Tampere University

Niilo Viheriäranta

CO-SIMULATION USING PYTHON AND SIMULINK

Bachelor's thesis
Faculty of Engineering and
Natural Sciences
Examiner: Mohammad Heravi
1/2024

ABSTRACT

Niilo Viheriäranta: Co-simulation using Python and Simulink
Bachelor’s thesis
Tampere University
Teknisten tieteiden kandidaatin tutkinto-ohjelma
January 2024

Co-simulation is a methodology of computer analysis and simulation. This technique involves different kinds of specialized simulation programs interacting and working simultaneously to reach more comprehensive results regarding complex systems. It is an approach that is very beneficial for multidisciplinary experiments that include various subsystems of a model that require distinct areas of expertise or simulation approaches.

This thesis studies co-simulation using Python and Simulink. The thesis starts with a short literature review where different methods of communication between Python and Simulink are discovered and their suitability for co-simulation is discussed. In addition, some prior research on the topic is presented. Co-simulation using these two platforms is quite a new technology so the research available was limited. Out of the methods discovered in the literature review four were chosen as most suitable for co-simulation. The chosen methods were MATLAB Engine API, Python code in Simulink, TCP/IP (Transmission control protocol/internet protocol), and extracting a model from Simulink to C-code and embedding it in Python.

Each method was used to perform a simulation and the process of conducting the simulation was explained. For the MATLAB Engine API, a computational cost experiment was conducted because it could be used in two different ways. The results showed that one way is more suitable for simple simulations and the other for more complex simulations. The other methods were found to have their own benefits and limitations as well. Embedded C-code was very fast but only works for discrete systems. Python classes can be used to control Simulink models, but the Simulation needs to be run from a script and requires precise setup steps. TCP/IP is the most suitable for parallel processing but could prove difficult to use with very complex data. Finally, co-simulation was used to train a reinforcement learning agent to play a dice game. The experiment was successful, and it showcased the possible real-life utility of co-simulation.

Keywords: Co-simulation, Python, Simulink, reinforcement learning

TIIVISTELMÄ

Niilo Viheriäranta: Yhteissimulaatio Pythonia ja Simulinkiä käyttäen
Kandidaatintyö
Tampereen yliopisto
Teknisten tieteiden kandidaatin tutkinto-ohjelma
Tammikuu 2024

Yhteissimulaatio on tietokoneanalyysin tekniikka, jossa kahta tai useampaa simulaatioalustaa hyödynnetään yhtaikaisesti monimutkaisten ongelmien ratkaisemiseen. Sen avulla pystytään saavuttamaan kattavampia tuloksia ongelmissa, joissa systeemin eri osat vaativat erilaisia mallinnuskyvykkyyksiä. Yhteissimulaation hyötyjä ovat esimerkiksi laajemmat ratkaisijavaihtoehdot, rinnakkaisprosessointi ja immateriaalioikeuksien suojaaminen.

Tässä kandidaatintyössä tutkitaan yhteissimulaation toteuttamista Pythonin ja Simulinkin avulla. Alussa on toteutettu lyhyt kirjallisuuskatsaus, jossa ensin tutkitaan erilaisia tapoja siirtää dataa sekä toiminnallisuutta alustojen välillä ja sitten etsitään aiempaa tutkimusta yhteisimulaatiosta. Koska yhteissimulaatio näitä kahta alustaa käyttäen on melko uusi aihe, tutkimuksia löytyi suhteellisen vähän. Kirjallisuuskatsauksessa löydetyistä metodeista valittiin neljä parhaimmin yhteissimulaatioon sopivinta tapaa tutkittavaksi. Nämä ovat Python koodin käyttö Simulinkissä, MATLAB Engine API, mallin muuttaminen C-koodiksi ja sen sulauttaminen Pythoniin sekä TCP/IP-yhteyden (Transmission control protocol / internet protocol) käyttäminen.

Jokaisen tavan toteuttaminen esiteltiin ja niitä kaikkia käytettiin simulaation ajamiseen. Lisäksi MATLAB Engine API:lla pystyi toteuttamaan simulaation kahdella eri tyylillä, joten tyylien resurssivaatimukset mitattiin ja data esiteltiin. Huomattiin, että toinen tyyli sopii paremmin lyhyisiin yksinkertaisiin simulaatioihin ja toinen taas paremmin pidempiin monimutkaisiin simulaatioihin. Muistakin tavoista huomattiin erilaisia hyötyjä ja haittoja. Sulautettu C-koodi on nopein, mutta se toimii vain diskreeteillä systeemeillä. Python-koodia pystytään käyttämään Simulinkissä niin että molempien hyödyt saadaan käyttöön, mutta valmistelut vaativat tarkkoja määreitä. TCP/IP-yhteys sopii parhaiten rinnakkaisprosessointiin mutta monimutkaisen datan siirtäminen voi muuttua vaikeaksi. Lopuksi esitellään käytännön esimerkki, jossa vahvistusoppimisen avulla koulutetaan toimija pelaamaan erästä noppapeliä mahdollisimman tehokkaasti. Toimija oli tehokkaampi kuin satunnaisesti valitseva toimija, mikä osoitti yhteissimulaation hyödyllisyyden esimerkiksi koneoppimisessa.

Avainsanat: Yhteissimulaatio, Python, Simulink, vahvistusoppiminen

PREFACE

I want to thank Mohammad Heravi for his guidance in making this thesis. Our weekly meetings were always constructive and helpful. He also provided me with useful resources to aid in research as well as in writing the thesis.

Tampere, 14 January 2024

Niilo Viheriäranta

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This thesis studies co-simulation using two different platforms MATLAB's Simulink and Python. Co-simulation is a methodology of computer analysis and simulation. This technique involves different kinds of specialized simulation programs interacting and working simultaneously to reach more comprehensive results regarding complex systems. It is an approach that is very beneficial for multidisciplinary experiments that include various subsystems of a model which require distinct areas of expertise or simulation approaches.

Co-simulation has many benefits compared to using single-platform simulation. Some models include subsystems that have very different physical behaviours and time scales. With these systems, it could prove difficult to develop a dynamic formulation that represents every phenomenon involved with the system. Co-simulation allows for assigning a specific solver to each subsystem meaning its equations and implementation can be tailored to represent its real-life counterpart more realistically.

In addition, using single-platform simulation sometimes requires sacrificing intellectual property in order to reach an accurate result. This problem arises because every subsystem and its details need to be accessible to the system solver. In some industrial applications safeguarding at least some of the information regarding the system is very important. Co-simulation allows for determining a particular interface of data that is shared between the solvers and the rest can remain private.

Furthermore, co-simulation allows for parallel processing of heavy models. Simulating very heavy systems could result in increasingly high simulation times making the work inefficient, with co-simulation the computational workload can be shared between two or more different processing units. This could increase efficiency in many applications.

This thesis specifies on co-simulation with Python and Simulink. Simulink is an extension application of MATLAB provided by MathWorks. It allows users to model a large variety of different scenarios. It provides toolboxes for almost all fields of industry and is widely used in companies. However, Simulink also has limitations for example in its machine learning capabilities. [1] In comparison, Python is an open-source coding language that has a wide range of libraries that could be used to address some of the limitations of Simulink. This is why it would be very beneficial to find approaches to co-simulation between these two platforms. Furthermore, many companies have legacy models in Simulink and because Python is free to use, they could harness it without enduring any extra costs.

The objective of this paper is to find approaches to co-simulation with the two platforms presented above. In addition, the benefits and limitations of each approach will be presented and their effects on usefulness will be assessed. Finally, a use case will be defined and through it the benefits of co-simulation will be presented.

2. LITERATURE REVIEW

This chapter focuses on previous research around Python, Simulink and their integration in terms of co-simulation. In addition, studies that use the two platforms in similar tasks are considered to gain an insight into the advantages and disadvantages of each platform. Exploration of the research will enable better understanding of different ways to use Simulink and Python as complimentary tools to each other. Finally, some studies that have implemented co-simulation with the two platforms will be explore.

