Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

A TUTORIAL ON COMBINING FLEXSIM WITH PYTHON FOR DEVELOPING DISCRETE-EVENT SIMHEURISTICS

Jonas F. Leon
Paolo Marone

Spindox España S.L.
Universitat Oberta de Catalunya
305 Calle Muntaner
Barcelona, 08021, SPAIN

Mohammad Peyman
Yuda Li
Laura Calvet

Computer Science Dept.
Universitat Oberta de Catalunya
156 Rambla Poblenou
Barcelona, 08018, SPAIN

Mohammad Dehghanimohammadabadi

Dept. of Mechanical and Industrial Engineering
Northeastern University
360 Huntington Avenue
Boston, MA, 02115, USA

Angel A. Juan

Dept. of Applied Statistics & Operations Research
Universitat Politècnica de València
Plaza Ferrandiz-Carbonell
Alcoy, 03801, SPAIN

ABSTRACT

Connecting commercial discrete-event simulation packages to external software or programming languages is essential to advance simulation modeling capabilities. For instance, this connectivity allows users to link the simulation environment to metaheuristic optimization algorithms or to machine learning methods. However, implementing these connections is not a trivial task, and may require API access and proper settings configurations. This tutorial provides a step-by-step guideline to connect the FlexSim commercial simulator with the popular Python programming language via sockets. Using this type of connection, a simheuristic algorithm coded in Python aims at optimizing the product allocation in a warehouse, which has been previously modeled in the aforementioned simulator. In addition, potential future applications of this software combination will be discussed to provide insights into future developments such as more advanced simheuristics or combinations of simulation with learnheuristics.

1 INTRODUCTION

Simulation represents a powerful tool to model and study complex and dynamic systems with non-linear interactions. In particular, discrete-event simulation (DES) constitutes a common simulation approach in many industries. There are several commercial simulators (e.g., Simio, AnyLogic, FlexSim, etc.) and also non-commercial ones (SimPy, Salabim, etc.) that enable DES modeling. These simulators can significantly enhance their potential when connected to other tools capable of performing advanced optimization or machine learning studies (Peyman et al. 2021). In addition, it is usually desirable to integrate simulation with a dedicated computational tool that leverages some of the computational work (Dehghanimohammadabadi and Keyser 2017). Therefore, connecting DES platforms with external programming languages like Python, Julia, R, or MATLAB enriches the simulation modeling by providing more mathematical and algorithmic capabilities. Unfortunately, a publicly available API for the connection between a DES platform and some of the aforementioned programming languages is not always available. Python is an interpreted high-level general-purpose open-source programming language (van Rossum and Drake 2009). It has a diverse, wide, and international community of programmers. Calling simulation software from Python may represent limitless advantages for data processing and manipulation, experimentation, optimization, results analysis, and visualization. This is due to the multiple libraries for machine learning, optimization, data science, and big data already available for this language, as well as to the possibility of developing powerful metaheuristic algorithms over it. The DES platform selected for this work is FlexSim, which is an object-oriented software environment used to develop, model, simulate, visualize, and monitor dynamic-flow process activities and systems. It supports DES, as well as agent-based simulation and continuous simulation, enables 3D simulation modeling and analysis, and has diverse potential applications in many fields, among others: manufacturing, material handling, healthcare, and warehousing.

In this context, the proposed tutorial presents a two-fold goal: *(i)* to explain the integration between FlexSim and Python; and *(ii)* to describe an illustrative example by implementing a simheuristic algorithm designed to support decisions in warehouse management. Simheuristics represent a hybrid methodology that extends (meta)-heuristics through simulation to solve stochastic combinatorial optimization problems or COPs (Juan et al. 2018). Thus, this methodology enables us to deal with real-life uncertainty in a natural way by integrating simulation (in any of its variants) into a metaheuristic-driven framework. The COP studied in this work is the storage location assignment problem (SLAP), which is an *NP-hard* problem of great importance in supply chain management. As described by Reyes et al. (2019), the SLAP is "an operational decision associated with the accommodation and picking process, influencing batch definition, classification, routing, and order sequencing". Typically, the objective functions are related to warehouse space utilization and the time required for order preparation and picking operations. Diverse restrictions such as available storage capacity, order-picking resource capacities, and dispatching policies may be considered as well. The rest of the paper is structured as follows: Section 2 presents a short literature review on some of the keys topics analyzed in this paper. Sections 3 and 4 explain the steps required to connect FlexSim and Python. Section 5 shows an illustrative example based on the SLAP problem. Afterwards, Section 6 describes a few potential applications of discrete-event simheuristics combining DES software with a programming language in the logistics field. Finally, Section 7 highlights the main conclusions of this work.

2 LITERATURE REVIEW

This section reviews some related works in the areas of: *(i)* the main software that will be employed in this study; and *(ii)* the incorporation of 'intelligence' inside simulation models.

2.1 Software (DES): FlexSim

FlexSim is a state-of-the-art simulation modeling software used to analyze, visualize, and improve real-world processes. It has an easy-to-use interface and a large library of standard objects to help users quickly create rich discrete-event simulation models. The online manual, available at www.flexsim.com, is a clear and complete source of detailed information. FlexSim provides a trial version of the software, intended to new users for evaluation purposes, along with academic and professional versions. There are various tools available in FlexSim to deploy optimization methods for the simulation models: (*i*) The build-in optimization engine OptQuest, that can be used for running multi-objective optimization experiments; (*ii*) External DLLs (dynamic-link libraries) to combine FlexSim with C and C++ programming language; (*iii*) Flexiscript (internal script language) to exchange data with third-party software (like Python, as is the case of the present study) and advanced customization.

In recent years, many researchers have employed FlexSim for different types of simulations. Nordgren (2002) simulated manufacturing, warehousing, material handling processes, semiconductor manufacturing, marine container terminal processes, and shared access storage network (SANS) using FlexSim. Zhu et al.

