Enhancing PowerFactory Dynamic Models with Python for Rapid Prototyping

Claudio David López, Miloš Cvetković and Peter Palensky
Department of Electrical Sustainable Energy
Delft University of Technology
The Netherlands

***Abstract*—DIgSILENT PowerFactory is among the most widely adopted power system analysis tools in research and industry. It provides a comprehensive library of device models and it allows users to define their own. Models for dynamic simulation can be defined in the DIgSILENT Simulation Language (DSL). When the functionality of DSL is insufficient, new DSL functions can be defined in C or C++. However, C and C++ can be challenging for inexperienced programmers. Furthermore, every time the C or C++ code is modified, it needs to be recompiled and PowerFactory needs to be restarted for the changes to take effect, which slows down the workflow, model development, and inhibits rapid prototyping. In this paper we present an open source library that allows users to call Python functions and methods from DSL with minimal effort. Python is a powerful and much easier to use language than C or C++. Additionally, Python programs do not need to be compiled. Furthermore, with this library PowerFactory does not need to be restarted every time the Python code is changed. To illustrate what can be accomplished with our library we present three example use cases related to load modeling, co-simulation, and fault detection based on machine learning. The examples show that it becomes straightforward to enhance DSL with Python and that sophisticated models can be produced with reduced effort using popular open source Python libraries. As a consequence, PowerFactory users gain access to enhanced modeling capabilities and user-friendliness, and a more speedy workflow, which is beneficial for rapid prototyping.**

***Index Terms*—Co-simulation, DSL, dynamic simulation, machine-learning, PowerFactory, Python**

I. Introduction

DIgSILENT PowerFactory is among the most widely adopted power system analysis tools in research and industry. It is known for its versatility, since it allows users to perform a wide range of static and dynamic analyses on power system models, and provides a comprehensive library of electrical power system device models that users can customize and expand with new models. In the case of dynamic simulations (i.e., transient stability and electromagnetic transient simulation), users can define new models in the DIgSILENT Simulation Language (DSL).

DSL is a language for modeling continuous linear and non-linear systems that is well suited for defining control structures. When the functionality of DSL is insufficient for a given application, new DSL functions can be defined by the user. These functions must be specified in either the C or C++ programming language and compiled into a Dynamic Link Library (DLL) that PowerFactory can access [1]. This is a powerful feature, as it creates the opportunity for potentially unlimited expansion of the simulation capabilities of PowerFactory. Additionally, this feature is also useful when implementing a model in DSL becomes too cumbersome, for example, when `if`/`else`, `while` and/or `for` statements are required, none of which are provided in DSL.

However, creating new DSL models in C or C++ poses some challenges. C is a low-level programming language by today's standards, so considerable programming effort is needed to implement sophisticated functionality. Furthermore, some of its features have proven troublesome to inexperienced programmers, such as direct memory addressing and dynamic memory management. On the other hand, C++ is a higher level language than C, but with a steep learning curve, and most electrical power engineers are not well versed in it. Using these languages can also slow down the simulation development workflow; Every time the C or C++ code is modified, the DLL must be recompiled and PowerFactory must be restarted for the changes to take effect. Altogether, using C or C++ for creating DSL models is inconvenient when the code is modified often and/or when the main goal is implementation with minimal effort, which is the case during model development and rapid prototyping.

Python in comparison to C and C++ is easier to learn, read, write and debug. It is a high-level language with an extensive standard library. In addition, Python's popularity within the scientific and engineering communities has materialized in a vast collection of reliable and user-friendly open source libraries, like SciPy [2] for scientific computing, scikit-learn [3] for machine learning, and pandas [4] for manipulation and analysis of large datasets. These characteristics make it appealing when the priority is to minimize programming effort. Since version 15.1 PowerFactory provides a Python 3 API [5], however, this API cannot be invoked during a dynamic simulation, much less from a DSL model [1], as it is mainly intended for automating simulation tasks.

