Modelling and simulation of the operation of diesel generators on offshore drilling rigs by using the Python programming language

Čemeljić, Hrvoje

UNIVERSITY OF RIJEKA

FACULTY OF MARITIME STUDIES

HRVOJE ČEMELJIĆ

Modelling and simulation of the operation of diesel generators on offshore drilling rigs by using the Python programming language

MASTER'S THESIS

Zagreb, 2020.

UNIVERSITY OF RIJEKA

FACULTY OF MARITIME STUDIES

Modelling and simulation of the operation of diesel generators on offshore drilling rigs by using the Python programming language

MASTER'S THESIS

Subject: Modelling and simulations

Study programme: Marine Electronics Engineering and Information Technology

Professor: Asst. Prof. Dario Ogrizović, Ph.D.

Author: Hrvoje Čemeljić

Zagreb, 2020.

Student/studentica: Hrvoje Čemeljić

Studijski program: Elektroničke i Informatičke Tehnologije u Pomorstvu

JMBAG: 0112059342

IZJAVA O SAMOSTALNOJ IZRADI DIPLOMSKOG RADA

Kojom izjavljujem da sam diplomski rad s naslovom

Modelling and simulation of the operation of diesel generators on offshore drilling rigs by using the Python programming language
(naslov diplomskog rada)

izradio/la samostalno pod mentorstvom
Asst. Prof. Dario Ogrizović, Ph.D.

(prof. dr. sc. / izv. prof. dr. sc. / doc dr. sc Ime i Prezime)

te komentorstvom _____

stručnjaka/stručnjakinje iz tvrtke _____
(naziv tvrtke).

U radu sam primijenio/la metodologiju izrade stručnog/znanstvenog rada i koristio/la literaturu koja je navedena na kraju diplomskog rada. Tuđe spoznaje, stavove, zaključke, teorije i zakonitosti koje sam izravno ili parafrazirajući naveo/la u diplomskom radu na uobičajen, standardan način citirao/la sam i povezao/la s fusnotama i korištenim bibliografskim jedinicama, te nijedan dio rada ne krši bilo čija autorska prava. Rad je pisan u duhu hrvatskoga jezika.

Suglasan/na sam s trajnom pohranom diplomskog rada u cjelovitom tekstu u mrežnom digitalnom repozitoriju Pomorskog fakulteta Sveučilišta u Rijeci te Nacionalnom repozitoriju Nacionalne i sveučilišne knjižnice.

Za navedeni rad dozvoljavam sljedeće pravo i razinu pristupa mrežnog objavljivanja:
(zaokružiti jedan ponuđeni odgovor)

(a) rad u otvorenom pristupu

b) pristup svim korisnicima sustava znanosti i visokog obrazovanja RH

c) pristup korisnicima matične ustanove

d) rad nije dostupan

Student/studentica

(potpis)

Ime i prezime studenta/studentice: Hrvoje Čemeljić

ON THE MODEL DEVELOPMENT AND DATA SOURCE

The model developed in the programming language Python, for the purpose of this Master's Thesis is solely used for academic purposes and is not intended for any financial gain. The entire code was developed by the author of this Thesis. The author does not claim any rights to the code and wishes to make it publicly available in order to encourage sharing and collaboration by both industry and academia.

Github repository is available at: https://github.com/hcemeljic/Diesel-Generators-Simulator

The dataset used to develop the model is a result of real-world measurements taken by the author under a non-disclosure agreement (NDA), while working in offshore oil and gas drilling. The data was measured, securely stored and randomised to protect the data source, and as such will not be made available publicly.

ABSTRACT

The study researches the operational performance of the DGs (diesel generators) and the PMS (power management system) on board an offshore drilling rig. The focus of this study is the reduction of running hours, fuel consumption and CO_2 emissions of the DGs while increasing the reliability, stability and performance of the power generation system on offshore jackup drilling rigs.

To estimate the running hours and the fuel consumption, a Pythion simulation model is developed. The model allows the author to virtually run the engines and observe the engine behaviour. By running simulations under different operational parameters, the author can compare the outcomes of each simulation. Comparison of outcomes allows the author to determine the optimum parameters for operating the PMS.

The study provides an insight into a specific problem in offshore oil and gas drilling associated with *tripping* and *POOH* (pulling out of the hole) operations. The phenomena is observed and a proposed solution to the problem is given in the form of adjustment of the operational parameters.

The annual savings in the form of reduced running hours, reduced fuel consumption and reduced CO_2 emissions are provided.

Keywords: diesel generator, drilling, power management system, Python, tripping

SAŽETAK

U ovom radu istražene su karakteristike rada Diesel generatora i PMS-a (sustav upravljanja energijom) na platformi za podmorno bušenje nafte i plina. Glavni cilj istraživanja je redukcija radnih sati i potrošnje goriva generatora, uz prirast pouzdanosti, stabilnosti i učinka generatora.

Razvijen je Python program kojim se može simulirati rad generatora u svrhu estimiranja potrošnje goriva i radnih sati. Provođenje simulacija pod različitim operativnim parametrima rezultira različitim estimiranim vrijednostima potrošnje goriva i radnih sati. Usporedba estimiranih vrijednosti omogućuje određivanje optimalnih parametara za rad generatora.

Istraživanje prikazuje poteškoće u radu generatora tijekom izvođenja specifičnih operacija bušenja nafte i plina. Ponuđeno je rješenje koje otklanja poteškoće u radu generatora.

Predstavljene su godišnje uštede radnih sati, potrošnje goriva i CO_2 emisija.

Ključne riječi: diesel generator, bušenje, sustav upravljanja energijom, Python, tripping

CONTENTS

1. SYSTEM DESIGN

Note that an offshore jackup drilling rig is used as a case study for this thesis. Due to their fixed position, and a hull that is elevated above the surface of the sea, jackup rigs in general operate under less dynamic loads with lesser chance and less severe consequences of blackouts than DP controlled floating objects.

Key system design is briefly discussed.

1.1. MAIN DIESEL GENERATOR ENGINES

The power management system on which the study is based consists of five identical main diesel generator engines (DGs) nominally rated at 1830 ekW, 690 VAC, 60 Hz. Diesel generator engines work in a standardised load sharing, parallel mode [1].

For the purpose of this study, only the key characteristics of the DGs are mentioned.

Table 1. Load, power and fuel consumption for the 1830 ekW marine MDG

PERCENT LOAD [%]	ENGINE POWER [BHP]	FUEL CONSUMPTION [GAL/HR]
100	2,588	124.6
90	2,329	113.9
80	2,071	102.5
75	1,941	96.4
70	1,812	90.5
60	1,553	79.0
50	1,294	66.9
40	1,035	55.0
30	776	43.5
25	647	38.0
20	518	32.5
10	259	21.9

Source: Technical information portal (2020, June.) CAT. Available at https://www.cat.com/en_ZA/additional-product-information/product-families/technical-informationportal.html (26. 03. 2020.) [2]

Each of the DGs is controlled by an automation system that remotely starts and stops the DGs depending on the power requirement in the facility. When the demand for power increases the system will start an additional engine. When the demand for power decreases the system will stop the engine that was last started.

1.2. AUTOMATION SYSTEM

The automation system is responsible for remote control of the DGs. The system consists of numerous electronic components, PLCs and a central HMI station. For this study, the hardware of the system is not examined. Instead, the focus is on the operation principle, data logging and control features of the system.

Key features of the system are presented:

- PMS is used to automatically start and stop generator engines, depending on load percentages and time spent in certain load zones.
- PMS automatically starts a generator engine when the demand for power increases. This is known as the load-dependent start [3].
- PMS automatically stops a generator engine when the demand for power decreases. This is known as the load-dependent stop [4].
- In addition, the system offers a comprehensive data logger that stores many physical quantities measured on DGs, switchboards and other instrumentation. These stored measurements serve as the basis for the attempt to develop a PMS simulator program in Python and to run simulations in order to determine the optimal parameters for operating the engine room.

2. POWER MANAGEMENT SYSTEM OPERATION PRINCIPLE

PMS handles many controls and monitoring options, but in this study the focus is on two key features:

- Load dependent start
- Load dependent stop

2.1. LOAD DEPENDENT START

Load-dependent start is a process of automatically starting an additional engine when the power requirements increase and the engines that are already running cannot cope with the power demand.

When the demand for power increases the PMS will start an additional engine.

The load dependent start is achieved via two interdependent parameters:

- START LIMIT [%]
- START TIME [s]

When an engine continuously runs at load higher than the value of the START LIMIT parameter, for a period equal to the value of the START TIME parameter, a new engine will be started. If the load of the engine drops below the START LIMIT parameter, the starting of the additional engine will be cancelled.

For each consecutive second spent at load higher than the value of the START LIMIT parameter, a countdown from the value of START TIME to zero will be engaged. Should the countdown reach zero, a command to start the additional engine will be issued by the PMS. Should the load of the engine drop below the value of the START LIMIT parameter, a countdown will be reset back to the original value and the starting will be cancelled.

If the engine load is below the value of the START LIMIT parameter, no countdown will commence, and no additional engines will be started.

Example of a load dependent start:

