Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Power Engineering

Åsmund Sælen

Topflow, a Toolbox for Specialized Power System Analysis

Master's thesis in Energy and Environmental Engineering
Supervisor: Olav Bjarte Fosso
July 2020

NTNU
Kunnskap for en bedre verden

Åsmund Sælen

Topflow, a Toolbox for Specialized Power System Analysis

Master's thesis in Energy and Environmental Engineering
Supervisor: Olav Bjarte Fosso
July 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Power Engineering

NTNU
Norwegian University of
Science and Technology

Abstract

Power system analysis is a branch of electrical engineering which is essential in designing electrical power systems. Simulations show if systems operate as expected, can withstand stress, and protect against failures. Many tools for power system analysis exist; one is a toolbox created by NTNU professor Olav Bjarte Fosso in the 90s. The toolbox consists of various FORTRAN-routines that do simulations such as Newton-Rapshon load-flow, continuation load-flow, contingency analysis, and security-constrained DC load-flow. Previous projects have worked towards making the toolbox available in Python, a popular object-orientated programming language. The speed and the functionality of the program is preserved by writing parts of the modernized code in C. The primary goal of this master's thesis is to update the simulation-function "acsolve", which performs a Newton-Rapshon load-flow, and make sure it matches the performance of existing open-source toolboxes in Python.

The specialization project "Toolbox for Power System Analysis" [1] from 2019 by Åsmund Sælen purposed an initial implementation of acsolve. However, the implementation lacked functions for initializing the input-data; hence, the program could not run simulations on large power systems. This master's thesis presents Python-functions that communicates with Excel-files, and a more sophisticated way of initializing the parameters of the simulation-functions. An additional algorithm named "decsolve", which performs fast-decoupled load-flows, is implemented. The updated program is tested on various MATPOWER standard-cases to study the code's reliability and performance. Profiling-tools available in Python are used to optimize the initial implementations, and the module "timit" is used to benchmarks the program against other open-source projects.

The result of the master's thesis is the toolbox "Topflow", which consists of translated load-flow functions that originate from the FORTRAN-code developed by Olav Bjarte Fosso. Topflow uses a combination of Python and C to boost the program's speed, and the functions use sparse methods to save memory and optimize the performance. The Newton-Rapshon load-flow function (acsolve) gives reliable results in all the tests, and the speed matches the open-source Python-projects "pypower" and "pandapower". The second simulation-function, decsolve, gave satisfactory results on small and medium-large power systems, but not on large. The conclusion of the master's thesis is that it reached the primary goal of updating acsolve, and gave recommendations on how to resolve the apparent bugs in decsolve. Future contributors can use the documentations in this thesis to complete the translation of the original toolbox.

Samandrag

Kraftsystemanalyse er ein vesentleg del av å utvikle elektriske kraftsystem. Simuleringar viser om systema opererer som forventa og kan beskytte mot feil. Det finnast mange verktøy for å analysere kraftsystem, eit av dei er ein kjeldekode som NTNU professor Olav Bjarte Fosso utvikla på 90-talet. Koden, som har fått namn «Topfllow», er bygd opp av ulike FORTRAN-rutinar som simulerer blant anna Newton-Rhapson last-flyt, CPF-analyser, utfalls-analyser og optimal DC last-flyt. Tidlegare prosjekt har arbeidt for å implementere Topflow i Python, eit mykje brukt objekt-orientert programmeringsspråk. Ytelsen og funksjonaliteten til det originale programmet er teke vare på ved å skrive delar av den moderniserte koden i C. Primærmålet med denne masteroppgåva er å oppdatere simuleringsfunksjonen "acsolve", som utøver Newton-Rhapson last-flyt, og sørgjer for at den køyrer like effektivt som liknande modular tilgjengeleg i Python.

Spesialiserings prosjektet "Toolbox for Power System Analysis" [1] frå 2019 av Åsmund Sælen presenterte eit førsteutkast til oppdateringa av acsolve. Denne implementeringa mangla funksjonar for å setje input-data; noko som medførte at programmet ikkje var i stand til å køyre simuleringar på store kraftsystem. Denne masteroppgåva presentera funksjonar som kommuniserer med Excel-filer, og ein meir sofistikert måte å setje parameterane av simuleringsfunksjonane på. Topflow er også oppdatert med ein ny rutine, "decsolve", som utfører last-flyten FDLF. Det oppdaterte programmet er testa på fleire MATPOWER standard system for å undersøke kor påliteleg og rask koden er til å gjennomføre analysane. Tilgjengelege modular i Python er brukt for å optimalisere simularings- funksjonane.

Denne masteroppgåva resulterer i den oppdaterte versjonen av Topflow. Topflow nyttar no ein kombinasjon av Python og C for å auke programmet sin ytelse, saman med metoder som utnytter fordelane til glissne (sparse) matriser. Newton-Rhapson last-flyt funksjonen (acsolve) gjer pålitelege resultat på alle systema den er testa på, og den er minst like rask som dei eksisterande Python-programma "pypower" og "pandapower". Den andre simuleringsfunksjonen, decsolve, gjer tilfredsstillande resultat for små og middels store kraftsystem, men ikkje for store. Denne masteroppgåva konkluderer med at primærmålet, å oppdatere acsolve, er nådd. Oppgåva inneheld også tilrådingar for korleis framtidige prosjekt kan finne feila i rutinen decsolve. Framtidige bidragsytarar kan bruke dokumentasjonen i denne masteroppgåva i arbeidet mot å fullføre omsetjinga av den opphavlege FORTAN-koden, og ferdigstille Topflow.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Previous work

This master's thesis contributes to the work of updating the toolbox for specialized power system analysis, which prof. Olav Bjarte Fosso developed in the 90s. Previous work includes implementations by Leif Warland at Statnett, the master thesis "Toolbox for Specialized Power System Analysis" by Hege Bruvik Kvandal [2], and the specialization project by Åsmund Sælen [1], which leads to this thesis.

The original toolbox contains multiple tools for analyzing power systems, such as Newton-Rapshon load-flow, contingency analysis, continuous power-flow, and DC-optimal load-flow. Prof. Olav Bjarte Fosso wrote the code in FORTRAN, a high-performance programming language, which is suited for scientific computing. The toolbox uses sparse methods to save memory and enhance performance, which, combined with the qualities of FORTRAN, makes the program efficient and able to run on large systems.

Prof. Fosso designed the original program for study purposes only, however, Statnett showed interest in modernizing the code and making it available in Python. Leif Warland began the work of translating the code and laid the foundation for the new toolbox, which he named "Topflow". Hege Bruvik Kvandal continued the work in her specialization project [3] and master's thesis [2].

The focus of the specialization project by Åsmund Sælen [1] was to present a first update of the load-flow routine "acsolve" from the original toolbox. The implementations were tested on a 3-bus system, and compared with solved values form the lecture slides by prof Olav Bjarte Fosso [4], and hand-calculations. The test gave satisfactory results, which was an essential first step in translating the activity.

1.2 Scope and problem formulation

The most crucial goal of this thesis is to complete and verify the simulation-function acsolve. This routine performs a Newton-Rapshon load-flow, which is considered the "working horse" of power system analysis [4]. The specialization project by Åsmund Sælen [1] presented an initial update, and gave recommendations on how to make further contributions to the toolbox. These recommendations are listed below.

- Develop functions which initializes the parameters of acsolve

- Test the reliability of the code on larger power-systems
- Test the performance of the program
- Find a different solver for the differential equation-system
- Translate other activities from the original toolbox

This master's thesis builds on the recommendations from [1] to define clear milestones for the work. The milestones are listed below in order of priority.

1. Develop functions which initializes the parameters of acsolve
2. Make acsolve reliable on large systems
3. Find a solver which handles sparse matrices
4. Match the performance of similar open-source projects in Python
5. Translate the fast-decoupled load-flow routine "decsolve" from the original toolbox

Notice that milestone 4 tackles the "performance" of Topflow. The performance of a program may refer to several types of measurements; however, this thesis uses it interchangeably with the code's speed.

The rest of this thesis will discuss the work of completing the milestones above. Chapter 2 explains the basic theory that is needed to understand the work conducted in this master's thesis. Chapter 3 provides a user-guide, which gives the reader an overview of the program and shows the user-interface. Chapter 4 explains the design of the updated toolbox, and introduces the methods for testing the program's reliability and performance. Chapter 5 shows the resulting implementations, and aims to give future contributors a deeper understanding of the current toolbox. Chapter 6 provides the results of the reliability-tests and the speed-tests, and, finally, Chapter 7 concludes the master thesis.

Chapter 2

Theory

The sections 2.1- 2.3, 2.5.7 and 2.5.9 are based on the theory-chapter in the specialization project by Åsmund Sælen from 2019[1].

2.1 Programming language

This thesis uses three different programming languages. The original code is written in Fortran 95, while the updated version uses C and Python. A vast amount of information is available online, and platforms such as YouTube and Tutorialspoint provide excellent tutorials for free . This section assumes that the reader is familiar with basics of programming, such as statements and loops, and will only give a brief introduction to each programming language. Python will be emphasised because it's the primary language of the updated program, Topflow.

2.1.1 Fortran

As one of the oldest programming languages, FORTRAN was created in the 1950s and developed for scientific calculations [5]. FORTRAN is still used, in particular, as a high-performance computing language. It's especially suitable for numerical analysis, which suits the techniques in the original toolbox.

A distinctive feature of FORTRAN is that routines can share data by using so-called "Common-blocks" (separate files that stores input data). If a routine changes a common-block parameter, this change applies to all other routines that share that common-block. Programs written in FORTRAN should minimize the use of this feature, since it may be a source of error that is difficult to resolve. The original toolbox uses common-blocks to initialize the parameters of various functions, an approach that the modernized code must handle differently.[6]

2.1.2 C

C is a general-purpose programming language created in 1972, and has since become one of the most used programming languages in the world [7]. It has many similarities to FORTRAN as it's a compiled language that uses many of the same default data types. C has also a thin abstraction layer and a low overhead compared with other high-level programming languages, which makes it useful for computationally intense programs. Programmers can, therefore, rewrite

FORTRAN-codes in C, and simultaneously keep the program close to its original design and performance.

2.1.3 Python

Different from Fortran and C, Python is an interpreted language which offers dynamically typing. There is no need for declarations of variables, because the data types are stated at runtime [8]. This feature, among others, make Python easy to learn and gives it intuitive syntax that facilitates productivity and readability. However, the flexibility of Python comes with lower performance compared with FORTRAN and C.

One of the most significant advantages of Python is the comprehensive standard library, and the fact that there are a vast number of third-party libraries and packages available. Among these, some modules make it possible to interface Python with other programming languages. A common way of boosting the performance of Python-programs, is to rewrite slow parts of the code in C and wrap them to pure Python (C_extensions). Topflow uses this technique to exploit the best of both worlds.[9]

The SciPy stack

The SciPy stack is a collection of libraries that support scientific computing in Python. It consists the 6 open-source packages NumPy, SciPy library, Matplotlib, IPython, SymPy and pandas.[10],[11]

NumPy stands for "Numerical Python" and is a specialized library that enables numerical computing in Python. Its primary usage is to define and do operations on large arrays and matrices. The NumPy array objects benefit from a range of native functions, which rely on well-optimized C-codes. These arrays are, therefore, more efficient than the inbuilt Python-lists.[12]

The SciPy library is another core package of the SciPy stack. It supports efficient numerical routines for linear algebra, with functions based on implemented standard algorithms [13]. In particular, Topflow relies on the function scipy.sparse.linalg.spsolve() to solve the sparse linear system $Ax = b$.

2.2 C-extensions

Topflow uses Python as the user-interface, but the program relies on underlying C-codes to boost the performance of the functions. There exist several tools that makes C-codes available in Python (also called wrapping), including Python-C-Api, SWIG, Ctypes, and Cython [14]. The specialization project by Hege Bruvik Kvandal [3] studied these tools, and the following thesis by the same author [2] implemented C-extensions by wrapping the codes with Ctypes, a foreign function library for Python [15]. All the previous work on Topflow uses Ctypes, and the implementations presented in this thesis are no exceptions. The rest of this section explains the process of writing C-extensions with Ctypes.

2.2.1 Shared libraries

Ctypes uses shared libraries to export the C code and wrap them to Python. Shared libraries are files containing code that several modules or applications can use simultaneously. The benefits

are that, unlike static linking, the linker does not copy the code into all the modules, which saves memory and disk storage space [16]

On a Windows system, shared libraries are called "dynamic linked libraries" (DLL), hence a ".dll"-file is created when building the library. The documentation "Building C/C++ Extensions on Windows" on python.org [17] states the following on building DLLs: "*Windows Python is built in Microsoft Visual C++; using other compilers may or may not work (though Borland seems to)*". Based on that information, this project utilizes the Visual Stuido C/C++ compiler for creating DLLs.

Figure 2.1 shows an example of a C-file (operators.c), which contains the functions "multiply" and "add". "multiply" takes in two arguments of the type "double" (decimals) and multiplies the numbers, while "add" takes in two integers and returns their summation. The "__declspec(dllexport)" statement in front of the function declaration is required by Windows to export the code from the DLL.[16]

operators.c

```
#include <stdio.h>

__declspec(dllexport) double multiply(double a, double b)
{
   return a*b;
}

__declspec(dllexport) int add(int a, int b){
   return a+b;
}
```

Figure 2.1: C-extension exported using __declspec(dllexport)

Topflow uses SCons, a free construction-tool [18], to build DLLs with the function *SharedLibrary('foo', ['f1.c', 'f2.c','f3.c',...])*. The first argument of the function sets the name of the DLL, and the second specifies which file(s) to include. The file which contains the "SharedLibrary"-function must be called "SConstruct.py" because SCons look for this particular filename when it builds the DLL [18]. Experiences from the specialization project [1] show that locating the SConstruct-file in the same directory as the C source files is the best practice, as it avoids linking problems. Figure 2.2 shows the SConstruct-file that put "add" and "multiply" from operators.c in a DLL named "cfunctions". The action that builds the DLL is to run the command "scons" in the terminal at the directory of the SConstruct-file. Figure 2.3 provides the output of this command, which shows that SCons successfully built the DLL "cfunctions", which now makes "add" and "multiply" accessible in Python. The next step is to help Python use these functions, by wrapping them to pure Python-code.

SConstruct.py

```
SharedLibrary(
    'cfunctions', 'operators.c'
)
```

Figure 2.2: The SConstruct file

```
PS C:\Users\Åsmund\PycharmProjects\acsolve> scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /Fooperators.obj /c operators.c /nologo
operators.c
link /nologo /dll /out:cfunctions.dll /implib:cfunctions.lib operators.obj
   Creating library cfunctions.lib and object cfunctions.exp
scons: done building targets.
```

Figure 2.3: Command prompt output from a successfully built DLL

2.2.2 Wrapper-functions

Ctypes provides C compatible data-types, and the ability to load DLLs and wrap the exported functions to pure Python. The wrapped functions are available from Python like any other callable. [15]

Figure 2.4 shows the Python-file "wrapperfunctions.py", which wraps the exported functions from the DLL "cfunctions" (defined in Figure 2.2). Wrapper-functions must specify the data types of arguments and return values, since Python uses dynamic typing and C does not. The example in Figure 2.4 starts off by importing ctypes and the required C-compatible data types, which in this case are c_double and c_int. The function "ctypes.cdll.LoadLibrary()" loads the DLL "cfunctions" and assigns it to the object "clib". "ctypes.cdll.LoadLibrary()" requires only one input-argument, which is the name or the path of the DLL. The script goes on to specify the data types of the wrapper functions. The function "add" uses integer-types for both its arguments and return values, while "multiply" uses double-types. The corresponding C compatible data types are "c_int" for integers and "c_double" for double. Finally, "wrapperfunctions.py" defines the python-functions "add" and "multiply", which are callable from the Python API. Figure 2.4 show how these functions take input arguments from Python, passes them on to the C-extension that does the work, and returns the result. The script "test.py" in Figure 2.5 tests the functions on a simple example. The output of the test, which is given in Figure 2.6, shows that both "add" and "multiply" are working as expected.

wrapperfunction.py

```
import ctypes
from ctypes import c_double
from ctypes import c_int

clib=ctypes.cdll.LoadLibrary('cfunctions')

clib.multiply.restype = c_double
clib.multiply.argtypes = [c_double, c_double]

clib.add.restype = c_int
clib.add.argtypes = [c_int, c_int]

def multiply(a,b):
    return clib.multiply(a,b)

def add(a,b):
    return clib.add(a,b)
```

Figure 2.4: The wrapper function

test.py

```
import wrapperfunction as wf

c=2
d=3

print('2 + 3 = ',wf.add(c,d))
print('2 * 3 = ',wf.multiply(c,d))
```

Figure 2.5: The Python file test.py testing the c-extensions "add" and "multiply"

```
PS C:\Users\Åsmund\PycharmProjects\acsolve> python test.py
2 + 3 =  5
2 * 3 =  6.0
```

Figure 2.6: Output of test.py

Ctypes supports NumPy array objects through the library Ctypeslib. In this project, the function numpy.ctypeslib.ndpointer is used to describe the return type and argument types of functions which uses NumPy arrays.[19]

2.3 Visual Studio Code

The free source editor Visual Studio Code (VSC) operates as the integrated development environment (IDE) for this project. It was recommended by the previous master's thesis [2] because it provides built-in support to a handful of programming languages, as well as extensions to others. This thesis uses the extensions for Python, C/C++, and Fortran, which provide support such as syntax highlighting and debugging. The IDE makes it possible to compare files side by side, which is very convenient when translating code between multiple languages. [20]

2.4 Black

Black is a code formatter for Python which structures the code automatically and makes the layout look the same regardless of the project [21]. The purpose of using Black in this project, is to make work consistent, and the collaboration as smooth as possible. The existing code by Leif Warland and Hege Bruvik Kvandal is formatted with Black, so is the updated program from this thesis. Visual Studio Code supports the formatter, and enables it to format the code every time a file is saved.

2.5 Power system analysis

Power system analysis is essential in the work of obtaining optimal operation of the existing power system, as well as planning expansions for the future. Topflow models the grid in a simplified way by line-diagrams and a per unit system, with parameters such as voltages, generated power and consumed power. Some of these parameters are known, others are calculated by

using well-established numerical techniques that gives good approximations. This section explains the theory behind the way Topflow models power systems and uses algorithms to perform simulations. The subsections 2.5.6, 2.5.7, and 2.5.9 are from the specialization project [1] by Åsmund Sælen.

2.5.1 Fundamental electrical equations

This subsection contains four fundamental equalities, which the next subsections will use to derive electrical models and algorithms. (2.1) gives the relationship between the admittance (Y), impedance (Z), conductance (G) and suscpetance (B). Ohm's law (2.2) states that the voltage across two points on a conductor is proportional to the current through the conductor. (2.3) is Krichoff's current law, which expresses that the total current that flows into a node is equal to the total currents that flows out. Kirchoff's voltage law (KVL), (2.4), states that the sum of the voltages in a closed loop is zero. Finally, (2.5) specifies that complex power is the product of voltage and the conjugate of the current.

$$Y = \frac{1}{Z} = G + jB \tag{2.1}$$

$$V = ZI \tag{2.2}$$

$$\sum I_i = 0 \tag{2.3}$$

$$\sum V_i = 0 \tag{2.4}$$

$$S = P + jQ = VI^* \tag{2.5}$$

2.5.2 Buses

The interconnection point between several components of the power system is called a "bus". A bus may be coupled to four different types of components: generators, loads, transmission lines and shunt elements. Figure 2.7 show the configuration of bus "i", which have all the four different component-types connected.

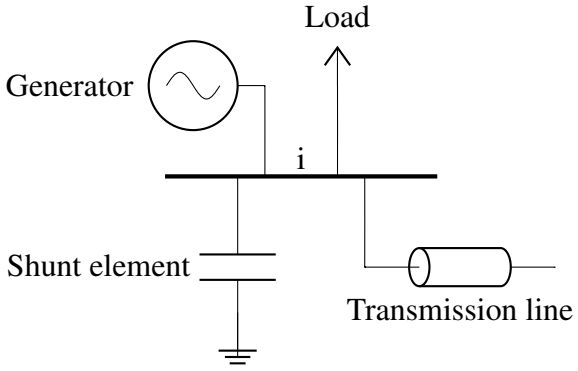


Figure 2.7: Bus "i"

Each bus in the system have four variables which is either known or unknown: voltage magnitude ($|V|$), voltage angle (θ), active power injected (P) and reactive power injected (Q). More specifically, $P + jQ$ is the differences between the generated power ($P_G + jQ_G$) and the demanded power ($P_D + jQ_D$), so that $P = P_G - P_D$ and $Q = Q_G - Q_D$. The buses are classified into different types based on the combination of the known and unknown variables, as shown in Table 2.1. Which variables that are known depends on whether the bus connects to a load or a generator.

Loads consume a certain amount of active and reactive power, hence load-connected buses are classified as PQ-buses since both P and Q are known. Generators controls the voltage magnitude and the active generated power at the bus, hence generator-connected buses are classified as PV-buses. The reactive power at a generator, Q_G, is dependent on the topology of the system, since transmission lines, transformers and loads consume reactive power. Furthermore, reactive power is used to control voltages in the system, which is why generators may consume or produce reactive power. Buses such as the one in Figure 2.7, which has both a generator and a load connected, are classified as PV-buses since $Q = Q_G - Q_D$ is unknown.

The power system model uses an arbitrary PV-bus (often the largest) as a reference to all other buses, hence called "reference bus" or "slack bus". The voltage magnitude of the slack bus is set at 1.0pu, and the angle at 0°. This bus holds a special role in the load-flow studies, as it balances the total power in the system and provides for the losses.

Bus type	**Known**	**Unknown**
Generator (PV)	$P, \|V\|$	Q, θ
Load (PQ)	P, Q	$\|V\|, \theta$
Slack	$\|V\|, \theta$	P, Q

Table 2.1: Bus types.

2.5.3 Transmission lines

Power is transferred between buses through transmission lines. The representation of these lines depends on the length, broadly categorized into short, medium and long lines. The program presented in this thesis applies the same branch-model for all lines, the nominal-π model, which is commonly used to represent medium length transmission-lines (80km-240km).

Observing Figure 2.8, the line has an impedance of Z_{ij} and a total line charging admittance, Y_{Sij}. By symmetry, the model lumps an equal portion of the total charging admittance on each end of the line. The real part of $Y_{Sij} = G_{S12} + JB_{Sij}$ is very small, in most cases 0, which is why B_{Sij} often is the only given value for the shunt element.

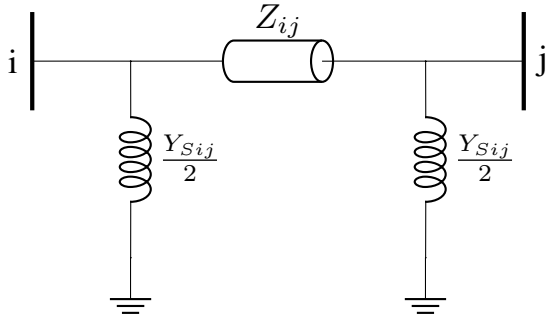


Figure 2.8: π-equivalent transmission line

2.5.4 Transformers

Transformers are modeled in the same branch model as section 2.5.3, with some modifications due to the tap ratio and the shift-angle. A transformer with tap ratio t and a phase-shifting angle θ_s is placed at the "from-end" of the line, as shown in Figure 2.9. The following derivation will show how to represent the transformer-branch on the same model as Figure 2.8.

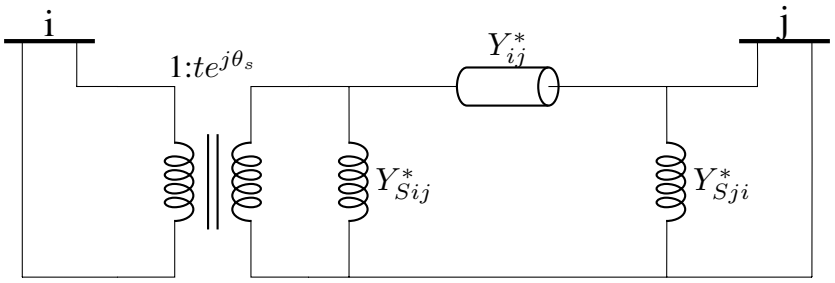


Figure 2.9: Transformer-branch

KCL (2.3) is applied to Figure 2.9 to obtain expressions for I_1 (2.6) and I_2 (2.7).

$$\frac{I_i}{te^{-j\theta_s}} = V_i te^{j\theta_s}(\frac{Y^*_S}{2} + Y^*_{ij}) - V_j Y^*_{ij} \tag{2.6}$$

$$I_j = -te^{j\theta_s}Y^*_{ij}V_i + V_j(\frac{Y^*_S}{2} + Y^*_{ij}) - te^{j\theta_s}V_2 Y^*_{ij} \tag{2.7}$$

This system of equation can be written as the augmented matrix shown in (2.8).

$$\begin{bmatrix} I_i \\ I_j \end{bmatrix} = \begin{bmatrix} t^2(Y^*_{ij} + \frac{Y^*_s}{2}) & -te^{-j\theta_s}Y^*_{ij} \\ -te^{j\theta_s}Y^*_{ij} & Y^*_{ij} + \frac{Y^*_s}{2} \end{bmatrix} \begin{bmatrix} V_i \\ V_j \end{bmatrix} \tag{2.8}$$

(2.8) shows that Figure 2.9 reduces to the π-equivalent model in Figure 2.8 by substituting the following parameters:

$$Z_{ij} = \frac{1}{te^{-j\theta_s}Y_{ij}^*}$$
$$Y_{Sij} = t^2(Y_{ij}^* + \frac{Y_s^*}{2}) - te^{-j\theta_s}Y_{ij}^*$$
$$Y_{Sji} = \frac{Y_s^*}{2}$$

Notice that a transformer with tap ratio t = 1 and shift angle θ_s = 0 is mathematically equivalent to a none-transformer branch.

2.5.5 Shunt Element

Shunt elements are inductors or capacitors connected to the power system to control the reactive power and thereby control the voltage. Line-losses increases if the loading level of the system is high, which leads to higher voltage drops. To prevent voltages below acceptable levels during such conditions, capacitor banks are switched on at weak busses to increase the voltage. These capacitors provide reactive power, which will increase the voltage.

Shunt elements can be connected either at a bus or on a line, but since reactive compensators have the greatest effect locally, these elements are most often bus-connected. This thesis converts data-sheets from MATPOWER [22] to initialize parameters of power systems. These sheets includes only bus-connected shunt elements, however, Topflow support both types of connections.

2.5.6 The problem formulation

Section 2.5.2 explained how buses are categorized by their known and unknown parameters. The goal of a power-flow study is to obtain good approximations for the unknown parameters by solving the so-called "load-flow equations". The derivation of these equations starts by applying Ohm's law (2.2) and Kirchoff's current law (2.3) to each bus in a system with N buses, and thereby obtain the matrix equation-system (2.9).

$$\begin{bmatrix} I_1 \\ \vdots \\ I_N \end{bmatrix} = Y_{bus} \begin{bmatrix} V_1 \\ \vdots \\ V_N \end{bmatrix} \tag{2.9}$$

V_i is the voltage at bus i, I_i is the sum of the line-currents flowing in and out of bus i, while Y_{bus} is the $N \times N$ admittance matrix shown in (2.10):

$$Y_{bus} = \begin{bmatrix} Y_{11} & \dots & Y_{1n} \\ \vdots & \ddots & \vdots \\ Y_{n1} & \dots & Y_{nn} \end{bmatrix} \tag{2.10}$$

(2.11) shows that the diagonal elements of the Y_{bus} are the sum of the line admittances and shunt admittances that are connected to the respective bus. The off-diagonal element Y_{ij} is the

negative value of the line admittance between bus i and j. In a large power system, most buses are not connected directly to another, which means that most element of the Y_{bus} are zero. This is the definition of a sparse matrix.[23]

$$Y_{ii} = \sum_{i \neq j}^{n} y_{ij} \tag{2.11}$$

$$Y_{ij} = -y_{ij} \tag{2.12}$$

The load-flow equations (2.13)-(2.15) are finally obtained by substituting I_i from (2.9) in the equation for complex power (2.5). The equations calculates the net power-injections at bus i. G_{ij}, B_{ij} and θ_{ij} are the conductance, susceptance and phase shift between bus i and j.

$$S_i = P_i + jQ_i = V_i \sum_{j}^{n} Y_{ij} V_j \tag{2.13}$$

$$P_i = |V_i| \sum_{j}^{n} |V_j| (G_{ij} cos(\theta_{ij}) + B_{ij} sin(\theta_{ij})) \tag{2.14}$$

$$Q_i = |V_i| \sum_{j}^{n} |V_j| (G_{ij} sin(\theta_{ij}) - B_{ij} cos(\theta_{ij})) \tag{2.15}$$

The load-flow equations are non-linear, and therefore solved numerically by approximating the unknown parameters. A common approach is to approximate the voltages by solving the equations that contain known values for P and Q. Section 2.5.7 and section 2.5.8 explains the theory of two algorithms that uses this approach to solve (2.14) and (2.15).

2.5.7 Newton Rapshon load flow

The first algorithm for solving the load-flow equations (2.14) and (2.15) is based on Newton's method. This method approximates the the solution of problem (2.16) through an iterative process. f is a function defined for the variable x, and c is a known constant. (2.17) shows Newton's method for the problem (2.16), and is derived from the first order Taylor-series for the function f. f^k and x^k are the approximations of f and x in the kth iteration.[23]

$$f(x) = c \tag{2.16}$$

$$x^{k+1} - x^k = \frac{c - f(x^k)}{f'(x)} \tag{2.17}$$

The method start with the iteration-counter $k = 0$, and an initial guess of the value of x: x^0. (2.17) gives the formula for calculating the next approximation for x: x^1. The iterative process

continues and gives successively better approximations, until the error $c - f(x^k)$ is of adequate size. The speed of convergence is related to how close the initial guess is to the final solution.

The Newton-Rapshon load-flow (2.19) is a multi-dimensional version of (2.17), which solves the matrix equation system (2.18). In this version, f is a vector that consists of the known active (P) and reactive (Q) power injections in a power system. x is a vector that contains the angles (θ) and magnitudes ($|V$) of the unknown voltages: $\begin{bmatrix} \theta \\ |V| \end{bmatrix}$. c is a vector of the scheduled active (P_{sch}) and reactive (Q_{sch}) powers and, finally, J is the Jacobian matrix of the vector f. (2.20) show the definition of this matrix, which consists of the derivatives of (2.14) and (2.15) with respect to θ and $|V|$. The sub-matrices $\left[\frac{\partial P}{\partial \theta}\right]$, $\left[\frac{\partial P}{\partial |V|}\right]$, $\left[\frac{\partial Q}{\partial \theta}\right]$ and $\left[\frac{\partial Q}{\partial |V|}\right]$ are often referred to as $J1$, $J2$, $J3$ and $J4$.

$$\begin{bmatrix} P \\ Q \end{bmatrix} = \begin{bmatrix} P_{sch} \\ Q_{sch} \end{bmatrix} \tag{2.18}$$

$$\begin{bmatrix} \theta^{k+1} - \theta^k \\ |V|^{k+1} - |V|^k \end{bmatrix} = J^{-1} \begin{bmatrix} P_{sch} - P^k \\ Q_{sch} - Q^k \end{bmatrix} \tag{2.19}$$

$$J = \begin{bmatrix} \frac{\partial P}{\partial \theta} & \frac{\partial P}{\partial |V|} \\ \frac{\partial Q}{\partial \theta} & \frac{\partial Q}{\partial |V|} \end{bmatrix} \tag{2.20}$$

The inverse of the Jacobian matrix can be found in a number of ways, including Gauss-Jordan elimination and Gauss elimination [24]. However, the process of obtaining the inverse of a large matrix is time consuming, which is why programs commonly uses LU-factorization to solve (2.19). [4], [23]

Introducing $T_{ij} = G_{ij}cos(\theta_{ij}) + B_{ij}sin(\theta_{ij})$ and $U_{ij} = G_{ij}sin(\theta_{ij}) - B_{ij}cos(\theta_{ij})$ is convenient when obtaining the equations for calculating the derivatives in (2.20). The set of equations (2.21) shows how these derivatives are calculated. The variables $G_{ii} = \sum_{j \neq i}(G_{Sij} + G_{ij})$ and $B_{ii} = B_{Si} + \sum_{j \neq i}(B_{Sij} + B_{ij})$, where G_{Sij} and B_{Sij} are the conductance and susceptance of the line-shunt elements between bus i and bus j. B_{Si} is the imaginary part of the bus-connected shunt elements at bus i.

$$\begin{aligned}
\frac{\partial P_i}{\partial \theta_i} &= |V_i| \sum_{j \neq i} |V_j| U_{ij} & \frac{\partial P_i}{\partial |V_i|} &= 2|V_i|G_{ii} - |V_i| \sum_{j \neq i} |V_j| T_{ij} \\
\frac{\partial P_i}{\partial \theta_j} &= -|V_i||V_j|U_{ij} & \frac{\partial P_i}{\partial |V_j|} &= -|V_i|T_{ij} \\
\frac{\partial Q_i}{\partial \theta_i} &= -|V_i| \sum_{j \neq i} |V_j| T_{ij} & \frac{\partial Q_i}{\partial |V_i|} &= 2|V_i|B_{ii} - |V_i| \sum_{j \neq i} |V_j| U_{ij} \\
\frac{\partial Q_i}{\partial \theta_j} &= |V_i||V_j|T_{ij} & \frac{\partial Q_i}{\partial |V_j|} &= -|V_i|U_{ij}
\end{aligned} \tag{2.21}$$

The number of elements in the vectors from (2.18) are dependent on the topology of the power system. In a system with the total number of n buses and g generators (including slack

bus), the number of equations of the type (2.14) is equal to $n-1$, while the number of equations of the type (2.15) is equal to $n-g$. It follows mathematically that J is a $(2n-g-1)\times(2n-g-1)$ matrix.

Figure 2.10 shows an example of a three-bus system that consists of one PQ-bus, one PV-bus, and one slack bus. The known parameters are given for each bus.

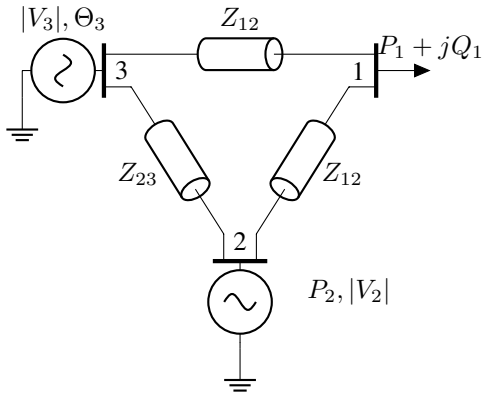


Figure 2.10: Three bus system

(2.22), (2.23) and (2.24) show how to write (2.18), (2.19) and (2.20) for the three-bus system in Figure 2.10.

$$\begin{bmatrix} P_1 \\ P_2 \\ Q_1 \end{bmatrix} = \begin{bmatrix} P_{1(sch)} \\ P_{2(sch)} \\ Q_{1(sch)} \end{bmatrix} \tag{2.22}$$

$$\begin{bmatrix} \theta_1^{k+1} - \theta_1^k \\ \theta_2^{k+1} - \theta_2^k \\ |V|_1^{k+1} - |V|_1^k \end{bmatrix} = J^{-1} \begin{bmatrix} P_{(1)sch} - P_1^k \\ P_{(2)sch} - P_2^k \\ Q_{(1)sch} - Q_1^k \end{bmatrix} \tag{2.23}$$

$$J = \begin{bmatrix} \frac{\partial P_1}{\partial \theta_1} & \frac{\partial P_1}{\partial \theta_2} & \frac{\partial P_1}{\partial |V|_1} \\ \frac{\partial P_2}{\partial \theta_1} & \frac{\partial P_2}{\partial \theta_2} & \frac{\partial P_2}{\partial |V|_1} \\ \frac{\partial Q_1}{\partial \theta_1} & \frac{\partial Q_1}{\partial \theta_2} & \frac{\partial Q_1}{\partial |V|_1} \end{bmatrix} \tag{2.24}$$

The advantages of the Newton-Rapshon method is that the region of convergence is large, and the number of iterations are few (if it converges). However, the speed of convergence is relatively slow if the initial guess is of poor choice. The Guass-Siedel method is another algorithm with the advantage of getting to the correct region fast, while it converges slow compared to Newton-Rapshon method. Some load-flow solutions are found by combining the two: using Guass-Siedel to get to the right region, followed by Newton-Rapshon to obtain the final solution. [4]

2.5.8 Fast-decoupled load-Flow

Another widely-used technique to solve the load-flow equations (2.14) and (2.15) is the Fast-decoupled load-low (FDLF). As the name suggests, it is faster and simpler than the regular Newton-Rapshon load flow, as it uses approximations to:

1. Decouple the system of equations into active ($P\theta$) and reactive ($Q|V|$) sub-problems.

2. Construct constant matrices which are independent of the voltage magnitudes and angles.

Despite apparent significant simplifications, the method has shown to perform remarkably well. The paper [25] showed that the success of the FDLF is related to the iteration scheme, which updates either angels or magnitudes after each sub-problem. Furthermore, it showed that some of the approximations commonly used to derive the FDLF are unnecessary due to the successive way of updating the magnitudes and angles. These approximations are still seen in the literature today, even after the publication of [25]. This section will explain three different versions of FDLF and give a brief overview of the findings in [25].

Derivations

The derivation of the method starts with the first iteration of Newton-Rapshon using flat start, shown in (2.25)

$$\begin{bmatrix} H & N \\ M & L \end{bmatrix} \begin{bmatrix} \Delta\theta \\ \Delta|V| \end{bmatrix} = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} \tag{2.25}$$

The matrices H, N, M, L corresponds to the four submatrices of the Jacobian matrix calculated at flat start ($|V| = 1.0$ and $\theta = 0$). Note that the flat start results in H and L being the imaginary part of the admittance matrix $Y_{bus} = G + JB$, in the structures of $\frac{\partial P}{\partial \theta}$ and $\frac{\partial Q}{\partial |V|}$ respectively.