2.1 Different methods of communication between MATLAB and Python

MathWorks presents three different ways to directly communicate between MATLAB and Python. The first of these is calling Python directly in MATLAB. Python modules can be called in MATLAB by using the “py” prefix followed by a period and the module’s name after which the inputs are entered in brackets. The version and default interpreter can be changed by running the command “pyenv” and editing the fields of the resulting MATLAB struct to match the desired properties of Python. The user is able to call self-created modules and third-party libraries in addition to the standard libraries of Python. This requires all the needed libraries to be installed and the modules to be located in the active folder in MATLAB. [2] One way this way of communication can be utilized with Simulink is to create a custom MATLAB function that takes inputs from Simulink then calls Python inside the function and returns the outputs to Simulink. This can be achieved as Simulink allows custom functions to be added to the models as blocks.

The second way presented is to call MATLAB functions from Python using the MATLAB Engine API offered by MathWorks. The engine can be imported from the Python libraries and then needs to be started by calling the appropriate method. Once the process is running the user is free to call functions from MATLAB as well as run any user-created scripts. This way uses MATLAB directly as a computational engine and so requires an active license. [3] Both of the ways mentioned above are easy and quick to set up and use. However, they have a disadvantage in that they do not support exchanging MATLAB tables as noted by Haider et al. [4].

The third direct way of communication is creating a custom blockset in Simulink. The Simulink blockset [5] designer offers a feature called Python Importer. The Importer wizard can be given a single file of Python code or a directory of several files. It scans the files and recognizes functions in them. After that, the user can choose which functions will be created as Simulink blocks and then check the types and sizes of the inputs and outputs. When the process is complete the user will have a custom blockset that they can use in Simulink models. [5]

There are also several ways of indirect communication between the platforms. Most of these are also provided by MathWorks themselves. From Simulink, models can be extracted using the Embedded Coder. The coder can compile a shared library of C-code to be used directly in Python or it can generate raw C/C++ -code.[6] Python has a library called Ctypes that provides methods and datatypes for Python to interface directly with shared libraries written in C-code. Integrating the platforms in this way could provide very useful as already compiled C-code is greatly faster than either Python or MATLAB are even individually.

The MATLAB Compiler SDK was released in 2015. This allows for compiling programs into packages that can be inserted into the Python code. The compiler works for Simulink models as well. However, Simulink models are exported using the Functional Mock-up Interface standard (FMI) when they are needed for co-simulation. [7] MATLAB also allows for the importing of Deep learning models through the Open Neural Network Exchange (ONNX) format [8]. However, for the purposes of this study, this is not very useful as it only works for pre-trained models thus Simulink would not be able to be used in the training of the model.

Another way of indirect communication is using different types of files to store data and transfer it across platforms. Keshavarz and Mojra [9] save data from MATLAB into Python files and return results from Python as txt files. MathWorks [8] suggests using Parquet files to transfer data in a tabular format. Most simulation models save data as vectors in Simulink so transferring tabular data is not very relevant for this study.

As can be seen from the research done above there are many ways to transfer data and functionality from MATLAB to Python and vice versa. The direct ways allow for the use of both platforms simultaneously whereas the indirect ways use the platforms in turns for different tasks. Excluding the MATLAB Compiler SDK which builds the wanted model as a Python package that can be run in the Python code.

2.2 Prior research

There is a limited amount of previous research on the possible uses of integrating the two platforms in terms of simulation. Tshiani and Umenne [10] used the approach of calling Python directly from a MATLAB function and then incorporating that function block into the Simulink model. Their work shows that using the two platforms together can be beneficial to a multitude of industries that already rely heavily on Simulink to model processes. Another use-case proposed concerning models that have their properties heavily change as the process advances is to build multiple models for different phases of the process and use Python as an intermediate to convey data from one model to another as proposed by Mehlhase [11]. This sort of modular system could be very useful in the chemical industry in the study of buffer solutions for example. The approach is less relevant to this study as the ways of including Python functionality into the simulation itself are studied in this thesis. Their method uses Python to initialize the next model with the results of the previous one in the modular design which is not co-simulation but rather simulating multiple models in succession. From the lack of prior research, the need for further research can be seen quite clearly.

2.3 Conclusions from the literature review

From the literature review, it can be concluded that there are several different methods to communicate between the two platforms. Some of them are more suitable for co-simulation than others which is to be expected. However, even with multiple methods available, there is very little prior research regarding the topic. Some studies were found where co-simulation was utilized to answer a different research question, but it was not the main topic of research.

The aim of this thesis is to provide some more insight into co-simulation with the two chosen platforms. Next, some of the methods discovered that were more suitable for co-simulation will be explored more thoroughly. Some concrete examples will be shown with experimental data as well. Finally, a use case will be presented to showcase the capabilities of co-simulation that could be utilized in real-life problems.

3. DIFFERENT METHODS OF CO-SIMULATION WITH PYTHON AND SIMULINK

This chapter presents how the different ways of communication discovered in Chapter 2 could be utilized in co-simulation. Not all of the ways are discussed but the ones that were found to be best suited for co-simulation purposes. The technical details of conducting a simulation will be discussed and for some methods, experimental data on computational cost is also presented.

3.1 Using Python in Simulink

In the following chapter, the usage of Python code in MATLAB with Simulink is discussed. An approach to co-simulate using this approach is presented as well as its benefits and limitations. Like it is discussed in Chapter 2 when a compatible interpreter is installed on the machine MATLAB can run Python code. After starting the Python environment in MATLAB, the user needs to make sure that all of the locations of the modules they want to use are included in the Python search path. Once the needed module locations are in the search path their classes and functions can be used with the following syntax: “py.moduleName.functionName(arguments)”.

In this thesis, the focus was on using a Python class as the controller of a Simulink model. With this in mind, one of the limitations of this approach is that it was found that an instance of a Python class cannot be used as an input or output of a Simulink user-defined function because the model compiler is not able to determine its size in advance. This means that a MATLAB script is needed to run the simulation. The script saves the input into the workspace which the Simulink model accesses and then one timestep is simulated and the model saves the outputs into the workspace as well. From there the script uses the latest output to get the control from an instance of the Python class. This procedure is then repeated until the simulation has reached its end. The script for running the simulation can be seen below in Program 1.

```
1   mld = "python_from_simulink_example";
2   simIn = Simulink.SimulationInput(mld);
3   simIn = setModelParameter(simIn,"SaveFinalState","on");
4   simIn = setModelParameter(simIn,"SaveOperatingPoint","on");
5   
6   time_span = 1:1:100;
7   x = sin(time_span * 0.04);
8   y = zeros(1, length(time_span));
9   py.importlib.import_module("pi_control");
10  ctrl = py.pi_control.PiController();
11  
12  c = 0;
13  ref = 10
14  
15  for i = time_span
16  
17      u = x(i) + c;
18      simIn = simIn.setModelParameter("StopTime",num2str(i *
19  0.1));
20      out = sim(simIn);
21      vdpOP = out.xFinal;
22      simIn = simIn.setInitialState(vdpOP);
23      y(i) = out.simout(2);
24  
25      c = double(ctrl.control(ref-y(i)));
26  end
27  
28  plot(time_span, y)
```

Program 1: MATLAB script for running a simulation with a Pi-controller implemented in Python.

On line 10 the Python-module which contains the controller is imported into the MATLAB session and then on line 11 an instance of the controller class is called and saved to a variable. After this we can use its method on line 25 to retrieve the appropriate control from the controller. The code in “pi_control.py” can be seen below in Program 2.

```python
class PiController:
    def __init__(self):
        self.y = 0
        self.e_sum = 0
        self.u = 0

    def control(self, e):

        self.e_sum += e
        self.u = 1 * e + 0.001 * self.e_sum

        return self.u
```

Program 2: Pi-controller implementation in Python.

In this paper the Python program is a Pi-controller however there are many possibilities of what types of programs could be used. For example, different sorts of filters or machine learning modules could be utilized this way to see their compatibility with an existing Simulink model. The communication with Python does not slow down the simulation noticeably at least with this simple controller algorithm however with more complex Python programs a larger running time is to be expected.