(2014) simulated the operation of a cold-chain logistics distribution center for fruits and vegetables. They examined the simulation results to determine the system's bottleneck and idle resources. With some system adjustments, they were able to improve the turnover rate of cold chain goods as well as the utilization rates of equipment and workers. Wang and Chen (2016) used the time Petri net model and FlexSim to simulate an automobile assembly workshop and production logistics system. They analyzed the simulation results and proposed optimization methods to improve processes and efficiency while also increasing the energy-producing effects of the entire assembly workshop.

2.2 Intelligent Perspective of Simulation

Artificial Intelligence (AI) is concerned with comprehending and learning the phenomena of human intelligence, as well as developing computer systems capable of imitating human behavioral patterns and creating knowledge relevant to problem-solving (Mahroof 2019). The combination of AI and simulation has applications in a variety of industries. For instance, von Rueden et al. (2020) demonstrated the versatile, potential combination possibilities of the two modeling approaches, machine learning and simulation, in order to inspire and foster future developments of hybrid systems. Zhou et al. (2021) presented new cyber-physical integration in smart factories for online scheduling of low-volume-high-mix orders. In addition, based on Reinforcement Learning (RL), new reward functions were designed to improve the decision-making abilities of multiple AI schedulers. Furthermore, Zielinski et al. (2021) presented a tool for converting supervisory control theory controllers into RL simulation environments for intelligent processing. In addition, an RL-based approach was proposed that recognizes and treats the context in which the selected set of stochastic events occur, with the goal of finding appropriate decision making as a supplement to the deterministic outcomes of supervisory control theory. Also, Dehghanimohammadabadi and Keyser (2017) demonstrated how to connect Simio with MATLAB to enhance the optimization power of the DES environment.

3 CONNECTION BETWEEN FLEXSIM AND PYTHON

This section explains the technical details associated with the creation of a link between FlexSim and Python. In Figure 1 the general scheme of the connection and intercommunication process to be developed in this paper is presented.

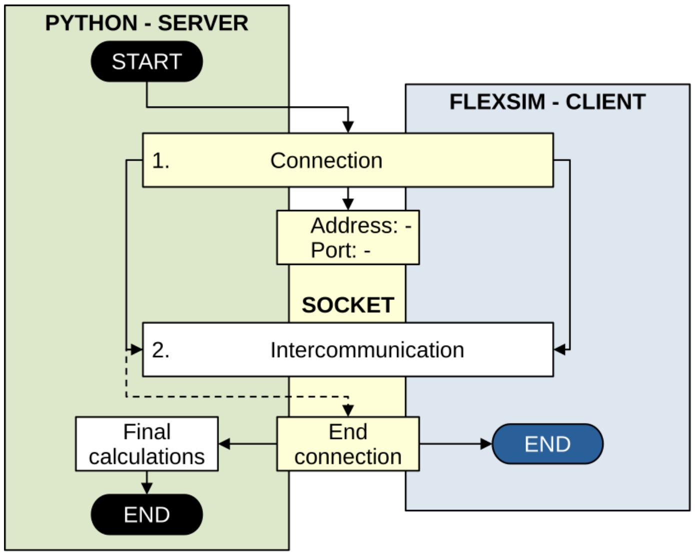

Figure 1: High level flow diagram of the connection and intercommunication process.

3.1 General Considerations

There are multiple ways of connecting FlexSim and Python (and in general, any two pieces of software). This connection might depend upon the type of interaction intended or the function to be performed by each part. First of all, it is important to state that FlexSim is a commercial application (or program), and that Python is a programming language. Therefore, what is shown in this tutorial, is how to connect and communicate from a Python script to a single instance of FlexSim. A practical aspect to be defined beforehand is the roles for each part of the connection, namely, what piece of software will start the connection, what would be the protocol of communication, and how will the communication be terminated.

3.2 Communication

In this tutorial, it is assumed that the Python script behaves as the “server”, and FlexSim as the “client”. In reality, both are communicating on par with each other, but it is Python the one that starts the communication process, the one that listens for request/messages from FlexSim and provides an answer, and the one that eventually closes the simulation and the communication channel. Also, in theory, multiple instances of FlexSim could be communicating with the Python “server”. The communication is established through sockets. Sockets are a robust, standard and potentially secure way for applications to communicate with each other, allowing the extension of the simulation software capabilities (Rodrigues et al. 2005). The process starts with Python creating a socket with the specified address and port. By default, the “localhost” address and any port above port “1024” (e.g., “5005”) can be used for creating a local connection. Afterwards, Python starts an instance of FlexSim, which in turn tries to establish a connection through the same socket (i.e., the same address and port). The code details can be seen in Section 3.3 for Python and in Section 3.4 for FlexSim.

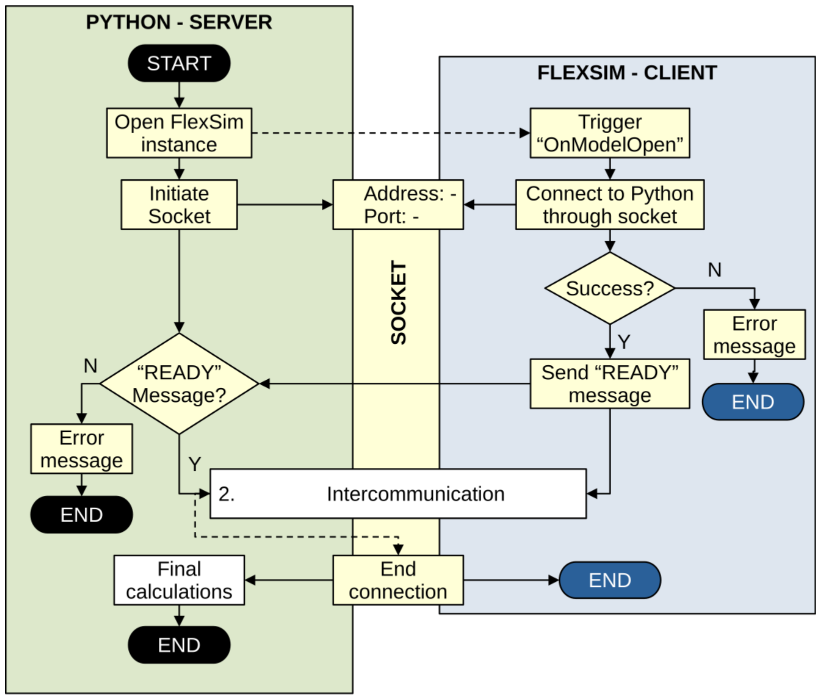

Figure 2: Detail flow diagram of the connection part of the process.

