Institute for electrical energy systems
Chairman: Univ.-Prof. Dr.-Ing. Matthias Luther

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

[March 21, 2024 at 11:40 – preliminary version 0.1]

Todo list

Author's declaration

I certify that I have prepared this Student Research Paper without outside help and without using sources other than those specified and that the thesis has not been submitted in the same or a similar form to any other examination authority and has not been accepted by them as part of an examination. All statements that have been copied verbatim or in spirit are marked as such.

Erlangen, March 21, 2024

Maximilian Köhler, B. Eng.

Note:

For reasons of readability, the generic masculine is primarily used in this Student Research Paper. Female and other gender identities are explicitly included where this is necessary for the statement.

Abstract

Abstract in english.

Write abstract in english

Kurzfassung

Kurzfassung in deutscher Sprache.

Kurzfassung in deutsch schreiben

Contents

List of Figures

List of Tables

List of Listings

[MK1]: Warum hat das einen anderen Zeilenabstand als alle anderen Verzeichnisse?

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 [1] and other standard literature like [2]–[6]. 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; .

Insert senseful schematic picture/graph

The goal of this Student Research Paper is the implementation of a CCT determing Python algorithm for a 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

Input of basic knowledge for system modelling; Maybe supplementary knowledge

General sources in terms of standard literature: [2]–[5]

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 $\Delta\omega$:

$$\frac{d\delta}{dt} = \Delta\omega \tag{2.1}$$

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

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 [3])

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.

$$P_{\text{e}} = \frac{E_{\text{p}} \cdot E_{\text{ibb}}}{X_{\text{res}}} \cdot sin(\delta) \tag{2.3}$$

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

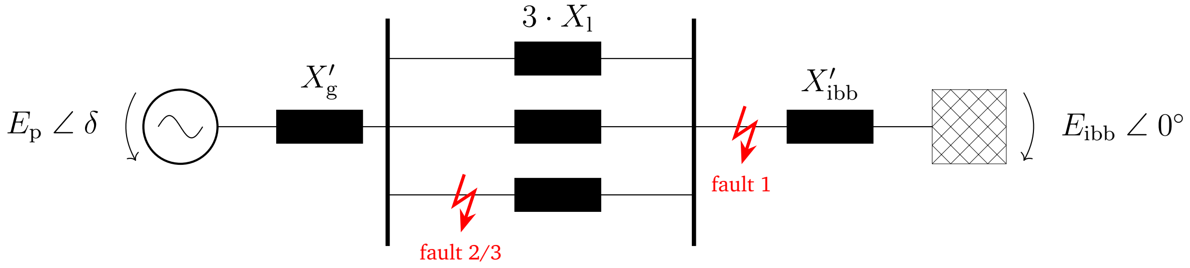


Figure 2.1: Representative circuit of a single machine infinite bus (SMIB) model with pole wheel voltage $E_{\text{p}} \angle \delta$ and infinite bus bar (IBB) voltage $E_{\text{ibb}} \angle 0°$; positions of considered faults 1 to 3 are marked with red lightning arrows

[MK2]: Update and correct SMIB model graphic

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 energys can be calculated through the area under the curve of the power difference $\Delta P = P_{m} - P_{e}$, while the accelerating area is between 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

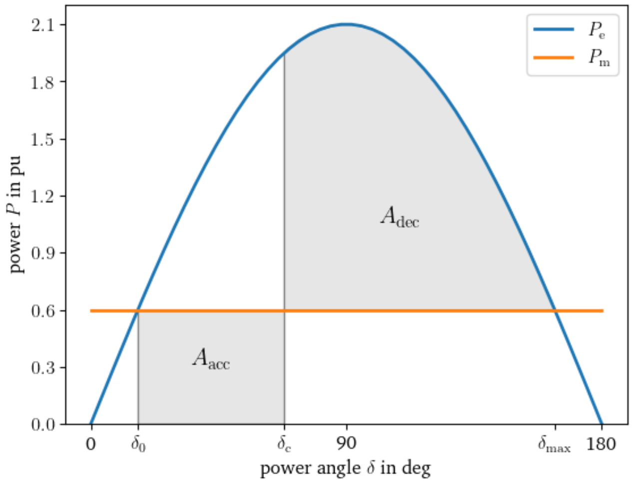


Figure 2.2: Illustrated equal area criterion (EAC) in the P-δ-curve where $A_{acc} = A_{dec}$

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

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

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

With consideration of $\delta_{max} = \pi - \delta_0$, $P_{e,normal} = P_{max} \cdot sin(\delta_0)$, $P_{e,fault} = 0$, and some rearrangements, this leads to the final expression of the critical clearing angle:

$$\delta_{cc} = arccos\big[\, sin(\delta_0) \cdot (\pi - 2 \cdot \delta_0) - cos(\delta_0) \,\big] \tag{2.6}$$

[March 21, 2024 at 11:40 – preliminary version 0.1]

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).

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

Both expressions Equation 2.6 and Equation 2.7 are only valid for clearing faults with an electric power drawing of 0 p.u.. For a partial line or power fault, the expressions tend to complicate more. One have to use $P_{\text{e}} = P_{\text{max, fault}} \cdot sin(\delta)$ before both integrations of the area equations and the swing equation itself. Finally giving two expressions for the critical power angle δ_{cc} and the CCT t_{cc}:

[MK4]: Sinnvoll da den vollständigen Rechenweg in den Anhang mit aufzunehmen?

$$\delta_{\text{cc}} = arccos\left[\frac{P_{\text{m}}}{P_{\text{max}} - P_{\text{max, fault}}} \cdot (\pi - 2 \cdot \delta_0) - \frac{P_{\text{max, fault}}}{P_{\text{max}} - P_{\text{max, fault}}} \cdot cos(\delta_0) - \frac{P_{\text{max}}}{P_{\text{max}} - P_{\text{max, fault}}} \cdot cos(\delta_0)\right] \tag{2.8}$$

$$t_{\text{cc}} = \sqrt{\frac{4H_{\text{gen}} \cdot (\delta_{\text{cc}} - \delta_0)}{\omega \cdot (P - P_{\text{max, fault}} \cdot sin(\delta_{\text{cc}}))}} \tag{2.9}$$

[MK5]: Für den zweiten Fall erscheint mir die Berechnung der CCT etwas hoch -> P_e auch nicht von t abhängig... Muss ich da einfach statt $\delta_c c$ die zeitabhängige δ-Gleichung einsetzen?

2.4 Numerical methods for system modeling

- **solving second order ODEs (explicit)**
- Differentiation explicit/impolicit, 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 resoluted 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

$$f_{\mathrm{t+1}} = f_{\mathrm{t}} + \left(\frac{df}{dt}\right)_{\mathrm{t}} \cdot \Delta t, \tag{2.10}$$

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 modeling

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 of the main model with an example fault, as well as the parameter sets of the three considered faults is included in Appendix B.

3.1 Structure of the CCT assessment

The central interest in the algorithm is to determine the time, until a system failure has to be resolved, so that it can remain stable and synchronized. In general, an enough accurate and easy approach for a single machine infinite bus (SMIB) system is the equal area criterion (EAC). For a more complex and coupled machine system, other approaches are more targeting [8]. Further interest is to determine the associated crictical clearing angle. This is the maximum possible power angle at the CCT, with which the fault can just be cleared into a stable system. At last one is interested in the time domain solution, just shortly before and after the CCT. This shall illustrate the convergent and divergent behavior of the power angle and therefore the rotor speed.

Figure 3.1 illustrates the rough stucture of the numerical calculation. The single processes are further described in section 3.3 for the time resoluted solving with *odeint()*, and section 3.4 for the function *determine_cct()*. The initiialziation and plotting are documented via the complete code in Appendix B. The iterative solving of stable and unstable regie solutions is, neglecting the fault start and ending as initial conditions, an identical procedure and using the same function set as the described TDS.

[1] documentation and manual can be found on *https://scipy.org/* [7], similiar for *matplotlib*, and *numpy* packages

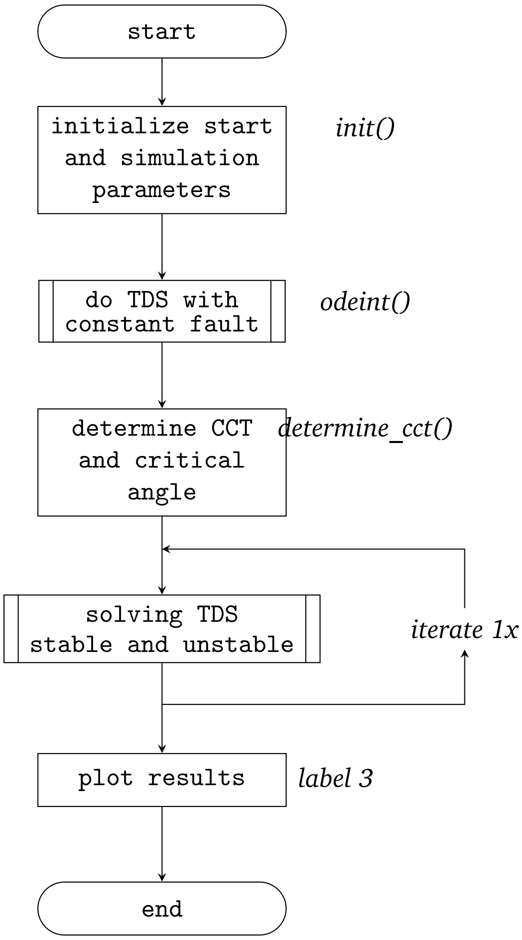


Figure 3.1: Program plan proposal for determining the critical clearing time (CCT) t_{cc}, critical power angle δ_{cc} and the time domain solution (TDS) of the single machine infinite bus (SMIB)-model; including the associated main function name

3.2 Electrical simplifications and scenario setting

3.2.1 Electric networks

The SMIB-model presented in Figure 2.1 has to be devided into the two states *during fault* and *steady state*, and can further be simplified. This leads to a single replacement reactance and is more easy to use in the simulation.

