PowerFactory and Python Scripting for Evaluation of Active Distribution Networks Impact on Grid Short-term Voltage Stability

Aleksandar Boričić, Jose Luis Rueda Torres, Marjan Popov
Delft University of Technology, Faculty of EEMCS, Delft, the Netherlands
a.boricic@tudelft.nl, J.L.RuedaTorres@tudelft.nl, M.Popov@tudelft.nl

Abstract

This chapter deals with improving the understanding of the driving parameters of short-term voltage stability in modern power systems. The approach utilizes DIgSILENT PowerFactory 2020 SP2A paired with Python API, enabling the evaluation of a complex multivariable problem efficiently by running a large number of dynamic simulations automatically. The presented approach is not only limited to short-term voltage stability, but can be also utilized for various large-scale dynamic studies. As systems shift towards more complexity in both generation and consumption, similar studies shall become indispensable in the power systems field.

Keywords: DIgSILENT Dynamic Simulation, Python Scripting, Short-Term Voltage Stability, Distributed Energy Resources, Dynamic loads

1.1 Introduction

The number of distributed energy resources (DER) is increasing rapidly in power systems worldwide. Coincidentally, the number and complexity of dynamic loads are also growing. This leads to increasingly intricate and relevant impacts of Active Distribution Networks (ADN) on the overall system dynamics and stability. Furthermore, as synchronous generation is phased out, system strength and inertia are decreased, and system dynamics are enhanced and accelerated. Short-term voltage stability consequently becomes one of the major issues that renewables-driven power systems are facing. It is therefore of the uttermost importance to be able to simulate and analyse grid stability comprehensively, even with a large number of components and possible control parameters that define their operation. However, this is a complex multivariable challenge that requires an innovative approach. Utilizing advanced simulation software such as DIgSILENT PowerFactory is necessary and valuable [1], but often not sufficient. There is also a need for a programming interface to automate simulations for a large number of potential operational scenarios and parameters. As power systems digitalize and move towards control-driven dynamics, rather than electromechanical, the ability to perform dynamic simulations with a wide range of parameters efficiently is vital. This chapter describes a fundamental study on short-term voltage stability, utilizing some of the most advanced voltage stability models in the process. The dynamic analysis is automatized with Python, enabling the efficient execution of thousands of simulations with varying operating conditions and relevant parameters. The original analysis is performed in [2], where further technical details can be found.

The chapter is organized as follows. Section 1.2 briefly describes the problematics of short-term voltage stability and the effects of active distribution networks and their parameters. Section 1.3 introduces the test models. In Section 1.4, the utilized Python scripting and its benefits are demonstrated, with detailed codes and relevant simulation results. Finally, Section 1.5 concludes the chapter.

1.2 Power Systems Short-term Voltage Stability

Systems are as strong as their weakest link. With the decentralization of power systems generation, as well as intricate technical challenges brought by inverter-based resources (IBRs) and dynamic loads, finding the weakest link and its interactions with other elements in the system becomes exceptionally difficult. Meanwhile, preventing power system instability and its consequences such as blackouts is crucial in the interconnected and electricity-powered society. To prevent it, one must first understand it and be able to model it accurately. Power system stability is commonly defined as follows [3]:

Power system stability is the ability of an electric power system, for a given initial operating condition, to regain a state of operating equilibrium after being subjected to a physical disturbance, with most system variables bounded so that practically the entire system remains intact.

It comprises several different classes. The most recent commonly accepted definition and classification of stability were established in [3]. This classification is presented in Figure 1.

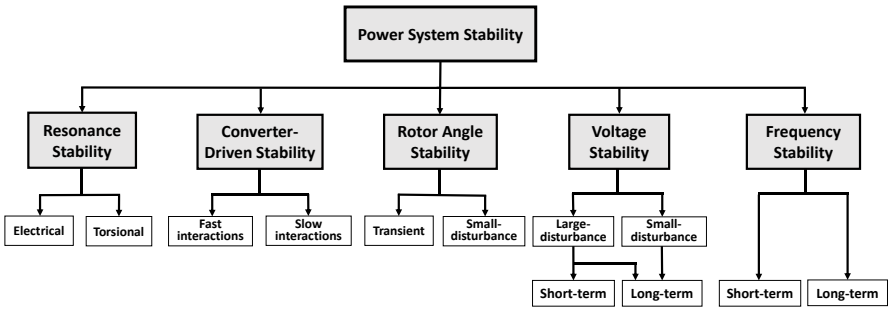


Figure 1: Classification of power system stability [3]

This chapter focuses on short-term stability, particularly short-term voltage stability (STVS). Furthermore, a holistic perspective of short-term instabilities is taken, where each instability mechanism is described in Figure 2. A more detailed discussion on this can be found in [2, 4] and is omitted from this chapter for brevity.

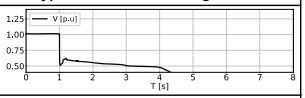
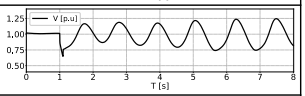
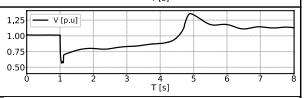
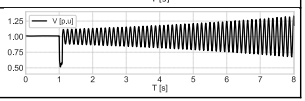
Instability Mechanism	Distinctive Characteristics of Each Short-Term Instability Phenomenon		
	Typical Location	Typical Dynamical Causes and Effects	Typical Post-Fault Voltage Deviation
Short-Term Voltage Instability (STVI)	Grid sections with high amounts of (dynamic) load and DER;	Motors dynamics and stalling; HVDC links dynamics; IBR and DER dynamics; Combination of any of the above;	
Transient Rotor Angle Instability (TRAI)	Tie-lines; Near-SG buses;	Large rotor angle deviations; Low CCT - loss of synchronism; Electromechanical oscillations (<2.5Hz);	
Fault-Induced Delayed Voltage Recovery (FIDVR)	Areas with a high share of dynamic load (particularly A/C units) and DER;	Massive induction motors stalling (mostly A/C) and disconnection; Massive DER disconnection and/or blocking;	
Converter-driven Slow Interactions Instability (CSII)	Remote or "weak" grid sections (low strength); Areas with many IBRs; High IBR control gains;	Interactions between IBR/DER controls, grid, and SG controls; Interactions with dynamic load; Oscillations (typically 7-10Hz range);	

Figure 2: A concise overview of the four distinctive short-term instability phenomena, characteristics, and illustrative voltage deviations [4].