In this paper we present a small open source library[1] that addresses these challenges by allowing PowerFactory users to easily call Python functions and methods from their DSL models. Our library has the added benefit of a more speedy simulation development workflow. Unlike C and C++, Python is an interpreted scripting language that does not need to be compiled. Moreover, the library is designed so PowerFactory

[1] https://github.com/claudiodavidlopez/digexfunPyDSL

```python
import numpy as np
from external_script import external_function


def square_to_polar(re, im):
    """Converts a phasor from square to polar
    coordinates.
    """
    mag = float(np.sqrt(re**2 + im**2))
    ang = float(np.arctan2(im, re))
    return mag, ang


CALLABLE_REGISTRY = [
    external_function, # pyFunID = 0
    square_to_polar    # pyFunID = 1
]
```

Listing 1. Example of a valid script.py file.

```
! Block inputs: re, im
! Block outputs: mag, ang

pyFunID = 1 ! From CALLABLE_REGISTRY

! ----------------------------------
! Executed only during initialization
! ----------------------------------
! square_to_polar takes 2 arguments
inc(dummy0) = LoadPyFun(2, pyFunID)

! ----------------------------------
! Executed every time step
! ----------------------------------
! Set arguments
dummy1 = SetPyFunArg(re, 0, pyFunID)
dummy2 = SetPyFunArg(im, 1, pyFunID)

! Call square_to_polar
dummy3 = CallPyFun(pyFunID)

! Get returned values
mag = GetPyFunRetVal(0, pyFunID)
ang = GetPyFunRetVal(1, pyFunID)
```

Listing 2. DSL code for calling the Python function `square_to_polar`.

does not need to be restarted when the Python code is modified. All of these characteristics make the library valuable for rapid prototyping.

To illustrate what can be accomplished with our library we present three example use cases. In the first one we implement a load model in Python and compare it to its DSL implementation to highlight how some models are easier to implement in Python, in the second one we present a co-simulation interface that shows how the library can be used to couple PowerFactory with external simulators and models, and in the third one we present a machine learning-based fault detector that emphasizes how this library can be used to incorporate models that are beyond classic power engineering. Together, these examples show that it becomes straightforward to enhance DSL with Python and that sophisticated models can be produced with reduced effort using popular Python libraries. As a consequence, PowerFactory users gain access to enhanced modeling capabilities and user-friendliness, and a more speedy workflow, which is beneficial for rapid prototyping.

This paper is structured as follows: Section II presents the library design, Section III describes some key aspects of the library implementation, Section IV introduces the example use cases, and Section V concludes the paper.

II. Library Design

In this section we describe the main aspects of the library design, namely the requirements it fulfills, the assumptions that it makes to find and access Python code, and the functions it provides to the user.

A. Requirements

The library fulfills the following requirements:

- Multiple Python functions and/or methods may be called from a DSL model.
- Each Python function and/or method can take a different number of arguments.
- Each Python function and/or method can return a different number of values.
- Each Python function and/or method takes only floating point arguments.
- Each Python function and/or method returns only floating point values.
- Modifications to the Python code take effect on the next simulation run, without restarting PowerFactory.

B. Assumptions

The library assumes that the Python functions and methods that are to be called from DSL are referenced in a Python list called `CALLABLE_REGISTRY`, which should be defined in a file called script.py located in PowerFactory's installation folder. Since only references to the functions and methods are expected in `CALLABLE_REGISTRY`, the functions and methods themselves may be defined either in script.py or in an external Python script.

Listing 1 shows an example of a valid script.py. This script defines one function and it imports an external function from and external Python script. Both functions are then referenced in `CALLABLE_REGISTRY` so they can be called from DSL. When called from DSL, these functions are identified by their `pyFunID`, which is their index in `CALLABLE_REGISTRY`.