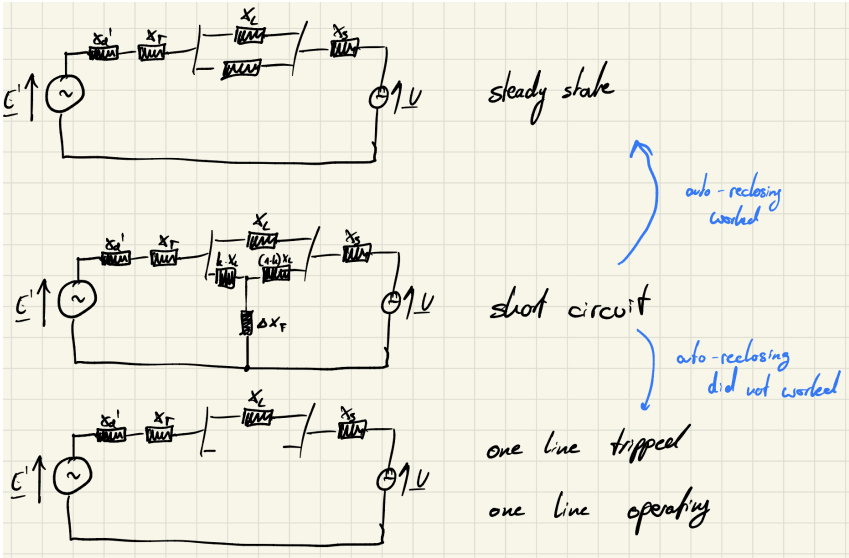


Figure 3.2: Electrical networks for simulation

Figure 3.2 shows the steady state circuit, the system during the fault and the possible network after and with no clearing of the fault. Normally the third state is not necessary. The networks in Figure 3.3 are the represented ones in the simulation, the reactances thus have to be simplified for a targeted simulation.

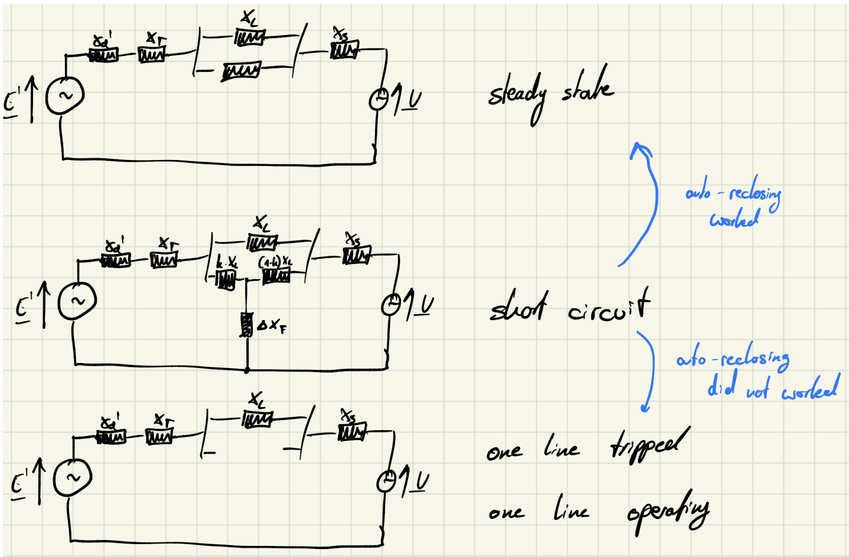


Figure 3.3: Simplified electrical networks

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.

3.2.3 Initial value calculation

For the setting of starting values, the per unit system is preferred. Because of the relative nature of this unit, it acts as generalization and can be applied to concrete examples with known nominal sizes. Generally speaking, $P_{\text{e,max}}$, $P_{\text{e}} = P_{\text{mech}}$, E_{ibb}, E_{gen}, δ_{ibb} and δ_{gen} are needed to be predefined for the simulation.

As the point of interest in most calculations, the voltage at the at the IBB is set to $1\ \text{p.u.}$. Most of the until now presented equations are referring to power angle differences. For a simplified calculation, it is handsome to set the power angle of the IBB to $0°$, thus all the power angle and angle developments dependent on the time are solely related to the absolute power angle of the generator. The maximum electrical power of the generator is arbitrarily set to $1.2\ \text{p.u.}$, the real power extracted from the generator into the grid node is set to $0.9\ \text{p.u.}$. With these predefinitions and both following equations we can calculate the remaining two values.

[MK6]: Values nicht immer korrekt -> Fault 3!; Mit Transformator Reaktanz auch noch einmal anders

$$E_{\text{gen}} = \frac{P_{\text{e,max}} \cdot X}{E_{\text{ibb}} \cdot sin(90°)} \approx 1.14\ \text{p.u.}$$

$$\delta_{\text{gen}} = arcsin\left(\frac{P_{\text{e}} \cdot X}{E_{\text{ibb}} \cdot E_{\text{gen}}}\right) \approx 48.6°$$

The reactances for the different fault scenarios can be derived from the electrical networks in subsection 3.2.1. Therefore the general reactance in operation mode is constituted with the reactance of the generator, the line, and the IBB. For fault 1 the additional fault reactance is getting very high in addition to the generator, thus it can be neglected. Fault 2, considering just a partial line tripping, is increasing the contribution from the line to $\frac{3}{2}$ of its initial value. The other contributions stay consistent. For fault scenario 3 the reactances are consistent whether the fault is present or not. The initial electrical power is reduced to 0.6 p.u. and thus the initial power angle of the generator is set to $30°$.

[MK7]: Ja? Wie ist das wirklich?

[MK8]: delta berechnen

A compromised overview of the initial values and the values in the fault cases is given in section A.1.

3.3 Implementation of the time domain solution

The TDS shall be solved with a python integrated solver, due to the fact that numerical solvin methods are not scope of this paper. The solver *odeint()* from the *scipy*-package is therefore used as preferred algorithm. This requires a time array with all the timesteps of interest and a differential function, which is solved through every time step. Due to the second order nature of the swing-equation and just the possibility to solve first order ones, the equation has to be split up into two first order equations and solved simultaneously. This can be realized via using a solution array instead of a variable.

Looking deeper into the swing equation, both the demanded electrical power from the grid or connected network and the mechanical power put into the roter from the steam turbine have to be calculated at each time step. While the mechnanical power is a bit easier with the dependency

[MK9]: Include the equation for the mechanical power

$$P_{\mathrm{m}} = P,$$

the electrical power is a bit more complex. In order to get a good representation, the algebraic equations describing the connected network, have to be solved at every time step as well.

Describing the algebraic equation system and how to implement in python

What does algebraic or non-algebraic mean?

3.4 Implementation of the equal area criterion

The equal area criterion (EAC) is computed as the name states. It is comparing the accelerated area with the decelerable area, and therefore comparing the stored to the braking (or re-synchronizing) energy. The main function *deteming_cct()* is differentiating between clearing and non-clearing mode. First one is taking the pre-fault status of the connected network also as post-fault condition, thus calculating the clearing time and angle the generatr and the network can remain in the fault state. The non-clearing mode is taking the fault condition as post-fault condition and is calculating a new stable power angle convergent.

Listing 3.1: Main functions for the determination of the critical clearing time (CCT) with the equal area criterion (EAC)

```
def stability_eac(delta_0, delta_act, omega_act, delta_max, alg):
    area_acc = sp.integrate.quad(P_r_deg, delta_0, delta_act, args=(omega_act, True, alg))
    area_dec = sp.integrate.quad(P_r_deg, delta_act, delta_max, args=(omega_act, (not clearing), alg))

    if abs(area_acc[0]) < abs(area_dec[0]): # True: stable, False: NOT stable
        return True
    else:
        return False

def determine_cct(t_sim, delta, omega, delta_0, alg):
    global delta_max_fault
    if clearing:
        delta_max = np.pi - delta_0
    else:
        delta_max = delta_max_fault

    i = 0
    t_cc, delta_cc, omega_cc = -1, -1, -1

    while stability_eac(delta_0, delta[i], omega[i], delta_max, alg) and i < np.size(t_sim)-1 and delta[i] < delta_max_fault:
        t_cc = t_sim[i]
        delta_cc = delta[i]
        omega_cc = omega[i]
        i = i + 1

    if t_cc < 0:
        return False, -1, -1, -1
    else:
        if clearing:
            return True, t_cc, delta_cc, omega_cc
        else:
            return True, t_cc, delta_0_fault, omega_gen_init
```

The main though is iterating through the TDS at each time step, looking if enough braking reserve is left and saving the current time and angle as solution. If the loop continues, the solution is overwritten. As pre-set the solution is negative. This enables a quick understanding of simulation faults or a general unstable initial condition set.

As a helping function *stability_eac()* allows a simple check in the loop. It calculates the currently passed acceleration area, and the until the maximum dynamically stable power angle $\delta_{\max} = \pi - \delta_0$ possible decelerating area, it can compare and state stability or instability at the current time point.

Further functions: Power calculation?

Another possible way would be to check the stability first under the $P-\delta$-curve, gathering the critical angle. After that a simple run through the TDS can deliver the searched CCT. Within this approach the angle spectrum has to be searched in addition to the TDS. This doubled vector searching seemed as easy improvement in comparison to the previous method and was therefore neglected.