3.3 Server Side (Python)

The information provided in this section is based upon various examples available at https://www.flexsim.com. All the code shown herein is for illustrative purposes only, and has been simplified to a "minimal working example" size. The creation of the socket connection, which is the start of the connection process shown in Figure 2, is done by instantiating a Python *class*, which contains all the necessary functions for connection and communication. The simplified code, for declaring the Python class and the functions within it, is shown in Figure 3. The libraries for socket connection and for running sub-processes are imported at the beginning of the code. The functions defined within the class are listed below:

- *_socket_init*: opens the socket at the specified address and port, using the Python "socket" library.
- *_socket_end*: closes the socket at the specified address and port, using the Python "socket" library.
- *_launch_flexsim*: starts a FlexSim instance using the Python "subprocess" library, and initiates the socket by calling the *_socket_init* function.
- *_close_flexsim*: kills the FlexSim sub-process and shuts down the socket using the *_socket_end* function.
- *_socket_send*: sends a message to the client (FlexSim).
- *_socket_recv*: receives a message from the client (FlexSim).

```python
import subprocess
import socket
class FlexSimConnection():
    def __init__(self, flexsimPath, modelPath, address='localhost', port=5005):
        self.flexsimPath = flexsimPath
        self.modelPath = modelPath
        self.address = address
        self.port = port
    def _launch_flexsim(self):
        args = [self.flexsimPath, self.modelPath] # option to add aditional arguments
        self.flexsimProcess = subprocess.Popen(args)
        self._socket_init(self.address, self.port)
    def _close_flexsim(self):
        self.flexsimProcess.kill()
        self._socket_end(self.address, self.port)
    def _socket_init(self, host, port):
        self.serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.serversocket.bind((host, port))
        self.serversocket.listen();
        (self.clientsocket, self.socketaddress) = self.serversocket.accept()
        message = self._socket_recv()
        if message != b"READY":
            raise RuntimeError("Did not receive READY! message")
    def _socket_end(self, host, port):
        self.serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.serversocket.bind((host, port))
        self.serversocket.close()
    def _socket_send(self, msg):
        totalsent = 0
        while totalsent < len(msg):
            sent = self.clientsocket.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError("Socket connection broken")
            totalsent = totalsent + sent
    def _socket_recv(self):
        chunks = []
        while 1:
            chunk = self.clientsocket.recv(2048)
            if chunk == b'':
                raise RuntimeError("Socket connection broken")
            if chunk[-1] == ord('!'):
                chunks.append(chunk[:-1])
                break;
            else:
                chunks.append(chunk)
        return b''.join(chunks)
```

Figure 3: [Python Code] Class for FlexSim environment connection.

3.4 Client Side (FlexSim)

The information provided within this section and the settings required to set up the client side of the communication are well documented in the aforementioned FlexSim website. The code is written in *FlexScript*, which is the programming language that FlexSim uses internally. Alternatively, FlexSim allows employing C++ dedicated *DLLs* for the same purpose. The connection has to start right after the FlexSim instance is created, if the program is to be run automatically and without external intervention. For that, a (global) event trigger called "OnModelOpen" can be used within FlexSim. This trigger, which fires when the FlexSim instance is opened, executes the necessary code for the connection. A simplified example of this connection code that is to be written inside the "OnModelOpen" FlexSim trigger is provided in Figure 4.

```
connecttoserver("127.0.0.1",5005);
clientsend(getnodenum(node("/Tools/client", model())), "READY!");
// Request global parameters from server (e.g. simulation end time) or import tables
applicationcommand("reset");
applicationcommand("run");
```

Figure 4: [FlexScript Code] "OnModelOpen" trigger code.

The "connecttoserver" function, which appears in the first line of the "OnModelOpen" trigger code, is in charge of establishing the connection through the socket. Obviously, the address and port must be the same as used on the Python server side. The rest of the lines are mainly for confirming the connection by sending a "READY" message (alternative "shake-hand" protocols can be defined). Then, additional messages can be exchanged in order to obtain global simulation parameters. Finally, the simulation is reset and started. The code of that "connecttoserver" function is provided in Figure 5, and can be found in the FlexSim website.

```
treenode connection = assertsubnode(node("/Tools",model()), "client", DATATYPE_NUMBER);
string hostname = parstr(1);
int portnum = parval(2);
int client = 0;
// Initialize the socket before trying to create a connection. socketinit() return TRUE
// if the initialization is successful and reutrns FALSE if it is not successful.
if(socketinit()) {
    client = clientcreate(); // Start the process to use Windows sockets as a client.
}
// clientcreate() returns TRUE if the process could start properly and FALSE if it couldn't.
if(client) {
    // Make the connection to this machine using given hostname and port number.
    if(clientconnect(client,hostname,portnum)) {
        // Set the client node created earlier with the connectin value.
        setnodenum(connection,client);
        return 1; // Return a 1 to indicate the connection was made.
    }
    else {
        msg("Client Error", "Failed to connect to server.");
        if(!clientclose(client))
        msg("Socket Error", "Failed to close client socket.");
        return 0; // Return a 0 to indicate an issue.
    }
}
else // The process for socket connection could not start.
    msg("Socket Error", "Failed to create client socket.");
return client; // Return value in client variable (0) to indicate an error.
```

Figure 5: [FlexScript Code] "connecttoserver" function.

4 INTELLIGENT DISCRETE-EVENT SIMULATIONS

In this section, we go one step further in terms of how to connect Python with FlexSim via a synchronous link.