The decoupling of the problem, so that $\Delta\theta$ and $\Delta|V|$ can be calculated separately, can be done in several ways:

1. The reactive power equations of (2.25) are subtracted by the active power equations, which are premultiplied with MH^{-1}:

$$\begin{bmatrix} H & N \\ 0 & L_{eq} \end{bmatrix} \begin{bmatrix} \Delta\theta \\ \Delta|V| \end{bmatrix} = \begin{bmatrix} \Delta P \\ \Delta Q - MH^{-1}\Delta P \end{bmatrix} \tag{2.26}$$

2. The active power equations of (2.25) are subtracted by the reactive power equations, which are premultiplied with NL^{-1}:

$$\begin{bmatrix} H_{eq} & 0 \\ M & L \end{bmatrix} \begin{bmatrix} \Delta\theta \\ \Delta|V| \end{bmatrix} = \begin{bmatrix} \Delta P - NL^{-1}\Delta Q \\ \Delta Q \end{bmatrix} \tag{2.27}$$

3. Both operations (1 and 2) are applied on system (2.25):

$$\begin{bmatrix} H_{eq} & 0 \\ 0 & L_{eq} \end{bmatrix} \begin{bmatrix} \Delta\theta \\ \Delta|V| \end{bmatrix} = \begin{bmatrix} \Delta P - NL^{-1}\Delta Q \\ \Delta Q - MH^{-1}\Delta P \end{bmatrix} \tag{2.28}$$

Where the new equivalent matrices are given by:

$$H_{eq} = H - NL^{-1}M \tag{2.29}$$

$$L_{eq} = L - MH^{-1}N \tag{2.30}$$

Solving the equation-systems

Algorithms can be used to solve the equation-systems (2.26) - (2.28) without introducing any major approximations. For example: the Primal-method solves (2.26) in the following way:

1. Initialize iteration count: $k = 0$
2. Calculated the active power mismatches, and compute angle corrections: $\Delta\theta^{(k)} = \frac{H^{-1}\Delta P^{(k)}(|V|^{(k)},\theta^{(k)})}{|V|^{(k)}}$
3. Update the angles: $\theta^{(k+1)} = \theta^{(k)} + \Delta\theta^{(k)}$
4. Calculate the reactive power mismatches, and compute magnitude corrections: $\Delta|V|^{(k)} = \frac{L_{eq}^{-1}\Delta Q^{(k)}(|V|^{(k)},\theta^{(k+1)})}{|V|^{(k+1)}}$
5. Update the magnitudes: $|V|^{(k+1)} = |V|^{(k)} + \Delta|V|^{(k)}$
6. $k = k + 1$, go to step 2.

Multiple algorithms can solve the equation-system (2.26) - (2.28) in similar ways as the Primal-method. Appendix(A) provides two other versions, namely the Standard-method and the Dual-method. In contrast to the Primal-method, the Dual starts with a $Q\theta$-iteration and ends with a $P|V|$-iteration. The Dual-method uses the sub-matrices H_{eq} and L, while the Standard-method uses H_{eq} and L_{eq}. The methods use approximations to obtain these matrices, which will be discussed in the next section.

Approximations

It is essential to notice that the derivation of equations (2.26)-(2.28) uses no approximations, merely matrix operations. Each of these equations is, therefore, the same as the first iteration of Newton-Rapshon using flat-start. It's assumed that the voltage magnitudes and angles are close to the flat-start values, so that $|V| \approx 1pu$, $sin\theta \approx 0$ and $cos\theta \approx 1$. These assumptions keep the sub-matrices constant throughout the load-flow.

The paper [25] showed that the impacts of the matrices M and N are automatically taken into account when solving the Primal-method successively by performing a $P\theta$ iteration and use the updated angles in the following $Q|V|$ sub-problem. The same is true for the Dual-method; therefore, there is no need to neglect these matrices, which was the standard in the traditional derivation of the FDLF methods. [25] [26] [27]

Submatrix L_{eq} is the submatrix L with b_{ij} substituted by $1/x_{ij}$ (which is why the primal method is often called the BX-version; it consists of H and L_{eq}). This substitution is exactly true for radial systems and systems with constant r/x ratios, because the operation performed to obtain (2.26) naturally cancel out the resistances in L_{eq}. It is a common mistake in the literature, even after the publication of [25], to assume that the resistances are neglected; this is only an apparent approximation. For systems which are not radial nor have constant r/x ratios, on the other hand, the representation of L_{eq} is an approximation, but still an excellent one. [25]

The same holds for H_{eq}, its obtained by calculating H when b_{ij} is substituted by $1/x_{ij}$. Additional approximations need to be done, namely ignoring the effect of PV-buses and shunts. These approximations are why the Primal method performs better than the Dual in most cases.

Summary

To summarize, the following approximations are made for the FDLF-mehods:

1. $|V_i| = 1$
2. $sin(\theta_{ij}) = 0$
3. $cos(\theta_{ij}) = 1$
4. The effect of PV-buses and shunts ar neglected when forming H_{eq} (XB- and XX-version only)
5. L_{eq} is the submatrix L calculated with b_{ij} substituted by $1/x_{ij}$
6. H_{eq} is the submatrix H calculated with b_{ij} substituted by $1/x_{ij}$

   (Note: if the system is radial or have constant r/x ratios, 6. and 7. are not approximations, but exactly true.)

With the above approximations, the general FDLF-equations become (2.31). Table 2.2 show the meaning of B' and B'', along with the sequence of the $P\theta$- and $Q|V|$-iterations.

$$\Delta\theta = \frac{B'^{-1}\Delta P}{|V|} \qquad\qquad \Delta|V| = \frac{B''^{-1}\Delta Q}{|V|} \tag{2.31}$$

Algorithm	B'	B''	Iteration scheme
Primal / BX	H	L_{eq}	$P\theta$ - $Q\|V\|$
Dual / XB	H_{eq}	L	$Q\|V\|$ - $P\theta$
Standard / XX	H_{eq}	L_{eq}	$P\theta$ - $Q\|V\|$

Table 2.2: B' and B'' for the different algorithms

It can be concluded from the material presented in this section that the resulting method is simpler and much faster than the Newton-Rapshon load flow, and it has a wide range of practical applications [26]. Despite its apparent significant simplifications, the method has shown to perform remarkably well. The paper [25] showed that the success of the decoupled method is very much related to the iteration scheme, where angels/magnitudes are updated after each sub-problem. Values which are apparently neglected in one iteration, are actually taken into account in the next. The various versions have shown to perform differently dependent on the system; on most systems the Primal method is favoured, however the Dual method has shown quick convergence in some cases where the Primal method did not. The Standard version is certainly the easiest of the three, since both sub-matrices are built only by the inverse of the line reactances. Knowing about the different versions of the FDLF gives the method more flexibility and greater performance, which is why all versions are implemented in the toolbox presented in this thesis.

2.5.9 Reactive power limitation

The operation of a synchronous generator is limited by three limits: the armature current heating, the field current heating, and the end region heating. Together they form the reactive capability chart of a generator, which show the area the generator can deliver power safely.[28]

Figure 2.11 shows a typical capability chart of a synchronous generator, which describes how the limitations discussed above restricts the reactive generation. The reactive generations of the PV-buses in a power system are unknown, and calculated in each iteration of a load-flow by estimating values for the voltages. These estimations might violate the capability requirements, if the limitations are not taken into account. A common approach to enforce reactive limits, is to consider PV-buses that violate the capability requirements as PQ buses, and set the reactive power at the limit. It is necessary to keep track on which buses are real PQ-buses and which are PV-buses (VAR-limited).**pv_var_limit**

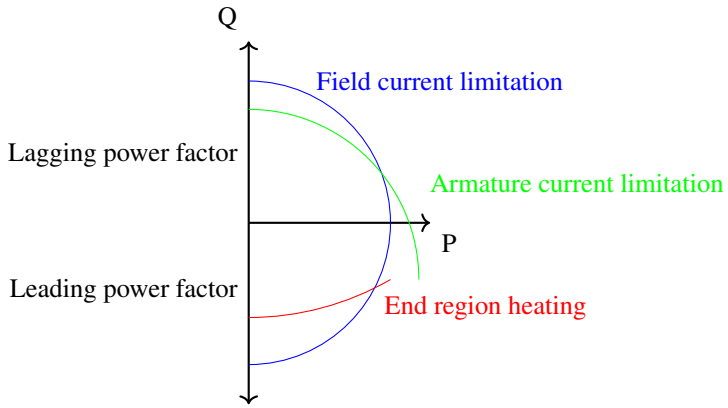


Figure 2.11: Reactive capability chart of a synchronous generator

2.6 Sparse matrices

A matrix which mostly contains elements of zero value is called a sparse matrix. This phenomenon often appears in engineering and scientific computing, especially when solving partial differential equations. Building the full representation of a sparse matrix is inefficient, since the program will spend most of the execution time on storing and processing zeros. Most of the buses in a realistic power system are not connected directly to each other; hence the Y_{bus} is sparse. Therefore, the advantages of exploiting the sparsity of such a matrix are of great importance for the code presented in this thesis. This section offers three different ways to store sparse matrices, along with their advantages and disadvantages. Figure 2.12 will be used as an example to explain the different methods.

2.6.1 Coordinated list (COO-format)

The first method consists of the three arrays *Data, Row* and *Col*, which are all of equal size. The names are quite self-explanatory; the Data-array contains the values of the non-zero elements, in any order, the Row- and Col-array specifies which row and column this element holds in the dense matrix. The arrays below shows Figure 2.12 on COO-format.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 9 & 8 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Figure 2.12: Sparse matrix example

$$\begin{aligned} Data &= \begin{bmatrix} 1 & 2 & 3 & 4 & 7 & 9 & 8 & 5 & 6 \end{bmatrix} \\ Row &= \begin{bmatrix} 1 & 1 & 2 & 3 & 3 & 4 & 4 & 5 & 6 \end{bmatrix} \\ Col &= \begin{bmatrix} 1 & 6 & 3 & 2 & 5 & 4 & 5 & 1 & 6 \end{bmatrix} \end{aligned}$$

Figure 2.13: Figure 2.12 on COO-format

The Python library scipy.sparse supports the coo-format with the class *coo_matrix*. The advantages of this format are efficient conversion to and from other sparse formats, especially CSC and CSR. The disadvantages are that the format does not support arithmetic operations or slicing. The intended usage of COO is to construct sparse matrices and facilitate fast conversion among other formats. It's convenient to construct the matrix of the equation $Ax = b$ on this format before converting it, since different solvers require different formats.[29]

2.6.2 Compressed sparse formats (CSC- and CSR-format)

Formats for efficient access and matrix operation are stored on compressed sparse column (CSC) or compressed sparse row (CSR) formats. The CSC-format consist of three arrays: *Data* is the (top to bottom, then left to right) non-zero elements of the dense matrix, *Row* is the row-position of a given element in *Data*, *Colptr* holds one element per column, which specifies the position in *Data* where the given column starts. The arrays below show Figure 2.12 on CSC-format.[30]

$$\begin{aligned} Data &= \begin{bmatrix} 1 & 2 & 3 & 4 & 7 & 9 & 8 & 5 & 6 \end{bmatrix} \\ Row &= \begin{bmatrix} 1 & 1 & 2 & 3 & 3 & 4 & 4 & 5 & 6 \end{bmatrix} \\ Colptr &= \begin{bmatrix} 1 & 4 & 3 & 6 & 5 & 2 \end{bmatrix} \end{aligned}$$

Figure 2.14: Figure 2.12 on CSC-format

The CSR-format is similar to the CSC-format, however this variant compresses the row information instead of the column information. *Data* is the (left to right, then top to bottom) non-zero elements of the dense matrix, *Col* is the column-position of a given element in *Data*, *Rowptr* holds one element per row, which specifies the position in *Data* where the given row starts. The arrays below show Figure 2.12 on CSR-format.[31]

$$
\begin{aligned}
Data &= \begin{bmatrix} 1 & 2 & 3 & 4 & 7 & 9 & 8 & 5 & 6 \end{bmatrix} \\
Col &= \begin{bmatrix} 1 & 6 & 3 & 2 & 5 & 4 & 5 & 1 & 6 \end{bmatrix} \\
Rowptr &= \begin{bmatrix} 1 & 3 & 4 & 6 & 8 & 9 \end{bmatrix}
\end{aligned}
$$

Figure 2.15: Figure 2.12 on CSR-format

Both CSC and CSR are supported in Python through the classes *csc_matrix* and *csr_matrix* from the library *scipy.sparse*. The advantages of these formats are efficient arithmetic operations and fast matrix-vector products. The CSC format facilitates efficient column slicing, while the CSR format provides efficient row slicing. The solver used in this toolbox requires the sparse representation of the Jacobian matrix to be on either CSC- or CSR-format. Modifications to matrices on these formats are on the other hand expensive, which is why it's common to use other formats to construct and store sparse matrices before converting them to CSR or CSC for arithmetic operations.[30][31]

Chapter 3

Topflow user guide

Topflow is a toolbox for power system analysis. The current program contains functions for running load-flows on power-systems, which the user can initialize by specifying the input-data in Excel-files. This chapter provides an overview of the toolbox and explains how to setup the program and make use of the implemented load-flow methods.

3.1 System requirements

The items needed to use the program are:

1. Python
2. Visual Studio C/C++ compiler

Topflow is only tested with the Visual Studio C/C++ compiler, other compilers may or may not work.

3.2 Installation

In addition to the system requirements, it is necessary to have various Python packages installed to run the code. These can be installed with the "*pip install*"-command in the cmd/terminal on a computer that has successfully installed Python. The full list of required packages, along with their installation-command, are listed below.

1. numpy: *pip install numpy*
2. scipy: *pip install scipy*
3. scons: *pip install scons*
4. openpyxl: *pip install openpyxl*

The end-user can skip these installations, and automatically manage the dependencies by installing Topflow. Since Topflow is not an open-source project, users must acquire the package from one of the contributors, and install it locally. Locally installations are done by feeding the full path of the package to pip:

```
pip install -e  c:\\users\\aasel\\programmering\\topflow
```

Figure 3.1: How to install Topflow locally

3.3 Running load-flows

The application of Topflow is to run simulations and solve load-flow problems. The workflow of a load-flow study includes: 1) Preparing the input data in an Excel-file. 2) Initializing the parameters. 3) Solving the case with the proper load flow method(s). 4) Viewing the result of the simulation. The remaining subsections will go through these steps for a 14-bus example case.

3.3.1 Input Data

Topflow can only read Excel-files of a specific format. The format contains four separate spreadsheets, which the user must name as follows:

1. Case-identification
2. Bus-data
3. Generator-data
4. Line-data

The sheets must hold these names because this is how the routines that initialize the system identifies the different parameters. Common to all the spreadsheets is that they structure the data on a specific matrix-form, where the first row holds the names of the parameters. Similar to the spreadsheet-names, the user must name these parameters as specified in the next subsections.

Case-identification

The first data record consists of only two parameters; IC and SBASE. IC = 0 states that the case is a base case; hence simulations do not permanently change the data. If IC = 1, the case is not a base case, and simulations may alter the system parameters. SBASE gives the system base in MVA.

Figure 3.2 show how the Case-identification sheet is structured as a 2×2 matrix. The routine that reads the sheet identifies the correct information by searching for "IC". The matrix may start in other cells than A1, but the matrix structure can not be changed.

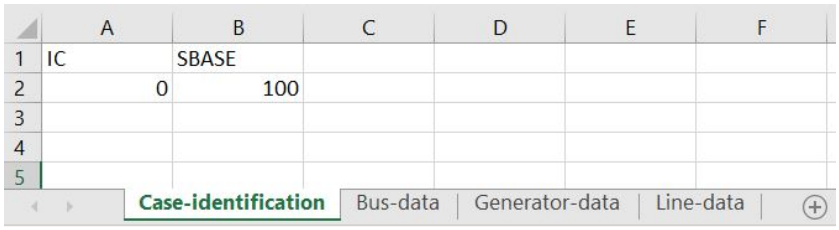


Figure 3.2: The Case-identification data sheet for the 14-bus

Bus-data

The bus-data record contains information about voltages, shunt elements, and loads at the buses. Table 3.1 gives a full list of the parameters. Notice how the consumed power for each bus (PLOAD and QLOAD) is present, while the generated power is not. That information belongs to the "Generator-data" spreadsheet.

I	External bus numbers
NAME	Name of the buses
IDE	List of the buscodes for all buses; 1-PQ, 2-PV, 3-Slack
PLOAD	Active load [MW]
QLOAD	Reactive load[MVAr]
GL	Active component of bus-connected shunt element [MW]
BL	Reactive component of bus-connected shunt element [MVAr]
AREA	Area postions for each bus
VM	Voltage magnetudes [pu]
VA	Voltage angles [radians]
PLOAD	Active load [MW]
QLOAD	Reactive load[MVAr]
BASEKV	Base voltage at each bus [kV]
ZONE	Zone postition for each bus
VMAX	Maximum voltage allowed [pu]
VMIN	Minimum voltage allowed [pu]

Table 3.1: The parameters in the Bus-data record

Similarly to the Case-data, the user must type the bus-data in a matrix where the first row contains the name of the parameters. The routine that reads the bus-data sheet identifies the matrix by searching for "I", and reads the data as long as the column of I is not empty. The matrix may start at an arbitrary cell, but the structure must be of the one in Figure 3.3, and the cell below the last bus-number must be empty.

Topflow reads the information of each row consecutively and initializes the corresponding parameters. The program uses internal bus-numbers when performing operations, which simply are the sequence of which the busses were added to the program, starting at 0. Conversions between external and internal bus-numbers are done by the functions ext2int and int2ext.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	I	NAME	IDE	PLOAD	QLOAD	GL	BL	AREA	VM	VA	BASEKV	ZONE	VMAX	VMIN	
2	1	'Bus 1	3	0	0	0	0	1	1,06	0	0	1	1,06	0,94	
3	2	'Bus 2	2	21,7	12,7	0	0	1	1,045	-4,9826	0	1	1,06	0,94	
4	3	'Bus 3	2	94,2	19	0	0	1	1,01	-12,725	0	1	1,06	0,94	
5	4	'Bus 4	1	47,8	-3,9	0	0	1	1,018	-10,313	0	1	1,06	0,94	
6	5	'Bus 5	1	7,6	1,6	0	0	1	1,02	-8,7739	0	1	1,06	0,94	
7	6	'Bus 6	2	11,2	7,5	0	0	1	1,07	-14,221	0	1	1,06	0,94	
8	7	'Bus 7	1	0	0	0	0	1	1,062	-13,36	0	1	1,06	0,94	
9	8	'Bus 8	2	0	0	0	0	1	1,09	-13,36	0	1	1,06	0,94	
10	9	'Bus 9	1	29,5	16,6	0	19	1	1,056	-14,939	0	1	1,06	0,94	
11	10	'Bus 10	1	9	5,8	0	0	1	1,051	-15,097	0	1	1,06	0,94	
12	11	'Bus 11	1	3,5	1,8	0	0	1	1,057	-14,791	0	1	1,06	0,94	
13	12	'Bus 12	1	6,1	1,6	0	0	1	1,055	-15,076	0	1	1,06	0,94	
14	13	'Bus 13	1	13,5	5,8	0	0	1	1,05	-15,156	0	1	1,06	0,94	
15	14	'Bus 14	1	14,9	5	0	0	1	1,036	-16,034	0	1	1,06	0,94	
16															

Case-identification | **Bus-data** | Generator-data | Line-data

Figure 3.3: The Bus-data sheet for the 14-bus IEEE system

Generator data

Table 3.2 gives the complete description of the generator-parameters. The user must add the data in a separate sheet on the form shown in Figure 3.4. The same directions for inserting the data applies for the generator-sheet: the matrix must be of correct structure, and the cell below the last bus-number must be empty.

I	External bus-numbers
PG	Active power for generators [MW]
QG	Reactive power for generators [MVAr]
QMAX	Maximum reactive generation [MVAr]
QMIN	Minimum reactive generation [MVAr]
VS	Voltage setpoint for the generators [pu]
MBASE	MVA base for the machines (Default is MBASE = SBASE)
STAT	Status of the generator (in service: 1, else: 0)
PMAX	Maximum active generation [MW]
PMIN	Minimum active generation [MW]

Table 3.2: The parameters in the Generator-data record

	A	B	C	D	E	F	G	H	I	J	K
1	I	PG	QG	QMAX	QMIN	VS	MBASE	STAT	PMAX	PMIN	
2	1	232,4	-16,9	10	0	1,06	100	1	332	0	
3	2	40	42,4	50	-40	1,045	100	1	140	0	
4	3	0	23,4	40	0	1,01	100	1	100	0	
5	6	0	12,2	24	-6	1,07	100	1	100	0	
6	8	0	17,4	24	-6	1,09	100	1	100	0	
7											

Case-identification | Bus-data | **Generator-data** | Line-data

Figure 3.4: The Generator-data sheet for the 14-bus IEEE system

Line data

The last spreadsheet contains the data for the transmission-lines, which also includes information about transformers. Table 3.3 gives the full description of the line-parameters, and Figure 3.5 show how the data must is structured in a Excel-file.

I	External bus-numbers (from bus)
J	External bus-numbers (to bus)
R	Line resistance [pu]
X	Line reactances [pu]
B	Total line charging susceptance [pu]
RATEA	phase A current rating [MVA]
RATEB	phase b current rating [MVA]
RATEC	phase C current rating [MVA]
GI	Conductance of line shunt placed at "I" end [pu]
BI	Suceptance of line shunt placed at "I" end [pu]
GJ	Conductance of line shunt placed at "J" end [pu]
BJ	Suceptance of line shunt placed at "J" end [pu]
RATIO	Ratio of the transformer
ANGLE	shift angle of transformers
STATUS	Indicates status for all lines (In service: 1)

Table 3.3: The parameters in the Line-data record

	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	I	J	R	X	B	RATEA	RATEB	RATEC	GI	BI	GJ	BJ	RATIO	ANGLE	STAT	
2	1	2	0,01938	0,05917	0,0528	0	0	0	0	0	0	0	0	0	1	
3	1	5	0,05403	0,22304	0,0492	0	0	0	0	0	0	0	0	0	1	
4	2	3	0,04699	0,19797	0,0438	0	0	0	0	0	0	0	0	0	1	
5	2	4	0,05811	0,17632	0,034	0	0	0	0	0	0	0	0	0	1	
6	2	5	0,05695	0,17388	0,0346	0	0	0	0	0	0	0	0	0	1	
7	3	4	0,06701	0,17103	0,0128	0	0	0	0	0	0	0	0	0	1	
8	4	5	0,01335	0,04211	0	0	0	0	0	0	0	0	0	0	1	
9	4	7	0	0,20912	0	0	0	0	0	0	0	0	0,978	0	1	
10	4	9	0	0,55618	0	0	0	0	0	0	0	0	0,969	0	1	
11	5	6	0	0,25202	0	0	0	0	0	0	0	0	0,932	0	1	
12	6	11	0,09498	0,1989	0	0	0	0	0	0	0	0	0	0	1	
13	6	12	0,12291	0,25581	0	0	0	0	0	0	0	0	0	0	1	
14	6	13	0,06615	0,13027	0	0	0	0	0	0	0	0	0	0	1	
15	7	8	0	0,17615	0	0	0	0	0	0	0	0	0	0	1	
16	7	9	0	0,11001	0	0	0	0	0	0	0	0	0	0	1	
17	9	10	0,03181	0,0845	0	0	0	0	0	0	0	0	0	0	1	
18	9	14	0,12711	0,27038	0	0	0	0	0	0	0	0	0	0	1	
19	10	11	0,08205	0,19207	0	0	0	0	0	0	0	0	0	0	1	
20	12	13	0,22092	0,19988	0	0	0	0	0	0	0	0	0	0	1	
21	13	14	0,17093	0,34802	0	0	0	0	0	0	0	0	0	0	1	
22																

Case-identification | Bus-data | Generator-data | **Line-data**

Figure 3.5: The Line-data sheet for the 14-bus IEEE system

3.3.2 Initializing a Case

At the core of the toolbox is the class "Case". A Case-object encapsulates all the system information needed in load flow studies and contains various class functions for initializing the data and performing arithmetic operations. Simulation-functions will typically take in an initialized Case-object to perform load-flows on the system. Figure 3.6 shows the easiest way of initializing a Case-object: to pass the path of the Excel-file that contains the input-data when creating an instance of the class.

```
>>> import topflow as tf
>>> case1 = tf.Case('C:/Users/aasel/programmering/topflow/topflow/example_cases/case14.xlsx')
```

Figure 3.6: How to load the input-data from the Excel-file that contains the 14-bus system

The object "case1" from Figure 3.6 now contains the system parameters for the 14-bus system. The user can verify this with the Case-class function "print_all", which prints all system parameters to the terminal.

```
>>> case1.print_all()
```

Figure 3.7: Print all the parameters of case1 to screen

Another way of initializing a Case is through the function topflow.example_case(). This function grants access to several example-cases available in Topflow. The 14-bus system used in this user guide is one of them, and Figure 3.8 shows how to load that particularly system by its case-name: 'case14'. A full list of example cases is available in section 5.4.

```
>>> case1 = tf.example_case('case14')
```

Figure 3.8: How to load case14 with the function "topflow_example_case()"

Notice that only cases stored in the Topflow example-library are available through this method, customized cases must be loaded with the approach from Figure 3.6.

3.3.3 Solving the Case

The simulation-functions that perform analysis on the system takes in an initialized Case-object along with the desired settings. Default options apply if personal settings are not specified. The function "topflow.loadflow()" performs a loadflow on the system, and supports regular Newton-Rapshon load flow as well as several fast-decoupled load-flow versions. Figure 3.9 shows how to run a Newton-Rapshon load-flow with default settings on case1 from section 3.3.2.

```
>>> tf.loadflow(case1)
```

Figure 3.9: A regular Newton-Rapshon load-flow

The simulation-functions do not alter the input object in any way, which means that multiple functions can use the same Case-object as a base-case.

3.3.4 Accessing the Results

The result of a loadflow simulation is accessible through three different ways:

1. The result is pretty printed to the screen.
2. A Case-object containing the final results is returned from the simulation-functions.
3. The data of a simulation is saved to an Excel-file.

While 2. and 3. are optional, the result is by default printed to the screen, showing the number of iterations for convergence, and the execution time of the simulation. Figure 3.10 show the output of the simulation from subsection 3.3.3.

```
Convergence.
Number of iterations: 3 in 0.012997627258300781 seconds
```

Figure 3.10: A typical default terminal-output from running a load-flow

The loadflow-function from Figure 3.9 returns the result as a Case-object, which allows the user to use the data in subsequent analysis. Figure 3.11 show how the solved values of the loadflow-function are assigned to the instance "result1". The Case-class function "get()" makes it easy to access the parameters of that instance. get() takes in three arguments where the first specifies the component name (either 'bus', 'gen' / 'generator' or 'line'), the second is the component number(s), and the third is the parameter-name. It is possible to access multiple parameters with the same function call by defining the second argument as a list of numbers. The function will then return the corresponding parameters in the same order as the input-list. Figure 3.12 shows how to access the solved values in the instance return1 form fig:user$_guide_assign_result.The parameters that are accessed are : voltage magnitude at bus 4, the active powe$

```
>>> result1 = tf.loadflow(case1)
```

Figure 3.11: How to assign the result-object to an instance

```
>>> bus4_vm = result1.get('bus', 4, 'vm')
>>> bus169_pd = result1.get('bus', [1,6,9], 'pd')
>>> gen_pg = result1.get('gen', 'all', 'pg')
```

Figure 3.12: Parameters can be accessed with the Case.get-function

The last way to access the result is by saving the solved data to an Excel-file. Solved cases that are assigned to an instance, such as "result1", can save the data with the Case-class-function "save2xl()". Figure 3.13 shows how to save the instance from Figure 3.11 to an Excel-file.

```
>>> result1.save2xl('result1')
```

Figure 3.13: How to save the data of a result-instance to an Excel-file

Case.save2xl() has only one required argument, "filename", which specifies the name of the Excel-file. When calling the function, the file is by default saved in the current working

directory, and will automatically "pop-up" on the screen. Section 5.2.3 will discuss available settings for saving the data.

Figure 3.13 shows another way to save the case; by specifying the filename when running a load-flow. Case.save2xl is then automatically invoked, and the resulting Excel-file will pop-up in the end of the simulation.

```
>>> tf.loadflow(case1, filename = 'result1')
```

Figure 3.14

This approach can provide information on all the iterations of the load-flow, in contrast to saving the Case-object returned by the loadflow-function, which only contains the final result.

3.3.5 Settings

Topflow allows the user to choose between a range of options, including choice of loadflow algorithm, various settings for saving the result to an Excel-file, and how much output the load-flow prints to the screen.

The easiest way to set the desired options is to type them in as keyword arguments in the simulation-function. For example, Figure 3.15 show how to run a fast decoupled loadflow (XX-version) on case1 (from Figure 3.8), with no printed output, saved in a file called "Result_FDXX". Topflow will save the file in the current working directory.

```
>>> result1 = tf.loadflow(case1, version = 'FDXX', print_verbose = 0, filename = 'Result_FDXX')
```

Figure 3.15: How to specify options in the loadflow-function

Alternatively, the settings can be set before the simulation by initializing a topflow.Settings() object. Settings() is a class which contains all the available options for running a loadflow, stored as instance-variables. By calling Settings() without any arguments, the object is initialized with the default settings. This class is convenient to use if the user wants to run multiple simulations with specific settings. Figure 3.16 show how to set personal settings by initializing a Settings() object, and use them in multiple loadflows.

```
>>> import topflow as tf
>>>
>>> #Initialize the cases:
>>> case0 = tf.example_case('case3')
>>> case1 = tf.example_case('case14')
>>> case2 = tf.example_case('case118')
>>>
>>> #Set personal settings:
>>> mysettings = tf.Settings(version = 'FDXX', print_verbose = 0)
>>>
>>> #Run the loadflows:
>>> result0 = tf.loadflow(case0, mysettings)
>>> result1 = tf.loadflow(case1, mysettings, print_verbose = 1)

Convergence. Number of iterations: 6.5
>>> result2 = tf.loadflow(case2, mysettings)
>>>
```

Figure 3.16: How to use a Settings-instance to specify options

Notice that if a setting is directly specified when running a loadflow, it will override the equivalent setting of the Settings() object. For that reason, the second loadflow in Figure 3.16 is the only one printing output to the screen.

Chapter 4

Method

There are many decisions to make when updating a program, such as the choice of programming language, the architecture of the code, and the user interface. Previous contributors decided to use Python as the administrative language and C-extensions to boost slow functions. The idea is to draw benefits from the flexibility of Python, and the speed of C. Existing implementations uses a well-established technique to interface Python with C, which section 2.2 explained in detail. The updated toolbox in this thesis uses the same technique, since it works well and makes the contribution consistent with the previous work.

The "topflow.loadflow()"-function introduced in the user-guide have two callable sub-activities; "acsolve" and "decsolve". acsolve is based on the Newton-Rapshon load-flow from subsection 2.5.7, and decsolve is based on the fast-decoupled load-flow from subsection 2.5.8. As stated in Chapter 1, the primary goal of this thesis is to ensure that acsolve is reliable, and matches the performance of similar activities in open-source Python-projects. This chapter will, therefore, emphasize on describing how acsolve was implemented and optimized.

4.1 The file structure

This project is not aiming to distribute an open-source toolbox, however it is structured as a Python-package ready for distribution. The structure corresponds with the Python Packaging Authority's (PyPA) guidelines on how to package a project [32]. The reason for choosing this structure is that it enables Topflow to be easily installed locally with "pip install", if the contributors share the project. It also makes it easier to distribute the toolbox in the future, if that is desired.

```
topflow/
  topflow/
    __init__.py
  tests/
  setup.py
  README.md
```

Figure 4.1: The top level structure of the toolbox

The folder "topflow" is the root of the repository, which is folder available from GitHub using the pull- or copy-command. The sub-directory, also called "topflow", is the package containing all the source files. This sub-directory also contains a "__init__.py" file, which is required

to import the directory as a package. This file also makes some functions available from the top level of the package. For example, instead of typing "topflow.loadflow.loadflow()" to access the function "loadflow()" from the file loadflow.py", the user can simply type "topflow.loadflow()".

The second sub-directory of the root, "tests", contains the tests-scripts of the project that are used to study the reliability and performance of the code. Section 4.3 and section 4.4.1 will introduce these tests.

The "setup.py" file is the scripts that builds the distribution using "setuptools". At a minimum, it specifies the name, version, a short description of the package, and which source files to include. This file can also handle dependencies during installation.

Finally, the README.me is a text file that introduces and explains the package.

4.2 User interface

The specialization project by Åsmund Sælen [1] focused on translating the Newton-Rapshon algorithm, acsolve, from the original toolbox to Python and C. The updated code did not include functions for initializing the input data, nor a well-designed user interface. The script that tested acsolve initialized the data manually, and passed the parameters as arguments to the function. The first issue with this method is the way the parameters are initialized. The sub-functions of acsolve requires many parameters, and the total number of arguments would increase even more if the activity should include various options. To initialize all the parameters manually in separate objects is cumbersome, even for small systems. The second issue is related to how Python passes variables: functions with lists as arguments can change these globally. The user of the function can prevent this by passing a copy of the list, but again this is inconvenient to do for many objects.

This thesis solves the issues discussed above by introducing two new classes, namely "Case" and "Settings". A Case-object encapsulates all the information acsolve needs to perform a load-flow, while the Settings-object specifies options such as how much information the load-flow function prints to the terminal or saves to a file. The new classes reduces the number of arguments in acsolve to only two, and introduces new ways to initialize the parameters. The Case-class has functions that can read/write Excel-files on a specific format. This particular format is almost identical to the one used by MATPOWER, because it makes it easy to use the open-source MATPOWER test-cases to verify the implementations of Topflow.

4.3 Reliability

The primary goal of this thesis is to verify the Newton-Rapshon load-flow activity, acsolve. This activity creates the foundation of the toolbox, since other techniques relies on the same sub-functions. Other parts of the toolbox are easier to develop and verify, once the Newton-Rapshon load-flow is established as a reliable routine.

4.3.1 Automated tests

This thesis uses automated tests to verify that the implemented functions are working as expected. The automated tests are functions which execute code, and check if specific assertions are true. These assertions are based on the fact that a known fixed input creates a know fixed output.

There are two types of automated tests: unit tests and integration tests. Unit tests check a single component, or in this case: a function. Integration tests, on the other hand, checks the interaction between multiple functions.[33] This project uses "pytest" as a test-runner. Pytest is a framework that has features such as detailed information about failed assert-statements and auto-discovery of test-scripts. This makes running automated tests efficient and easy.[34]

Unit tests

Appendix C.6 and C.7 contains the unit tests that analyses the sub-functions in Topflow. The outcome of most functions are easy to validate by checking the result of a small example. The examples used to check the functions are imaginary, and customized to check if the function handles all relevant inputs as expected.

For example, Figure 4.2 shows the unit test for the function "maxmism", which finds the positions of the worst power mismatches. The first part of the test sets up the input data, which attempts to check all aspects of the function. The numpy-arrays "pinj" and "qinj" contain the active and reactive power mismatches . "mismloc[0]" and "mismloc[1]" stores the position of the worst mismatch in pinj and qinj respectively. "nbuses" is the number of buses in the system, and "buscod" are the bus-codes. Buscod shows that the first element in pinj and qinj is a PV-bus (2), the second is the slack-bus (3), the third is a PQ-bus (1), and the last is a VAr-limited PV-bus (-2).

```
import numpy as np

def test_maxmism():
    nbuses = 4
    buscod = np.array([2,3,1,-2])
    mismloc = np.array([0,0])
    pinj = np.array([0.0,150.0,-100.0,10.0])
    qinj = np.array([150.0, 0.0, 10.0, -100.0])

    for pqv in [1,2,3]:
        mismloc = np.array([0,0])
        maxmism(pqv, nbuses, buscod, mismloc, pinj, qinj)
        if(pqv == 1):
            assert_array_equal(mismloc, np.array([2,0]))
        elif(pqv == 2):
            assert_array_equal(mismloc, np.array([0,3]))
        else:
            assert_allclose(mismloc, np.array([2,3]))
```

Figure 4.2: The unit test for maxmism

The test in Figure 4.2 places the highest numbers of pinj (150.0) and qinj (also 150.0) at the slack-bus and a generator-bus, respectively. Those placements is to verify that maxmismat ignores these numbers correctly, because of the nature of the bus-types. Negative numbers are used as the worst mismatch in both qinj and pinj to check if maxmism can deal with worst absolute mismatches. The next part of the test calls the function maxmism. The test calls the function inside a for-loop to check all the different variants of the argument "pqv". This approach makes sure that the function also operates as expected for the fast-decoupled load-flow algorithm, "decsolve". pqv == 1: only the worst active mismatch is found, pqv == 2: only the worst reactive mismatches is found, pqv == 3: both active and reactive mismatches are found. The final stage is to check if the result of the function call is as expected. This is done

by the numpy-function "numpy.testing.assert_array_equal", which compares the known result and the worst mismatch-positions calculated by maxmism. The assertions are that the worst active power-mismatch is -100 at position 2 in pinj, and that the worst reactive power-mismatch is -100 at position 3 in qinj.

The known result of running maxmism on the system in Figure 4.2 can be seen beforehand by evaluating the input data. The same thing cannot be said about the results of more complex functions. Unit tests of such functions bases its assertions on the known result of performing a load-flow on the three-bus system in Figure 4.3. The specialization project [1] studied that system, and verified the result by hand-calculations and the lecture notes form the course ELK-14 at NTNU [4]. For example, the unit test for the function which builds the Jacobian matrix (2.20) bases its assertions on the matrix calculated by [1] and [4].

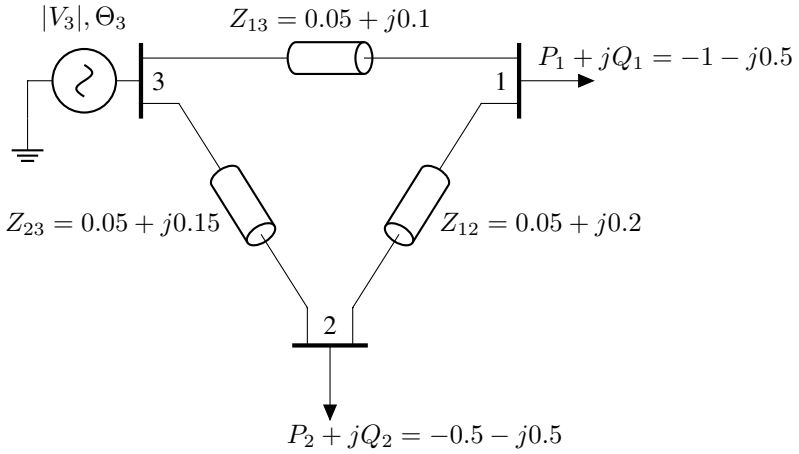


Figure 4.3: Three bus system [1]

The system in Figure 4.3 is set up as a "fixture" in Python. Fixtures are functions that are recognized by pytest and initializes input-data for test functions. A fixture-object can be passed to a test-function as an input argument to provide the required data. Therefore, the data of the system in Figure 4.3 is only initialized once, and passed to multiple test-functions. The consistency of the input data makes the unit tests produce reliable results. [35]

Integration tests

Integration tests check if the sub-functions of an algorithm interact as expected. They are essential to ensure a robust program that produces consistent, reliable results during the code's development. To run integration tests during development is an efficient way to see if the program still works as expected.

Appendix C.5 contains the integration test for acsolve. The test takes in fixture-objects, which contains input-data of two power systems; the first is the three-bus system in Figure 4.3, and the second is a 14-bus system that origins from the MATPOWER-repository on GitHub [36]. The integration test bases the first system's assertions on the solution that the specialization project [1] provides. Section 6.2 will show that two different programs, namely pandapower and pypower, gives the same solution when running a Newton-Rapshon load-flow on the 14-bus system. Topflow uses that solution to develop assertions for the integration test with the 14-bus system as input data.

