

Student Research Paper
Critical clearing time of synchronous generators

Author: Maximilian Köhler, B. Eng.
(23176975)

Supervisor: Ilya Burlakin, M. Sc.

Submission date: March 28, 2024



Contents

1	Introduction	1
2	Fundamentals	3
2.1	Basics synchronous generators	3
2.2	System stability esp. transient context	3
2.3	Analytical calculation of the critical clearing time	5
2.4	Numerical methods for system modeling	6
3	Numerical modelling	7
3.1	Structure of the CCT assessment	7
3.2	Electrical simplifications and scenario setting	8
3.2.1	Electric networks	8
3.2.2	Simulation cases and boundaries	9
3.2.3	Initial value calculation	9
3.3	Implementation of the time domain solution	9
3.4	Implementation of the equal area criterion	10
4	Results	11
4.1	Analytical results	11
4.2	Numerical results	11
4.2.1	Simulated faults	12
4.2.2	Using algebraic calculations vs. non-algebraic	13
4.2.3	Solving the CCT without TDSs	13
4.2.4	Parameter influence analysis	13
4.3	Discussion	13
4.4	Limitations	13
5	Summary and outlook	15
	Acronyms	IX
	Symbols	XI
	Appendix	XIII

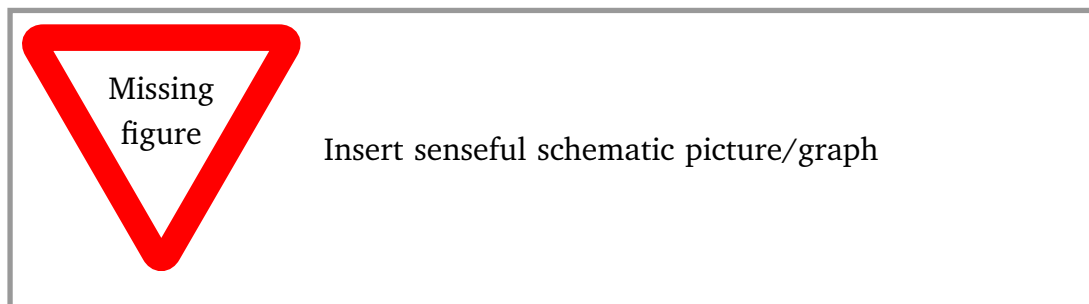
1 Introduction

Bullet points for the thesis from Ilya:

- Swing equation of synchronous generators
- Solving the Swing equation with the help of Python -> Solving of second order ODEs
- Equal-area criterion -> Derivation of the equations
- Simulation of a fault -> applying the equal-area criteria with the help of Python.
- Comparison between analytical and (numerical) simulation results

Introduction via [vdeverbandderelektrotechnikelektronikinformationstechnike.v.Perspektiven] and other standard literature like [gloverPowerSystemAnalysis2017, kundurPowerSystemStability2003, machowskiPowerSystemDynamics2020, oedingElektrischeKraftwerkeUnd2016, schwabElektrischeKraftwerkeUnd2016].
Need for understanding of Transient stability and therefore critical pole angle and fault clearing time assessment: Running and maintaining the electrical grid; Adding virtual inertia in FACTS and HVDC; Better and faster predicting, due to shorter (critical) fault clearing times; .

[MK1]: Write Introduction



The goal of this Student Research Paper is the implementation of a critical clearing time (CCT) determining Python algorithm for a single machine infinite bus (SMIB) model. Therefore a handful of faults or fault scenarios shall be simulated with the program. In combination with a few visualizations the concepts of transient stability assessment, and therefore determining the CCT and the critical power angle, are illustrated.

2 Fundamentals

[MK2]: Write
Chap Funda-
mentals

[Input of basic knowledge for system modelling; Maybe supplementary knowledge](#)

General sources in terms of standard literature: [oedingElektrischeKraftwerkeUnd2016, gloverPowerSystemAnalysis2017, kundurPowerSystemStability2022, machowskiPowerSystem

2.1 Basics synchronous generators

- [characteristics of a synchronous generator; structure and types of SG's](#)
- [mathematical background and description of the behavior -> dynamic modelling](#)
- **Swing equations**
- [Damping: not interesiting for us](#)

The final swing equation system can be derived to following two equations, which have to be solved in every time step to determine the pole angle δ and the rotor speed ω , respectively the rotor speed change from its base value $\Delta\omega$:

$$\frac{d\delta}{dt} = \Delta\omega \quad (2.1)$$

$$\frac{d\Delta\omega}{dt} = \frac{1}{2 \cdot H_{\text{gen}}} \cdot (P_{\text{m}} - P_{\text{e}}) \quad (2.2)$$

The generation of a time domain solution (TDS) for this equation system takes place in section 3.3.

2.2 System stability esp. transient context

- [What is to be analyzed? And why? -> different stability analysis](#)
- [rotor angle stability](#),
- **derivation of EAC**,
- [basic assessment models \(single machine infinite bus, see \[kundurPowerSystemStability2022](#)

With respect to the limitations, that

1. the machine is operating under balanced three-phase positive-sequence conditions,
2. the machine excitation is constant,
3. the machine losses, saturation, and saliency are neglected,

a simplified single machine infinite bus (SMIB) model can be considered for transient stability assessment (see Figure 2.1). The infinite bus bar (IBB) is working with a constant voltage E_{ibb} and angle δ_{ibb} , typically set to 0° . The real power flowing from the synchronous generator (SG) to the IBB is then expressed within the Equation 2.3 and only dependent on the power angle δ . The reactance X_{res} is expressing the simplified reactance from the respective circuit.

$$P_e = \frac{E_p \cdot E_{ibb}}{X_{res}} \cdot \sin(\delta) \quad (2.3)$$

The mechanical power of the turbine is assumed constant, due to the short occurrence of transient stability problems.

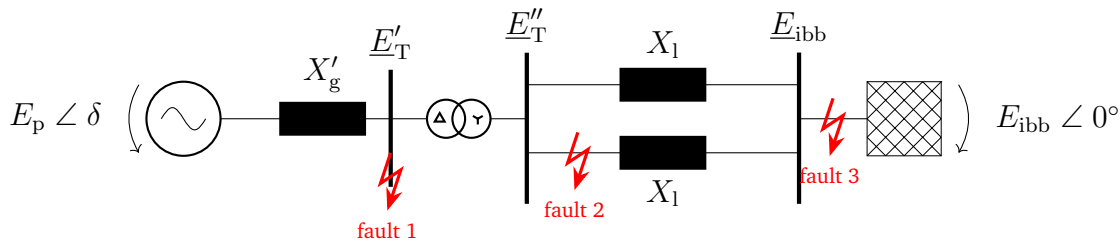


Figure 2.1: Representative circuit of a single machine infinite bus (SMIB) model with pole wheel voltage $E_p \angle \delta$ and infinite bus bar (IBB) voltage $E_{ibb} \angle 0^\circ$; positions of considered faults 1 to 3 are marked with red lightning arrows

2.3 Analytical calculation of the critical clearing time

[MK3]: make text in image bigger

For the analytical solution of the swing equation and following the CCT, there is the need to find the critical power angle δ_{cc} first. For this, the most common approach is the equal area criterion (EAC), considering that the amount of stored energy through acceleration (during the short or failure) is equal to the released energy (decelerating the rotor) when synchronizing again. These both energies can be calculated through the area under the curve of the power difference $\Delta P = P_m - P_e$, while the accelerating area is be-