The benefit of this approach is that the Python module can use all of the various libraries of Python which gives an abundance of flexibility in terms of what sort of attributes can be implemented into the simulation. In addition, all of the superior modeling capabilities of Simulink can be used. This is very important because Simulink has a lot of exclusive toolboxes for modelling complex systems and its interface is very easy to use. Companies that already have existing Simulink models regarding their products can utilize this to implement different useful Python programs as a part of the simulations quite easily using this method. One inconvenience of the method is that every time MATLAB is started the user needs to manually add the Python program's location into the Python search path.

3.2 MATLAB Engine API

As already discussed in chapter 2 the MATLAB engine provides a library of methods for the user to call MATLAB functions directly from Python. The engine starts a session of MATLAB which will be used to carry out the computations of the functions. The functions that can be carried out include performing Simulink simulations with changing the inputs from Python as well as extracting the outputs.

For the user to start the use of the engine the MATLAB Python library needs to be installed on the user's machine. The installation can be executed using the Python package installer(pip) from the terminal when a version of Python is already installed. After that the user is able to import the library like any other Python library with the "import"-command when writing the program. With the library imported a MATLAB session can be started by calling the method `connect_matlab`. When saved into a variable this variable is an object representation of the MATLAB session and MATLAB functions can be called as methods of the object.

This paper presents two methods of running a simulation using the engine and introduces experimental data regarding the computational cost of each method. The first of the methods is accessing the status of the simulation using the `set_param` and `get_param` functions of the engine. These functions can start, pause, continue, and see if the simulation has ended when given the correct arguments. The former of the functions takes an odd number of arguments. The first argument is the name of the model or a path to its specific block that is to be modified then the next two arguments specify the attribute that is to be changed and its new value. The function can take more than one of these attribute-value pairs and they are executed from left to right, but all the pairs must affect the same part of the model. This can be used to set the input of the model by changing the value of a constant block in the Simulink model. The latter function of the two is called similarly it only loses the final argument so the value of the attribute because that is the function's return value. All of the values associated with these functions must be of type string which means the arguments of both functions and the return value of `get_param`. Finally, the outputs of the simulations need to be saved into the workspace in the Simulink model and then can be accessed with the engine as one of its attributes is the workspace and a specific output can be accessed using the syntax of a Python dictionary with the key being the output variables name. The interactions between MATLAB Engine and Python can be seen below in a sequence diagram in Figure 1.

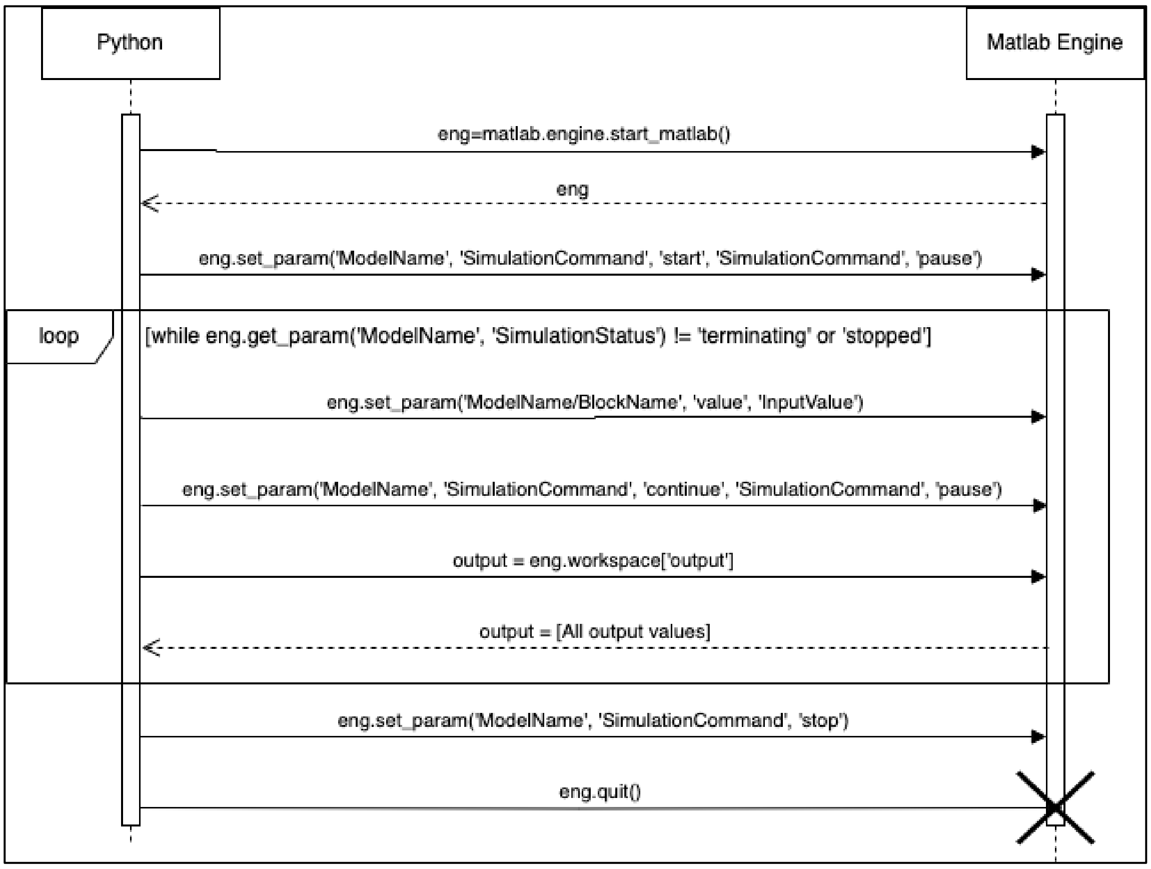

Python
Matlab Engine
eng=matlab.engine.start_matlab()
eng
eng.set_param('ModelName', 'SimulationCommand', 'start', 'SimulationCommand', 'pause')
loop
[while eng.get_param('ModelName', 'SimulationStatus') != 'terminating' or 'stopped']
eng.set_param('ModelName/BlockName', 'value', 'InputValue')
eng.set_param('ModelName', 'SimulationCommand', 'continue', 'SimulationCommand', 'pause')
output = eng.workspace['output']
output = [All output values]
eng.set_param('ModelName', 'SimulationCommand', 'stop')
eng.quit()

Figure 1: Sequence diagram of the first method of conducting a simulation with MATLAB Engine.

The second method of conducting a simulation using the MATLAB Engine is to create a SimulationInput -object. This object allows the user to edit the Simulink model's parameters such as the stop time. Using this method, the input and output are accessed through the workspace, and the operating point of the model is saved after each timestep and used as the initial state of the next timestep. The operating point is a data object that contains all the necessary information about the simulation. The simulation of one timestep is conducted by setting the stop time equal to the timestep with the help of the SimulationInput -object. The interactions between the two platforms using this method can be seen below in Figure 2.