4 Results

4.1 Analytical results

The analytical calculation follows the equations from section 2.3. For fault 1 the simplified ones could be used, the more complex and advanced are needed for fault case 2. Therefore the base values for the first two input scenarios are used like in the numerical simulation. For the third one no CCT is calculatable, due to the stable nature of the fault scenario. The results of this calculation are shown in Table 4.1.

[MK10]: Was ist mit nicht vollständigen Fehlern? Gleichungen für analytical beschreiben nichts...

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

	δ_{cc}	t_{cc}
fault 1	65.01°	0.116 s
fault 2	93.99°	0.595 s

4.2 Numerical results

Table 4.2 is summarizing the results for the CCT-calculation of the different set scenarios in section 3.2. Like in the anayltical results section before, the third fault can not be displayed in the context of the first two clearing ones. The maximum reached power angle for fault 3 is $69.3°$, while the new stable power angle is around $48.6°$. At first the system oszillates around this new angle, until the damping factor results in a new stable and steady operation point.

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

Scenario	δ_{cc}	t_{cc}
fault 1	65.9°	0.119 s
fault 2	95.9°	0.394 s

4.2.1 Simulated faults

Looking deeper into the numerical results is possible through plotting the development of the power angle over the time. In addition to that the used energies or respectively areas in the $P - \delta$-curves. Figure 4.1 is looking deeper into the non-clearing fault three.

Because of spacial reasons, the complete plot sets for all faults are included in the section A.2, section A.3 and section A.4.

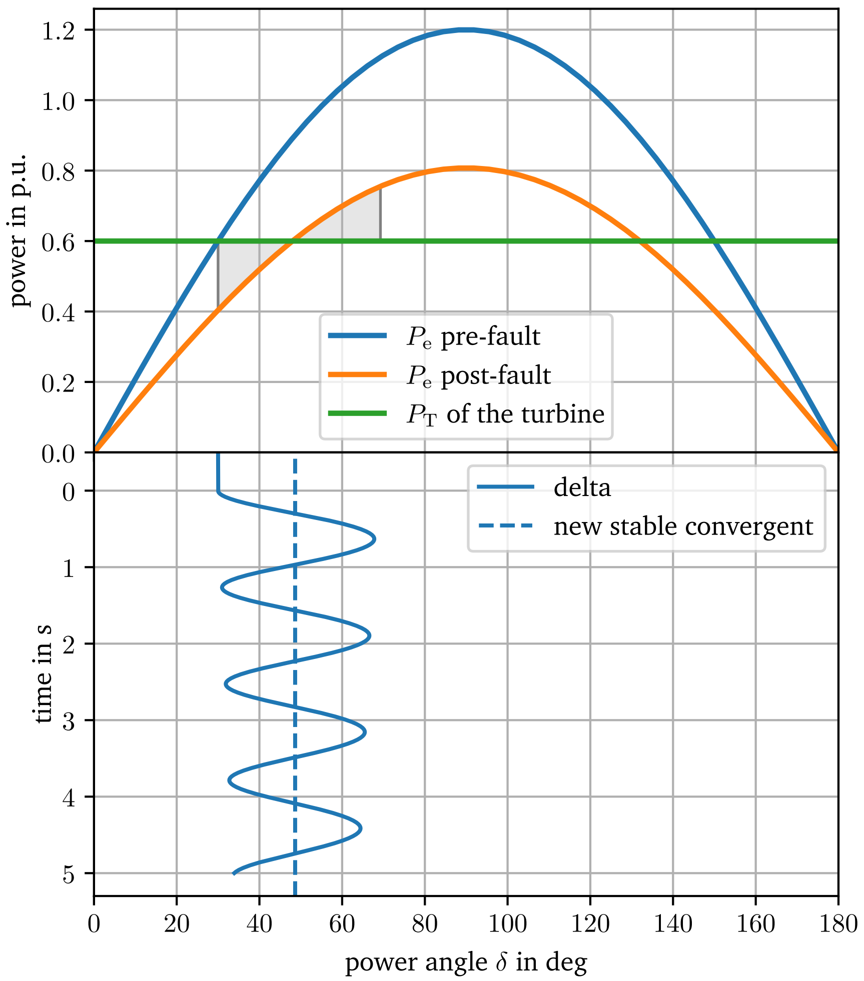


Figure 4.1: Power angle plot of fault 3 in time and power domain; grey areas illustrate the used area under the curve for storing and releasing kinetic energy in/out of the rotor

Both partial sinus functions represent the electrical power demanded from the electrical connected network. During the fault, in this case post-fault as well, is lower in the peak and thus giving another intersection point with the straight horizontal mechanical power from the turbine. The new stable convergent is falling together with this intersection point. The used area is reaching from the starting power angle over the post-fault intersection point to the maximum swept power angle in the time resolution. In this case the fault is starting to occur at the time point 0 s.

For fault one and two the whole area until the maximum dynamic stable operation point, the second intersection point with the pre-fault partial-sinoidal wave, is used. Therefore the critical angle is not at the intersection with the during fault electrical power curve. The unstable scenario, clearing the fault just a few time steps after the CCT leads to a divergent power angle. Clearing just in time shows a stable, oszillating behavior, while resulting in a maximum overswing in the region of the maximum dynamic stable power angle. This behavior can be seen in fault one as well as in fault two, while in fault one the static stable angle cannot be reached, in fault two it is even overstepped.

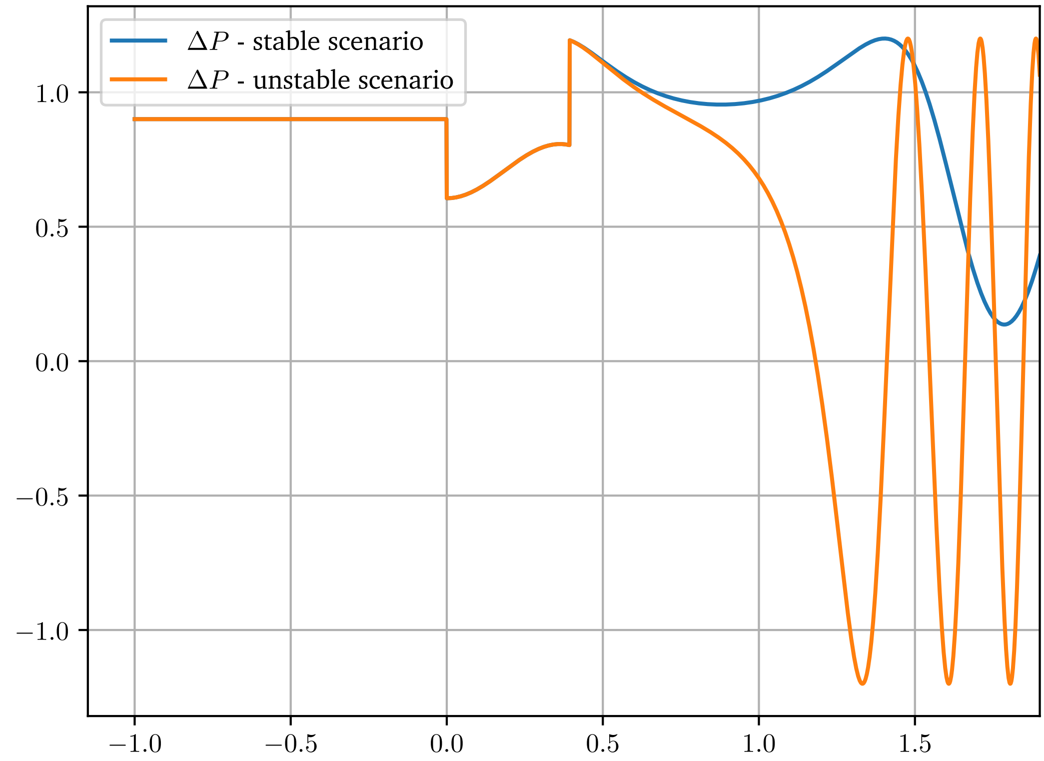


Figure 4.2: Power difference behavior over time for fault 2

4.2.2 Using algebraic calculations vs. non-algebraic

Using the non-algebraic approach in calculating, the TDS seems to have a similar result. Differences can be spotted in the swings of the power difference ΔP and the duration of a swing period of the power angle and rotor angle speed.

Graphics and further look inside needed!

Problem: Completely the same result...

4.2.3 Parameter influence analysis

Other variables changing the CCT of a generator are the power difference ΔP and H_{gen} (see Equation 2.2). The resulting CCT while variing this parameters in the ranges

$$H_{\text{gen}} = [0,\ 12]\ \text{s},\ \text{and}$$
$$\Delta P = [25,\ 91.7]\ \%$$

is shown in Figure 4.3.

