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 7, 2024 at 17:18 – preliminary version 0.1]

Contents

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; .

[MK1]: Write Introduction

Missing figure

Insert senseful schematic picture/graph

The goal of this Student Research Paper is the implementation of a critical clearing time (CCT) determing 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 determing the CCT and the critical power angle, is illustrated.

2 Fundamentals

[MK2]: Write Chap 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 $\Delta\omega$:

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

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

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.

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

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

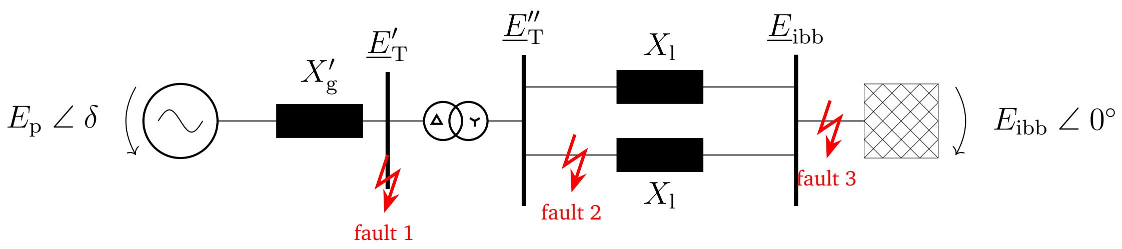

Figure 2.1: Representative circuit of a single machine infinite bus (SMIB) model with pole wheel voltage $E_{\mathrm{p}} \angle \delta$ and infinite bus bar (IBB) voltage $E_{\mathrm{ibb}} \angle 0°$; positions of considered faults 1 to 3 are marked with red lightning arrows

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), with considering that the amount of stored energy through acceleration (during the short or failure) is equal to the released energy (decellerating 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 decellerating area between δ_c and the maximum dynamically stable angle δ_{max}. Figure 2.2 is illustrating this approach. Following on this approach a generalized expression is formed to

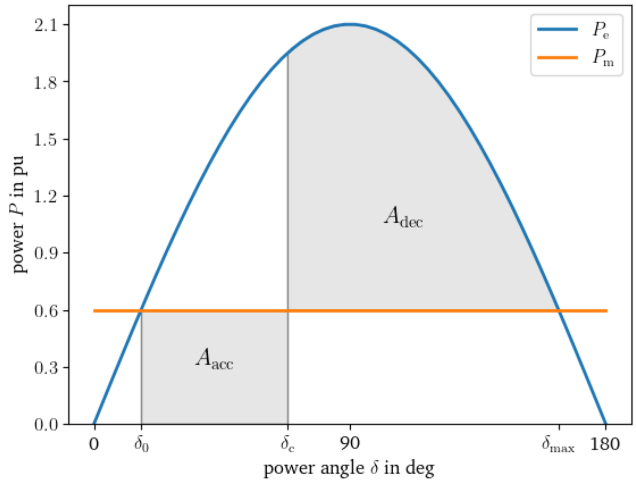

Figure 2.2: Illustrated equal area criterion (EAC) in the P-δ-curve

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

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

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

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).

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

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

$$f_{\text{t}+1} = f_{\text{t}} + \left(\frac{df}{dt}\right)_{\text{t}} \cdot \Delta t, \tag{2.8}$$

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 Methods

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/* [7], similiar for *matplotlib*, and *numpy* packages