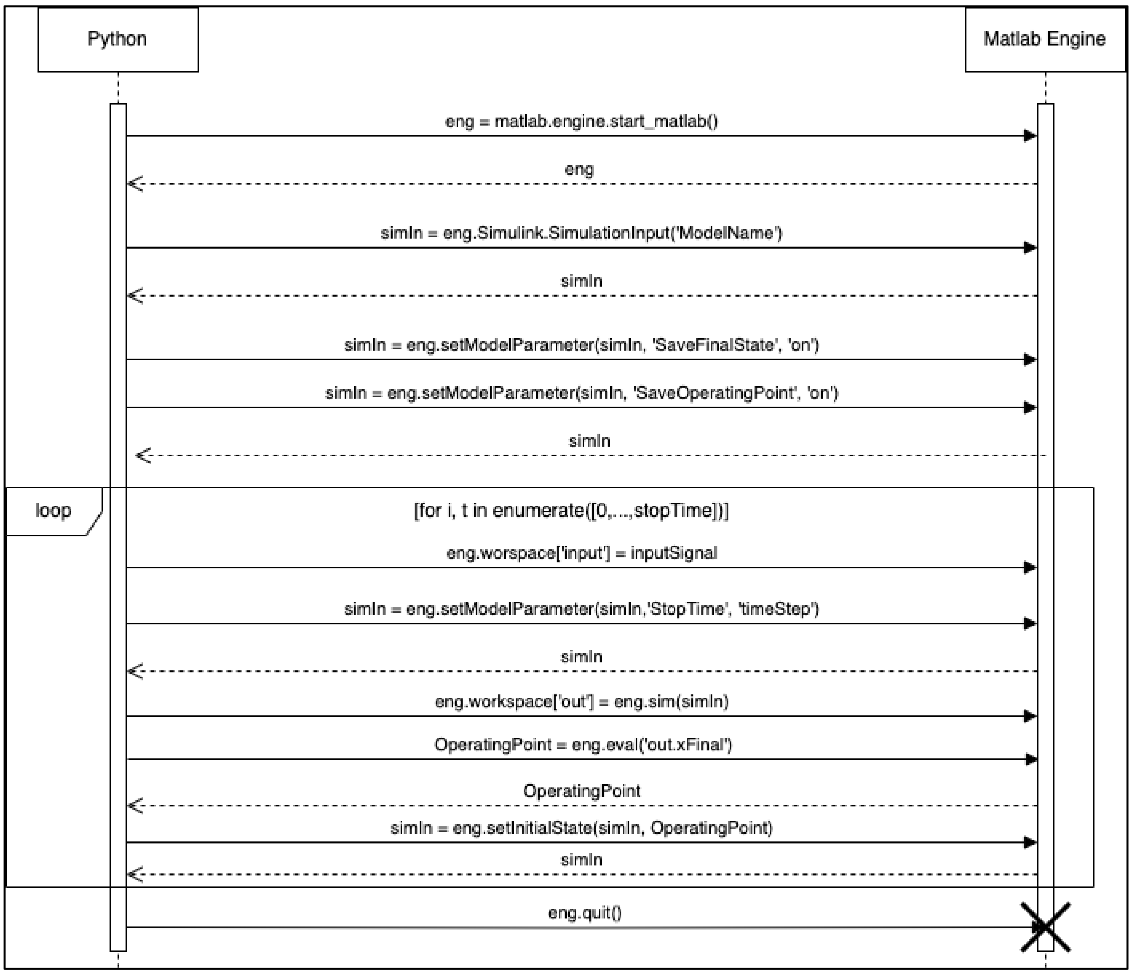

Python
Matlab Engine
eng = matlab.engine.start_matlab()
eng
simIn = eng.Simulink.SimulationInput('ModelName')
simIn
simIn = eng.setModelParameter(simIn, 'SaveFinalState', 'on')
simIn = eng.setModelParameter(simIn, 'SaveOperatingPoint', 'on')
simIn
loop
[for i, t in enumerate([0,...,stopTime])]
eng.worspace['input'] = inputSignal
simIn = eng.setModelParameter(simIn,'StopTime', 'timeStep')
simIn
eng.workspace['out'] = eng.sim(simIn)
OperatingPoint = eng.eval('out.xFinal')
OperatingPoint
simIn = eng.setInitialState(simIn, OperatingPoint)
simIn
eng.quit()

Figure 2: Sequence diagram of conducting a simulation using the second method with MATLAB Engine.

Each of these two methods was simulated ten times with two different Simulink models. One of the models is very simple and the other is more complex with the difference in computational cost around the order of ten. The Simulink model complexity comparison was done with Simulink’s built-in model profiler. During the simulations, each function that communicates with the MATLAB Engine was timed as well as the time for one timestep of the simulation. The results of the timing can be seen in the tables below. First, the results for the simpler model are shown for each method in Tables 1 and 2 respectively.

Table 1: Times to run each communication with MATLAB for the first method with a simple.

Action (s)	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	Run 6 (s)	Run 7 (s)	Run 8 (s)	Run 9 (s)	Run 10 (s)
connect_to_matlab	28,08983	17,16702	14,85229	15,14788	14,55788	14,34156	13,98869	15,59587	14,19086	14,38254
set_param_input	0,00236	0,00230	0,00221	0,00218	0,00217	0,00216	0,00214	0,00209	0,00208	0,00254
set_param_sim	0,02096	0,01899	0,01813	0,01751	0,01771	0,01693	0,01744	0,01667	0,01659	0,01902
eng.workspace[‘output’]	0,00486	0,00471	0,00438	0,00459	0,00429	0,00453	0,00438	0,00431	0,00429	0,00497
while loop (1 loop)	0,03029	0,02802	0,02637	0,02620	0,02590	0,02540	0,02562	0,02475	0,02473	0,02859

Table 2: Times to run each communication with MATLAB for the second method with a simple model.

Action	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	Run 6 (s)	Run 7 (s)	Run 8 (s)	Run 9 (s)	Run 10 (s)
start_matlab	4,79301	5,25251	5,62030	5,44484	5,01799	5,13013	4,87337	5,45661	5,11198	5,03494
SimulationInput	2,71804	2,75303	2,75685	2,70729	2,66346	2,66646	2,67936	2,73686	2,70340	2,67692
setModelParameter SaveFinalState	0,05334	0,05200	0,05413	0,05651	0,05401	0,05252	0,05401	0,05601	0,05300	0,05800
eng.sim	0,22515	0,22275	0,22683	0,22036	0,22220	0,22318	0,22021	0,22734	0,22026	0,22437
setInitialstate	0,06789	0,06865	0,06962	0,06742	0,06815	0,06872	0,06772	0,06948	0,06779	0,06818
setModelParameter 'SaveOperatingPoint'	0,02100	0,02045	0,02000	0,02200	0,01951	0,02000	0,01952	0,01951	0,02202	0,02100
setModelParameter 'StopTime'	0,03775	0,03843	0,03911	0,03788	0,03845	0,03873	0,03801	0,03878	0,03798	0,03819
eng.workspace['input']	0,00158	0,00166	0,00164	0,00155	0,00157	0,00164	0,00167	0,00160	0,00162	0,00164
for loop (1 iteration)	0,33501	0,33403	0,33979	0,32983	0,33298	0,33480	0,33016	0,33975	0,33020	0,33494

The tables show times for each communication made with MATLAB during the simulation as well as finally the time for executing one timestep of the simulation fully. The actions carry similar names to those in the sequence diagrams above. From this data, the means for each timed unit were calculated and can be seen below in Tables 3 and 4.

Table 3: Mean values corresponding to the data in Table 1.

Action	Mean (s)
connect_to_matlab	16,231442
set_param_input	0,002223
set_param_sim	0,017995
eng.workspace[‘output’]	0,004531
while loop (1 iteration)	0,026587

Table 4: Mean values corresponding to the data in Table 2.

Action	Mean (s)
start_matlab	5,17357
SimulationInput	2,70617
setModelParameter 'SaveFinalState'	0,05435
eng.sim	0,22327
setInitialstate	0,06836
setModelParameter 'SaveOperatingPoint'	0,02050
setModelParameter 'StopTime'	0,03833
eng.workspace['input']	0,00162
for loop (1 iteration)	0,33415

As can be seen from the mean values above for each method the most time-consuming step is the initial connection to MATLAB. Compared to the time it takes for the connection the other steps in each method are significantly faster. However, it can be seen that the first method takes longer to initialize the simulation than the second but is then faster in running the simulation. This could be useful to know when running longer simulations as the longer time to run a timestep with the second method could result in unnecessarily long simulations.

Next, the heavier Simulink model was timed in terms of the same functions as the simpler model the Python codes stayed constant only the name of the model needed to be changed. This assured a fair comparison of how the co-simulations react to a more complex Simulink model in terms of efficiency. The results for the timing of the heavier model using the two methods can be seen below in Tables 5 and 6.

Table 5: The times of conducting a simulation using the first method with the complex model.