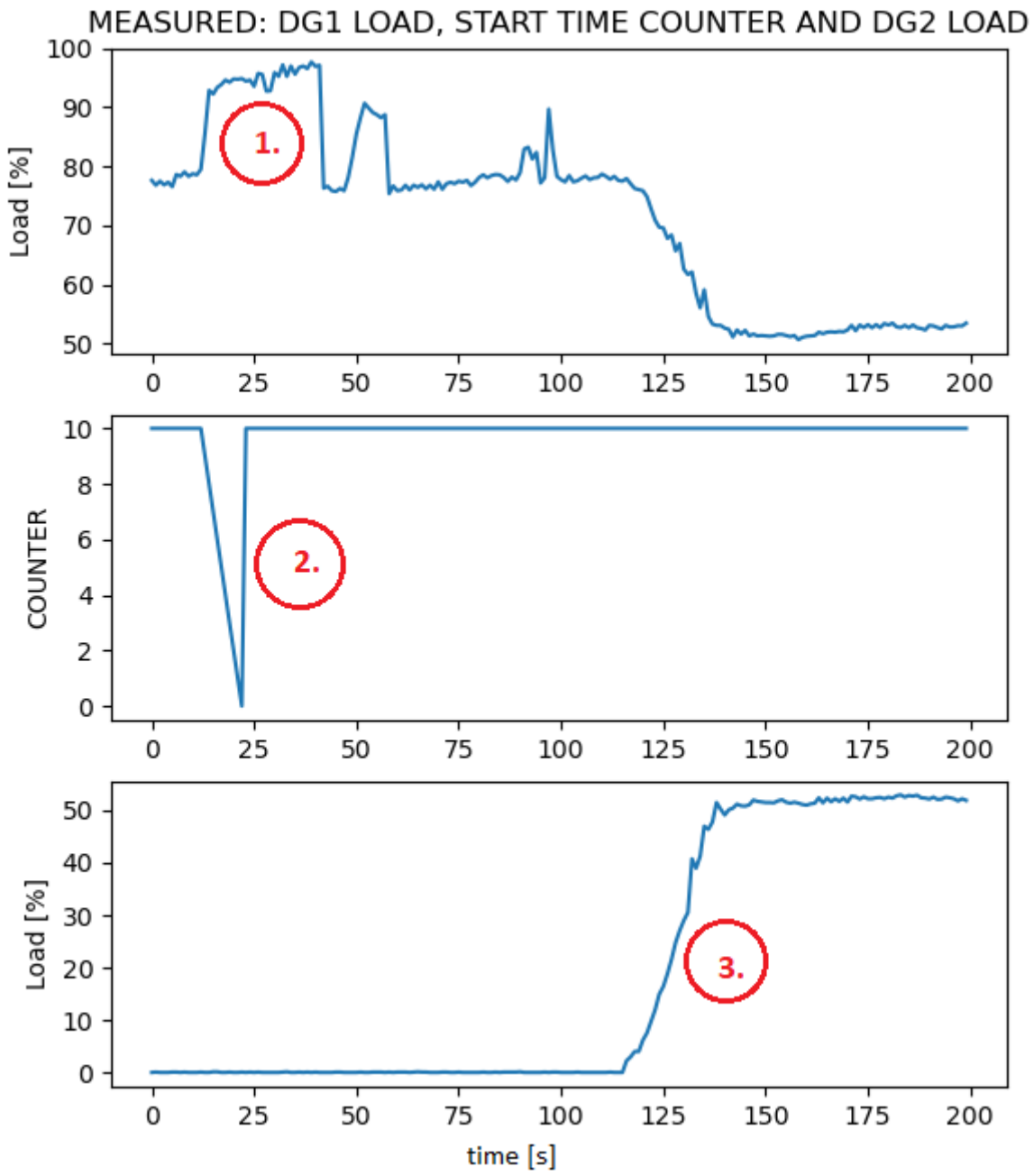

Figure 1. Example of a load dependent start

Source: Created by the author using the matplotlib library

Figure 1 shows an example of an automated starting of DG2 based on a load dependent start and the following parameter values:

START LIMIT = 85%

START TIME = 10 seconds

The graph is analysed according to the numbers circled in red:

1. DG1 increases load from approximately 80% to approximately 95%. DG1 now runs at load higher than the START LIMIT parameter (95% > 85%).
2. Countdown from 10 to zero is initiated. DG1 stays above the START LIMIT parameter for 10 consecutive seconds and the countdown to zero is reached. At this time, an automatic start of DG2 is initiated by the PMS.
3. With a warm-up delay of approximately 100 seconds, DG2 synchronises and starts load sharing. Mean load is established at approximately 50%. Warm-up delay is associated with a pre-lube cycle, build-up of the starting air, the time for diesel to ignite, etc. [5]

2.2. LOAD DEPENDENT STOP

Load-dependent stop is a process of automatically stopping the engine that was last started when the demand for power decreases and the engines begin to run at undesirably low load.

When a demand for power decreases the PMS will stop an engine, unless exactly one engine is running.

The load dependent stop is achieved via two interdependent parameters:

- STOP LIMIT [%]
- STOP TIME [s]

The following condition must be satisfied for the load-dependent stop to become active:

$$\frac{m * L}{m - 1} < \text{STOP LIMIT}$$

Where:

- m = the number of running engines
- L = load per engine

When an engine continuously runs at load that satisfies the condition above, for a period equal to the value of the STOP TIME parameter, the last engine that was started will be

unloaded and stopped. If the load of the engine increases and the condition is not satisfied, the stopping of the engine will be cancelled.

For each consecutive second spent at load that satisfies the condition, a countdown from the value of STOP TIME to the value of zero will be engaged. Should the countdown reach zero, a command to stop the last started engine will be outputted. Should the load of the engine change in such a way that the condition is not satisfied, a countdown will be reset back to the original value and the stopping will be cancelled.

If the engine load does not satisfy the condition, no countdown will commence and no engine will be stopped.

Example of a load-dependent stop:

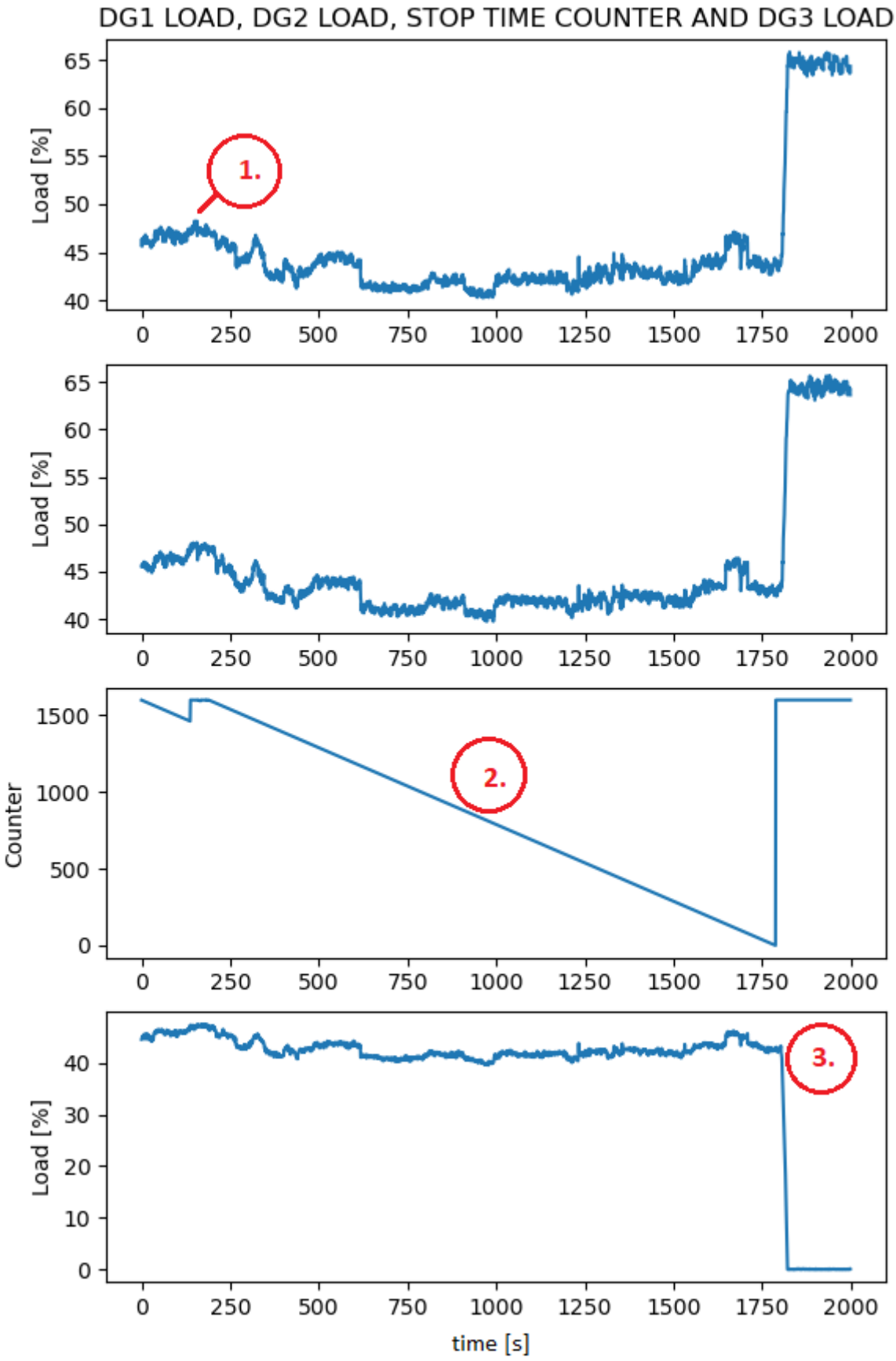

Figure 2. Example of a load dependent stop

Source: Created by the author using the matplotlib library

Figure 2 shows an example of an automated stopping of DG3 based on the load-dependent stop and the following parameter values:

STOP LIMIT = 71%

STOP TIME = 1599 seconds

The graph is analysed according to the numbers circled in red. Note that in this example three engines are running.

1. DG1, DG2 and DG3 decrease load from approximately 48% to approximately 42%, entering the zones for which the following condition is true:

$$\frac{m * L}{m - 1} < \text{STOP LIMIT}$$

   Where:

   - m = the number of running engines
   - L = load per engine

   When the load equals 42% and three engines are running, the calculation gives an estimated mean load of 63% for two running engines, satisfying the condition for countdown (64% < 71%).

2. Countdown from 1599 to zero is initiated. Both DG1, DG2 and DG3 stay below the STOP LIMIT for 1599 consecutive seconds and the countdown to zero is reached. At this time, an automatic stop is initiated from the PMS.
3. DG3 is unloaded, while DG1 and DG2 are ramped-up to compensate for the power previously delivered by DG3. DG3 reaches zero load and is disconnected, while DG1 and DG2 continue load sharing. Mean load is established at approximately 65%.

At 65% load, the system is balanced and no load-dependent starting or stopping will commence. Load of 65% is lower than the START LIMIT parameter (65% < 85%) so a load-dependent start is not required. Likewise, the condition for load dependent stop would give an estimate of 130% per engine in case one fewer engine was running, so a load-dependent stop is not required either.

Example of a combined load dependent start and stop:

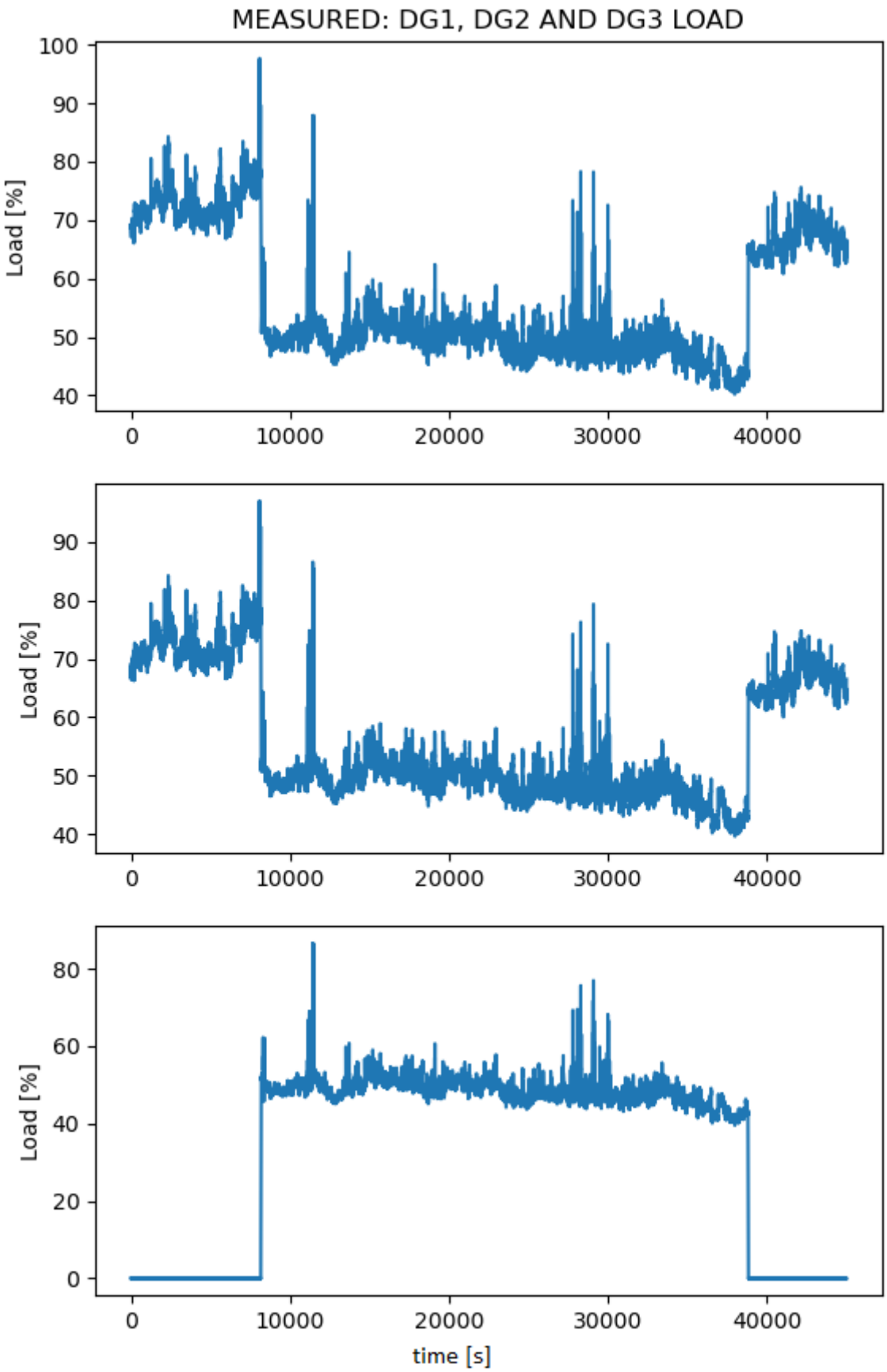

Figure 3. DG1, DG2, DG3 load dependent start and stop

Source: Created by the author using the matplotlib library

Figure 3 displays a line graph of DG1, DG2 and DG3 load over a period of 40000 seconds. Early time is spent with DG1 and DG2 running between 70% and 80%, while DG3 is offline. At this time, the system is balanced and there is no requirement for a load-dependent start or stop. As the load increases above 85%, load-dependent start of DG3 is initiated. DG3 is ramped-up, while DG1 and DG2 are ramped down. After DG3 begins to load-share, mean load is established between 50% and 60%. After 30000 seconds, load drops below 50%, which results in a load dependent stop of DG3. DG3 is unloaded and disconnected, while DG1 and DG2 are ramped-up. Mean load is established around 70%. The system is balanced and there are no requirements for load-dependent starting or stopping.

2.3. HIGH-LOAD START

In addition to load-dependent starts and stops, the system also automatically starts an additional engine if the load on any of the engines becomes greater than 100% of their nominal capacity. In this case the control logic will be ignored, and a direct start command will be issued by the PMS regardless of the countdown.

2.3.1. Operational parameters

In the previous chapter the correlation between the operational parameters and the performance of the DGs is demonstrated. START LIMIT and STOP LIMIT values directly affect the PMS decision threshold, while the START TIME and STOP TIME values directly affect the countdown durations. By modifying the parameter values, a significant impact on the performance of the PMS and the DGs can be achieved.

The goal is to explain how a change in one or more parameters affects the performance of the DGs and the drilling facility. In particular, the author is interested in reduction of running hours, reduced fuel consumption, reduced CO_2 emission, stability of the Main SWBD voltage and frequency, reliability of power during peak demand by the drill-floor, establishing a healthy mean load per engine and reducing the stress on the switchgear.

Mentioned points of interest are dependent on the operational parameters of the PMS; thus, the goal of this study is to find the optimal parameters for balanced, safe, cost-effective, and efficient operation of the drilling rig.

To test how a change in parameter values affects the performance of the rig, a Python model of the PMS that can virtually run the engines and help us determine the optimal parameter values is developed.

The model is evaluated by running it against measured values and observing the behaviour of the model. Should the model prove accurate, the value of the total power, recorded over a duration of 55 consecutive days, will be used as an input to the model. Desired outputs will be the estimates of total fuel consumption, running hours, behaviours of starting and stopping sequences, times spent at high and/or low loads, etc.

2.3.2. Commissioned and Modified values

Before developing the Python program, the parameter values at the time of commissioning and at the time of data logging are clarified.

The operational parameters used for load-dependent start and stop were set to the following values during the commissioning of the rig:

```
start_limit = 80  # Given in [%]
start_time = 10  # Given in [s]
stop_limit = 70  # Given in [%]
stop_time = 200  # Given in [s]
```

However, during the time of data logging, the system was operating with the following values:

```
start_limit = 85  # Given in [%]
start_time = 15  # Given in [s]
stop_limit = 71  # Given in [%]
stop_time = 1599  # Given in [s]
```

The reason for the change of the parameters and the significance of the change will be explained in the remainder of the study.

3. CLEAN-UP AND ERROR DETECTION ON THE DATASETS

To perform correct data analysis and create a model that accurately mimic the PMS operation, the recorded data needs to be reviewed for errors. The following chapter describes the methodology of error review and cleaning of the data before it is stored in an SQLite database.

3.1. FILE FORMAT

The data is extracted in MS Excel file format. A typical file consists of more than 60000 rows and 45 columns, containing the recorded physical quantities over a certain period. Physical quantities are recorded at a frequency of one row per second. Each row consists of the following values:

- Timestamp
- Total power [kW]
- Available power [kW]
- SWBD voltage [V]
- Frequency [Hz]
- Number of running engines

Also, for each DG the following values are recorded:

- Power [kW]
- Load [%]
- Reactive power [kVAr]
- Phase factor [cos ϕ]
- Frequency [Hz]
- Voltage [V]

DG4 PHASE FACTOR	DG4 FREQUENCY	DG4 VOLTAGE	DG5 POWER [kW]	DG5 POWER [%]
0.843895018	59.94728088	679.3225098	1283.909302	70.15897369
0.847350061	59.96589661	679.3225098	1295.216309	70.77684784
0.844574332	59.97001648	678.6221313	1274.753906	69.65868378
0.847823083	59.97749329	678.979187	1295.216309	70.77684784
0.846193254	59.99214172	678.979187	1276.035767	69.72872925
0.850855172	59.97001648	678.3062744	1290.546997	70.521698
0.844367027	59.96589661	678.979187	1270.359375	69.41854095
0.848698914	59.94728088	678.6221313	1285.099609	70.22402191
0.839878857	59.94377136	679.3225098	1268.116333	69.29597473
0.840768278	59.9584198	678.979187	1280.476074	69.97137451
0.847391188	59.9546051	679.3225098	1268.116333	69.29597473
0.844821393	59.95109558	678.6221313	1280.476074	69.97137451
0.850806355	59.9546051	679.3225098	1286.152466	70.28155518
0.846571267	59.9584198	679.3225098	1280.476074	69.97137451
0.851113677	59.9546051	679.3225098	1296.269165	70.8343811
0.843012094	59.96589661	678.979187	1285.099609	70.22402191

Figure 4. Representation of a typical XLS table containing recorded values

Source: Created by the author using the MS Excel software

3.2. INITIAL ERROR TESTING

Before writing a Python script for automatic error detection and clean-up of the data, familiarisation with the characteristics of a typical row containing an error is necessary.

The Load of DG2 over time is plotted in matplotlib [6].

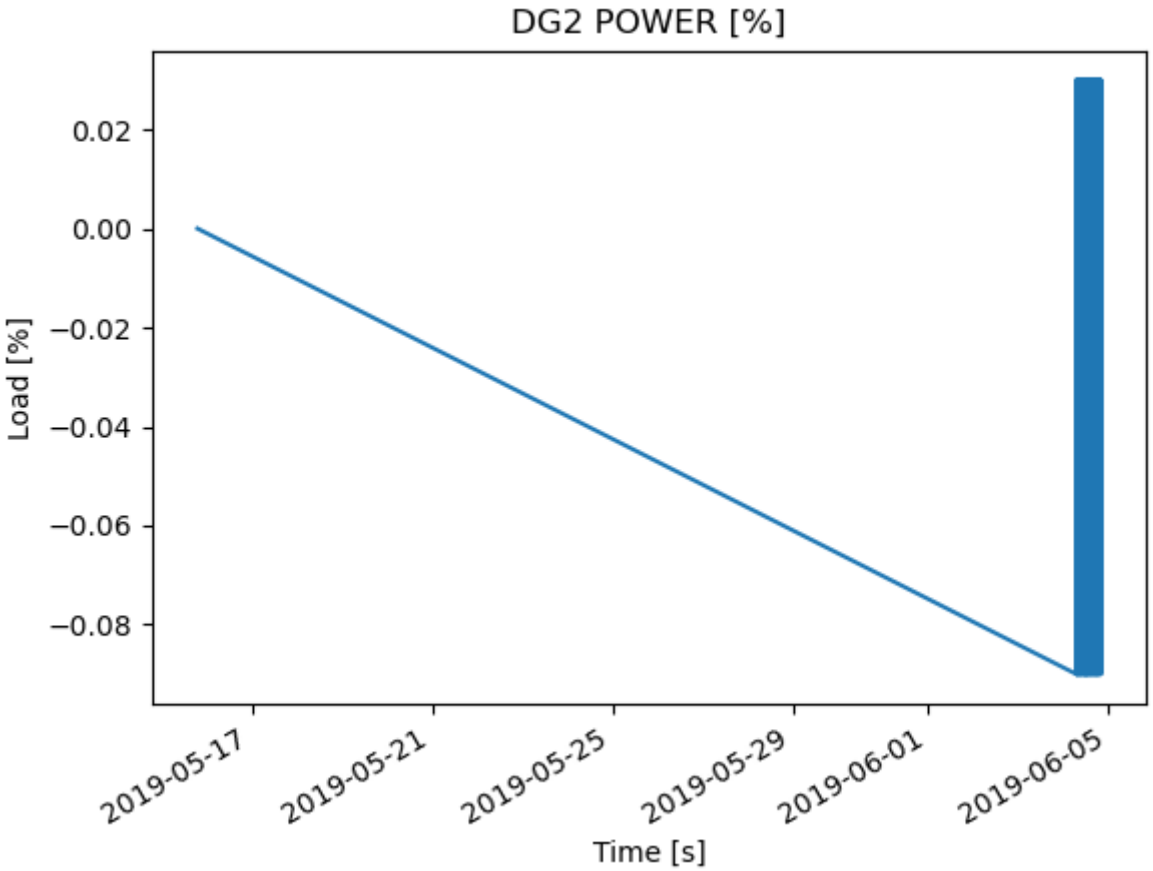

Figure 5. Incomprehensible plot of DG2 power over time

Source: Created by the author using the matplotlib library

An incomprehensible graph points to an error somewhere along the data.