4.3.2 Comparison tests

The tests in Appendix C.2 and C.3 evaluates the reliability of Topflow further by comparing it with open-source programs in Python, namely pandapower [37] and pypower [38]. The programs are tested on six standard cases from the MATPOWER repository on GitHub [22]. The cases origins from two different sources: "case14", "case30" and "case118" are IEEE test-cases from [39], while "case1354pegase", "case2869pegase" and "case9241pegase" stems from the Pan European Grid Advanced Simulation and State Estimation (PEGASE) [40], [41]. The digits in the case-names specify the number of buses in the system, which means that "case14" and "case9241pegase" are the smallest and largest systems respectively.

The test-script in Appendix C.2 tests the reliability of acsolve by calling the load-flow functions ("loadflow" for Topflow, "runpp" for pandapower and "runpf" for pypower) for each program, and compare the calculated voltages. The functions use flat-start, and a convergence criteria of 10^{-8}.

4.4 The design of the Python-C interface

Table 4.1 show the design of Topflow which was purposed in the specialization project [1]. The project argued to write functions that handle heavy calculations in C and keep the less time-consuming functions in Python. Which functions that were considered "less time-consuming" was not well justified, since Topflow was not able to study large systems.

Function	**Environment**
acsolve	Python
Sub-function	**Environment**
giibii	C
netinj	C
mismat	C
zerosp	Python
fmaxsp	Python
enfqlim	C
bujac	C
addel	C
numpy.linalg.solve	Python
t_u	C
jacpy	Python

Table 4.1: The design purposed in [1]

The new Case-class (see section 4.2) makes it possible to initialize data and run load-flows on large power-systems. One of the goals of this thesis is to develop a toolbox that matches the performance of existing open-source programs for Python. This goal introduces a new motivation when deciding the Python-C interface, namely, to optimize the functions. The following sections describe the method used to decide the new design of the toolbox.

4.4.1 Optimiziation

Optimization is the process of modifying the code to increase the quality and efficiency of the program. Section 4.3 has already conducted the initial step of optimizing the code by writing automated tests. These tests detect bugs in the current program, but they also ensure that updates do not break the functionality of the code.

This thesis uses tools that profiles code in the process of making Topflow more efficient. Profiling gives information on the time consumed by various parts of the code, and can be used to decide which tasks to optimize. There are many profiling tools, and they give different information on the speed of the program. This project uses a combination of three tools to profile the program, namely "timeit", "cProfile" and "LineProfiler". These tools are all available in Python and can easily be installed as packages with the "pip install"-command.

4.4.2 cProfile

The first profiling tool is called cProfile. This module provides statistics of a function, such as how long the execution spends on various parts of the code, and how many times different sub-functions are called. The code snippet in Figure 4.4 is from the user manual of "The Python Profilers" [42], and shows how to profile a function which takes in a single argument.

```
import cProfile
import re

cProfile.run('re.compile("foo|bar")')
```

Figure 4.4: How to profile the function "re.compile()" with cProfile.run() [42]

The example in Figure 4.4 will run the function "re.compile()", and print the result to the screen. Figure 4.4 displays the output.

```
         244 function calls (237 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 enum.py:278(__call__)
        2    0.000    0.000    0.000    0.000 enum.py:557(__new__)
        9    0.000    0.000    0.000    0.000 enum.py:654(name)
```

Figure 4.5: Output from running the code in Figure 4.4

Observing Figure 4.5, the sentence "order by: standard name" specifies that the filenames are ordered alphabetically by name, not by the order in which the functions are called. Table 4.2 gives a short description of the columns in the output from Figure 4.5.

cProfile provides a quick overview of the program's execution time by showing how much time each function spends. This is an excellent first step in locating "bottlenecks" in the program, which makes it easier to optimize the program. The module is designed to provide statistics on a given program, not for comparison with other programs (benchmarking).[42]

Columns	Description
ncalls	number of calls
tottime	Total time spent in the given function (**exluding** time spent in sub-functions)
percall	tottime devided by ncalls
cumtime	cumulative time spent in the given function (**including** time spent in sub-functions)
percall	cumtime devided by ncalls
filename:lineno(function)	provides the respective data of each function

Table 4.2: Description of the columns in the output of cProfile.run()

4.4.3 LineProfiler

The second profiling tool is "LineProfiler". Time-consuming parts of the code are sometimes not caused by sub-functions, but by actual lines in the code. This is especially true for programs that do scientific computing and uses libraries such as numpy. cProfiler is not able to identify these hotspots in the code, since the module only measures the execution time of functions calls. Line-profiler, on the other hand, measures the execution time of each line inside the given function, and gives therefore additional information about the code. A typical workflow should limit the use of LineProfiler to specific functions, since measuring and printing the execution time of each line in the code would be overwhelming. The approach of this thesis is to us cProfile as the first step in profiling the code, and LineProfiler as the second. [43]

Figure 4.6 shows the file "primes,py", which contains the function "primes()" that calculates the "n" first prime numbers. The function is decorated with @profile, which tells LinePorifler to profile it. The function is called at the end of the script with n = 1000 to initialize an instance.

```
@profile
def primes(n):
    A = [True] * (n+1)
    A[0] = False
    A[1] = False
    for i in range(2, int(n**0.5)):
        if A[i]:
            for j in range(i**2, n+1, i):
                A[j] = False

    return [x for x in range(2, n) if A[x]]

primes(1000)
```

Figure 4.6: The file primes.py, which has decorated the function primes()

The easiest way to run LineProfiler is to use the "kernprof"-script. This script comes with the installation of LineProfiler, and is convenient for running the package. Figure 4.7 shows the command that runs the kernprof-script, and profiles the decorated function in primes.py (see Figure 4.6).

The command from Figure 4.7 has to be run in the same directory as the relevant file (primes.py in this case), and the "-v"-option makes LineProfiler immediately print the result

```
kernprof -l -v primes.py
```

Figure 4.7: The command that invokes the kernprof-script

to the terminal. The output of the command is given in Figure 4.8. The results are formatted in a table of 5 columns, which are described in Table 4.3.

Columns	Description
Line	Line number in the function
Hits	The number of times the line was executed
Time	Total execution-time of the line (in the timer's unit)
Per Hit	Execution-time of the line per call (in the timer's unit)
%Time	Percentage of time spent on the line relevant to the time spent on the whole function
Line Content	The actual code written on the line

Table 4.3: Description of the columns in the output from running LineProfiler

```
Wrote profile results to line_profiler_ex.py.lprof
Timer unit: 1e-07 s

Total time: 0.0109534 s
File: line_profiler_ex.py
Function: primes at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def primes(n=1000):
     5         1      60505.0  60505.0     55.2      A = [True] * (n+1)
     6         1         43.0     43.0      0.0      A[0] = False
     7         1         19.0     19.0      0.0      A[1] = False
     8        30        569.0     19.0      0.5      for i in range(2, int(n**0.5)):
     9        29        462.0     15.9      0.4          if A[i]:
    10      1419      20163.0     14.2     18.4              for j in range(i**2, n+1, i):
    11      1409      22311.0     15.8     20.4                  A[j] = False
    12
    13         1       5462.0   5462.0      5.0      return [x for x in range(2, n) if A[x]]
```

Figure 4.8: The output of profiling the prime-function

The first lines of the output gives information about the execution. This includes the "Timer unit", the total execution time of the code, as well as the file and the functions which were profiled. Timer unit specifies the conversion factor to seconds for the information provided in the table containing the results.

Observing the output in Figure 4.8, it can be seen that the most time-consuming lines are lines 7 and 8. These lines should be the focus when optimizing the code. One way of improving the performance of this particular code is to use the NumPy-library, which provides faster array-operations compared with the inbuilt Python-lists.

4.4.4 Timeit

The last module for analyzing the performance of the program is "timeit". timeit contains time-related functions, making it simple to measure the execution time of a specified part of the code.

The module avoid common traps for measuring execution times, and is more reliable than the inbuilt Python timer "time", and the modules cProfile and LineProfile [44]. The purpose of using timeit in this thesis, is to benchmark Topflow against pandapower and pypower.

Figure 4.9 show a simple example of how to use the timeit-module to measure the execution time of a sample code. The execution time of the code in Figure 4.9 is found by using the timeit.timeit() function. This function gives the user the opportunity to specify three input arguments. The keyword argument "stmt" stands for "statement", and is the actual code that the user wants to time. Timit runs the argument "setup" before the actual statement. timeit uses the setup to import the modules that the statement requires. The final argument, "number", refers to the number of times timeit.timeit() runs the code. A high number of measurements gives a more reliable result because it minimizes the influence of background processes. The default value of 1 million measurements can make the tests very slow, so there is a trade-off between the precision and speed of the test itself.

```
import timeit
#Code to executed only once
my_setup = '''
import numpy as np
list = np.array([0,1,2,3,4])
'''
#Code whose execution time is to be measured:
my_code = '''
for i in range(0,5):
    list = np.append(list, i)
'''
print(timeit.timeit(setup = my_setup, stmt = my_code, number = 1000))
```

Figure 4.9: An example of a script which uses the timeit-module

4.4.5 Profiling acsolve

The focus in the process of optimizing Topflow is on increasing the performance of the function acsolve. This function performs a Newton-Rapshon load-flow, and its sub-routines creates the foundation for other techniques in the toolbox.

All the profiling in this section is done on the MATPOWER test-case case1354pegase with default settings. The first stage in locating the most time-consuming part of acsolve, is to profile the code with cProfile.run(), which reveals the most time consuming sub-functions. The result is summarized in Table 4.4, which orders the relevant functions from the most time consumed to the least. Functions with a cumulative time of less than 0.005s are not included.

Observing Table 4.4, the most time consuming functions are jacpy, which creates the full the Jacobian matrix (2.20), and "numpy.linalg.solve", which solves the differential equation system (2.19). This is no surprise, since the solver numpy.linalg.solve requires a dense matrix as input. A promising alternative to the numpy-solver is another solver available in Python via the Scipy-library, namely "scipy.sparse.linalg.spsolve()". This function solves a sparse linear system, and provides high performance by using the C library "UMFPACK". The input matrix, "A", should be a sparse matrix on CSC or CSR form to ensure efficiency. Because building the full Jacobian matrix is expensive, a new function called "COO_conv()" is implemented as a replacement for jacpy. This function returns a sparse matrix on COO-format. This format is used because NumPy provides fast conversion from COO format to CSR and CSC through the

Function	Cumulative time in seconds
acsolve	1.732
Sub-function	**Cumulative time in seconds**
numpy.linalg.solve	0.983
jacpy	0.648
zerosp	0.021
maxmism	0.019

Table 4.4: The most time consumin functions of acsolve

functions "to.csr()" and "to.csc". Having the Jacobian matrix on COO-format makes it more flexible, since it can efficiently switch the format required by a specific solver.[45],[46]

The other notably slow functions from Table 4.4 are "zerosp" and "maxmism". It was argued in the specialization project, [1], to write these functions in pure Python, because they are simple, and not expected to do complicated computations. The profiling with cProfile shows, on the other hand, that these functions are slower than the more complex routines that are C-extensions. One way of further increasing the code's performance is, therefore, to write all functions as C-extensions.

The impact of the modifications discussed above is analyzed by profiling the new code with cProfile on the same system and settings as the initial profiling-test. The results are summarized in Table 4.5.

Function	Cumulative time in seconds (s)
acsolve	0.069
Sub-function	**Cumulative time in seconds (s)**
scipy.sparse.linalg.spsolve	0.024
to.csr()	0.003
coo_conv	0.001
zerosp	0.000
maxmism	0.000

Table 4.5: The optimized functions

Comparing Table 4.4 and Table 4.5, the modifications have lowered the total execution time of acsolve with 1.663 seconds. This improvement means that the optimized code is 25 times faster than the original design. The increase in performance is mostly due to the sparse linear solver, whose performance is approx. 4000% higher than the NumPy-solver. The functions coo_conv() and csr_matrix() restructures the Jacobian matrix 162 times faster than jacpy, and the cumulative times of zerosp and maxmism are also improved significantly.

Table 4.5 shows that scipy.sparse.linalg.spsolve is the most time consuming function with a cumulative time of 0.024s. The cumulative times of all the other functions are each lower than 0.005s. In fact, the sum of all the sub-functions does not add up to the total execution time of 0.069s. Figure 4.10 shows a snapshot of the output from cProfile, which reveals that acsolve has a "tottime" of 0.030s. This means that acsolve is spending 0.030s outside its sub-functions and 0.39s inside.

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    61    0.000    0.000    0.000    0.000 <__array_function__ internals>:2(can_cast)
    15    0.000    0.000    0.000    0.000 <__array_function__ internals>:2(copyto)
    10    0.000    0.000    0.000    0.000 <__array_function__ internals>:2(empty_like)
    10    0.000    0.000    0.000    0.000 <__array_function__ internals>:2(ndim)
    10    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:389(parent)
     1    0.000    0.000    0.069    0.069 <string>:1(<module>)
   332    0.001    0.000    0.001    0.000 __init__.py:506(cast)
   115    0.000    0.000    0.000    0.000 _asarray.py:16(asarray)
   332    0.000    0.000    0.000    0.000 _internal.py:251(__init__)
   332    0.000    0.000    0.001    0.000 _internal.py:268(data_as)
   332    0.000    0.000    0.001    0.000 _internal.py:346(_as_parameter_)
    10    0.000    0.000    0.000    0.000 _methods.py:28(_amax)
    10    0.000    0.000    0.000    0.000 _methods.py:32(_amin)
    40    0.000    0.000    0.000    0.000 _util.py:129(_prune_array)
     1    0.030    0.030    0.069    0.069 acsolve.py:17(acsolve)
     6    0.000    0.000    0.001    0.000 acsolve_wrapper.py:127(mismat)
```

Figure 4.10: a snippet of the output from cProfile

Nearly all the heavy calculations are performed by the sub-routines of acsolve, which makes it unrealistic that almost half of the cumulative time is spent outside these functions. Information about the source of this time-consumption is not accessible from cProfile, since it is not caused by function-calls. LineProfiler, on the other hand, gives information about each line in the function. The code is profiled with LineProfiler by adding the declaration "@profile" before the definition of acsolve and running the command "kernprof -l -v acsolve.py" in the terminal.

The output from running LineProfiler showed that acsolve spends much time on imposing flat start, setting up the right-hand side of the equation system (2.19), and updating the voltages. The design from the specialization project [1] handles these operations directly in acsolve as pure Python-code. Figure 4.11 is a snippet of the output from LineProfiler, and shows the part of acsolve which updates the voltages. Column number 5 from the left gives the percentage of the total time spent on a particular line. It can be seen that the activity of updating the voltages in total occupies 35.7% of the cumulative time spent in acsolve. Together, the three operations discussed above account for the time-consumption outside the sub-functions of acsolve. Each of the operations is written as C-extensions to improve the performance of the code further.

```
250                                                 # update voltages and angles:
251
252         5       116.0     23.2      0.0         ip = 0
253         5        65.0     13.0      0.0         iq = 0
254      6775     74909.0     11.1      3.8         for ib in range(0, obj.nbuses):
255      6770    115395.0     17.0      5.9             if(obj.buscod[ib]!=3):
256      6765    127939.0     18.9      6.5                 obj.voang[ib] += correction[ip]
257      6765     82635.0     12.2      4.2                 ip += 1
258      6765    115785.0     17.1      5.9                 if(obj.buscod[ib] != 2):
259      5470    114835.0     21.0      5.9                     obj.vomag[ib] += correction[obj.nbuses-1+iq]
260      5470     69266.0     12.7      3.5                     iq += 1
```

Figure 4.11: A snippet of the output from LineProfiler

The impacts of the newest modifications are analyzed by profiling acsolve with cProfile. Figure 4.12 shows a snippet of the output. A comparison of Figure 4.10 and Figure 4.12 show that the time acsolve spends outside its sub-functions has decreased from 0.030 seconds to 0.001 seconds. The total execution time of acsolve is almost cut in half thanks to the most recent modifications.

The optimizations discussed in this section have greatly improved the performance of the program. The previous design of acsolve presented in [1], measured an execution time of 1.732 seconds on the system case1354pegase. The new design executes the same operations in 0.038

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    61    0.000    0.000    0.000    0.000 <__array_function__ internals>:2(can_cast)
    15    0.000    0.000    0.000    0.000 <__array_function__ internals>:2(copyto)
    10    0.000    0.000    0.000    0.000 <__array_function__ internals>:2(empty_like)
    10    0.000    0.000    0.000    0.000 <__array_function__ internals>:2(ndim)
    10    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:389(parent)
     1    0.000    0.000    0.039    0.039 <string>:1(<module>)
   380    0.001    0.000    0.001    0.000 __init__.py:506(cast)
   115    0.000    0.000    0.000    0.000 _asarray.py:16(asarray)
   380    0.000    0.000    0.000    0.000 _internal.py:251(__init__)
   380    0.000    0.000    0.001    0.000 _internal.py:268(data_as)
   380    0.000    0.000    0.001    0.000 _internal.py:346(_as_parameter_)
    10    0.000    0.000    0.000    0.000 _methods.py:28(_amax)
    10    0.000    0.000    0.000    0.000 _methods.py:32(_amin)
    40    0.000    0.000    0.000    0.000  util.py:129(_prune_array)
     1    0.001    0.001    0.038    0.038 acsolve.py:17(acsolve)
     6    0.000    0.000    0.001    0.000 acsolve_wrapper.py:127(mismat)
```

Figure 4.12: A snippet of the output from cprofile

seconds, which is an improvement in the computational performance of approximately 4500%. In other words, the new design is 45 times faster than the original.

4.5 Approach to work

The process of updating Topflow has been time-consuming and demanding, yet exciting and educational. In particular, the field of computer programming required a steep learning curve. With previous experiences limited to introduction-courses in MATLAB and C++, the project introduced many new concepts since it utilizes FORTRAN, C, and Python. The fundamental theoretical knowledge and programming skills were acquired during the specialization project "Toolbox for Specialized Power System Analysis" by Åsmund Sælen [1], which leads to this thesis. This section is for that reason divided into two part: subsection 4.5.1 are paragraphs from the specialization-project, while subsection 4.5.2 describes to working-process of this thesis.

4.5.1 Theoretical research and skill development

Theoretical knowledge of power system analysis and the skill of programming with different programming languages are crucial when working on the toolbox. The specialization course "ELK-14: Methods and Algorithms for Power Systems" at NTNU provided useful insight into the algorithms used in the toolbox. This thesis uses some of the lecture slides as references, because they gave a compact, thorough description of the required methods. Besides, the mandatory exercises of the course encouraged the use of Python to create the algorithms. "Learning by doing" is arguably the best way to develop skills in a new programming language, which is why "Elk-14" was also very helpful in learning Python.

Free tutorials from online platforms, in particular YouTube and Tutorialspoint, covers all the relevant programming languages. These sites can be recommended to future contributors as a place to get familiar with coding. Other sites for users to ask and answer questions, such as Stack Overflow, are great for fixing bugs in the code, and has been used extensively through this project.

The original Fortran code was provided by professor Olav Bjarte Fosso, who also provided additional documentation of the code. One resource which was especially helpful was a set of videos where he explained parts of the program in detail. These video-clips were made

available on Google Drive. The work done by Statnett's representative, Leif Warland, as well as the code from the previous master thesis by Hege Kvandal [2], was made available in the private repository "topflow" on Github. This repository consisted of a considerable amount of already translated code, both rewritten C-code and wrapper-functions in Python. Both collaborators shared the same goal of modernizing the toolbox, making Python the administrative language which displays the program to the user. These contributors laid the foundation of the work on the toolbox, and established the technique of using Ctypes and Scons to create C-extensions. The scope and the direction of the work on this thesis came naturally with the established decisions and techniques. [1]

4.5.2 Master thesis

The specialization course [1] presented a first update of the Newton-Rapshon load-flow function, acsolve. The goal of the project was to make the code run on a 3-bus system. That system did not consist of components such as transformers and shunt-capacitors. Although the test passed, the simplicity of the system called for additional examination of the implementations.

Designing the user interface, which is essential when initializing the data for large power systems, makes up a considerable proportion of the time spent on this thesis. The structure of the input-data was kept close to the data-formats used by MATPOWER [47], because this software is the primary source of the test-cases used to verify the functions in Topflow. The choice of using Excel as the source of the input data is the result of previously experienced that Excel is a flexible and commonly used software. More research could have been conducted in finding alternative options, which may be more efficient and flexible. The same is true for the current method of reading Excel-files, which uses the Python-package "openpyxl". An alternative package is "pandas", which is part of the SciPy-stack (see section subsection 2.1.3) and potentially provides faster functions for communicating with Excel. The intent of this function was, on the other hand, to develop a functional routine of initializing the parameters of large power-systems, something the current method does.

Another time-consuming part of the work was achieving satisfactory results when running load-flows on large systems. Script for running the comparison-test described in subsection 4.3.2 were utilized after the implementation of the functions that initializes the system-parameters. The tests showed that acsolve did not converge for all the test-cases. The approach of detecting the bugs was to print the results of each sub-function, and manually check the values. This method was inefficient, unreliable, and unsuccessful in locating the errors. The errors were resolved thanks to the discovery of automated tests. These tests made validating code more efficient and reliable, since it's a computer that checks the values, not a human. Section 6.2 will discuss the success of the automated tests further.

The toolbox presented in this thesis uses the same technique for writing C-extensions as the previous contributors, and many of the functions are modifications of code originally translated by Leif Warland at Statnett. However, the lack of documentation made it difficult to understand the overall system, which is why the code of this thesis is not merged back with the original branch on GitHub.

Chapter 5

Implementation

This section aims to give future contributors a deeper understanding of the existing implementations of the toolbox.

5.1 Installation test

The installation process and the setup.py file are tested in a virtual environment in a different directory than the Topflow-project. Virtual environments have their own site-directories and Python-packages, isolated from the system site directories [48]. This makes them suitable to test if "pip" installes Topflow correctly. Figure 5.1 shows the command that is typed to create the virtual environment called "topflow-env".

```
C:\Users\aasel\programmering\environments>python -m venv topflow_env
```

Figure 5.1: Create the virtual environment topflow_env

To enter the virtual environment, it must be activated by using a script in the virtual environment's binary directory. The script is invoked by running the terminal-command "<venv> Script\activate.bat" in the directory of the virtual environment (<venv> is the name of the virtual environment) [48]. Figure 5.2 shows the command that activates the virtual environment topflow_env (which was created in Figure 5.1).

```
C:\Users\aasel\programmering\environments>topflow_env\Scripts\activate.bat
```

Figure 5.2: Active/enter the virtual environment topflow_env

The name of the virtual environment will now appear in brackets in the far left end of the terminal. This means that the user is inside the virtual environment. The installed python-packages are seen by running "pip-list", as shown in Figure 5.3.

```
(topflow_env) C:\Users\aasel\programmering\environments>pip list
Package    Version
---------- -------
pip        19.2.3
setuptools 41.2.0
```

Figure 5.3: The installed packages in the newly created virtual environment "topflow_env"

There are currently only two packages installed in the virtual environment: pip and setuptools. Topflow can now be installed to check if the setup.py installs the package and all its dependencies correctly. This example uses "pip install -e" to install the package in developing-mode (the package appears to be installed, but is still editable) [32]. Since Topflow is not an open source project, pip requires the full path of the local Topflow-project. Figure 5.4 shows the command that installs Topflow.

```
(topflow_env) C:\Users\aasel\programmering\environments>pip install -e c:\users\aasel\programmering\topflow
```

Figure 5.4: How to install Topflow locally

To validate the installation, "pip list" is once again used to see the installed packages in the virtual environment. Figure 5.5 provides the output of the command, which shows that Topflow and all its dependencies are installed correctly.

```
(topflow_env) C:\Users\aasel\programmering\environments>pip list
Package          Version Location
---------------- ------- ------------------------------------
et-xmlfile       1.0.1
jdcal            1.4.1
numpy            1.18.4
openpyxl         3.0.3
pip              19.2.3
scipy            1.4.1
scons            3.1.2
setuptools       41.2.0
topflow-aasmunsa 0.0.1   c:\users\aasel\programmering\topflow
```

Figure 5.5: The result of installing Topflow

5.2 The Case-class

At the core of the toolbox, is the class "Case". A Case-object encapsulates all the information needed in load flow studies, and contain various class functions for initializing the data and performing other operations. An initialized Case-object is typically passed to the functions which performs load flows on the system, also referred to as "simulation-functions". Most of the variables of the Case-class are NumPy-arrays, which provides efficient storage and better ways of handling data compared with the inbuilt Python-lists. Another benefit of using NumPy-arrays is the mechanism to specify the data type of the content, which is convenient when working with C-extensions. Appendix B.5 provides the source code of the Case-class as well as Table 2, which lists the class-variables. This section will discuss the Case-class functions.

5.2.1 Printing output to the screen

The first way to access the result of a Case-object is by using the print-functions. These are class-functions which provide a quick display of the data directly to the terminal. They are great for checking if the case has been initialized properly, and can be used to print iteration-summaries to the screen when performing simulations. The latter is done by specifying the "print_verbose" argument when running a loadflow, which range from 0 (no output at all) to 3 (very verbose output). The functions are primarily meant for small cases, since the output will

become overwhelming for large systems. Table Table 5.1 show the different functions with a short description of their operations.

Function	Description
topflow.Case.print_all()	Prints all the data of the Case-instance to the screen
topflow.Case.print_buses()	Prints the bus-data of the Case-instance to the screen
topflow.Case.print_gens()	Prints the generator-data of the Case-instance to the screen
topflow.Case.print_lines()	Prints the line data of the Case-instance to the screen

Table 5.1: A description of the print-functions

5.2.2 Accessing the parameters with "get"

Another way to access the parameters of the Case-class is function "get(component, number, variable)". This function returns the desired parameter based on the input from the user. The three required arguments must be of the type string, and must be known to function.

The different variants of the arguments are listed in table Table 5.2. The argument "number" can either be a single component-number, or a list of numbers, as described in Section 3.3.4. Notice that the generator- and line-numbers are the order the user has placed the components in the Excel-file. All the parameters which can be accessed by the component "gen" can also be accessed by "bus", the only difference is that the user can access them using generator-numbers instead of bus-numbers. The get-function is convenient if the user only wants to access parts of the result, and it enables the parameters to be used in further analysis.

Component	Number	Variable	Description
bus	Any external bus-number(s)	vm	Voltage magnitude of the bus(es)
		va	Voltage angle of the bus(es)
		pd	Active power demand (load) of the bus(es)
		qd	Reactive power demand of the bus(es)
		pg	Active power generation of the bus(es)
		qg	Reactvie power generation of the bus(es)
gen	Any generator -number(s)	vm	Voltage magnitude
		va	Voltage angle
		pg	Active power generation
		qg	Reactvie power generation
line	Any line -number(s)	pf	Active power injected at the "from"-end
		qf	Reactive power injected at the "from"-end
		pt	Active power injected at the "to"-end
		qt	Reactive power injected at the "to"-end

Table 5.2: The parameters in the Generator-data record

The variables of a Case-instance are also accessible with the default Python-approach: "instance.variable". The purpose of the get-function is merely to give the user an intuitive interface.

5.2.3 Loading and saving data

This thesis uses Excel as the source of the input and output data. This is mainly because Excel is a widely used and flexible software. The function for loading data from a Excel-file is "topflow.Case.loadxl(filepath)", which requires that the data is structured on the form described in Section 3.3.1. The only argument, "filepath", is the path of the Excel-file which contains the data. If the file is placed in the current working-directory, the name of the file can also be used. Section 3.3.2 showed how to initialize a Case-object by calling the class with the file-path as an input argument. Case has no required input arguments, but when it's called in that way, the loadxl-function will be called automatically to load the data.

Similarly, "topflow.Case.save2xl(filename, save_path = None, save_verbose = 1)" is the function for writing data from a Case-object to a Excel-file. The only required argument is "filename", which is what the user wants to name the file. The second argument, "save_path" tells the function where to save the file. If this argument is not specified, the file will be saved in the current working directory. Lastly, "save_verbose" tells the function how much information it shall save. If save_verbose = 1, it only saves the final iteration, while if save_verbose = 2, it creates two separate spreadsheets: one for the final results, and one for all the iteration-summaries.

To this date, using loadxl and save2xl is the most efficient way of working on large systems with the current toolbox. However it's encouraged to implement new functions and ways to initialize the Case-objects in the future, to enhance flexibility.

5.2.4 External and internal bus-numbers

The program distinguishes between internal and external bus-numbers to provide flexibility to the code. The internal bus-numbers are a result of the order the buses were added to the Case-object when initializing the data. This means that the external bus-numbers can be chosen arbitrary, as long as they are unique, and the order of which they are placed doesn't matter. This also applies for the slack-bus. Conversion between internal and external bus data are done with the Case-class functions "int2ext" and "ext2int".

5.3 The Settings-class

As described in the example from Section 3.3.5, the class "Settings" can be used to specify options before running a simulation. A initialized Settings-object can be passed to the simulation-functions, which will customize its operations based on this input. Table 5.3 gives a full list over the available options, and specifies the default settings. If a Settings-object is initialized without any input arguments, these settings will apply. The class only deals with settings which are relevant to the function topflow.loadflow(), since other simulation-functions from the original toolbox are not yet translated. Future contributors should update the Settings-class (source-code given in Appendix B.20) along with new translations.

Name	Variant	Description	Default	Type
version		**Load-flow algorithms**	'NR'	string
	'NR'	Newton-Rapshon load-flow		
	'FDXX'	Fast-Decoupled XX-version		
	'FDBX'	Fast-Decoupled BX-version		
	'FDXB'	Fast-Decoupled XB-version		
	'FDBB'	Fast-Decoupled BB-version		
flat_start	True	Start with flat start	True	bool
	False	Start with specified voltages		
enf_qlim	True	Consider reactive generation limits	False	bool
	False	Ignore reactive generation limits		
max_it		Maximum number of iterations allowed	20	int
conv_tol		Convergence tolerance	10^{-6}	float
pqv		**FDLF only**	FDXX: 1 FDBX: 1	int
	1	Start with a $P\|V\|$-iteration	FDBB: 1	
	2	Start with a $Q\theta$-iteration	FDXB: 2	
filename		Name of the file containing the results	None	string
save_path		Path of the file containing the results	cwd*	string
save_verbose	1	Save the final results	1	int
	2	Save final result + each iteration-summary		
print_verbose	0	Don't print anything	1	int
	1	Print the final result		
	2	Print the final result more verbose		
	3	Print each iteration-summary		

Table 5.3: Varaibles of the Settings-class.
*cwd = current working directory

5.4 Example cases

"topflow.example_cases()" is a Case-class function that gives access to several test-cases. Topflow has converted these cases from the MATPOWER repository at GitHub [22]. The cases origins either from IEEE test-cases [39], the Pan European Grid Advanced Simulation and State Estimation (PEGASE) [40],[41] or the book "Power System Analysis" by J.J.Grainger and W.D.Stevenson [49]. Table 5.4 lists all the example cases available in Topflow. The digits in the case-names specify the number of buses in the system; hence case 9241pegase, which represents the size and complexity of the European high voltage transmission network, is the largest system.

Example case	Origin
case4gs	J.J.Grainger and W.D.Stevenson
case14	IEEE
case30	IEEE
case118	IEEE
case300	IEEE
case1354pegase	PEGASE
case2869pegase	PEGASE
case9241pegase	PEGASE

Table 5.4: The design purposed in [1]

5.5 The loadflow-function

As the name suggests, "topflow.loadflow(..)" is the function for running load-flows. Based on the input from the user, it calls the correct simulation-function, which performs analyzes on a initialized Case-object. The simulation-functions take in the Case-object as an argument, and makes a deep-copy before performing any other operations. This action ensures that the original Case-object is not modified in any way. The deep-copy is then modified by a iteration-process, and returned from the function. A initialized Case-object can therefore be used as a base-case and passed to multiple simulations without the need of reloading the data from the Excel-file. This approach gives a better performance compared to modifying the original Case-object, since loading a Excel-file can be time consuming and should be done as rarely as possible.

This project presents two different simulation-functions: "acsolve" and "decsolve", which are callable through topflow.loadflow(). They are both based on routines in the original toolbox by prof. Olav Bjarte Fosso.

5.5.1 Acsolve

The first function callable from topflow.loadflow(), acsolve, performs a Newton-Rapshon load-flow (see section 2.5.7). A initial proposal of this method was presented in the specialization project [1]. That proposal was only tested on a small 3bus-system, which could not provide any information on the speed of the program. The specialization project [1] argued to keep the less complex sub-functions in Python because they would not affect the performance greatly. Section 4.4.5 in this thesis showed the benefits of having more code in C, and as a result, all the sub-functions of acsolve are now C-extensions. Figure 5.6 show the structure of Topflow, highlighting acsolve and all its sub-routines. A flowchart of the algorithm is provided by Figure 5.7.

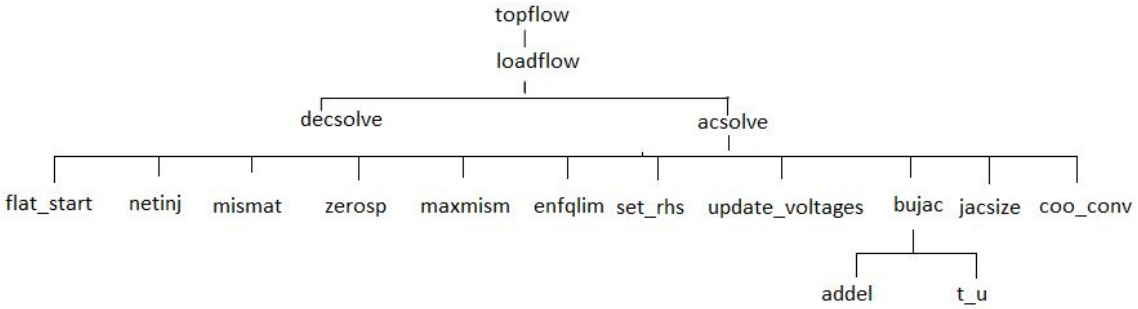


Figure 5.6: The structure of Topflow, highlighting the branch of acsolve

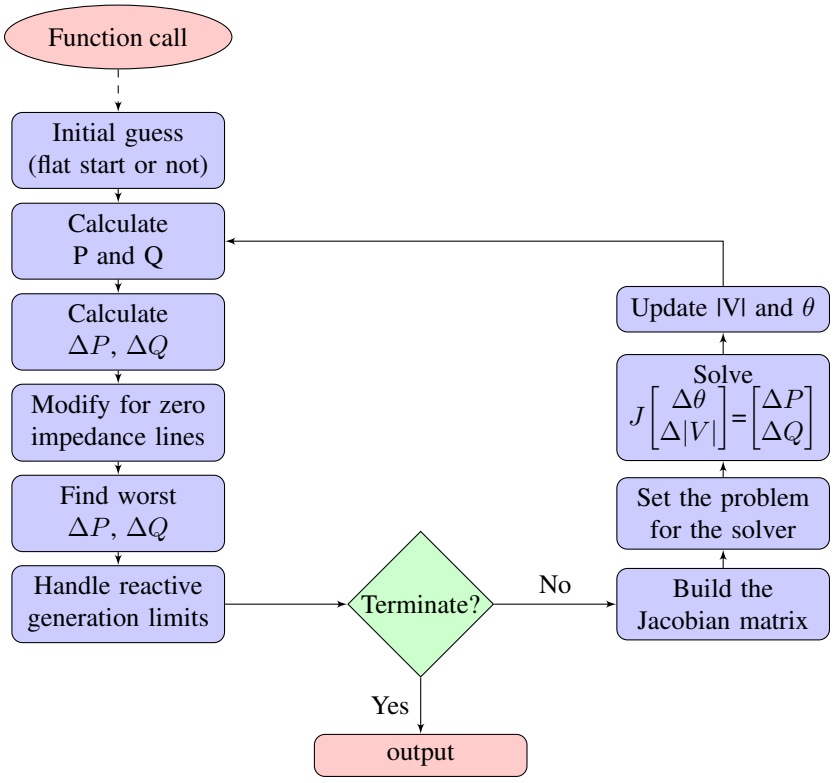


Figure 5.7: Flow-chart of the Newton-Rapshon load-flow algorithm, acsolve

5.5.2 Decsolve

The second simulation-function, decsolve, is based on the fast decoupled load-flow (FDLF) method from section 2.5.8. Figure 5.9 shows the structure of Topflow, highlighting decsolve, and all its subroutines. The original Fortran-code had separate functions for acsolve and decsolve. A comparison of Figure 5.6 and Figure 5.9 shows that the two activities now shares many of the same sub-functions. This is achieved by introducing the argument "pqv". pqv stands for P-Q-version, and tells the functions whether to calculate active parameters (pqv = 1), reactive parameters (pqv = 2) or both (pqv=3). For example: the sub-function "netinj" calculates either only the active power injections, only the reactive power injections, or both, based on the value of pqv.