C. DSL Application Programming Interface

The Application Programming Interface (API) is composed of four DSL functions. Together they make it possible to call the Python functions and methods in `CALLABLE_REGISTRY`. These four functions are defined as follows:

```python
class ElectronicLoadWECC:
    def __init__(self, Vd1, Vd2, frcel,
                 Pel0, Qel0, Vmin0=1.0):
        self.Vd1 = Vd1
        self.Vd2 = Vd2
        self.frcel = frcel
        self.Pel0 = Pel0
        self.Qel0 = Qel0
        self.Vmin = Vmin0

    def calc_pq(self, V):
        if V < self.Vmin:
            self.Vmin = V

        if self.Vmin < self.Vd2:
            self.Vmin = self.Vd2

        if V < self.Vd2:
            Fvl = 0.0
        elif V < self.Vd1:
            if V <= self.Vmin:
                Fvl = (V - self.Vd2)/(self.Vd1
                      - self.Vd2)
            else:
                Fvl = (self.Vmin - self.Vd2
                      + self.frcel*(V
                      - self.Vmin))/(self.Vd1
                      - self.Vd2)
        else:
            if self.Vmin >= self.Vd1:
                Fvl = 1.0
            else:
                Fvl = (self.Vmin - self.Vd2
                      + self.frcel*(self.Vd1
                      - self.Vmin))/(self.Vd1
                      - self.Vd2)

        return Fvl*self.Pel0, Fvl*self.Qel0


eload = ElectronicLoadWECC(0.866, 0.7, 0.7,
                           20, 10)


CALLABLE_REGISTRY = [
    eload.calc_pq # pyFunID = 0
]
```

Listing 3. Python implementetion of the WECC electronic load model.

```
! Block input: V
! Block outputs: Pel, Qel

pyFunID = 0 ! From CALLABLE_REGISTRY

! Load eload.calc_pq
inc(dummy0) = LoadPyFun(1, pyFunID)

! Call eload.calc_pq
dummy1 = SetPyFunArg(V, 0, pyFunID)
dummy2 = CallPyFun(pyFunID)
P = GetPyFunRetVal(0, pyFunID)
Q = GetPyFunRetVal(1, pyFunID)
```

Listing 4. DSL code that calls `eload.calc_pq`.

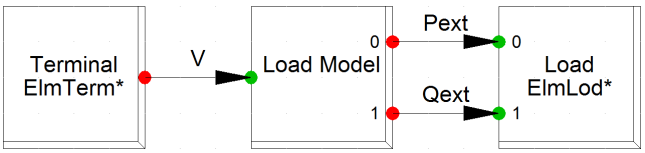

Fig. 1. PowerFactory Composite Frame of the WECC electronic load model, where V is the voltage magnitude measured at the load terminals, and Pext and Qext are the active and reactive power consumption.

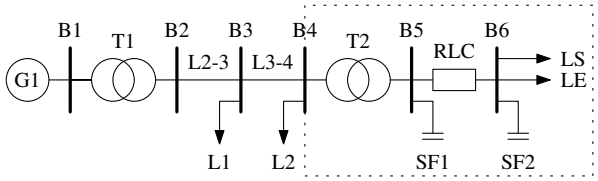

Fig. 2. Test grid from [6]. The WECC Composite load model is enclosed in the dashed rectangle. The motor loads were removed. The electronic load is marked as LE. The loads L1, L2 and LS, and the filters SF1 and SF2 are static.

`LoadPyFun(argNum, pyFunID)`
Loads a Python function into PowerFactoy, where

- `argNum` is the number of expected arguments, and
- `pyFunID` specifies which function.

`SetPyFunArg(argVal, argID, pyFunID)`
Sets the value of one of the arguments of a Python function, where

- `argVal` is the value of the argument,
- `argID` specifies which argument, and
- `pyFunID` specifies which function.

`CallPyFun(pyFunID)`
Calls the Python function specified by `pyFunID`.

`GetPyFunRetVal(retValID, pyFunID)`
Gets one of the values returned by a Python function, where

- `retValID` specifies which argument, and
- `pyFunID` specifies which function.

The reason why we require three different DSL functions each time we call a Python function or method is that DSL functions cannot take a variable number of arguments and return only one value. Since the library needs to accommodate Python functions and methods that take different numbers of arguments and return different numbers of values, setting arguments and retrieving returned values one by one circumvents this limitation.