In the past, long-term voltage stability has generally been the focus of academia and industry. However, short-term (voltage) instability becomes much more pronounced with the proliferation of distributed energy resources (DER) and various types of dynamic loads such as induction motors and electronically controlled motors [2]. This presents a challenge as distribution networks become Active Distribution Networks (ADNs), with a potentially significant impact on system stability [2, 5]. This impact ought to be analysed in modern power systems, which is the goal of the presented analysis.

Meanwhile, parameter uncertainty is the other very important aspect that limits the ability to analyse the impacts of ADN on bulk power systems comprehensively. This is where DIgSILENT PowerFactory and Python API synergy may provide significant benefits, as showcased in the following sections.

1.3 Model and Methodology Description

To study STVS, the usage of suitable models is a very important first step. Simple models are unable to represent relevant dynamics that occur and lead to short-term instabilities. For this study, the advanced test grid for voltage stability is utilized:

IEEE Test System for Voltage Stability Analysis and Security Assessment [6]. The grid is based on the Nordic Power System, and its single-line diagram and description are provided in Figure 3.

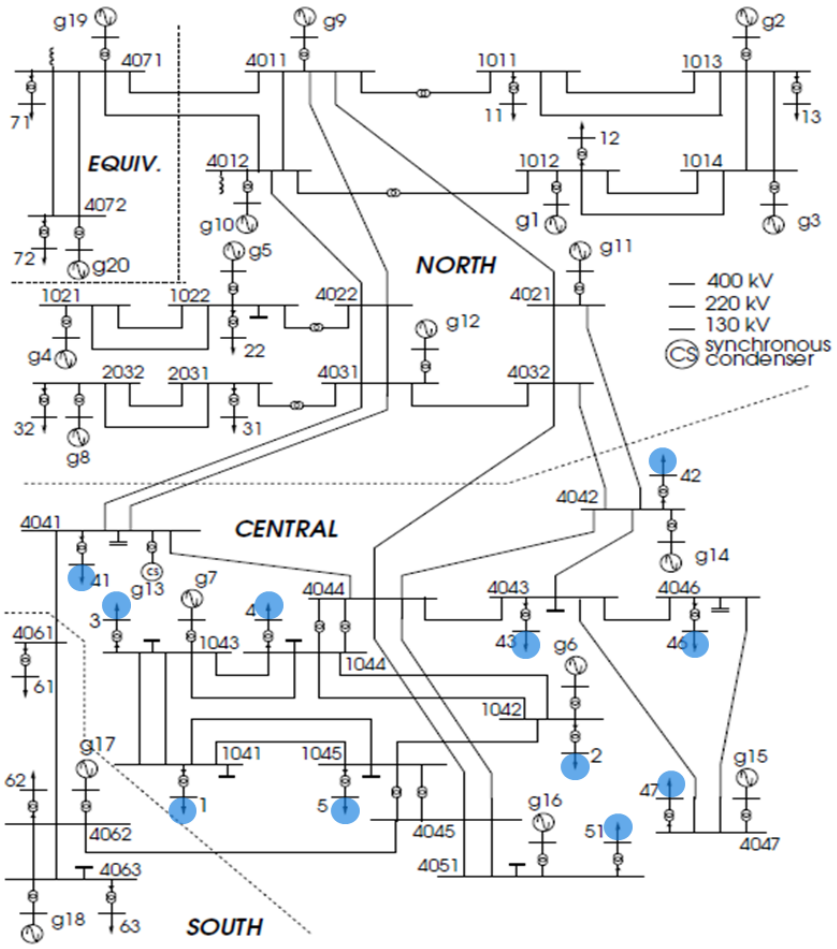


Figure 3: IEEE Test System for Voltage Stability Analysis and Security Assessment [6], with blue circles indicating ADN locations [2]

To study the impacts of ADNs on STVS, it is necessary to introduce representative models of ADNs in the test system. For this task, the latest advanced dynamic load and DER models are introduced to the test system in locations depicted in blue in

Figure 3. Such model enhancements improve the ability to represent the dynamic behaviour seen in real modern grids.

Figure 4 describes the WECC Composite Load Model, commonly utilized to represent the aggregated response of various dynamic loads [7-9]. The model consists of three different types of 3-phase motors (A, B, and C), a single-phase A/C motor (D), electronic load, and static load. Furthermore, feeders and busbars are also modelled. The entire model is available in DIgSILENT templates library, under Loads, titled WECC Dynamic Composite Load. The model is very attractive in representing numerous load-grid dynamics that common static models such as the ZIP model simply cannot reproduce. More discussion on the benefits and details of this model can be found in [7-9].

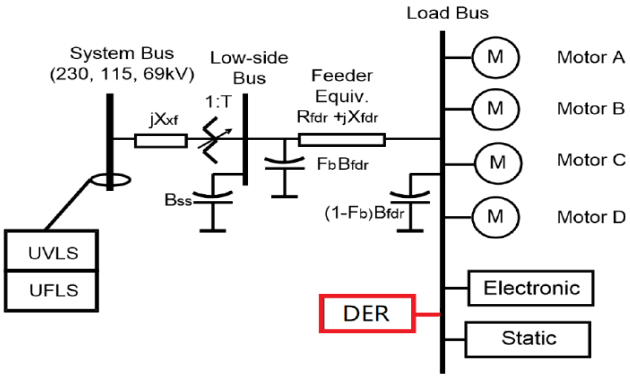


Figure 4: WECC Composite Load model (incl. DER_A model) [2]

Furthermore, distributed generation is also represented in these ADNs. The cutting-edge model for representing DERs in bulk power system stability studies is the DER_A model, with its diagram depicted in Figure 5. The DER_A model is a successor of the well-known PVD1 model, with enhanced abilities to characterize various control strategies of a distributed generator with high fidelity in both static and dynamic operation. It is therefore introduced alongside the WECC load model for a complete ADN model, utilizing DIgSILENT template WECC DER Generation. Details on the DER_A model, its usage, validation, parametrization, etc. can be found in [9-13].