The file is checked manually in Excel by plotting the time column:

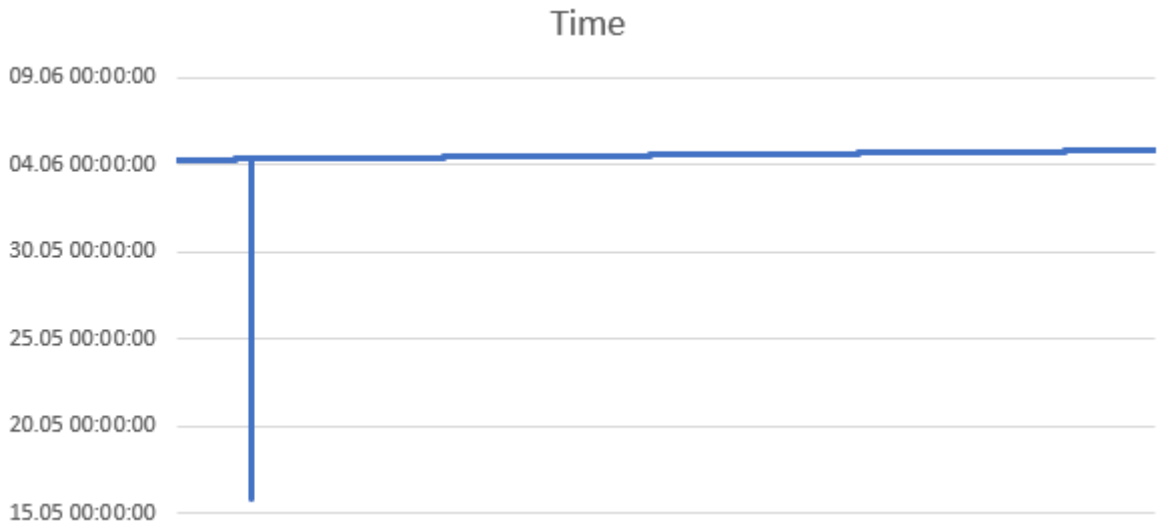

Figure 6. Plot of the time column in an XLS file

Source: Created by the author using the MS Excel software

Non-linear passage of time appears as a result of a faulty timestamp along the time column. The point is inspected manually:

Time	TOTAL POWER	AVAILABLE POWER	SWBD VOLTAGE	SWBD FREQUENCY
04.06 08:07:39	2547.447998	1112.552002	680.3662109	59.98588562
04.06 08:07:40	2541.72583	1118.27417	680.7095337	59.99320984
04.06 08:07:41	2530.327148	1129.672852	680.3662109	59.99320984
04.06 08:07:42	2521.309326	1138.690674	680.3662109	60.00068665
15.05 18:58:12	0	2.8026E-45	0	0
04.06 08:07:44	2529.182861	1130.817139	680.3662109	59.99320984
04.06 08:07:45	2531.517578	1128.482422	680.3662109	59.98970032
04.06 08:07:46	2522.453613	1137.546387	681.0253906	59.98970032
04.06 08:07:47	2534.859375	1125.140625	680.7095337	59.98970032
04.06 08:07:48	2543.923096	1145.46582	680.7095337	59.98970032
04.06 08:07:49	2510.139648	1149.860352	680.7095337	59.98588562
04.06 08:07:50	2513.389648	1146.610352	680.3662109	59.99320984
04.06 08:07:51	2499.793945	1160.206055	680.3662109	60.00450134

Figure 7. Row containing incorrect data

Source: Created by the author using the MS Excel software

By manually reviewing the data, an error is found in the form of an incorrect timestamp and zeros instead of recorded data points. By applying the same methodology, numerous rows

with the same characteristics are found, which leads to the development of a simple cleaning script written in Python.

3.3. AUTOMATED CLEAN-UP SCRIPT

The following part of the paper describes the code and logic behind the automated error detection and clean-up script written in Python.

3.3.1. Automated reading of the XLS files

```
# The pattern of scanning all the files with an .XLS extension.
pattern = '*.XLS'
xls_files = glob.glob(pattern)
```

The list named xls_files contains the names of all the .XLS files in the project directory. The script opens and reads each of the files in the list.

3.3.2. Automated accessing of each sheet of the XLS files

```
sheets = pd.ExcelFile(xls_files[i])
for sheet in sheets.sheet_names:
    print('Handling sheet: ' + str(sheet))
    xls = pd.DataFrame(pd.read_excel(xls_files[i], sheet_name=sheet,
dtype={col: np.float16 for col in float16_cols}))
```

The script accesses each sheet of each .XLS file. Each sheet is read as a Pandas DataFrame [7] data structure. Some columns are converted to a float16 format for reduced memory usage [8].

3.3.3. Determining incorrect timestamps

```
xls['Int Time'] = pd.to_datetime(xls['Time']).astype(np.int64)
error_rows = xls[np.abs(xls['Int Time'] - xls['Int Time'].mean()) > 2 *
xls['Int Time'].std()].index.values
xls = xls.drop(error_rows)
```

A new column named *Int Time* is created in the DataFrame. Timestamps from the default *Time* column are converted to an integer type for easier manipulation. This allows us to determine the mean and standard deviation of time and find an outlier in the form of an incorrect timestamp.

If an absolute value of the difference of **Int Time** and the mean of **Int Time** is greater than the standard deviation of **Int Time** multiplied by 2, there is a high probability of an incorrect timestamp.

Rows containing errors are then dropped.

3.3.4. Determining duplicated rows

```
if last_time is None:
    last_time = xls.loc[xls.index[-1], 'Int Time']
    xls.to_sql('Engine_Room_Database', connection, if_exists='append',
index=True)
else:
    try:
        new_row = xls.index[xls['Int Time'] == last_time]
        xls = xls[new_row[0]+1: -1]
    except:
        pass
    last_time = xls.loc[xls.index[-1], 'Int Time']
    xls.to_sql('Engine_Room_Database', connection, if_exists='append',
index=True)
```

Since exporting of the recorded values was done daily, many of the exported files contain duplicated rows.

The script compares the first value of the *Int Time* of the currently accessed sheet with the last recorded value of the *Int Time* of the previously accessed sheet. The script then slices the currently accessed sheet in such a way that only unique rows are added to the database.

3.3.5. Full code for reading, cleaning, and inserting the data into the SQLite database

```
for i in range(len(xls_files)):
    print('Handling file: ' + str(xls_files[i]))
    sheets = pd.ExcelFile(xls_files[i])
    for sheet in sheets.sheet_names:
        print('Handling sheet: ' + str(sheet))
        xls = pd.DataFrame(pd.read_excel(xls_files[i], sheet_name=sheet,
dtype={col: np.float16 for col in float16_cols}))
        xls['Int Time'] = pd.to_datetime(xls['Time']).astype(np.int64)
```

```
        error_rows = xls[np.abs(xls['Int Time'] - xls['Int Time'].mean()) > 2 *
xls['Int Time'].std()].index.values
        xls = xls.drop(error_rows)
        if last_time is None:
            last_time = xls.loc[xls.index[-1], 'Int Time']
            xls.to_sql('Engine_Room_Database', connection, if_exists='append',
index=True)
        else:
            try:
                new_row = xls.index[xls['Int Time'] == last_time]
                xls = xls[new_row[0]+1: -1]
            except:
                pass
            last_time = xls.loc[xls.index[-1], 'Int Time']
            xls.to_sql('Engine_Room_Database', connection, if_exists='append',
index=True)
```

3.3.6. Testing for duplicated rows

To prove that no duplicated rows have been added to the database, the script reads the entire table and applies an SQL query to get the index value of a duplicated row.

```
query11 = 'SELECT "Time" AS TIME FROM Engine_Room_Database'
df = pd.DataFrame(pd.read_sql_query(query11, connection))
df = df[df.duplicated(keep='first')]

for i in df.index:
    write = 'UPDATE Engine_Room_Database SET duplicates = 1 WHERE ROWID = ' +
str(i) + ';'
    print(write)
```

If a duplicated row exists, Python will print an SQL statement as follows:

```
UPDATE Engine_Room_Database SET duplicates = 1 WHERE ROWID =  + (i) + ;
```

where (i) equals the index value of a duplicated row.

Printed outputs can then be copied and pasted directly to a DB editor [9] as SQL statements for marking of the duplicated rows. Rows with **duplicates = 1** can then be ignored or deleted from the table.

If there are no duplicates there will be no printed output.

3.3.7. Testing for incorrect timestamps

To test for incorrect timestamps, the passage of time is plotted on a line plot. Any deviation from a straight linear increase would point to an error in a timestamp.

```
query = 'SELECT "Int Time" AS Time from Engine_Room_Database'
df = pd.DataFrame(pd.read_sql_query(query, con))

plt.plot(df['Time'])
plt.title('PASSAGE OF TIME')
plt.show()
```

Returns:

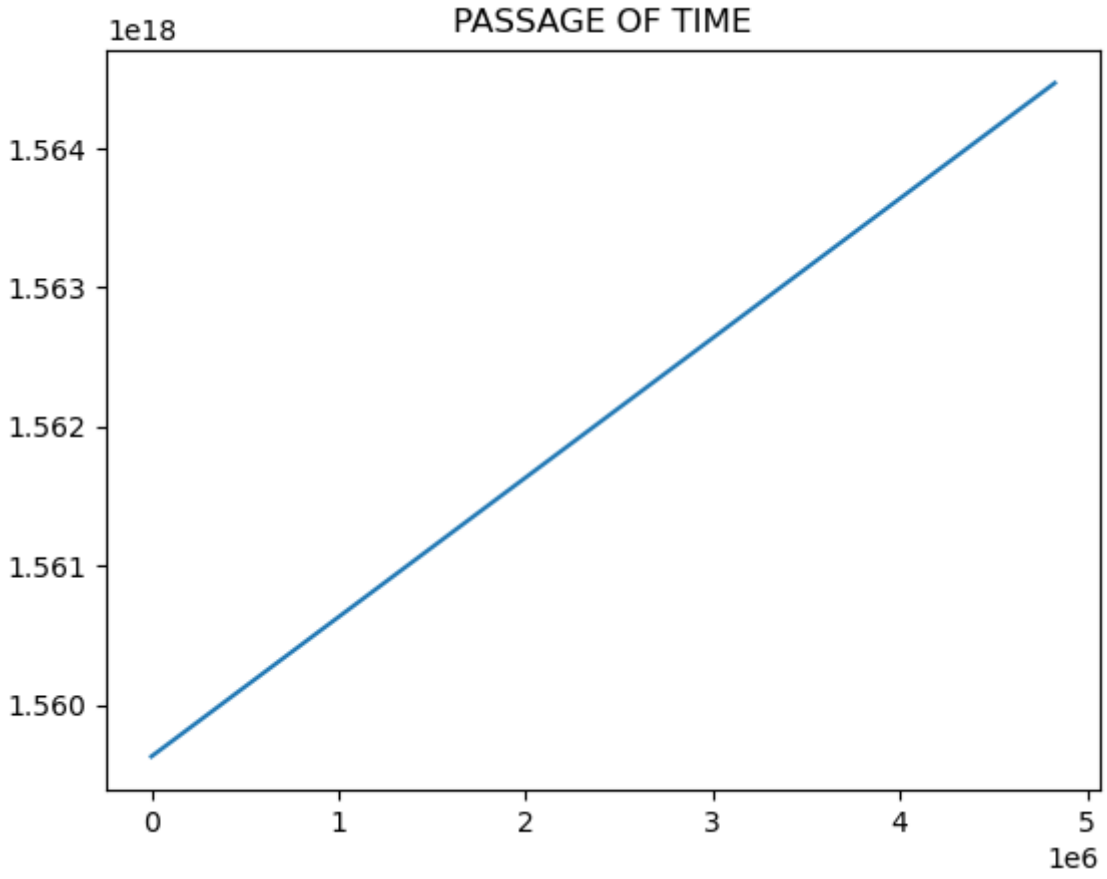

Figure 8. Matplotlib time plot on over four million data points.