Listing 2 shows how this API can be used to call the function `square_to_polar` defined in Listing 1. This DSL listing must be placed inside a PowerFactory Block Definition whose inputs are `re` and `im`, and whose outputs are `mag` and `ang`. Note that the only API function whose returned value matters is `GetPyFunRetVal`.

III. Library Implementation

To implement the library we relied on the facts that new DSL functions can be defined in C++ [1] and that Python

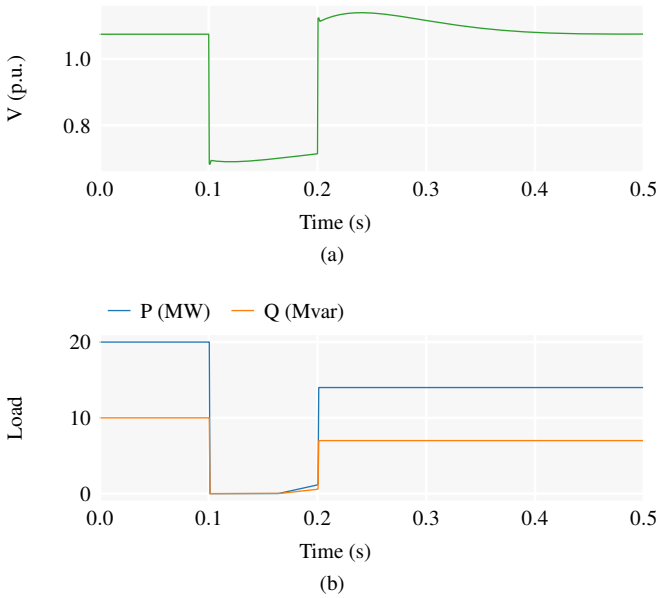

Fig. 3. Response of the electronic load to voltage variations resulting from a high-impedance fault. (a) Terminal voltage input. (b) Power consumption output.

```
Fvl = select(V < Vd2, 0.0, select(V < Vd1,
    select(V <= Vmin, (V - Vd2)/(Vd1 - Vd2),
    (Vmin - Vd2 + frcel*(V - Vmin))/(Vd1 -
    Vd2)), select(Vmin > Vd1, 1.0, (Vmin -
    Vd2 + frcel*(Vd1 - Vmin))/(Vd1 - Vd2))))
```

Listing 5. DSL code that calculates `Fvl`.

provides an API for embedding Python code in C++ [7]. Thus, it is possible to create an interface between DSL and Python in C++. The role of this C++ interface is twofold: the front end provides the API defined in Section II-C, while the back end is in charge of loading script.py and manipulating the functions and methods in `CALLABLE_REGISTRY`, at the request of the API. This C++ interface must be compiled as a DLL, given a name that starts with the digexfun prefix (e.g., digexfunPyDSL.dll), and placed in PowerFactory's installation folder. This is because PowerFactory loads at start-up all DLLs located in its installation folder that have the digexfun prefix in their names [1]. To ensure that up-to-date Python code is always executed without having to restart PowerFactory to force the DLL to be reloaded, the library detects when a new simulation is initialized and reloads script.py.

IV. Example Use Cases

The possible applications of Python-enhanced DSL models are potentially endless. To illustrate some of the possibilities we present three simple use cases that we hope can provide the reader a starting point from which to tackle more complex problems. Although we omit some details for the sake of brevity, these examples are available in full in the library repository.[1]

```
import zmq


class ElectronicLoadWECC:
    ⋮


eload = ElectronicLoadWECC(0.866, 0.7, 0.7,
                           20, 10)

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:7000")

while True:
    sim_ins = socket.recv_json()
    V = sim_ins[0]
    P, Q = eload.calc_pq(V)
    sim_outs = [P, Q]
    socket.send_json(sim_outs)
```

Listing 6. External electronic load script for co-simulation (eload.py).