Action	**Run 1 (s)**	**Run 2 (s)**	**Run 3 (s)**	**Run 4 (s)**	**Run 5 (s)**	**Run 6 (s)**	**Run 7 (s)**	**Run 8 (s)**	**Run 9 (s)**	**Run 10 (s)**
connect_matlab	21,31984	19,13692	19,82316	19,04033	20,35102	19,49312	18,84882	20,32681	18,84342	19,39703
set_param_input	0,00224	0,00214	0,00248	0,00216	0,00225	0,00205	0,00210	0,00209	0,00212	0,00227
set_param_sim	0,01836	0,01772	0,01695	0,01781	0,01906	0,01735	0,01725	0,01755	0,01765	0,01794
eng.workspace['output']	0,00460	0,00430	0,00421	0,00447	0,00470	0,00431	0,00425	0,00433	0,00437	0,00468
while loop (1 loop)	0,02698	0,02598	0,02530	0,02618	0,02781	0,02542	0,02529	0,02580	0,02586	0,02703

Table 6: The times of conducting a simulation using the second method with the complex model.

Action	**Run 1 (s)**	**Run 2 (s)**	**Run 3 (s)**	**Run 4 (s)**	**Run 5 (s)**	**Run 6 (s)**	**Run 7 (s)**	**Run 8 (s)**	**Run 9 (s)**	**Run 10 (s)**
start_matlab	9,33926	5,88936	4,82640	5,47434	7,80838	5,19396	5,36976	5,34137	4,95820	5,33553
SimulationInput	8,37110	2,92150	2,61677	2,64511	4,95042	2,83598	2,78225	2,73057	2,67012	2,67928
SetModelParameter SaveFinalState	0,06751	0,05500	0,05200	0,05252	0,05732	0,05501	0,05400	0,05400	0,05152	0,05152
eng.sim	0,51324	0,46355	0,45551	0,46211	0,49717	0,48909	0,47175	0,46601	0,46132	0,46597
setInitialstate	0,14615	0,14555	0,14523	0,14513	0,14632	0,15225	0,14722	0,14876	0,14550	0,14639
setModelParameter SaveOperatingPoint	0,56866	0,02341	0,02010	0,02035	0,38809	0,02052	0,02252	0,02000	0,02000	0,02346
setModelParameter StopTime	0,07707	0,07703	0,07640	0,07666	0,07786	0,08050	0,07771	0,07867	0,07701	0,07740
eng.workspace['input']	0,00156	0,00165	0,00171	0,00158	0,00168	0,00168	0,00165	0,00167	0,00164	0,00165
for loop (1 iteration)	0,74120	0,69087	0,68196	0,68860	0,72605	0,72686	0,70145	0,69831	0,68864	0,69452

From these values, the means were calculated just like with the simpler model. These mean values can be seen in Tables 7 and 8 respectively.

Table 7: Mean values corresponding to the data in Table 5.

Action	Mean
connect_matlab	19,658047
set_param_input	0,00219
set_param_sim	0,01776
eng.workspace[’output’]	0,00442
while loop (1 loop)	0,02617

Table 8: Mean values corresponding to the data in Table 6.

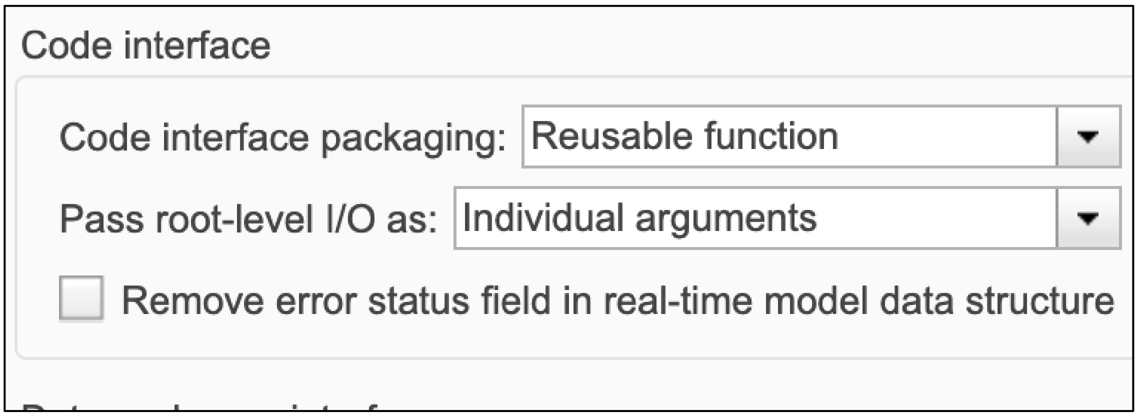
Action	Mean
start_matlab	5,95366
SimulationInput	3,52031
setModelParameter 'SaveFinalState'	0,05504
eng.sim	0,474572
setInitialstate	0,14685
setModelParameter 'SaveOperatingPoint'	0,11271
setModelParameter 'StopTime'	0,077631
eng.workspace['input']	0,001647
for loop (1 iteration)	0,703846

This data shows some interesting results. Firstly, for the first method, the times are very similar which indicates that using this method the complexity of the model has only small effects on the speed of the simulation. However, for the second method, some of the times have nearly doubled. This result indicates the different functions that are affected by the complexity of the Simulink model. Affected functions are the one that simulates a timestep, the one that saves the operating point, and the one that sets the last operating point as the initial state for the next step. It does seem logical that these functions are affected because simulating a more complex timestep will take longer and the rest of the functions interact with the operating point and in a more complex model the operating point includes a larger amount of data thus making the functions slower.

From the results above it can be concluded that the second method is more useful when dealing with simpler models and smaller simulation times because it has a smaller connection time and is easier to write in code. However, when more complex models are used, or simulation times are very long the first method proves to be a more competent tool. Despite its longer connection time, this method is faster in simulating one timestep and suffers only small performance losses when used with more complex Simulink models.

3.3 Using a shared library of C-code

The next approach to using a Simulink model in Python code is to use the Embedded coder extension of Simulink to compile the model as a shared library of C-code. The Embedded coder allows the user to select beneficial settings for compilation and then compiles the model as either only the code files or the user can choose a shared library. This will in addition to the code files generate a dynamic shared library which the user can then access from Python. For the Python integration, the most important settings are shown in picture 1 below.


Code interface

Code interface packaging: Reusable function

Pass root-level I/O as: Individual arguments

☐ Remove error status field in real-time model data structure

Picture 1: The settings needed for exporting the model and integrating it in Python.

In Python, the integration of the shared library is done using a library called Ctypes. This library contains data types that make it possible to use types compatible with C-code in Python. The user then needs to duplicate some of the data structures in the generated code to be able to call all needed functions with appropriate syntax. With the settings shown above the C-code will have three functions that allow the user to interact with the simulation. Those three functions are initialize, step, and terminate. Initialize takes as arguments pointers to the real-time model structure as well as all inputs and outputs.

Step is similar in calling convention but all of the pointers to inputs are replaced with the actual values of the inputs. Terminate only takes the real-time model structure pointer as an argument. Integration requires carefulness because the data structures in the C-code need to be duplicated exactly for the simulation to be able to be ran.

After successfully integrating the system in Python a test was conducted to see if this approach could produce accurate results. The model was simulated using both co-simulation and only Simulink with identical inputs. The plots of the simulations were recorded and are shown below in Figures 3 and 4.