Source: Created by the author using the matplotlib library

Figure 8 displays linear passage of time on more than four million rows. The conclusion that the data is clean of any duplicates or incorrect timestamps is reached.

4. PYTHON MODEL OF A VIRTUAL ENGINE ROOM

In the following chapter, the Python code developed for virtually running the engines and simulating engine behaviour under different operational parameters is presented.

Independent variables used as a model input are the recorded total power and the timestamps.

4.1. FUNCTIONS

The entire code is written under the functional programming paradigm [10]. Each function is presented and explained.

4.1.1. Mean load per engine

```
def estimate_load_per_engine(total_power_in_kw):
    return round(total_power_in_kw / current_engines_online / 1830 * 100, 2)
```

Function takes the total power in kW (independent variable) as an argument and estimates the mean load per each running engine. Estimated load is shown as percentage of the maximum nominal load. Factor of 1830 represents the nominal power of the DGs which is 1830 kW.

Returns:

```
LOAD/ENG:52.24
LOAD/ENG:51.91
LOAD/ENG:52.09
LOAD/ENG:51.99
LOAD/ENG:52.17
LOAD/ENG:52.82
LOAD/ENG:52.24
LOAD/ENG:52.68
LOAD/ENG:52.57
LOAD/ENG:52.71
LOAD/ENG:52.31
LOAD/ENG:52.86
```

Figure 9. Screenshot of simulation output estimating the mean load per engine

Source: Created by the author using the Python programming language

4.1.2. Load dependent start

```
def count_to_start(load):
    global start_time, starting_in, current_engines_online, ramping_up
    if load > start_limit and starting_in == 0:
        start_time = start_time - 1
        if start_time < 0:
            start_time = reset_start_time
            starting_in = randint(40, 87)
            return 'ENGINE'
        else:
            return start_time
    elif starting_in != 0:
        starting_in = starting_in - 1
        if starting_in == 0:
            current_engines_online = current_engines_online + 1
            ramping_up = 40
            return starting_in
        else:
            return 'ONLINE IN: ' + str(starting_in)
    else:
        start_time = reset_start_time
        return np.nan
```

Function takes the mean load per engine as an argument, compares it with the START LIMIT parameter and initiates the countdown from the value of START TIME to the value of zero if the mean load is greater than the value of START LIMIT. If countdown to zero is reached, the number of running engines is incremented by 1. Then, a random warm-up period between 40 and 87 seconds is initiated, after which the engine will start to ramp-up.

Returns:

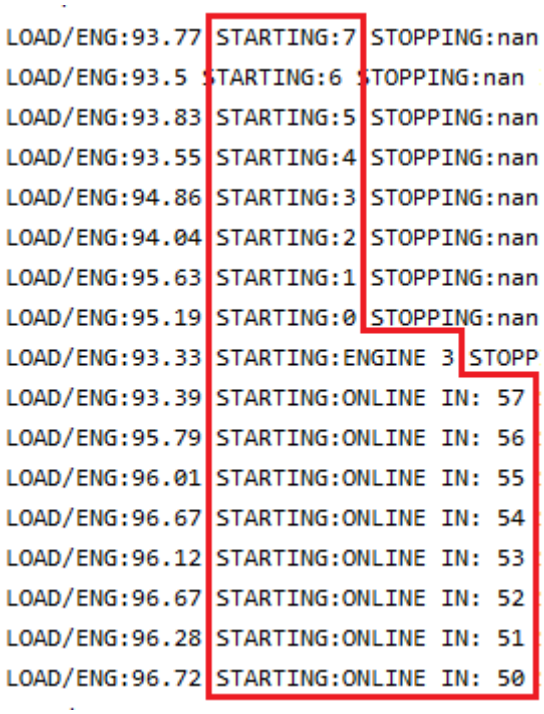

Figure 10. Screenshot of the simulation output showing the countdown, command to start and the warm-up countdown for DG3

Source: Created by the author using the Python programming language

Figure 10 shows the last seven seconds of the countdown, followed by the command to start DG3, followed by the warm-up period randomly chosen at 57 seconds.

Ramping is done so that the mean load per engine is established gradually instead of simultaneously, which makes the simulation more accurate and realistic (see ramping functions below).

4.1.3. Load dependent stop

```
def count_to_stop(engine_power):
    global stop_time, stopping_in, current_engines_online, offline_ramping
    if engine_power / (online_engines - 1) / 1830.0 * 100.0 <= stop_limit and
stopping_in == 0:
        stop_time = stop_time - 1
        if stop_time < 0:
            stop_time = reset_stop_time
            stopping_in = 20
            return 'ENGINE ' + str(current_engines_online+1)
        else:
            return stop_time
```

```
elif stopping_in != 0:
    stopping_in = stopping_in - 1
    if stopping_in == 0:
        current_engines_online = current_engines_online - 1
        return stopping_in
    else:
        offline_ramping = True
        return 'OFFLINE IN: ' + str(stopping_in)
else:
    stop_time = reset_stop_time
    offline_ramping = False
    return np.nan
```

Function takes the total power as an argument and evaluates if the condition presented in chapter 3 is satisfied. If the condition is satisfied, a countdown from the value of STOP TIME to the value of zero is initiated. If the countdown to zero is reached, the number of running engines is decremented by 1.

Returns:

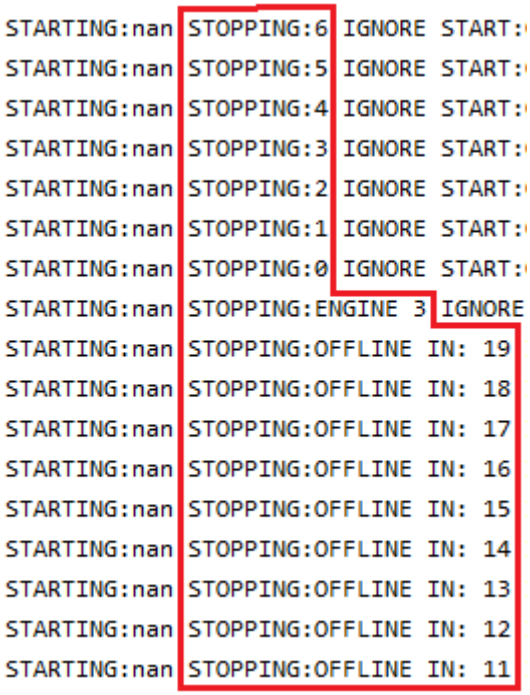

Figure 11. Screenshot of the simulation output showing the countdown, command to stop and the unloading of DG3

Source: Created by the author using the Python programming language

Figure 11 shows the last six seconds of countdown, followed by the command to stop DG3, followed by the unloading of the engine.

Ramping is done so that the mean load per engine is established gradually instead of simultaneously, which makes the simulation more accurate and realistic (see ramping functions below).

4.1.4. Estimated load of each engine

```
def each_engine_load(engines):
    engines_matrix = np.zeros((1, 5))
    engines_matrix[0][0] = round(mean_load_per_engine, 3)
    for j in range(1, engines):
        engines_matrix[0][j] = round(mean_load_per_engine, 3)
    return engines_matrix
```

As an argument, function takes the number of engines that are currently running and creates a Numpy array [11] with the shape of [1, 5] filled with zeros [12]. It then replaces zeros with the mean load per engine for those engines that are currently running. Array positions of engines that are currently not running remain at zero.

For example, if two engines are running at 63.99%, the function would return:

```
array([[63.99, 63.99,  0.  ,  0.  ,  0.  ]])
```

4.1.5. Ramping up of a newly started engine

```
def online_ramp_up(engines):
    global ramping_up, ramp_up_plus_chunk
    pct_per_sec = round(engines[0][current_engines_online - 1] / 20, 3)
    if ramping_up == 0:
        ramp_up_plus_chunk = 0
        return None
    else:
        ramp_up_plus_chunk = ramp_up_plus_chunk + pct_per_sec
        engines[0][current_engines_online - 1] = ramp_up_plus_chunk
        ramping_up = ramping_up - 1
        return round(ramp_up_plus_chunk, 3)
```

As an argument, function takes the Numpy array of engine loads and ramps up the newly started engine gradually. This results in a smooth load increase rather than an instantaneous jump from zero to the mean load per engine.

Returns:

```
EACH ENGINE: [[77.21    77.21     0.      0.    0.   ]]
EACH ENGINE: [[88.338   88.338    2.994   0.    0.   ]]
EACH ENGINE: [[84.746   84.746    5.917   0.    0.   ]]
EACH ENGINE: [[73.402   73.402    8.506   0.    0.   ]]
EACH ENGINE: [[72.639   72.639   11.112   0.    0.   ]]
EACH ENGINE: [[71.129   71.129   13.712   0.    0.   ]]
EACH ENGINE: [[70.094   70.094   16.321   0.    0.   ]]
EACH ENGINE: [[68.407   68.407   18.917   0.    0.   ]]
EACH ENGINE: [[66.947   66.947   21.507   0.    0.   ]]
EACH ENGINE: [[66.197   66.197   24.116   0.    0.   ]]
EACH ENGINE: [[64.51    64.51    26.712   0.    0.   ]]
EACH ENGINE: [[63.537   63.537   29.318   0.    0.   ]]
EACH ENGINE: [[62.13    62.13    31.921   0.    0.   ]]
EACH ENGINE: [[61.153   61.153   34.535   0.    0.   ]]
EACH ENGINE: [[59.846   59.846   37.149   0.    0.   ]]
EACH ENGINE: [[57.787   57.787   39.738   0.    0.   ]]
EACH ENGINE: [[56.699   56.699   42.334   0.    0.   ]]
EACH ENGINE: [[55.401   55.401   44.93    0.    0.   ]]
EACH ENGINE: [[54.059   54.059   47.524   0.    0.   ]]
EACH ENGINE: [[52.01    52.01    50.092   0.    0.   ]]
EACH ENGINE: [[52.64    52.64    52.724   0.    0.   ]]
```

Figure 12. Screenshot of the simulation output showing the effect of ramping up of the newly started engine

Source: Created by the author using the Python programming language

Figure 12 shows a simulated process of DG3 ramping up after it is started. Without the function the mean load per engine would change instantly once the number of the currently running engines is incremented. At the last line of the simulation full load-sharing is established and all loads are equal.