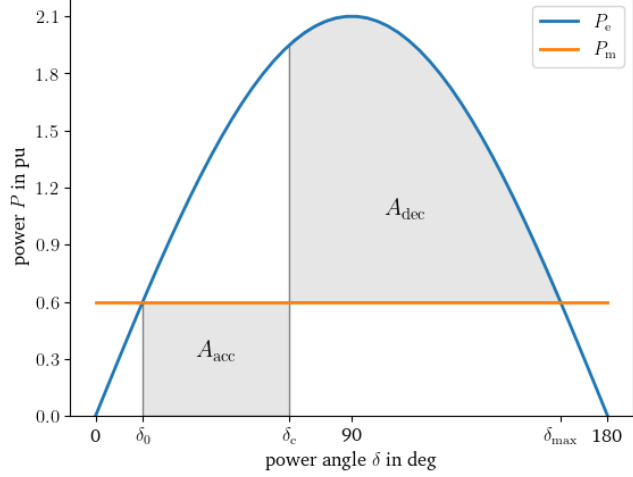


Figure 2.2: Illustrated equal area criterion (EAC) in the P- δ -curve

tween the first stable operating angle δ_0 and the clearing angle δ_c , the decelerating area between δ_c and the maximum dynamically stable angle δ_{max} . Figure 2.2 is illustrating this approach. Following this approach a generalized expression is formed to

$$\int_{\delta_0}^{\delta_1} \Delta P d\delta = 0, \quad (2.4)$$

while the more expressive can be achieved through splitting up the integral borders and equalize both areas:

$$\int_{\delta_0}^{\delta_c} (P_m - P_e) d\delta = \int_{\delta_c}^{\delta_{max}} (P_e - P_m) d\delta \quad (2.5)$$

With consideration of $\delta_{max} = \pi - \delta_0$ and $P_m = P_{max} \cdot \sin(\delta_0)$, and some rearrangements, this leads to the final expression of the critical clearing angle:

$$\delta_{cc} = \arccos \left[\sin(\delta_0) \cdot (\pi - 2 \cdot \delta_0) - \cos(\delta_0) \right] \quad (2.6)$$

The second step is the calculation of the CCT dependent on the critical clearing angle. Splitting the differentiated variables $d^2\delta$ and dt in the combined swing equation and integrating twice, leads to the equation

$$\delta = \frac{\omega \cdot \Delta P}{4H_{\text{gen}}} \cdot t^2 + \delta_0.$$

Rearranging this gives an expression for calculating the critical clearing time t_{cc} (see Equation 2.7).

$$t_{\text{cc}} = \sqrt{\frac{4H_{\text{gen}} \cdot (\delta_{\text{c}} - \delta_0)}{\omega \cdot \Delta P}} \quad (2.7)$$

2.4 Numerical methods for system modeling

- [solving second order ODEs \(explicit\)](#)
- [Differentiation explicit/implicit, inertial value problems, boundary value problems, ...](#)

System dynamics is a method for describing, understand, and discuss complex problems in the context of system theory **[SOURCE]**. They often can be described through a set of coupled ordinary differential equations (ODEs), most resolved in time dimension. [How to bridge towards different boundary types, explicit and implicit methods, ...; Different solving methods, ..., Dirichlet-boundaries, von-Neumann-boundaries, ...](#)

ODEs can be solved through numerical integration with different methods. An easy and less complex method is Euler's method. It uses a linear extrapolation to calculate the functions value at the next timestep, so following the iterable function

$$f_{t+1} = f_t + \left(\frac{df}{dt} \right)_t \cdot \Delta t, \quad (2.8)$$

with t being the time and f an on t dependent function. Generally a system of second order ODEs can be rewritten as two first order equations. This often simplifies the calculation or the use of numerical methods. The presented swing equation of a SG in Equation 2.1 and Equation 2.2 has been split up by that principle.

3 Numerical modelling

Following chapter will describe the implementation of Python Code for solving the derived ODE system (see section 2.1). For this the Python version 3.9 was used, in combination with the packages *scipy*, *numpy*, and *matplotlib*.¹ The complete code is included in the Appendix A.

[MK4]: Write
Chap Meth-
ods

3.1 Structure of the CCT assessment

Program plan for determination / algorithm structure, containing:

- Pre questions:
 1. What do I want to know from the algorithm?
 2. What do I want to see?
- Answers / Hints for the algorithm:
 1. What are needed inputs?
 2. What are needed functions?
 3. How do partial results interact with each other / puzzle together to the superior question?

¹ documentation and manual can be found on <https://scipy.org/> [virtanenSciPyFundamentalAlgorithms2020], similiar for *matplotlib*, and *numpy* packages

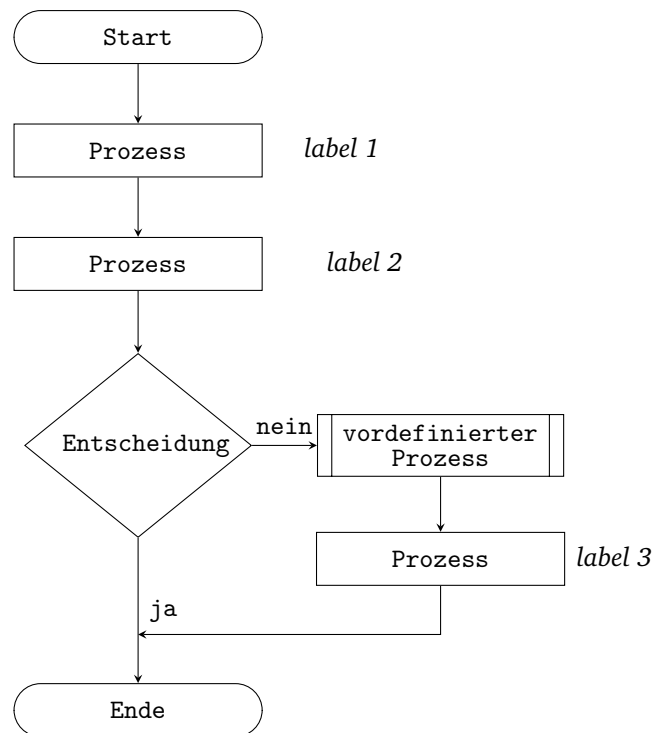
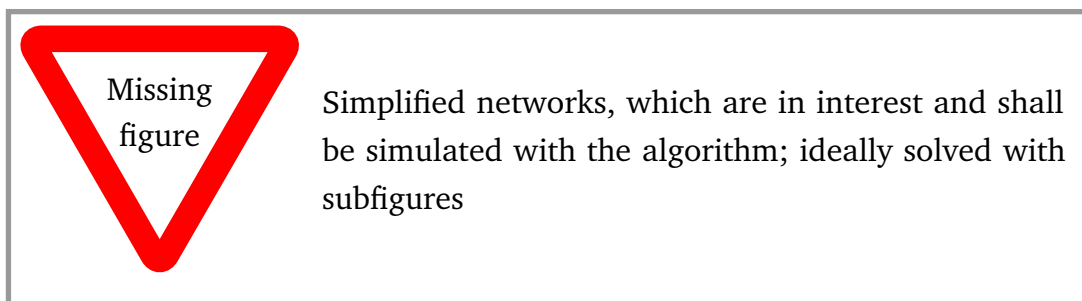


Figure 3.1: Program plan for determining the critical clearing time (CCT)

3.2 Electrical simplifications and scenario setting

- Simplification of all the components in SMIB network to a simple network
- Transforming into symmetrical components (for determination of shorts -> e.g. transformer)

3.2.1 Electric networks



3.2.2 Simulation cases and boundaries

The boundaries of the single cases, which are simulated, are given with a Python dictionary. Therefore the values of the variables have to be predefined and documented. Considering fault cases, there are three differences in the interest:

1. A full line fault, meaning the complete electrical power is disconnected. The CCT of the fault has to be determined, a TDS with fault clearing shortly before and after the critical clearing is carried out and displayed as *stable* and *unstable*;
2. A partial line fault, meaning just a defined percentage of the electrical power is disconnected. The further evaluation is carried out like in scenario 1.;
3. A partial line fault, meaning just a defined percentage of the electrical power is disconnected. But with consideration, that the fault condition is stable at a new operation point. This point is calculated in the time domain.