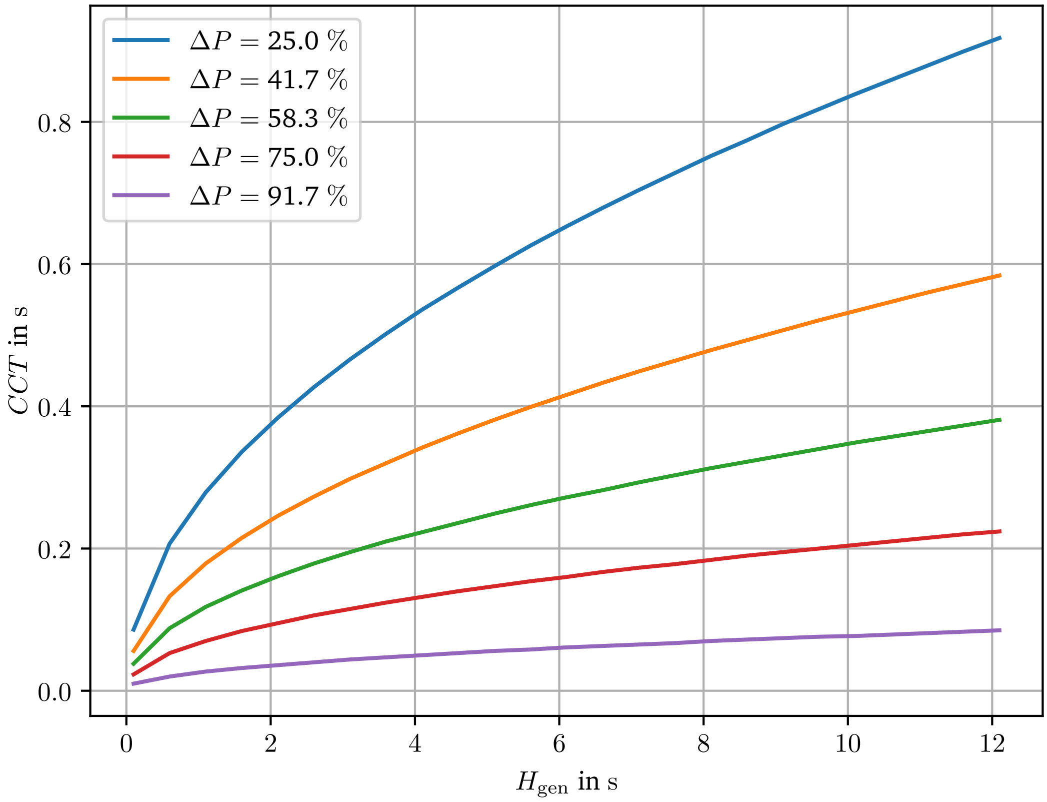


Figure 4.3: Influence of variing the parameters power difference ΔP and H_{gen} on the critical clearing time (CCT)

4.3 Discussion

4.4 Limitations

- Just stable or unstable, not metastable (first swing ok, after that unstable development)
- No damping -> new stable point with low oszillation amplitude in fault 3 is not reached within a few seconds
- Simplified generator model, only one generator. No machine interaction considered. -> Other algorithms
- Differences: Calculation via TDS and area, via area under the curve solely, stability criteria in the TDS (function $P(\delta)$ -> find extreme points and compare them with the maximum power angles; Paper from Ilya: [8])

5 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 [9]).

- 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

Bibliography

[1] "Perspektiven der elektrischen Energieübertragung in Deutschland," VDE Verband der Elektrotechnik Elektronik Informationstechnik e.V., Ed., Frankfurt am Main, Apr. 2019.

[2] J. D. Glover, T. J. Overbye, and M. S. Sarma, "Power system analysis & design," Boston, MA, 2017.

[3] P. S. Kundur and O. P. Malik, *Power System Stability and Control*, Second edition. New York Chicago San Francisco Athens London Madrid Mexico City Milan New Delhi Singapore Sydney Toronto: McGraw Hill, 2022, 948 pp., ISBN: 978-1-260-47354-4.

[4] J. Machowski, Z. Lubosny, J. W. Bialek, and J. R. Bumby, *Power System Dynamics: Stability and Control*, Third edition. Hoboken, NJ, USA: John Wiley, 2020, 1 p., ISBN: 978-1-119-52636-0 978-1-119-52638-4.

[5] D. Oeding and B. R. Oswald, *Elektrische Kraftwerke und Netze*, 8. Auflage. Berlin [Heidelberg]: Springer Vieweg, 2016, 1107 pp., ISBN: 978-3-662-52702-3. DOI: 10.1007/978-3-662-52703-0.

[6] A. J. Schwab, *Elektroenergiesysteme: smarte Stromversorgung im Zeitalter der Energiewende*, 7. Auflage. Berlin [Heidelberg]: Springer Vieweg, 2022, 871 pp., ISBN: 978-3-662-64773-8.

[7] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, Mar. 2, 2020, ISSN: 1548-7091, 1548-7105. DOI: 10.1038/s41592-019-0686-2. [Online]. Available: https://www.nature.com/articles/s41592-019-0686-2 (visited on 01/13/2024).

[8] S. Batchu, Y. Raghuvamsi, and K. Teeparthi, "A Comparative Study on Equal Area Criterion Based Methods for Transient Stability Assessment in Power Systems," in *2022 22nd National Power Systems Conference (NPSC)*, New Delhi, India: IEEE, Dec. 17, 2022, pp. 124–129, ISBN: 978-1-66546-202-0. DOI: 10.1109/NPSC57038.2022.10069303. [Online]. Available: https://ieeexplore.ieee.org/document/10069303/ (visited on 12/14/2023).

[9] Z. Gao, W. Du, and H. Wang, “Transient stability analysis of a grid-connected type-4 wind turbine with grid-forming control during the fault,” *International Journal of Electrical Power & Energy Systems*, vol. 155, p. 109 514, Jan. 2024, ISSN: 01420615. DOI: `10.1016/j.ijepes.2023.109514`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S0142061523005719` (visited on 12/15/2023).