4.1.6. Ramping down of the engines that were already running

```
def online_ramp_down(engines):
    global ramping_up, ramp_up_minus_chunk
    if ramping_up == 0 or ramping_up == 39:
        ramp_up_minus_chunk = 0
        pass
    else:
        ramping_up = ramping_up - 1
```

```
        load_difference = ((mean_load_per_engine * current_engines_online) /
                           (current_engines_online - 1)) - engines[0][0]
        ramp_up_minus_chunk = ramp_up_minus_chunk + (load_difference/20)
        load_difference = round(load_difference - ramp_up_minus_chunk, 3)
        engines[0][0:current_engines_online - 1] =
engines[0][0:current_engines_online - 1] + load_difference
        return load_difference
```

As an argument, function takes the Numpy array of engine loads and calculates the difference between the current load and the estimated mean load under full load-sharing with the newly started engine. The value is then gradually added to the value of Numpy array positions where engines were already running. The result is a gradual decrease in engine load. The simulation is run and the effects of the function are presented:

```
EACH ENGINE: [[77.21    77.21    0.       0.    0.    ]]
EACH ENGINE: [[88.338   88.338   2.994    0.    0.    ]]
EACH ENGINE: [[84.746   84.746   5.917    0.    0.    ]]
EACH ENGINE: [[73.402   73.402   8.506    0.    0.    ]]
EACH ENGINE: [[72.639   72.639   11.112   0.    0.    ]]
EACH ENGINE: [[71.129   71.129   13.712   0.    0.    ]]
EACH ENGINE: [[70.094   70.094   16.321   0.    0.    ]]
EACH ENGINE: [[68.407   68.407   18.917   0.    0.    ]]
EACH ENGINE: [[66.947   66.947   21.507   0.    0.    ]]
EACH ENGINE: [[66.197   66.197   24.116   0.    0.    ]]
EACH ENGINE: [[64.51    64.51    26.712   0.    0.    ]]
EACH ENGINE: [[63.537   63.537   29.318   0.    0.    ]]
EACH ENGINE: [[62.13    62.13    31.921   0.    0.    ]]
EACH ENGINE: [[61.153   61.153   34.535   0.    0.    ]]
EACH ENGINE: [[59.846   59.846   37.149   0.    0.    ]]
EACH ENGINE: [[57.787   57.787   39.738   0.    0.    ]]
EACH ENGINE: [[56.699   56.699   42.334   0.    0.    ]]
EACH ENGINE: [[55.401   55.401   44.93    0.    0.    ]]
EACH ENGINE: [[54.059   54.059   47.524   0.    0.    ]]
EACH ENGINE: [[52.01    52.01    50.092   0.    0.    ]]
EACH ENGINE: [[52.64    52.64    52.724   0.    0.    ]]
```

Figure 13. Screenshot of the simulation output showing the effect of ramping down of the engines that were already running

Source: Created by the author using the Python programming language

Figure 13 shows a simulated process of DG1 and DG2 ramping down after DG3 is started. Without the function, the mean load per engine would change instantaneously with the incrementation of the number of running engines. At the last line of the simulation full load-sharing is established and all loads are equal.

4.1.7. Ramping down due to a load dependent stop

```
def offline_ramp_down(engines):
    global ramp_down_minus_chunk
    pct_per_sec = round(engines[0][current_engines_online - 1] / 20, 3)
    while offline_ramping is False or stop_time != reset_stop_time:
        ramp_down_minus_chunk = 0
        return offline_ramping
    else:
        ramp_down_minus_chunk = ramp_down_minus_chunk - pct_per_sec
        engines[0][current_engines_online - 1] =
engines[0][current_engines_online - 1] + ramp_down_minus_chunk
        if simulate_running[-1][2] != current_engines_online:
            engines[0][current_engines_online - 1] =
engines[0][current_engines_online - 2]
            engines[0][current_engines_online] = 0
            return round(ramp_down_minus_chunk, 3)
```

As an argument, function takes the Numpy array of engine loads, calculates the percentage of the load that must be unloaded each second and subtracts that value from the load of the engine that is stopping. The result is a gradual decrease of the load over a period of 20 seconds.

Returns:

```
EACH ENGINE: [[42.99     42.99     42.99     0.      0. ]]
EACH ENGINE: [[44.5465   44.5465   41.287    0.      0. ]]
EACH ENGINE: [[44.96675  44.96675  38.496    0.      0. ]]
EACH ENGINE: [[46.5905   46.5905   36.868    0.      0. ]]
EACH ENGINE: [[47.1925   47.1925   34.254    0.      0. ]]
EACH ENGINE: [[48.2645   48.2645   32.11     0.      0. ]]
EACH ENGINE: [[49.7465   49.7465   30.346    0.      0. ]]
EACH ENGINE: [[50.44925  50.44925  27.831    0.      0. ]]
EACH ENGINE: [[51.8295   51.8295   25.971    0.      0. ]]
EACH ENGINE: [[52.756    52.756    23.668    0.      0. ]]
EACH ENGINE: [[54.12975  54.12975  21.79     0.      0. ]]
EACH ENGINE: [[54.87525  54.87525  19.309    0.      0. ]]
EACH ENGINE: [[56.289    56.289    17.471    0.      0. ]]
EACH ENGINE: [[57.18825  57.18825  15.133    0.      0. ]]
EACH ENGINE: [[58.6365   58.6365   13.317    0.      0. ]]
EACH ENGINE: [[59.243    59.243    10.694    0.      0. ]]
EACH ENGINE: [[60.7295   60.7295   8.921     0.      0. ]]
EACH ENGINE: [[61.51875  61.51875  6.473     0.      0. ]]
EACH ENGINE: [[61.9625   61.9625   3.725     0.      0  ]]
EACH ENGINE: [[63.918    63.918    2.424     0.      0. ]]
EACH ENGINE: [[65.102    65.102    0.        0.      0. ]]
```

Figure 14. Screenshot of the simulation output showing the effect of ramping down of DG3 due to a load dependent stop

Source: Created by the author using the Python programming language

4.1.8. Ramping up due to a load dependent stop

```
def offline_ramp_up(engines):
    global ramp_down_plus_chunk
    while offline_ramping is False or stop_time != reset_stop_time:
        ramp_down_plus_chunk = 0
        return offline_ramping
    else:
        load_difference = round(((mean_load_per_engine *
current_engines_online) /
                                 (current_engines_online - 1)) - engines[0][0],
3)
        ramp_down_plus_chunk = ramp_down_plus_chunk + (load_difference / 20)
        engines[0][0:current_engines_online - 1] =
engines[0][0:current_engines_online - 1] + ramp_down_plus_chunk
        if simulate_running[-1][2] != current_engines_online:
            engines[0][current_engines_online - 1] =
engines[0][current_engines_online - 1] + ramp_down_plus_chunk
        return round(ramp_down_plus_chunk, 3)
```

As an argument, function takes the Numpy array of engine loads, calculates the difference that must be added to the engines that will continue running after the load-dependent stop and adds that difference to the array positions of the running engines. The result is a gradual increase in load over a period of 20 seconds.

Returns:

```
EACH ENGINE: [[42.99     42.99     42.99     0.     0. ]]
EACH ENGINE: [[44.5465   44.5465   41.287    0.     0. ]]
EACH ENGINE: [[44.96675  44.96675  38.496    0.     0. ]]
EACH ENGINE: [[46.5905   46.5905   36.868    0.     0. ]]
EACH ENGINE: [[47.1925   47.1925   34.254    0.     0. ]]
EACH ENGINE: [[48.2645   48.2645   32.11     0.     0. ]]
EACH ENGINE: [[49.7465   49.7465   30.346    0.     0. ]]
EACH ENGINE: [[50.44925  50.44925  27.831    0.     0. ]]
EACH ENGINE: [[51.8295   51.8295   25.971    0.     0. ]]
EACH ENGINE: [[52.756    52.756    23.668    0.     0. ]]
EACH ENGINE: [[54.12975  54.12975  21.79     0.     0. ]]
EACH ENGINE: [[54.87525  54.87525  19.309    0.     0. ]]
EACH ENGINE: [[56.289    56.289    17.471    0.     0. ]]
EACH ENGINE: [[57.18825  57.18825  15.133    0.     0. ]]
EACH ENGINE: [[58.6365   58.6365   13.317    0.     0. ]]
EACH ENGINE: [[59.243    59.243    10.694    0.     0. ]]
EACH ENGINE: [[60.7295   60.7295   8.921     0.     0. ]]
EACH ENGINE: [[61.51875  61.51875  6.473     0.     0. ]]
EACH ENGINE: [[61.9625   61.9625   3.725     0.     0  ]]
EACH ENGINE: [[63.918    63.918    2.424     0.     0. ]]
EACH ENGINE: [[65.102    65.102    0.        0.     0. ]]
```

Figure 15. Screenshot of the simulation output showing the effect of ramping up of DG1 and DG2 due to a load dependent stop of DG3

Source: Created by the author using the Python programming language

4.1.9. Immediate start due to high load

```
def detect_high_load(engines):
    global current_engines_online, starting_in, high_load_counter
    if engines[0][0] > 100:
        current_time = current_engines_online * 1
        high_load_counter += current_time
        high_load_detected = True
        starting_in = randint(40, 87)
        return high_load_detected
    else:
        high_load_detected = False
        return high_load_detected
```

As an argument, function takes the Numpy array of engine loads and issues a direct command to start an additional engine if the load on any of the engines becomes greater than 100%. Note that in this case there is no countdown for determining if the start is required and that all other programmed logic is ignored. The engine immediately goes to the warm-up stage, after which it synchronises and engages in full load-sharing.

Function also calculates the total time spent in zones where load is greater than 100%. Note that the load of 100% is the nominal load of the DGs, but that the engines can run at load greater than 100% when a sudden surge in power occurs.

Returns:

```
STARTING:nan    EACH ENGINE:[[70.87     70.87   0.    0.    0.   ]] HIGH LOAD:False
STARTING:nan    EACH ENGINE:[[71.64     71.64   0.    0.    0.   ]] HIGH LOAD:False
STARTING:nan    EACH ENGINE:[[83.99     83.99   0.    0.    0.   ]] HIGH LOAD:False
STARTING:14     EACH ENGINE:[[97.81     97.81   0.    0.    0.   ]] HIGH LOAD:False
STARTING:nan    EACH ENGINE:[[69.67     69.67   0.    0.    0.   ]] HIGH LOAD:False
STARTING:nan    EACH ENGINE:[[70.71     70.71   0.    0.    0.   ]] HIGH LOAD:False
STARTING:14     EACH ENGINE:[[86.12     86.12   0.    0.    0.   ]] HIGH LOAD:False
STARTING:13     EACH ENGINE:[[100.16    100.16  0.    0.    0.   ]] HIGH LOAD:True
ONLINE IN: 78   EACH ENGINE:[[82.84     82.84   0.    0.    0.   ]] HIGH LOAD:False
ONLINE IN: 77   EACH ENGINE:[[83.61     83.61   0.    0.    0.   ]] HIGH LOAD:False
ONLINE IN: 76   EACH ENGINE:[[69.62     69.62   0.    0.    0.   ]] HIGH LOAD:False
ONLINE IN: 75   EACH ENGINE:[[69.23     69.23   0.    0.    0.   ]] HIGH LOAD:False
```

Figure 16. Screenshot of the simulation output showing the detection of load greater than 100%

Source: Created by the author using the Python programming language

Figure 16 shows an output of the simulation during the time of load greater than 100%. Boolean value changes to True, after which a random warm-up period of 78 seconds begins immediately.

4.1.10. Counting the total number of starts and stops

```
def count_change_in_number_of_engines(online_engines):
    global change_no_engines
    if online_engines != current_engines_online:
        change_no_engines += 1
        return change_no_engines
    else:
        return change_no_engines
```

Function takes the number of running engines as an argument and then counts how many times the number of running engines has changed since the beginning of the simulation.