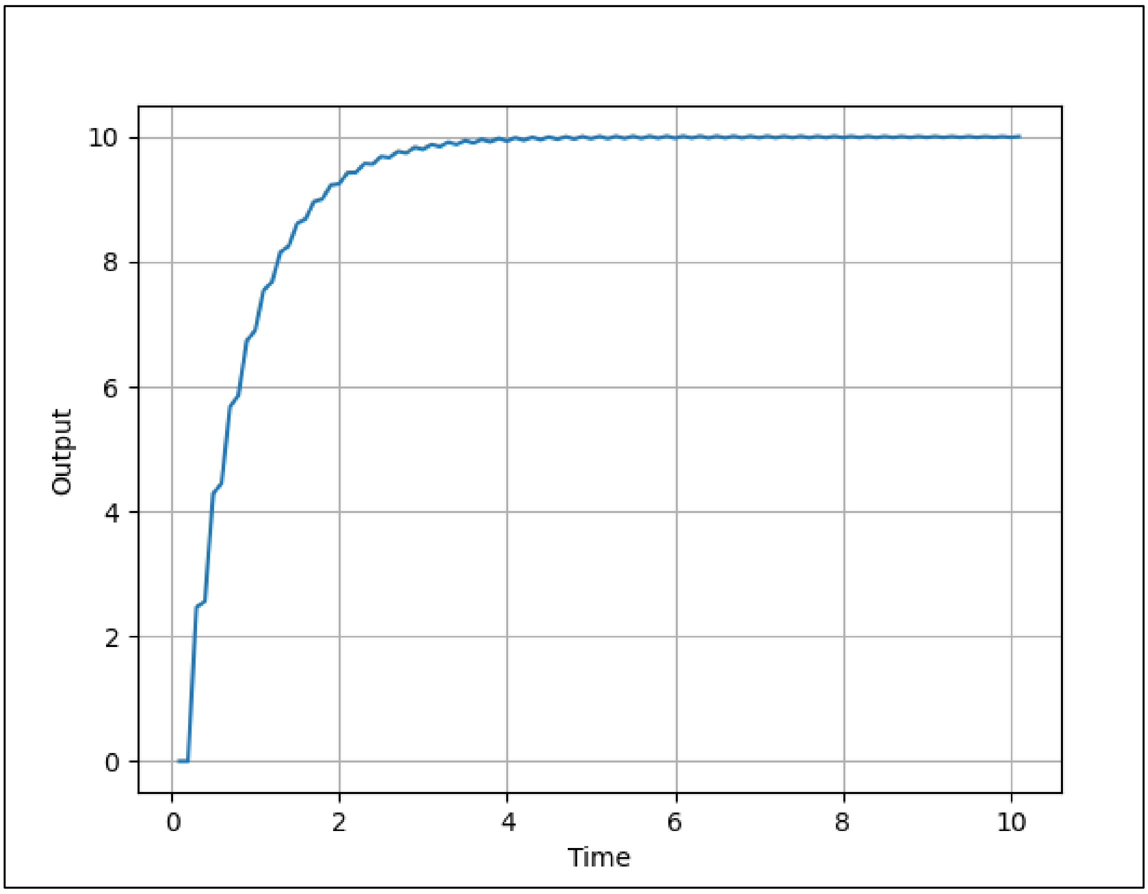


Figure 3: Plot of the simulation produced by co-simulation.

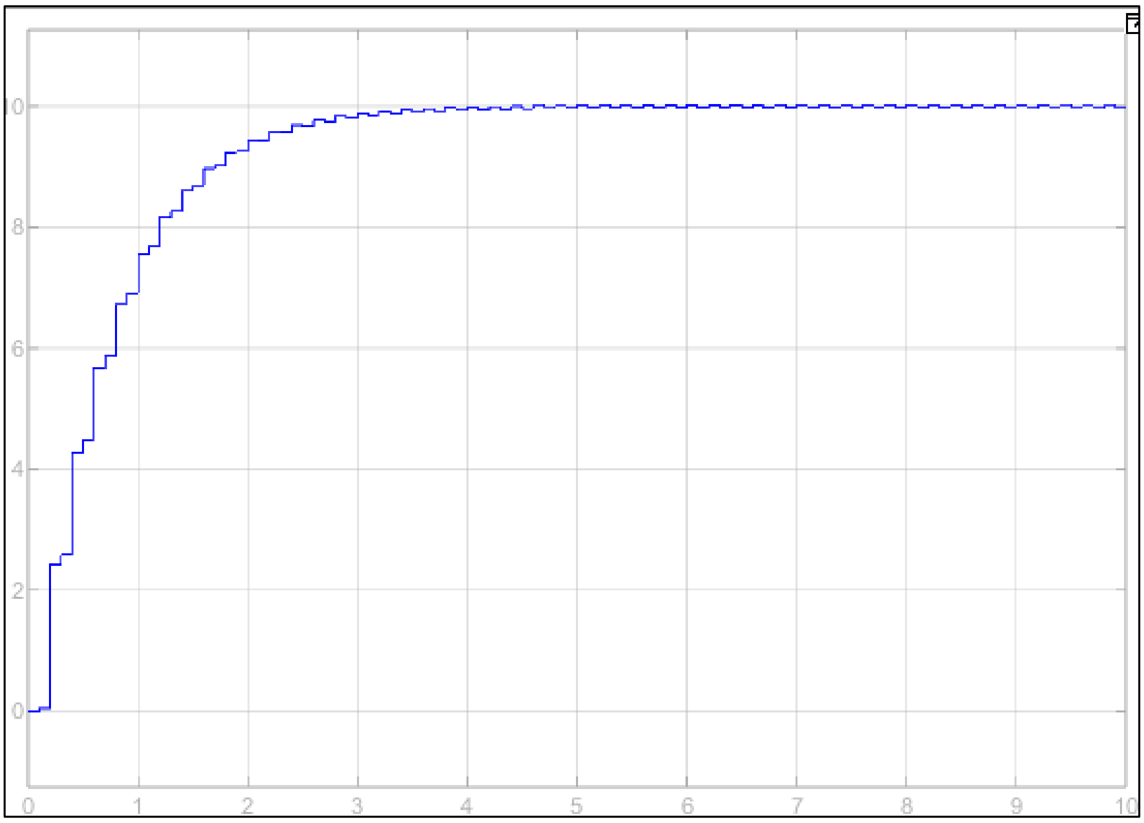


Figure 4: Plot of the simulation using only Simulink.

As can be seen from the plots the results are nearly identical. From this, it can be deduced that this approach is a viable method of co-simulation.

The benefit of this approach is that it is very fast. Code written in C is generally faster than that written in Python. The reason is that C-code is compiled whereas Python code is interpreted which means it is run line by line, so memory is not used as efficiently as it is used in C-code. This means that the only speed limitations of this approach are from the Python side. Another benefit is that after the system is extracted from Simulink to C, an active MATLAB license is no longer required. One large limitation of the generated code is that only discrete models can be integrated into Python. This is because some of the datatypes used in the generated code with continuous solvers cannot be accessed in Python even with the Ctypes library. However, with discrete models, this approach is the best to use because of its superior performance.

3.4 TCP/IP connection

A final way is using the transmission control protocol/internet protocol (TCP/IP) to send data between the two platforms. The connection transfers data through the local network using a client-server policy. For connection an IP address needs to be specified and a specific port assigned to be used by the client and the server.

Establishing the connection between the platforms is done by using the “socket”-library of Python and the “Instrument Control Toolbox” in Simulink. The specific identifiers of the connection need to match for the data to be sent. In Simulink, the toolbox provides specific blocks for both sending and receiving data using a TCP/IP connection. Because of using this connection, the data transforms into a package of bytes, thus in Python the received data needed to be unpacked using the “struct”-library, and also data to be sent back needed to be packed into a package of bytes. In Simulink, the blocks do this automatically. The utilization of this method is explained further in the next chapter. However, the basic communication structure can be seen in Figure 5 below.