The flowchart of decsolve is given in Figure 5.10. It shows that after an iteration with pqv = 1, the algorithm switches to pqv=2 in the next iteration, and vica versa. The user chooses the initial value of pqv by specifying the FDLF-version when calling the function. The built-in versions are: FDXX, FDXB, FDBX and FDBB. The two first letters stands for "Fast-Decoupled" and the two last refers to the version of B' and B'' (see subsection 2.5.8). For example, the version FDBX specifies that B' is built with b_{ij}, while B'' is built by substituting $1/x_{ij}$ for b_{ij}. By default all the versions start with a $P\theta$-iteration (pqv = 1), except from FDXB which starts with a $Q|V|$-iteration (pqv = 2). The default versions are based on the theory from the paper [25], however, the user can choose to override the default settings by specifying pqv when calling the "loadflow"-function. Figure 5.8 shows an example where the version "FDXX" is run with pqv = 2 as the initial value. The order of the $P\theta$- and $Q|V|$-iterations are not expected to show any significant difference, but it makes the code flexible, as the user can customize the FDLF-algorithm for study purposes.

```
>>> import topflow as tf
>>> case1 = tf.example_case('case14')
>>> result1 = tf.loadflow(case1, version = 'FDXX', pqv=2)

Convergence. Number of iterations: 6.5
```

Figure 5.8: A customized FDLF-version

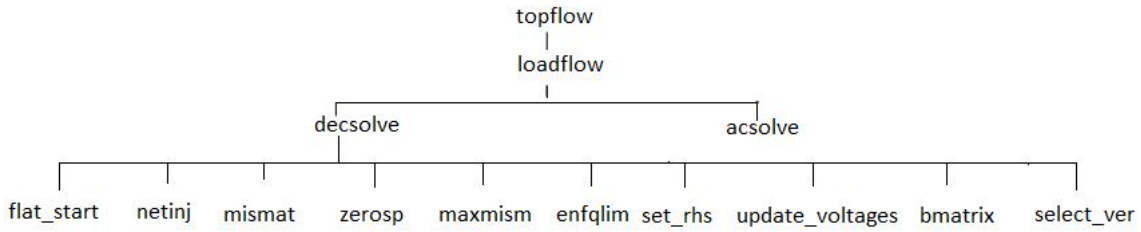


Figure 5.9: The structure of Topflow, highlighting the branch of decsolve

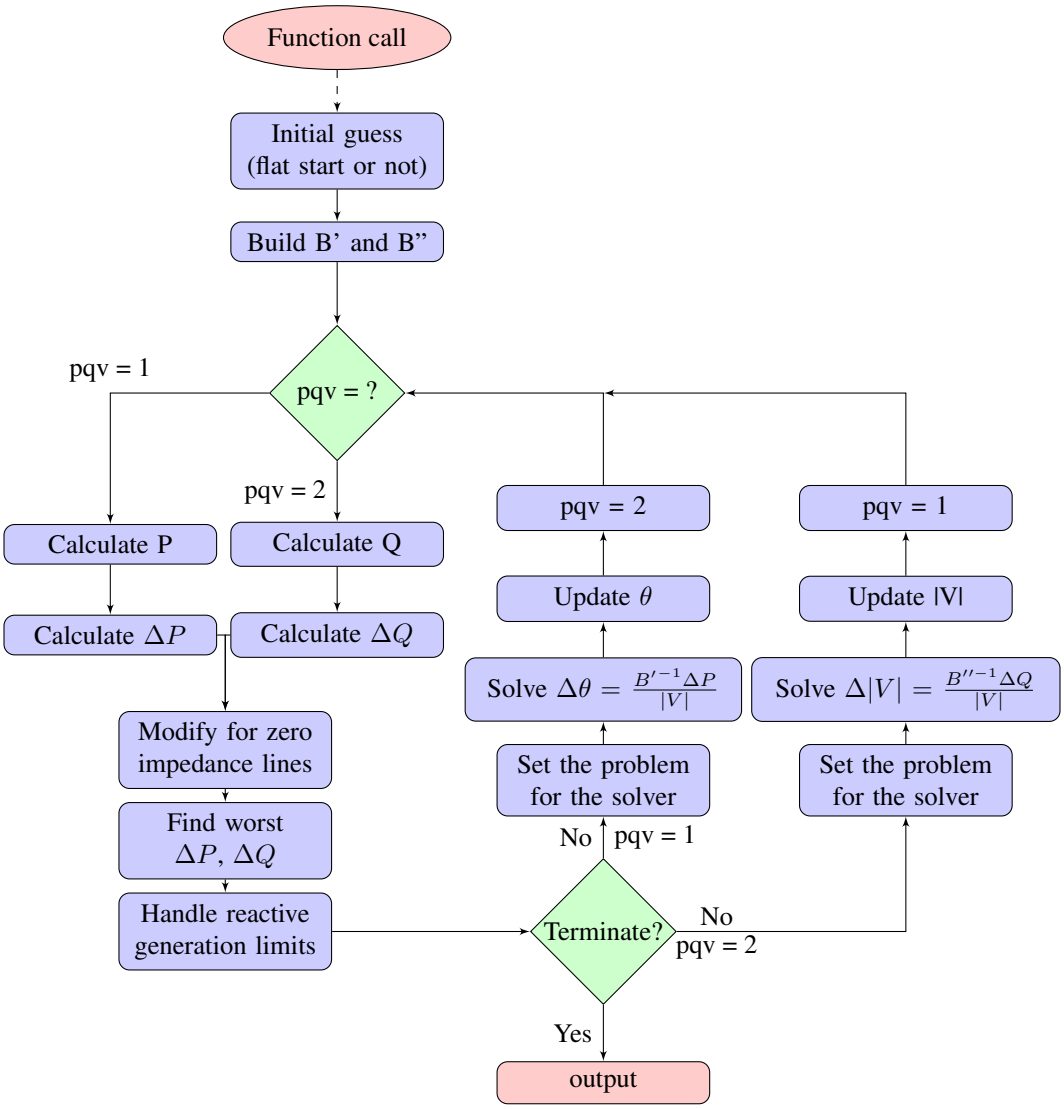


Figure 5.10: Flow-chart of the fast-decoupled load-flow algorithm, decsolve

Handling generators on the reactive power limits

A big difference between acsolve and decsolve is that while the Jacobian matrix is built in each iteration in the Newton-Rapshon load-flow technique, constant matrices are built only once in the fast decouple load-flow technique. This also affect the way the program handles generators put on the reactive generation limit (also called Var-limited PV-buses). These buses are treated as PQ-buses, as described in section 2.5.9. If a generator is put on the reactive limit, the dimensions of the Jacobian sub-matrix $J4$ will change. The only operation required by acsolve to handles this is to change the bus-code of the generator. The Jacobian matrix will then change its dimensions automatically in the next iteration. If the VAR-limitd generator is the slack bus, the next availablle PV-bus is chosen as the new reference.

The same approach could be used in decsolve, however, this would mean that the constant matrices would have to be rebuilt before the next iteration. If generators were often put on and off the reactive limits, the advantage of having "constant matrices" would disappear since they would have to be rebuilt often. To avoid this, the matricies B' and B'' are not built in the dimensions of $J1$ and $J4$, as the theory from section 2.5.8 suggests, but in the dimension $N \times N$

(N being the total number of buses). The rows and columns which must be ignored (slack bus for B', slack bus and PV-buses for B'') are disregarded by adding a large number, M (10^{10} is used in Topflow) on the diagonal element of the relevant buses. This will in practice result in the contribution of these rows and columns being zero. When a PV-bus is put on the reactive limit, the diagonal element in B'' of that bus is subtracted by M, and the relevant row and column are active. To put a Var-limited PV-bus off the limit, the opposite operation is performed by adding Mz on the diagonal. This approach preserves the advantage of the constant matrices, as well as it makes the program more flexible. Some programs require specific ordering of the buses, such as having PQ-buses numbered before PV-buses and the slack bus being the last bus, but this algorithm allows arbitrary numbering of the buses as well as multiple slack-buses.[50]

Chapter 6

Reliability and performance

6.1 Result and discussion of the automated tests

The purposed program from the specialization project, [1] was only tested on a 3-bus system, and showed a number of bugs when it was tested on larger systems. Prior to the automated tests, the approach of detecting bugs was to manually check the calculated values of each function by printing the result to the terminal. This approach is neither efficient nor reliable.

Automated test makes detecting and resolving bugs easier, and the test-runner, pytest, was invoked frequently during the development of the code. Figure 6.1 shows an output from running pytest, which resulted in one failure. Observe that pytest prints the unit-test that failed, highlights the false assertion, and reveals the differences between the assertion and the values calculated by the function. The function that failed is called "maxmism", which is a C-extension and sub-function of acsolve. It calculates the active and the reactive power mismatches (ΔP and ΔQ from (2.19) in subsection 2.5.7). A closer look at the code revealed that the function declared a variable as "int" (integer), instead of "double" (decimal number). The wrong declaration caused the C-function to round off numbers, resulting in inaccurate calculations. This bug was particularly hard to detect, since acsolve converged on all the tests-systems, gave satisfactory results on small system, but slightly inaccurate results on larger systems. The fault could, therefore, been a result of inaccurate or missing information from the large system's input-data. This example shows the utility and efficiency of having automated tests.

Figure 6.2 shows the result of running pytest on the final implementations. Three Python-scripts were identified as tests: the integration test for acsolve (Appendix C.5), the unit tests for acsolve (Appendix C.6), and the unit test for the Case-class (Appendix C.7). The latter contains the test for the function "loadxl", which loads data from Excel-files. The last line of the output shows that all 12 test passed, which means that they are working as expected. The 3 skipped tests are manually marked with a "skip"-decoration because they are functions with "test" in their names that are not automated tests.

The result from Figure 6.2 gives a strong indication that the functions are implemented correctly. Having automated tests increases the reliability of the current implementations, and it's a quick way to check if the functionality of the program is kept when changes are made to the code in the future. However, the test do not guarantee that there are no bugs in the program. Functions may pass automated tests if the assertions are not good enough to check all the situations a function might encounter. Programmers may also make conceptual mistakes, so that the functions work as expected, but the outcome is incorrect. Conceptual mistakes are

```
============================= test session starts =============
platform win32 -- Python 3.8.1, pytest-5.4.2, py-1.8.1, pluggy-0.13.1
rootdir: C:\Users\aasel\programmering\topflow
collected 12 items / 1 skipped / 11 selected

tests\test_acsolve_integration.py .
tests\test_acsolve_unit.py ...F......
tests\test_case.py .

================================== FAILURES ===================
_____ test_maxmism _____

    def test_maxmism():
        nbuses = 4
        buscod = np.array([2,3,1,-2])
        mismloc = np.array([0,0])
        pinj = np.array([0.0,0.150,-0.100,0.010])
        qinj = np.array([0.150, 0.0, 0.010, -0.100])

        for pqv in [1,2,3]:
            mismloc = np.array([0,0])
            ac.maxmism(pqv, nbuses, buscod, mismloc, pinj, qinj)
            if(pqv == 1):
>               assert_array_equal(mismloc, np.array([2,0]))
E               AssertionError:
E               Arrays are not equal
E
E               Mismatched elements: 1 / 2 (50%)
E               Max absolute difference: 1
E               Max relative difference: 0.5
E                x: array([3, 0])
E                y: array([2, 0])

tests\test_acsolve_unit.py:177: AssertionError
=========================== short test summary info ===========
FAILED tests/test_acsolve_unit.py::test_maxmism - AssertionError:
========== 1 failed, 11 passed, 1 skipped in 17.83s ==
```

Figure 6.1: The result of running pytest, which shows an error in the code.

more likely to be picked up by integrated test than unit tests, and it's often challenging to find the source of the error.

```
============================= test session starts =====
platform win32 -- Python 3.8.1, pytest-5.4.2, py-1.8.1, pluggy-0.13.1
rootdir: C:\Users\aasel\programmering\topflow
collected 12 items / 3 skipped / 9 selected

tests\test_acsolve_integration.py .
tests\test_acsolve_unit.py .........
tests\test_case.py .

========== 12 passed, 3 skipped in 3.70s
```

Figure 6.2: The result of the automated tests in the final implementation

The weakest unit tests implemented in the current program are the ones that make use of the data from the three-bus system in Figure 4.3. This is a small system, which lacks complexity and components such as transformers and bus-connected shunt elements. The system is used due to the lack of a better alternative to obtain verified information, such as a Jacobian matrix for a given input. Future contributors should conduct a literature-study to search for better assertions for these unit tests. On the other hand, the integration test for acsolve asserts the final result of a 14-bus system as well as the three-bus system. The 14-bus system includes several transformers, and bus-connected capacitors. The success of the integration test indicates that the implemented transformer-model work as expected.

6.2 Comparison-test of acsolve

Section 4.3.2 explained how the test in Appendix C.2 analyzes the reliability of acsolve further. Table 6.1 summarizes the results of the tests, by displaying how much the voltages calculated by Topflow deviates from pandapower and pypower respectively. All the simulations are performed by running a Newton-Rapshon load-flow with flat-start and a convergence criteria of 10^{-8}. A precision of 6 decimals is used for both the voltage magnitudes and angles.

Case	**Software**	**Worst $\|V\|$ deviation [pu]**	**Worst θ deviation[degree]**
case14	pypower pandapower	0.0 0.0	0.0 0.0
case30	pypower pandapower	0.0 0.0	0.0 0.0
case118	pypower pandapower	0.0 0.000041	0.0 0.002694
case1354pegase	pytpower pandapower	0.0 0.0	0.0 0.0
case2869pegase	pypower pandapower	0.0 0.0	0.0 0.0
case9241pegase	pypower pandapower	0.0 0.0	0.0 0.0

Table 6.1: The results calculated by Topflow compared with pandapower and pypower when running a Newton-Rapshon load-flow

It can be observed from Table 6.1 that the IEEE-cases show no difference between Topflow and pypower with a precision of 6 decimals. Pandapower gives similar results, except from case118. The voltages calculated by pandapower differs from both Topflow and pypower on that system, with a worst deviation of 0.000041pu for the magnitudes, and 0.002694° for the angles. It's unlikely that this deviation is caused by implementation-errors in any of the packages, since they both pypower and pandapower include validation-models which compares the packages against commercial software [51]. A more likely explanation is that pypower and pandapower uses different input data for the case118. Pandapower has converted the case from pypower, so the source of error may be this transition. This would explain why Topflow and pypower show the same result, since they uses almost identical date-formats (influenced by MATPOWER).

The tests conducted on the PEGASE-cases show no differences among Topflow, pandapower and pypower. These results are of significant importance for the validation of the implemented transformer-model in Topflow. The IEEE-cases use only ideal transformers, which means that the shift-angles are set to 0. The PEGASE-cases on the other hand, are more complex, and introduces phase shift transformers. These cases also includes values different from 0 for the active components of bus-connected shunt-elements (GL). The fact that Topflow finds the same voltages with a precision of more than 10^{-6} show that the program can handle these special cases.

6.3 Comparison of decsolve

The primary focus of this thesis has been on analyzing acsolve. However, the last milestone (see Section 1.2) is to translate the second load-flow function, decsolve, from the original FORTRAN-code. Section 5.5.2 showed how this function is implemented, and that it uses many of the same sub-functions as acsolve. Therefore, most parts of decsolve are already verified by the tests performed on acsolve.

An additional test (see Appendx C.3), compares the voltages calculated by decsolve and acsolve on the same cases that the test from section 6.2 used. The deviation between the two algorithms should be small, since the use the same function to calculate the power injections, and the same condition for terminating the iterative process. The test uses flat-start and a convergence-criteria of 10^{-8} for each simulation. The worst voltage-deviations are calculated with a precision of 10^{-6} (for both the magnitudes and the angles).

Table 6.2 summarizes the results of the test, and shows that all the three FDLF-versions (which are given in Appendix A) calculates satisfactory results for the IEEE-cases. However, all the simulations performed on the PEGASE-cases diverge. Pandapower has implemented the Primal- and the Dual-method, and a quick simulation in Python showed that these methods converged on the PEGASE-cases listed in Table 6.2. These results show that there are current bugs in the implementation of decsolve. The error is most certain located in one of the distinct functions of decsolve, since the sub-functions of acsolve show reliable results in both the automated tests and the comparison-test from section 6.2.

Case	**FDLF-version**	**Succsess-flag**	**Worst $\|V\|$ deviation [pu]**	**Worst θ deviation[deg]**
case14	Standard (XX)	Convergence	0.0	0.0
	Primal (BX)	Convergence	0.0	0.0
	Dual (XB)	Convergence	0.0	0.0
case30	Standard(XX)	Convergence	0.0	0.0
	Primal(BX)	Convergence	0.0	0.0
	Dual (XB)	Convergence	0.0	0.0
case118	Standard (XX)	Convergence	0.0	0.0
	Primal (BX)	Convergence	0.0	0.0
	Dual (XB)	Convergence	0.0	0.0
case1354pegase	Standard (XX)	Divergence	–	–
	Primal (BX)	Divergence	–	–
	Dual (XB)	Divergence	–	–
case2869pegase	Standard (XX)	Divergence	–	–
	Primal (BX)	Divergence	–	–
	Primal (XB)	Divergence	–	–
case9241pegase	Standard (XX)	Divergence	–	–
	Primal (BX)	Divergence	–	–
	Dual (XB)	Divergence	–	–

Table 6.2: The results calculated by Topflow compared with pandapower and pypower when running a Newton-Rapshon load-flow

Apparent bugs in decsolve are not yet resolved, due to the focus and the limited time-frame

of this thesis. The best approach for future contributors in locating the error(s) is to write unit tests for the distinct sub-functions of decsolve. These tests makes it easier to study the action of each part of the program, and resolve the incorrect implementations.

6.4 Conclusion on the reliability of Topflow

Having reliable results of the implemented functions is the most important milestone of this thesis. Writing automated tests is considered a good practice in all stages of a project to ensure that the functions are acting as expected. The fact that all the automated tests passed increases confidence that the implementations are correct. As argued in section 6.1, the automated test do not guarantee that there are no bugs in the program, which is why section 6.2 conducted additional tests. These tests compared simulations performed by Topflow, pypower and pandapower on IEEE- and PEGASE-cases. The results of the simulations compliments the automated tests, and concludes that the Newton-Rapshon load-flow is implemented correctly, and gives reliable results.

Section 6.3 analyzed the reliability of the second load-flow function, decsolve, by running simulations on the cases from section 6.2, and comparing the calculated voltages with acsolve. The results was satisfactory for the IEEE-cases, but not for the PEGASE-cases. The apparent bug is most likely located in one of the distinct sub-functions of decsolve, which is why future contributors are recommended to write unit tests for all the sub-functions of decsolve.

6.5 Performance

The computational performance of Topflow is analyzed by the script in Appendix C.4, which compares the convergence times of Topflow, pypower and pandapower when running the Newton-Rapshon load-flow. The simulations are performed on the cases from section 6.2 that showed lower voltage deviations than 10^{-6} for both the magnitudes and angles. Case118 is the only system that is not suited to compare the performances of the programs. All the load-flows are performed by using flat-start and a convergence criteria of 10^{-8}. The tests are performed on a computer with AMD Ryzen 5 processor at 210 GHz. The module "Timeit", which was discussed in subsection 4.4.4, is used to measure the execution-times of the programs.

Figure 6.3 compares the average values of 100 measurements for each system using Topflow, pandapower and pypower. The calculated times are given in milliseconds. It's observed that the three programs show similar results on all the systems. Topflow is faster than pypower and pandapower for all cases, except for case9241pegase, where pandapower show the shortest convergence time. The similarities are a result of the fact that the three programs uses the same C-library, UMFPACK, through SciPy to solve the sparse linear equation systems. The speed of Topflow is high due to multiple C-extensions, while pandapower uses the just-in-time (jit) compiler numba [52] with success for larger systems [37]. Pypower provides neither, and is therefore the slowest of the three programs.

The jit-compiler used by pandapower translates pure Python-code to machine-code at run-time to give the program speed similar to C and FORTRAN. This compiler will on the other hand cause a delay the first time a function is called, because it has to compile the code [52]. This delay is not included in the calculated times in Figure 6.3, since it was made sure that timeit.timeit() included an initial Newton-Rapshon load-flow in the setup for pandapower, to invoke the jit-compiler before the actual measurements. The script in Appendix C.1 measures

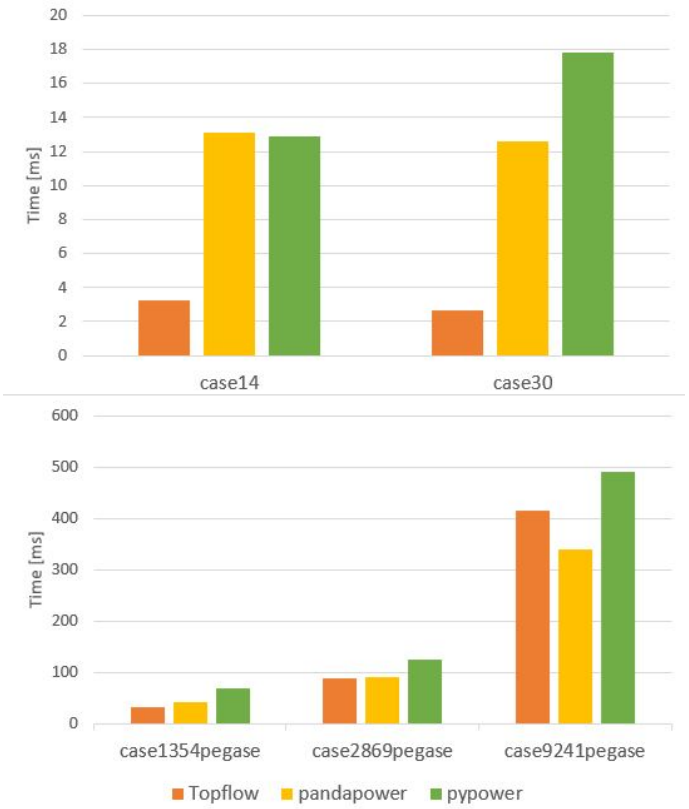


Figure 6.3: Calculation times for the Newton-Rapshon load-flow on Standard MATPOWER cases

the delay by subtracting the first runtime of the Newton-Rapshon function of pandapower by the mean of the next 100 runtimes. This operation is done to ensure that the execution-time of the function itself do not add to the delay. The script was invoked 10 times to force the jit-compiler to translate the function each time. The delay measured 2.75 seconds on average, regardless of the size of the system.

The delay is obviously worth it when the user wants to call the same pandapower-function multiple times, especially on large systems. Figure 6.3 shows that the jit-compiler has less effect for smaller systems, also, the delay makes work which involves few function calls more time consuming, not less. Regardless, the test shows that the jit-compiler is a powerful software that shrinks the gap in performance between Python and C.

One of the advantage of Topflow is that it do not need the jit-compiler to enhance its performance, thanks to the implemented C-extensions. This makes the program efficient on small systems and low-demanding work, as well larger systems and more demanding work.

Chapter 7

Conclusion

This master thesis has conducted work on updating Topflow: a toolbox for power system analysis. NTNU professor Olav Bjarte Fosso wrote the original code in FORTRAN, but previous projects have worked towards the end goal of making the toolbox available in Python. These projects have used C-extensions to preserve the speed of the FORTRAN-program. The contribution of this thesis builds on an initial update of the Newton-Rapshon load-flow function from the specialization project "Toolbox for Specialized Power System Analysis" by Åsmund Sælen [1]. Newly added features to the toolbox includes functions which read and initializes data from Excel-files. These functions make Topflow able to run simulations on large power-systems.

Automated tests are now implemented to ensure the reliability of the code. The result of running a Newton-Rapshon load-flow with Topflow has been compared with corresponding simulations by other programs to complement the automated tests. The largest test-system is case9241pegase, which represents the size and complexity of the European high voltage transmission network [53]. The tests confirm that the implementations are working as expected, and show that Topflow gives reliable results on large systems.

Various profiling-tools have been used to optimize the Newton-Rapshon load-flow function, and justify the design of the Python-interface with C. The most significant optimization is the use of the new solver, scipy.sparse.linalg.spsolve(), which handles sparse matrices. The updated version of Topflow is 45 times faster than the initial version from the specialization project [1], and matches the speed of the open-source projects pandapower and pypower.

The fast-decoupled load-flow routine "decsolve" is also implemented in the updated toolbox presented in this thesis. This implementation is not as well-documented as the Newton-Rapshon load-flow function, but it shares many of the same sub-routines, which have been verified by automated tests.

This thesis's conclusions complete the milestones, and the primary goal set at the start of the working process. Future work should complete the documentation of decsolve, by creating automated tests for the remaining sub-functions, and comparing the result of the load-flow with the Newton-Rapshon activity. The natural next step in translating the remaining parts of the original toolbox is to update the routine for continuation power-flow and contingency analysis. These routines are extensions of the load-flow activities implemented in this thesis, and will introduce new functionality to the program. Hopefully, future contributors can use the documentation presented by this thesis to complete the toolbox, and add additional features to handle the challenges in the future's power systems.

Bibliography

[1] Å. Sælen, "Toolbox for power system analysis," Dec. 2019. DOI: 10.13140/RG.2.2.19031.70564.

[2] H. Kvandal, *Toolbox for specialized power system analysis*. Master Thesis: 2019.

[3] ——, *Toolbox for specialized power system analysis*. Specialization project: 2018.

[4] O. Fosso, *ELK14: Lecture 2 – Modelling and basic Power Flows*. 2018.

[5] J. Backus, "The history of fortran i, ii, and iii," *IEEE Annals of the History of Computing*, vol. 20, no. 4, pp. 68–78, Oct. 1998, ISSN: 1934-1547. DOI: 10.1109/85.728232.

[6] S. U. (1995-7), *Common blocks*, https://web.stanford.edu/class/me200c/tutorial_77/13_common.html, Accessed: 09.09.2019.

[7] Tutorialspoint, *C tutorial*, https://www.tutorialspoint.com/cprogramming/index.htm, accessed: 20.09.2019.

[8] ——, *Why Python is called Dynamically Typed?* https://www.tutorialspoint.com/why-python-is-called-dynamically-typed, accessed: 02.10.2019.

[9] Python^{TM}, *General information - what is python*, https://docs.python.org/3/faq/general.html, accessed: 20.09.2019.

[10] Scipy.org, *Scientific computing tools for python*, https://www.scipy.org/about.html, accessed: 02.06.2020.

[11] ——, *The scipy stack specification*, https://www.scipy.org/stackspec.html, accessed: 02.06.2020.

[12] NumPy.org, *Numpy*, https://numpy.org/, accessed: 18.12.2019.

[13] Scipy.org, *Scipy library*, https://www.scipy.org/scipylib/index.html, accessed: 02.06.2020.

[14] V. Haenel, *Interfacing with c*, https://scipy-lectures.org/advanced/interfacing_with_c/interfacing_with_c.html, accessed: 03.06.2019).

[15] Python^{TM}, *Ctypes- a foreign function library for python*, https://docs.python.org/3.7/library/ctypes.html?highlight=ctypes#module-ctypes, accessed: 15.10.2019).

[16] Microsoft, *About dynamic-link libraries*, https://docs.microsoft.com/en-us/windows/win32/dlls/about-dynamic-link-libraries, accessed: 28.10.2019.

[17] python^{TM}, *Using dlls in practice*, https://docs.python.org/3/extending/windows.html, accessed: 08.06.2019.

[18] S. Knight, *Scons 3.1.1*, https://www.scons.org/doc/HTML/scons-user/, accessed: 28.09.2019).

[19] SciPy.org, *C-types foreign function interface (numpy.ctypeslib)*, https://docs.scipy.org/doc/numpy/reference/routines.ctypeslib.html, accessed: 18.12.2019.

[20] Microsoft, *Getting started*, https://code.visualstudio.com/docs, accessed: 10.09.2019.

[21] L. Langa, *Black the uncomprimising code formatter*, https://github.com/psf/black,accessed:10.09.2019, accessed: 10.09.2019.

[22] Matpower, *Data*, https://github.com/MATPOWER/matpower/blob/master/data, accessed: 30.04.2020.

[23] M. Albadi, "Power flow analysis," *IntechOpen*, pp. 565–576, 2019. DOI: 10.5772/intechopen.83374. [Online]. Available: https://www.intechopen.com/online-first/power-flow-analysis.

[24] E. Kreyszig, *Advanced Engineering Mathematics, tenth edition*. 2011, pp. 302–304.

[25] A. J. Monticelli, A. Garcia, and O. R. Saavedra, "Fast decoupled load flow: Hypothesis, derivations, and testing," *IEEE Transactions on Power Systems*, vol. 5, no. 4, pp. 1425–1431, 1990.

[26] B. Stott and O. Alsac, "Fast decoupled load flow," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-93, no. 3, pp. 859–869, 1974.

[27] R. A. M. van Amerongen, "A general-purpose version of the fast decoupled load flow," *IEEE Transactions on Power Systems*, vol. 4, no. 2, pp. 760–770, 1989.

[28] P. Kundur, *Power System Stability and Control*. New York: McGraw-Hill, Inc, 1993.

[29] SciPy.org, *Scipy.sparse.coo_matrix*, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html#scipy.sparse.coo_matrix, accessed: 14.04.2020.

[30] ——, *Scipy.sparse.csc_matrix*, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csc_matrix.html#scipy.sparse.csc_matrix, accessed: 14.04.2020.

[31] ——, *Scipy.sparse.csr_matrix*, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html#scipy.sparse.csr_matrix, accessed: 14.04.2020.

[32] P. P. Authority, *Packaging python projects*, https://packaging.python.org/tutorials/packaging-projects/, accessed: 13.05.2020.

[33] A. Shaw, *Getting started with testing in python*, https://realpython.com/python-testing/, accessed: 22.05.2020.

[34] pytest, *Pytest: Helps ypu write better programs*, https://docs.pytest.org/en/latest/, accessed: 22.05.2020.

[35] ——, *Pytest fixtures: Explicit, modular, scalable*, https://docs.pytest.org/en/latest/fixture.html, accessed: 22.05.2020.

[36] Matpower, *Case14.m*, https://github.com/MATPOWER/matpower/blob/master/data/case14.m, accessed: 30.04.2020.

[37] L. Thurner, A. Scheidler, F. Schäfer, J. Menke, J. Dollichon, F. Meier, S. Meinecke, and M. Braun, "Pandapower — an open-source python tool for convenient modeling, analysis, and optimization of electric power systems," *IEEE Transactions on Power Systems*, vol. 33, no. 6, pp. 6510–6521, Nov. 2018, ISSN: 0885-8950. DOI: http://dx.doi.org/10.13140/RG.2.2.19031.70564.

[38] R. Lincoln, *Pypower 5.1.4*, https://pypi.org/project/PYPOWER/, accessed: 25.05.2020.

[39] R. Christiey, *Power systems test case archive*, ://labs.ece.uw.edu/pstca/.

[40] S. Fliscounakis, P. Panciatici, F. Capitanescu, and L. Wehenkel, *Ac power flow data in matpower and qcqp format: Itesla, rte snapshots, and pegase*, https://arxiv.org/abs/1603.0, accessed: 13.05.2020.

[41] ——, "Contingency ranking with respect to overloads in very large power systems taking into account uncertainty, preventive, and corrective actions," *IEEE Transactions on Power Systems*, vol. 28, no. 4, pp. 4909–4917, 2013.

[42] python^{TM}, *The python profilers*, https://docs.python.org/3/library/profile.html, accessed: 27.05.2020.

[43] R. Kern, *Line_profiler and kernprof*, https://github.com/pyutils/line_profiler, accessed: 27.05.2020.

[44] python^{TM}, *Timeit — measure execution time of small code snippets*, https://docs.python.org/3/library/timeit.html, accessed: 27.05.2020.

[45] SciPy.org, *Scipy.sparse.linalg.spsolve*, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.spsolve.html#scipy.sparse.linalg.spsolve, accessed: 14.04.2020.

[46] Scipy.org, *Numpy.linalg.solve*, https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html, accessed: 6.11.2019.

[47] R. D. Zimmerman and C. E. Murillo-Sánchez, "Matpower user's manual, version 7.0," 2019. DOI: 10.5281/zenodo.3251118. [Online]. Available: http://www-cs-faculty.stanford.edu/~uno/abcde.html, (accessed: 01.09.2016).

[48] Python^{TM}, *Venv — creation of virtual environments*, https://docs.python.org/3/library/venv.html, accessed: 20.09.2019.

[49] J. J. Grainger and W. D. Stevenson, *Power System Analysis*. McGraw-Hill, 1994, pp. 337–338.

[50] Y. Yao and M. Li, "Designs of fast decoupled load flow for study purpose," *Energy Procedia*, vol. 17, pp. 127–133, Dec. 2012. DOI: 10.1016/j.egypro.2012.02.073.

[51] L. Thurner, A. Scheidler, F. Schäfer, J. Menke, J. Dollichon, F. Meier, S. Meinecke, and M. Braun, *About pandapowers*, http://www.pandapower.org/about/, accessed: 20.05.2020.

[52] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15, Austin, Texas: Association for Computing Machinery, 2015, ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. [Online]. Available: https://doi.org/10.1145/2833157.2833162.

[53] Matpower, *Case9241pegase*, https://raw.githubusercontent.com/MATPOWER/matpower/master/data/case9241pegase.m, accessed: 30.04.2020.

Appendix

Appendix A. contains additional information on the Fast-decoupled load-flow form section subsection 2.5.8. Appendix B. provides the source-code of Topflow. Existing parts of the package that are not utilized in this master's thesis (such as the part Hege Bruvik Kvandal developed concerning DC optimal load-flow [2]) are not included. Appendix C. consists of the tests discussed in chapter 6, which analyses the reliability and performance of the program. The tests uses Excel-files that are too large to include in the Appendix, however, the data is accessible in the GitHub repository [22] where the cases were converted from. Finally, Appendix D. contains the setup.py-file, which is not a part of the Topflow source-code, but located at the root of the repository to support local installations.

A Fast-decoupled load-flow versions

Approximations

1. $|V_i| = 1$
2. $sin(\theta_{ij}) = 0$
3. $cos(\theta_{ij}) = 1$
4. The effect of PV-buses and shunts ar neglected when forming H_{eq} (XB- and XX-version only)
5. L_{eq} is the submatrix L calculated with b_{ij} substituted by $1/x_{ij}$
6. H_{eq} is the submatrix H calculated with b_{ij} substituted by $1/x_{ij}$

   (Note: if the system is radial or have constant r/x ratios, 6. and 7. are not approximations, but exactly true.)

Standard Algorithm (XX-version)

1. Initialize iteration count: $k = 0$
2. Calculated the active power mismatches, and compute angle corrections: $\Delta\theta^{(k)} = \frac{H_{eq}^{-1}\Delta P^{(k)}(|V|^{(k)},\theta^{(k)})}{|V|^{(k)}}$
3. Update the angles: $\theta^{(k+1)} = \theta^{(k)} + \Delta\theta^{(k)}$
4. Calculate the reactive power mismatches, and compute magnitude corrections: $\Delta|V|^{(k)} = \frac{L_{eq}^{-1}\Delta Q^{(k)}(|V|^{(k)},\theta^{(k+1)})}{|V|^{(k+1)}}$
5. Update the magnitudes: $|V|^{(k+1)} = |V|^{(k)} + \Delta|V|^{(k)}$
6. $k = k + 1$, go to step 2.

Primal Algorithm (BX-version)

1. Initialize iteration count: $k = 0$
2. Calculated the active power mismatches, and compute angle corrections: $\Delta\theta^{(k)} = \frac{H^{-1}\Delta P^{(k)}(|V|^{(k)},\theta^{(k)})}{|V|^{(k)}}$
3. Update the angles: $\theta^{(k+1)} = \theta^{(k)} + \Delta\theta^{(k)}$
4. Calculate the reactive power mismatches, and compute magnitude corrections: $\Delta|V|^{(k)} = \frac{L_{eq}^{-1}\Delta Q^{(k)}(|V|^{(k)},\theta^{(k+1)})}{|V|^{(k+1)}}$
5. Update the magnitudes: $|V|^{(k+1)} = |V|^{(k)} + \Delta|V|^{(k)}$
6. $k = k + 1$, go to step 2.

Dual Algorithm (XB-version)

1. Initialize iteration count: $k = 0$
2. Calculate the reactive power mismatches, and compute magnitude corrections: $\Delta|V|^{(k)} = \frac{L^{-1}\Delta Q^{(k)}(|V|^{(k)},\theta^{(k)})}{|V|^{(k)}}$
3. Update the magnitudes: $|V|^{(k+1)} = |V|^{(k)} + \Delta|V|^{(k)}$
4. Calculated the active power mismatches, and compute angle corrections: $\Delta\theta^{(k)} = \frac{H_{eq}^{-1}\Delta P^{(k)}(|V|^{(k+1)},\theta^{(k)})}{|V|^{(k+1)}}$
5. Update the angles: $\theta^{(k+1)} = \theta^{(k)} + \Delta\theta^{(k)}$
6. $k = k + 1$, go to step 2.

B Topflow

This section contains the source-code of Topflow. Appendix sorts the files alphabetically, and Table 1 gives an overview with a short description of the function(s)/class(es) within.