Appendix

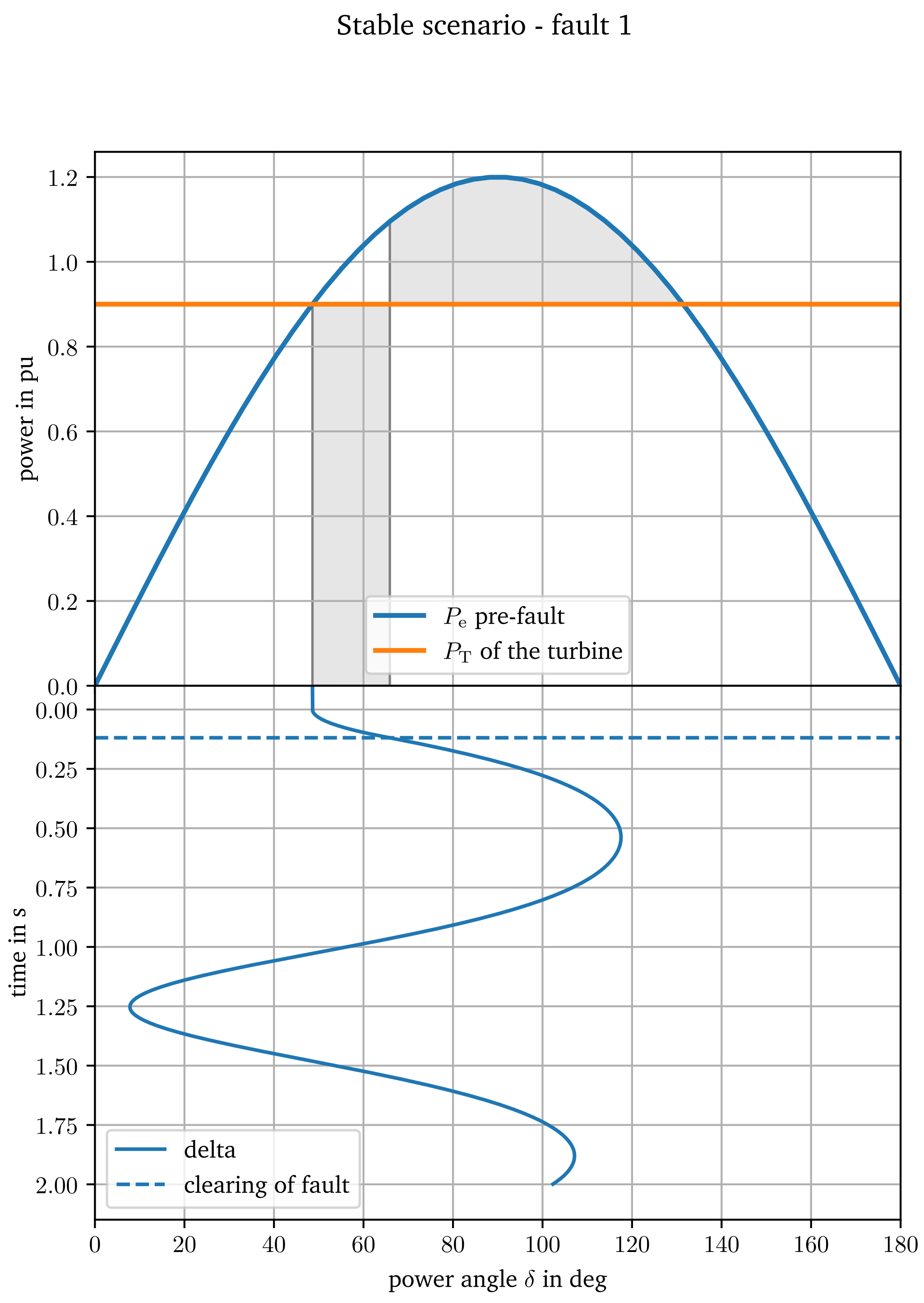
A Graphics and tables

A.1 Initial values

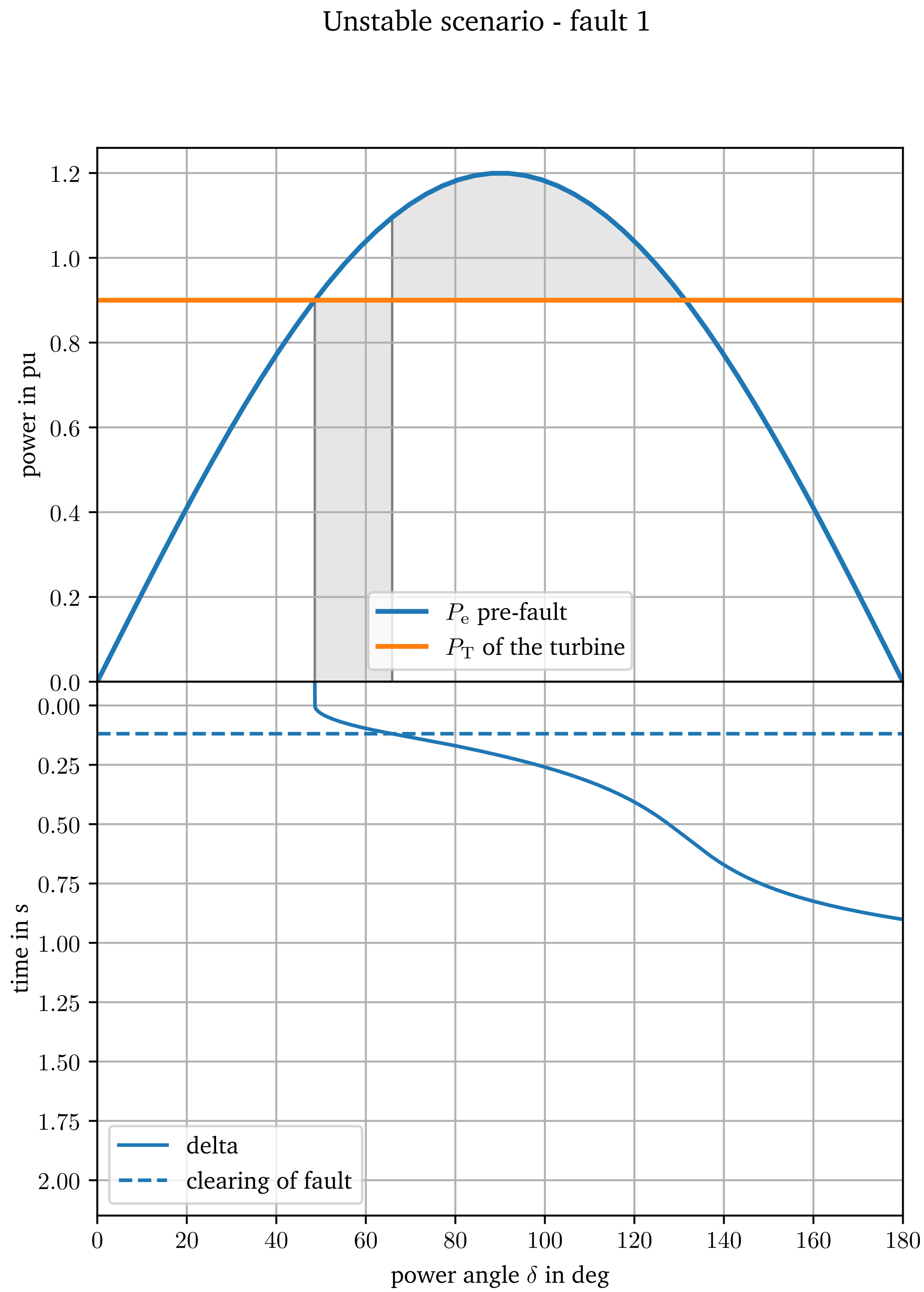
Table A.1: title

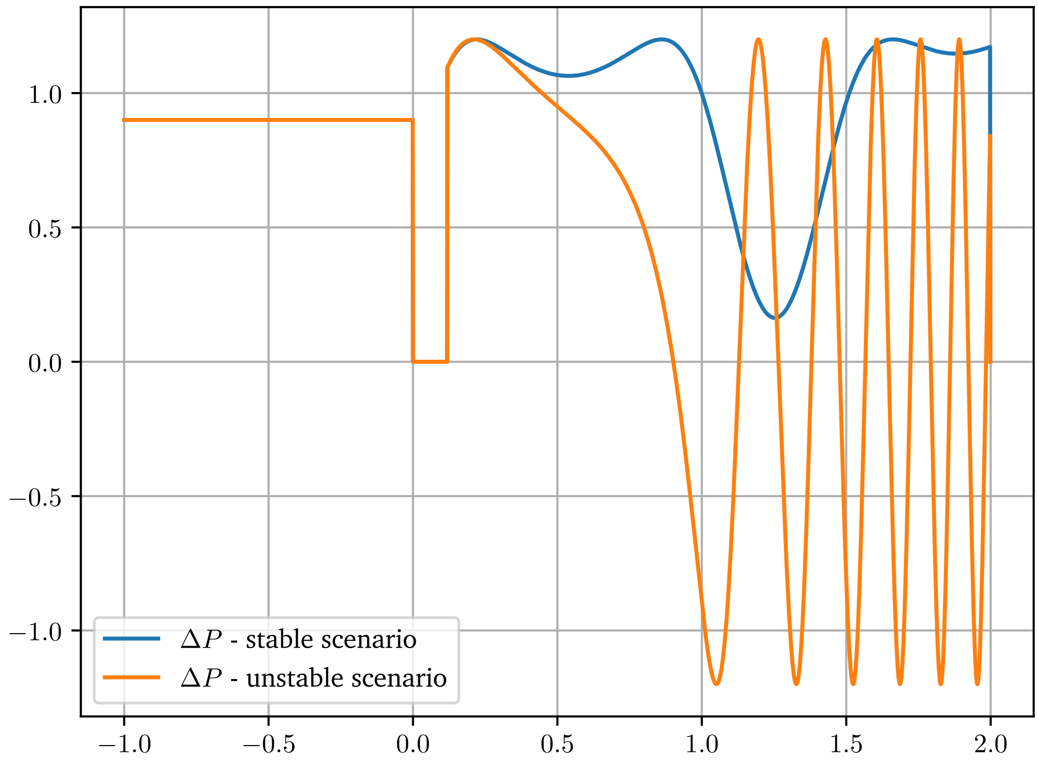
a	b

A.2 Fault 1

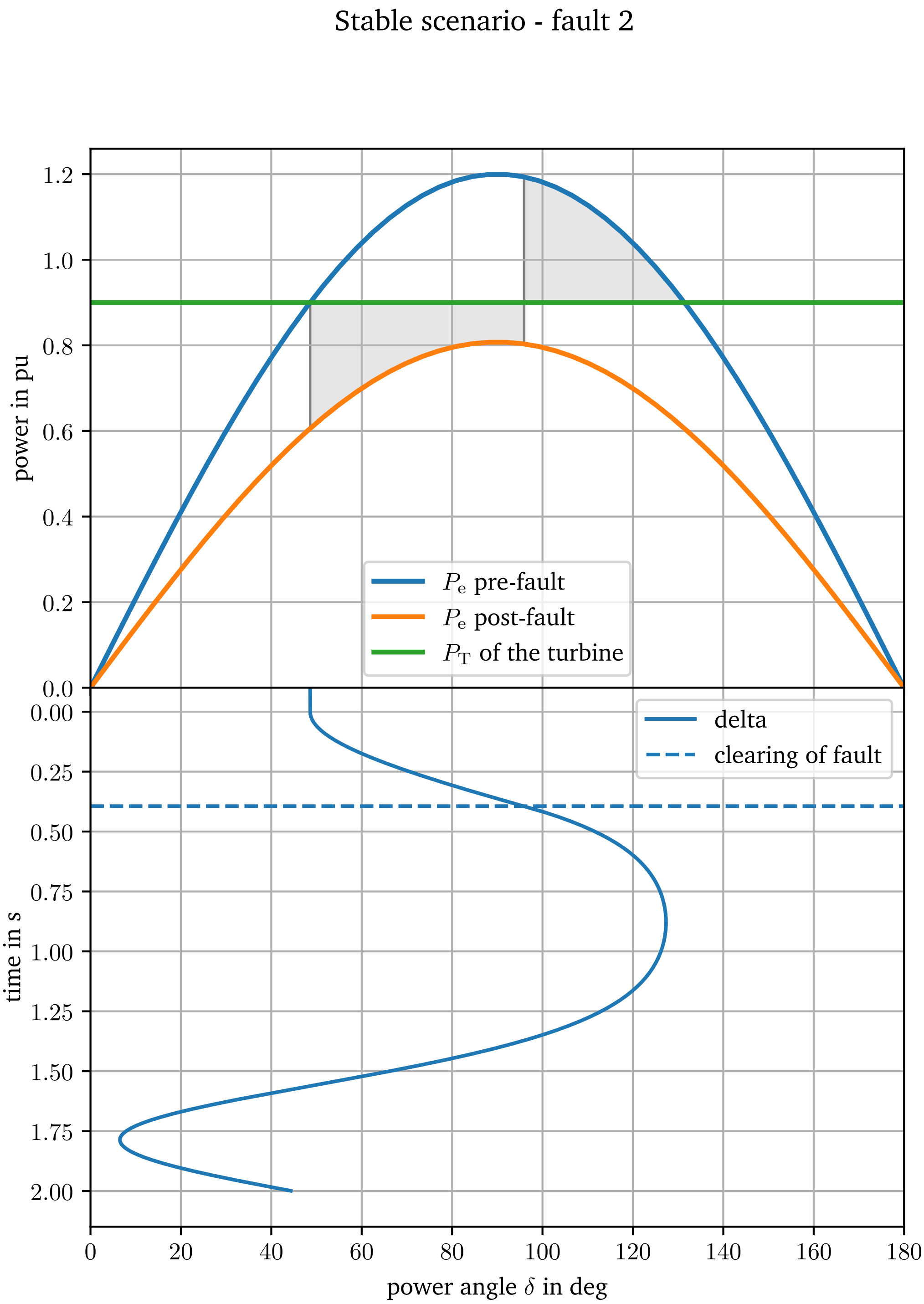


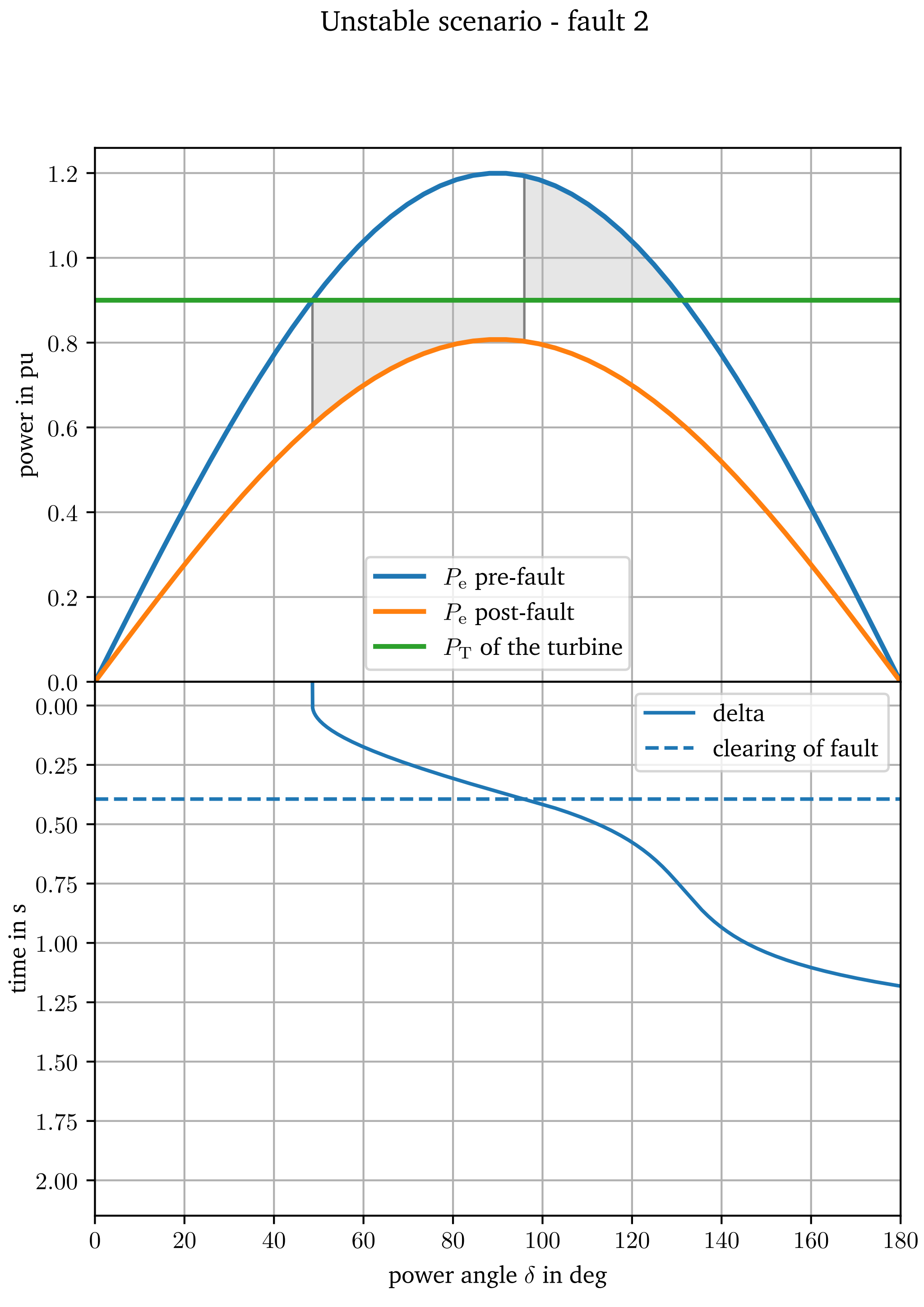
Stable scenario - fault 1
1.2
1.0
0.8
0.6
0.4
0.2
0.0
power in pu
P_e pre-fault
P_T of the turbine
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
time in s
delta
clearing of fault
0
20
40
60
80
100
120
140
160
180
power angle δ in deg


Unstable scenario - fault 1
power in pu
1.2
1.0
0.8
0.6
0.4
0.2
0.0