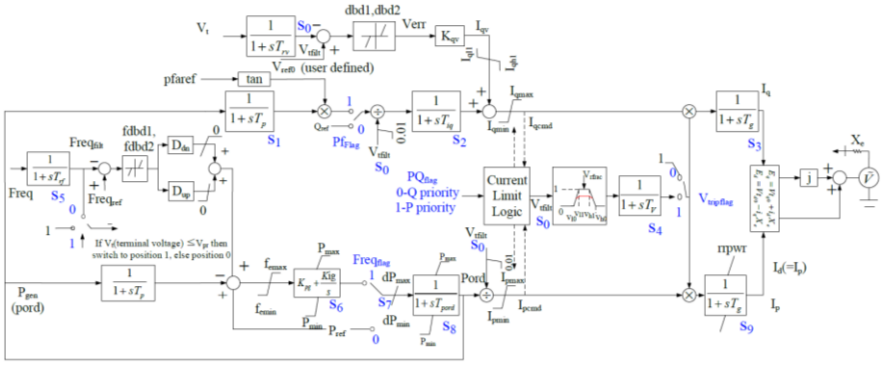


Figure 5: The detailed diagram of the DER_A model [10]

All these models are incorporated together as shown in Figure 3, which creates possibilities to evaluate short-term instabilities in various operational settings. However, this is an extremely broad task, as the derived system has many parameters that may have large effects on system stability. To unpack and understand these effects, running simulations manually one by one is unfeasible from the time perspective. Instead, a massive number of simulations need to be performed automatically, which is where Python API provides enormous benefits. Figure 6 depicts interactions and data exchange between the created Python script and DIgSILENT PowerFactory through Python API.

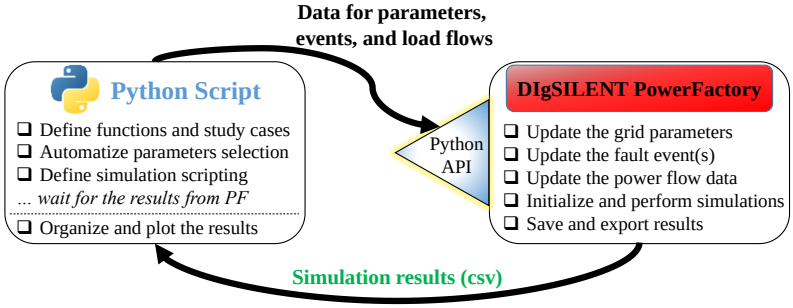


Figure 6: Exchange of data between Python and DIgSILENT PowerFactory

Chapter 1.4 describes in detail how the Python script is created to analyse the STVS of the created test system efficiently, with several varying parameters of interest.

1.4 Python Scripting and Simulation Results

In this section, the Python code used in the analyses will be shown in detail. Deeper discussions on some aspects are omitted and some parts were simplified for brevity. Where missing, the reader is advised to look for further explanations in [2]. The entire analysis is performed in Python 3.8.3 (using Jupyter Notebook) and DIgSILENT PowerFactory 2020 SP2A in the graphical user interface (GUI) [14, 15]. Alternatively, it is also possible to run simulations and the script (with minor adjustments) in the so-called *engine mode*, where an external Python script can run PowerFactory without the need for GUI. More details on this can be found in [14-15], with some examples in [16].

1.4.1 Libraries and functions

First, the libraries that shall be used in the script need to be imported (Figure 7). It is particularly important to make sure that the `powerfactory` module is successfully imported, so Python API can run. It may be necessary to check and adjust the Python path in case of an error and check if the compatible versions are used.

```
1.  import sys
2.  import powerfactory as pf
3.  import numpy as np
4.  import pandas as pd
5.  app = pf.GetApplication()
6.  if app is None:
7.      raise Exception("Getting PowerFactory application failed")
```

Figure 7: Importing necessary libraries

As this script will be relatively long with possible repetitiveness, it is convenient to define several functions to be used. Firstly, two functions for preparing and running simulations are created and shown in Figure 8. Next, two functions for defining simulation events are created. These functions will be used to adjust fault parameters such as fault time, location, duration, impedance, etc. automatically.

```
1.  def setupSimultation(comInc, comSim):
2.      # Initializiation
```

```
3.      comInc.iopt_sim = "rms"
4.      comInc.start = 0
5.      # Defining simulation time
6.      comSim.tstop = 8
7.
8.  def runSimulation(comInc, comSim):
9.      app.EchoOff() #disables PF user interface
10.     comInc.Execute()
11.     app.EchoOn()
12.     comSim.Execute()

13. def clearSimEvents(): ## deletes all previous simulation events
14.     faultFolder =
    app.GetFromStudyCase("Simulationsereignisse/Fehler.IntEvt")
15.     cont = faultFolder.GetContents()
16.     for obj in cont:
17.         obj.Delete()
18.
19. ## creates a new simulation fault event
20. def addFaultEvent(obj, sec, faultType, R, X):
21.     faultFolder =
    app.GetFromStudyCase("Simulationsereignisse/Fehler.IntEvt")
22.     event = faultFolder.CreateObject("EvtShc", obj.loc_name)
23.     event.p_target = obj  # object that will be short-circuited
24.     event.time = sec   ## time instance when the fault occurs
25.     event.i_shc = faultType ## fault type (e.g. 3-phase SC)
26.     if faultType == 0:
27.         event.R_f = R  ## here we define fault R and X
28.         event.X_f = X
```

Figure 8: Defining several functions

Next, to access a specific list of elements in the system shown in Figure 3, a convenient function is created to avoid repeating the same line of code multiple times. The code is shown in Figure 9.

```
1.  ## returns the list of selected elements (in name order)
2.  def getSelectedElements(all_elements,wanted_elements):
3.  #takes a list of objects and a list of strings(names)
4.      elements = []
5.      for i in range(0,len(wanted_elements)):
6.          for z in range(0,len(all_elements)):
7.              if str(all_elements[z].loc_name) ==
    wanted_elements[i]:
8.                  elements.append(all_elements[z])
9.      if len(elements)==0:
```

```
        raise Exception("Something went wrong, the returned list is empty")
    return elements
```

Figure 9: Defining a function for accessing grid elements of interest

The function *getSelectedElements()* works by collecting the list of passed elements (*wanted_elements*) from the passed list of objects, by using the internal *loc_name* characteristic of the objects. Later in the section, it will be shown how this can be used to select a list of various elements such as buses, loads, generators, etc. It is of course necessary to first define these corresponding names in PowerFactory when creating the elements, so they can be accessed. This can also be automatized by using DPL; however, this is out of the scope of this chapter.