Further of interest is a parameter variation of both influences H_{gen} and ΔP . This last parameter is not meant to describe the absolute power difference, which is inserted into the swing equation, but more the power difference relative to the maximum electrical power output of the generator. Due to the relation between the maximum power output and the disconnected electrical power in the acceleration and deceleration of the rotor, it seems more significant to use this as a relative parameter. The CCT in dependency of these two influences shall be elaborated.

Lastly, a comparison between calculating with the algebraic equation system of the network and the calculation solely with the power flowing dependent on both (fixed) bus voltages, the reactance and the power angle takes place.

3.2.3 Initial value calculation

- Load flow analysis
- Calculation of E_i , I_i , P_i , δ_i , ..., as well for IBB

3.3 Implementation of the time domain solution

- Different levels of TDS; With/without solving of algebraic equations at each timestep; With/without calculation of turbine momentum at each timestep (dependent on omega)

- Utilization as a function: calculation with clearing and without clearing: for determination of CCT needed

3.4 Implementation of the equal area criterion

- Iterative process needed? Due to omega and delta dependencies of P_e and P_m
- Different methods of CCT calculation: P- δ -curve; P-t-curves

4 Results

[MK5]: Write
Chap Results

4.1 Analytical results

The analytical calculation follows the equations from section 2.3. Therefore the base values for the three input scenarios are used. The results of this calculation are shown in Table 4.1.

Table 4.1: Analytical results for the three fault-scenarios; considering δ_{cc} and t_{cc}

	δ_{cc} in deg	t_{cc} in s
Fault 1		

4.2 Numerical results

Table 4.2 is summarizing the results for the CCT-calculation of the different set scenarios in section 3.2. The single scenarios are further described in the following.

Table 4.2: Results (CCT and δ_{cc}) for numerical solving the faults 1, 2, and 3

Scenario	CCT	δ_{cc}
Fault 1		
Fault 2		
Fault 3		

4.2.1 Simulated faults

Stable scenario

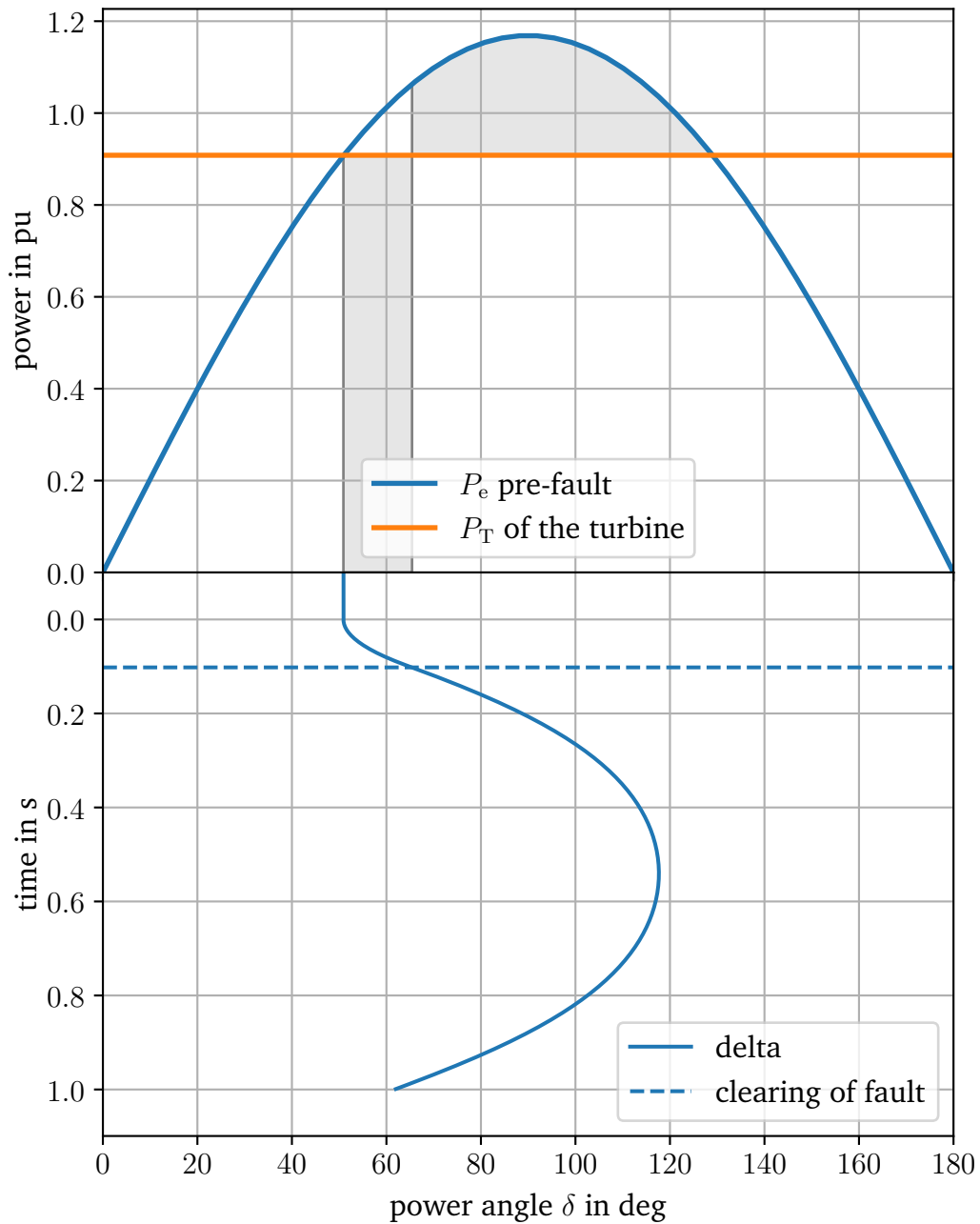


Figure 4.1: Fault 1

4.2.2 Using algebraic calculations vs. non-algebraic

4.2.3 Solving the CCT without TDSs

4.2.4 Parameter influence analysis

The influence of the parameters H_{gen} and ΔP has been carried out with the described code of [...]. The results are shown in Figure 4.2.