File	Description of the function(s)/class(es)
__init__.py	Defines that Topflow is a package
acsolve_wrapper.py	Wrapper-functions for the C-extensions of the function "acsolve"
acsolve.py	Performs the Newton-Rapshon load-flow
bmatrix.py	Builds the constant sub-matrices B' and B" for the FDLF-method
case.py	Stores the system-parameters, communicates with Excel
coo_conv.c	Restructures the sparse matrix built by "bujac" to coo-format.
decsolve_wrapper.py	Wrapper-functions for the C-extensions of the function "decsolve"
decsolve.py	Performs the fast-decoupled load-flow (FDLF)
enforce_qlim.py	Enforces reactive power limits on generators
flatstart.py	Imposes flat start
jacobi.c	Constructs the Jacobian matrix
jacobi.h	Headers for jacobi.c
loadflow.py	Administers the loadflow-functions "acsolve" and "decsolve".
maxmism.py	Finds the power-mismatches (ΔP and ΔQ)
netinj.c	Calculates the net power-injections (P and Q
sConstruct	Recuired by SCons to build the DLL that exports the C-extensions
select_ver.c	Selects the versions of B' and B" in the FDLF
set_rhs.c	Sets the righ-hand-side of the equation system (2.19)
Settings	Specifies settings (load-flow-version, print- and save-options, etc.)
topflow.h	Defines some basic identities
update_voltages.py	Updates voltage magnitudes, ($\|V\|$) and voltage angles (θ)
zerosp.c	Handles zero impedance lines

Table 1: The files in Toplow's source-code

B.1 __init__.py

```
1 #This file defines Topflow as a python-package
2 #It also makes the functions in case.py and run.py avail-
3 # able for import directly by "from topflow import somefunction"
4 from .case import Case
5 from .loadflow import *
6 from .example_cases.example_cases import example_case
7 from .example_cases.example_cases import example_case_list
8 from .settings import *
```

B.2 acsolve.py

This file contains the function acsolve, which performs the Newton-Rapshon load-flow.

```
1  import sys
2
3  sys.path.append(".")
4  import numpy as np
5  import copy
6  import time
7  import scipy
8  import topflow.acsolve_wrapper as ac
9  from scipy.sparse import csr_matrix
10 from scipy.sparse import isspmatrix_csr
11 from topflow.jacpy import *
12
13
14 #acsolve performs a Newton-Rapshon load-flow
15
16 def acsolve(
17     case,
18     opt
19 ):
20     #Start the timer
21     start_clock = time.time()
22
23     #Make a deep copy to prevent changes of the base-case
24     obj = copy.deepcopy(case)
25     obj.convergence = False
26
27     #set some required values
28     obj.icount = 0
29     melem = 5 * (obj.nbuses + 2 * obj.nlines)
30     pqv = 3
31     #The following values are set to 0, because they are used in CPF,
       not NR:
32     pdelta = 0.0
33     qdelta = 0.0
34     alfa = np.zeros(obj.nbuses, dtype=np.double)
35     beta = np.zeros(obj.nbuses, dtype=np.double)
36
37     ac.flatstart(
38         obj.nbuses,
39         obj.buscod,
40         obj.vomag,
41         obj.voang,
42         opt.flat_start
43         )
44
45     while True:
46
```

```
47          # Caclulate net injection at all buses
48          ac.netinj(
49              pqv,
50              obj.nbuses,
51              obj.nlines,
52              obj.pinj,
53              obj.qinj,
54              obj.vomag,
55              obj.voang,
56              obj.gii,
57              obj.bii,
58              obj.gij,
59              obj.bij,
60              obj.ratio,
61              obj.shift_angle,
62              obj.ifrom,
63              obj.ito,
64              obj.ibstat
65          )
66
67          # Update reactive gneration and calculate the missmatch
    vector
68          # pinj and qinj are updated; they are now pdelta and qdelta (
    rhs)
69          # mism[0] contains the bus with worst Pmism, m[1] contains
    the bus with worst Qmism
70          ac.mismat(
71              pqv,
72              obj.nbuses,
73              obj.genbus,
74              obj.buscod,
75              obj.pgen,
76              obj.qgen,
77              obj.pload,
78              obj.qload,
79              obj.pinj,
80              obj.qinj,
81              alfa,
82              beta,
83              pdelta,
84              qdelta,
85          )
86
87          # Modify for zero impedance lines
88          ac.zerosp(obj.nlines, obj.ifrom, obj.ito, obj.ibstat, obj.
    xinv, obj.pinj)
89          ac.zerosp(obj.nlines, obj.ifrom, obj.ito, obj.ibstat, obj.
    xinv, obj.qinj)
90
```

```
91          # Enforce reactive limitations for generators. If they are on
     the limit, the buscode is changed to -2
92          if (obj.icount > 1 and opt.enf_qlim == True):
93              #Enforce reactive limits
94              obj.mismloc[1] = ac.enforce_qlim(True, obj.nbuses, obj.
    mismloc[1], obj.buscod, obj.genbus, obj.numbus, obj.pinj, obj.qinj
    , obj.qgen, obj.qmin, obj.qmax, obj.vomag, obj.vgbus)
95              #update the number of generators on the reactive limit
96              obj.set_nlimgens()
97
98
99              #Update isa
100             #obj.set_isa()
101
102         # Find worst location of mismatches after zero-imp
    modification
103         ac.maxmism(pqv, obj.nbuses, obj.buscod, obj.mismloc, obj.pinj
    , obj.qinj)
104
105
106         # iteration summary
107         if(opt.print_verbose == 3):
108             print('\niteration:',obj.icount)
109             print('Voltage magnetudes:', obj.vomag)
110             print('Voltage angles:', obj.voang)
111             print("Worst active power mismatch:", obj.pinj[obj.
    mismloc[0]], "at bus:", obj.mismloc[0])
112             print("Worst reactive power mismatch:", obj.qinj[obj.
    mismloc[1]], "at bus:", obj.mismloc[1])
113
114         #Write to Excel file
115         if(opt.filename is not None and opt.save_verbose == 2):
116             #Saves each iteration only if save_verbose is True (its
    False by default)
117             obj.save2xl(opt.filename, opt.save_path, save_verbose =
    True)
118
119         #Terminate?
120         if abs(obj.pinj[obj.mismloc[0]]) < opt.conv_tol and abs(obj.
    qinj[obj.mismloc[1]]) < opt.conv_tol:
121             #Convergence
122             obj.convergence = True
123             stop_clock = time.time()
124             runtime = stop_clock - start_clock
125
126             if(opt.print_verbose != 0):
127                 print("\nConvergence. \nNumber of iterations:", obj.
    icount, 'in', runtime, 'seconds')
128                 if(opt.print_verbose > 1):
129                     print("Final voltage angle:", obj.voang)
```

```
130                     print("Final voltag magnitude:", obj.vomag)
131                     print('pgen', obj.pgen)
132 
133             if(opt.filename is not None):
134                 obj.save2xl(opt.filename, opt.save_path, save_verbose
    = opt.save_verbose)
135             return obj
136         elif obj.icount > opt.max_it:
137             print("\nDivergence")
138             break
139 
140 
141         # Initializes the jacobian lists with -1
142         jacbi = np.zeros((melem), dtype=np.double)
143         jcol = -np.ones((melem), dtype=np.int32)
144         ipv = -np.ones((melem), dtype=np.int32)
145         isa = -np.ones((2*obj.nbuses), dtype = np.int32)
146 
147         # Building the sparse representation of the jacobian matrix
148         ac.bujac(
149             obj.nbuses,
150             obj.nlines,
151             jacbi,
152             obj.vomag,
153             obj.voang,
154             obj.gii,
155             obj.bii,
156             obj.gij,
157             obj.bij,
158             obj.ratio,
159             jcol,
160             ipv,
161             obj.ifrom,
162             obj.ito,
163             obj.ibstat,
164         )
165 
166         #Restructure the matrix form bujac to COO-format:
167 
168         #Find the sixe of the matrix, to initialize the lists for the
    coo-matrix
169         jacsize = ac.jacsize(jcol)
170         #Initialize the list for the coo-matrix
171         row = np.zeros((jacsize), dtype = np.int32)
172         col = np.zeros((jacsize), dtype = np.int32)
173         data = np.zeros((jacsize), dtype = np.double)
174         #Build the COO-matrix
175         ac.coo_conv(
176             obj.nbuses,
177             obj.ngens,
```

```
            obj.buscod,
            isa,
            ipv,
            row,
            jcol,
            col,
            jacbi,
            data
        )


        #Build the sparse matrix on CSR-format
        sparse = csr_matrix((data, (row,col)), [2*obj.nbuses-obj.
    ngens + obj.nlimgens-1, 2*obj.nbuses-obj.ngens + obj.nlimgens-1])


        # Set right hand side in the equations:

        rhs = np.zeros(2*obj.nbuses-obj.ngens + obj.nlimgens - 1)
        ac.set_rhs(
            3,
            obj.nbuses,
            obj.buscod,
            obj.vomag,
            obj.pinj,
            obj.qinj,
            rhs
        )

        # Solve the eqution:
        correction = scipy.sparse.linalg.spsolve(sparse,rhs)


        # update voltages and angles:
        ac.update_voltages(
            3,
            obj.nbuses,
            obj.buscod,
            obj.vomag,
            obj.voang,
            correction
        )

        #obj.savejacobi("LoadFlowResult.xlsx",icount, jacobi)

        obj.icount += 1
```

B.3 acsolve_wrapper.py

```
1 import numpy as np
2 import numpy.ctypeslib as npct
3 from pathlib import Path
4 import ctypes
5 from ctypes.util import find_library
6 from ctypes import c_int
7 from ctypes import c_double
8 from ctypes import c_bool
9 from ctypes import POINTER
10 from ctypes import byref
11 import copy
12 from scipy.sparse import coo_matrix
13 import os
14
15 libpath = str(Path(__file__).parent.absolute()) + '\\clibrary.dll'
16 # input type for the samplelib function
17 # must be a double array, with single dimension that is contiguous
18 ar_1d_double = npct.ndpointer(dtype=np.double, ndim=1, flags="
      CONTIGUOUS")
19 ar_1d_int = npct.ndpointer(dtype=np.int32, ndim=1, flags="CONTIGUOUS"
      )
20 ar_1d_bool = npct.ndpointer(dtype=np.bool, ndim=1, flags="CONTIGUOUS"
      )
21
22
23 clib = ctypes.cdll.LoadLibrary(libpath)
24
25 #aclib.giibii.restype = None
26 #aclib.giibii.argtypes = (
27 #    [c_int] * 3 + [ar_1d_double] * 13 + [ar_1d_int] * 4 + [
      ar_1d_bool] * 2
28 #)
29
30 clib.set_isa.restype = None
31 clib.set_isa.argtypes = [c_int] + [ar_1d_int]*2
32 clib.flatstart.restype = None
33 clib.flatstart.argtypes = [c_int] + [ar_1d_int] + [ar_1d_double]*2 +
      [c_bool]
34
35 clib.netinj.restype = None
36 clib.netinj.argtypes = (
37     [c_int] * 3 + [ar_1d_double] * 10 + [ar_1d_int] * 2 + [ar_1d_bool
      ]
38 )
39
40 clib.mismat.restype = None
41 clib.mismat.argtypes = [c_int]*2 + [ar_1d_int] * 2 + [ar_1d_double] *
       8 + [c_double] * 2
```

```
42
43 clib.maxmism.restype = None
44 clib.maxmism.argtypes = [c_int]*2 + [ar_1d_int] *2 + [ar_1d_double]*2
45
46 clib.enforce_qlim.restype = c_int
47 clib.enforce_qlim.argtypes = (
48     [c_bool] + [c_int] * 2 + [ar_1d_int] * 3 + [ar_1d_double] * 7
49 )
50
51 clib.zerosp.restype = None
52 clib.zerosp.argtypes = [c_int] + [ar_1d_int]*2 + [ar_1d_bool] + [
       ar_1d_double]*2
53
54 clib.set_rhs.restype = None
55 clib.set_rhs.argtypes = [c_int]*2 + [ar_1d_int] + [ar_1d_double]*4
56
57 clib.t_u.restype = None
58 clib.t_u.argtypes = [ar_1d_double] * 2 + [c_double] * 4
59
60 clib.addel.restype = c_int
61 clib.addel.argtypes = (
62     [c_int] * 2 + [c_double] + [c_int] + [ar_1d_double] + [ar_1d_int]
       * 2 + [c_int]
63 )
64
65 clib.bujac.restype = None
66 clib.bujac.argtypes = [c_int] * 2 + [ar_1d_double] * 8 + [ar_1d_int]
       * 4 + [ar_1d_bool]
67
68 clib.jacsize.restype = c_int
69 clib.jacsize.argtypes = [ar_1d_int]
70
71 clib.coo_conv.restype = None
72 clib.coo_conv.argtypes = [c_int]*2 + [ar_1d_int]*6 + [ar_1d_double]*2
73
74 clib.update_voltages.resytpe = None
75 clib.update_voltages.argtypes = [c_int]*2 + [ar_1d_int] + [
       ar_1d_double]*3
76
77 def set_isa(
78     nbuses,
79     buscod,
80     isa
81 ):
82     clib.set_isa(
83         nbuses,
84         buscod,
85         isa
86     )
87
```

```
 88 def flatstart(
 89     nbuses,
 90     buscod,
 91     vomag,
 92     voang,
 93     flat_start
 94 ):
 95     clib.flatstart(
 96     nbuses,
 97     buscod,
 98     vomag,
 99     voang,
100     flat_start
101     )
102 
103 def netinj(
104     pqv,
105     nbuses,
106     nlines,
107     pinj,
108     qinj,
109     vomag,
110     voang,
111     gii,
112     bii,
113     gij,
114     bij,
115     ratio,
116     shift_angle,
117     ifrom,
118     ito,
119     ibstat
120 ):
121     clib.netinj(
122         pqv,
123         nbuses,
124         nlines,
125         pinj,
126         qinj,
127         vomag,
128         voang,
129         gii,
130         bii,
131         gij,
132         bij,
133         ratio,
134         shift_angle,
135         ifrom,
136         ito,
137         ibstat
```

```
138     )
139
140
141 def mismat(
142     pqv,
143     nbuses,
144     genbus,
145     buscod,
146     pgen,
147     qgen,
148     pload,
149     qload,
150     pinj,
151     qinj,
152     alfa,
153     beta,
154     pdelta,
155     qdelta,
156 ):
157     clib.mismat(
158         pqv,
159         nbuses,
160         genbus,
161         buscod,
162         pgen,
163         qgen,
164         pload,
165         qload,
166         pinj,
167         qinj,
168         alfa,
169         beta,
170         pdelta,
171         qdelta,
172     )
173
174 def maxmism(
175     pqv,
176     nbuses,
177     buscod,
178     mismloc,
179     pinj,
180     qinj
181 ):
182     clib.maxmism(
183         pqv,
184         nbuses,
185         buscod,
186         mismloc,
187         pinj,
```

```
        qinj
    )


def enforce_qlim(
    verbose,
    nbuses,
    qmism,
    buscod,
    genbus,
    numbus,
    pinj,
    qinj,
    qgen,
    qmin,
    qmax,
    vomag,
    vgbus,
):
    return clib.enforce_qlim(
        verbose,
        nbuses,
        qmism,
        buscod,
        genbus,
        numbus,
        pinj,
        qinj,
        qgen,
        qmin,
        qmax,
        vomag,
        vgbus,
    )

def zerosp(
    nlines,
    ifrom,
    ito,
    ibstat,
    xinv,
    xinj
):
    clib.zerosp(
        nlines,
        ifrom,
        ito,
        ibstat,
        xinv,
        xinj
```

```
238     )
239
240 def set_rhs(
241     pqv,
242     nbuses,
243     buscod,
244     vomag,
245     pinj,
246     qinj,
247     rhs
248 ):
249     clib.set_rhs(
250     pqv,
251     nbuses,
252     buscod,
253     vomag,
254     pinj,
255     qinj,
256     rhs
257     )
258
259 def t_u(u, t, delta_angle, gij, bij, ratio):
260     clib.t_u(u, t, delta_angle, gij, bij, ratio)
261
262
263 def addel(i, j, elem, idim, aa, jcol, ipv, ip):
264     return clib.addel(i, j, elem, idim, aa, jcol, ipv, ip)
265
266
267 def bujac(
268     nbuses,
269     nlines,
270     jacbi,
271     vomag,
272     voang,
273     gii,
274     bii,
275     gij,
276     bij,
277     ratio,
278     jcol,
279     ipv,
280     ifrom,
281     ito,
282     ibstat,
283 ):
284     clib.bujac(
285         nbuses,
286         nlines,
287         jacbi,
```

```
        vomag,
        voang,
        gii,
        bii,
        gij,
        bij,
        ratio,
        jcol,
        ipv,
        ifrom,
        ito,
        ibstat,
    )

def jacsize(jcol):
    return clib.jacsize(jcol)


def coo_conv(
    nbuses,
    ngens,
    buscod,
    isa,
    ipv,
    row,
    jcol,
    col,
    jacbi,
    data
):
    clib.coo_conv(
        nbuses,
        ngens,
        buscod,
        isa,
        ipv,
        row,
        jcol,
        col,
        jacbi,
        data
    )

def update_voltages(
    pqv,
    nbuses,
    buscod,
    vomag,
    voang,
    correction
```

```
338 ):
339     clib.update_voltages(
340     pqv,
341     nbuses,
342     buscod,
343     vomag,
344     voang,
345     correction
346 )
```

B.4 bmatrix.c

```
1 #include<stdbool.h>
2 #include<math.h>
3
4
5 __declspec(dllexport) void bmatrix(double *xx, double *value, int *
    row, int *col, int *isa, int *ifrom, int *ito, bool *ibstat, int
    nbuses, int nlines, int pqv){
6     //Bygg submatrisene
7     //Tar inn enten B eller X som xx
8     //bver str for B-matrix version: 1 = B', 2 = B''
9     //Kva diagonaler som skal vere store blir modifisert seinare nr
    du veit kva type det skal vere
10
11     int i,k;
12     int j = 0;
13     int l = nbuses;
14     int offset = 0;
15
16     if(pqv == 2){
17         //the bus-positions in B'' lies in the scond half of isa
18         offset += nbuses;
19     }
20
21     for(k=0; k<nlines; k++)
22     {
23         if(ibstat[k]){
24
25             //Add the values of the diagonal elements
26             value[ifrom[k]] += xx[k];
27             value[ito[k]] += xx[k];
28
29             //Add the position of the diagonal elements
30
31             row[ifrom[k]] = ifrom[k];
32             col[ifrom[k]] = ifrom[k];
33
34             row[ito[k]] = ito[k];
35             col[ito[k]] = ito[k];
36
37             //Set a big number at a diagonal element of a bus which
    is not included in the jacobien
38             //That is slack for J1 or generator buses for J4
39             if(isa[ifrom[k] + offset] == -1 && value[ifrom[k]] < pow
    (10,10)){
40                 value[ifrom[k]] += pow(10,10);
41             }
42             if(isa[ito[k] + offset] == -1 && value[ito[k]] < pow
    (10,10)){
```

```
43                  value[ito[k]] += pow(10,10);
44              }
45
46              //Build the lower triangle of the Bmatrix (its
    symmetrical)
47              //if(ifrom[k] > ito[k]){
48                  value[l] = -xx[k];
49                  row[l] = ifrom[k];
50                  col[l] = ito[k];
51                  l++;
52              //}
53              //else{
54                  value[l] = -xx[k];
55                  row[l] = ito[k];
56                  col[l] = ifrom[k];
57                  l++;
58             // }
59
60          }
61      }
62  }
```

B.5 case.py

Class variable	Description	Datatype
sbase	Apparent power base of the system	double
Bus data		
nbuses	Number of buses in the system	int
busname[nbuses]	Name of the buses	numpy.str
area	Area postions for each bus	numpy.str
zone	Zone postition for each bus	numpy.str
numbus[nbuses]	External bus numbers	numpy.int32
buscod[nbuses]	List of the buscodes for all buses; 1-PQ, 2-PV, 3-Slack, -2-PV (VAR limited)	numpy.int32
basekv[nbuses]	Base voltage at each bus	numpy.double
gl[nbuses]	Active component of bus-connected shunt element	numpy.double
bl[nbuses]	Reactive component of bus-connected shunt element	numpy.double
vomag[nbuses]	Voltage magnetudes in pu	numpy.double
voang[nbuses]	Voltage angles in radians	numpy.double
pload[nbuses]	Active load vector for the base case	numpy.double
qload[nbuses]	Reactive load vector for the base case	numpy.double
Generator data		
nlimgens	Number of Generators on the reactive limit	int
ngens	Number of generators in the system	numpy.int32
genbus[nbuses]	Pointers to internal generators	numpy.int32
vgbus[ngens]	Voltage setpoint for the generators	numpy.double
pgen[ngens]	Active power for generators	numpy.double
qgen[ngens]	Reactive power for generators	numpy.double
pmax	Maximum active generation	numpy.double
qmax	Maximum reactive generation	numpy.double
qmax	Minimum reactive generation	numpy.double
Line data		
nlines	Number of lines in the system	int
ifrom[nlines]	From bus number for all lines	numpy.int32
ito[nlines]	To bus number for all lines	numpy.int32
ibstat[nlines]	Indicates status for all lines (In service: 1)	numpy.bool
xinv[nlines]	Inverse of the line reactances	numpy.double
gij[nlines]	Real part of line suceptance	numpy.double
bij[nlines]	Imaginary part of line suceptance	numpy.double
gii[nlines]	Real part of the diagonal element in the Ybus	numpy.double
bii[nlines]	Imaginary part the diagonal element in the Ybus	numpy.double
tapno[nlines]	Pointers to transformer description (0-not a transformer)	numpy.int32

Table 2: Class variables of the Case-class. [] denotes the size of the array

```
1 import os
2 import openpyxl as opyxl
3 import numpy as np
4 import cmath
5 import math
6 import copy
7
8 #This file contains the Case-class, which is used to store the
9 #data of power-systems
10
11 class Case:
12     def __init__(
13     self, filepath = None
14     ):
15         #Case data
16         self.icount = 0
17         self.convergence = False
18
19
20         #Bus data
21         self.sbase = 500 #Default MVA value
22         self.numbus = np.empty(0, dtype = np.int32)
23         self.slackbusnr = []
24         self.busname = np.empty(0, dtype = np.str)
25         self.buscod = np.empty(0, dtype = np.int32)
26         self.basekv = np.empty(0, dtype = np.double)
27         self.gs = np.empty(0, dtype = np.double)
28         self.bs = np.empty(0, dtype = np.double)
29         self.area = np.empty(0, dtype = np.int32)
30         self.zone = np.empty(0, dtype = np.int32)
31         self.nbuses = 0
32         self.vomag = np.empty(0, dtype = np.double)
33         self.voang = np.empty(0, dtype = np.double)
34         self.mismloc = np.zeros(2, dtype = np.int32)
35
36         #Generator data
37         self.genbus = np.empty(0, dtype = np.int32)
38         self.ngens = 0
39         self.nlimgens = 0
40         self.genstat = np.empty(0, dtype = np.bool)
41         self.genbase = np.empty(0, dtype = np.double)
42         self.vgbus = np.empty(0, dtype = np.double)
43         self.pgen = np.empty(0, dtype = np.double)
44         self.qgen = np.empty(0, dtype = np.double)
45         self.qmax = np.empty(0, dtype = np.double)
46         self.qmin = np.empty(0, dtype = np.double)
47         self.pmax = np.empty(0, dtype = np.double)
48         self.pmin = np.empty(0, dtype = np.double)
49         self.qlim_out = 0.01
50         self.qlim_in = 0.005
```

```
51          self.dvlim = 0.002
52
53          #Branch data
54          self.nlines = 0
55          self.ito = np.empty(0, dtype = np.int32)
56          self.ifrom = np.empty(0, dtype = np.int32)
57          self.xinv = np.empty(0, dtype = np.double)
58          self.b = np.empty(0, dtype = np.double)
59          self.gi = np.empty(0, dtype = np.double)
60          self.bi = np.empty(0, dtype = np.double)
61          self.gj = np.empty(0, dtype = np.double)
62          self.bj = np.empty(0, dtype = np.double)
63          self.gii = np.empty(0, dtype = np.double)
64          self.bii = np.empty(0, dtype = np.double)
65          self.gij = np.empty(0, dtype = np.double)
66          self.bij = np.empty(0, dtype = np.double)
67          self.ratio = np.empty(0, dtype = np.double)
68          self.shift_angle = np.empty(0, dtype = np.double)
69          self.ibstat = np.empty(0, dtype = np.bool)
70          self.tapno = np.empty(0, dtype = np.int32)
71
72          #Load data
73          self.pload = np.empty(0, dtype = np.double)
74          self.qload = np.empty(0, dtype = np.double)
75          self.pdelta = np.empty(0, dtype = np.double)
76          self.qdelta = np.empty(0, dtype = np.double)
77          self.alfa = np.empty(0, dtype = np.double)
78          self.beta = np.empty(0, dtype = np.double)
79
80
81
82          if(filepath is not None):
83              self.loadxl(filepath)
84
85
86      def loadxl(self,filepath):
87          #This functions reads data from an Excel-file which has a
    specific format,
88          #and initilizes the inctance variables.
89          #"filepath" can only be the name of the file if it's saved in
     the same folder as the toolbox
90          #it's therefore recomended to use the full path of the file
    when calling loadxl()
91
92          #Open the Excel-file, and get the sheets
93          wb = opyxl.load_workbook(filepath)
94          case = wb['Case-identification']
95          bus = wb['Bus-data']
96          gen = wb['Generator-data']
97          line =wb['Line-data']
```

```
98
99      #Case identification
100         istart = 1
101         jstart = 1
102         #In case the user did not begin the data in cell (1,1);
103         stop = False
104         if(str(case.cell(row = istart, column = jstart).value) != 'IC
    '):
105             for r in range(1, 40):
106                 for c in range(1,29):
107                     if(str(case.cell(row = r, column = c).value) == '
    IC'):
108                         istart = r
109                         jstart = c
110                         stop = True
111                         break
112                 if(stop):
113                     break
114         #Read the data from the case-sheet
115         self.sbase = float(case.cell(row = istart+1, column = jstart
    + 1).value)
116
117
118      #Bus data
119         self.nbuses = 0
120
121         istart = 1
122         jstart = 1
123
124         #In case the user did not begin the data in cell (1,1);
125         stop = False
126         if(str(bus.cell(row = istart, column = jstart).value) != 'I')
    :
127             for r in range(1, 40):
128                 for c in range(1,29):
129                     if(str(bus.cell(row = r, column = c).value) == 'I
    '):
130                         istart = r
131                         jstart = c
132                         stop = True
133                         break
134                 if(stop):
135                     break
136
137
138         buscod_col = jstart + 2
139         pload_col = jstart + 3
140         qload_col = jstart + 4
141         gs_col = jstart + 5
142         bs_col = jstart + 6
```

```
143         area_col = jstart + 7
144         vomag_col = jstart + 8
145         voang_col = jstart + 9
146         basekv_col = jstart + 10
147         zone_col = jstart + 11
148 
149         #Read the data from the bus-sheet
150         i = istart + 1
151         while (bus.cell(row = i,column = jstart).value is not None):
152 
153             self.numbus = np.append(self.numbus,int(bus.cell(row = i,
     column = jstart).value))
154             if(int(bus.cell(row = i, column = buscod_col).value) ==
    3):
155                 self.slackbusnr = self.nbuses
156             self.buscod = np.append(self.buscod, int(bus.cell(row = i
    , column = buscod_col).value))
157             self.pload = np.append(self.pload, float(bus.cell(row = i
    , column = pload_col ).value)/self.sbase)
158             self.qload = np.append( self.qload, float(bus.cell(row =
    i, column = qload_col).value)/self.sbase)
159             self.gs = np.append(self.gs, float(bus.cell(row = i,
    column = gs_col).value)/self.sbase)
160             self.bs = np.append(self.bs, float(bus.cell(row = i,
    column = bs_col).value)/self.sbase)
161             self.area = np.append(self.area,int(bus.cell(row = i,
    column = area_col).value))
162             self.vomag = np.append(self.vomag, float(bus.cell(row = i
    , column = vomag_col).value))
163             self.voang = np.append(self.voang, float(bus.cell(row = i
    , column = voang_col).value)/180*math.pi)
164             self.basekv  = np.append(self.basekv,float(bus.cell(row =
     i, column = basekv_col).value))
165             self.zone = np.append(self.zone,int(bus.cell(row = i,
    column = zone_col).value))
166             self.nbuses += 1
167             i+=1
168 
169         self.pinj = np.zeros(self.nbuses, dtype=np.double)
170         self.qinj = np.zeros(self.nbuses, dtype=np.double)
171         self.set_isa()
172 
173 
174      #Generator data
175 
176         istart = 1
177         jstart = 1
178         self.genbus = -np.ones(self.nbuses,dtype = np.int32)
179 
180         #In case the user did not begin the data in cell (1,1);
```

```
181         stop = False
182         if(str(gen.cell(row = istart + 1, column = jstart).value) !=
    'I'):
183             for r in range(1, 40):
184                 for c in range(1,29):
185                     if(str(gen.cell(row = r, column = c).value) == 'I
    '):
186                         istart = r
187                         jstart = c
188                         stop = True
189                         break
190                 if(stop):
191                     break
192 
193         pgen_col = jstart + 1
194         qgen_col = jstart + 2
195         qmax_col = jstart + 3
196         qmin_col = jstart + 4
197         vgbus_col = jstart + 5
198         genbase_col = jstart + 6
199         genstat_col = jstart + 7
200         pmax_col = jstart + 8
201         pmin_col = jstart + 9
202 
203         #Read the data from the generato-sheet
204         i = istart + 1
205         while (gen.cell(row = i, column = jstart).value is not None):
206             mbase = float(gen.cell(i,genbase_col).value)
207             self.bs[self.ext2int(int(gen.cell(row = i, column =
    jstart).value))] = self.bs[self.ext2int(int(gen.cell(row = i,
    column = jstart).value))]*self.sbase/mbase
208             self.genbus[self.ext2int(int(gen.cell(row = i, column =
    jstart).value))] = i-istart-1
209             self.pgen = np.append(self.pgen, float(gen.cell(row = i,
    column = pgen_col).value)/mbase)
210             self.qgen = np.append(self.qgen, float(gen.cell(row = i,
    column = qgen_col).value)/mbase)
211             self.qmax = np.append(self.qmax, float(gen.cell( row = i,
     column = qmax_col).value)/mbase)
212             self.qmin = np.append(self.qmin, float(gen.cell( row = i,
     column = qmin_col).value)/mbase)
213             self.vgbus = np.append(self.vgbus, float(gen.cell( row =
    i, column = vgbus_col).value))
214             self.vomag[self.ext2int(int(gen.cell(row = i, column =
    jstart).value))] = float(gen.cell( row = i, column = vgbus_col).
    value)
215             self.genbase = np.append(self.genbase, mbase)
216             self.genstat = np.append(self.genstat, bool(gen.cell(i,
    genstat_col).value))
217             self.pmax = np.append(self.pmax, float(gen.cell(i,
```

```
    pmax_col).value)/mbase)
218             self.pmin = np.append(self.pmin, float(gen.cell(i,
    pmin_col).value)/mbase)
219             i += 1
220         self.ngens = self.pgen.size
221 
222      #Line data
223         istart = 1
224         jstart = 1
225         #In case the user did not begin the data in cell (1,1);
226         stop = False
227         if(str(line.cell(row = istart + 1, column = jstart).value) !=
     'I'):
228             for r in range(1, 40):
229                 for c in range(1,29):
230                     if(str(line.cell(row = r, column = c).value) == '
    I'):
231                         istart = r
232                         jstart = c
233                         stop = True
234                         break
235                 if(stop):
236                     break
237 
238 
239         ito_col = jstart + 1
240         r_col = jstart + 2
241         x_col = jstart + 3
242         b_col = jstart + 4
243         gi_col = jstart + 8
244         bi_col = jstart + 9
245         gj_col = jstart + 10
246         bj_col = jstart + 11
247         ratio_col = jstart + 12
248         shift_col = jstart + 13
249         ibstat_col = jstart + 14
250 
251         #Read the data from the line-sheet
252         i = istart + 1
253         while (line.cell(row = i,column = jstart).value is not None):
254             busi = int(line.cell(row = i, column = jstart).value)
255             busj = int(line.cell(row = i, column = ito_col).value)
256             r = float(line.cell(row = i, column = r_col).value)
257             x = float(line.cell(row = i, column = x_col).value)
258             ratio = float(line.cell(row = i, column = ratio_col).
    value)
259 
260             self.ifrom = np.append(self.ifrom, self.ext2int(busi))
261             self.ito = np.append(self.ito, self.ext2int(busj))
262             self.xinv = np.append(self.xinv, 1/x)
```

```
263             self.b = np.append(self.b, float(line.cell(row = i,
    column = b_col).value))
264             self.gi = np.append(self.gi,float(line.cell(row = i,
    column = gi_col).value))
265             self.bi = np.append(self.bi,float(line.cell(row = i,
    column = bi_col).value))
266             self.gj = np.append(self.gj,float(line.cell(row = i,
    column = gj_col).value))
267             self.bj = np.append(self.bj,float(line.cell(row = i,
    column = bj_col).value))
268             self.ratio = np.append(self.ratio, ratio) if ratio != 0.0
     else np.append(self.ratio, 1.0)
269             self.shift_angle = np.append(self.shift_angle, float(line
    .cell(row = i, column = shift_col).value)/180*math.pi)
270             self.ibstat = np.append(self.ibstat, bool(line.cell(row =
     i, column = ibstat_col).value))
271
272             self.gij = np.append(self.gij,(1/complex(r,x)).real)
273             self.bij = np.append(self.bij,(1/complex(r,x)).imag)
274             self.nlines += 1
275             i += 1
276
277         self.gii = np.zeros(self.nbuses, dtype = np.double)
278         self.bii = np.zeros(self.nbuses, dtype = np.double)
279         self.tapno = -np.ones(self.nlines)
280         self.giibii()
281
282
283
284     def save2xl(self, filename, save_path = None, save_verbose = 1):
285
286         #This function saves the Case-object to a Excel file (which
    it creates)
287
288         #In case the user did not end the file-name with ".xlsx":
289         save_as = filename if filename[-5:] == '.xlsx' else filename
    + '.xlsx'
290         if(save_path is not None):
291             #Specify the location of the result-file. (The default
    location will be the current working dir)
292             save_as = save_path + '/' + save_as
293
294         #The Final-result-sheet
295         if(self.convergence):
296             #Final-results sheet
297             try:
298                 #If the workbook already exist: overwrite it
299                 wb = opyxl.load_workbook(save_as)
300                 try:
301                     #If a Final-result sheet already exist, reset it
```

```
302                     ws = wb.get_sheet_by_name('Final-results')
303                     wb.remove_sheet(ws)
304                     ws = wb.create_sheet("Final-results")
305                 except:
306                     #If it doesent exist, create it
307                     ws = wb.create_sheet("Final-results")
308                 if(save_verbose == 1):
309                     try:
310                         #Try to delete the iteration sheet from
    previous loadflows
311                         ws2 = wb.get_sheet_by_name("Iteration-summary
    ")
312                         wb.remove_sheet(ws2)
313                     except:
314                         pass
315             except:
316                 #If the workbook doesen't exist: create a new one
317                 wb = opyxl.Workbook()
318                 ws = wb.active
319                 ws.title = 'Final-results'
320 
321 
322             ws.append([])
323             ws.append(["Iterations",self.icount])
324             ws.append(['Worst mismatch', 'mism[pu]', 'Bus'])
325             ws.append(['P',self.pinj[self.mismloc[0]],self.mismloc
    [0]])
326             ws.append(['Q',self.qinj[self.mismloc[1]], self.mismloc
    [1]])
327             ws.append([])
328             ws.append(['Bus', 'type', 'Vomag[V]', 'Voang[deg]','Pload
    ', 'Qload', 'Pgen', 'Qgen','Qmax', 'Qmin', 'Violation'])
329             for i in range(0, self.nbuses):
330                 #Check if there are any reactive power violations
331                 ig = self.genbus[i]
332                 qmax = None if ig == -1 else self.qmax[ig]
333                 qmin = None if ig == -1 else self.qmin[ig]
334                 violation = None
335                 if(ig>=0):
336                     violation = 1 if self.qgen[ig]>self.qmax[ig] or
    self.qgen[ig]<self.qmin[ig] else 0
337                 #Write generated power to the sheet only if it is a
    generator
338                 pg = None
339                 qg = None
340                 if(self.genbus[i] != -1):
341                     pg = self.pgen[self.genbus[i]]
342                     qg = self.qgen[self.genbus[i]]
343                 ws.append([self.numbus[i], self.buscod[i], self.vomag
    [i], self.voang[i]*180/math.pi, self.pload[i], self.qload[i], pg,
```

```
    qg, qmax, qmin, violation])
344
345         elif(save_verbose == 2):
346             #Iteration-summary sheet
347             try:
348                 #Check if the workbook already exist
349                 wb = opyxl.load_workbook(save_as)
350                 if(self.icount == 0):
351                     #Initialize the Iteration sheet
352                     try:
353                         #If a Iteration sheet already exist,
    overwrite it
354                         ws2 = wb.get_sheet_by_name("Iteration-summary
    ")
355                         wb.remove_sheet(ws2)
356                         ws2 = wb.create_sheet("Iteration-summary")
357                     except:
358                         #If it doesent exist, create it
359                         ws2 = wb.create_sheet("Iteration-summary")
360                 else:
361                     #Add data to the sheet in each iteration
362                     ws2 = wb.get_sheet_by_name("Iteration-summary")
363             except:
364                 #The workbook doesen't exist: create a new one
365                 wb = opyxl.Workbook()
366                 ws2 = wb.active
367                 ws2.title = "Iteration-summary"
368
369             #Append this to the sheet each iteration:
370             ws2.append([])
371             ws2.append(["iteration",self.icount])
372             ws2.append(['Worst mismatch', 'mism[pu]', 'Bus'])
373             ws2.append(['P',self.pinj[self.mismloc[0]],self.mismloc
    [0]])
374             ws2.append(['Q',self.qinj[self.mismloc[1]], self.mismloc
    [1]])
375             ws2.append([])
376             ws2.append(['Bus','type', 'Vomag[V]', 'Voang[deg]', '
    Pload', 'Qload', 'Pgen', 'Qgen','Qmax', 'Qmin', 'Violation'])
377             for i in range(0, self.nbuses):
378                 #Check if there are any reactive power violations:
379                 ig = self.genbus[i]
380                 qmax = None if ig == -1 else self.qmax[ig]
381                 qmin = None if ig == -1 else self.qmin[ig]
382                 violation = None
383                 if(ig>=0):
384                     violation = 1 if self.qgen[ig]>self.qmax[ig] or
    self.qgen[ig]<self.qmin[ig] else 0
385                 #Write generated power to the sheet only if it is a
    generator
```

```
386                 pg = None
387                 qg = None
388                 if(self.genbus[i] != -1):
389                     pg = self.pgen[self.genbus[i]]
390                     qg = self.qgen[self.genbus[i]]
391                 ws2.append([self.numbus[i], self.buscod[i], self.
    vomag[i], self.voang[i]*180/math.pi, self.pload[i], self.qload[i],
     pg, qg,qmax, qmin, violation])
392
393
394         wb.save(save_as)
395
396         if(self.convergence):
397             #pop up the excel file
398             os.startfile(save_as)
399
400
401      #isa[2*nbuses] is a list which contains the buses position in the
         jacobian matrix.
402      #If an element = -1, the row is not a part of the jacobian matrix
403      def set_isa(self):
404         ip = 0
405         iq = self.nbuses-1
406         self.isa = -np.ones(self.nbuses*2, dtype = np.int32)
407         for ib in range(0,self.nbuses):
408             if(self.buscod[ib] < 3):
409                 self.isa[ib] = ip
410                 ip += 1
411             if(self.buscod[ib]<2):
412                 self.isa[ib+self.nbuses] = iq
413                 iq += 1
414
415      def giibii(self):
416         #This function is used by loadxl() when it reads the line-
    data.
417         #It calculates system parameters that are needed to
418         # impose the impacts of transfromers and shunt-elements.
419         #Consider writing it as a C-extension
420         self.gii = copy.deepcopy(self.gs)
421         self.bii = copy.deepcopy(self.bs)
422
423
424         for il in range(0,self.nlines):
425             ifr = self.ifrom[il]
426             itr = self.ito[il]
427
428             if(self.ibstat[il] != 0):
429
430                 self.gii[ifr] += self.gi[il]
431                 self.bii[ifr] += self.bi[il]
```

```
432                 self.gii[itr] += self.gj[il]
433                 self.bii[itr] += self.bj[il]
434 
435                 tap = 1/self.ratio[il]
436 
437                 self.gii[ifr] += self.gij[il]*tap*tap
438                 self.bii[ifr] += (self.bij[il] + self.b[il]/2)*tap*
    tap
439 
440                 self.gii[itr] += self.gij[il]
441                 self.bii[itr] += (self.bij[il] + self.b[il]/2)
442                 #self.xii[ifr] += tap*tap*(self.xinv[il] + self.b[il
    ]/2)
443                 #self.xii[itr] += self.xinv[il]*tap + self.b[il]/2
444 
445     def maxmism(self, pqv):
446         #Only used for comparison of speed between python and C.
447         #It's replaced by the C-extesnion "maxmism"
448 
449         #Finds the largest missmatch and return the positionvector
    mismloc
450         #mismloc[0]: position of worst P mismatch in pinj
451         #mismloc[1]: position of worst Q mismatch in qinj
452         ptemp = 0
453         qtemp = 0
454         for ib in range(0, self.nbuses):
455             if(self.buscod[ib] != 3 and abs(self.pinj[ib]) > ptemp
    and pqv != 2):
456                 self.mismloc[0] = ib
457                 ptemp = abs(self.pinj[self.mismloc[0]])
458 
459             if(self.buscod[ib] < 2 and abs(self.qinj[ib]) > qtemp and
     pqv != 1):
460                 self.mismloc[1] = ib
461                 qtemp = abs(self.qinj[self.mismloc[1]])
462 
463     def set_nlimgens(self):
464         self.nlimgens = 0
465         for k in self.buscod:
466             if k == -2: self.nlimgens +=1
467 
468     def get(self, component, number, variable):
469         #the get function gives the user access the the parameters of
     a Case-instance
470 
471         #To handle both a singe component, and a list of componenets
472         number = np.array(number)
473         output = np.zeros(0)
474 
475         if(str.lower(component) == 'bus'):
```

```
476         #Gets access to data at the buses (loads and generators
    included)
477             #Check if the busnumber is correct
478 
479             if(number.size == 1):
480                 return self.get_bus(number, variable)
481             else:
482                 for i in range(0,number.size):
483                     if(self.get_bus(number[i], variable) == None):
484                         return None
485                     output = np.append(output, self.get_bus(number[i
    ], variable))
486                 return output
487 
488         elif(str.lower(component) in ['gen', 'generator']):
489             #Get access to generator-parameters
490             if(number.size == 1):
491                 return self.get_gen(number, variable)
492             else:
493                 for i in range(0,number.size):
494                     if(self.get_gen(number[i], variable) == None):
495                         return None
496                     output = np.append(output, self.get_gen(number[i
    ], variable))
497                 return output
498 
499         elif(str.lower(component) == 'line'):
500             #Get access to line-parameters
501             if(number.size == 1):
502                 return self.get_line(number, variable)
503             else:
504                 for i in range(0, number.size):
505                     if(self.get_line(number[i], variable) == None):
506                         return None
507                     output = np.append(output, self.get_line(number[i
    ], variable))
508                 return output
509 
510     def get_bus(self,busnumber, variable):
511         if(busnumber not in ['all', 'ALL']):
512             try:
513                 busnumber = self.ext2int(busnumber)
514             except:
515                 print('Error:',busnumber, ' is not a external bus-
    number' )
516                 return None
517 
518 
519         if(str.lower(variable) == 'vm'):
520             output = self.vomag if busnumber in ['all', 'ALL'] else
```

```
    self.vomag[busnumber]
521         elif(str.lower(variable) == 'va'):
522             output = self.voang/math.pi*180 if busnumber in ['all', '
    ALL'] else self.voang[busnumber]/math.pi*180
523         elif(str.lower(variable) == 'pd'):
524             output = self.pload if busnumber in ['all', 'ALL'] else
    self.pload[busnumber]
525         elif(str.lower(variable) == 'qd'):
526             output = self.qload if busnumber in ['all', 'ALL'] else
    self.qload[busnumber]
527         elif(str.lower(variable) == 'pq'):
528             output = self.pload if busnumber in ['all', 'ALL'] else
    self.pload[self.genbus[busnumber]]
529         elif(str.lower(variable) == 'qg'):
530             output = self.qload if busnumber in ['all', 'ALL'] else
    self.qload[self.genbus[busnumber]]
531         else:
532             print('Error: ',busnumber, ' has no variable ', variable)
533             return None
534         return output
535
536     def get_gen(self, genumber, variable):
537
538         if(genumber not in ['all', 'ALL'] and (self.ngens<genumber or
     genumber<0) ):
539             print('Error: ', genumber, 'is not a valid generator-
    number')
540             return None
541
542         if(str.lower(variable) == 'vm'):
543             output = self.vomag if genumber in ['all', 'ALL'] else
    self.vomag[genumber]
544         elif(str.lower(variable) == 'va'):
545             output = self.voang/math.pi*180 if genumber in ['all', '
    ALL'] else self.voang[genumber]/math.pi*180
546         elif(str.lower(variable) == 'pg'):
547             output = self.pgen if genumber in ['all', 'ALL'] else
    self.pgen[genumber]
548         elif(str.lower(variable) == 'qg'):
549             output = self.qgen if genumber in ['all', 'ALL'] else
    self.qgen[genumber]
550         else:
551             print('Error: ',genumber, ' has no variable ', variable)
552             return None
553         return output
554
555     def get_line(self, genumber, variable):
556         #Gets access to line-flow data
557
558         if(self.ngens<genumber or genumber<0):
```

```
559             print('Error: ', genumber, 'is not a valid generator-
    number')
560             return None
561 
562         if(str.lower(variable) == 'pf'):
563             return None
564         elif(str.lower(variable) == 'qf'):
565             return None
566         elif( str.lower(variable) == 'pt'):
567             return None
568         elif( str.lower(variable) == 'qt'):
569             return None
570         else:
571             print('Error: ', variable, ' is not a valid line-variable
    ')
572             return None
573 
574 
575     def print_buses(self):
576 
577         print('Bus data:')
578         print('Busnr:',self.numbus)
579         print('Busnames:',self.busname)
580         print('BaseKV:',self.basekv)
581         print('Buscode:', self.buscod)
582         print('slack-bus:',self.int2ext(self.slackbusnr))
583         print('Bus-connected shunt(Active comp), gl:',self.gs)
584         print('Bus-connected shunt(Reactive comp), bl',self.bs)
585         print('Area:',self.area)
586         print('Zones:',self.zone)
587         print('Voltage magnitudes:',self.vomag)
588         print('Voltage angles',self.voang)
589         print('Pinj:',self.pinj)
590         print('Qinj:',self.qinj)
591         print('Active load:',self.pload)
592         print('Reactive load:',self.qload)
593 
594     def print_gens(self):
595 
596         print('Generator data:')
597         print('Genbus:',self.genbus)
598         print('Ngen:', self.ngens)
599         print('pgen:',self.pgen)
600         print('qgen:',self.qgen)
601         print('qmax:',self.qmax)
602         print('qmin:',self.qmin)
603         print('vgbus:',self.vgbus)
604 
605     def print_lines(self):
606 
```

```
607         print('Linedata:')
608         print('Ifrom:', self.ifrom)
609         print('Ito:', self.ito)
610         print('GII:', self.gii)
611         print('BII:', self.bii)
612         print('GIJ:', self.gij)
613         print('BIJ:', self.bij)
614         print('Ratio:', self.ratio)
615         print('Shift_angle:',self.shift_angle)
616 
617     def print_all(self):
618         #print all the data
619         self.print_buses()
620         print('\n')
621         self.print_gens()
622         print('\n')
623         self.print_lines()
624 
625     def ext2int(self,busnr):
626         #convert busnumber (from external to internal)
627         return int(np.where(self.numbus == busnr)[0][0])
628 
629     def int2ext(self,busnr):
630         #convert busnumber (from internal to external)
631         return self.numbus[busnr]
```