Finally, a function to add result variables of interest for exporting is created as shown in Figure 10.

```
def addRecordedResult(elmRes, obj, param):
    if type(obj) is str:
        for elm in app.GetCalcRelevantObjects(obj):
            elmRes.AddVariable(elm, param)
    elif type(obj) is list:
        for elm in obj:
            elmRes.AddVariable(elm, param)
    else:
        elmRes.AddVariable(obj, param)
```

Figure 10: Creating a function for result variables

1.4.2 Selecting the required grid elements

To evaluate dynamic system performance for various fault locations, 12 different buses throughout the system are selected by utilizing the created *getSelectedElements()* function, as shown in Figure 11.

```
# List of bus names of interest
fault_buses_of_interest = ['4042','4043','4044','4041','4062',
'4031','4032','1041','1042','1043','1044','1045']
fault_buses = [] # Empty list to which we will add the buses
# Load all terminals (buses)
all_buses = app.GetCalcRelevantObjects("*.ElmTerm")
# Select the ones of interest by using the pre-defined function
```

```
8.  fault_buses =
    getSelectedElements(all_buses,fault_buses_of_interest)
```

Figure 11: Selecting the buses of interest for simulating faults

The variable *fault_buses* now contains the selected 12 buses of interest. This list shall be automatically looped through to change the fault location during the analysis. Similarly as in Figure 11, one can select other objects of interest, such as loads and DERs defined in Chapter 1.3, and buses for which the output variables are to be monitored (in this case short-term voltages). Furthermore, folders that contain the WECC Dynamic Load scripts shall be also selected. This is all illustrated in Figure 12.

```
1.  ## List of loads
2.  load_names_of_interest = ['01','02','03','04',
3.                            '05','41','42','43','46','47','51']
4.  loads = []
5.  all_loads = app.GetCalcRelevantObjects('*.ElmLod')
6.  loads = getSelectedElements(all_loads,load_names_of_interest)
7.  ## List of DERs
8.  DER_names_of_interest =
    ['DER(1)','DER(2)','DER(3)','DER(4)','DER(5)','DER(41)',
9.   'DER(42)','DER(43)','DER(46)','DER(47)','DER(51)']
10. all_DERs = app.GetCalcRelevantObjects('*.ElmGenstat')
11. DERs = []
12. DERs = getSelectedElements(all_DERs,DER_names_of_interest)
13. ## List of buses for variables output
14. buses_of_interest = ['1041','1042','1043','1045','4041',
15.                      '4042','4043','4046','4047','4051']
16. buses = []

17. buses = getSelectedElements(all_buses,buses_of_interest)
18.
19. # Selecting WECC load folders
20. Nordic = app.GetCalcRelevantObjects('Nordic.ElmNet')[0]
21. all_folders = Nordic.GetContents()
22. folders = []
23. folders_of_interest = ['WECC CMPLDW (01)', 'WECC CMPLDW (02)',
24.     'WECC CMPLDW (03)', 'WECC CMPLDW (04)','WECC CMPLDW (05)',
25.     'WECC CMPLDW (41)', 'WECC CMPLDW (42)','WECC CMPLDW (43)',
26.     'WECC CMPLDW (46)', 'WECC CMPLDW (47)','WECC CMPLDW (51)']
27. folders = getSelectedElements(all_folders,folders_of_interest)
```

Figure 12: Selecting other elements of interest (loads, DERs, WECC folders)

Now that the required WECC folders are located, we can loop through them to get the DPL and DSL objects of relevance that can be used to adjust the dynamic load model parameters in simulations. The code is shown in Figure 13.

```
scripts = []
DSLs_motors = []

for i in range(0,len(folders)):
    scripts.append(folders[i].GetContents('*.ComDpl',1)[0])

for i in range(0,len(folders)):
    DSLs_motors.append(folders[i].GetContents('Motor D dynamic model.ElmDsl',1)[0])
```

Figure 13: Selecting WECC scripts and DSL files for D-motors

A similar process can be repeated to access other dynamic model parameters, such as parameters of motors A/B/C, static load, feeder(s) parameters, etc.

Finally, to access DER dynamic parameters, their respective DSL files need to be selected, as shown in Figure 14.

```
DSL_names_of_interest = ['DER (01)','DER (02)','DER (03)','DER (04)','DER (05)','DER (41)', 'DER (42)','DER (43)','DER (46)','DER (47)','DER (51)']
DSLs = []
all_DSLs = app.GetCalcRelevantObjects('*.ElmDsl')

for i in range(0,len(DSL_names_of_interest)):
    for z in range(0,len(all_DSLs)):
        if str(all_DSLs[z].chr_name) == DSL_names_of_interest[i]:
            DSLs.append(all_DSLs[z])
```

Figure 14: Selecting WECC scripts and DSL files for D-motors

Notice that the pre-defined function was not used in this instance, as these DSL objects are collected based on the *chr_name* characteristic instead of *loc_name*. This can be of course integrated into the function (or *loc_name* of DSLs can be adjusted so the function can be also used) if desired.

The following lists are therefore collected so far:

- List of 12 different fault buses (*fault_buses*)

- List of 11 static loads in the central area (*Loads*)
- List of 11 WECC dynamic loads (and D-motor DSLs) in the central area (*scripts* and *DSLs_motors*)
- List of 11 DERs (and their DSLs) in the central area (*DERs* and *DSLs*)
- List of 10 buses for which the STVS will be evaluated (*buses*)

1.4.3 Initializing the script

First, several empty arrays are initialized as shown in Figure 15, which shall be used to organize the results.

```
Sim_number = 0 #counter

output_size = 1320 # total amount of planned simulations
Output_Iteration_number = np.zeros(output_size)
Output_Load_ratio = np.zeros(output_size)
Output_Motor_A_share = np.zeros(output_size)
Output_Motor_B_share = np.zeros(output_size)
Output_Motor_C_share = np.zeros(output_size)
Output_Motor_D_share = np.zeros(output_size)
Output_Fault_bus = np.zeros(output_size)
Output_Instability = np.zeros(output_size)
Indexes_3s = np.zeros(output_size)
Motors_case_output = np.zeros(output_size)
```

Figure 15: Initializing several empty arrays to be used for results

Ten different cases of dynamic load types and percentages are defined in the matrix *Motor_abcd_matrix* so that the impact of various motors on STVS can be evaluated. Next, a *load_ratio_vector* is defined, which shall be used to define the ratio of static/dynamic load in the grid, from zero to 50% in 5% steps (Figure 16). More details on both can be found in [2].

```
#Share of motors for 10 ABCD scenarios              #Case
Motor_abcd_matrix = ([0,    0,    0,    0],          #0    'N'
                     [0.15, 0.15, 0.15, 0.15],       #1    '0'
                     [0.3,  0.1,  0.1,  0.1],        #2    'A'
                     [0.45, 0.05, 0.05, 0.05],       #3    'AA'
                     [0.1,  0.3,  0.1,  0.1],        #4    'B'
                     [0.05, 0.45, 0.05, 0.05],       #5    'BB'
                     [0.1,  0.1,  0.3,  0.1],        #6    'C'
                     [0.05, 0.05, 0.45, 0.05],       #7    'CC'
```

```
                    [0.1,  0.1,  0.1,  0.3],        #8    'D'
                    [0.05, 0.05, 0.05, 0.45],)      #9    'DD'

Load_ratio_vector = [1, 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.65,
    0.6, 0.55, 0.5]
```