P_e pre-fault
P_T of the turbine
time in s
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
delta
clearing of fault
0
20
40
60
80
100
120
140
160
180
power angle δ in deg

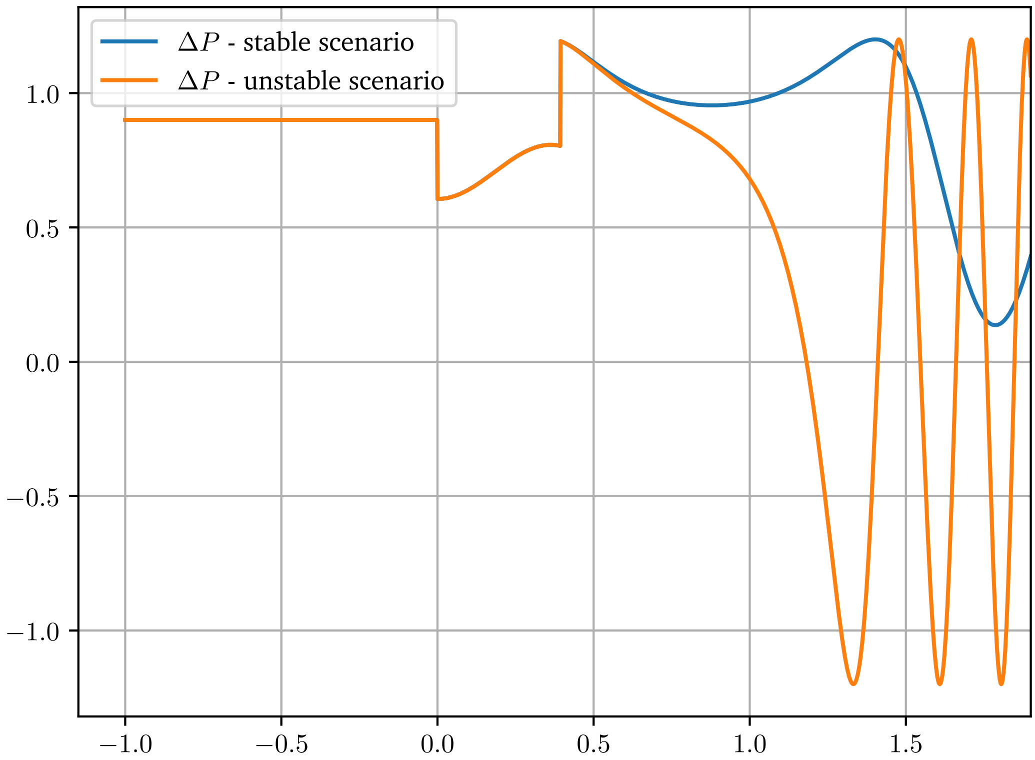

1.0
0.5
0.0
−0.5
−1.0
ΔP - stable scenario
ΔP - unstable scenario
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

A.3 Fault 2


Stable scenario - fault 2
power in pu
1.2
1.0
0.8
0.6
0.4
0.2
0.0
P_e pre-fault
P_e post-fault
P_T of the turbine
time in s
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
delta
clearing of fault
0
20
40
60
80
100
120
140
160
180
power angle δ in deg