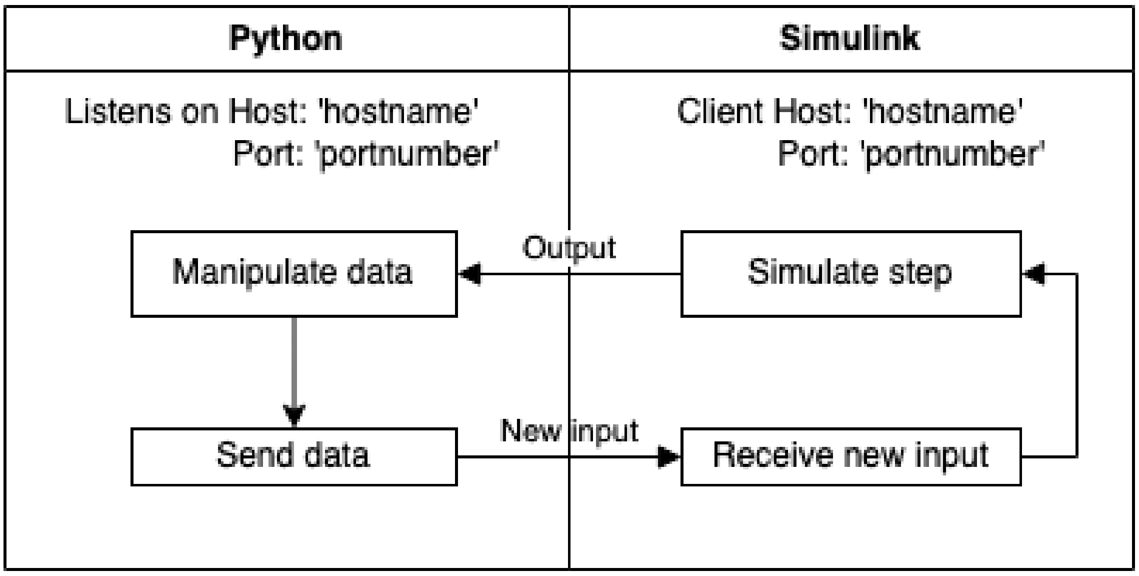


Figure 5: The communication structure between Python and Simulink with TCP/IP

In the figure in Simulink, the client’s host and port number need to match the ones Python is listening on after that the simulation can be started. Python starts processing the data once it receives data through the connection. After processing it sends the new input to Simulink through the same established connection and the loop starts again until the simulation finishes. The loop could also work in the other direction as well.

This approach also has benefits regarding its speed. In addition, for use cases that would benefit from parallel processing, this connection would be ideal. It allows the workloads of Python and Simulink to be split for two different processing units very easily as each task can be executed on one unit and a local network connects the units. In this thesis, the connection was used to send single values between the platforms. Sending more complicated data could prove to be unnecessarily complex and thus provide limitations in terms of usefulness.

4. USE-CASE AND ANALYSIS

4.1 The dice game

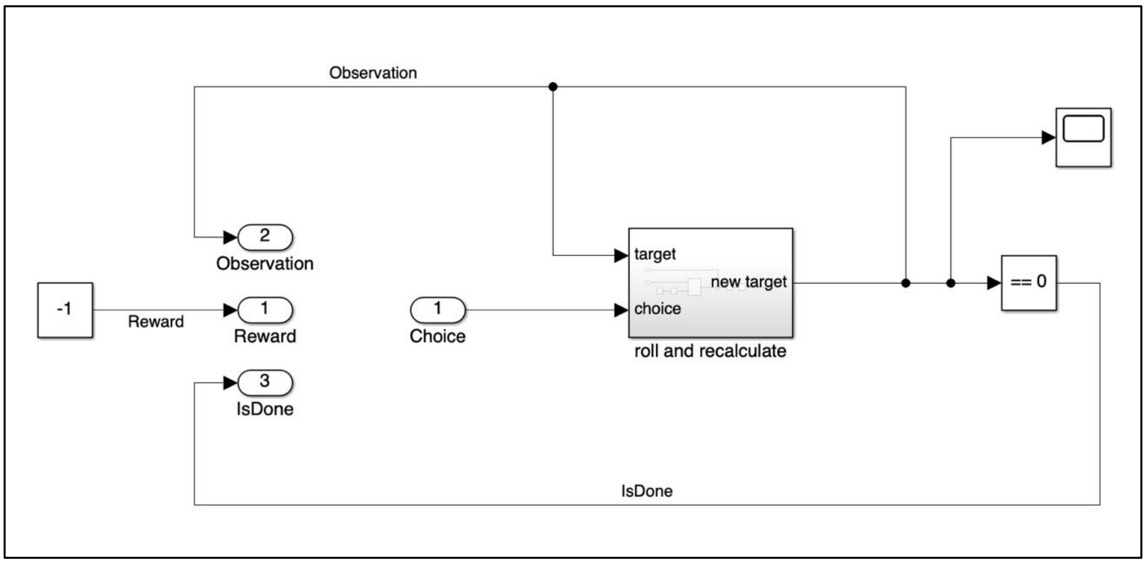
This chapter presents how the different approaches of co-simulation could be utilized through a use-case. The example system is a dice game where the player has a choice of five dice and is given a starting number between one and twenty. The player needs to reach zero from the starting number with the least number of die throws where the result of each throw is deducted from their number. Each die is different in terms of the numbers on its six sides. The dice used in this use-case are shown in Table 9 below.

Table 9: The different sides of the dice used in the game.

	Side number					
Die 1	1	1	1	5	5	8
Die 2	2	2	3	3	6	6
Die 3	1	4	4	4	5	7
Die 4	2	2	6	6	7	8
Die 5	3	4	5	7	8	8

So, when playing the game, the player needs to make logical decision based on how far away their sum of results is from the target. If the player goes over zero their score is updated to be as much away from the target as they went over the target. For example, a player's number is 2 and they throw a six that means their score goes four over zero which means their new number is updated to 4. In this use-case one approach of co-simulation is used to train a reinforcement learning agent to have the best possible policy to play this game. Then another approach of co-simulation is used to test this policy against an agent doing random choices between the dice.

First the model of the game was decided to be exported from Simulink using a shared library of C-code because that was found to be the fastest approach and this game is completely discrete so that did not limit the choice. The models architecture can be seen in Picture 2 below.



Picture 2: The Simulink architecture of the dice game.

The subsystem “roll and recalculate” rolls a random side of the die that is indicated by the input choice. The input “target” indicates how far away the player is from zero and the result of the die roll is deducted from this and the result’s absolute value is taken which is the output of the subsystem named “new target”. The output of the system named “Observation” indicates that value and the output named “IsDone” indicates if the game has finished. The final output named “Reward” indicates the penalty endured for each throw it takes for the agent to finish the game.

4.2 Co-simulation integration and reinforcement learning algorithm

To integrate the model into Python the same procedure was followed that was used in Chapter 3.3. The presence of multiple outputs does not complicate the integration process it just means all the additional outputs need to be declared in the Python code just like they are declared in the generated code. In addition, the calling conventions of the functions that interact with the model should be checked so that all the arguments are correct as either values, references, or pointers.

Once the model was integrated into Python the reinforcement learning algorithm needed to be implemented. In this paper, an Epsilon greedy Monte Carlo method was used adapted from Sezer [12]. This method starts with determining a random policy then it plays the game according to that policy and records the returns received from doing an action from a specific state of the game. The method also sometimes deviates from the policy to promote more exploration and the frequency of this is determined by a probability set by the user. In this example, the agent follows the policy with a 92.5%

probability. This means the agent is greedy as mentioned in the name. A greedy agent will favour the fastest possible solution instead of exploration. An agent that favours more exploration might achieve more accurate results, however it needs more repetitions to reach those results which could prove unnecessarily time consuming. Once the algorithm has played one game and recorded the results and penalties endured, it will calculate quality values (Q-values) for each action from each state. Then a new policy is determined based on the Q-values. The action that has the highest Q-value from a specific state is chosen as the action in the new policy. Finally, the process is repeated a large number of times to achieve a more reliable policy.

In this example, ten thousand repetitions were used in the training of the model. From this the final policy was extracted and it can be seen in Table 10 below.

Table 10: Policy determined by the reinforcement learning agent.

Policy	
State	**Choice**
20	3
19	5
18	5
17	5
16	3
15	5
14	3
13	5
12	5
11	3
10	4
9	5
8	4
7	3
6	2
5	1
4	2
3	1
2	1
1	1

In this table, state refers to how far away the player is from zero and choice refers to the number of the die the agent has decided to throw at that state. The policy was also saved to a text file for further usage.

4.3 Testing using TCP/IP connection

Next this policy was tested in terms of if it produced a result that would be more effective than choosing at random between the dice. The TCP/IP connection was chosen to showcase its utility. The connection was established as explained in Chapter 3.4. For the usage of this testing only one value at a time would need to be send. So, the rolling of the dice was done in Simulink and the target was updated. This target was then sent to Python where the policy was read from the text file mentioned above and a choice was made according to the policy and then sent back to Simulink.