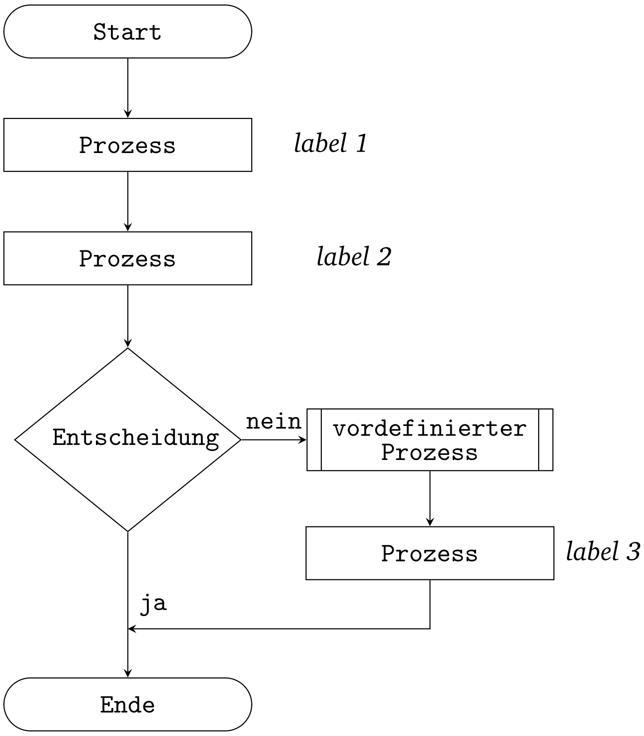

Figure 3.1: Program plan for determining the critical clearing time (CCT)

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

Simplified networks, which are in interest and shall be simulated with the algorithm; ideally solved with subfigures

3.2.2 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

4.2 Numerical results

Table 4.1 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 4.1: 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

