14b demonstrate the result.




(a) Simul8 example model.

```
Generating...
Importing module
18.0
18.0
18.0
13.0
18.0
85.0
85.0
4.0
374.0
374.0
0.0
```

(b) Result in Python.

Figure 14: Simul8 model and result

## 6 CONCLUSIONS AND FUTURE WORK

This paper offers a tutorial on how to integrate several DES platforms with Python. Some examples are introduced to illustrate the potential applicability of combining Python with SimPy, AnyLogic, and Simul8. Each of these models require different settings and offer their own pros and cons. To draw some insights, Figure 15 depicts a comparison of the tested DES platforms, which have been evaluated according to the following metrics:

- **Easy to integrate with Python**: the API capability of each application to be integrated with Python.
- **Documentation**: the availability of documentation provided by vendors or users.
- **Remote access**: ability to be accessed remotely without opening the software application.
- **Simulation capability**: simulation modeling features and complexity.

Notice that SimPy is an effective package with complete documentation that can help users to build a simulation model. However, Simul8 and AL have more capabilities for building complex DES models.

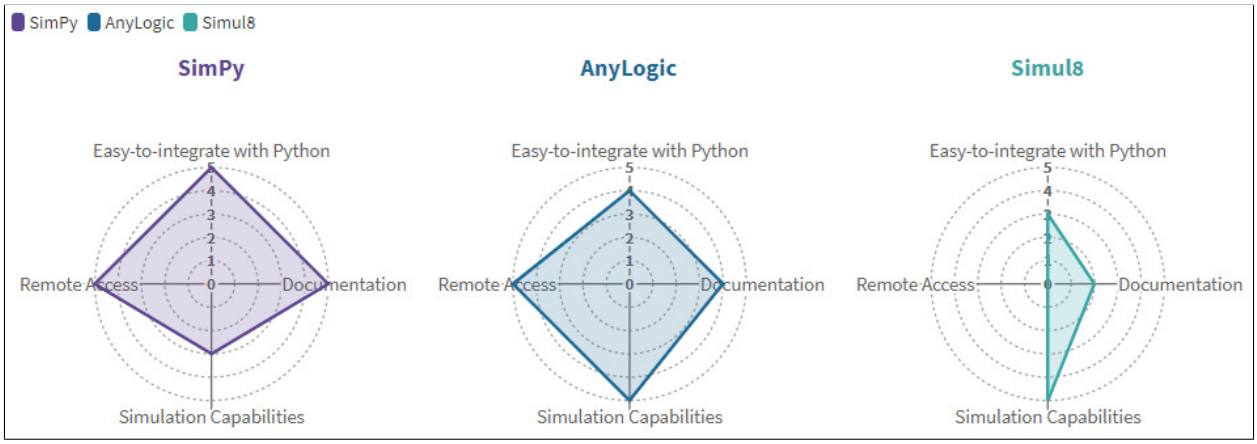



Figure 15: A comparison of different integration schemes.

Between these two simulation platforms, AL is easier to integrate with Python when designing simulation-optimization approaches.

As many real-life systems work under uncertainty conditions and might need to be optimized, simheuristic algorithms have become an excellent option. They combine scalable metaheuristic components with simulation. Therefore this work helps users to develop rich simheuristic algorithms that combine sophisticated metaheuristic frameworks with complex DES models. Of course, these rich simheuristics might contribute to generate insights in many real-life stochastic optimization problems, ranging from transportation and logistics to manufacturing and production or smart cities.

In future work, we plan to extend this tutorial to other simulation platforms. However, this has not been possible so far due to the lack of documentation and Python-specific APIs. Since Python has become the standard programming language in data science, we encourage the firms offering state-of-the-art simulation software included in (Simio, Arena, WITNESS, FlexSim, ProModel, ExtendSim, PlantSimulation, SimProcess, AutoMod, Micro Saint, QUEST, Enterprise Dynamics, Process Model, etc.) to increase their connectivity options with the Python programming language, which is being employed by a vast number of data scientists worldwide.

## ACKNOWLEDGMENTS

This work has been partially supported by the Spanish Ministry of Science (PID2019-111100RB-C21/AEI/10.13039/501100011033, RED2018-102642-T), and the 'DigiLab4U' project (https://digilab4u.com/).

## REFERENCES

Borshchev, A. 2013. "Multi-Method Modeling". In *Proceedings of the 2013 Winter Simulation Conference*, edited by R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, 4089–4100. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Chica, M., A. A. Juan, C. Bayliss, O. Cordón, and W. D. Kelton. 2020. "Why Simheuristics? Benefits, Limitations, and Best Practices when Combining Metaheuristics with Simulation". *SORT-Statistics and Operations Research Transactions* 44:311–334.

Dagkakis, G., and C. Heavey. 2016. "A Review of Open Source Discrete Event Simulation Software for Operations Research". *Journal of Simulation* 10(3):193–206.

Dehghanimohammadabadi, M., and T. K. Keyser. 2017. "Intelligent Simulation: Integration of SIMIO and MATLAB to deploy Decision Support Systems to Simulation Environment". *Simulation Modelling Practice and Theory* 71:45–60.