A. WECC Electronic Load Model

The WECC Composite Load Model, developed by the Western Electricity Coordinating Council (WECC), represents the dynamic characteristics of end-use loads [8]. One of the components of the WECC Composite Load Model is the electronic load model. This model relates active and reactive power consumption to the voltage magnitude at the load terminals.

The `ElectronicLoadWECC` in Listing 3 is a Python implementation of the electronic load model. In this class, the `calc_pq` method establishes the relationship between terminal voltage and power consumption, and the `__init__` method initializes the model parameters and the `Vmin` variable. This variable is defined as a member variable because its value must be remembered between calls to the `calc_pq` method. Below the class definition is the class instantiation, that creates the `eload` object with the desired set of model parameters. Once `eload` has been created, the `eload.calc_pq` method is added to `CALLABLE_REGISTRY` so it can be called from DSL.

Listing 4 shows the DSL code that calls `eload.calc_pq`. This code is embedded in the Load Model block inside of the PowerFactory Composite Frame from Fig. 1. In the Composite Model, the Terminal block represents a bus in the grid model and it provides the `V` argument to `eload.calc_pq`. In turn, the Load Model block provides the active and reactive power returned by `eload.calc_pq` to the Load block, which represents a load in the grid model.

To test the Python implementation of the electronic load model we used the grid from Fig. 2 as developed in [6], but removed the motor loads from the WECC Composite Load Model and replaced the DSL electronic load with our Python implementation. Fig. 3 shows the response of the electronic load to the voltage variations that result from a high-impedance

fault at bus B3.

Even though the electronic load model was implemented in DSL in [6], the Python implementation is easier to create and understand. This becomes apparent when comparing the DSL code used for calculating the `Fvl` variable, shown in Listing 5, to the Python code from Listing 3. While the Python code is a series of `if`/`else` statements, the DSL code is several nested `select` functions. The advantage of using Python in this particular case becomes even clearer when comparing the Python implementation of the electronic load model to the pseudocode used to define it in [8]; translating the pseudocode into Python is almost trivial.

B. Co-Simulation Interface

In a co-simulation two or more simulators simulate cooperatively by exchanging variables at run time. These simulators may run on the same computer or separate computers. To exemplify how this can be accomplished with our library, we removed the electronic load model from the previous example, and placed it in the eload.py script from Listing 6, that can be executed independently from PowerFactory. The objective now is to create an interface to couple the grid model from Fig. 2 to the electronic load model in eload.py. This means that at every simulation time step PowerFactory must send the voltage at bus B6 to eload.py, and eload.py must reply with its active and reactive power consumption. Listing 7 presents the co-simulation interface that couples PowerFactory and eload.py. The communication is implemented with PyZMQ, which provides a Python API to the ØMQ messaging library [9], and JSON-encoded messages. The interface is tasked with sending outputs from PowerFactory to eload.py, and receiving inputs from eload.py. Note that the `cs_int.exchange_variables` method can take a variable number of arguments, so it can be used to exchange as many variables as needed, provided the number is specified in the `argNum` argument to the `LoadPyFun` function. Using PyZMQ much more sophisticated co-simulations settings can be implemented, such as those in [10].

C. Machine Learning for Fault Detection

DSL on its own is certainly inadequate for simulations that require machine learning functionality, but Python is a common choice for these applications. In this example we take a naive approach to fault detection using Linear Discriminant Analysis (LDA) for classification [11]. The objective is to train LDA to detect when and where in the grid a fault happens, and to use this information to clear the fault. We assume that faults occur only at buses and that they can have an impedance between 0 and 40 Ω. Thus, LDA must classify a set of measurements obtained from the grid as characteristic of normal operation or a fault at a specific bus.