Figure 16: Initializing dynamic load type and penetration scenarios

1.4.4 Scripting: Inside the loops

Now everything is ready for creating the scripting scenarios. This shall be done by nesting three **for** loops, as shown in Figure 17. The first loop (*x*) shall go through different penetration of dynamic loads, defined through the *Load_ratio_vector* variable. The second loop (*y*) will select the dynamic motor types scenarios, defined in the *Motor_abcd_matrix* variable. Finally, the third loop (*z*) will be used to change the fault location by utilizing the list of buses pre-defined in *fault_buses*.

```
#-------------Set up simulation cases--------------------------
for x in range(0, len(Load_ratio_vector)):  # Loop 1
    Load_ratio = Load_ratio_vector[x]
    for y in range(0, len(Motor_abcd_matrix)):  # Loop 2
        Motors_abcd = Motor_abcd_matrix[y]
        for z in range(0, len(fault_buses)):       # Loop 3
            Fault_bus = fault_buses[z]
```

Figure 17: Defining three nested for loops for scripting

With the defined variables, there is a total of 1320 simulations inside the three nested **for** loops (11 x 10 x 12). Each simulation shall be a dynamic simulation performed within PowerFactory, lasting 8 seconds (defined in Figure 8). For each iteration, the Python script adjusts the parameters as selected.

Inside the three loops, each iteration needs to be initialized. First, load flow data is calculated and implemented. For this, nominal values are defined [6], and each load's power flow is then adjusted by the *Load_ratio* of the current iteration.

```
P_nom = [600, 330, 260, 840, 720, 540, 400, 900, 700, 100, 800]
Q_nom = [148, 71, 84, 252, 190, 131, 127, 254, 212, 44, 258]
# Load flow for static loads, see [2, 6]
for i in range(0,len(loads)):
    loads[i].plini = P_nom[i]*Load_ratio
    loads[i].qlini = Q_nom[i]*Load_ratio
```

```
## Load flow for dynamic loads
for i in range(0,len(scripts)):
    scripts[i].SetInputParameterDouble('Pset', P_nom[i] -
loads[i].plini + 0.1) # 0.1 added to avoid errors due to = 0
    scripts[i].SetInputParameterDouble('Qset', Q_nom[i] -
loads[i].qlini + 0.1)

    scripts[i].SetInputParameterDouble('Fma', Motors_abcd[0])
    scripts[i].SetInputParameterDouble('Fmb', Motors_abcd[1])
    scripts[i].SetInputParameterDouble('Fmc', Motors_abcd[2])
    scripts[i].SetInputParameterDouble('Fmd', Motors_abcd[3])
    scripts[i].SetInputParameterDouble('Fel', 0.15)


    scripts[i].Execute() # execute to update the settings!
```

Figure 18: Setting the load flow and parameters of static and dynamic loads

Next, the load flow for WECC dynamic loads is defined. As the goal is to test how different load *composition* affects the STVS, the total load needs to remain the same, i.e. nominal values. Therefore, each WECC load has a power flow of the difference between the nominal load and the corresponding static load. This can be done by using the `scripts` list defined previously, to access the relevant parameters for P, Q, and dynamic motor share, per type. The lines of code are depicted in Figure 18. As DERs are not considered in the first part of the analysis, they can be all set out of service by a simple line of code shown in Figure 19.

```
for i in range(0,len(DERs)):
    DERs[i].outserv = 1
```

Figure 19: Setting all DERs to 'out-of-service'

Since all the variables of interest are now initialized, it is time to implement the simulation fault scenarios, and then initialize and run the simulations. An example of how to implement this is given in Figure 20.

```
comInc = app.GetFromStudyCase("ComInc")
comSim = app.GetFromStudyCase("ComSim")
comInc.Execute() # Run initial conditions
setupSimultation(comInc, comSim)

clearSimEvents() # Delete existing (previous) simulation events
faultFolder =
app.GetFromStudyCase("Simulationsereignisse/Fehler")
```

```
SC_element = Fault_bus # Current fault bus object from loop z

time = 1 # Fault time
clearTime = 1.1 # Fault clearing time
faultType = 0  # Fault type; 0 is 3-phase short-circuit
faulClear = 4  # Fault type; 4 is fault clearing
R = 20 # Fault resistance
X = 0 # Fault reactance

addFaultEvent(SC_element, time, faultType, R, X)  #Add SC event
addFaultEvent(SC_element, clearTime, faulClear, R, X) #Clear
runSimulation(comInc, comSim) # Run simulation
```

Figure 20: Simulation events, initialization, and running of simulations

As iterations in the z-loop take place, different fault buses will be selected, as defined in the *Fault_bus* variable. An RMS dynamic simulation will be performed for each iteration, varying the dynamic load percentage, load composition, and fault location, by using loops *x*, *y*, and *z*, respectively.

1.4.5 Outputs of simulations

The output results need to be defined, organized, and automatized. Therefore, the result files are updated with the variables of interest. In Figure 21, it is shown how each simulation (iteration) is finished by saving a corresponding CSV file that contains all the relevant variables.

```
COMRES = app.GetFromStudyCase("ASCII Results Export.ComRes")
ElmRes = app.GetCalcRelevantObjects("*.ElmRes")[0]
# Adding all the voltage variables to COMRES/ElmRes so they are
# exported to csv files
for i in range(0,len(buses)):
    addRecordedResult(ElmRes, buses[i], "m:u1")

COMRES.element = [] # reset values
COMRES.variable = []
COMRES.element = [ElmRes] + buses #select time and voltages
COMRES.variable = ["b:tnow"] + len(buses)*['m:u1']

COMRES.iopt_exp = 6 # type 6 is a csv file
COMRES.f_name = (r'C:\Users\aboricic\PLOTS' + '\\' +
str(Sim_number) + '_STVS.csv') # Sim_number used for file names
COMRES.Execute()
```

Figure 21: Defining variables of interest for results and outputs

To evaluate STVS, the methodology described in Figure 22 is used. More details on this methodology can be found in [2], while an overview of common STVS evaluation methods can be found in [17], as well as more discussion in [4].