Dehghanimohammadabadi, M., M. Rezaeiahari, and T. K. Keyser. 2017. "Simheuristic of Patient Scheduling using a Table-Experiment Approach: Simio and Matlab Integration Application". In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, 2929–2939. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Dias, L. M., A. A. Vieira, G. A. Pereira, and J. A. Oliveira. 2016. "Discrete Simulation Software Ranking — A Top List of the Worldwide most Popular and Used Tools". In *Proceedings of the 2016 Winter Simulation Conference*, edited by T. M. K. Roeder, P. I. Frazier, R. Szechtman, E. Zhou, T. Huschka, and S. E. Chick, 1060–1071. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Elder, M. 2014. *DES View on Simulation Modelling: SIMUL8*, 199–214. Chichester, UK: John Wiley & Sons, Ltd.

Guimarães, A. M. C., J. E. Leal, and P. Mendes. 2018. "Discrete-Event Simulation Software Selection for Manufacturing based on the Maturity Model". *Computers in Industry* 103:14–27.

Guimarans, D., O. Dominguez, J. Panadero, and A. A. Juan. 2018. "A Simheuristic Approach for the Two-Dimensional Vehicle Routing Problem with Stochastic Travel Times". *Simulation Modelling Practice and Theory* 89:1–14.

Liu, J. 2020. "Simulus: Easy Breezy Simulation in Python". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 2329–2340. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Merkuryeva, G., and V. Bolshakovs. 2010. "Vehicle Schedule Simulation with AnyLogic". In *Proceedings of the 2010 International Conference on Computer Modelling and Simulation*, edited by D. Al-Dabass, A. Orsoni, R. Cant, and A. Abraham, 169–174. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Muravev, D., H. Hu, A. Rakhmangulov, and P. Mishkurov. 2021. "Multi-Agent Optimization of the Intermodal Terminal main Parameters by using AnyLogic Simulation Platform: Case Study on the Ningbo-Zhoushan Port". *International Journal of Information Management* 57:102133.

Rabe, M., M. Deininger, and A. A. Juan. 2020. "Speeding Up Computational Times in Simheuristics Combining Genetic Algorithms with Discrete-Event Simulation". *Simulation Modelling Practice and Theory* 103:102089.

Raschka, S., and V. Mirjalili. 2019. *Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow 2*. Birmingham, UK: Packt Publishing Ltd.

Sanguesa, J. A., S. Salvatella, F. J. Martinez, J. M. Marquez-Barja, and M. P. Ricardo. 2019. "Enhancing the NS-3 Simulator by introducing Electric Vehicles Features". In *Proceedings of the 2019 International Conference on Computer Communication and Networks*, edited by J. Marquez-Barja, 1–7. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Schriber, T. J., D. T. Brunner, and J. S. Smith. 2012. "How Discrete-Event Simulation Software Works and Why it Matters". In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, 1–15. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

van der Ham, R. 2018. "Salabim: Open Source Discrete Event Simulation and Animation in Python". In *Proceedings of the 2018 Winter Simulation Conference*, edited by M. Rabe, A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, 4186–4187. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Yuriy, G., and N. Vayenas. 2008. "Discrete-Event Simulation of Mine Equipment Systems Combined with a Reliability Assessment Model based on Genetic Algorithms". *International Journal of Mining, Reclamation and Environment* 22(1):70–83.

## AUTHOR BIOGRAPHIES

**MOHAMMAD PEYMAN** is a PhD candidate and researcher in the ICSO group at the IN3 – Universitat Oberta de Catalunya. He holds a BSc in Mechanical Engineering from Azad University of Iran and a MSc in Computer Science from the Universitat Autonoma de Barcelona, Spain. His main research interests are data science, machine learning, and simulation-optimization algorithms. His email address is mpeyman@uoc.edu.

**MOHAMMAD DEHGHANIMOHAMMADABADI** is Assistant Teaching Professor of Mechanical and Industrial engineering, Northeastern University, Boston, MA, USA. His research is mainly focused on developing and generalizing simulation and optimization frameworks in difference disciplines. This article is part of his initiatives to develop a framework to integrate Discrete Event Simulation with Reinforcement Learning. His e-mail address is m.dehghani@northeastern.edu.

**PEDRO J. COPADO-MENDEZ** is a post-doctoral researcher in the ICSO group at the IN3 – UOC. He completed his PhD at the University Rovira i Virgili (Spain). His main research interests include metaheuristics, hybrid heuristics and their applications. His email address is pcopadom@uoc.edu.

**JAVIER PANADERO** is an Assistant Professor of Simulation and High Performance Computing in the Computer Science, Multimedia and Telecommunication Department at the Universitat Oberta de Catalunya (Barcelona, Spain). He is also a Lecturer at the Euncet Bussines School, and a member of the ICSO@IN3 research group. He holds a Ph.D. and a M.S. in Computer Science. His major research areas are: high performance computing, modeling and analysis of parallel applications, and simheuristics. He has co-authored more than 50 articles published in journals and conference proceedings. His website

address is http://www.javierpanadero.com and his email address is jpanaderom@uoc.edu.

**ANGEL A. JUAN** is a Full Professor of Operations Research & Industrial Engineering in the Computer Science Department at the Universitat Oberta de Catalunya (Barcelona, Spain). He is also the Director of the ICSO research group at the Internet Interdisciplinary Institute and Senior Lecturer at the Euncet Business School. Dr. Juan holds a Ph.D. in Industrial Engineering and an M.Sc. in Mathematics. He completed a predoctoral internship at Harvard University and postdoctoral internships at Massachusetts Institute of Technology and Georgia Institute of Technology. His main research interests include applications of simheuristics and learnheuristics in computational logistics and transportation. He has published over 115 articles in JCR-indexed journals and over 265 papers indexed in Scopus. His website address is http://ajuanp.wordpress.com and his email address is ajuanp@uoc.edu.