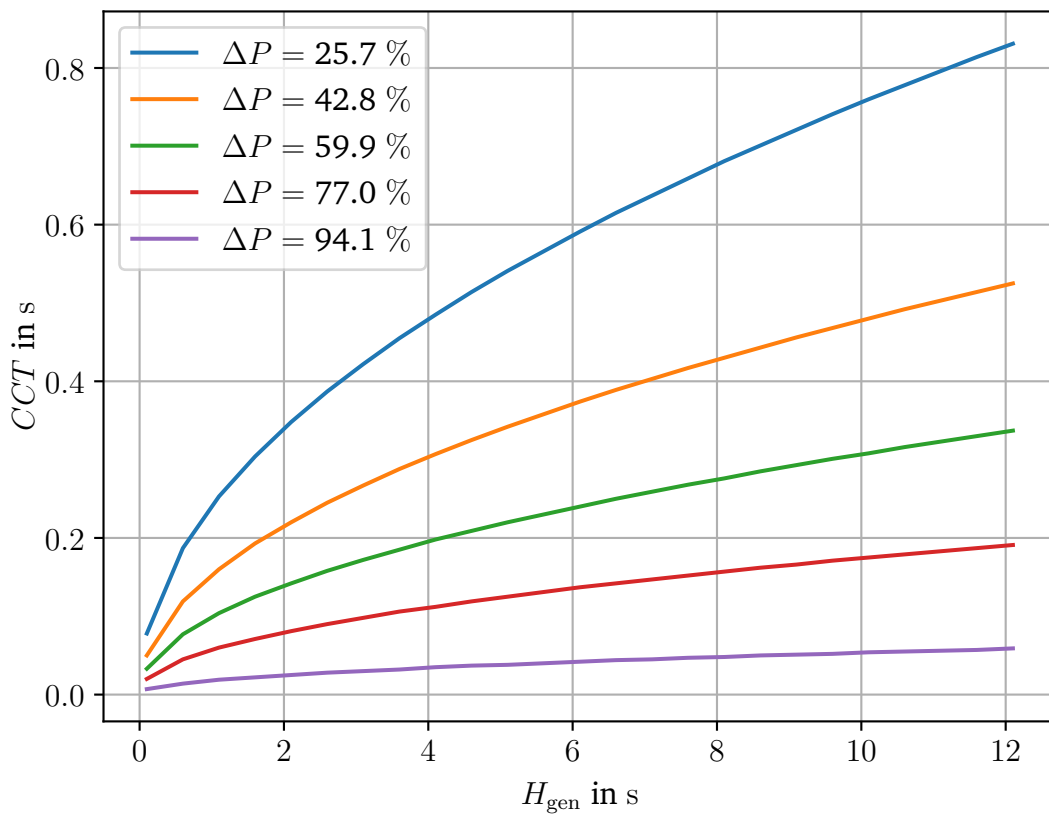


Figure 4.2: Parameter comparison

4.3 Discussion

4.4 Limitations

- Just stable or unstable, not metastable (first swing ok, after that unstable development)

- No damping
- Simplified generator model, only one generator. No machine interaction considered
-

5 Summary and outlook

[MK6]: Write summary and outlook

Short summary of the results.

A brief look in the future and why this topic is in the interest, maybe for slightly other applications as well (see [gaoTransientStabilityAnalysis2024]).

- Usage of CCT assessment for different topics like: Grid coupling (stations); transient balancing processes (RoCoF?)
- Using of TDS assessment for TSA: controlling of regenerative energy sources like wind turbines; controlling of stability devices like phasor-shifting, grid-forming power electronics
- Support of reactive and real power flow controlling: Slower expansion of transient disturbances through grids for stabilization with (comparably slow) primary control

Acronyms

CCT	critical clearing time
EAC	equal area criterion
IBB	infinite bus bar
ODE	ordinary differential equation
SG	synchronous generator
SMIB	single machine infinite bus
TDS	time domain solution

Symbols

Complete list of Symbols

H_{gen}	s	inertia constant of a synchronous generator (SG)
P	W	Power; electrical or mechanical

Appendix

A	Code	XV
A.1	Model functions	XV
A.2	Main model	XV
A.3	Fault models	XXIV
A.4	Additional comparison	XXIX
B	Additional	XXXIII
B.1	Jupyter notebook for development	XXXIII

A Code

A.1 Model functions

A.2 Main model

```
1 #####
2 # Base module with the smib model.
3 # Input of the interested variables delta_0, E_bus, E_gen, P_m, ...,
4 #   fault_start, fault_end, ...
5 # Export of stability, t_cc, delta_cc, t_sim, delta(t_sim), omega(t_sim)
6 #   ,
7 # Condsideration of TDS in just stable and just unstable regime
8 #####
9
10 import matplotlib.pyplot as plt
11 from matplotlib.patches import Polygon
12 import matplotlib as mpl
13 import numpy as np
14 import scipy as sp
15 from scipy.integrate import odeint
16
17 # redefining plot save parameters
18 plt.rcParams.update({
19     "text.usetex": True,
20     "font.family": "serif",
21     "font.serif": ["Charter"],
22     "font.size": 12
23 })
24
25 # uncomment for updating savefig options for latex export
26 # mpl.use("pgf")
27
28 # helping function for calculation with complex numbers
29 def mag_and_angle_to_cmplx(mag, angle):
30     return mag * np.exp(1j * angle)
31
32 def algebraic(delta_gen, fault_on):
33     global E_fd_gen
34     global E_fd_ibb
35     global delta_ibb_init
36     global X_gen, X_line, X_ibb
```

```

35
36 # If the SC is on, the admittance matrix is different.
37 # The SC on busbar 0 is expressed in the admittance matrix as a very
    large admittance (1000000) i.e. a very small impedance.
38 if fault_on:
39     y_adm = np.array([X_fault,
40                       [1j / X_line, -1j / X_line - 1j / X_ibt]])
41 else:
42     y_adm = np.array([[ -1j / X_gen - 1j / X_line, 1j / X_line],
43                       [1j / X_line, -1j / X_line - 1j / X_ibt]])
44
45 # Calculate the inverse of the admittance matrix ( $Y^{-1}$ )
46 y_inv = np.linalg.inv(y_adm)
47
48 # Calculate current injections of the generator and the infinite
    busbar
49 i_inj_gen = mag_and_angle_to_cmplx(E_fd_gen, delta_gen) / (1j *
    X_gen)
50 i_inj_ibt = mag_and_angle_to_cmplx(E_fd_ibt, delta_ibt_init) / (1j *
    X_ibt)
51
52 # Calculate voltages at the bus by multiplying the inverse of the
    admittance matrix with the current injections
53 v_bb_gen = y_inv[0, 0] * i_inj_gen + y_inv[0, 1] * i_inj_ibt
54 v_bb_ibt = y_inv[1, 0] * i_inj_gen + y_inv[1, 1] * i_inj_ibt
55
56 return v_bb_gen
57
58 def P_e(delta, fault_on):
59     # function for determining P_e WITHOUT algebraic help
60     global X_gen
61     global X_line
62     global X_fault
63     global E_fd_gen
64     global E_fd_ibt
65
66     if fault_on:
67         X = 1
68         E_ibt = 0
69     else:
70         X = X_gen + X_line
71         E_ibt = E_fd_ibt
72
73     P_e_gen = E_fd_gen * E_ibt / X * np.sin(delta)
74     return P_e_gen
75
76 def P_e_alg(delta, fault_on):