B.6 coo_conv.c

```
1 #include <stdbool.h>
2 #include <math.h>
3
4
5 __declspec(dllexport) void coo_conv(int nbuses, int ngens, int *
    buscod, int *isa, int *ipv, int *row, int *jcol, int *col, double
    *jacbi, double *data){
6
7     int ib, ip, iq, nz, ir, ic, inext;
8
9     ip = 0;
10    iq = nbuses - 1;
11
12    //set isa
13    for(ib = 0; ib <nbuses; ib++){
14        isa[ib] = -1;
15        isa[ib+nbuses] = -1;
16        if(abs(buscod[ib] != 3)){
17            isa[ib] = ip;
18            ip++;
19        }
20        if(abs(buscod[ib])<2){
21            isa[ib+nbuses] = iq;
22            iq ++;
23        }
24    }
25
26    //copy the information of jacbi, ipv, and jcol
27    //over to data, row and col, which will be on coo-format
28
29    nz = 0;
30
31    for(ir = 0; ir<2*nbuses; ir++){
32        if(isa[ir] != -1){
33            inext = ir;
34            while(inext != -1){
35                ic = jcol[inext];
36                if(isa[ic] != -1 && jacbi[inext] != 0.0){
37                    row[nz] = isa[ir];
38                    col[nz] = isa[ic];
39                    data[nz] = jacbi[inext];
40                    nz ++;
41                }
42                inext = ipv[inext];
43            }
44
45        }
46
```

```
47          }
48
49
50  }
```

B.7 decsolve_wrapper.py

```
1 import numpy as np
2 from pathlib import Path
3 import numpy.ctypeslib as npct
4 import ctypes
5 from ctypes.util import find_library
6 from ctypes import c_int
7 from ctypes import c_double
8 from ctypes import c_bool
9 from ctypes import c_char_p
10 from ctypes import POINTER
11 from ctypes import byref
12 import copy
13 from scipy.sparse import coo_matrix
14 import os
15
16 libpath = str(Path(__file__).parent.absolute()) + '\\clibrary.dll'
17
18 # input type for the samplelib function
19 # must be a double array, with single dimension that is contiguous
20 ar_1d_double = npct.ndpointer(dtype=np.double, ndim=1, flags="
      CONTIGUOUS")
21 ar_1d_int = npct.ndpointer(dtype=np.int32, ndim=1, flags="CONTIGUOUS"
      )
22 ar_1d_bool = npct.ndpointer(dtype=np.bool, ndim=1, flags="CONTIGUOUS"
      )
23
24 clib = ctypes.cdll.LoadLibrary(libpath)
25
26 clib.h_matrix.restype = None
27 clib.h_matrix.argtypes = [ar_1d_double] * 3 + [ar_1d_int] * 5 + [
      c_int] * 2
28
29 clib.l_matrix.restype = None
30 clib.l_matrix.argtypes = [ar_1d_double] * 3 + [ar_1d_int] * 5 + [
      c_int] * 4
31
32 clib.heq_matrix.restype = None
33 clib.heq_matrix.argtypes = [ar_1d_double] * 2 + [ar_1d_int] * 5 + [
      c_int] * 3
34
35 clib.leq_matrix.restype = None
36 clib.leq_matrix.argtypes = [ar_1d_double] * 2 + [ar_1d_int] * 5 + [
      c_int] * 4
37
38 clib.bmatrix.restype = None
39 clib.bmatrix.argtypes = [ar_1d_double] * 2 + [ar_1d_int] * 6 + [
      ar_1d_bool] + [c_int] * 3
40
```

```
clib.selectver.restype = None
clib.selectver.argtypes = [ar_1d_double]*4 + [c_int] + [c_char_p]

'''def netpinj(
    nbuses,
    nlines,
    pinj,
    vomag,
    voang,
    gii,
    bii,
    gij,
    bij,
    ratio,
    ifrom,
    ito,
    ibstat
):
    clib.netpinj(
        nbuses,
        nlines,
        pinj,
        vomag,
        voang,
        gii,
        bii,
        gij,
        bij,
        ratio,
        ifrom,
        ito,
        ibstat
    )

def netqinj(
    nbuses,
    nlines,
    qinj,
    vomag,
    voang,
    gii,
    bii,
    gij,
    bij,
    ratio,
    ifrom,
    ito,
    ibstat
):
    clib.netqinj(
```

```
91          nbuses,
92          nlines,
93          qinj,
94          vomag,
95          voang,
96          gii,
97          bii,
98          gij,
99          bij,
100         ratio,
101         ifrom,
102         ito,
103         ibstat
104     )
105
106
107 def pmismat(
108     nbuses,
109     genbus,
110     buscod,
111     pgen,
112     pload,
113     pinj,
114     beta,
115     pdelta,
116 ):
117     clib.pmismat(
118         nbuses,
119         genbus,
120         buscod,
121         pgen,
122         pload,
123         pinj,
124         beta,
125         pdelta
126     )
127
128 def qmismat(
129     nbuses,
130     genbus,
131     buscod,
132     qgen,
133     qload,
134     qinj,
135     alfa,
136     qdelta,
137 ):
138     clib.qmismat(
139         nbuses,
140         genbus,
```

```
141         buscod,
142         qgen,
143         qload,
144         qinj,
145         alfa,
146         qdelta,
147     )
148 '''
149 def h_matrix(bii, bij, buscod, ifrom, ito, nbuses, nlines):
150     melem = 5 * (nbuses + 2 * nlines)
151     value = np.zeros(melem, dtype = np.double)
152     row = np.zeros(melem, dtype = np.int32)
153     col = np.zeros(melem, dtype = np.int32)
154 
155     clib.h_matrix(bii, bij, value, row, col, buscod, ifrom, ito,
    nbuses, nlines)
156     erase = np.where(value == 0.0)
157     value = np.delete(value, erase)
158     row = np.delete(row, erase)
159     col = np.delete(col, erase)
160     return coo_matrix((value, (row,col)), [nbuses-1, nbuses-1])
161 
162 def l_matrix(bii, bij, buscod, ifrom, ito, nbuses, nlines, ngens,
    nglim):
163     melem = 2 * (nbuses + 2*nlines)
164     value = np.zeros(melem, dtype = np.double)
165     row = np.zeros(melem, dtype = np.int32)
166     col = np.zeros(melem, dtype = np.int32)
167 
168 
169     clib.l_matrix(bii, bij, value, row, col, buscod, ifrom, ito,
    nbuses, nlines, ngens, nglim)
170     erase = np.where(value == 0.0)
171     value = np.delete(value, erase)
172     row = np.delete(row, erase)
173     col = np.delete(col, erase)
174     return coo_matrix((value, (row,col)), [nbuses-ngens + nglim,
    nbuses-ngens + nglim])
175 
176 def heq_matrix(xinv, buscod, ifrom, ito, nbuses, nlines, slackbusnr):
177     melem = 2 * (nbuses + 2*nlines)
178     value = np.zeros(melem, dtype = np.double)
179     row = np.zeros(melem, dtype = np.int32)
180     col = np.zeros(melem, dtype = np.int32)
181 
182 
183     clib.heq_matrix(xinv, value, row, col, buscod, ifrom, ito, nbuses
    , nlines, slackbusnr)
184     erase = np.where(value == 0.0)
185     value = np.delete(value, erase)
```

```
186     row = np.delete(row, erase)
187     col = np.delete(col, erase)
188     return coo_matrix((value, (row,col)), [nbuses-1, nbuses-1])
189
190 def leq_matrix(xinv, buscod, ifrom, ito, nbuses, nlines, ngens, nglim
        ):
191     melem = 2 * (nbuses + 2*nlines)
192     value = np.zeros(melem, dtype = np.double)
193     row = np.zeros(melem, dtype = np.int32)
194     col = np.zeros(melem, dtype = np.int32)
195
196
197     clib.leq_matrix(xinv, value, row, col, buscod, ifrom, ito, nbuses
        , nlines, ngens, nglim)
198     erase = np.where(value == 0.0)
199     value = np.delete(value, erase)
200     row = np.delete(row, erase)
201     col = np.delete(col, erase)
202     return coo_matrix((value, (row,col)), [nbuses-ngens + nglim,
        nbuses-ngens + nglim])
203
204 def bmatrix(xx, buscod, ifrom, ito, ibstat, nbuses, nlines, pqv):
205     melem = 2 * (nbuses + 2*nlines)
206     value = np.zeros(melem, dtype = np.double)
207     row = np.zeros(melem, dtype = np.int32)
208     col = np.zeros(melem, dtype = np.int32)
209     isa = -np.ones((2*nbuses), dtype = np.int32)
210
211     clib.bmatrix(xx, value, row, col, buscod, isa, ifrom, ito, ibstat
        , nbuses, nlines, pqv)
212     '''erase = np.where(value == 0.0)
213     value = np.delete(value, erase)
214     row = np.delete(row, erase)
215     col = np.delete(col, erase)'''
216
217     return coo_matrix((value, (row,col)), [nbuses, nbuses])
218
219 def selectver(xp, xq, xinv, bij, nlines, bver):
220     clib.selectver(xp, xq, xinv, bij, nlines, bver.encode('utf-8'))
```

B.8 decsolve.py

```
1 import sys
2
3 sys.path.append(".")
4 import numpy as np
5 import copy
6 from scipy.sparse.linalg import spsolve
7 from scipy.sparse import csr_matrix
8 import topflow.acsolve_wrapper as ac
9 import topflow.decsolve_wrapper as dec
10
11
12 #decsolve performs a fast-decoupled load-flow
13 #Versions:
14 #FDXX (Standard-method): pqv = 1, bver = 'XX'
15 #FDBX (Primal-method): pqv = 1, bver = 'BX'
16 #FDXB (Dual-method): pqv = 2, bver = 'XB'
17 #Custom: the user can choose iteration-sequence(pqv) and the versions
       of B' and B''(bver)
18
19 def decsolve(case, opt):
20
21
22     #make a deep copy to prevent change of the base-case
23     obj = copy.deepcopy(case)
24     obj.convergence = False
25
26     pdelta = 0.0
27     qdelta = 0.0
28     xp = np.zeros(obj.nlines)
29     xq = np.zeros(obj.nlines)
30     alfa = np.zeros(obj.nbuses, dtype=np.double)
31     beta = np.zeros(obj.nbuses, dtype=np.double)
32     genlim = np.zeros(obj.ngens, dtype = np.bool)
33
34     if(opt.pqv == 1):
35             #The standard starts with P-calculations
36             pqv = 1
37     elif(opt.pqv == 2):
38         #Dual algorithm starts with Q-calculations
39         pqv = 2
40     else:
41         print('Error: pqv must have the value 1 or 2, not', opt.pqv)
42
43     #Set flat start (or not)
44     ac.flatstart(
45       obj.nbuses,
46       obj.buscod,
47       obj.vomag,
```

```
48        obj.voang,
49        opt.flat_start
50        )
51
52     #Select bm and bmm version (XX, XB, BX, BB)
53     dec.selectver(xp, xq, obj.xinv, obj.bij, obj.nlines, opt.bver)
54
55     #Build the constant matrices
56     bm = csr_matrix(dec.bmatrix(xp,obj.buscod, obj.ifrom, obj.ito,
       obj.ibstat, obj.nbuses, obj.nlines, 1))
57     bmm = csr_matrix(dec.bmatrix(xq,obj.buscod, obj.ifrom, obj.ito,
       obj.ibstat, obj.nbuses, obj.nlines, 2))
58
59     while True:
60         # Caclulate net injection (pinj or qinj) at all buses
61
62         ac.netinj(
63             pqv,
64             obj.nbuses,
65             obj.nlines,
66             obj.pinj,
67             obj.qinj,
68             obj.vomag,
69             obj.voang,
70             obj.gii,
71             obj.bii,
72             obj.gij,
73             obj.bij,
74             obj.ratio,
75             obj.shift_angle,
76             obj.ifrom,
77             obj.ito,
78             obj.ibstat
79         )
80
81         # Update reactive gneration and calculate the missmatch
       vector
82         # pinj and qinj are updated; they are now pdelta and qdelta (
       rhs)
83         # mism[0] contains the bus with worst Pmism, m[1] contains
       the bus with worst Qmism
84
85         ac.mismat(
86             pqv,
87             obj.nbuses,
88             obj.genbus,
89             obj.buscod,
90             obj.pgen,
91             obj.qgen,
92             obj.pload,
```

```
93              obj.qload,
94              obj.pinj,
95              obj.qinj,
96              alfa,
97              beta,
98              pdelta,
99              qdelta
100         )
101
102         # Modify for zero impedance lines
103         if(pqv == 1):
104             ac.zerosp(obj.nlines, obj.ifrom, obj.ito, obj.ibstat, obj
    .xinv, obj.pinj)
105         elif(pqv == 2):
106             ac.zerosp(obj.nlines, obj.ifrom, obj.ito, obj.ibstat, obj
    .xinv, obj.qinj)
107
108         # Find worst location of mismatches after zero-imp
    modification
109         ac.maxmism(pqv, obj.nbuses, obj.buscod, obj.mismloc, obj.pinj
    , obj.qinj)
110
111         #Enfroce reactive limitations
112         if (obj.icount > 1.5 and opt.enf_qlim == True and pqv == 2):
113             obj.mismloc[1] = ac.enforce_qlim(True, obj.nbuses, obj.
    mismloc[1],  obj.buscod, obj.genbus, obj.numbus, obj.pinj, obj.
    qinj, obj.qgen, obj.qmin, obj.qmax, obj.vomag, obj.vgbus)
114             #Update bmm
115             for ib in range(0, obj.nbuses):
116                 if(obj.buscod[ib] == -2 and bmm[ib,ib] >= pow(10,10))
    :
117                     #Set PV bus as Var-limited
118                     bmm[ib,ib] -= pow(10,10)
119                 elif(obj.buscod[ib] == 2 and bmm[ib,ib] < pow(10,10))
    :
120                     #Set Var-limited PV bus back as PV-bus
121                     bmm[ib,ib] += pow(10,10)
122
123         # iteration summary
124         if(opt.print_verbose == 3):
125             print('\niteration:',obj.icount)
126             print('Voltage magnetudes:', obj.vomag)
127             print('Voltage angles:', obj.voang)
128             print("Worst active power mismatch:", obj.pinj[obj.
    mismloc[0]], "at bus:", obj.mismloc[0])
129             print("Worst reactive power mismatch:", obj.qinj[obj.
    mismloc[1]], "at bus:", obj.mismloc[1])
130
131           #Write to Excel file
132         if(opt.filename is not None and opt.save_verbose == 2):
```

```
133             #Saves each iteration only if save_verbose is True (its
    False by default)
134             obj.save2xl(filename = opt.filename, save_path = opt.
    save_path, save_verbose = 2)
135
136         #Terminate?
137         if abs(obj.pinj[obj.mismloc[0]]) < opt.conv_tol and abs(obj.
    qinj[obj.mismloc[1]]) < opt.conv_tol:
138             #Convergence
139             obj.convergence = True
140
141             if(opt.print_verbose != 0):
142                 print("\nConvergence. Number of iterations:", obj.
    icount)
143                 if(opt.print_verbose > 1):
144                     print("Final voltage angle:", obj.voang)
145                     print("Final voltag magnitude:", obj.vomag)
146
147             if(opt.filename is not None):
148                 obj.save2xl(opt.filename, opt.save_path, save_verbose
     = opt.save_verbose)
149
150             return obj
151         elif obj.icount > opt.max_it*2:
152             print("\nDivergence")
153             return 0
154
155
156         #Right hand side in the equations:
157         rhs = np.zeros(obj.nbuses)
158         ac.set_rhs(
159             pqv,
160             obj.nbuses,
161             obj.buscod,
162             obj.vomag,
163             obj.pinj,
164             obj.qinj,
165             rhs
166         )
167
168         #Solve the equation
169
170         if(pqv == 1):
171             correction = spsolve(bm, rhs)
172         elif(pqv == 2):
173             correction = spsolve(bmm, rhs)
174
175
176         #Update voltages (magnitude or angles):
177
```

```
178         ac.update_voltages(
179             pqv,
180             obj.nbuses,
181             obj.buscod,
182             obj.vomag,
183             obj.voang,
184             correction
185         )
186 
187         if(pqv == 1):
188             #The standard ends with Q-calculations
189             pqv = 2
190         elif(pqv == 2):
191             #Dual algorithm ends with P-calculations
192             pqv = 1
193 
194         obj.icount += 0.5
```

B.9 enforce_qlim.c

```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4 #define ABS(x) (((x) < 0) ? (-(x)) : ((x))) /* if loop: ABS(X) gives
    the absolute value of x*/
5
6 //verbose = True; specificaions are printed to the screen.
7
8 __declspec(dllexport) int enforce_qlim(bool verbose, int nbuses, int
    qmism,
9                 int *buscod, int *genbus, int *numbus, double *pinj,
10                double *qinj, double *qgen, double *qmin, double *
    qmax, double *vomag, double *vgbus)
11 {
12     double qmarg;
13     bool violation;
14     int ig;
15     int s = 1;
16
17     for (int k=0; k<nbuses; k++)
18     {
19         ig = genbus[k];
20         if (ig >= 0)
21         {
22             if (buscod[k] >= 2)
23                 {
24                     /* Enforce violated reactive units */
25                     violation = (qmax[ig] < qgen[ig]);
26                     /* Exceeds maximum ?*/
27                     if (violation)
28                         {
29                             if (verbose)
30                                 {
31                                     printf("Ulim gen @ bus %5d ",
    numbus[k]);
32                                     printf("qmax = % 5.4f, qgen = %
    6.4f \n", qmax[ig], qgen[ig]);
33                                 }
34
35
36                                 //qmax is now the setpoint, find the
    missmatch (qinj[k])
37                                 qinj[k] = qmax[ig] - qgen[ig];
38                                 //the reactive generation is now
    fixed at maximum
39                                 qgen[ig] = qmax[ig];
40
41
```

```
42                              //choose another slack-bus if
43                              //the current slack-bus violates the
    limits
44                              if(buscod[k] == 3){
45                                  pinj[k] = 0.0;
46                                  //chose the next PV-bus as the new
    slack-bus
47                                  if(s == nbuses){
48                                      //go to the start of buscod if
    slack-bus
49                                      // is the last generator
50                                      s -= nbuses;
51                                  }
52                                  while(buscod[k+s] != 2){
53                                      s++;
54                                  }
55                                  buscod[k+s] = 3;
56                                  s = 1;
57                              }
58                              buscod[k] = -2;
59                          }
60                      else
61                          {
62                              //Below minimum?
63                              violation = (qmin[ig] > qgen[ig]);
64
65                              if (violation){
66                                  if(verbose)
67                                      {
68                                          printf("Llim gen @ bus %5d ",
     numbus[k]);
69                                          printf("qmin = % 5.4f, qgen =
     % 6.4f \n", qmin[ig], qgen[ig]);
70                                      }
71
72                                  //qmin is now the setpoint, find the
    missmatch(qinj[k])
73                                  qinj[k] = qmin[ig] - qgen[ig];
74                                  //the reactive generation is now
    fixed at minumum
75                                  qgen[ig] = qmin[ig];
76
77
78                                  //choose another slack-bus if it
    violates the limit
79                                  if(buscod[k] == 3){
80                                      pinj[k] = 0.0; //0 mismatch
81                                      //if the slack bus is on the
    limit;
82                                      //choose the next PV-bus as the
```

```
    new slack-bus
83                                      if(s >= nbuses){
84                                          s = 1;
85                                      }
86
87                                      while(buscod[k+s] != 2){
88                                          s++;
89                                      }
90                                      buscod[k+s] = 3;
91                                      s = 1;
92                                  }
93                                  buscod[k] = -2;
94                              }
95                          }
96                      if (violation &&ABS(qinj[k]) > ABS(qinj[qmism])){
97                          //update the worst mismatch
98                          qmism = k;
99                      }
100                 }
101                 //Relax nonbinding units
102
103             else if (buscod[k] == -2 && qmin[ig] != qmax[ig] )
104                 {
105                     if (vgbus[ig] == 0.0)
106                     {
107                         vgbus[ig] = vomag[k];
108                     }
109
110                     if ((qgen[ig] == qmax[ig] && vomag[k]>vgbus[ig])
111                     || (qgen[ig] == qmin[ig] && vomag[k]<vgbus[ig]))
112                         {
113                             if(verbose)
114                             {
115                                 printf("Relaxed generator constraints
    on %d \n", numbus[k]);
116                             }
117
118                             vomag[k] = vgbus[ig];
119                             buscod[k] = 2;
120                             qinj[k] = 0;
121                         }
122                 }
123         }
124     }
125     return qmism;
126 }
```

B.10 flatstart.c

```
1 #include <stdbool.h>
2 #include <math.h>
3
4 __declspec(dllexport) void flatstart(int nbuses, int *buscod, double
    *vomag, double *voang, bool flat_start){
5
6 int ib;
7
8 if(flat_start == true){
9     for(ib= 0; ib<nbuses; ib++){
10         if(buscod[ib] != 3){
11             voang[ib] = 0.0;
12             if(buscod[ib] != 2 && vomag[ib] != 1.0){
13                 vomag[ib] = 1.0;
14             }
15         }
16     }
17 }
18
19 }
```

B.11 jacobi.c

```
1 #include <stdbool.h>
2 #include <math.h>
3
4
5 #include "jacobi.h"
6
7 void t_u(double *u, double *t, double delta_angle, double gij, double
      bij, double ratio)
8 {
9     double cosda, sinda;
10
11     if(ratio != 0.0){
12         cosda = cos(delta_angle)/ratio;
13         sinda = sin(delta_angle)/ratio;
14     }
15     else{
16         cosda = cos(delta_angle);
17         sinda = sin(delta_angle);
18     }
19
20     *u = gij*sinda - bij*cosda;
21     *t = gij*cosda + bij*sinda;
22
23 }
24
25 int addel(int i, int j, double elem, int idim, double *aa, int *jcol,
      int *ipv, int ip)
26 {
27
28     if (i == j)
29         {
30             aa[i] += elem;
31             jcol[i] = i;
32         }
33     else
34         while (true)
35             if (ipv[i] == -1) //0 used in fortran. -1 here because
     index begins with 0 in C/python
36                 {
37                     ipv[i] = ip;
38                     jcol[ip] = j;
39                     aa[ip] = elem;
40                     ip++;
41                     break;
42                 }
43             else
44                 {
45                     i = ipv[i];
```

```
                    if (jcol[i] == j)
                        {
                            aa[i] += elem;
                            break;
                        }
                }
    return ip;
}

void bujac(int nbuses, int nlines,
           double *jacbi, double *vomag, double *voang, double *gii,
    double *bii, double *gij,
           double *bij, double *ratio,
           int *jcol, int *ipv, int *ifrom, int *ito, bool *ibstat
           )
{

    int k, i,j, i2, j2, idim, i_p, i_q, ip;
    double volt, elem, delta_angle;
    double uij[1], uji[1], tij[1], tji[1];

    idim = 2*nbuses;
    ip = 2*nbuses;
    for (i=0; i<nbuses; i++)
        {
            volt = 2*vomag[i];
            elem = volt*gii[i];

            j = i + nbuses;
            ip = addel(i, j, elem, idim, jacbi, jcol, ipv, ip);
            elem = -volt*bii[i];
            ip = addel(j, j, elem, idim, jacbi, jcol, ipv, ip);
        }

    for (k=0; k<nlines; k++)
    {
        if (ibstat[k])
        {
            i = ifrom[k];
            j = ito[k];
            i2 = i + nbuses;
            j2 = j + nbuses;

            delta_angle = voang[i] - voang[j];
            volt = vomag[i]*vomag[j];

            t_u(uij, tij, delta_angle, gij[k], bij[k], ratio[k]);
            t_u(uji, tji,-delta_angle, gij[k], bij[k], ratio[k]);

            // DP/DA
```

```
95              elem = -volt**uij;
96              ip = addel(i, j, elem, idim, jacbi, jcol, ipv, ip);
97              ip = addel(i, i,-elem, idim, jacbi, jcol, ipv, ip);
98
99              elem = -volt**uji;
100             ip = addel(j, i, elem, idim, jacbi, jcol, ipv, ip);
101             ip = addel(j, j,-elem, idim, jacbi, jcol, ipv, ip);
102
103             // DP/DV
104             elem = -vomag[i]**tij;
105             ip = addel(i, j2, elem, idim, jacbi, jcol, ipv, ip);
106             elem = -vomag[j]**tji;
107             ip = addel(j, i2, elem, idim, jacbi, jcol, ipv, ip);
108             elem = -vomag[j]**tij;
109             ip = addel(i, i2, elem, idim, jacbi, jcol, ipv, ip);
110             elem = -vomag[i]**tji;
111             ip = addel(j, j2, elem, idim, jacbi, jcol, ipv, ip);
112
113             // DQ/DA
114             elem = volt**tij;
115             ip = addel(i2, j, elem, idim, jacbi, jcol, ipv, ip);
116             ip = addel(i2, i,-elem, idim, jacbi, jcol, ipv, ip);
117             elem = volt**tji;
118             ip = addel(j2, i, elem, idim, jacbi, jcol, ipv, ip);
119             ip = addel(j2, j,-elem, idim, jacbi, jcol, ipv, ip);
120
121             // DQ/DV
122             elem = -vomag[i]**uij;
123             ip = addel(i2, j2, elem, idim, jacbi, jcol, ipv, ip);
124             elem = -vomag[j]**uij;
125             ip = addel(i2, i2, elem, idim, jacbi, jcol, ipv, ip);
126
127             elem = -vomag[j]**uji;
128             ip = addel(j2, i2, elem, idim, jacbi, jcol, ipv, ip);
129             elem = -vomag[i]**uji;
130             ip = addel(j2, j2, elem, idim, jacbi, jcol, ipv, ip);
131
132
133         }
134     }
135 }
136
137 int jacsize(int *jcol){
138     //This routine finds the number of element in the
139     //sparse jacobian matrix bult by bujac,
140     //which are differnet form 0.
141     int size = 0;
142     while(true){
143         if(jcol[size] == -1){
144             return size;
```

```
145             }
146             size++;
147         }
148         return 0;
149 }
```

B.12 jacobi.h

```
1 #ifndef HEADER_jacobi
2 #define HEADER_jacobi
3
4 __declspec(dllexport) void t_u(double *u, double *t, double
     delta_angle, double gij, double bij, double ratio);
5 __declspec(dllexport) int addel(int i, int j, double elem, int idim,
     double *aa, int *jcol, int *ipv, int ip);
6 __declspec(dllexport) void bujac(int nbuses, int nlines,
7             double *jacbi, double *vomag, double *voang, double *gii,
     double *bii, double *gij,
8             double *bij, double *ratio,
9             int *jcol, int *ipv, int *ifrom, int *ito, bool *ibstat
10            );
11
12 __declspec(dllexport) int jacsize(int *jcol);
13 #endif /* HEADER_jacobi */
```

B.13 loadlflow.py

```
1
2 from topflow.case import Case
3 from topflow.acsolve import acsolve
4 from topflow.settings import Settings
5 from topflow.decsolve import decsolve
6 import copy
7
8 #This is the administrative function of the load-flow algorithms
    acsolve and decsolve
9
10 def loadflow(Case, mysettings = Settings(), version = None,
    flat_start = None, enf_qlim = None, max_it = None, conv_tol = None
    , pqv = None, filename = None, save_path = None, save_verbose =
    None, print_verbose = None):
11
12     args = locals()
13     myset_copy = copy.deepcopy(mysettings)
14     for i in args:
15         if( i not in ['Case', 'mysettings', 'args'] and args[i] is
    not None):
16             myset_copy.set_var(i, args[i])
17
18
19
20     if(myset_copy.version == 'NR'):
21         result = acsolve(Case, myset_copy)
22     elif(myset_copy.version in ['FDXB', 'FDBX', 'FDXX', 'FDBB']):
23         result =  decsolve(Case, myset_copy)
24     else:
25         print(myset_copy.version,' is not a valid load flow version')
26         return None
27     return result
```

B.14 maxism.c

```
1 #include <stdbool.h>
2 #include <math.h>
3 #include "topflow.h"
4
5 __declspec(dllexport) void maxmism(int pqv, int nbuses, int *buscod,
    int *mismloc, double *pinj, double *qinj){
6
7     int ib;
8     double ptemp = 0.0;
9     double qtemp = 0.0;
10
11    for(ib = 0; ib < nbuses; ib++){
12
13            if((buscod[ib] != 3) && (ABS(pinj[ib]) > ptemp) && (pqv
    != 2)){
14                mismloc[0] =  ib;
15                ptemp = ABS(pinj[mismloc[0]]);
16            }
17            if((buscod[ib] < 2) && (ABS(qinj[ib]) > qtemp) && (pqv !=
     1)){
18                mismloc[1] =  ib;
19                qtemp = ABS(qinj[mismloc[1]]);
20            }
21
22
23    }
24
25 }
```

B.15 mismat.c

```
1
2 #include "math.h"
3
4
5 void pmismat(int ib, int *genbus, int *buscod, double *pgen, double *
    pload, double *pinj,
6                 double *beta, double pdelta)
7 {
8     //This subroutine calculates the active power mismatch and
    updates the active generation
9     //for the bus ib
10
11     int ig = genbus[ib];
12     if(abs(buscod[ib]) == 2){
13         //pinj is now the mismatch of the PV-bus (including Var-
    limited PV-bus)
14         pinj[ib] = pgen[ig]-pload[ib]-pinj[ib]-beta[ib]*pdelta;
15     }
16     else if (buscod[ib] == 3){
17         //Update active generation for the slack bus
18         pgen[ig]=pinj[ib]+pload[ib] + beta[ib]*pdelta;
19         //pinj[ib] = 0;
20     }
21     else{
22         // Active mismatches of PQ-buses
23         pinj[ib]=-pload[ib]-pinj[ib]-beta[ib]*pdelta;
24     }
25 }
26
27 void qmismat(int ib, int *genbus, int *buscod, double *qgen, double *
    qload, double *qinj,
28                 double *alfa, double qdelta)
29 {
30     //This subroutine calculates the reactive power missmatch and
    updates the reactive generation
31     //for the bus ib
32
33     int ig = genbus[ib];
34
35     if(buscod[ib] >= 2){
36         //Update reactive genertation at PV- and slack-buses
37         qgen[ig]= qinj[ib]+qload[ib] + alfa[ib]*qdelta;
38         //qinj[ib] = 0;
39     }
40     else if(buscod[ib] == -2){
41         //Reactive mismatch of the Var-limited generator buses
42         qinj[ib] = qgen[ig]-qload[ib]-qinj[ib]-alfa[ib]*qdelta;
43     }
```

```
44      else{
45          //Reactive mismatches of PQ-buses
46          qinj[ib]=-qload[ib]-qinj[ib]-alfa[ib]*qdelta;
47      }
48  }
49
50  __declspec(dllexport) void mismat(int pqv, int nbuses, int *genbus,
        int *buscod,
51                  double *pgen, double *qgen, double *pload, double *
        qload, double *pinj, double *qinj,
52                   double *alfa, double *beta, double pdelta, double
        qdelta)
53  {
54      int ib;
55      //This subroutine calculates the relevant mismatches, and updates
        the active and/or reactive generations
56      //pqv decides what to calculate
57      //pqv = 1: Active power, 2: Reactive power, 3: Both active and
        reactive power
58
59      for (ib=0;ib<nbuses;ib++){
60          if (buscod[ib] != 4)
61          {
62              if(pqv != 1){
63                  qmismat(ib, genbus, buscod, qgen, qload, qinj, alfa,
        qdelta);
64              }
65              if(pqv !=2){
66                  pmismat(ib, genbus, buscod, pgen, pload, pinj, beta,
        pdelta);
67              }
68          }
69      }
70  }
```

B.16 netinj.c

```
1 #include <stdbool.h>
2 #include <math.h>
3 #include "jacobi.h"
4
5 __declspec(dllexport) void netinj(int pqv, int nbuses, int nlines,
    double *pinj, double *qinj, double *vomag, double *voang,
6             double *gii, double *bii, double *gij, double *bij,
    double *ratio, double *shift_angle, int *ifrom, int *ito, bool *
    ibstat)
7 {
8     int i, j, k;
9     double volt, volt2, delta_angle, sina, cosa;
10    double uij[1],tij[1], uji[1],tji[1];
11
12    /*Calculate the injection on all buses*/
13    //pqv decides what netinj calculates:
14    //1: pinj, 2: qinj, 3: both pinj and qinj
15
16    for (i=0; i<nbuses; i++)
17        {
18            volt2 = vomag[i]*vomag[i];
19            if(pqv!= 1){
20                qinj[i] = -volt2*bii[i];
21            }
22            if(pqv != 2)
23                pinj[i] =  volt2*gii[i];
24        }
25
26    for (k=0; k<nlines; k++)
27        {
28            if (ibstat[k])
29                {
30                    i = ifrom[k];
31                    j = ito[k];
32                    volt = vomag[i]*vomag[j];
33                    delta_angle = voang[i] - voang[j] - shift_angle[k
    ];
34
35                    t_u(uij,tij,delta_angle,gij[k],bij[k],ratio[k]);
36                    t_u(uji,tji,-delta_angle,gij[k],bij[k],ratio[k]);
37
38                    if(pqv != 1){
39                        qinj[i] -= volt**uij;
40                        qinj[j] -= volt**uji;
41                    }
42                    if(pqv != 2){
43                        pinj[i] -= volt**tij;
44                        pinj[j] -= volt**tji;
```

```
45                      }
46                  }
47          }
48  }
49
50  __declspec(dllexport) void netpinj(int nbuses, int nlines, double *
    pinj, double *vomag, double *voang,
51              double *gii, double *bii, double *gij, double *bij,
    double *ratio, int *ifrom, int *ito, bool *ibstat)
52  {
53      int i, j, k;
54      double volt, volt2, delta_angle, sina, cosa;
55      double uij[1],tij[1], uji[1],tji[1];
56
57      /*Calculate the injection on all buses*/
58
59      for (i=0; i<nbuses; i++)
60          {
61              volt2 = vomag[i]*vomag[i];
62              pinj[i] =  volt2*gii[i];
63          }
64
65      for (k=0; k<nlines; k++)
66          {
67              if (ibstat[k])
68                  {
69                      i = ifrom[k];
70                      j = ito[k];
71                      volt = vomag[i]*vomag[j];
72                      delta_angle = voang[i] - voang[j];
73
74                      t_u(uij,tij,delta_angle,gij[k],bij[k],ratio[k]);
75                      t_u(uji,tji,-delta_angle,gij[k],bij[k],ratio[k]);
76
77                      pinj[i] -= volt**tij;
78                      pinj[j] -= volt**tji;
79                  }
80          }
81  }
82
83  __declspec(dllexport) void netqinj(int nbuses, int nlines, double *
    qinj, double *vomag, double *voang,
84              double *gii, double *bii, double *gij, double *bij,
    double *ratio, int *ifrom, int *ito, bool *ibstat)
85  {
86      int i, j, k;
87      double volt, volt2, delta_angle, sina, cosa;
88      double uij[1],tij[1], uji[1],tji[1];
89
90      /*Calculate the injection on all buses*/
```

```
91
92      for (i=0; i<nbuses; i++)
93          {
94              volt2 = vomag[i]*vomag[i];
95              qinj[i] = -volt2*bii[i];
96          }
97
98      for (k=0; k<nlines; k++)
99          {
100             if (ibstat[k])
101                 {
102                     i = ifrom[k];
103                     j = ito[k];
104                     volt = vomag[i]*vomag[j];
105                     delta_angle = voang[i] - voang[j];
106
107                     t_u(uij,tij,delta_angle,gij[k],bij[k],ratio[k]);
108                     t_u(uji,tji,-delta_angle,gij[k],bij[k],ratio[k]);
109
110                     qinj[i] -= volt**uij;
111                     qinj[j] -= volt**uji;
112                 }
113         }
114 }
```