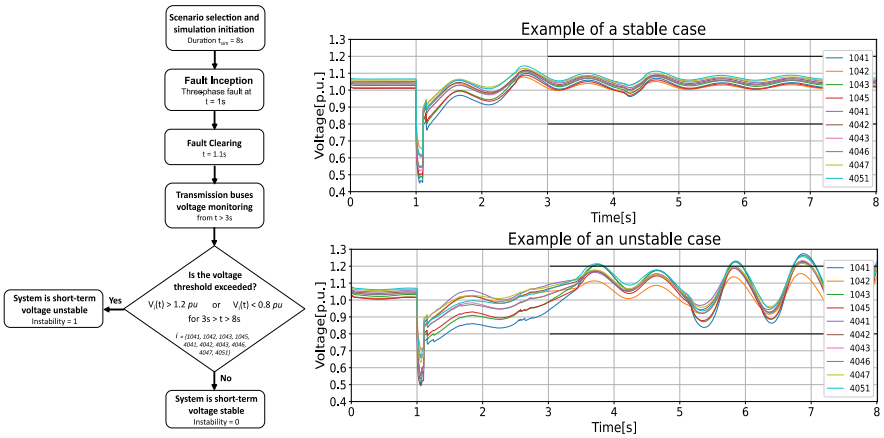


Figure 22: Methodology for STVS evaluation with an example [2]

This is implemented in the Python code as displayed in Figure 23.

```
voltages = pd.DataFrame(pd.read_csv(COMRES.f_name, skiprows=0, decimal='.'))

DataSet = voltages.loc[2:].apply(pd.to_numeric)
DataSet.columns = ['Time'] + buses_of_interest
DataSet.reset_index(drop=True,inplace=True)

t_round = np.floor(DataSet['Time'])
for i, time in enumerate(t_round): # Find index for t=3sec
    if time == 3:
        index_3s = i
        break

Instability = 0    # Apply the methodology from Figure 22
if ((DataSet.iloc[index_3s:,1:] > 1.2).values.any() == True) or ((DataSet.iloc[index_3s:,1:] < 0.8).values.any() == True):
    Instability = 1
```

Figure 23: Code for differentiating stable and unstable STVS cases

Lastly, all the results are organized in an excel sheet, so they can be easily extracted

and visualized as heatmaps. This is indicated in Figure 24. Alternatively, it could be also done using the built-in *.append()* function.

```
Output_Iteration_number[Sim_number-1] = Sim_number
Output_Load_ratio[Sim_number-1] = Load_ratio
Output_Motor_A_share[Sim_number-1] = Motors_abcd[0]
Output_Motor_B_share[Sim_number-1] = Motors_abcd[1]
Output_Motor_C_share[Sim_number-1] = Motors_abcd[2]
Output_Motor_D_share[Sim_number-1] = Motors_abcd[3]
Output_Fault_bus[Sim_number-1] = int(Fault_bus.loc_name)
Output_Instability[Sim_number-1] = Instability
Indexes_3s[Sim_number-1] = index_3s
Motors_case_output[Sim_number-1] = y
## Following code is out of loops
```

Figure 24: Organizing the data in an overview file

It is important to keep in mind that these last lines (Figure 25) of the code should be *outside* of the three nested loops. Otherwise, a file would be created for each iteration, instead of a final single overview file.

```
Data = {'Simulation Number': Output_Iteration_number,
        'Load Ratio': Output_Load_ratio,
        'Motor A share': Output_Motor_A_share,
        'Motor B share': Output_Motor_B_share,
        'Motor C share': Output_Motor_C_share,
        'Motor D share': Output_Motor_D_share,
        'Fault Bus': Output_Fault_bus,
        'Instability': Output_Instability,
        'Index_3sec': Indexes_3s,
        'Motors_case': Motors_case_output}
DataFrame = pd.DataFrame(Data, columns = ['Simulation Number',
'Load Ratio', 'Motor A share', 'Motor B share', 'Motor C share',
'Motor  D  share',  'Fault  Bus','Instability','Index_3sec',
'Motors_case'])
DataFrame.to_excel(r'C:\Users\aboricic\Results\Simulations
Overview.xlsx', index = False, header=True)
```

Figure 25: Saving the data in an overview file

The output excel sheet with 1320 rows (simulations) is exemplified in Figure 26.

	A	B	C	D	E	F	G	H	I	J
1	Simulation Number	Load Ratio	Motor A share	Motor B share	Motor C share	Motor D share	Fault Bus	Instability	Index_3sec	Motors_case
2	1	1	0	0	0	0	4042	0	309	0
3	2	1	0	0	0	0	4043	0	315	0
4	3	1	0	0	0	0	4044	0	316	0
5	4	1	0	0	0	0	4041	0	313	0
6	5	1	0	0	0	0	4062	0	309	0
7	6	1	0	0	0	0	4031	0	309	0
8	7	1	0	0	0	0	4032	0	308	0
9	8	1	0	0	0	0	1041	0	303	0
10	9	1	0	0	0	0	1042	0	303	0
11	10	1	0	0	0	0	1043	0	303	0
12	11	1	0	0	0	0	1044	0	303	0
13	12	1	0	0	0	0	1045	0	303	0
14	13	1	0.15	0.15	0.15	0.15	4042	0	361	1
15	14	1	0.15	0.15	0.15	0.15	4043	0	371	1
16	15	1	0.15	0.15	0.15	0.15	4044	0	373	1
17	16	1	0.15	0.15	0.15	0.15	4041	0	344	1
18	17	1	0.15	0.15	0.15	0.15	4062	0	314	1

⋮ ⋮ ⋮

	A	B	C	D	E	F	G	H	I	J
1	Simulation Number	Load Ratio	Motor A share	Motor B share	Motor C share	Motor D share	Fault Bus	Instability	Index_3sec	Motors_case
1304	1303	0.5	0.1	0.1	0.1	0.3	4032	1	413	8
1305	1304	0.5	0.1	0.1	0.1	0.3	1041	0	552	8
1306	1305	0.5	0.1	0.1	0.1	0.3	1042	1	580	8
1307	1306	0.5	0.1	0.1	0.1	0.3	1043	1	734	8
1308	1307	0.5	0.1	0.1	0.1	0.3	1044	1	859	8
1309	1308	0.5	0.1	0.1	0.1	0.3	1045	1	644	8
1310	1309	0.5	0.05	0.05	0.05	0.45	4042	1	417	9
1311	1310	0.5	0.05	0.05	0.05	0.45	4043	1	417	9
1312	1311	0.5	0.05	0.05	0.05	0.45	4044	1	407	9
1313	1312	0.5	0.05	0.05	0.05	0.45	4041	1	410	9
1314	1313	0.5	0.05	0.05	0.05	0.45	4062	1	406	9
1315	1314	0.5	0.05	0.05	0.05	0.45	4031	1	433	9
1316	1315	0.5	0.05	0.05	0.05	0.45	4032	1	417	9
1317	1316	0.5	0.05	0.05	0.05	0.45	1041	0	611	9
1318	1317	0.5	0.05	0.05	0.05	0.45	1042	1	717	9
1319	1318	0.5	0.05	0.05	0.05	0.45	1043	0	567	9
1320	1319	0.5	0.05	0.05	0.05	0.45	1044	0	504	9
1321	1320	0.5	0.05	0.05	0.05	0.45	1045	1	812	9