4.1 Synchronous Communication

The connection method explained in Section 3 could already be used to run a single instance of a FlexSim simulation with some initial parameters. However, this only allows for a somewhat limited algorithm to be deployed. In this section the intercommunication process will be explained further to enrich the capabilities of the proposed method. In terms of the type of interaction between software, it is common in the simulation-optimization field that the communication process is in certain sense “asynchronous”. This means that the optimization is performed in order to find a good candidate solution and after that, the simulation is run in order to evaluate or confirm the goodness of the proposed solution. Alternatively, the simulation could be run in order to obtain certain stochastic parameters that are then input into the analytical optimization model. In any case, the communication is a one-off event, which happens before or after the simulation run (Figueira and Almada-Lobo 2014). For this type of connection, FlexSim already provides some options, like passing of initial parameters to a predefined simulation, or outputting simulation values to an external file. In contrast, in the intended application shown in this paper, the interaction will be continuous or “synchronous”, meaning that the simulation software and the Python script are exchanging messages (maybe thousands) as the DES evolves, with potential re-optimizations on every step. The intercommunication process is detailed in Figure 6.

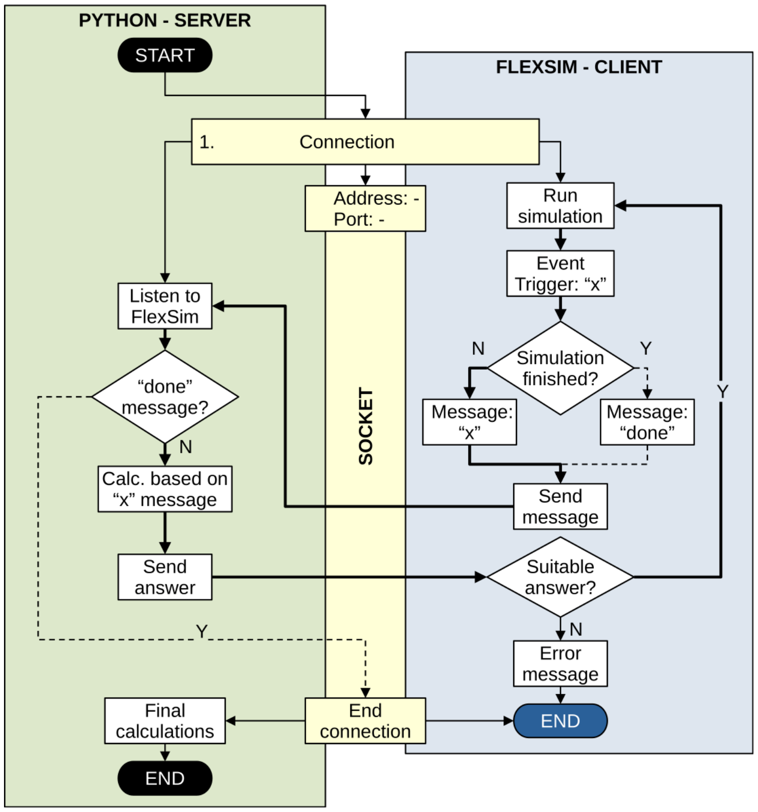

Figure 6: Detail flow diagram of the intercommunication part of the Simheuristic process.

4.2 Implementation

After the connection is successfully established, the FlexSim simulation starts and the Python script remains listening. Like within any DES software, as the FlexSim simulation proceeds different events will occur (e.g., in a warehouse context, a new order is received, a worker finishes her task, etc.), thus triggering other events or actions. The key idea of this type of “continuous” connection is that the action to be executed by FlexSim is to send a message to the Python script through the socket, and wait for the answer. In this regard, FlexSim allows for a “blocking” or “non blocking” behavior. This means that the FlexSim program execution could be frozen until something is received from the Python server. Since there could be potentially multiple messages sent during a small fraction of the simulation time, the blocking behavior option is to be used in this type of communication, in order to avoid misunderstanding between both software due to the lag between message and response. On the Python side, each message received must be classified. This means that all possible events, that are defined in FlexSim as requiring input from Python, have to be mapped to an answer in the Python script. These Python answers are normally a calculation based on the status of the simulation. For that reason, it is important that the status is either contained in the message passed through the socket or through reading directly the information from an external table or a database. The answer to be sent by the Python script is always a string, and the “meaning” of that string is to be interpreted by FlexSim. This is what makes this communication protocol so versatile. In a sense, it could be understood as the basis for well researched intelligence enhancers for simulation, like simheuristics, reinforcement learning, or learnheuristics (Calvet et al. 2017). The Python *main* code, i.e., the one that controls the “runtime” execution (as opposed to the communication class definition), is shown in Figure 7. Once the *FlexSimConnection* class is instantiated (as *FS* in the code example), it can be used to open FlexSim and to send or receive messages. There can be some initial communication in order to establish some simulation parameters (e.g., for how long the simulation will run, or the number or resources to be employed). This parameter setting needs to happen before the simulation starts running in FlexSim. The most important piece of code is the *while* loop, which keeps the Python server “listening” to the messages coming from FlexSim. Once a message is received, it is classified and replied using the *if* and *elif* logic.

```python
if __name__ == "__main__":
    FS = FlexSimConnection(
        flexsimPath = "C:/Program Files/FlexSim 2022/program/flexsim.exe",
        modelPath = "C:/Users/.../simulation.fsm")
    FS._launch_flexsim()
    # Simulation initial set-up (optional):
    END_TIME = 1000 # example: time to end the simulation
    FS._socket_send(str(END_TIME).encode())
    # Running simulation:
    done = False
    while not done:
        try:
            # Receive message
            msg_recv = FS._socket_recv().decode('utf-8')
            if msg_recv == "x":
                ANSWER = calc_answer_x()
                FS._socket_send(str(ANSWER).encode())
            # Other message answer calculations in here...
            elif msg_recv == "done":
                done = True
        except:
            done = True
            FS._close_flexsim()
    FS._close_flexsim()
```

Figure 7: [Python Code] Main section of the Python script.