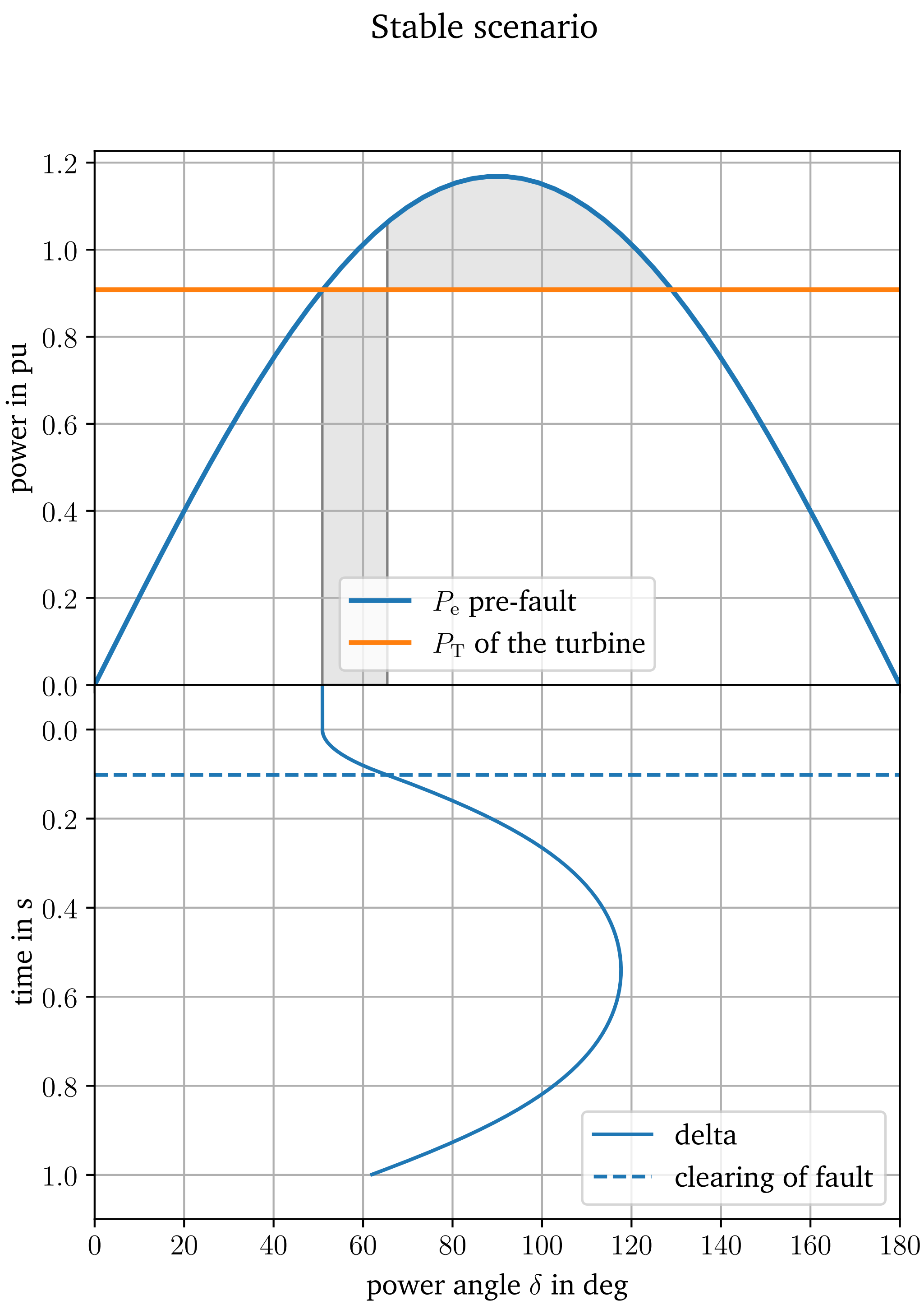

Figure 4.1: Fault 1

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.

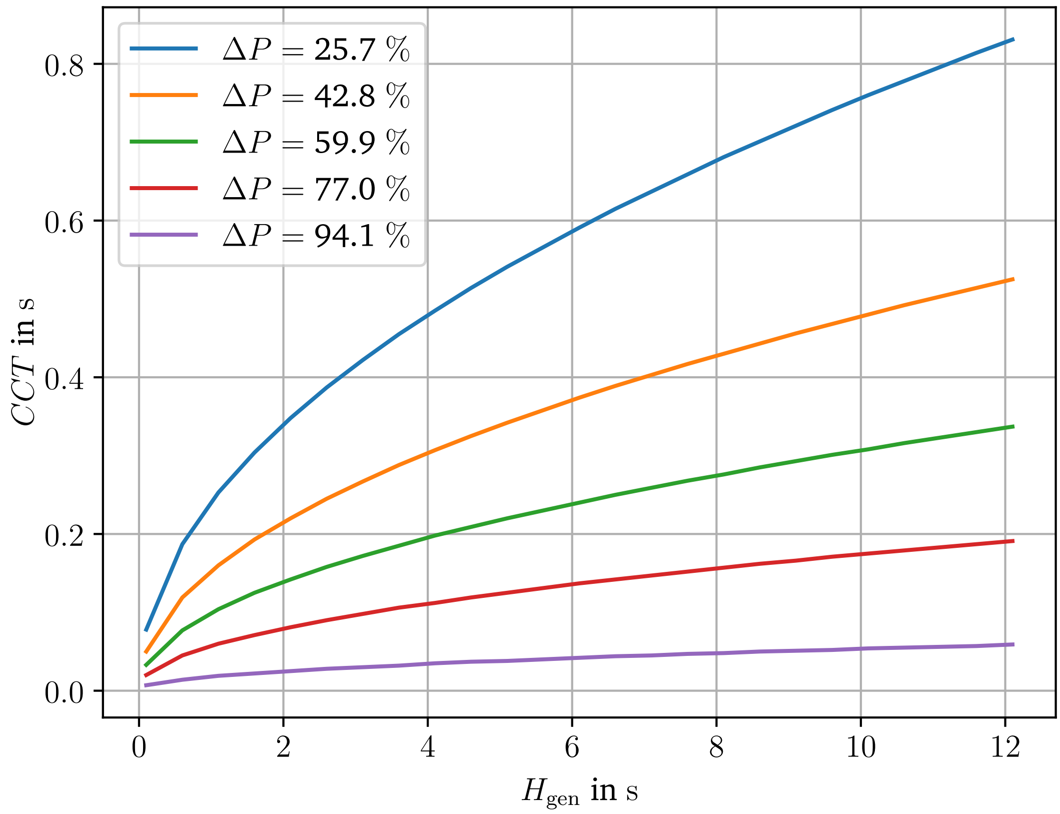

Figure 4.2: Parameter comparison

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 [8]).