B.17 sConstruct.py

```
#This file is used bu SCons to build the shared library "clibrary"
#Clibrary contains all the C-extesnions of Topflow

SharedLibrary(
    "clibrary",
    [
    "set_isa.c",
    "flatstart.c",
    "netinj.c",
    "mismat.c",
    "maxmism.c",
    "zerosp.c",
    "enforce_qlim.c",
    "set_rhs.c",
    "jacobi.c",
    "update_voltages.c",
    "coo_conv.c",
    "bmatrix.c",
    "hlmatrix.c",
    "selectver.c"
    ]
)
```

B.18 select_ver.c

```
#include<stdbool.h>
#include<math.h>
#include<string.h>

#define ABS(x) (((x) < 0) ? (-(x)) : ((x))) /* if loop: ABS(X) gives
    the absolute value of x*/

__declspec(dllexport) void selectver(double *xp,double *xq,double *
    xinv, double *bij, int nlines, char *bver){
     int k;
     //strcmp compare two strings
     if(strcmp(bver, "XX")){
         //STD version
         for(k = 0; k<nlines;k++){
             xp[k] = xinv[k];
             xq[k] = xinv[k];
         }
     }
     else if(strcmp(bver,"XB")){
         //XB-version (dual)
         for(k = 0; k<nlines;k++){
             xp[k] = xinv[k];
             xq[k] = ABS(bij[k]);
```

```
22          }
23      }
24      else if(strcmp(bver, "BX")){
25          //BX-version (primal)
26          for(k = 0; k<nlines;k++){
27              xp[k] = ABS(bij[k]);
28              xq[k] = xinv[k];
29          }
30      }
31      else if(strcmp(bver, "BB")){
32          //BB version
33          for(k = 0; k<nlines;k++){
34              xp[k] = ABS(bij[k]);
35              xq[k] = ABS(bij[k]);
36          }
37      }
38  }
```

B.19 set_rhs.c

```
#include <stdbool.h>


__declspec(dllexport) void set_rhs(int pqv, int nbuses, int *buscod,
    double *vomag, double *pinj, double *qinj, double *rhs){

     int iq, ip, ib;
     iq = ip = 0;

         for( ib = 0; ib <nbuses; ib++){
             if(pqv == 3){
                 //Right hand side of the Newton Rapshon equations
                 if(buscod[ib] != 3){
                     rhs[ip] = pinj[ib];
                     if(buscod[ib] != 2){
                         rhs[iq + nbuses-1] = qinj[ib];
                         iq += 1;
                     }
                     ip ++;
                 }
             }
             else if(pqv == 1){
                 //Right hand side of the active power equations in
    FDLF
                 rhs[ib] = pinj[ib]/vomag[ib];
             }
             else if(pqv == 2){
                 //Right hand side of the reactive power equations in
    FDLF
                 rhs[ib] = qinj[ib]/vomag[ib];
             }
         }
}
```

B.20 settings.py

```
1 #This file contains the Setting-class, which is used to specify
2 #the settings of the loadflow-functions (currently acsolve and
     decsolve)
3 class Settings:
4     def __init__(self, version = 'NR', flat_start = True, enf_qlim =
     False, max_it = 20, conv_tol = 0.000001, pqv = None, filename =
     None, save_path = None, save_verbose = 1, print_verbose = 1):
5         self.version = version
6         self.flat_start = flat_start
7         self.enf_qlim = enf_qlim
8         self.max_it = max_it
9         self.conv_tol = conv_tol
10        self.pqv = pqv
11        self.bver = 'XX' #Default
12        self.set_fdlf_ver(version, pqv) #sets self.bver and pqv
13        self.filename = filename
14        self.save_path = save_path
15        self.save_verbose = save_verbose
16        # 1: Save only the final result (Default).
17        # 2: Save the final reslt in one sheet, and the iteration-
     summaries in a second sheet.
18        self.print_verbose = print_verbose
19        # 0: Do NOT print anything to screen.
20        # 1: Print the final result to screen: Convergence/Divergence
     . (Default)
21        # 2: Print a more verbose final result-summary to the screen.
22        # 3: Same as 2: + print a summery of each iteration
23
24
25    def print_settings(self):
26        print('\nVersion:',self.version,
27        '\nflat_start:',self.flat_start,
28        '\nEnforce reactive limits:', self.enf_qlim,
29        '\nmaximum iterations:', self.max_it,
30        '\nconvergence tolerance',self.conv_tol)
31
32    def set_fdlf_ver(self, version, pqv):
33        if(version in ['FDXB', 'FDBX', 'FDXX', 'FDBB']):
34            self.bver = version[2:]
35            if(pqv is None):
36                #if the iteration-version is not specified, the
     default values for the variuos FDLF-versions are used
37                self.pqv = 2 if self.version == 'FDXB' else 1
38            elif(pqv in [1,2]):
39                self.pqv = pqv
40            else:
41                print('unvalid value for pqv')
42
```

```
43     def set_var(self, name, value):
44         #set the instance variables
45         setattr(self,name, value)
46         if(name in ['version', 'pqv']):
47             #update bver and pqv
48             self.set_fdlf_ver(self.version, self.pqv)
```

B.21 topflow.h

```
1 #ifndef HEADER_nettanalyse /*to prevent double declaration of
     identifiers*/
2 #define HEADER_nettanalyse
3
4 #define PQ_BUS 1
5 #define PV_BUS 2
6 #define SWING_BUS  3
7 #define DISCONNECTED_BUS 4
8
9
10 #define ABS(x) (((x) < 0) ? (-(x)) : ((x))) /* if loop: ABS(X) gives
     the absolute value of x*/
11 #define CONNECTED(i, con) (con[i] < 0 ? i : con[i])
12
13 #endif
```

B.22 update_voltages.py

```
#include <stdbool.h>

__declspec(dllexport) void update_voltages(int pqv, int nbuses, int *
    buscod, double *vomag, double *voang, double *correction){
    int ip, iq, ib;

    ip = iq = 0;

    for( ib = 0; ib <nbuses; ib++){
        if( pqv == 3){
            //Update for Newton Rapshon algorithm
            if(buscod[ib] != 3){
                voang[ib] += correction[ip];
                ip ++;
                if(buscod[ib] != 2){
                    vomag[ib] += correction[iq + nbuses-1];
                    iq ++;
                }
            }
        }
        else if(pqv == 1){
            //Update voltage anges (Active power eq fro FDLD)
            if(buscod[ib] != 3){
                voang[ib] += correction[ib];
            }
        }
        else if(pqv == 2){
            //Update voltage magnitudes (Reactive power eq for FDLF)
            if(buscod[ib] != 3 && buscod[ib] != 2){
                vomag[ib] += correction[ib];
            }
        }
    }
}
```

B.23 zerosp.c

```
#include <stdbool.h>
#include <math.h>

__declspec(dllexport) void zerosp(int nlines, int *ifrom, int *ito,
    bool *ibstat, double *xinv, double *xinj){
    int il, ifr, itr;

    for(il = 0; il < nlines; il++){
        if(fabs(xinv[il] > 9000 && ibstat[il] == true)){
            ifr = ifrom[il];
            itr = ito[il];
```

```
11              if(fabs(xinj[ifr]) > fabs(xinj[itr])){
12
13                  xinj[ifr]=xinj[ifr]+xinj[itr];
14                  xinj[itr]=0.0;
15              }
16              else{
17                  xinj[itr]=xinj[itr]+xinj[ifr];
18                  xinj[ifr]=0.0;
19              }
20          }
21      }
22  }
```

C Tests

C.1 pandapower_delay.py

```
1 import timeit
2
3 #Measure the delay of the numba-compiler in pandapower
4
5 mysetup = (
6
7 '''import pandapower as pp
8 import pandapower.networks as pn
9 net = pn.case14()
10 '''
11 )
12
13 numba_delay = timeit.timeit( setup = mysetup, stmt = 'pp.runpp(net)',
      number = 1)
14 runtime_case14 = timeit.timeit( setup = mysetup, stmt = 'pp.runpp(net
    )', number = 100)/100
15
16 numba_delay = numba_delay - runtime_case14
17
18 print('Numba delay:', numba_delay)
```

C.2 reliability_acsolve.py

```
import sys
sys.path.append("..")
import topflow as tf
import numpy as np
import math
import pandapower as pp
import pandapower.networks as pn

#Reliability test
#Compare topflow results to pandapower and pypower

#Initialize topflow cases
case4gs = tf.example_case('case4gs')
case14 = tf.example_case('case14')
case30 = tf.example_case('case30')
case118 = tf.example_case('case118')
case300 = tf.example_case('case300')
case1354  = tf.example_case('case1354pegase')
case2869  = tf.example_case('case2869pegase')
case9241  = tf.example_case('case9241pegase')

#Solve topflow cases
result4gs = tf.loadflow(case4gs, conv_tol = 0.00000001, print_verbose
    = 0)
result14 = tf.loadflow(case14,  conv_tol = 0.00000001, print_verbose
    = 0)
result30 = tf.loadflow(case30, conv_tol = 0.00000001, print_verbose =
    0)
result118 = tf.loadflow(case118,  conv_tol = 0.00000001,
    print_verbose = 0)
result300 = tf.loadflow(case300, conv_tol = 0.00000001, print_verbose
    = 0)
result1354  = tf.loadflow(case1354 , conv_tol = 0.00000001,
    print_verbose = 0)
result2869  = tf.loadflow(case2869 , conv_tol = 0.00000001,
    print_verbose = 0)
result9241  = tf.loadflow(case9241 , conv_tol = 0.00000001,
    print_verbose = 0)

#initialize pandapower cases
net4gs = pn.case4gs()
net14 = pn.case14()
net30 = pn.case30()
net118 = pn.case118()
net300 = pn.case300()
net1354  = pn.case1354pegase()
net2869  = pn.case2869pegase()
net9241  = pn.case9241pegase()
```

```
41
42 #solve pandapower cases
43 pp.runpp(net4gs)
44 pp.runpp(net14)
45 pp.runpp(net30)
46 pp.runpp(net118)
47 pp.runpp(net300)
48 pp.runpp(net1354)
49 pp.runpp(net2869)
50 pp.runpp(net9241)
51
52
53 #compare case4gs
54 panda_vm_dev = 0
55 panda_va_dev = 0
56 worst_panda_vm_bus = 0
57 worst_panda_va_bus = 0
58 pyp_vm_dev = 0
59 pyp_va_dev = 0
60 worst_pyp_vm_bus = 0
61 worst_pyp_va_bus = 0
62
63 for i in range(0, result4gs.vomag.size):
64     ext = case4gs.int2ext(i)
65     if abs(result4gs.get('bus', ext, 'vm')-net4gs.res_bus.vm_pu[i])>
    abs(panda_vm_dev):
66         panda_vm_dev = result4gs.get('bus', ext, 'vm')-net4gs.res_bus
    .vm_pu[i]
67         worst_panda_vm_bus =ext
68     if abs(result4gs.get('bus', ext, 'va')-net4gs.res_bus.va_degree[i
    ])> abs(panda_va_dev):
69         panda_va_dev = result4gs.get('bus', ext, 'va')-net4gs.res_bus
    .va_degree[i]
70         worst_panda_va_bus = ext
71     if abs(result4gs.vomag[i]-case4gs.vomag[i])> abs(pyp_vm_dev):
72         pyp_vm_dev = result4gs.vomag[i]-case4gs.vomag[i]
73         worst_pyp_vm_bus = case4gs.int2ext(i)
74     if abs(result4gs.voang[i]-case4gs.voang[i])> abs(pyp_va_dev):
75         pyp_va_dev = result4gs.voang[i]-case4gs.voang[i]
76         worst_pyp_va_bus = case4gs.int2ext(i)
77
78 print('\nPandaPower:')
79 print('Worst VM devition case4gs:', panda_vm_dev, 'pu, at bus:',
    worst_panda_vm_bus)
80 print('Worst VA devition case4gs:', panda_va_dev, 'deg, at bus:',
    worst_panda_va_bus)
81 print('PyPower:')
82 print('Worst VM devition case4gs:', pyp_vm_dev, 'pu, at bus:',
    worst_pyp_vm_bus)
83 print('Worst VA devition case4gs:', pyp_va_dev, 'deg, at bus:',
```

```
    worst_pyp_va_bus)
84
85
86 #compare case14
87 panda_vm_dev = 0
88 panda_va_dev = 0
89 worst_panda_vm_bus = 0
90 worst_panda_va_bus = 0
91 pyp_vm_dev = 0
92 pyp_va_dev = 0
93 worst_pyp_vm_bus = 0
94 worst_pyp_va_bus = 0
95
96 for i in range(0, result14.vomag.size):
97     ext = case14.int2ext(i)
98     if abs(result14.get('bus', ext, 'vm')-net14.res_bus.vm_pu[i])>
    abs(panda_vm_dev):
99         panda_vm_dev = result14.get('bus', ext, 'vm')-net14.res_bus.
    vm_pu[i]
100         worst_panda_vm_bus =ext
101     if abs(result14.get('bus', ext, 'va')-net14.res_bus.va_degree[i])
    > abs(panda_va_dev):
102         panda_va_dev = result14.get('bus', ext, 'va')-net14.res_bus.
    va_degree[i]
103         worst_panda_va_bus = ext
104     if abs(result14.get('bus',ext,'vm')-case14.get('bus', ext, 'vm'))
    > abs(pyp_vm_dev):
105         pyp_vm_dev = result14.get('bus',ext,'vm')-case14.get('bus',
    ext, 'vm')
106         worst_pyp_vm_bus = ext
107     if abs(result14.get('bus',ext,'va')-case14.get('bus', ext, 'va'))
    > abs(pyp_va_dev):
108         pyp_va_dev = result14.get('bus',ext,'va')-case14.get('bus',
    ext, 'va')
109         worst_pyp_va_bus = ext
110
111 print('\nPandaPower:')
112 print('Worst VM devition case14:', panda_vm_dev, 'pu, at bus:',
    worst_panda_vm_bus)
113 print('Worst VA devition case14:', panda_va_dev, 'deg, at bus:',
    worst_panda_va_bus)
114 print('PyPower:')
115 print('Worst VM devition case14:', pyp_vm_dev, 'pu, at bus:',
    worst_pyp_vm_bus)
116 print('Worst VA devition case14:', pyp_va_dev, 'deg, at bus:',
    worst_pyp_va_bus)
117
118
119 #compare case30
120 panda_vm_dev = 0
```

```
panda_va_dev = 0
worst_panda_vm_bus = 0
worst_panda_va_bus = 0
pyp_vm_dev = 0
pyp_va_dev = 0
worst_pyp_vm_bus = 0
worst_pyp_va_bus = 0

for i in range(0, result30.vomag.size):
    ext = case30.int2ext(i)
    if abs(result30.get('bus', ext, 'vm')-net30.res_bus.vm_pu[i])>abs(panda_vm_dev):
        panda_vm_dev = result30.get('bus', ext, 'vm')-net30.res_bus.vm_pu[i]
        worst_panda_vm_bus =ext
    if abs(result30.get('bus', ext, 'va')-net30.res_bus.va_degree[i]) > abs(panda_va_dev):
        panda_va_dev = result30.get('bus', ext, 'va')-net30.res_bus.va_degree[i]
        worst_panda_va_bus = ext
    if abs(result30.get('bus',ext,'vm')-case30.get('bus', ext, 'vm')) > abs(pyp_vm_dev):
        pyp_vm_dev = result30.get('bus',ext,'vm')-case30.get('bus', ext, 'vm')
        worst_pyp_vm_bus = ext
    if abs(result30.get('bus',ext,'va')-case30.get('bus', ext, 'va')) > abs(pyp_va_dev):
        pyp_va_dev = result30.get('bus',ext,'va')-case30.get('bus', ext, 'va')
        worst_pyp_va_bus = ext

print('\nPandaPower:')
print('Worst VM devition case30:', panda_vm_dev, 'pu, at bus:', worst_panda_vm_bus)
print('Worst VA devition case30:', panda_va_dev, 'deg, at bus:', worst_panda_va_bus)
print('PyPower:')
print('Worst VM devition case30:', pyp_vm_dev, 'pu, at bus:', worst_pyp_vm_bus)
print('Worst VA devition case30:', pyp_va_dev, 'deg, at bus:', worst_pyp_va_bus)


#compare case118
panda_vm_dev = 0
panda_va_dev = 0
worst_panda_vm_bus = 0
worst_panda_va_bus = 0
pyp_vm_dev = 0
pyp_va_dev = 0
```

```
159 worst_pyp_vm_bus = 0
160 worst_pyp_va_bus = 0
161
162 for i in range(0, result118.vomag.size):
163     ext = case118.int2ext(i)
164     #pp_int = net118.bus.name[ext]
165     if abs(result118.get('bus', ext, 'vm')-net118.res_bus.vm_pu[i])>
    abs(panda_vm_dev):
166         panda_vm_dev = result118.get('bus', ext, 'vm')-net118.res_bus
    .vm_pu[i]
167         worst_panda_vm_bus =ext
168     if abs(result118.get('bus', ext, 'va')-net118.res_bus.va_degree[i
    ])> abs(panda_va_dev):
169         panda_va_dev = result118.get('bus', ext, 'va')-net118.res_bus
    .va_degree[i]
170         worst_panda_va_bus = ext
171     if abs(result118.get('bus',ext,'vm')-case118.get('bus', ext, 'vm'
    ))> abs(pyp_vm_dev):
172         pyp_vm_dev = result118.get('bus',ext,'vm')-case118.get('bus',
     ext, 'vm')
173         worst_pyp_vm_bus = ext
174     if abs(result118.get('bus',ext,'va')-case118.get('bus', ext, 'va'
    ))> abs(pyp_va_dev):
175         pyp_va_dev = result118.get('bus',ext,'va')-case118.get('bus',
     ext, 'va')
176         worst_pyp_va_bus = ext
177
178 print('\nWorst VM devition case118:', panda_vm_dev, 'pu, at bus:',
    worst_panda_vm_bus)
179 print('Worst VA devition case118:', panda_va_dev, 'deg, at bus:',
    worst_panda_va_bus)
180 print('PyPower:')
181 print('Worst VM devition case118:', pyp_vm_dev, 'pu, at bus:',
    worst_pyp_vm_bus)
182 print('Worst VA devition case118:', pyp_va_dev, 'deg, at bus:',
    worst_pyp_va_bus)
183
184
185 #compare case300
186 panda_vm_dev = 0
187 panda_va_dev = 0
188 worst_panda_vm_bus = 0
189 worst_panda_va_bus = 0
190 pyp_vm_dev = 0
191 pyp_va_dev = 0
192 worst_pyp_vm_bus = 0
193 worst_pyp_va_bus = 0
194
195 for i in range(0, result300.vomag.size):
196     ext = case300.int2ext(i)
```

```
197     if abs(result300.get('bus', ext, 'vm')-net300.res_bus.vm_pu[i])>
        abs(panda_vm_dev):
198         panda_vm_dev = result300.get('bus', ext, 'vm')-net300.res_bus
        .vm_pu[i]
199         worst_panda_vm_bus =ext
200     if abs(result300.get('bus', ext, 'va')-net300.res_bus.va_degree[i
        ])> abs(panda_va_dev):
201         panda_va_dev = result300.get('bus', ext, 'va')-net300.res_bus
        .va_degree[i]
202         worst_panda_va_bus = ext
203     if abs(result300.get('bus',ext,'vm')-case300.get('bus', ext, 'vm'
        ))> abs(pyp_vm_dev):
204         pyp_vm_dev = result300.get('bus',ext,'vm')-case300.get('bus',
         ext, 'vm')
205         worst_pyp_vm_bus = ext
206     if abs(result300.get('bus',ext,'va')-case300.get('bus', ext, 'va'
        ))> abs(pyp_va_dev):
207         pyp_va_dev = result300.get('bus',ext,'va')-case300.get('bus',
         ext, 'va')
208         worst_pyp_va_bus = ext
209
210 print('\nPandaPower:')
211 print('Worst VM devition case300:', panda_vm_dev, 'pu, at bus:',
        worst_panda_vm_bus)
212 print('Worst VA devition case300:', panda_va_dev, 'deg, at bus:',
        worst_panda_va_bus)
213 print('PyPower:')
214 print('Worst VM devition case300:', pyp_vm_dev, 'pu, at bus:',
        worst_pyp_vm_bus)
215 print('Worst VA devition case300:', pyp_va_dev, 'deg, at bus:',
        worst_pyp_va_bus)
216
217
218 #compare case1354
219 panda_vm_dev = 0
220 panda_va_dev = 0
221 worst_panda_vm_bus = 0
222 worst_panda_va_bus = 0
223 pyp_vm_dev = 0
224 pyp_va_dev = 0
225 worst_pyp_vm_bus = 0
226 worst_pyp_va_bus = 0
227
228 for i in range(0, result1354.vomag.size):
229     ext = case1354.int2ext(i)
230     if abs(result1354.get('bus', ext, 'vm')-net1354.res_bus.vm_pu[i])
        > abs(panda_vm_dev):
231         panda_vm_dev = result1354.get('bus', ext, 'vm')-net1354.
        res_bus.vm_pu[i]
232         worst_panda_vm_bus =ext
```

```
233     if abs(result1354.get('bus', ext, 'va')-net1354.res_bus.va_degree
        [i])> abs(panda_va_dev):
234         panda_va_dev = result1354.get('bus', ext, 'va')-net1354.
        res_bus.va_degree[i]
235         worst_panda_va_bus = ext
236     if abs(result1354.get('bus',ext,'vm')-case1354.get('bus', ext, '
        vm'))> abs(pyp_vm_dev):
237         pyp_vm_dev = result1354.get('bus',ext,'vm')-case1354.get('bus
        ', ext, 'vm')
238         worst_pyp_vm_bus = ext
239     if abs(result1354.get('bus',ext,'va')-case1354.get('bus', ext, '
        va'))> abs(pyp_va_dev):
240         pyp_va_dev = result1354.get('bus',ext,'va')-case1354.get('bus
        ', ext, 'va')
241         worst_pyp_va_bus = ext
242
243 print('\nWorst VM devition case1354 :', panda_vm_dev, 'pu, at bus:',
        worst_panda_vm_bus)
244 print('Worst VA devition case1354 :', panda_va_dev, 'deg, at bus:',
        worst_panda_va_bus)
245 print('PyPower:')
246 print('Worst VM devition case1354:', pyp_vm_dev, 'pu, at bus:',
        worst_pyp_vm_bus)
247 print('Worst VA devition case1354:', pyp_va_dev, 'deg, at bus:',
        worst_pyp_va_bus)
248
249 #compare case2869
250 panda_vm_dev = 0
251 panda_va_dev = 0
252 worst_panda_vm_bus = 0
253 worst_panda_va_bus = 0
254 pyp_vm_dev = 0
255 pyp_va_dev = 0
256 worst_pyp_vm_bus = 0
257 worst_pyp_va_bus = 0
258
259 for i in range(0, result2869.vomag.size):
260     ext = case2869.int2ext(i)
261
262     if abs(result2869.get('bus', ext, 'vm')-net2869.res_bus.vm_pu[i])
        > panda_vm_dev:
263         panda_vm_dev = result2869.get('bus', ext, 'vm')-net2869.
        res_bus.vm_pu[i]
264         worst_panda_vm_bus =ext
265     if abs(result2869.get('bus', ext, 'va')-net2869.res_bus.va_degree
        [i])> panda_va_dev:
266         panda_va_dev = result2869.get('bus', ext, 'va')-net2869.
        res_bus.va_degree[i]
267         worst_panda_va_bus = ext
268     if abs(result2869.get('bus',ext,'vm')-case2869.get('bus', ext, '
```

```
    vm'))> abs(pyp_vm_dev):
269         pyp_vm_dev = result2869.get('bus',ext,'vm')-case2869.get('bus
    ', ext, 'vm')
270         worst_pyp_vm_bus = ext
271      if abs(result2869.get('bus',ext,'va')-case2869.get('bus', ext, '
    va'))> abs(pyp_va_dev):
272         pyp_va_dev = result2869.get('bus',ext,'va')-case2869.get('bus
    ', ext, 'va')
273         worst_pyp_va_bus = ext
274
275 print('\nWorst VM devition case2869 :', panda_vm_dev, 'pu, at bus:',
    worst_panda_vm_bus)
276 print('Worst VA devition case2869 :', panda_va_dev, 'deg, at bus:',
    worst_panda_va_bus)
277 print('PyPower:')
278 print('Worst VM devition case2869:', pyp_vm_dev, 'pu, at bus:',
    worst_pyp_vm_bus)
279 print('Worst VA devition case2869:', pyp_va_dev, 'deg, at bus:',
    worst_pyp_va_bus)
280
281 #compare case9241
282 panda_vm_dev = 0
283 panda_va_dev = 0
284 worst_panda_vm_bus = 0
285 worst_panda_va_bus = 0
286 pyp_vm_dev = 0
287 pyp_va_dev = 0
288 worst_pyp_vm_bus = 0
289 worst_pyp_va_bus = 0
290
291 for i in range(0, result9241.vomag.size):
292     ext = case9241.int2ext(i)
293     if abs(result9241.get('bus', ext, 'vm')-net9241.res_bus.vm_pu[i])
    > panda_vm_dev:
294         panda_vm_dev = result9241.get('bus', ext, 'vm')-net9241.
    res_bus.vm_pu[i]
295         worst_panda_vm_bus =ext
296     if abs(result9241.get('bus', ext, 'va')-net9241.res_bus.va_degree
    [i])> panda_va_dev:
297         panda_va_dev = result9241.get('bus', ext, 'va')-net9241.
    res_bus.va_degree[i]
298         worst_panda_va_bus = ext
299     if abs(result9241.get('bus',ext,'vm')-case9241.get('bus', ext, '
    vm'))> abs(pyp_vm_dev):
300         pyp_vm_dev = result9241.get('bus',ext,'vm')-case9241.get('bus
    ', ext, 'vm')
301         worst_pyp_vm_bus = ext
302     if abs(result9241.get('bus',ext,'va')-case9241.get('bus', ext, '
    va'))> abs(pyp_va_dev):
303         pyp_va_dev = result9241.get('bus',ext,'va')-case9241.get('bus
```

```
    ', ext, 'va')
304         worst_pyp_va_bus = ext
305
306 print('\nWorst VM devition case9241 :', panda_vm_dev, 'pu, at bus:',
    worst_panda_vm_bus)
307 print('Worst VA devition case9241:', panda_va_dev, 'deg, at bus:',
    worst_panda_va_bus)
308 print('PyPower:')
309 print('Worst VM devition case9241:', pyp_vm_dev, 'pu, at bus:',
    worst_pyp_vm_bus)
310 print('Worst VA devition case9241:', pyp_va_dev, 'deg, at bus:',
    worst_pyp_va_bus)
```

C.3 reliability_decsolve.py

```
1 import sys
2 sys.path.append("..")
3 import topflow as tf
4
5 #this test compares decsolve with acsolve. acsolve has been tested
     against other programs and should give reliable results
6 #Initialize topflow cases
7
8
9 for i in range(0,7):
10     if(i == 0):
11         print('\n##### case4gs #####')
12
13         #initialize the case
14         case = tf.example_case('case4gs')
15     elif(i == 1):
16         print('\n##### case14 #####')
17
18         #initialize the case
19         case = tf.example_case('case14')
20     elif(i == 2):
21         print('\n##### case30 #####')
22
23         #initialize the case
24         case = tf.example_case('case30')
25     elif(i == 3):
26         print('\n##### case118 #####')
27
28         #initialize the case
29         case = tf.example_case('case118')
30     elif(i == 4):
31         print('\n##### case300 #####')
32
33         #initialize the case
34         case = tf.example_case('case300')
35     elif(i == 5):
36         print('\n##### 1354pegase #####')
37
38         #initialize the case
39         case = tf.example_case('case1354pegase')
40     elif(i == 6):
41         print('\n##### 2869pegase #####')
42
43         #initialize the case
44         case = tf.example_case('case2869pegase')
45     elif(i == 7):
46         print('\n##### case9241pegase #####')
47
```

```
48          #initialize the case
49          case = tf.example_case('case9241pegase')
50
51      #initialize the worst deviations
52      worst_vm_fdxx = 0
53      worst_va_fdxx = 0
54
55      worst_vm_fdbx = 0
56      worst_va_fdbx = 0
57
58      worst_vm_fdxb = 0
59      worst_va_fdxb = 0
60
61      #run the load-flows
62      resnr = tf.loadflow(case, version = 'NR', print_verbose = 0)
63      resxx = tf.loadflow(case, version = 'FDXX', print_verbose = 0)
64      resbx = tf.loadflow(case, version = 'FDBX', print_verbose = 0)
65      resxb = tf.loadflow(case, version = 'FDXB', print_verbose = 0)
66
67      #Compare the results, if the loadflow()-function returns 0 it
    means the test diverged
68
69      for i in range(0, resnr.vomag.size):
70          #FDXX:
71          if(resxx == 0):
72              break
73          if abs(resnr.vomag[i]-resxx.vomag[i])> abs(worst_vm_fdxx):
74              worst_vm_fdxx = abs(resnr.vomag[i]-resxx.vomag[i])
75          if abs(resnr.voang[i]-resxx.voang[i])> abs(worst_va_fdxx):
76              worst_va_fdxx = abs(resnr.voang[i]-resxx.voang[i])
77          #FDBX:
78          if(resbx == 0):
79              break
80          if abs(resnr.vomag[i]-resbx.vomag[i])> abs(worst_vm_fdbx):
81              worst_vm_fdbx = abs(resnr.vomag[i]-resbx.vomag[i])
82          if abs(resnr.voang[i]-resbx.voang[i])> abs(worst_va_fdbx):
83              worst_va_fdbx = abs(resnr.voang[i]-resbx.voang[i])
84          #FDXB
85          if(resxb == 0):
86              break
87          if abs(resnr.vomag[i]-resxb.vomag[i])> abs(worst_vm_fdxb):
88              worst_vm_fdxb = abs(resnr.vomag[i]-resxb.vomag[i])
89          if abs(resnr.voang[i]-resxb.voang[i])> abs(worst_va_fdxb):
90              worst_va_fdxb = abs(resnr.voang[i]-resxb.voang[i])
91
92      #Print the result:
93
94      #FDXX
95      if(resxx == 0):
96          print('\nFDXX: DIVERGENCE')
```

```
97      else:
98          print('\nFDXX: CONVERGENCE: number of iterations:', resxx.
        icount)
99          print('Worst VM deviation:', worst_vm_fdxx)
100         print('Worst VA deviation:', worst_va_fdxx)
101     #FDBX
102     if(resbx == 0):
103         print('\nFDBX: DIVERGENCE')
104     else:
105         print('\nFDBX: CONVERGENCE: number of iterations:', resbx.
        icount)
106         print('Worst VM deviation:', worst_vm_fdbx)
107         print('Worst VA deviation:', worst_va_fdbx)
108     #FDXB
109     if(resxb == 0):
110         print('\nFDXB: DIVERGENCE')
111     else:
112         print('\nFDXB: CONVERGENCE: number of iterations:', resxb.
        icount)
113         print('Worst VM deviation:', worst_vm_fdxb)
114         print('Worst VA deviation:', worst_va_fdxb)
```

C.4 speed_acsolve.py

```
1
2 import timeit
3 from pypower.api import case14 as pypcase14
4 from pypower.api import case30 as pypcase30
5 from pypower.api import runpf, ppoption
6
7 #This tests compares the speed of running a Newton-Rapshon loadflow
     with Topflow
8 #With the speed of pandapower and pypower
9
10 #Measure the execution time of topflow
11 mysetup_tf = (
12     '''
13 import sys
14 sys.path.append("..")
15 import topflow as tf
16
17 '''
18 )
19
20 tf_runtime14 = timeit.timeit(setup = mysetup_tf +
21 '''
22 case = tf.example_case("case14")
23 ''',
24 stmt = 'tf.loadflow(case, conv_tol = 0.00000001, print_verbose = 0)',
25 number = 100)/100
26
27 tf_runtime30 = timeit.timeit(setup = mysetup_tf +
28 '''
29 case = tf.example_case("case30")
30 ''',
31 stmt = 'tf.loadflow(case, conv_tol = 0.00000001, print_verbose = 0)',
32 number = 100)/100
33
34 tf_runtime1354 = timeit.timeit(setup = mysetup_tf +
35 '''
36 case = tf.example_case("case1354pegase")
37 ''',
38 stmt = 'tf.loadflow(case, conv_tol = 0.00000001, print_verbose = 0)',
39 number = 100)/100
40
41 tf_runtime2869 = timeit.timeit(setup = mysetup_tf +
42 '''
43 case = tf.example_case("case2869pegase")
44 ''',
45 stmt = 'tf.loadflow(case, conv_tol = 0.00000001, print_verbose = 0)',
46 number = 100)/100
47
```

```
48 tf_runtime9241 = timeit.timeit(setup = mysetup_tf +
49 '''
50 case = tf.example_case("case9241pegase")
51 ''',
52 stmt = 'tf.loadflow(case, conv_tol = 0.00000001, print_verbose = 0)',
53 number = 100)/100
54
55 #Measure the exectution time of pypower
56 mysetup_pyp14 = (
57 '''
58 from pypower.api import runpf, ppoption
59 from pypower.api import case14
60 ppopt = ppoption(VERBOSE = 0, OUT_ALL = 0)
61 case14 = case14()
62 ''')
63
64
65 pyp_runtime14 = timeit.timeit(setup = mysetup_pyp14, stmt = 'ppr =
      runpf(case14, ppopt)[0]', number = 100)/100
66
67
68 mysetup_pyp30 = (
69 '''
70 from pypower.api import runpf, ppoption
71 from pypower.api import case30
72 ppopt = ppoption(VERBOSE = 0, OUT_ALL = 0)
73 case30 = case30()
74 '''
75 )
76
77 pyp_runtime30 = timeit.timeit(setup = mysetup_pyp30, stmt =  'ppr =
      runpf(case30, ppopt)[0]', number = 100)/100
78
79
80 #The cases which are not available in pypower are converted
81 #to pypower-cases from pandapower by using the pandapower.converter
82 mysetup_pyp1354 = (
83 '''
84 from pypower.api import runpf, ppoption
85 import pandapower.converter as pc
86 import pandapower.networks as pn
87 ppopt = ppoption(VERBOSE = 0, OUT_ALL = 0)
88 case1354 = pc.to_ppc(pn.case1354pegase())
89 '''
90 )
91
92 pyp_runtime1354 = timeit.timeit(setup = mysetup_pyp1354, stmt = 'ppr
      = runpf(case1354, ppopt)[0]', number = 100)/100
93
94 mysetup_pyp2869 = (
```

```
95 '''
96 from pypower.api import runpf, ppoption
97 import pandapower.converter as pc
98 import pandapower.networks as pn
99 ppopt = ppoption(VERBOSE = 0, OUT_ALL = 0)
100 case2869 = pc.to_ppc(pn.case2869pegase())
101 '''
102 )
103
104 pyp_runtime2869 = timeit.timeit(setup = mysetup_pyp2869, stmt ='ppr =
      runpf(case2869, ppopt)[0]', number = 100)/100
105
106 mysetup_pyp9241 = (
107 '''
108 from pypower.api import runpf, ppoption
109 import pandapower.converter as pc
110 import pandapower.networks as pn
111 ppopt = ppoption(VERBOSE = 0, OUT_ALL = 0)
112 case9241 = pc.to_ppc(pn.case9241pegase())
113 '''
114 )
115
116 pyp_runtime9241 = timeit.timeit(setup = mysetup_pyp9241, stmt ='ppr =
      runpf(case9241, ppopt)[0]', number = 100)/100
117
118
119 #Measure the execution time of pandapower
120 mysetup_pp = (
121     '''
122 import pandapower as pp
123 import pandapower.networks as pn
124 net = pn.case14()
125 pp.runpp(net)
126     '''
127 )
128
129 pp_runtime14 = timeit.timeit(setup = mysetup_pp +
130 '''
131 net = pn.case14()
132 ''',
133 stmt = 'pp.runpp(net)',
134 number = 100)/100
135
136 pp_runtime30 = timeit.timeit(setup = mysetup_pp +
137 '''
138 net = pn.case30()
139 ''',
140 stmt = 'pp.runpp(net)',
141 number = 100)/100
142
```

```
pp_runtime1354 = timeit.timeit(setup = mysetup_pp +
'''
net = pn.case1354pegase()
''',
stmt = 'pp.runpp(net)',
number = 100)/100

pp_runtime2869 = timeit.timeit(setup = mysetup_pp +
'''
net = pn.case2869pegase()
''',
stmt = 'pp.runpp(net)',
number = 100)/100

pp_runtime9241 = timeit.timeit(setup = mysetup_pp +
'''
net = pn.case9241pegase()
''',
stmt = 'pp.runpp(net)',
number = 100)/100


#Print the result to screen
print('\nRuntimes on case14:')
print('Topflow:', tf_runtime14)
print('pandapower:', pp_runtime14)
print('pypower:', pyp_runtime14)

print('\nRuntimes on case30:')
print('Topflow:', tf_runtime30)
print('pandapower:', pp_runtime30)
print('pypower:', pyp_runtime30)

print('\nRuntime on case1354pegase:')
print('Topflow:', tf_runtime1354)
print('pandapower:', pp_runtime1354)
print('pypower:', pyp_runtime1354)

print('\nRuntime on case2869pegase:')
print('Topflow:', tf_runtime2869)
print('pandapower:', pp_runtime2869)
print('pypower:', pyp_runtime2869)

print('\nRuntime on case9241pegase:')
print('Topflow:', tf_runtime9241)
print('pandapower:', pp_runtime9241)
print('pypower:', pyp_runtime9241)
```