At specific events (e.g., every time that a product arrives to the warehouse), which are programmed beforehand in FlexSim, some information might be needed from the Python server script (e.g., where to place the new item optimally in the warehouse). For that, a message has to be sent, and an answer must be awaited. The FlexScript code shown in Figure 8 allows this event-driven intercommunication. The code can be adapted and used at almost every point where a decision have to be made, overriding the default FlexSim decision making. This is especially powerful when combined with the FlexSim *process flow tool* for defining the simulation logic.

```
clientsend(getnodenum(node("/Tools/client", model())), "x!");
string answer = clientreceive(getnodenum(node("/Tools/client",model())), NULL, 1024, 0);
// do something with the answer like assign it to a label
token.label = answer;
```

Figure 8: [FlexScript Code] How to send and receive messages through the socket.

5 A SIMHEURISTIC ILLUSTRATIVE EXAMPLE

In this section, FlexSim and Python are combined to implement a simheuristic example related to warehousing logistics.

5.1 Problem Description

Warehouses are a crucial component in supply chain management. The improvement of efficiency in daily warehouse operations can provide significant benefits for companies. Typical activities in a warehouse include reception, storage, order-picking, and dispatch. Storage is defined as a buffer of accumulated products to guarantee the quantities demanded in the shortest possible time. Product storage and retrieval are considered as the most crucial and resource-intensive activities in warehouses (van den Berg and Zijm 1999). These processes imply product allocation decisions that affect the general performance of the storage systems. The storage location assignment problem is an operational decision problem that concerns the allocation of products into a storage space and the optimization of the storage space utilization. There are many solution approaches available in the literature, being one of the most common approaches based on policies and rules. The most representative policies include random storage, dedicated storage, and class-based storage. For a more comprehensive review of this problem, readers can refer to Reyes et al. (2019). The SLAP that this paper tries to solve aims to locate, in a certain warehouse, N types of products in M spaces, where $M > N$ (i.e., there will be free spaces after all items are located). The goal of this problem consists in minimizing the traveled distance of all warehouse workers (both inbound and outbound). This minimization of the traveled distance increases the *throughput*, defined as the number of items dealt with per unit time, which is a measure of the efficiency of the warehouse. The size and shape of the warehouse has been chosen based on some toy examples from the literature (Xie et al. 2014). Figure 9 shows the scheme of the warehouse instance evaluated in this work. At this stage, the exact size of the warehouse is somehow irrelevant, since the main objective of the tutorial is to prove the advantage of implementing a simheuristic method that helps solving the SLAP.

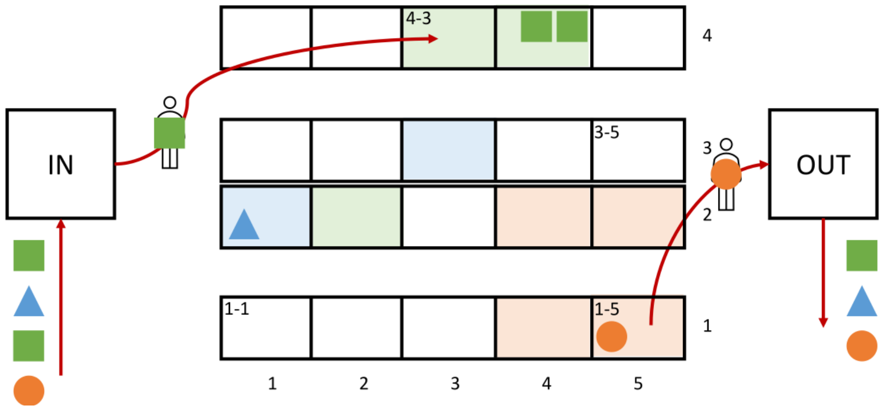

Figure 9: Problem description scheme.

As in real life, some types of product have higher arrival and pick-up frequency than others. Therefore, the products and the orders arrive at a certain rate (randomly), and according to a given probability distribution. As the products arrive to the inbound area, these are placed in their corresponding slot in the warehouse. Upon order arrival (which can be assumed to be a request for a certain type of product), a product needs to be retrieved from the warehouse shelves. The problem, therefore, consists in assigning specific slots to the arriving products, such in a way that the overall efficiency of the process is optimized.

The two main policies in the literature that are proposed for tackling the SLAP are the *random assignment* and the dedicated or *fixed assignment* strategy. The former implies that the location for each type of product is decided randomly. The latter has a pre-assigned fixed location for each type of product. This pre-assignment is usually done based on the relative demand frequency for each product, trying to minimize the distance traveled by the workers. Each strategy has its own applicability and advantages depending on the warehouse configuration. For the warehouse instance investigated in this example, since the type of products N are less than the available slots M, the fixed location is expected to provide significantly better results. This is because the more demanded type of products can be placed closer to the exit reducing the workers' average traveled distance. As can be seen in Section 5.4, our simheuristic method was able to validate this expected behavior. In contrast with solving a "static" instance of the warehouse, different "dynamic" simulation-based policies can be tested using the intercommunication method presented in Section 3. The simulation part of the method represents a natural way for treating stochastic demand constraints, as well as allowing for a more realistic evaluation of the objective function (which takes into account the warehouse geometry, interaction between workers, sequence of operations, etc.). After the method is validated with the comparison between random and fixed policy, an *improved* storage location assignment policy was designed (see Section 5.2). This policy will try to beat the previous policies by incorporating information about the incoming warehouse orders "dynamically", i.e., during the simulation time.

5.2 The Proposed Algorithm

The algorithm proposed to further improve the fixed allocation strategy was not an optimization based on the current status of the warehouse, but rather a heuristic action based upon the future orders to be fulfilled. In a certain sense, the list of orders that need to be prepared in the warehouse is indeed part of the current status of the warehouse, but neither the random or fixed strategies make use of this available information. For that reason, it is expected that incorporating this additional information will improve the warehouse efficiency. The improved heuristic for placing a new item in the warehouse is shown in Algorithm 1.