Figure 26: Example of the output overview excel file

After the simulations have been completed and the results saved, visualization can be performed in a separate Python notebook. The code in Figure 27 takes the *.xlsx* file and plots it on a heatmap, as in [2]. Furthermore, each case visualization or further analysis can be performed with saved CSV files. Some of it is shown in [2].

```
1.  import numpy as np
2.  import pandas as pd
3.  from pandas import DataFrame
4.  import seaborn as sns
5.  import matplotlib.pyplot as plt
6.  DataSet   =   pd.read_excel('Simulations   Overview_WECC.xlsx',
    index_col=0)
7.  # Loop sizes we used
8.  x=10
9.  y=11
10. z=12
11. ##--------Selecting and reshaping the data as desired---------
```

```
# Select the instability column and split it into several equal arrays that represent a case scenario for all buses
B = np.array_split(DataSet['Instability'].iloc[:x*y*z], x*y)
Instab_sum = []
# Sum instability numbers to get the number of instabilities per case scenario for all buses
for i in range (0,len(B)):
    Instab_sum.append(sum(B[i]))
heatmap = np.reshape(Instab_sum, (11, 10)).T # Form a matrix for the heatmap shape
heatmap = np.delete(heatmap, 0, 1) # First column not needed
#labels
Motors_case = ['0','N','A','AA','B','BB','C','CC','D','DD']
Load_ratio = [5,10,15,20,25,30,35,40,45,50] #See Figure 16

heatmap = pd.DataFrame(heatmap, index=Motors_case, columns=Load_ratio) # Make dataframe for plotting

# Plotting the heatmap
heatmap_plot = sns.heatmap(heatmap, linewidths=.5, annot=True, cmap="RdYlGn_r",cbar_kws={'label': 'Number of unstable cases'})

plt.tight_layout()
plt.title("Influence of WECC Dynamic Load on STVS")
plt.xlabel("WECC Dynamic Load Percentage [%]", fontsize=12)
plt.ylabel("Motor case scenario", fontsize=12)

plt.gcf().subplots_adjust(bottom=0.15)
plt.gcf().subplots_adjust(left=0.1)
```

Figure 27: Code for plotting the heatmap from the simulations overview file

The resulting plot, which shows the number of STVS unstable simulations per scenario, is given in Figure 28. Utilizing the number of unstable cases as a relevant metric, the heatmap indicates a couple of things: (i) how different dynamic load percentage affects STVS; (ii) how different dynamic load compositions affect STVS. From the heatmap, one can conclude that larger dynamic load penetrations degrade STVS, with D-type motors (e.g. single-phase A/C units) significantly affecting STVS, unlike B-type motors (e.g. ventilation systems). Further discussion on the results and various related analyses can be found in [2].

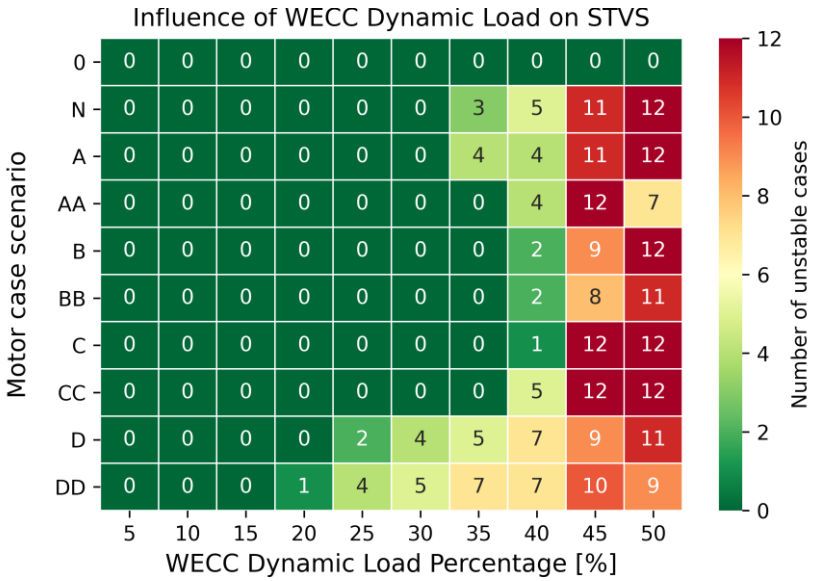


Figure 28: Heatmap resulting from 1320 dynamic simulations [2]

Except for the described analysis, several more analyses of a similar type are performed in [2]. Besides dynamic loads, the impact of DERs and their respective control strategy on STVS was analysed. This was also performed through Python scripting, and some of the relevant code additions necessary for the study are indicated in Figure 29.

```
# Select all the synchronous generators
generators = getSelectedElements('*.ElmSym')
P_nom_gen = # [...] Nominal generators' power data from [2,6]
for i in range(0,len(generators)):
    gen[i].pgini = P_nom_gen[i] * Gen_ratio
# Gen_ratio can be defined/looped similarly to load_ratio

# set power for each DER
DER_p = # ...  depending on the desired scenario
# in [2] it is used to replace Synch. Gen. power
for i in range(0,len(DERs)):
    DERs[i].outserv = 0
    DERs[i].sgn = DER_p
    DERs[i].pgini = DER_p
    DERs[i].Pmax_ucPU = DER_p
    DERs[i].Pmax = DER_p
```

```
17.
18. # Define the control strategy of DERs
19. # (ride-through Q-priority example, see Table A1 in [2])
20. for i in range(0,len(DSLs)):
21.     DSLs[i].vl0 = 0.15    # Voltage break-point
22.     DSLs[i].vl1 = 0.9     # Voltage break-point
23.     DSLs[i].tvl0 = 0.1    # Timer for vl0
24.     DSLs[i].tvl1 = 1.5    # Timer for vl1
25.     DSLs[i].Vtripflag = 0 # Enable voltage trip
26.     DSLs[i].Pqflag = 0    # P/Q priority
27.
28. # Example parameters to simulate FIDVR (see Section 4.3 in [2])
29. for i in range(0,len(DSLs_motors)):
30.     DSLs_motors[i].Vstall = 0.55 #Voltage at which A/Cs stall
31.     DSLs_motors[i].Tstall = 0.3 #Time needed to stall (<Vstall)
32.     DSLs_motors[i].Frst = 0.2  #Ratio that recovers post-FIDVR
```

Figure 29: Extra code example for various parametrization of WECC dynamic models and DER_A models used in [2, 4]

By utilizing approaches described so far, results such as the ones shown in Figure 30 are achieved. In a similar concept as Figure 28, the heatmaps provide the following insights: (i) how different penetrations of DER affect STVS, per control type; (ii) how different dynamic load compositions affect DER-penetrated systems' STVS, per DER control type; (iii) how DER units interact with different dynamic load models, per control type. One important conclusion that can be derived from the heatmap is that DER ride-through control strategies (P- and especially Q-priority) are much more favourable for preserving STVS, compared to disconnection and momentary cessation. Such a conclusion is more emphasized as DER penetration is increased. This can be concluded from the dominantly green right parts of the lower heatmaps, compared to the upper ones. A more detailed discussion of these results can be found in [2].