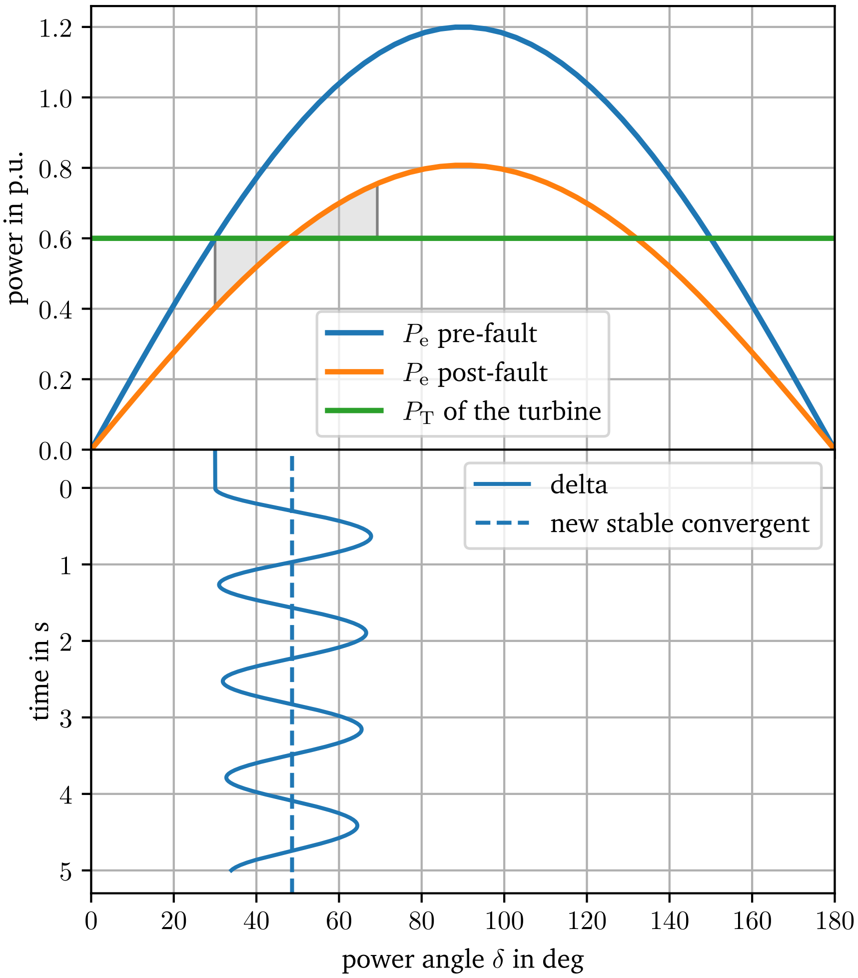


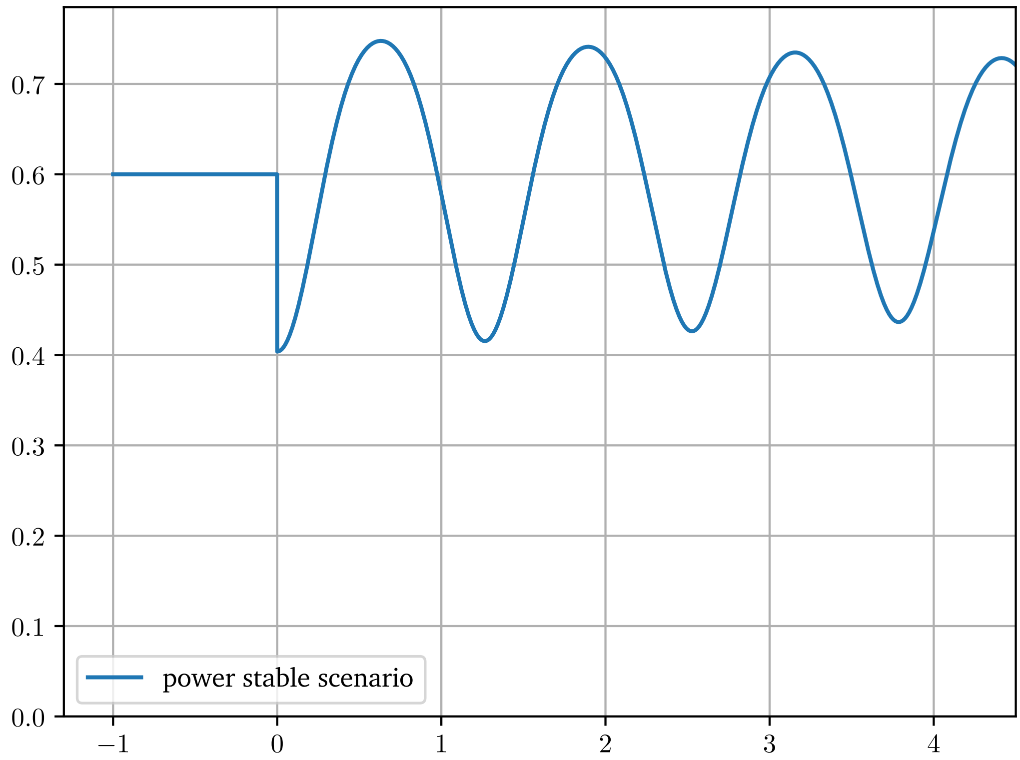
Unstable scenario - fault 2
power in pu
1.2
1.0
0.8
0.6
0.4
0.2
0.0
P_e pre-fault
P_e post-fault
P_T of the turbine
delta
clearing of fault
time in s
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
20
40
60
80
100
120
140
160
180
power angle δ in deg



ΔP - stable scenario
ΔP - unstable scenario
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.5

A.4 Fault 3


power in p.u.
1.2
1.0
0.8
0.6
0.4
0.2
0.0
P_e pre-fault
P_e post-fault
P_T of the turbine
time in s
0
1
2
3
4
5
delta
new stable convergent
0
20
40
60
80
100
120
140
160
180
power angle δ in deg


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−1
0
1
2
3
4
power stable scenario