Algorithm 1 Improved SLAP

```
1: $location \leftarrow \text{fixed}(status, item)$
2: if $item \in orders$ then
3:     loop: $slot$
4:     if $\text{out\_distance}(slot) < \text{out\_distance}(location)$ then
5:         if $\text{has\_space}(slot) = true$ then
6:             $location \leftarrow slot$
7: end
```

The logic followed can be explained as a sequence of actions (heuristic): *(i)* use the fixed location assignment as baseline; *(ii)* check if the next few orders (5 - 10) contain the item to place; *(iii)* check if there is available space closer to the exit than its default assigned place; and *(iv)* place the item in the best location. In Figure 10, the implementation of the algorithm in Python is provided. It is important to note that some simulation actions will be performed by FlexSim using its own pre-programmed logic. This means that the proposed simheuristic controls only part of the simulation logic, and other parts are controlled by FlexSim (e.g., the first-in-first-out logic in some queues). For that reason, it can be expected that further improvements on the efficiency can be obtained if more actions/operations are mediated by the heuristic.

```python
def simheuristic(list_prod, dict_loc, wh_status, list_orders, prod, f):
    # Get the address for product using fixed strategy:
    location = fixed(wh_status, list_prod, dict_loc, prod)
    # Check if product is in first "f" places of the orders list:
    if len(list_orders) >= f:
        list_orders = list_orders[0:f]
    if (prod in list_orders):
        # Loop through locations:
        for slot in list(dict_loc.keys()):
            # Compare output distances using dictionary of locations:
            if dict_loc[slot] < dict_loc[location]:
                # Check if has_space is True:
                if has_space(wh_status,slot):
                    location = slot
    return location
```

Figure 10: [Python Code] ”Algorithm 1” implementation in Python.

5.3 Intercommunication Sequence

The intercommunication sequence detailed in this section is based upon the more generic intercommunication process explained in Section 3 and depicted in Figure 2. Here, the more specific intercommunication protocol between Python and FlexSim will be explained. In particular, the communication-trigger events, its corresponding message and the calculated answer are listed below:

- *A new item arrives*: the message *“product”* is sent to Python. The type of product is calculated based on the probability associated with each type. FlexSim receives the item identification (ID) and the type of product to which the item belongs.
- *A new order arrives*: the message *“order”* is sent to Python. The type of product is calculated based on the probability associated with each type. FlexSim receives the type of product required in order to fulfill the order.
- *An item needs to be placed*: the message *“slot”* is sent to Python, followed by the associated type of product. The calculation is performed using the selected policy in the Python script. Three options are available: *1_random*, *2_fixed*, or *3_improved*. FlexSim receives the “address” where the item is to be placed according to its type.
- *A worker has finished her current job*: the message *“work”* is sent to Python. The next order is calculated based on a priority list and in the current status of the warehouse, which is available in an external file. FlexSim receives the next order to be picked-up from the warehouse.
- *A stopping condition has been met*: the message *“done”* is sent to Python. The FlexSim application is closed and Python perform some final metrics calculation based on the status of the warehouse and the total distance covered by the workers.

Heuristic computations are fast, which allows to provide a prompt answer for the FlexSim simulation, which remains frozen in the meanwhile. In the previously described model, half an hour of simulation time is computed in approximately 10 seconds, with more than 100 messages passed back and forth.

5.4 Results

Since the simulation is of stochastic nature, the results have to be treated statistically. In the example, the source of random variables is not in the arrival time for items and orders, but on the type of products that arrive and that are requested to be picked-up. The actual model implementation in FlexSim can be seen in Figure 11.

The main results are summarized in Table 1 and in Figure 12. The metrics employed were the *throughput* (τ) and the *average distance* ($\bar{d}$). The throughput can be defined in different ways depending on the context, but for a warehouse it is usually measured as the number of items processed by unit of time. Specifically, the throughput in our example was defined as the number of products that were output from the warehouse (n_{out}) divided by the total simulation time (t), as expressed in Equation (1):

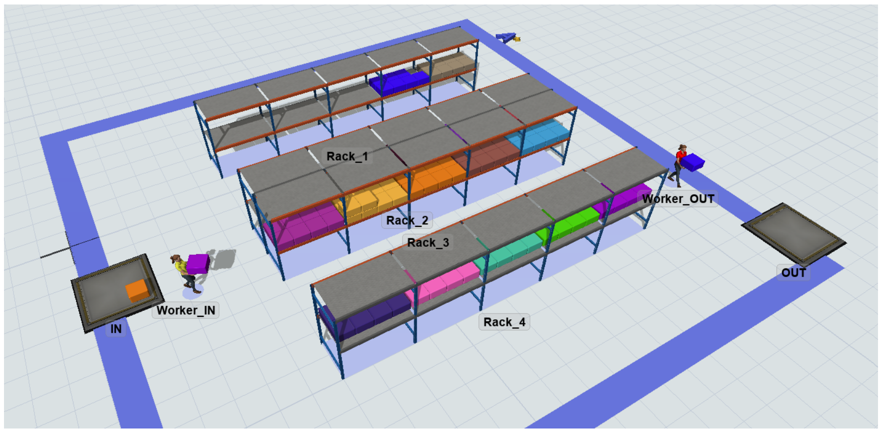

Figure 11: Screenshot from the FlexSim simulation.

$$\tau = \frac{n_{out}}{t}. \tag{1}$$

Similarly, the average distance was defined as the number of meters "walked" by unit of output products. This is calculated by summing all the individual distances (d_i^w) covered in each trip i completed by each worker w, divided by the number of products that exit the warehouse, as in:

$$\bar{d} = \frac{\sum_i \sum_w d_i^w}{n_{out}}.$$

Table 1: Results summary.

	τ		$\bar{d}$	
policy	mean	std. dev.	mean	std. dev.
1_random	227.8	6.9	62.7	1.9
2_fixed	280.2	9.9	51.0	1.8
3_improved	289.1	13.9	49.1	1.9

The results are the summary of 10 simheuristics runs for each policy (30 runs in total). It can be seen that implementing a dedicated or fixed assignment policy can indeed increase the efficiency of the warehouse. This is in line with the literature, and can be seen as a validation of the simheuristic model. The gains from introducing the improved placement logic (i.e., by looking into the future orders) are much smaller in comparison, but they are still worth the implementation. Even modest improvements in warehouse efficiency can result in significant operational cost reduction, specially for large warehouses.