```

```

77     # function for determing P_e WITH algebraic help
78     global E_fd_gen
79     global X_gen
80
81     v_bb_gen = algebraic(delta, fault_on)
82
83     E_gen_complex = mag_and_angle_to_cmplx(E_fd_gen, delta)
84     P_e_gen = (v_bb_gen * np.conj((E_gen_complex - v_bb_gen) / (1j *
85         X_gen))).real
86     return P_e_gen
87
88 def P_m(omega):
89     # returning the torque of the generator, depending on the rotor
90     speed
91     global P_m_gen
92     global omega_gen_init
93     P_t = P_m_gen / (1 + (omega_gen_init + omega))
94     return P_t
95
96 def get_max_delta(gen_parameters, sim_parameters, alg):
97     init(gen_parameters, sim_parameters)
98
99     area_acc = sp.integrate.quad(P_r_deg, delta_gen_init, delta_0_fault,
100         args=(0, True, alg))
101     area_dec = [0, 0]
102     max_delta = delta_0_fault
103     while abs(area_dec[0]) <= abs(area_acc[0]):
104         area_dec = sp.integrate.quad(P_r_deg, delta_0_fault, max_delta,
105             args=(0, True, alg))
106         max_delta = max_delta + 0.01
107
108     return max_delta
109
110 def get_delta_0(gen_parameters, sim_parameters, alg):
111     init(gen_parameters, sim_parameters)
112     x_rad = np.linspace(0, np.pi/2, 360)
113     delta = -1
114     for x in x_rad:
115         if abs(P_r_deg(x, 0, False, alg)) <= 0.01:
116             delta = x
117
118     return delta
119
120 # function for using odeint as ode-solver
121 def ODE_system(state, t, fault_start, fault_end, alg):
122
123     omega, delta = state

```

```

120
121     global H_gen
122     global E_fd_gen
123     global E_fd_abb
124     global X_gen
125     global X_line
126     global fn
127
128     if fault_start <= t < fault_end:
129         fault_on = True
130         # P_e_conv = P_e(E_fd_gen, 0, X_gen, delta)
131     else:
132         fault_on = False
133         # P_e_conv = P_e(E_fd_gen, E_fd_abb, X_gen + X_line, delta)
134
135     # including time dependent solving of algebraic equations
136     if alg:
137         P_e_gen = P_e_alg(delta, fault_on)
138     else:
139         P_e_gen = P_e(delta, fault_on)
140
141     d_omega_dt = 1 / (2 * H_gen) * (P_m(omega) - P_e_gen)
142     d_delta_dt = omega * 2 * np.pi * fn
143
144     return [d_omega_dt, d_delta_dt]
145
146 # functions for determining the critical clearing time
147 def P_r_deg(delta, omega, fault_on, alg):
148     # determining the P_e curve under input in degrees
149     if alg:
150         P_r = P_e_alg(delta, fault_on) - P_m(omega)
151     else:
152         P_r = P_e(delta, fault_on) - P_m(omega)
153     return P_r
154
155 def P_t_deg(x):
156     # determining the P_t curve under input in degrees
157     global P_m_gen
158
159     return P_m_gen*np.ones(np.size(x))
160
161 def stability_eac(delta_0, delta_act, omega_act, delta_max, alg):
162     # global delta_new, omega_new
163
164     # Compare the acceleration area until the given delta and compare it
165     # to the braking area left until the dynamic stability point is
166     # passed

```

```

165     area_acc = sp.integrate.quad(P_r_deg, delta_0, delta_act, args=(
166         omega_act, True, alg))
167
168     area_dec = sp.integrate.quad(P_r_deg, delta_act, delta_max, args=(
169         omega_act, (not clearing), alg))
170
171     if abs(area_acc[0]) < abs(area_dec[0]): # True: stable, False: NOT
172         stable
173         return True
174     else:
175         return False
176
177 def determine_cct(t_sim, delta, omega, delta_0, alg):
178     # t_sim and delta are result arrays
179     # delta_0 is the initial angle delta of the stable system pre-fault
180
181     # Save current time and delta at time point i; iterate through i to
182     # test any given time until stability can't be remained; delta_cc
183     # and t_cc is the angle and time at the last stable point
184     global delta_max_fault
185     if clearing:
186         delta_max = np.pi - delta_0
187     else:
188         delta_max = delta_max_fault
189
190     i = 0
191     t_cc, delta_cc, omega_cc = -1, -1, -1
192
193     while stability_eac(delta_0, delta[i], omega[i], delta_max, alg) and
194         i < np.size(t_sim)-1 and delta[i] < delta_max_fault:
195         t_cc = t_sim[i]
196         delta_cc = delta[i]
197         omega_cc = omega[i]
198         i = i + 1
199
200     if t_cc < 0:
201         return False, -1, -1, -1
202     else:
203         if clearing:
204             return True, t_cc, delta_cc, omega_cc
205         else:
206             return True, t_cc, delta_0_fault, omega_gen_init
207
208 # execution functions for simulation
209 def do_sim(gen_parameters, sim_parameters, alg):
210     init(gen_parameters, sim_parameters)
211
212     # setup simulation inputs

```

```

206     t_sim = np.arange(sim_parameters["t_start"], sim_parameters["t_end
        "], sim_parameters["t_step"])
207     initial_conditions = [gen_parameters["omega_gen_init"],
        gen_parameters["delta_gen_init"]]
208
209     delta_0 = gen_parameters["delta_gen_init"]
210     # delta_max = np.pi - delta_0
211
212     for i in range(1,4,1):
213         if i == 1: # first TDS with no fault-clearing
214             # solve ODE with python solver
215             solution = odeint(ODE_system, initial_conditions, t_sim,
                args=(sim_parameters["fault_start"], sim_parameters["
                fault_end"], alg))
216             stability, t_cc, delta_cc, omega_cc = determine_cct(t_sim,
                solution[:, 1], solution[:, 0], delta_0, alg)
217         elif i == 2: # second TDS with fault clearing just right
218             # solve ODE with python solver
219             fault_end = t_cc - 5 * sim_parameters["t_step"]
220             solution_stable = odeint(ODE_system, initial_conditions,
                t_sim, args=(sim_parameters["fault_start"], fault_end,
                alg))
221         elif i == 3: # second TDS with fault clearing just NOT right
222             fault_end = t_cc + 2 * sim_parameters["t_step"]
223             solution_unstable = odeint(ODE_system, initial_conditions,
                t_sim, args=(sim_parameters["fault_start"], fault_end,
                alg))
224
225     return stability, t_cc, delta_cc, t_sim, solution_stable,
        solution_unstable
226
227 def do_sim_simple(gen_parameters, sim_parameters, alg):
228     init(gen_parameters, sim_parameters)
229
230     # setup simulation inputs
231     t_sim = np.arange(t_start, t_end, t_step)
232     initial_conditions = [omega_gen_init, delta_gen_init]
233
234     delta_0 = delta_gen_init
235
236     solution = odeint(ODE_system, initial_conditions, t_sim, args=(
        fault_start, fault_end, alg))
237     stability, t_cc, delta_cc, omega_cc = determine_cct(t_sim, solution
       [:, 1], solution[:, 0], delta_0, alg)
238
239     return stability, t_cc, delta_cc, t_sim, solution
240