Returns:

```
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
CHANGE NO. ENGINES:5
```

Figure 17. Screenshot of the simulation output showing that the number of running engines has changed five times since the beginning of the simulation.

Source: Created by the author using the Python programming language

4.1.11. Calculating the total running hours

```
def count_running_hours(online):
    global running_hours
    running_hours = online + running_hours
    return round((running_hours / 3600), 2)
```

Function takes the number of running engines as an argument and calculates the total combined running hours for all engines in the facility.

Returns:

```
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.32
RUNNING HOURS:204.33
RUNNING HOURS:204.33
```

Figure 18. Screenshot of the simulation output showing the total combined running hours since the beginning of the simulation.

Source: Created by the author using the Python programming language

4.1.12. Calculating the fuel consumption

```
def get_closest_consumption(load_list, engine_load):
    global consumed
    for j in range(len(engine_load[0])):
        closest_load = min(load_list, key=lambda x: abs(x - engine_load[0, j]))
        closest_consumption = [j for j, x in enumerate(load_list) if x ==
closest_load][0]
        consumed = consumed + (consumption_per_h_gallons[closest_consumption] /
3600 * 3.78541)
    return round(consumed, 2)
```

Function takes two arguments: load_list, engine_load

The first argument is a list containing loads given by the equipment manufacturer. Each load in the first list corresponds to the fuel consumption from the second list. The lists are presented:

```
load_list = [0.1, 10, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90,
100]

consumption_per_h_gallons = [0, 21.9, 32.5, 38.0, 43.5, 49.25, 55.0, 60.95, 66.9,
72.95, 79.0,  84.75, 90.5, 96.4, 102.5, 108.2, 113.9, 124.6]
```

Function then iterates over the Numpy array of the running engines. For each load in the array, the function finds its closest load in the first list and the corresponding consumption for that load in the second list.

For example, if the DG load is 48%, the closest load is 50% and the corresponding fuel consumption is 66.9 gallons/h.

Returns:

```
CONSUMPTION:1970.0
CONSUMPTION:1970.21
CONSUMPTION:1970.42
CONSUMPTION:1970.64
CONSUMPTION:1970.85
CONSUMPTION:1971.06
CONSUMPTION:1971.27
CONSUMPTION:1971.48
CONSUMPTION:1971.69
CONSUMPTION:1971.9
CONSUMPTION:1972.11
CONSUMPTION:1972.32
CONSUMPTION:1972.54
CONSUMPTION:1972.75
CONSUMPTION:1972.96
CONSUMPTION:1973.17
```

Figure 19. Screenshot of the simulation output showing the total fuel consumption since the beginning of the simulation.

Source: Created by the author using the Python programming language

4.1.13. Sample of a simulation printout

The printed output of all the functions in the program for a period of five seconds is presented:

2019-06-27 12:35:48.328000 POWER:2554.0 ONLINE:2 CHANGE NO. ENGINES:0 RUNNING HOURS:0.03 MEAN LOAD/ENG:69.78 STARTING:ONLINE IN: 34 STOPPING:nan STARTING IN:34 STOPPING IN:0 EACH ENGINE:[[69.78 69.78 0. 0. 0.]] HIGH LOAD:False RAMP DOWN:None RAMP UP:None OFFLINE UP:False OFFLINE DOWN:False CONSUMPTION:9.82

2019-06-27 12:35:49.328000 POWER:2534.0 ONLINE:2 CHANGE NO. ENGINES:0 RUNNING HOURS:0.03 MEAN LOAD/ENG:69.23 STARTING:ONLINE IN: 33 STOPPING:nan STARTING IN:33 STOPPING IN:0 EACH ENGINE:[[69.23 69.23 0. 0. 0.]] HIGH LOAD:False RAMP DOWN:None RAMP UP:None OFFLINE UP:False OFFLINE DOWN:False CONSUMPTION:10.01

2019-06-27 12:35:50.328000 POWER:2580.0 ONLINE:2 CHANGE NO. ENGINES:0 RUNNING HOURS:0.03 MEAN LOAD/ENG:70.49 STARTING:ONLINE IN: 32 STOPPING:nan STARTING IN:32 STOPPING IN:0 EACH ENGINE:[[70.49 70.49 0. 0. 0.]] HIGH LOAD:False RAMP DOWN:None RAMP UP:None OFFLINE UP:False OFFLINE DOWN:False CONSUMPTION:10.2

2019-06-27 12:35:51.343000 POWER:2576.0 ONLINE:2 CHANGE NO. ENGINES:0 RUNNING HOURS:0.03 MEAN LOAD/ENG:70.38 STARTING:ONLINE IN: 31 STOPPING:nan STARTING IN:31 STOPPING IN:0 EACH ENGINE:[[70.38 70.38 0. 0. 0.]] HIGH LOAD:False RAMP DOWN:None RAMP UP:None OFFLINE UP:False OFFLINE DOWN:False CONSUMPTION:10.39

2019-06-27 12:35:52.343000 POWER:2560.0 ONLINE:2 CHANGE NO. ENGINES:0 RUNNING HOURS:0.03 MEAN LOAD/ENG:69.95 STARTING:ONLINE IN: 30 STOPPING:nan STARTING IN:30 STOPPING IN:0 EACH ENGINE:[[69.95 69.95 0. 0. 0.]] HIGH LOAD:False RAMP DOWN:None RAMP UP:None OFFLINE UP:False OFFLINE DOWN:False CONSUMPTION:10.58

5. BENCHMARK TESTS ON THE MODEL

In the following part of the study, benchmark tests on the model are performed in order to determine if the model accurately mimics the conditions, values and behaviours measured in the real world.

5.1. METHODOLOGY

SQL queries to fetch data from the database and then store the fetched data in a Pandas DataFrame are written. Matplotlib is then used to draw graphs. The first graph represents the change in the recorded physical quantities, while the second represents the quantities e stimated by the simulation output.

The quality of the model is evaluated by comparing the two graphs.

5.1.1. Testing on a small scale

The simulation is run on 200 seconds during the time of a load-dependent start. The load of the engine that was already running is plotted. Expected behaviour: increase in load, resulting in a load-dependent start, followed by a gradual decrease of the load as another engine is ramped-up and begins to share the load.

```
query = 'SELECT "TOTAL POWER" AS TOT_POW, "REQUIRED ENGINES" AS ENGINES,
"Time", "DG1 POWER [%]" AS DG1 from Engine_Room_Database LIMIT 200 OFFSET
18050'
df = pd.DataFrame(pd.read_sql_query(query, con))

# LINE SUBPLOTS OF DG1 MEASURED AND DG1 ESTIMATED LOAD
plt.subplot(2, 1, 1)
plt.plot(df['DG1'])
plt.title('DG1 LOAD MEASURED')
plt.subplot(2, 1, 2)
plt.plot([item[3][0,0] for item in simulate_running])
plt.title('DG1 LOAD ESTIMATED')
plt.show()
```

Returns:

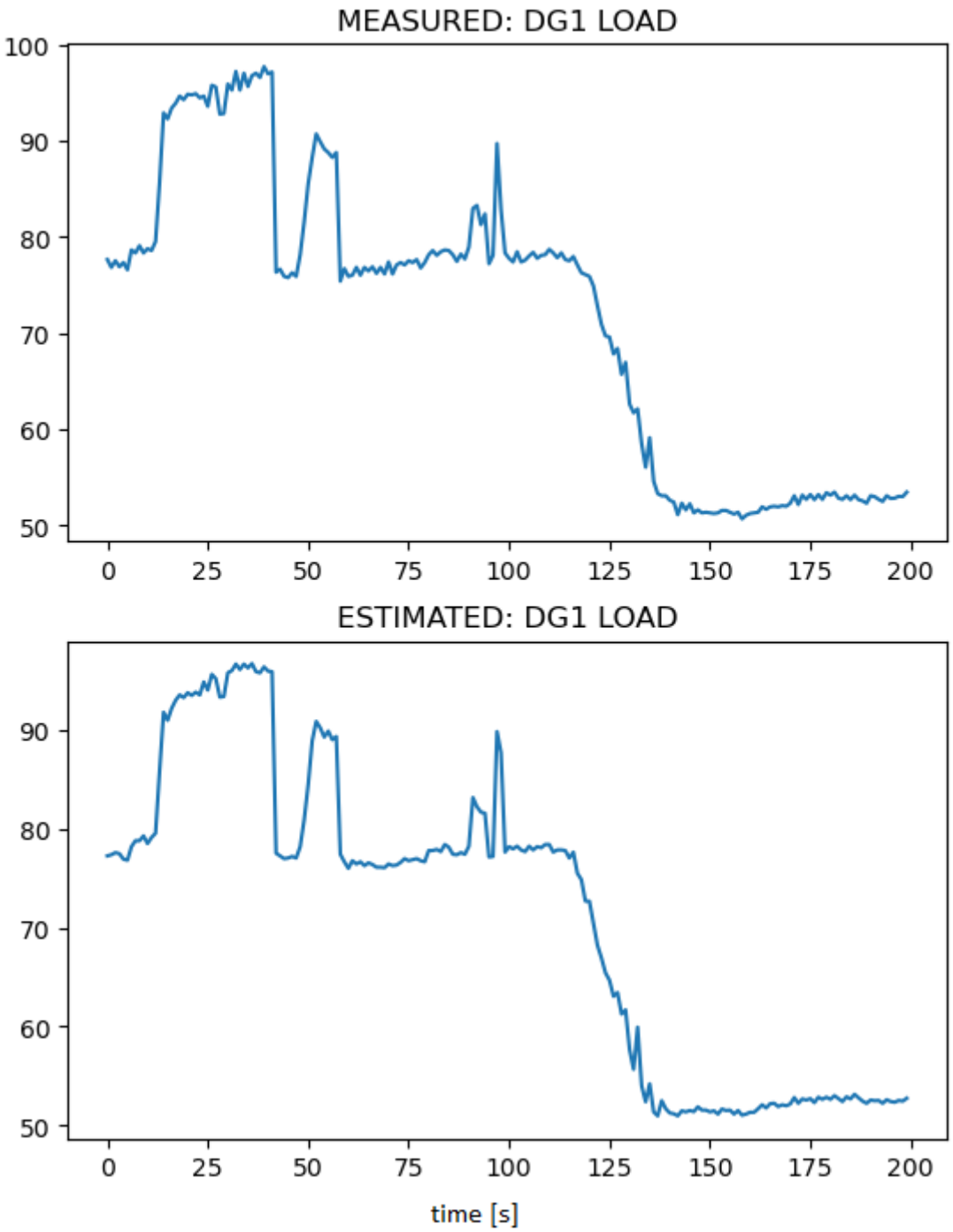

Figure 20. Comparison of measured and estimated load on DG1 before and after the load dependent start

Source: Created by the author using the matplotlib library

Figure 20 displays successful recreation of the measured conditions. All characteristic points are observable and there is no significant deviation from the measured data. The conclusion that the test is successful is reached.