To add more variance to the operating conditions we modified the grid from Fig. 2 so all loads are random, except for the electronic load. The random loads vary their power consumption within ±3% of the nominal value following a uniform distribution, and are implemented in Python.

```python
import zmq


ADDRESS = "tcp://localhost:7000"


class CosimInterface:
    def __init__(self, address):
        self.address = address
        self.context = None
        self.socket = None

    def _open_connection(self):
        self.context = zmq.Context()
        self.socket = self.context.socket(
            zmq.REQ)
        self.socket.connect(self.address)

    def _close_connection(self):
        self.socket.close()
        self.context.term()

    def exchange_variables(self, *sim_outs):
        self._open_connection()
        self.socket.send_json(sim_outs)
        sim_ins = self.socket.recv_json()
        self._close_connection()
        return sim_ins


cs_int = CosimInterface(ADDRESS)


CALLABLE_REGISTRY = [
    cs_int.exchange_variables
]
```

Listing 7. Simple co-simulation interface for PowerFactory.

Using this grid we created a training dataset by applying faults to every bus, each time varying the fault impedance between 0 and 40 Ω in steps of 10 Ω, and measuring the active and reactive power flowing through every branch. This produced a dataset where the samples are the results of each time step and the features are the power flows. To train LDA we labeled each sample in the dataset as normal (label $= -2$), post fault (label $= -1$) or fault, in which case the label is the number of the bus where the fault is happening.

Listing 8 shows the Python implementation of the random load (`random_load` function) and the LDA-based fault detector (`FaultDetector` class). The fault detector relies on the LDA implementation the scikit-learn library provides [3]. We train LDA in the `__init__` method and detect faults with the `detect` method, which takes the active and reactive power measurements from the grid and returns six flags (one per bus) that are set when a fault in the corresponding bus occurs.

Fig. 4 shows the inputs (branch power flows) and outputs (flags) of the LDA-based fault detector during a simulation where a fault with an impedance of 33 Ω is applied to bus B3 at 0.1 s and a fault with an impedance of 13 Ω is applied to

```
import random
import pandas as pd
import numpy as np
import sklearn.discriminant_analysis as skda


LDA = skda.LinearDiscriminantAnalysis


class ElectronicLoadWECC:
    ⋮


def random_load(Pnom, Qnom, max_dev):
    p_dev = random.uniform(-max_dev, max_dev)
    q_dev = random.uniform(-max_dev, max_dev)
    P = (1 + p_dev)*Pnom
    Q = (1 + q_dev)*Qnom
    return P, Q


class FaultDetector:
    def __init__(self, n_buses):
        self.n_buses = n_buses
        data, labels = load_training_data()
        self.lda = LDA()
        self.lda = self.lda.fit(data, labels)

    def detect(self, *args):
        fault_flags = [0.0]*self.n_buses
        label = int(self.lda.predict([args]))
        if label > 0:
            fault_flags[label-1] = 1.0

        return fault_flags


fdetector = FaultDetector(n_buses=6)


CALLABLE_REGISTRY = [
    eload.calc_pq,     # pyFunID = 0
    random_load,       # pyFunID = 1
    random_load,       # pyFunID = 2
    random_load,       # pyFunID = 3
    fdetector.detect,  # pyFunID = 4
]
```

Listing 8. Extended script.py with the implementation of the random load and the LDA-based fault detector.

bus B5 at 0.3 s. Both faults are cleared with a delay of 0.1 s. As Fig. 4 (b) shows, the flag that corresponds to the bus where the fault occurs is set as soon as the fault starts, and is reset once the fault is cleared.

D. Caveats

Note that in Listing 8 we register the `random_load` function three times in `CALL_REGISTRY`, once for each random load in the test system. This means that we can access it with three different `pyFunIDs`. We avoid calling the same function or method multiple times with different arguments