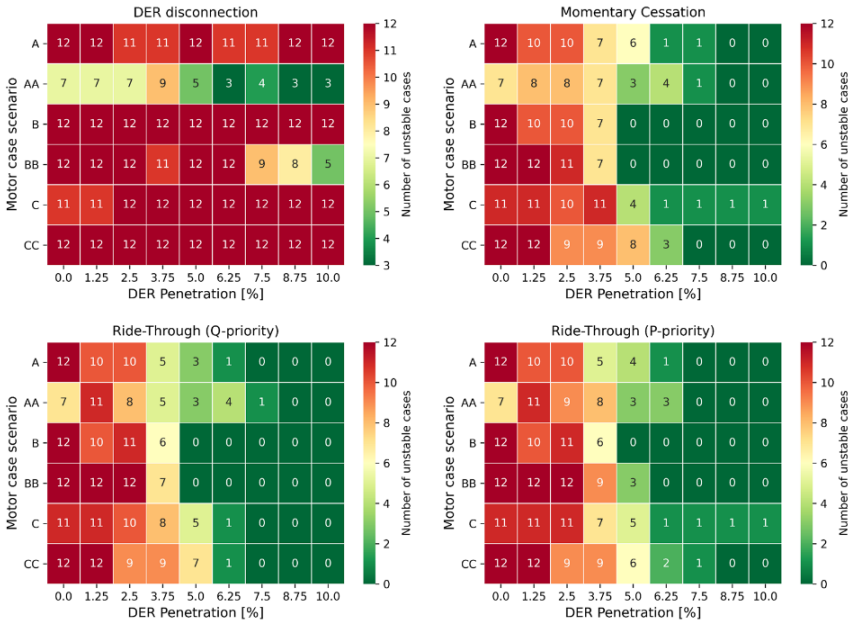


Figure 30: Heatmaps resulting from thousands of simulations for varying DER penetration and control, as well as load composition [2]

For a broader discussion on these and further related findings, as well as more details on the model, methodology, and scripting, curious readers are once again referred to [2]. Furthermore, another example of a similar scripting approach is presented in [4], where various short-term instabilities are modelled and quantified in a more advanced manner.

1.5 Conclusions

Evaluating complex dynamic phenomena in modern power systems is an exceedingly broad and challenging task. System stability, as one of the most important aspects of power system analysis, is becoming increasingly complex to model, evaluate, and ultimately maintain. The sheer number and complexity of inverter-based resources and dynamic loads often make these studies unmanageable. This is where programming and data science support from Python,

combined with advanced simulation software such as DIgSILENT PowerFactory, may bring tremendous value going forward. The described approach in this chapter highlights how short-term voltage stability concerns can be analysed efficiently regardless of the high system complexity. Despite the approach being focused on one specific aspect of power system analysis, the underlying concept of scripting for rapid scenario variation is widely applicable. The linkage between PowerFactory and Python creates enormous possibilities that are ultimately necessary for accelerating the energy transition. Finally, considering the well-established data science and machine learning libraries present in Python, the ability to apply these advanced data techniques to power systems easier and within the PowerFactory simulation environment presents a vast source of opportunities.

Acknowledgement

This work was financially supported by the Dutch Scientific Council NWO in collaboration with TSO TenneT, DSOs Alliander, Stedin, Enduris, VSL and General Electric in the framework of the Energy System Integration & Big Data program under the project “Resilient Synchromeasurement-based Grid Protection Platform, no. 647.003.004”.

References

[1] Gonzalez-Longatt FM, Rueda Torres JL. “PowerFactory Applications for Power System Analysis”, *Springer* 2014

[2] A. Boričić, J. L. R. Torres, M. Popov, “Fundamental Study on the Influence of Dynamic Load and Distributed Energy Resources on Power System Short-Term Voltage Stability”, *International Journal of Electrical Power & Energy Systems*, 2021.

[3] IEEE PES-TR77, “Stability definitions and characterization of dynamic behavior in systems with high penetration of power electronic interfaced technologies”, April 2020.

[4] A. Boričić, J. L. R. Torres, M. Popov, “Quantifying the Severity of Short-term Instability Voltage Deviations”, *International Conference on Smart Energy Systems and Technologies (SEST)*, Sept. 2022.

[5] A. Boričić, J. L. R. Torres, M. Popov, "Impact of Modelling Assumptions on the Voltage Stability Assessment of Active Distribution Grids," *IEEE PES Innovative Smart Grid Technologies (ISGT) Europe*, 2020.
[6] T. V. Cutsem, M. Glavic, W. Rosehart, et al. "Test Systems for Voltage Stability Studies", *IEEE Transactions on Power Systems*, 2020.
[7] WECC Dynamic Composite Load Model (CMPLDW) Specifications, January 2015
[8] WECC Modelling Group "Composite Load Model for Dynamic Simulations – Report 1.0
[9] Z. Ma, Z. Wang, Y. Wang, et al. "Mathematical Representation of WECC Composite Load Model", *Journal of Modern Power Systems and Clean Energy*, September 2020
[10] Electrical Power Research Institute (EPRI), "The New Aggregated Distributed Energy Resources (der a) Model for Transmission Planning Studies: 2019 Update,"
[11] NERC "Reliability Guideline Distributed Energy Resource Modelling", Sept. 2017
[12] NERC "Reliability Guideline Modelling Distributed Energy Resources in Dynamic Load Models", Dec. 2016
[13] NERC "Reliability Guideline Parameterization of the DER_A Model", Sept. 2019
[14] DIgSILENT PowerFactory Version 2020 SP2A "User Manual".
[15] DIgSILENT PowerFactory Version 2020 SP2A "Python Reference".
[16] Gonzalez-Longatt FM, Rueda Torres JL. "Advanced Smart Grid Functionalities Based on PowerFactory", *Springer* 2018
[17] A. Boričić, J. L. R. Torres, M. Popov, "Comprehensive Review of Short-Term Voltage Stability Evaluation Methods in Modern Power Systems", *Energies*, 2021.