6 POTENTIAL FUTURE APPLICATIONS

Integrating simulation software packages with programming languages such as Python, offers new opportunities to develop advanced intelligent simulation modeling. As a platform that generates clean and structured data, simulation output can be efficiently utilized by multiple algorithmic approaches to support decision making. In this section, a few potential future applications of the FlexSim-Python integration is discussed to provide insights for researchers and developers for further developments.

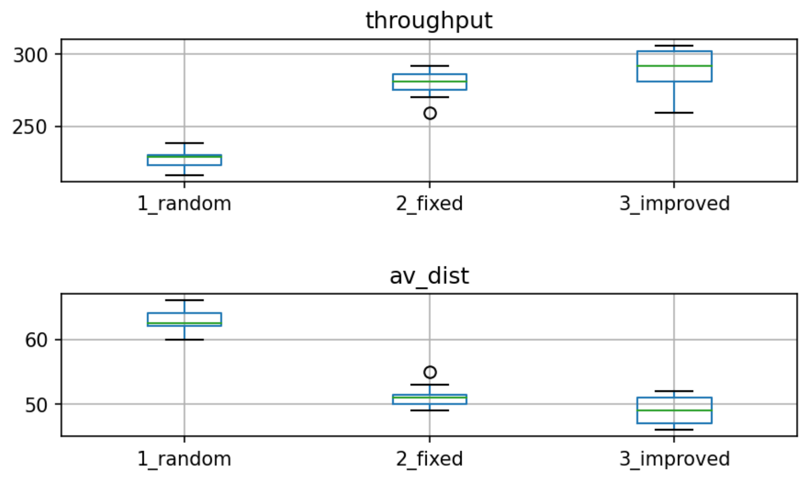

Figure 12: Simulation results showing the comparison among the different SLAP strategies.

6.1 Combined Simheuristics and Learnheuristics Methods

As simulation is not an optimization tool, it must be used in conjunction with optimization methods whenever the goal is to maximize system performance while using limited resources or minimize system operation costs while maintaining a given level of service quality. In this tutorial, we only provided an illustrative example of a simheuristic algorithm for solving SLAP. However, simheuristic algorithms are a general framework and can be employed to solve many other optimization problems under uncertain conditions (Grasas et al. 2016). For instance, simheuristic can be applied to solve stochastic vehicle routing problems with stochastic transportation time that vary according to the traffic conditions of the simulation. In this way, the solution obtained in the optimization process can be modeled in the stochastic environment of the simulation to evaluate its robustness and reliability under stochastic conditions. Furthermore, in addition to uncertain conditions, many real-world optimization problems also contain dynamic components. With the integration of FlexSim and Python we presented in this tutorial, learnheuristic-based approaches that integrate machine learning techniques within a metaheuristic or simheuristic framework can be used to address these dynamic components during the optimization process.

6.2 Applied Deep Reinforcement Learning

Reinforcement Learning (RL) is a rapidly emerging field of machine learning with promising applications to solve problems with sequential decision making. The two major components of RL are the *agent* and the *environment*. By interacting with the environment, the agent continuously improves its decision-making mechanism to maximize the expected accumulated reward. Therefore, in RL the agent discovers which actions yield the most reward by trial-and-error (Sutton and Barto 2018). In order to reach convergence, RL needs a large number of learning samples (observations) to complete its training process. This can be difficult for real-life applications, due to the limited resources and the expensive cost of performing experimental runs. Integrating RL with a simulated environment makes the learning process feasible with low cost and short time. The proposed intelligent model in this study can be applied to develop a simulation-based RL framework. A replica of a real-world model can be simulated in FlexSim, while the RL agent coded in Python can complete its training by integration with the simulation model. This could ease the use of RL in industrial applications. Moreover, RL training should be generalized enough to handle system's behavioral changes, hence updating its policy based on the continuous feedback it receives from the environment. Finally, DES packages such as FlexSim, provide visualization capabilities that can be used to illustrate the RL policy to managers and decision makers.

7 CONCLUSIONS

Modern simulation software offers many advantages to model and simulate complex stochastic systems and processes. Still, their capabilities can be largely extended by allowing DES models to interact with external optimization algorithms or machine learning methods. Thus, for example, the combination of DES models with metaheuristic optimization algorithms allow us to develop rich DES simheuristics, which can then be employed to optimize complex stochastic systems. In a similar way, DES models can also be combined with reinforcement learning or other machine learning methods, including learnheuristics. These different hybridized approaches allow for considering the system dynamic behavior and also to predict how the system will change in the future in response to the decisions made by a manager. In order to illustrate the previous ideas, the paper explains the technical details that allow to create a link between the FlexSim simulator and the popular Python programming language. In addition, a case study regarding the simulation and enhancement of a warehouse system is introduced. The numerical results support the need for including optimization methods inside DES models and, in particular, the benefits that can be obtained by developing DES simheuristics using Python and a simulation software.

As future work, we plan to: *(i)* investigate how to develop DES learnheuristics capable of addressing complex systems with both dynamic and stochastic components; and *(ii)* develop other applications of DES simheuristics in the areas of logistics, transportation, and manufacturing.

ACKNOWLEDGMENTS

This work has been partially funded by AGAUR–Spindox (2020 DI 116), the Spanish Ministry of Science (PID2019-111100RB-C21/AEI/ 10.13039/501100011033), and the Barcelona City Council – "la Caixa" Foundation under the framework of the Barcelona Science Plan 2020-2023 (grant 21S09355-001).

REFERENCES

Calvet, L., J. de Armas, D. Masip, and A. A. Juan. 2017. "Learnheuristics: Hybridizing Metaheuristics with Machine Learning for Optimization with Dynamic Inputs". *Open Mathematics* 15(1):261–280.

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.

Figueira, G., and B. Almada-Lobo. 2014. "Hybrid Simulation–Optimization Methods: A Taxonomy and Discussion". *Simulation Modelling Practice and Theory* 46:118–134.