The testing was done in a way where the initial condition was randomized between one and twenty. Then the game was played until the end and the amount of throws it took to finish the game was recorded. The results of the testing can be seen below in Table 11.

Table 11: Comparison of using the reinforcement learning policy to a random policy.

	Iterations	
Simulation	**Policy**	**Random**
Run 1	1	2
Run 2	6	6
Run 3	3	7
Run 4	3	4
Run 5	2	7
Run 6	4	4
Run 7	5	2
Run 8	6	7
Run 9	1	3
Run 10	5	5
Mean	**3,6**	**4,7**

From the table, it can be seen that for ten rounds of the game using the policy is on average one throw more efficient than using a random policy to play the game. The difference is quite modest which might be because there is no correlation between the initial conditions of the games played on each row of the table. A more drastic difference could be seen if for each round the two policies would start playing the game from the same initial condition. However, even with the completely random initial conditions the reinforcement learning agent's policy is more efficient which matches the expected result.

This use-case demonstrated how co-simulation with Python and Simulink could be used to train a reinforcement learning agent to perform a task more efficiently. In terms of the dice game the training was successful, and the agent was more efficient than an agent making choices at random. The approach used in the training of the agent is currently only applicable to discrete systems. However, for continuous systems, the MATLAB Engine could be utilized similarly to train the agent. This would take more time but that is a necessary cost that needs to be endured. This example highlights the usefulness of co-simulation because it allows for the usage of the benefits of both platforms as well as legacy models in Simulink without major changes or approximations. It also showed that different approaches to co-simulation can be used to perform different tasks within a project.

5. CONCLUSIONS

This thesis has studied co-simulation using Python and Simulink. The objectives were to find different approaches to conduct the co-simulation and outline their benefits and limitations. In addition, a use case was defined to demonstrate the usefulness of co-simulation in a concrete context.

Firstly, a literature review was conducted to get a sense of the different approaches to co-simulation with the given platforms. Furthermore, the literature review showed the need for this sort of research as prior research on the topic is very scarce. Based on the literature review four different approaches were chosen to be presented in this paper using Python code in Simulink, using the MATLAB Engine, using a shared library of generated C-code, and using a TCP/IP connection.

To explore the ability to use Python code in Simulink a simple Pi-controller was implemented in Python to be used to control a Simulink model. This approach presents the limitations that Python objects cannot be used as any input or output in Simulink. Thus, the simulation was conducted using a MATLAB script and it was successful. Another inconvenience of the approach is that for every session the user needs to redefine the location of their Python module into the search path of the Python environment in MATLAB.

Two different approaches to using Simulink models from Python were presented. First of these is the MATLAB Engine which could be used to run a simulation through two different methods. These two methods were benchmarked, and it was found that one method is more suitable for simpler and shorter simulations and the other is more suitable for more complex and longer simulations. The second approach to using Simulink in Python is using a shared library of generated C-code. This involved generating a dynamic library from a Simulink model according to specific settings so that the simulation could be accessed from Python. Then the “Ctypes”-library was used to implement the dynamic library in the Python code for co-simulation. The output of this approach was compared to the output of simulating an identical model in Simulink by itself and the results were very close to each other proving that the co-simulation suffered minimal accuracy losses. The limitation of this approach is that it can only be used for discrete models.

Table 12 summarizes the benefits and limitations of the different methods explored.

Table 12: Summary of benefits and limitations of each method

Method	Benefits	Limitations
Python in Simulink	- Easy to use benefits of both platforms - No speed loss	- Moderately slow - Requires meticulous setup
MATLAB Engine API	- Easy to use benefits of both platforms - Easy to setup and run simulations - Intuitive interfacing	- Slowest - Complex system require high processing power
Simulink to C to Python	- Fastest - After C extraction an active MATLAB license is not required	- Only for discrete systems - Requires more work to setup
TCP/IP	- Fast - Works for all types of systems - Moderately easy to setup - Best for parallel processing	- Sending complex data might be difficult - Requires some knowledge on how local networks are hosted and how the data is transported

Finally, the use-case used to present some of the possible capabilities of co-simulation was defined as a dice game. In the game, the player is given a random number between one and twenty and their goal is to reach zero in as few throws as possible. This model was exported using the embedded coder as a shared library of generated code. Then the exported model was used to train a reinforcement learning agent to find an ideal policy for playing the game. After training the agent a TCP/IP connection was used to co-simulate its effectiveness against an agent acting randomly, it was found that the co-simulation was successful in training the agent to be more efficient than completely random choices.

In conclusion, it was found that there are multiple approaches to co-simulation using Simulink and Python that are each useful in different scenarios. Especially useful was the finding that a co-simulation could be used to unite the ease of modeling in Simulink and the vast machine-learning capabilities of Python. Further research could be done into system identification and optimization through co-simulation as well.

REFERENCES

[1] MathWorks, “Simulink front page,” MathWorks website. Accessed: Dec. 15, 2023. [Online]. Available: https://se.mathworks.com/products/simulink.html

[2] MathWorks, “Access Python Modules from MATLAB - Getting Started,” MathWorks Documentation. Accessed: Oct. 14, 2023. [Online]. Available: https://se.mathworks.com/help/matlab/matlab_external/create-object-from-python-class.html

[3] MathWorks, “Get Started with MATLAB Engine API for Python,” MathWorks Documentation. Accessed: Oct. 15, 2023. [Online]. Available: https://se.mathworks.com/help/matlab/matlab_external/call-matlab-functions-from-python.html

[4] L. Haider, M. Baumgartner, D. Hayn, and G. Schreier, “Integration of Python Modules in a MATLAB-Based Predictive Analytics Toolset for Healthcare,” in *dHealth 2022: Proceedings of the 16th Health Informatics Meets Digital Conference*, G. Schreier, B. Pfeifer, M. Baumgartner, and D. Hayn, Eds., IOS Press BV, 2022, pp. 197–204. doi: 10.3233/SHTI220369.

[5] MathWorks, “Import Python Code to Simulink Using Python Importer Wizard,” MathWorks Documentation. Accessed: Dec. 01, 2023. [Online]. Available: https://se.mathworks.com/help/simulink/ug/import-python-code-using-python-importer.html

[6] MathWorks, “Embedded coder,” MathWorks Documentation. Accessed: Nov. 27, 2023. [Online]. Available: https://se.mathworks.com/help/ecoder/index.html?s_tid=CRUX_lftnav

[7] MathWorks, “Simulink compiler,” MathWorks Documentation. Accessed: Oct. 18, 2023. [Online]. Available: https://se.mathworks.com/help/slcompiler/index.html?s_tid=CRUX_lftnav

[8] MathWorks, “Using Python with MATLAB,” MathWorks Website. Accessed: Oct. 17, 2023. [Online]. Available: https://se.mathworks.com/products/matlab/matlab-and-python.html

[9] M. Keshavarz and A. Mojra, “Geometrical features assessment of liver’s tumor with application of artificial neural network evolved by imperialist competitive algorithm,” *Int J Numer Method Biomed Eng*, vol. 31, no. 5, pp. e02704-n/a, 2015, doi: 10.1002/cnm.2704.

[10] C. T. Tshiani and P. Umenne, “The Characterization of the Electric Double-Layer Capacitor (EDLC) Using Python/MATLAB/Simulink (PMS)-Hybrid Model,” *Energies (Basel)*, vol. 15, no. 14, p. 5193, 2022, doi: 10.3390/en15145193.

[11] A. Mehlhase, “A Python framework to create and simulate models with variable structure in common simulation environments,” *Math Comput Model Dyn Syst*, vol. 20, no. 6, pp. 566–583, 2014, doi: 10.1080/13873954.2013.861854.

[12] Ö. Sezer, “Monte Carlo Epsilon greedy demo,” GitHub. Accessed: Dec. 10, 2023. [Online]. Available: https://github.com/omerbsezer/Reinforcement_learning_tutorial_with_demo/blob/master/monte_carlo_epsilon_greedy_demo.ipynb