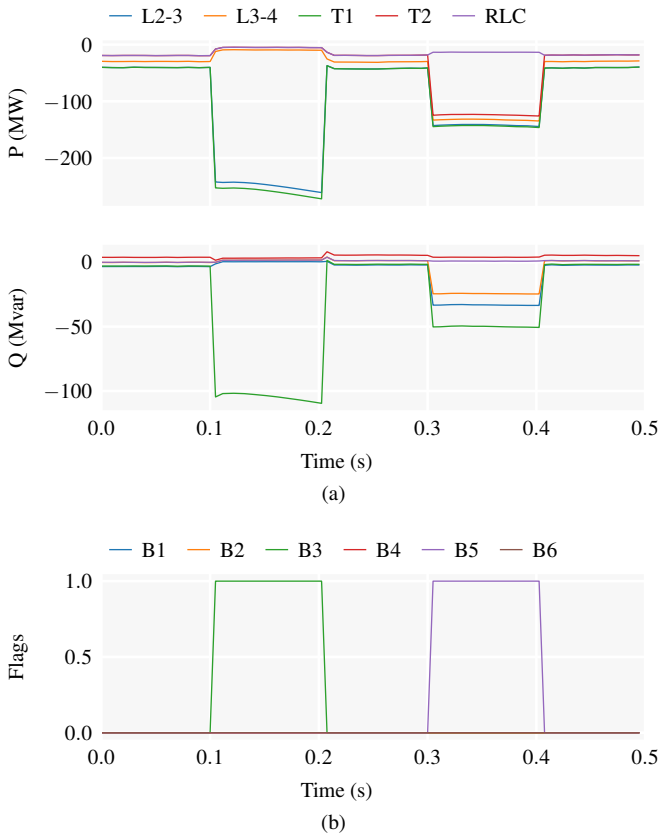

Fig. 4. Inputs and outputs of the LDA-based fault detector during faults at bus B3 and B5. (a) Input branch power flows. (b) Output fault flags.

using the same `pyFunID` to prevent inadvertently overriding these arguments.

During a dynamic simulation, PowerFactory might call the DSL functions, and therefore the Python functions or methods, more than once per time step. In the previous examples we do not deal with this problem, but we advise the reader to consider this behavior while designing a Python function or class for use with PowerFactory.

V. Conclusion

This paper presented a small open source library for enhancing DSL models with Python, and three example use cases related to load modeling, co-simulation, and fault detection based on machine learning. The example use cases show that with our library it is simple to embed Python code in DSL models, and that with the help of popular open source Python libraries it is possible to easily create sophisticated models that are beyond the boundaries of traditional power engineering. The library is especially well suited for situations where the model code is modified often and/or when the main goal is implementation with minimal effort. As a consequence, PowerFactory users gain access to enhanced modeling capabilities and user-friendliness, and a more speedy workflow, which is beneficial for model development and rapid prototyping.

REFERENCES

[1] M. Stifter, F. Andrén, R. Schwalbe, and W. Tremmel, "Interfacing PowerFactory: Co-simulation, real-time simulation and controller hardware-in-the-loop applications," in *PowerFactory Applications for Power System Analysis*, F. M. Gonzalez-Longatt and J. L. Rueda, Eds. Springer International Publishing, 2014, pp. 343–366.

[2] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, May 2007.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[4] W. McKinney, "Data structures for statistical computing in Python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 51–56.

[5] C. D. López and J. L. Rueda-Torres, "Python scripting for DIgSILENT PowerFactory: Leveraging the Python API for scenario manipulation and analysis of large datasets," in *Advanced Smart Grid Functionalities Based on PowerFactory*, F. M. Gonzalez-Longattand and J. L. Rueda-Torres, Eds. Springer, 2018, pp. 19–48.

[6] A. Joseph, M. Cvetković, and P. Palensky, "Predictive mitigation of short term voltage instability using a faster than real-time digital replica," in *Proceedings of the 2018 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, Oct. 2018.

[7] Python Software Foundation. Python/C API Reference Manual. [Online]. Available: docs.python.org/3/c-api/index.html

[8] Western Electricity Coordinating Council, "WECC dynamic composite load model (CMPLDW) specifications," WECC, Tech. Rep., Jan. 2015.

[9] iMatix. ØMQ. [Online]. Available: zeromq.org

[10] C. D. López, M. Cvetković, and P. Palensky, "Distributed co-simulation for collaborative analysis of power system dynamic behavior," in *Proceedings of the MEDPOWER 2018 Conference*, Nov. 2018.

[11] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, 2nd ed. Springer, 2001.