- 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] 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 Code

A.1 Model functions

A.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": 12
})

# 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
```

```

    # 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
        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
```


[March 7, 2024 at 17:18 - preliminary version 0.1]

```

    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_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"]

    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":       60,
        "H_gen":    3.5,
```

```
        "X_gen":     0.2,
        "X_ibb":     0.1,
        "X_line":    0.65,
        "X_fault":   0.0001,

        "E_fd_gen": 1.075,
        "E_fd_ibb": 1.033,
        "P_m_gen":  1998/2200,

        "omega_gen_init": 0,
        "delta_gen_init": np.deg2rad(50.9),
        "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 pu')

    ##############################
    # 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()
```

A.3 Fault models

Fault 2

```
############################
# simulation of fault 2
# t_cc, delta_cc and some plots
#
# scerario: partly line fault (P_e = XX * P_e), clearing mode (around
    t_cc)
############################

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

import smib_model as sim

# redefining plot save parameters
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": ["Charter"],
    "font.size": 12
})

# uncomment for updating savefig options for latex export
```

```
# mpl.use("pgf")

def init(gen_parameters, sim_parameters):
    global fn, H_gen, X_gen, X_ibb, X_line, 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"]

    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"]

    return

if __name__ == "__main__":
    # setup simulation inputs
    gen_parameters = {
        "fn":       60,
        "H_gen":    3.5,
        "X_gen":    0.2,
        "X_ibb":    0.1,
        "X_line":   0.65,
        "X_fault":  0.0001,

        "E_fd_gen": 1.075,
        "E_fd_ibb": 1.033,
        "P_m_gen":  1998/2200,

        "omega_gen_init": 0,
```

```
        "delta_gen_init": np.deg2rad(50.9),
        "delta_ibb_init": np.deg2rad(0)
    }

    sim_parameters = {
        "t_start":      -1,
        "t_end":        2,
        "t_step":       0.001,

        "fault_start":  0,
        "fault_end":    1,
        "clearing":     True
    }

    gen_parameters["X_fault"] = [(-1j / gen_parameters["X_gen"] - 1j*3 /
         gen_parameters["X_line"]), 1j / gen_parameters["X_line"]]

    init(gen_parameters, sim_parameters)

    # Execution of simulation
    alg = True
    stability, t_cc, delta_cc, t_sim, solution_stable, solution_unstable
         = sim.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]
    delta_unstable = solution_unstable[:,1]

    ##############################
    # Plot unstable 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 = sim.P_e_alg(x_rad, False)
    P_e_post = sim.P_e_alg(x_rad, True)
    P_t = sim.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 = sim.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 = sim.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 pu')

    ###############################
    # ax2
    ###############################
    axs[1].plot(np.rad2deg(delta_stable), t_sim, label='delta')
    fig.gca().invert_yaxis()
    axs[1].axhline(y=t_cc, linestyle='--', label='clearing of fault')
    axs[1].grid()
    axs[1].set_ylabel('time in s')
    axs[1].legend()
    plt.ylim(top=-.1)
    plt.xlim(left=0, right=180)
    plt.xlabel('power angle $\delta$ in deg')
```

```

    plt.suptitle('Stable scenario - fault 2')
    # plt.savefig('plots/fault1_stable.pgf')
    plt.show()
    # plt.close()

    ##############################
    # Plot UNstable 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 = sim.P_e_alg(x_rad, False)
    P_e_post = sim.P_e_alg(x_rad, True)
    P_t = sim.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 = sim.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 = sim.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 pu')