B Code

B.1 Fault scenario parameters

B.2 Main model

```
#############################
# Base module with the smib model.
# Input of the interested variables delta_0, E_bus, E_gen, P_m, ..., fault_start,
    fault_end, ...
# Export of stability, t_cc, delta_cc, t_sim, delta(t_sim), omega(t_sim),
# Condsideration of TDS in just stable and just unstable regime
#############################

import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
import matplotlib as mpl
import numpy as np
import scipy as sp
from scipy.integrate import odeint

# redefining plot save parameters
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": ["Charter"],
    "font.size": 10
})

# uncomment for updating savefig options for latex export
# mpl.use("pgf")

# helping function for calculation with complex numbers
def mag_and_angle_to_cmplx(mag, angle):
    return mag * np.exp(1j * angle)

def algebraic(delta_gen, fault_on):
    global E_fd_gen
    global E_fd_ibb
    global delta_ibb_init
    global X_gen, X_line, X_ibb, X_trans

    # If the SC is on, the admittance matrix is different.
    # The SC on busbar 0 is expressed in the admittance matrix as a very large
        admittance (1000000) i.e. a very small impedance.
    if fault_on:
        y_adm = np.array([X_fault,
                          [1j / X_line, -1j / X_line - 1j / X_ibb]])
    else:
        y_adm = np.array([[-1j / X_gen - 1j / X_line, 1j / X_line],
                          [1j / X_line, -1j / X_line - 1j / X_ibb]])
```

```
    # Calculate the inverse of the admittance matrix (Y^-1)
    y_inv = np.linalg.inv(y_adm)

    # Calculate current injections of the generator and the infinite busbar
    i_inj_gen = mag_and_angle_to_cmplx(E_fd_gen, delta_gen) / (1j * X_gen)
    i_inj_ibb = mag_and_angle_to_cmplx(E_fd_ibb, delta_ibb_init) / (1j * X_ibb)

    # Calculate voltages at the bus by multiplying the inverse of the admittance
        matrix with the current injections
    v_bb_gen = y_inv[0, 0] * i_inj_gen + y_inv[0, 1] * i_inj_ibb
    v_bb_ibb = y_inv[1, 0] * i_inj_gen + y_inv[1, 1] * i_inj_ibb

    return v_bb_gen

def P_e(delta, fault_on):
    # function for determing P_e WITHOUT algebraic help
    global X_gen
    global X_line
    global X_fault
    global E_fd_gen
    global E_fd_ibb

    if fault_on:
        X = 1
        E_ibb = 0
    else:
        X = X_gen + X_line + X_ibb
        E_ibb = E_fd_ibb

    P_e_gen = E_fd_gen * E_ibb / X * np.sin(delta)
    return P_e_gen

def P_e_alg(delta, fault_on):
    # function for determing P_e WITH algebraic help
    global E_fd_gen
    global X_gen

    v_bb_gen = algebraic(delta, fault_on)

    E_gen_complex = mag_and_angle_to_cmplx(E_fd_gen, delta)
    P_e_gen = (v_bb_gen * np.conj((E_gen_complex - v_bb_gen) / (1j * X_gen))).real
    return P_e_gen

def P_m(omega):
    # returning the torque of the generator, depending on the rotor speed
    global P_m_gen
    global omega_gen_init
    P_t = P_m_gen / (1 + (omega_gen_init + omega))
    return P_t

def get_max_delta(gen_parameters, sim_parameters, alg):
    init(gen_parameters, sim_parameters)

    area_acc = sp.integrate.quad(P_r_deg, delta_gen_init, delta_0_fault, args=(0,
        True, alg))
    area_dec = [0, 0]
    max_delta = delta_0_fault
    while abs(area_dec[0]) <= abs(area_acc[0]):
```

```
        area_dec = sp.integrate.quad(P_r_deg, delta_0_fault, max_delta, args=(0,
            True, alg))
        max_delta = max_delta + 0.01

    return max_delta

def get_delta_0(gen_parameters, sim_parameters, alg):
    init(gen_parameters, sim_parameters)
    x_rad = np.linspace(0,np.pi/2,360)
    delta = -1
    for x in x_rad:
        if abs(P_r_deg(x, 0, False, alg)) <= 0.01:
            delta = x

    return delta

# function for using odeint as ode-solver
def ODE_system(state, t, fault_start, fault_end, alg):

    omega, delta = state

    global H_gen
    global E_fd_gen
    global E_fd_ibb
    global X_gen
    global X_line
    global fn

    if fault_start <= t < fault_end:
        fault_on = True
        # P_e_conv = P_e(E_fd_gen, 0, X_gen, delta)
    else:
        fault_on = False
        # P_e_conv = P_e(E_fd_gen, E_fd_ibb, X_gen + X_line, delta)

    # including time dependent solving of algebraic equations
    if alg:
        P_e_gen = P_e_alg(delta, fault_on)
    else:
        P_e_gen = P_e(delta, fault_on)

    d_omega_dt = 1 / (2 * H_gen) * (P_m(omega) - P_e_gen)
    d_delta_dt = omega * 2 * np.pi * fn

    return [d_omega_dt, d_delta_dt]

# functions for determing the critical clearing time
def P_r_deg(delta, omega, fault_on, alg):
    # determing the P_e curve under input in degrees
    if alg:
        P_r = P_e_alg(delta, fault_on) - P_m(omega)
    else:
        P_r = P_e(delta, fault_on) - P_m(omega)
    return P_r

def P_t_deg(x):
    # determing the P_t curve under input in degrees
    global P_m_gen
```

```
    return P_m_gen*np.ones(np.size(x))

def stability_eac(delta_0, delta_act, omega_act, delta_max, alg):
    # global delta_new, omega_new

    # Compare the acceleration area until the given delta and compare it to the
        braking area left until the dynamic stability point is passed
    area_acc = sp.integrate.quad(P_r_deg, delta_0, delta_act, args=(omega_act, True
        , alg))
    area_dec = sp.integrate.quad(P_r_deg, delta_act, delta_max, args=(omega_act, (
        not clearing), alg))

    if abs(area_acc[0]) < abs(area_dec[0]): # True: stable, False: NOT stable
        return True
    else:
        return False

def determine_cct(t_sim, delta, omega, delta_0, alg):
    # t_sim and delta are result arrays
    # delta_0 is the initial angle delta of the stable system pre-fault

    # Save current time and delta at time point i; iterate through i to test any
        given time until stability can't be remained; delta_cc and t_cc is the
        angle and time at the last stable point
    global delta_max_fault
    if clearing:
        delta_max = np.pi - delta_0
    else:
        delta_max = delta_max_fault

    i = 0
    t_cc, delta_cc, omega_cc = -1, -1, -1

    while stability_eac(delta_0, delta[i], omega[i], delta_max, alg) and i < np.
        size(t_sim)-1 and delta[i] < delta_max_fault:
        t_cc = t_sim[i]
        delta_cc = delta[i]
        omega_cc = omega[i]
        i = i + 1

    if t_cc < 0:
        return False, -1, -1, -1
    else:
        if clearing:
            return True, t_cc, delta_cc, omega_cc
        else:
            return True, t_cc, delta_0_fault, omega_gen_init

# execution functions for simulation
def do_sim(gen_parameters, sim_parameters, alg):
    init(gen_parameters, sim_parameters)

    # setup simulation inputs
    t_sim = np.arange(sim_parameters["t_start"], sim_parameters["t_end"],
        sim_parameters["t_step"])
    initial_conditions = [gen_parameters["omega_gen_init"], gen_parameters["
        delta_gen_init"]]

    delta_0 = gen_parameters["delta_gen_init"]
```

```
    # delta_max = np.pi - delta_0

    for i in range(1,4,1):
        if i == 1: # first TDS with no fault-clearing
            # solve ODE with python solver
            solution = odeint(ODE_system, initial_conditions, t_sim, args=(
                sim_parameters["fault_start"], sim_parameters["fault_end"], alg))
            stability, t_cc, delta_cc, omega_cc = determine_cct(t_sim, solution[:,
                1], solution[:, 0], delta_0, alg)
        elif i == 2: # second TDS with fault clearing just right
            # solve ODE with python solver
            fault_end = t_cc - 5 * sim_parameters["t_step"]
            solution_stable = odeint(ODE_system, initial_conditions, t_sim, args=(
                sim_parameters["fault_start"], fault_end, alg))
        elif i == 3: # second TDS with fault clearing just NOT right
            fault_end = t_cc + 2 * sim_parameters["t_step"]
            solution_unstable = odeint(ODE_system, initial_conditions, t_sim, args
                =(sim_parameters["fault_start"], fault_end, alg))

    return stability, t_cc, delta_cc, t_sim, solution_stable, solution_unstable

def do_sim_simple(gen_parameters, sim_parameters, alg):
    init(gen_parameters, sim_parameters)

    # setup simulation inputs
    t_sim = np.arange(t_start, t_end, t_step)
    initial_conditions = [omega_gen_init, delta_gen_init]

    delta_0 = delta_gen_init

    solution = odeint(ODE_system, initial_conditions, t_sim, args=(fault_start,
        fault_end, alg))
    stability, t_cc, delta_cc, omega_cc = determine_cct(t_sim, solution[:, 1],
        solution[:, 0], delta_0, alg)

    return stability, t_cc, delta_cc, t_sim, solution

def init(gen_parameters, sim_parameters):
    global fn, H_gen, X_gen, X_ibb, X_line, X_trans, X_fault, E_fd_gen, E_fd_ibb,
        P_m_gen, omega_gen_init, delta_gen_init, delta_ibb_init, t_start, t_end,
        t_step, fault_start, fault_end, clearing

    fn = gen_parameters["fn"]
    H_gen = gen_parameters["H_gen"]
    X_gen = gen_parameters["X_gen"]
    X_ibb = gen_parameters["X_ibb"]
    X_line = gen_parameters["X_line"]
    X_fault = gen_parameters["X_fault"]
    X_trans = gen_parameters["X_trans"]

    E_fd_gen = gen_parameters["E_fd_gen"]
    E_fd_ibb = gen_parameters["E_fd_ibb"]
    P_m_gen = gen_parameters["P_m_gen"]

    omega_gen_init = gen_parameters["omega_gen_init"]
    delta_gen_init = gen_parameters["delta_gen_init"]
    delta_ibb_init = gen_parameters["delta_ibb_init"]

    t_start = sim_parameters["t_start"]
```

```
    t_end = sim_parameters["t_end"]
    t_step = sim_parameters["t_step"]

    fault_start = sim_parameters["fault_start"]
    fault_end = sim_parameters["fault_end"]
    clearing = sim_parameters["clearing"]

    # assessment of delta_0 and delta_max in fault case
    x_rad = np.linspace(0,np.pi, 360)
    i = 0
    global delta_0_fault, delta_max_fault
    delta_0_fault = np.pi
    delta_max_fault = np.pi
    while i < np.size(x_rad)/2:
        if abs(P_r_deg(x_rad[i], 0, True, True)) < 0.01:
            delta_0_fault = x_rad[i]
            delta_max_fault = np.pi - delta_0_fault
        i = i + 1

    return

if __name__ == "__main__":
    # setup simulation inputs
    gen_parameters = {
        "fn":        50,
        "H_gen":     3.3,
        "X_gen":     0.2,
        "X_trans":   0.1,
        "X_ibb":     0.1,
        "X_line":    0.65,
        "X_fault":   0.0001,

        "E_fd_gen": 1.14,
        "E_fd_ibb": 1.0,
        "P_m_gen":  0.9,

        "omega_gen_init": 0,
        "delta_gen_init": np.deg2rad(48.59),
        "delta_ibb_init": np.deg2rad(0)
    }

    sim_parameters = {
        "t_start":      -1,
        "t_end":        5,
        "t_step":       0.001,

        "fault_start":  0,
        "fault_end":    5,
        "clearing":     True
    }

    gen_parameters["X_fault"] = [(-1j / gen_parameters["X_gen"] - 1j /
         gen_parameters["X_line"]) + 1000000, 1j / gen_parameters["X_line"]]

    # Execution of simulation
    alg = True
    stability, t_cc, delta_cc, t_sim, solution_stable, solution_unstable = do_sim(
         gen_parameters, sim_parameters, alg)
```

```
    # Evaluation of results
    print('t_cc:\t\t' + str(round(t_cc, 3)) + ' s')
    print('delta_cc:\t' + str(round(np.rad2deg(delta_cc), 1)) + ' deg')

    delta_stable = solution_stable[:,1]
    omega_stable = solution_stable[:,0]

    ##############################
    # Plot stable result
    ##############################
    fig, axs = plt.subplots(2, 1, figsize=(6,8), sharex=True)

    # determine the boundary angles
    delta_0 = delta_gen_init # delta_gen_init
    delta_max = np.pi - delta_0

    # calculation of P_e_pre, P_e_post, and P_t
    x_deg = np.linspace(0, 180) # linear vector for plotting in deg
    x_rad = np.linspace(0, np.pi) # linear vector for calculation in rad
    P_e_pre = P_e_alg(x_rad, False)
    P_e_post = P_e_alg(x_rad, True)
    P_t = P_t_deg(x_rad)

    plt.subplots_adjust(hspace=.0)

    ##############################
    # ax1
    ##############################
    axs[0].plot(x_deg, P_e_pre, '-', linewidth=2, label='$P_\mathrm{e}$ pre-fault')
    # axs[0].plot(x_deg, P_e_post, '-', linewidth=2, label='$P_\mathrm{e}$ post-fault')
    axs[0].plot(x_deg, P_t, '-', linewidth=2, label='$P_\mathrm{T}$ of the turbine')
    axs[0].set_ylim(bottom=0)
    delta_0_deg = np.rad2deg(delta_0)
    delta_max_deg = np.rad2deg(delta_max)
    delta_c_deg = np.rad2deg(delta_cc)
    # axs[0].set_xticks([0, 180, delta_0_deg, delta_c_deg, delta_max_deg], labels=['0', '180', '$\delta_\mathrm{0}$', '$\delta_\mathrm{c}$', '$\delta_\mathrm{max}$'])

    ix1 = np.linspace(delta_0_deg, delta_c_deg)
    iy1 = P_e_alg(np.deg2rad(ix1), True)
    axs[0].fill_between(ix1, iy1, P_m_gen, facecolor='0.9', edgecolor='0.5')

    # Make the shaded region for area_dec, https://matplotlib.org/stable/gallery/lines_bars_and_markers/fill_between_demo.html
    ix2 = np.linspace(delta_c_deg, delta_max_deg) # -> does this have to be in rad or in deg?
    iy2 = P_e_alg(np.deg2rad(ix2), False)
    axs[0].fill_between(ix2, iy2, P_m_gen, facecolor='0.9', edgecolor='0.5')
    axs[0].grid()
    axs[0].legend()
    axs[0].set_ylabel('power in p.u.')

    ##############################
    # ax2
    ##############################
    axs[1].plot(np.rad2deg(delta_stable), t_sim, label='delta')
```

```
    fig.gca().invert_yaxis()
    # axs[1].axhline(y=fault_end, linestyle='--', label='clearing of fault')
    axs[1].grid()
    axs[1].set_ylabel('time in s')
    axs[1].legend()
    plt.ylim(top=-.5)
    plt.xlim(left=0, right=180)
    plt.xlabel('power angle $\delta$ in deg')

    plt.suptitle('Stable scenario')
    plt.show()
```

C Additional

C.1 Parameter variation