```



```

241 def init(gen_parameters, sim_parameters):
242     global fn, H_gen, X_gen, X_ibt, X_line, X_fault, E_fd_gen, E_fd_ibt,
        P_m_gen, omega_gen_init, delta_gen_init, delta_ibt_init,
        t_start, t_end, t_step, fault_start, fault_end, clearing
243
244     fn = gen_parameters["fn"]
245     H_gen = gen_parameters["H_gen"]
246     X_gen = gen_parameters["X_gen"]
247     X_ibt = gen_parameters["X_ibt"]
248     X_line = gen_parameters["X_line"]
249     X_fault = gen_parameters["X_fault"]
250
251     E_fd_gen = gen_parameters["E_fd_gen"]
252     E_fd_ibt = gen_parameters["E_fd_ibt"]
253     P_m_gen = gen_parameters["P_m_gen"]
254
255     omega_gen_init = gen_parameters["omega_gen_init"]
256     delta_gen_init = gen_parameters["delta_gen_init"]
257     delta_ibt_init = gen_parameters["delta_ibt_init"]
258
259     t_start = sim_parameters["t_start"]
260     t_end = sim_parameters["t_end"]
261     t_step = sim_parameters["t_step"]
262
263     fault_start = sim_parameters["fault_start"]
264     fault_end = sim_parameters["fault_end"]
265     clearing = sim_parameters["clearing"]
266
267     # assessment of delta_0 and delta_max in fault case
268     x_rad = np.linspace(0, np.pi, 360)
269     i = 0
270     global delta_0_fault, delta_max_fault
271     delta_0_fault = np.pi
272     delta_max_fault = np.pi
273     while i < np.size(x_rad)/2:
274         if abs(P_r_deg(x_rad[i], 0, True, True)) < 0.01:
275             delta_0_fault = x_rad[i]
276             delta_max_fault = np.pi - delta_0_fault
277         i = i + 1
278
279     return
280
281 if __name__ == "__main__":
282     # setup simulation inputs
283     gen_parameters = {
284         "fn": 60,
285         "H_gen": 3.5,

```

```

286     "X_gen":      0.2,
287     "X_ibt":      0.1,
288     "X_line":     0.65,
289     "X_fault":    0.0001,
290
291     "E_fd_gen":   1.075,
292     "E_fd_ibt":   1.033,
293     "P_m_gen":    1998/2200,
294
295     "omega_gen_init": 0,
296     "delta_gen_init": np.deg2rad(50.9),
297     "delta_ibt_init": np.deg2rad(0)
298 }
299
300 sim_parameters = {
301     "t_start":     -1,
302     "t_end":       5,
303     "t_step":      0.001,
304
305     "fault_start": 0,
306     "fault_end":   5,
307     "clearing":    True
308 }
309
310 gen_parameters["X_fault"] = [(-1j / gen_parameters["X_gen"] - 1j /
311     gen_parameters["X_line"]) + 1000000, 1j / gen_parameters["X_line"]
312     "]
313
314 # Execution of simulation
315 alg = True
316 stability, t_cc, delta_cc, t_sim, solution_stable, solution_unstable
317     = do_sim(gen_parameters, sim_parameters, alg)
318
319 # Evaluation of results
320 print('t_cc:\t\t' + str(round(t_cc, 3)) + ' s')
321 print('delta_cc:\t' + str(round(np.rad2deg(delta_cc), 1)) + ' deg')
322
323 delta_stable = solution_stable[:,1]
324 omega_stable = solution_stable[:,0]
325
326 #####
327 # Plot stable result
328 #####
329 fig, axs = plt.subplots(2, 1, figsize=(6,8), sharex=True)
330
331 # determine the boundary angles
332 delta_0 = delta_gen_init # delta_gen_init

```

```

330     delta_max = np.pi - delta_0
331
332     # calculation of P_e_pre, P_e_post, and P_t
333     x_deg = np.linspace(0, 180) # linear vector for plotting in deg
334     x_rad = np.linspace(0, np.pi) # linear vector for calculation in rad
335     P_e_pre = P_e_alg(x_rad, False)
336     P_e_post = P_e_alg(x_rad, True)
337     P_t = P_t_deg(x_rad)
338
339     plt.subplots_adjust(hspace=.0)
340
341     #####
342     # ax1
343     #####
344     axs[0].plot(x_deg, P_e_pre, '-', linewidth=2, label='$P_{\mathrm{e}}$
        pre-fault')
345     # axs[0].plot(x_deg, P_e_post, '-', linewidth=2, label='$P_{\mathrm{e}}$
        }$ post-fault')
346     axs[0].plot(x_deg, P_t, '-', linewidth=2, label='$P_{\mathrm{T}}$ of
        the turbine')
347     axs[0].set_ylim(bottom=0)
348     delta_0_deg = np.rad2deg(delta_0)
349     delta_max_deg = np.rad2deg(delta_max)
350     delta_c_deg = np.rad2deg(delta_cc)
351     # axs[0].set_xticks([0, 180, delta_0_deg, delta_c_deg, delta_max_deg
        ], labels=['0', '180', '$\delta_{\mathrm{0}}$', '$\delta_{\mathrm{c}}$
        }$', '$\delta_{\mathrm{max}}$'])
352
353     ix1 = np.linspace(delta_0_deg, delta_c_deg)
354     iy1 = P_e_alg(np.deg2rad(ix1), True)
355     axs[0].fill_between(ix1, iy1, P_m_gen, facecolor='0.9', edgecolor='
        0.5')
356
357     # Make the shaded region for area_dec, https://matplotlib.org/stable
        /gallery/lines\_bars\_and\_markers/fill\_between\_demo.html
358     ix2 = np.linspace(delta_c_deg, delta_max_deg) # -> does this have to
        be in rad or in deg?
359     iy2 = P_e_alg(np.deg2rad(ix2), False)
360     axs[0].fill_between(ix2, iy2, P_m_gen, facecolor='0.9', edgecolor='
        0.5')
361     axs[0].grid()
362     axs[0].legend()
363     axs[0].set_ylabel('power in pu')
364
365     #####
366     # ax2
367     #####

```

```

368     axs[1].plot(np.rad2deg(delta_stable), t_sim, label='delta')
369     fig.gca().invert_yaxis()
370     # axs[1].axhline(y=fault_end, linestyle='--', label='clearing of
        fault')
371     axs[1].grid()
372     axs[1].set_ylabel('time in s')
373     axs[1].legend()
374     plt.ylim(top=-.5)
375     plt.xlim(left=0, right=180)
376     plt.xlabel('power angle $\delta$ in deg')
377
378     plt.suptitle('Stable scenario')
379     plt.show()

```

A.3 Fault models

Fault 2

```

1 #####
2 # simulation of fault 2
3 # t_cc, delta_cc and some plots
4 #
5 # scenario: partly line fault (P_e = XX * P_e), clearing mode (around
        t_cc)
6 #####
7
8 import matplotlib.pyplot as plt
9 from matplotlib.patches import Polygon
10 import matplotlib as mpl
11 import numpy as np
12 import scipy as sp
13 from scipy.integrate import odeint
14
15 import smib_model as sim
16
17 # redefining plot save parameters
18 plt.rcParams.update({
19     "text.usetex": True,
20     "font.family": "serif",
21     "font.serif": ["Charter"],
22     "font.size": 12
23 })
24
25 # uncomment for updating savefig options for latex export