C.5 test_acsolve_integration.py

```
1 import sys
2 sys.path.append("..")
3 import topflow as tf
4 import topflow.acsolve_wrapper as ac
5 import numpy as np
6 from numpy.testing import assert_array_equal
7 from numpy.testing import assert_array_almost_equal
8 import pytest
9 import math
10
11 #This is a integration-test for acsolve
12
13 @pytest.fixture
14 def case3():
15     case3 = tf.Case()
16     #Initialize the Case-object with the data of case14
17     case3.icount == 0
18     case3.convergence == False
19     case3.sbase == 100
20     # Bus data
21     case3.nbuses = 3  # number of buses
22     case3.slackbusnr= 2 #the internal number of the slacbus
23     case3.numbus = np.array([0,1,2], dtype=np.int32)
24     case3.buscod = np.array([1, 1, 3], dtype=np.int32)  # bus type.
    1=PQ, 2=PV, 3=Slack, 4= disconnected
25     case3.pload = np.array([1.0, 0.5, 0.0], dtype=np.double)  # pload
    [nbuses]: active load vector (fixed)
26     case3.qload = np.array([0.5, 0.5, 0.0], dtype=np.double)  # qload
    [nbuses]: reactive load vector (fixed)
27     case3.pinj = np.zeros(case3.nbuses, dtype=np.double)  #
    calculated active power injection for all buses
28     case3.qinj = np.zeros( case3.nbuses, dtype=np.double)  #
    calculated reactive power injection for all buses
29     case3.vomag = np.ones(case3.nbuses, dtype=np.double)  # voltage
    magnetudes at all buses
30     case3.voang = np.zeros(case3.nbuses, dtype=np.double)  # voltage
    angles at all buses
31     case3.gs = np.ones(case3.nbuses)
32     case3.bs = np.ones(case3.nbuses)
33
34     # Generator data
35     case3.ngens = 1
36     case3.genbus = np.array([-1, -1, 0], dtype=np.int32)  # genbus[
    nbuses] if a generator: genbus[k]= generator nr. If not: genbus[k
    ]=-1
37     case3.vgbus = np.ones(case3.ngens, dtype=np.int32)
38     case3.pgen = np.array([0.0], dtype=np.double)  # pgen[ngen]:
    active power generated at generator nr
```

```
39      case3.qgen = np.array([0.0], dtype=np.double)  # qgen[ngen]:
    reactive power generated at generator nr
40      case3.qmax = np.array([9999])
41      case3.qmin = np.array([-999])
42
43      # Line data
44      case3.nlines = 3  # number of lines
45      case3.gii = np.array([5.17647059, 3.17647059, 6.0], dtype=np.
    double)  # real part of the diagonal element in the ybus
46      case3.bii = np.array([-12.70588235, -10.70588235, -14.0], dtype=
    np.double)  # imag part of the diagonal element in the ybus
47      case3.gij = np.array([1.17647059, 4.0, 2.0], dtype=np.double)  #
    real part of the off-diagonal element in the ybus
48      case3.bij = np.array([-4.70588235, -8.0, -6.0], dtype=np.double)
      # imag part of the off-diagonal element in the ybus
49      case3.gi = np.zeros(case3.nlines, dtype = np.double)
50      case3.bi = np.zeros(case3.nlines, dtype = np.double)
51      case3.gj = np.zeros(case3.nlines, dtype = np.double)
52      case3.bj = np.zeros(case3.nlines, dtype = np.double)
53      case3.xinv = np.array([1 / 0.2, 1 / 0.1, 1 / 0.15])  # inverse of
      the line reactances
54      case3.ratio = np.array([1.0, 1.0, 1.0], dtype=np.double)  #
    transformer ratio for all the lines
55      case3.shift_angle = np.zeros(3) #shift angles of the transformers
56      case3.ifrom = np.array([0, 0, 1], dtype=np.int32)  # from-bus for
      all the lines
57      case3.ito = np.array([1, 2, 2], dtype=np.int32)  # to-bus for all
      the lines
58      case3.ibstat = np.array([True, True, True], dtype=np.bool)  #
    status for all the lines. True= in service
59
60      return case3
61
62 @pytest.fixture
63 def case14():
64      #Create a empty Case-object
65      case14 = tf.Case()
66
67      #Initialize the Case-object with the data of case14
68      case14.icount = 0
69      case14.convergence = False
70      case14.sbase = 100
71
72      #Bus data
73      case14.nbuses = 14
74      case14.slackbusnr = 0
75      case14.numbus = np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14])
76      case14.buscod = np.array([3,2,2,1,1,2,1,2,1,1,1,1,1,1])
77      case14.basekv = np.zeros(14, dtype = np.double)
78      case14.gs = np.zeros(14, dtype = np.double)
```

```
79      case14.bs = np.array([0,0,0,0,0,0,0,0,0.19,0,0,0,0,0])
80      case14.area = np.ones(14, dtype = np.int32)
81      case14.zone = np.ones(14, dtype = np.int32)
82      case14.pload = np.array
        ([0,0.217,0.942,0.478,0.076,0.112,0,0,0.295,0.09,0.035,0.061,0.135,0.149])

83      case14.qload = np.array
        ([0,0.127,0.19,-0.039,0.016,0.075,0,0,0.166,0.058,0.018,0.016,0.058,0.05])

84      case14.vomag = np.array
        ([1.06,1.045,1.01,1.019,1.02,1.07,1.062,1.09,1.056,1.051,1.057,1.055,1.05,1.0

85      case14.voang = np.array
        ([0,-4.98,-12.72,-10.33,-8.78,-14.22,-13.37,-13.36,-14.94,-15.1,-14.79,-15.07
        /180*math.pi
86      case14.pinj = np.zeros(case14.nbuses, dtype = np.double)
87      case14.qinj = np.zeros(case14.nbuses, dtype = np.double)
88      case14.mismloc = np.zeros(2, dtype = np.int32)
89
90      #Generator data
91      case14.ngens = 5
92      case14.genbus = np.array([0,1,2,-1,-1,3,-1,4,-1,-1,-1,-1,-1,-1])
93      case14.genstat = np.array([True,True,True,True,True])
94      case14.genbase = np.array([100,100,100,100,100])
95      case14.vgbus = np.array([1.06,1.045,1.01,1.07,1.09])
96      case14.pgen = np.array([2.324,0.40,0,0,0])
97      case14.qgen = np.array([-0.169,0.424,0.234,0.122,0.174])
98      case14.qmax = np.array([0.10,0.50,0.40,0.24,0.24])
99      case14.qmin = np.array([0,-0.40,0,-0.06,-0.06])
100     case14.pmax = np.array([3.324,1.40,1.00,1.00,1.00])
101     case14.pmin = np.zeros(5, dtype = np.double)
102
103     #Branch data
104     case14.nlines = 20
105     case14.ifrom = np.array
        ([0,0,1,1,1,2,3,3,3,4,5,5,5,6,6,8,8,9,11,12], dtype = np.int32)
106     case14.ito = np.array
        ([1,4,2,3,4,3,4,6,8,5,10,11,12,7,8,9,13,10,12,13], dtype = np.
        int32)
107     case14.gij = np.array([4.9991316,1.02589745,1.13501919,
        1.68603315, 1.70113967, 1.98597571, 6.84098066, 0.0, 0.0, 0.0,
        1.95502856,
108                            1.52596744, 3.0989274, 0.0, 0.0,
        3.90204955, 1.42400549, 1.88088475, 2.48902459, 1.13699416])
109     case14.bij = np.array([-15.26308652, -4.23498368, -4.78186315,
        -5.11583833, -5.1939274,-5.06881698, -21.57855398, -4.78194338,
        -1.79797907,
110                            -3.96793905, -4.09407434, -3.17596397,
        -6.10275545, -5.67697985, -9.09008272, -10.36539413, -3.02905046,
        -4.40294375,
```

```
111                             -2.25197463,  -2.31496348])
112     case14.xinv = 1/np.array
        ([0.05917,0.22304,0.19797,0.17632,0.17388,0.17103,0.04211,0.20912,0.55618,0.2

113     case14.b = np.array
        ([0.0528,0.0492,0.0438,0.034,0.0346,0.0128,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])

114     case14.gi = np.zeros(20, dtype = np.double)
115     case14.bi = np.zeros(20, dtype = np.double)
116     case14.gj = np.zeros(20, dtype = np.double)
117     case14.bj = np.zeros(20, dtype = np.double)
118     case14.ratio = np.array
        ([1,1,1,1,1,1,1,0.978,0.969,0.932,1,1,1,1,1,1,1,1,1,1])
119     case14.shift_angle = np.zeros(20, dtype = np.double)
120     case14.ibstat = np.array([True,True,True,True,True,True,True,True
        ,True,True,True,True,True,True,True,True,True,True,True,True])
121     case14.gii = np.array([6.02502906, 9.52132361, 3.1209949,
        10.51298952, 9.56801778, 6.57992341, 0.0, 0.0, 5.32605504,
        5.78293431, 3.83591332, 4.01499203, 6.72494615, 2.56099964])
122     case14.bii = np.array([-19.44707021, -30.2721154,-9.82238013,
        -38.65417121, -35.53363946, -17.34073281,
        -19.54900595,-5.67697985, -24.09250638, -14.76833788, -8.49701809,
         -5.42793859, -10.66969355, -5.34401393])
123
124     return case14
125
126 def test_acsolve(case3, case14):
127
128     #Test case3
129     res3 = tf.loadflow(case3)
130     assert_array_almost_equal(res3.vomag, np.array([0.88496373,
        0.88444577, 1.0]), 6)
131     assert_array_almost_equal(res3.voang, np.array([-0.0786143,
        -0.0654881,  0.0]), 6)
132
133     #Test case14
134     for flat in [False, True]:
135         #Start with flat start or not; the result should be the same
136         res14 = tf.loadflow(case14, flat_start = flat)
137         assert_array_almost_equal(res14.vomag, np.array([1.06, 1.045,
         1.01, 1.01767086, 1.01951386, 1.07, 1.06151954,
138                                                 1.09 , 1.05593173,
        1.05098463, 1.05690652, 1.05518856,
139                                                 1.05038172,
        1.03552995]),6)
140         assert_array_almost_equal(res14.voang, np.array([0.0,
        -0.08696258, -0.22209489, -0.17999408, -0.15313263, -0.24820233,
141                                                 -0.23316948,
        -0.23316948, -0.26072638, -0.26349739, -0.25814505, -0.26311858,
142                                                 -0.26452692,
```

```
-0.27983988]),6)
```

C.6 test_acsolve_unit.py

```
1
2 import sys
3 sys.path.append("..")
4 import topflow as tf
5 import topflow.acsolve_wrapper as ac
6 import numpy as np
7 from numpy.testing import assert_array_equal
8 from numpy.testing import assert_allclose
9 import pytest
10 import math
11
12 #This file provides the unit-tests of the sub-functions of acsolve
13
14 @pytest.fixture
15 def case3():
16     case3 = tf.Case()
17     case3.sbase = 100 #MVa base
18     # Bus data
19     case3.nbuses = 3  # number of buses
20     case3.slackbusnr= 2 #the internal number of the slacbus
21     case3.numbus = np.array([0,1,2], dtype=np.int32)
22     case3.buscod = np.array([1, 1, 3], dtype=np.int32)  # bus type.
    1=PQ, 2=PV, 3=Slack, 4= disconnected
23     case3.pload = np.array([1.0, 0.5, 0.0], dtype=np.double)  # pload
    [nbuses]: active load vector (fixed)
24     case3.qload = np.array([0.5, 0.5, 0.0], dtype=np.double)  # qload
    [nbuses]: reactive load vector (fixed)
25     case3.pinj = np.zeros(case3.nbuses, dtype=np.double)  #
    calculated active power injection for all buses
26     case3.qinj = np.zeros( case3.nbuses, dtype=np.double)  #
    calculated reactive power injection for all buses
27     case3.vomag = np.ones(case3.nbuses, dtype=np.double)  # voltage
    magnetudes at all buses
28     case3.voang = np.zeros(case3.nbuses, dtype=np.double)  # voltage
    angles at all buses
29
30     # Generator data
31     case3.genbus = np.array([-1, -1, 0], dtype=np.int32)  # genbus[
    nbuses] if a generator: genbus[k]= generator nr. If not: genbus[k
    ]=-1
32     case3.vgbus = -np.ones(case3.nbuses, dtype=np.int32)
33     case3.pgen = np.array([0.0], dtype=np.double)  # pgen[ngen]:
    active power generated at generator nr
34     case3.qgen = np.array([0.0], dtype=np.double)  # qgen[ngen]:
    reactive power generated at generator nr
35
36     # Line data
37     case3.nlines = 3  # number of lines
```

```
38      case3.gii = np.array([5.17647059, 3.17647059, 6.0], dtype=np.
    double)  # real part of the diagonal element in the ybus
39      case3.bii = np.array([-12.70588235, -10.70588235, -14.0], dtype=
    np.double)  # imag part of the diagonal element in the ybus
40      case3.gij = np.array([1.17647059, 4.0, 2.0], dtype=np.double)  #
    real part of the off-diagonal element in the ybus
41      case3.bij = np.array([-4.70588235, -8.0, -6.0], dtype=np.double)
     # imag part of the off-diagonal element in the ybus
42      case3.xinv = np.array([1 / 0.2, 1 / 0.1, 1 / 0.15])  # inverse of
     the line reactances
43      case3.ratio = np.array([1.0, 1.0, 1.0], dtype=np.double)  #
    transformer ratio for all the lines
44      case3.shift_angle = np.zeros(3) #shift angles of the transformers
45      case3.ifrom = np.array([0, 0, 1], dtype=np.int32)  # from-bus for
     all the lines
46      case3.ito = np.array([1, 2, 2], dtype=np.int32)  # to-bus for all
     the lines
47      case3.ibstat = np.array([True, True, True], dtype=np.bool)  #
    status for all the lines. True= in service
48      return case3
49
50  def test_flatstart():
51      nbuses = 4
52      buscod = np.array([1,1,3,2])
53      vomag = np.array([1.01, 1.02, 1.03, 1.04])
54      voang = np.array([0.1, 0.2, 0.3, 0.4])
55      vomag_ref = np.array([1.01, 1.02, 1.03, 1.04])
56      voang_ref = np.array([0.1, 0.2, 0.3, 0.4])
57
58      for flat_start in [False, True]:
59
60          ac.flatstart(nbuses, buscod, vomag, voang, flat_start)
61
62          if(flat_start):
63              assert_allclose(vomag,np.array([1.00, 1.00, 1.03, 1.04]))
64              assert_allclose(voang,np.array([0.0, 0.0, 0.3, 0.0]))
65          else:
66              assert_allclose(vomag, vomag_ref)
67              assert_allclose(voang, voang_ref)
68
69
70  def test_netinj(case3):
71
72      #The NR-solution of case3:
73      case3.vomag = np.array([0.88496373, 0.88444577, 1.0])
74      case3.voang = np.array([-0.0786143, -0.0654881,  0.0])
75      for pqv in [1,2,3]:
76          #set initial values for pinj and qinj as zero to check the
    operation of pqv:
77          case3.pinj = np.zeros(case3.nbuses)
```

```
78          case3.qinj = np.zeros(case3.nbuses)
79          pinj_init = np.zeros(case3.nbuses)
80          qinj_init = np.zeros(case3.nbuses)
81
82          ac.netinj(pqv,case3.nbuses, case3.nlines, case3.pinj, case3.
        qinj, case3.vomag, case3.voang, case3.gii, case3.bii, case3.gij,
        case3.bij, case3.ratio, case3.shift_angle, case3.ifrom, case3.ito,
         case3.ibstat)
83          #The missmatch between calculated injections and known load
        should be smaller than 10^-8
84
85          if(pqv==1):
86              assert_allclose(case3.pinj[:2], -1*case3.pload[:2])
87              assert_allclose(case3.qinj, qinj_init)
88          elif(pqv == 2):
89              assert_allclose(case3.pinj, pinj_init)
90              assert_allclose(case3.qinj[:2], -1*case3.qload[:2])
91          else:
92              assert_allclose(case3.pinj[:2], -1*case3.pload[:2])
93              assert_allclose(case3.qinj[:2], -1*case3.qload[:2])
94
95
96  def test_missmatch():
97      #Set up the test case
98      nbuses = 4
99      buscod = np.array([1,1,3,2])
100     genbus = np.array([-1,-1,0,1])
101     pgen = np.array([0,0.010])
102     qgen = np.array([0,-0.010])
103     pload = np.array([0.010, 0.020, 0.0, 0.0])
104     qload = np.array([0.010, 0.020, 0.0, 0.0])
105     alfa = beta = np.zeros(4)
106     qdelta = pdelta = 0.0
107     #net injections:
108     pinj = np.array([-0.010, 0.0, 0.100, 0.005])
109     qinj = np.array([0.0,-0.020, 0.100, -0.010])
110
111     #the mismatches overwrites the net injections in the lists pinj
        and qinj.
112     #The exceptions are PV-buses in qinj and the slack-bus in both
        pinj and qinj. These posisitons keeps the net injections
113     # The mismatches should be xgen - xload - xinj for PV-buses and -
        xload - xinj for PQ buses.
114     #Therefore, expected mismatches are:
115     pmism = np.array([0.0,-0.020,0.100,0.005])
116     qmism = np.array([-0.010,0.0,0.100,-0.010])
117
118     #Test all variants of pqv:
119     for pqv in [1,2,3]:
120         #reset xinj values, and make copies
```

```
121         pinj = np.array([-0.010, 0.0, 0.100, 0.005])
122         qinj = np.array([0.0,-0.020, 0.100, -0.010])
123         pinj_init = np.array([-0.010, 0.0, 0.100, 0.005])
124         qinj_init = np.array([0.0,-0.020, 0.100, -0.010])
125 
126         #Run the mismat function
127         ac.mismat(pqv, nbuses, genbus, buscod, pgen, qgen, pload,
    qload, pinj, qinj, alfa, beta, pdelta, qdelta)
128 
129         #Excpected results:
130         if(pqv == 1):
131             assert_allclose(pinj, pmism)
132             assert_allclose(qinj, qinj_init)
133         elif(pqv == 2):
134             assert_allclose(pinj, pinj_init)
135             assert_allclose(qinj, qmism)
136         else:
137             assert_allclose(pinj, pmism)
138             assert_allclose(qinj, qmism)
139 
140 def test_maxmism():
141     nbuses = 4
142     buscod = np.array([2,3,1,-2])
143     mismloc = np.array([0,0])
144     pinj = np.array([0.0,0.150,-0.100,0.010])
145     qinj = np.array([0.150, 0.0, 0.010, -0.100])
146 
147     for pqv in [1,2,3]:
148         mismloc = np.array([0,0])
149         ac.maxmism(pqv, nbuses, buscod, mismloc, pinj, qinj)
150         if(pqv == 1):
151             assert_array_equal(mismloc, np.array([2,0]))
152         elif(pqv == 2):
153             assert_array_equal(mismloc, np.array([0,3]))
154         else:
155             assert_allclose(mismloc, np.array([2,3]))
156 
157 def test_enforce_qlim():
158     nbuses = 6
159     qmism = 1 #worst missmatch at 1
160     buscod = np.array([3,1,-2,-2,2,2])
161     genbus = np.array([0,-1,1,2,3,4])
162     numbus = np.array([0,1,2,3,4,5])
163     pinj = np.array([1.1,0.2,0.3,0.0,0,0.1])
164     qinj = np.array([0,0.3,0.1,0.2,0,0]) #missmatches
165     qgen = np.array([1.4,-0.2,0.5,0.9,-0.3])
166     qmax = np.array([10, 0.5, 0.5, 0.5, 0.5])
167     qmin = np.array([-0.2,-0.2,-0.2,-0.2,-0.2])
168     vomag = np.array([1.02,1.0, 0.92, 1.01, 1.03, 1.04])
169     vgbus = np.array([1.02, 0.98, 0.95, 1.03, 1.04])
```

```
170     #Can see that:
171     #bus 2 is var-limited on the lower q-limit
172     #bus 3 is var-limited on the upper q-limit
173     #vomag[2]<vgbus[genbus[2]] and vomag[3]>vgbus[genbus[3]]
174     #therefor both 2 and 3 should be put back as PV-buses
175     #bus 4 violates the upper limit by 0.4
176     #bus 5 violates the lover limit by -0.1
177     #Therefor bus 4 and 5 should be put inn as var-limited buses
178     #the new worst mismatch should be 0.4 at bus 4 after the function
        call
179
180     qmism = ac.enforce_qlim(False, nbuses,qmism,buscod,genbus,numbus,
      pinj,qinj,qgen,qmin,qmax,vomag,vgbus)
181
182     assert qmism == 4
183     assert_allclose(buscod, np.array([3,1,2,2,-2,-2]))
184     assert qgen[genbus[4]] == qmax[genbus[4]]
185     assert qgen[genbus[5]] == qmin[genbus[5]]
186     assert qinj[2] == 0.0
187     assert qinj[3] == 0.0
188
189 def test_bujac(case3):
190     melem = 5 * (case3.nbuses + 2 * case3.nlines) #allocate enough
      space for the sparse matrix
191     jacbi = np.zeros((melem), dtype=np.double)
192     jcol = -np.ones((melem), dtype=np.int32)
193     ipv = -np.ones((melem), dtype=np.int32)
194
195     ac.bujac(
196             case3.nbuses,
197             case3.nlines,
198             jacbi,
199             case3.vomag,
200             case3.voang,
201             case3.gii,
202             case3.bii,
203             case3.gij,
204             case3.bij,
205             case3.ratio,
206             jcol,
207             ipv,
208             case3.ifrom,
209             case3.ito,
210             case3.ibstat,
211         )
212     assert_allclose(jacbi, np.array([12.70588235, 10.70588235, 14.0,
      12.70588235, 10.70588235, 14.0,
213                                      5.17647059, 3.17647059,  6.0,
      -4.70588235, -4.70588235, -1.17647059,
214                                     -1.17647059, 1.17647059,
```

```
    -5.17647059, 1.17647059, -3.17647059, -4.70588235,
215                                      -4.70588235,-8.0,-8.0, -4.0,
    -4.0, 4.0,4.0, -6.0, -8.0, -8.0, -6.0, -6.0,
216                                      -2.0, -2.0,  2.0,  2.0, -6.0,
    -6.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]))
217
218     assert_allclose(jcol, np.array([0, 1, 2, 3, 4, 5, 3, 4, 5, 1, 0,
    4, 3, 1, 0, 0, 1, 4, 3, 2, 0, 5, 3, 2,
219                                     0, 2, 5, 3, 2, 1, 5, 4, 2, 1, 5,
    4, -1, -1, -1, -1, -1, -1, -1, -1, -1]))
220
221     assert_allclose(ipv, np.array([6, 7 , 8, 13 , 15, 24,  9, 10, 20,
     11, 12, 19, 28, 14, 17, 16, 18, 23, 32, 21, 22, -1, 29, 26,
222                                    25, 27, -1, 33, 30, 31, -1, -1, 34,
     35, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]))
223
224 def test_jacsize():
225     jcol = np.array([1,2,3,4,5,6,7,-1,-1,-1,-1])
226     jacsize = ac.jacsize(jcol)
227     assert jacsize == 7
228
229 def test_coo_conv(case3):
230
231     melem = 5 * (case3.nbuses + 2 * case3.nlines)
232     buscod = np.array([1,1,3])
233     jacbi = np.array([12.70588235, 10.70588235, 14.0, 12.70588235,
    10.70588235, 14.0,
234                                       5.17647059, 3.17647059,  6.0,
    -4.70588235, -4.70588235, -1.17647059,
235                                      -1.17647059, 1.17647059,
    -5.17647059, 1.17647059, -3.17647059, -4.70588235,
236                                      -4.70588235,-8.0,-8.0, -4.0,
    -4.0, 4.0,4.0, -6.0, -8.0, -8.0, -6.0, -6.0,
237                                      -2.0, -2.0,  2.0,  2.0, -6.0,
    -6.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
238     ipv = np.array([6, 7 , 8, 13 , 15, 24,  9, 10, 20, 11, 12, 19,
    28, 14, 17, 16, 18, 23, 32, 21, 22, -1, 29, 26,
239                                    25, 27, -1, 33, 30, 31, -1, -1, 34,
     35, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1])
240     jcol = np.array([0, 1, 2, 3, 4, 5, 3, 4, 5, 1, 0, 4, 3, 1, 0, 0,
    1, 4, 3, 2, 0, 5, 3, 2,
241                                     0, 2, 5, 3, 2, 1, 5, 4, 2, 1, 5,
    4, -1, -1, -1, -1, -1, -1, -1, -1, -1])
242     isa = -np.ones(melem, dtype = np.int32)
243     row = np.zeros((melem), dtype = np.int32)
244     col = np.zeros((melem), dtype = np.int32)
245     data = np.zeros((melem), dtype = np.double)
246
247     ac.coo_conv(case3.nbuses, case3.ngens, case3.buscod, isa, ipv,
    row, jcol, col, jacbi, data)
```

```
248
249     assert_array_equal(row, np.array([0, 0, 0, 0, 1, 1, 1, 1, 2, 2,
        2, 2, 3, 3, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
250                                       0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0]))
251
252     assert_array_equal(col, np.array([0, 2, 1, 3, 1, 3, 0, 2, 2, 1,
        0, 3, 3, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
253                                       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0]))
254
255     assert_allclose(data, np.array([12.70588235,  5.17647059,
        -4.70588235, -1.17647059, 10.70588235,  3.17647059,
256                                     -4.70588235, -1.17647059,
        12.70588235,  1.17647059, -5.17647059, -4.70588235,
257                                     10.70588235,  1.17647059,
        -3.17647059, -4.70588235,  0.0,  0.0, 0.0,
258                                      0.0, 0.0, 0.0, 0.0, 0.0,  0.0,
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  0.0, 0.0,
259                                      0.0, 0.0, 0.0, 0.0, 0.0,  0.0,
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  0.0, 0.0]))
260 def test_set_rhs():
261     nbuses = 4
262     ngens = 3 #number of generators (var-limited included)
263     nlimgens = 1 #number of limited generators
264     buscod = np.array([1,2,3,-2])
265     vomag = np.array([1.01, 1.02, 1.03, 1.04])
266     pinj = np.array([0.1, 0.2, 0.0, 0.4]) #mismatches
267     qinj = np.array([-0.1, 0.0, 0.0, -0.4]) #mismatches
268
269     for pqv in [1,2,3]:
270
271         if(pqv == 1):
272             rhs = np.zeros(nbuses)
273             ac.set_rhs(pqv, nbuses, buscod, vomag, pinj, qinj, rhs)
274             assert_allclose(rhs, np.array([0.1/1.01, 0.2/1.02, 0.0,
        0.4/1.04]))
275         elif(pqv == 2):
276             rhs = np.zeros(nbuses)
277             ac.set_rhs(pqv, nbuses, buscod, vomag, pinj, qinj, rhs)
278             assert_allclose(rhs, np.array([-0.1/1.01, 0.0, 0.0,
        -0.4/1.04]))
279         else:
280             rhs = np.zeros(2*nbuses-ngens+nlimgens-1)
281             ac.set_rhs(pqv, nbuses, buscod, vomag, pinj, qinj, rhs)
282             assert_allclose(rhs, np.array([0.1, 0.2, 0.4, -0.1,
        -0.4]))
283
284 def test_update_voltages():
285     nbuses = 4
```

```
286     ngens = 3 #number of generators (var-limited included)
287     nlimgens = 1 #number of limited generators
288     buscod = np.array([1,2,3,-2])
289     vomag_init = np.array([1.01, 1.02, 1.03, 1.04])
290     voang_init = np.array([0.1, 0.2, 0.3, 0.4])
291
292     for pqv in [1,2,3]:
293         vomag = np.array([1.01, 1.02, 1.03, 1.04])
294         voang = np.array([0.1, 0.2, 0.3, 0.4])
295
296         if(pqv == 1):
297             #the correction vector for the FDLF should be of
    dimension: nbuses
298             #all buses are included (also slack and generators)
299             correction = np.array([0.1,0.2,0.3, 0.4])
300             ac.update_voltages(pqv, nbuses, buscod, vomag, voang,
    correction)
301             assert_allclose(vomag, vomag_init)
302             assert_allclose(voang, np.array([0.2, 0.4, 0.3, 0.8]))
303         elif(pqv == 2):
304             #the correction vector for the FDLF should be of
    dimension: nbuses
305             #all buses are included (also slack and generators)
306             correction = np.array([0.01,0.02,0.03, 0.04])
307             ac.update_voltages(pqv, nbuses, buscod, vomag, voang,
    correction)
308             assert_allclose(vomag, np.array([1.02, 1.02, 1.03, 1.08])
    )
309             assert_allclose(voang, voang_init)
310         else:
311             #the correction vector for the NR should be of dimension
    2*nbuses-ngens+nlimgens(var-limited generators)
312             #slack bus is not part of correction-vector, neither is
    generors for the magnitude-corrections)
313             correction = np.array([0.1, 0.2, 0.4, 0.01, 0.04])
314             ac.update_voltages(pqv, nbuses, buscod, vomag, voang,
    correction)
315             assert_allclose(vomag, np.array([1.02, 1.02, 1.03, 1.08])
    )
316             assert_allclose(voang, np.array([0.2, 0.4, 0.3, 0.8]))
```

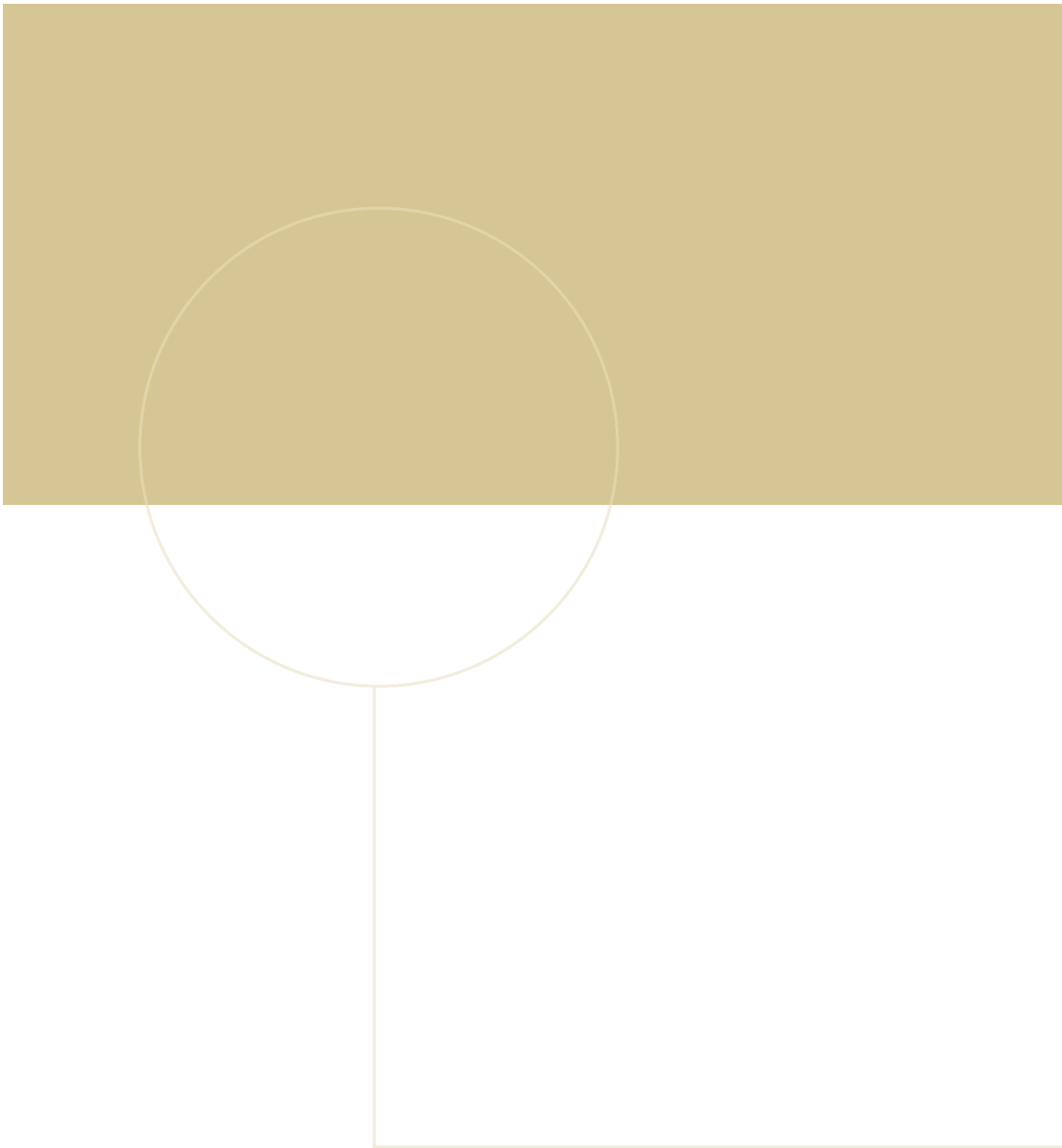
C.7 test_case.py

```
1
2 import sys
3
4 sys.path.append("..")
5
6 import math
7 import topflow as tf
8 import numpy as np
9 from numpy.testing import assert_array_equal
10 from numpy.testing import assert_allclose
11
12 #This file contains the unit-tests of the Case-calss
13
14 def test_loadxl():
15     case14 = tf.example_case('case14')
16     assert case14.icount == 0
17     assert case14.convergence == False
18     assert case14.sbase == 100
19
20     #Bus data
21     assert case14.nbuses == 14
22     assert case14.slackbusnr == 0
23     assert_array_equal(case14.numbus, np.array
      ([1,2,3,4,5,6,7,8,9,10,11,12,13,14]))
24     assert_array_equal(case14.buscod , np.array
      ([3,2,2,1,1,2,1,2,1,1,1,1,1,1]))
25     assert_array_equal(case14.basekv , np.zeros(14))
26     assert_array_equal(case14.gs , np.zeros(14))
27     assert_array_equal(case14.bs , np.array
      ([0,0,0,0,0,0,0,0,0.19,0,0,0,0,0]))
28     assert_array_equal(case14.area , np.ones(14, dtype = np.int32))
29     assert_allclose(case14.zone , np.ones(14, dtype = np.int32))
30     assert_allclose(case14.pload , np.array
      ([0,0.217,0.942,0.478,0.076,0.112,0,0,0.295,0.09,0.035,0.061,0.135,0.149])
      )
31     assert_allclose(case14.qload , np.array
      ([0,0.127,0.19,-0.039,0.016,0.075,0,0,0.166,0.058,0.018,0.016,0.058,0.05])
      )
32     assert_allclose(case14.vomag , np.array([1.06      , 1.045     ,
      1.01      , 1.01767085, 1.01951386,
33                                              1.07      , 1.06151953,
      1.09      , 1.05593172, 1.05098463,
34                                              1.05690652, 1.05518856,
      1.05038171, 1.03552995]))
35
36     assert_allclose(case14.voang , np.array([ 0.        ,
      -0.08696259, -0.22209489, -0.17999408, -0.15313264,
37                                              -0.24820234, -0.23316948,
```

```
    -0.23316948, -0.26072638, -0.26349739,
38                                          -0.25814505, -0.26311859,
    -0.26452692, -0.27983989]))
39
40      #Generator data
41      assert case14.ngens == 5
42      assert_array_equal(case14.genbus , np.array
    ([0,1,2,-1,-1,3,-1,4,-1,-1,-1,-1,-1,-1]))
43      assert_array_equal(case14.genstat , np.array([True,True,True,True
    ,True]))
44      assert_array_equal(case14.genbase , np.array
    ([100,100,100,100,100]))
45      assert_allclose(case14.vgbus , np.array
    ([1.06,1.045,1.01,1.07,1.09]))
46      assert_allclose(case14.pgen , np.array([2.324,0.40,0,0,0]))
47      assert_allclose(case14.qgen , np.array
    ([-0.169,0.424,0.234,0.122,0.174]))
48      assert_array_equal(case14.qmax , np.array
    ([0.10,0.50,0.40,0.24,0.24]))
49      assert_array_equal(case14.qmin , np.array
    ([0,-0.40,0,-0.06,-0.06,]))
50      assert_allclose(case14.pmax , np.array
    ([3.324,1.40,1.00,1.00,1.00]))
51      assert_array_equal(case14.pmin , np.zeros(5))
52
53      #Branch data
54      case14.nlines = 20
55      assert_array_equal(case14.ifrom , np.array
    ([0,0,1,1,1,2,3,3,3,4,5,5,5,6,6,8,8,9,11,12]))
56      assert_array_equal(case14.ito , np.array
    ([1,4,2,3,4,3,4,6,8,5,10,11,12,7,8,9,13,10,12,13]) )
57      assert_allclose(case14.xinv , 1/np.array
    ([0.05917,0.22304,0.19797,0.17632,0.17388,0.17103,0.04211,0.20912,0.55618,0.2
    )
58      assert_allclose(case14.b , np.array
    ([0.0528,0.0492,0.0438,0.034,0.0346,0.0128,0,0,0,0,0,0,0,0,0,0,0,0,0,0])
    )
59      assert_array_equal(case14.gi , np.zeros(20))
60      assert_array_equal(case14.bi , np.zeros(20))
61      assert_array_equal(case14.gj , np.zeros(20))
62      assert_array_equal(case14.bj , np.zeros(20))
63      assert_allclose(case14.ratio , np.array
    ([1,1,1,1,1,1,1,0.978,0.969,0.932,1,1,1,1,1,1,1,1,1,1]))
64      assert_array_equal(case14.shift_angle , np.zeros(20))
65      assert_array_equal(case14.ibstat , np.array([True,True,True,True,
    True,True,True,True,True,True,True,True,True,True,True,True,True,True,
    True,True,True]))
```

D setup.py

```
1 import os
2 from pathlib import Path
3 from setuptools import setup, find_packages
4 #This is the setup-file of Topflow
5 #It specifies dependencies,
6
7 setup(
8     name="topflow",
9     version="0.0.1",
10    packages=find_packages('topflow'),
11
12    # Project uses the following python packages, so ensure that
      these get
13    # installed or upgraded on the target machine:
14    install_requires=["numpy", "scipy", "scons", "openpyxl"],
15
16
17    # metadata to display on PyPI
18    author="aasmunsa",
19    author_email="aa.selen@hotmail.com",
20    description="Load flow",
21
22 )
23
24 #dir_path = str(Path(__file__).parent.absolute())
25 #os.chdir(dir_path + "/topflow")
26 #os.system('scons')
```



NTNU

Kunnskap for en bedre verden