Grasas, A., A. A. Juan, and H. R. Lourenço. 2016. "SimILS: A Simulation-Based Extension of the Iterated Local Search Metaheuristic for Stochastic Combinatorial Optimization". *Journal of Simulation* 10(1):69–77.

Juan, A. A., W. D. Kelton, C. S. Currie, and J. Faulin. 2018. "Simheuristics Applications: Dealing with Uncertainty in Logistics, Transportation, and Other Supply Chain Areas". In *Proceedings of the 2018 Winter Simulation Conference*, edited by M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, 3048–3059. Gothenburg, Sweden: Institute of Electrical and Electronics Engineers, Inc.

Mahroof, K. 2019. "A Human-Centric Perspective Exploring the Readiness Towards Smart Warehousing: The Case of a Large Retail Distribution Warehouse". *International Journal of Information Management* 45:176–190.

Nordgren, W. B. 2002. "FlexSim Simulation Environment". In *Proceedings of the Winter Simulation Conference*, edited by S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, 250–252. Orem, Utah: Institute of Electrical and Electronics Engineers, Inc.

Peyman, M., P. Copado, J. Panadero, A. A. Juan, and M. Dehghanimohammadabadi. 2021. "A Tutorial on How to Connect Python with Different Simulation Software to Develop Rich Simheuristics". In *Proceedings of the 2021 Winter Simulation Conference*, edited by S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper. Phoenix, Arizona: Institute of Electrical and Electronics Engineers, Inc.

Reyes, J., E. Solano-Charris, and J. Montoya-Torres. 2019. "The Storage Location Assignment problem: A Literature Review". *International Journal of Industrial Engineering Computations* 10(2):199–224.

Rodrigues, E. R., A. J. Preto, and S. Stephany. 2005. "A New Parallel Environment for Interactive Simulations Implementing Safe Multithreading with MPI.". In *Computer Architecture and High Performance Computing, Symposium on*, 151–158. Institute of Electrical and Electronics Engineers Computer Society.

Sutton, R. S., and A. G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

van den Berg, J. P., and W. H. Zijm. 1999. "Models for Warehouse Management: Classification and Examples". *International Journal of Production Economics* 59(1-3):519–528.

van Rossum, G., and F. L. Drake. 2009. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.

von Rueden, L., S. Mayer, R. Sifa, C. Bauckhage, and J. Garcke. 2020. "Combining Machine Learning and Simulation to a Hybrid Modelling Approach: Current and Future Directions". In *Advances in Intelligent Data Analysis XVIII*, edited by M. R. Berthold, A. Feelders, and G. Krempl, 548–560. Cham: Springer International Publishing.

Wang, Y. R., and A. N. Chen. 2016. "Production Logistics Simulation and Optimization of Industrial Enterprise based on FlexSim". *International Journal of Simulation Modelling* 15(4):732–741.

Xie, J., Y. Mei, A. T. Ernst, X. Li, and A. Song. 2014. "A Genetic Programming-based Hyper-Heuristic Approach for Storage Location Assignment Problem". In *2014 IEEE Congress on Evolutionary Computation (CEC)*. July 6th-11th, Beijing, China, 3000-3007.

Zhou, T., D. Tang, H. Zhu, and Z. Zhang. 2021. "Multi-Agent Reinforcement Learning for Online Scheduling in Smart Factories". *Robotics and Computer-Integrated Manufacturing* 72:102202.

Zhu, X., R. Zhang, F. Chu, Z. He, and J. Li. 2014. "A FlexSim-based Optimization for the Operation Process of Cold-Chain Logistics Distribution Centre". *Journal of Applied Research and Technology* 12(2):270–278.

Zielinski, K. M., L. V. Hendges, J. B. Florindo, Y. K. Lopes, R. Ribeiro, M. Teixeira, and D. Casanova. 2021. "Flexible Control of Discrete Event Systems Using Environment Simulation and Reinforcement Learning". *Applied Soft Computing* 111:107714.

AUTHOR BIOGRAPHIES

JONAS F. LEON is a predoctoral researcher at the ICSO group at the IN3 in the Computer Science Dept. at the Universitat Oberta de Catalunya. He also works as Business Analyst at Spindox. His main research interests include advanced simulation methods and complex systems analysis. His email address is jofule@uoc.edu.

MOHAMMAD PEYMAN is a PhD candidate and researcher in the ICSO group at Universitat Oberta de Catalunya. He holds a BSc in Mechanical Engineering from Azad University of Iran and a MSc in Modeling for Science and Engineering (Data 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.

YUDA LI is a predoctoral researcher at the ICSO research group at Universitat Oberta de Catalunya (Spain). He holds a BSc in Aeronautical Management from the Universitat Autonoma de Barcelona and a Msc in Computational Engineering and Mathematics from the Universitat Rovira i Virgili. His main research interests are optimization problems, metaheuristics, and machine learning. His email address is yli1@uoc.edu.

MOHAMMAD DEHGHANIMOHAMMADABADI is Associate 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 different 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.

LAURA CALVET is an Assistant Professor of Statistics in the Computer Science Dept. at the UOC. Her main lines of research are: (*i*) design of optimization algorithms relying on metaheuristics, machine learning and/or simulation applied to sustainable logistics & computing; (*ii*) applied statistics & economics. Her email address is lcalvetl@uoc.edu.

PAOLO MARONE is Spain Country Manager at Spindox, a technology company that offers products and services in the area of Advanced Analytics and Decision Support Systems. Additionally, he is Director of the Global Master of Supply Chain at the EAE Business School. He has extensive experience in the logistic industry in various managerial positions. His email address is paolo.marone@spindoxgroup.com.

ANGEL A. JUAN is a Full Professor in the Dept. of Applied Statistics & Operations Research at the Universitat Politècnica de València (Spain). 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 the Massachusetts Institute of Technology and the Georgia Institute of Technology. His main research interests include applications of simheuristics and learnheuristics in computational logistics, transportation, and finance. He has published more than 130 articles in JCR-indexed journals and over 300 papers indexed in Scopus. His website address is http://ajuanp.wordpress.com and his email address is ajuanp@eio.upv.es.