```

```

26 # mpl.use("pgf")
27
28 def init(gen_parameters, sim_parameters):
29     global fn, H_gen, X_gen, X_ibt, X_line, X_fault, E_fd_gen, E_fd_ibt,
        P_m_gen, omega_gen_init, delta_gen_init, delta_ibt_init,
        t_start, t_end, t_step, fault_start, fault_end, clearing
30
31     fn = gen_parameters["fn"]
32     H_gen = gen_parameters["H_gen"]
33     X_gen = gen_parameters["X_gen"]
34     X_ibt = gen_parameters["X_ibt"]
35     X_line = gen_parameters["X_line"]
36     X_fault = gen_parameters["X_fault"]
37
38     E_fd_gen = gen_parameters["E_fd_gen"]
39     E_fd_ibt = gen_parameters["E_fd_ibt"]
40     P_m_gen = gen_parameters["P_m_gen"]
41
42     omega_gen_init = gen_parameters["omega_gen_init"]
43     delta_gen_init = gen_parameters["delta_gen_init"]
44     delta_ibt_init = gen_parameters["delta_ibt_init"]
45
46     t_start = sim_parameters["t_start"]
47     t_end = sim_parameters["t_end"]
48     t_step = sim_parameters["t_step"]
49
50     fault_start = sim_parameters["fault_start"]
51     fault_end = sim_parameters["fault_end"]
52     clearing = sim_parameters["clearing"]
53
54     return
55
56 if __name__ == "__main__":
57     # setup simulation inputs
58     gen_parameters = {
59         "fn": 60,
60         "H_gen": 3.5,
61         "X_gen": 0.2,
62         "X_ibt": 0.1,
63         "X_line": 0.65,
64         "X_fault": 0.0001,
65
66         "E_fd_gen": 1.075,
67         "E_fd_ibt": 1.033,
68         "P_m_gen": 1998/2200,
69
70         "omega_gen_init": 0,

```

```

71     "delta_gen_init": np.deg2rad(50.9),
72     "delta_ibb_init": np.deg2rad(0)
73 }
74
75 sim_parameters = {
76     "t_start":      -1,
77     "t_end":        2,
78     "t_step":       0.001,
79
80     "fault_start":  0,
81     "fault_end":    1,
82     "clearing":     True
83 }
84
85 gen_parameters["X_fault"] = [(-1j / gen_parameters["X_gen"] - 1j*3 /
86     gen_parameters["X_line"]), 1j / gen_parameters["X_line"]]
87
88 init(gen_parameters, sim_parameters)
89
90 # Execution of simulation
91 alg = True
92 stability, t_cc, delta_cc, t_sim, solution_stable, solution_unstable
93     = sim.do_sim(gen_parameters, sim_parameters, alg)
94
95 # Evaluation of results
96 print('t_cc:\t\t' + str(round(t_cc, 3)) + ' s')
97 print('delta_cc:\t\t' + str(round(np.rad2deg(delta_cc), 1)) + ' deg')
98
99 delta_stable = solution_stable[:,1]
100 delta_unstable = solution_unstable[:,1]
101
102 #####
103 # Plot unstable result
104 #####
105 fig, axs = plt.subplots(2, 1, figsize=(6,8), sharex=True)
106
107 # determine the boundary angles
108 delta_0 = delta_gen_init # delta_gen_init
109 delta_max = np.pi - delta_0
110
111 # calculation of P_e_pre, P_e_post, and P_t
112 x_deg = np.linspace(0, 180) # linear vector for plotting in deg
113 x_rad = np.linspace(0, np.pi) # linear vector for calculation in rad
114 P_e_pre = sim.P_e_alg(x_rad, False)
115 P_e_post = sim.P_e_alg(x_rad, True)
116 P_t = sim.P_t_deg(x_rad)

```

```

116 plt.subplots_adjust(hspace=.0)
117
118 #####
119 # ax1
120 #####
121 axs[0].plot(x_deg, P_e_pre, '-', linewidth=2, label='$P_{\mathrm{e}}$
    pre-fault')
122 axs[0].plot(x_deg, P_e_post, '-', linewidth=2, label='$P_{\mathrm{e}}$
    post-fault')
123 axs[0].plot(x_deg, P_t, '-', linewidth=2, label='$P_{\mathrm{T}}$ of
    the turbine')
124 axs[0].set_ylim(bottom=0)
125 delta_0_deg = np.rad2deg(delta_0)
126 delta_max_deg = np.rad2deg(delta_max)
127 delta_c_deg = np.rad2deg(delta_cc)
128 # axs[0].set_xticks([0, 180, delta_0_deg, delta_c_deg, delta_max_deg
    ], labels=['0', '180', '$\delta_{\mathrm{0}}$', '$\delta_{\mathrm{c}}$
    $', '$\delta_{\mathrm{max}}$'])
129
130 ix1 = np.linspace(delta_0_deg, delta_c_deg)
131 iy1 = sim.P_e_alg(np.deg2rad(ix1), True)
132 axs[0].fill_between(ix1, iy1, P_m_gen, facecolor='0.9', edgecolor='
    0.5')
133
134 # Make the shaded region for area_dec, https://matplotlib.org/stable
    /gallery/lines\_bars\_and\_markers/
    fill\_between\_demo.html
135 ix2 = np.linspace(delta_c_deg, delta_max_deg) # -> does this have to
    be in rad or in deg?
136 iy2 = sim.P_e_alg(np.deg2rad(ix2), False)
137 axs[0].fill_between(ix2, iy2, P_m_gen, facecolor='0.9', edgecolor='
    0.5')
138 axs[0].grid()
139 axs[0].legend()
140 axs[0].set_ylabel('power in pu')
141
142 #####
143 # ax2
144 #####
145 axs[1].plot(np.rad2deg(delta_stable), t_sim, label='delta')
146 fig.gca().invert_yaxis()
147 axs[1].axhline(y=t_cc, linestyle='--', label='clearing of fault')
148 axs[1].grid()
149 axs[1].set_ylabel('time in s')
150 axs[1].legend()
151 plt.ylim(top=-.1)
152 plt.xlim(left=0, right=180)
153 plt.xlabel('power angle $\delta$ in deg')

```

```

154
155 plt.suptitle('Stable scenario - fault 2')
156 # plt.savefig('plots/fault1_stable.pgf')
157 plt.show()
158 # plt.close()
159
160 #####
161 # Plot UNstable result
162 #####
163 fig, axs = plt.subplots(2, 1, figsize=(6,8), sharex=True)
164
165 # determine the boundary angles
166 delta_0 = delta_gen_init # delta_gen_init
167 delta_max = np.pi - delta_0
168
169 # calculation of P_e_pre, P_e_post, and P_t
170 x_deg = np.linspace(0, 180) # linear vector for plotting in deg
171 x_rad = np.linspace(0, np.pi) # linear vector for calculation in rad
172 P_e_pre = sim.P_e_alg(x_rad, False)
173 P_e_post = sim.P_e_alg(x_rad, True)
174 P_t = sim.P_t_deg(x_rad)
175
176 plt.subplots_adjust(hspace=.0)
177
178 #####
179 # ax1
180 #####
181 axs[0].plot(x_deg, P_e_pre, '-', linewidth=2, label='$P_{\mathrm{e}}$
    pre-fault')
182 axs[0].plot(x_deg, P_e_post, '-', linewidth=2, label='$P_{\mathrm{e}}$
    post-fault')
183 axs[0].plot(x_deg, P_t, '-', linewidth=2, label='$P_{\mathrm{T}}$ of
    the turbine')
184 axs[0].set_ylim(bottom=0)
185 delta_0_deg = np.rad2deg(delta_0)
186 delta_max_deg = np.rad2deg(delta_max)
187 delta_c_deg = np.rad2deg(delta_cc)
188 # axs[0].set_xticks([0, 180, delta_0_deg, delta_c_deg, delta_max_deg
    ], labels=['0', '180', '$\delta_{\mathrm{0}}$', '$\delta_{\mathrm{c}}$',
    '$\delta_{\mathrm{max}}$'])
189
190 ix1 = np.linspace(delta_0_deg, delta_c_deg)
191 iy1 = sim.P_e_alg(np.deg2rad(ix1), True)
192 axs[0].fill_between(ix1, iy1, P_m_gen, facecolor='0.9', edgecolor='
    0.5')
193