    ##############################
    # ax2
    ##############################
    axs[1].plot(np.rad2deg(delta_unstable), t_sim, label='delta')
    fig.gca().invert_yaxis()
    axs[1].axhline(y=t_cc, linestyle='--', label='clearing of fault')
    axs[1].grid()
    axs[1].set_ylabel('time in s')
    axs[1].legend()
    plt.ylim(top=-.1)
    plt.xlim(left=0, right=180)
    plt.xlabel('power angle $\delta$ in deg')

    plt.suptitle('Unstable scenario - fault 2')
    # plt.savefig('plots/fault1_unstable.pgf')
    plt.show()
    # plt.close()
```

A.4 Additional comparison

Generator parameter

```
############################
# Comparison of different machine parameters and operational points.
############################

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
# import tikzplotlib as tp
```

```
import smib_model as sim

# redefining plot save parameterss
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": ["Charter"],
    "font.size": 12
})

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

# comparing different H_gen in t_cc, delta_cc and their TDS when a fault
     is ON


# comparing different degree of utilization (P_e / P_e_max OR delta_P;
    direct correlated: delta_0)


def init(gen_parameters, sim_parameters):
    global fn, H_gen, X_gen, X_ibb, X_line, 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

    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"]

    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"]

    return

if __name__ == "__main__":
    # setup simulation inputs
    gen_parameters = {
        "fn":       60,
        "H_gen":    3.5,
        "X_gen":    0.2,
        "X_ibb":    0.1,
        "X_line":   0.65,
        "X_fault":  0.001,

        "E_fd_gen": 1.075,
        "E_fd_ibb": 1.033,
        "P_m_gen":  1998/2200,

        "omega_gen_init": 0,
        "delta_gen_init": np.deg2rad(50.9),
        "delta_ibb_init": np.deg2rad(0)
    }

    sim_parameters = {
        "t_start":      0,
        "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
        "]]

    # init(gen_parameters, sim_parameters)

    # Execution of simulations and savon in t_ccs array
    alg = True

    sim.init(gen_parameters, sim_parameters)
    P_e_max = sim.P_e_alg(np.pi/2, False)

    H = np.arange(0.1, 12.5, 0.5) # [2.5, 3, 3.5, 4, 4.5, 5]
    delta_P = [0.3, 0.5, 0.7, 0.9, 1.1]
```

```
    t_ccs = np.zeros((np.size(delta_P), np.size(H)))
    i = 0
    for delta_P_new in delta_P:
        gen_parameters["P_m_gen"] = delta_P_new
        gen_parameters["delta_gen_init"] = sim.get_delta_0(
            gen_parameters, sim_parameters, alg)
        z = 0
        for H_new in H:
            gen_parameters["H_gen"] = H_new

            stability, t_cc, delta_cc, t_sim, solution = sim.
                do_sim_simple(gen_parameters, sim_parameters, alg)
            print(t_cc)

            if stability:
                t_ccs[i, z] = t_cc
            else:
                t_ccs[i, z] = -1
            z = z + 1

        plt.plot(H, t_ccs[i,:], label=("$\Delta P =$ " + str(round(
            delta_P_new/P_e_max*100, 1)) + " $\%$"))
        i = i + 1

    plt.legend()
    plt.xlabel("$H_\mathrm{gen}$ in $\mathrm{s}$")
    plt.ylabel("$CCT$ in $\mathrm{s}$")
    # plt.title("Influence from $H_\mathrm{gen}$ and $\Delta P$ on CCT")
    plt.grid()
    # figure = plt.gcf() # get current figure
    # figure.set_size_inches(8, 6)
    # plt.savefig('plots/parameter_comparison.pgf', dpi=300)
    plt.show()
    # plt.close()
```

B Additional

B.1 Jupyter notebook for development