```



```

194 # Make the shaded region for area_dec, https://matplotlib.org/stable
    /gallery/lines\_bars\_and\_markers/fill\_between\_demo.html
195 ix2 = np.linspace(delta_c_deg, delta_max_deg) # -> does this have to
    be in rad or in deg?
196 iy2 = sim.P_e_alg(np.deg2rad(ix2), False)
197 axs[0].fill_between(ix2, iy2, P_m_gen, facecolor='0.9', edgecolor='
    0.5')
198 axs[0].grid()
199 axs[0].legend()
200 axs[0].set_ylabel('power in pu')
201
202 #####
203 # ax2
204 #####
205 axs[1].plot(np.rad2deg(delta_unstable), t_sim, label='delta')
206 fig.gca().invert_yaxis()
207 axs[1].axhline(y=t_cc, linestyle='--', label='clearing of fault')
208 axs[1].grid()
209 axs[1].set_ylabel('time in s')
210 axs[1].legend()
211 plt.ylim(top=-.1)
212 plt.xlim(left=0, right=180)
213 plt.xlabel('power angle $\delta$ in deg')
214
215 plt.suptitle('Unstable scenario - fault 2')
216 # plt.savefig('plots/fault1_unstable.pgf')
217 plt.show()
218 # plt.close()

```

A.4 Additional comparison

Generator parameter

```

1 #####
2 # Comparison of different machine parameters and operational points.
3 #####
4
5 import matplotlib.pyplot as plt
6 from matplotlib.patches import Polygon
7 import matplotlib as mpl
8 import numpy as np
9 import scipy as sp
10 from scipy.integrate import odeint
11 # import tikzplotlib as tp

```

```

12
13 import smib_model as sim
14
15 # redefining plot save parameters
16 plt.rcParams.update({
17     "text.useTex": True,
18     "font.family": "serif",
19     "font.serif": ["Charter"],
20     "font.size": 12
21 })
22
23 # # update savefig options for latex export
24 # mpl.use("pgf")
25
26 # comparing different H_gen in t_cc, delta_cc and their TDS when a fault
    is ON
27
28
29 # comparing different degree of utilization (P_e / P_e_max OR delta_P;
    direct correlated: delta_0)
30
31
32 def init(gen_parameters, sim_parameters):
33     global fn, H_gen, X_gen, X_abb, X_line, X_fault, E_fd_gen, E_fd_abb,
        P_m_gen, omega_gen_init, delta_gen_init, delta_abb_init,
        t_start, t_end, t_step, fault_start, fault_end
34
35     fn = gen_parameters["fn"]
36     H_gen = gen_parameters["H_gen"]
37     X_gen = gen_parameters["X_gen"]
38     X_abb = gen_parameters["X_abb"]
39     X_line = gen_parameters["X_line"]
40     X_fault = gen_parameters["X_fault"]
41
42     E_fd_gen = gen_parameters["E_fd_gen"]
43     E_fd_abb = gen_parameters["E_fd_abb"]
44     P_m_gen = gen_parameters["P_m_gen"]
45
46     omega_gen_init = gen_parameters["omega_gen_init"]
47     delta_gen_init = gen_parameters["delta_gen_init"]
48     delta_abb_init = gen_parameters["delta_abb_init"]
49
50     t_start = sim_parameters["t_start"]
51     t_end = sim_parameters["t_end"]
52     t_step = sim_parameters["t_step"]
53
54     fault_start = sim_parameters["fault_start"]

```

```

55     fault_end = sim_parameters["fault_end"]
56
57     return
58
59 if __name__ == "__main__":
60     # setup simulation inputs
61     gen_parameters = {
62         "fn": 60,
63         "H_gen": 3.5,
64         "X_gen": 0.2,
65         "X_ibt": 0.1,
66         "X_line": 0.65,
67         "X_fault": 0.001,
68
69         "E_fd_gen": 1.075,
70         "E_fd_ibt": 1.033,
71         "P_m_gen": 1998/2200,
72
73         "omega_gen_init": 0,
74         "delta_gen_init": np.deg2rad(50.9),
75         "delta_ibt_init": np.deg2rad(0)
76     }
77
78     sim_parameters = {
79         "t_start": 0,
80         "t_end": 5,
81         "t_step": 0.001,
82
83         "fault_start": 0,
84         "fault_end": 5,
85         "clearing": True
86     }
87
88     gen_parameters["X_fault"] = [(-1j / gen_parameters["X_gen"] - 1j /
89         gen_parameters["X_line"]) + 1000000, 1j / gen_parameters["X_line"]
90         "]
91
92     # init(gen_parameters, sim_parameters)
93
94     # Execution of simulations and savon in t_ccs array
95     alg = True
96
97     sim.init(gen_parameters, sim_parameters)
98     P_e_max = sim.P_e_alg(np.pi/2, False)
99
100     H = np.arange(0.1, 12.5, 0.5) # [2.5, 3, 3.5, 4, 4.5, 5]
101     delta_P = [0.3, 0.5, 0.7, 0.9, 1.1]

```

```

100     t_ccs = np.zeros((np.size(delta_P), np.size(H)))
101     i = 0
102     for delta_P_new in delta_P:
103         gen_parameters["P_m_gen"] = delta_P_new
104         gen_parameters["delta_gen_init"] = sim.get_delta_0(
105             gen_parameters, sim_parameters, alg)
106         z = 0
107         for H_new in H:
108             gen_parameters["H_gen"] = H_new
109
110             stability, t_cc, delta_cc, t_sim, solution = sim.
111                 do_sim_simple(gen_parameters, sim_parameters, alg)
112             print(t_cc)
113
114             if stability:
115                 t_ccs[i, z] = t_cc
116             else:
117                 t_ccs[i, z] = -1
118                 z = z + 1
119
120             plt.plot(H, t_ccs[i, :], label=("$\Delta P = $ " + str(round(
121                 delta_P_new/P_e_max*100, 1)) + " $\%$"))
122             i = i + 1
123
124     plt.legend()
125     plt.xlabel("$H_{\mathrm{gen}}$ in $\mathrm{s}$")
126     plt.ylabel("$CCT$ in $\mathrm{s}$")
127     # plt.title("Influence from $H_{\mathrm{gen}}$ and $\Delta P$ on CCT")
128     plt.grid()
129     # figure = plt.gcf() # get current figure
130     # figure.set_size_inches(8, 6)
131     # plt.savefig('plots/parameter_comparison.pgf', dpi=300)
132     plt.show()
133     # plt.close()

```

B Additional

B.1 Jupyter notebook for development