5.1.2. Testing on a medium scale

The simulation is run on 20000 seconds. DG1 measured and estimated load is plotted.

```
query = 'SELECT "TOTAL POWER" AS TOT_POW, "REQUIRED ENGINES" AS ENGINES,
"Time", "DG1 POWER [%]" AS DG1 from Engine_Room_Database LIMIT 20000 OFFSET
18050'
df = pd.DataFrame(pd.read_sql_query(query, con))

# LINE SUBPLOTS OF DG1 MEASURED AND DG1 ESTIMATED LOAD
plt.subplot(2, 1, 1)
plt.plot(df['DG1'])
plt.title('DG1 LOAD MEASURED')
plt.subplot(2, 1, 2)
plt.plot([item[3][0,0] for item in simulate_running])
plt.title('DG1 LOAD ESTIMATED')
plt.show()
```

Returns:

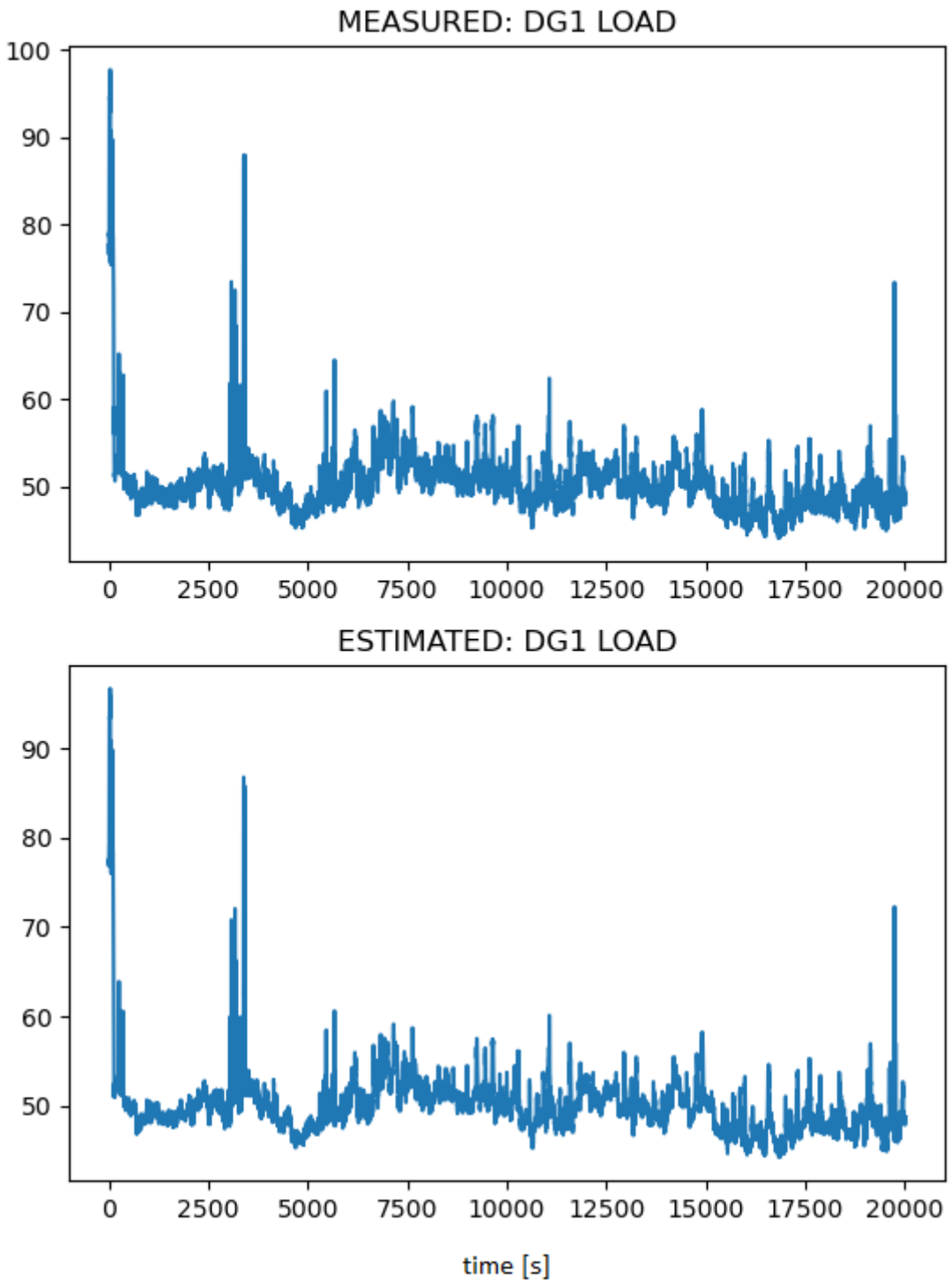

Figure 21. Comparison of measured and estimated load on DG1 for a period of 20000 seconds.

Source: Created by the author using the matplotlib library

Figure 21 displays successful recreation of the measured conditions. All characteristic points are observable and there is no significant deviation from the measured data.

The model is tested again by observing the measured and estimated change in the number of running engines on a period of 250000 seconds.

```
# LINE SUBPLOTS OF MEASURED AND ESTIMATED CHANGE IN NUMBER OF ENGINES
plt.subplot(2, 1, 1)
plt.plot(df['ENGINES'])
plt.title('CHANGE IN NUMBER OF ENGINES MEASURED')
plt.subplot(2, 1, 2)
plt.plot([item[2] for item in simulate_running])
plt.title('CHANGE IN NUMBER OF ENGINES ESTIMATED')
plt.show()
```

Returns:

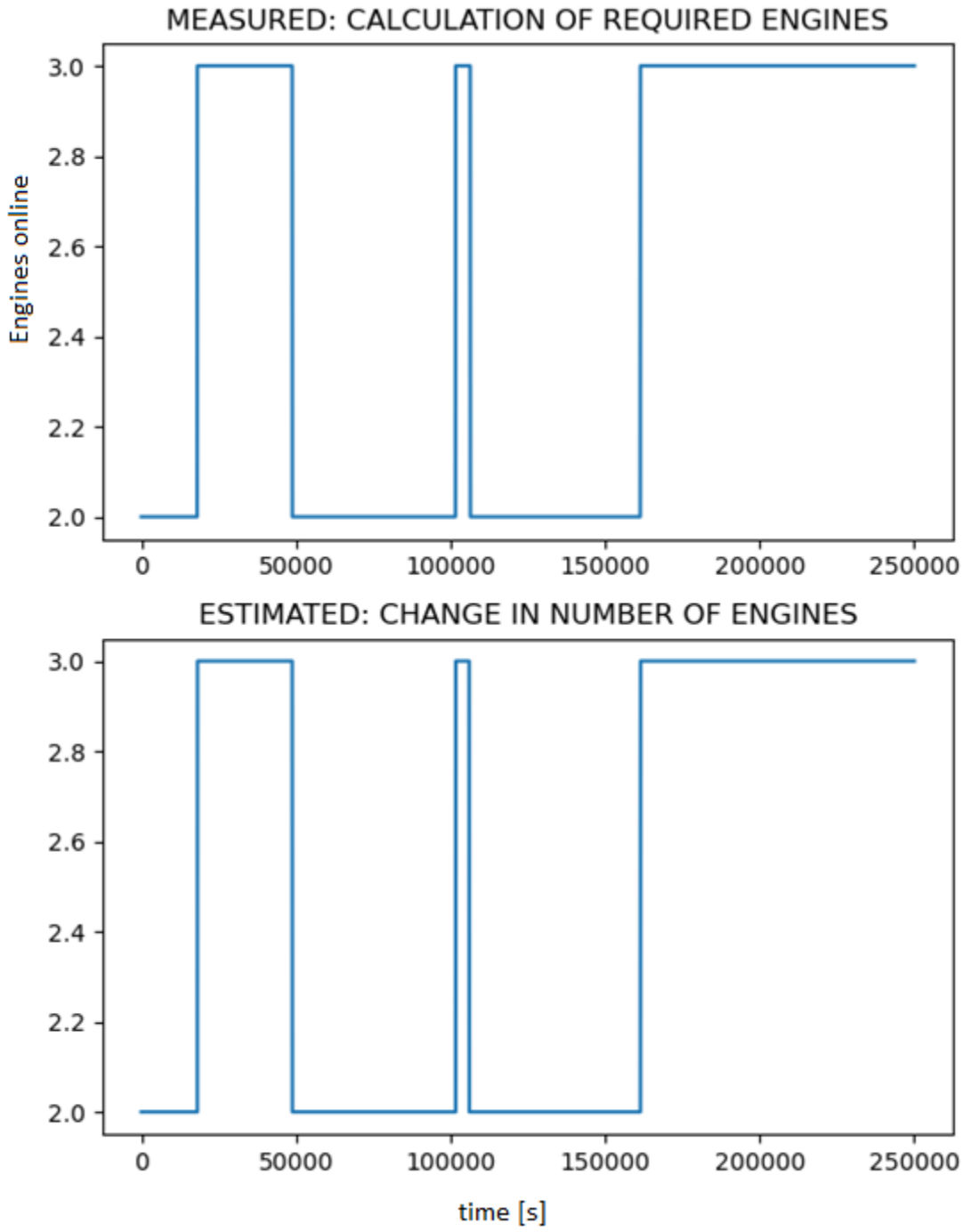

Figure 22. Comparison of measured and estimated change in the number of running engines for a period of 250000 seconds

Source: Created by the author using the matplotlib library

Figure 22 displays a successful recreation of the measured conditions. All characteristic points are observable and there is no significant deviation from the measured data.

5.1.3. Testing on a large scale

The simulation is run on 1 million seconds. Mean load per engine in the form of a histogram with a logarithmic y axis is observed.

Note that the recorded values are incrementing the **'ENGINES'** column as soon as the command to start is executed. Even if the engine is in the warm-up stage and is not yet running the system considers that engine to be running.

Because of this, the histogram shows somewhat less time spent under high load, as the total power is divided by the value in the **'ENGINES'** column immediately after a start command is issued by the PMS.

For example:

Table 2. Comparison of recorded and estimated DG loads

	Total Power [kw]	DG1 [%]	DG2 [%]	DG3 [%]	Mean load [%]
Recorded	4000	109%	109%	Warming-up	72.8%
Model	4000	109%	109%	Warming-up	109%

Source: Created by the author using the MS Word software

```
query = 'SELECT "TOTAL POWER" AS TOT_POW, "REQUIRED ENGINES" AS ENGINES,
"Time", "DG1 POWER [%]" AS DG1, "DG2 POWER [%]" AS DG2, "DG3 POWER [%]" AS
DG3,"DG4 POWER [%]" AS DG4,"DG5 POWER [%]" AS DG5 from Engine_Room_Database
LIMIT 1000000'
df = pd.DataFrame(pd.read_sql_query(query, con))

# MEAN LOAD PER ENGINE MEASURED AND ESTIMATED
plt.subplot(2, 1, 1)
df['MEAN POWER'] = df['TOT_POW'] / df['ENGINES'] / 1830 * 100
arr = plt.hist(df['MEAN POWER'], bins=bins, log=True, range=[-1, 120])
for i in range(bins):
    plt.text(arr[1][i], arr[0][i], str(arr[0][i]))
plt.title('MEASURED: MEAN LOAD PER ENGINE')
plt.subplot(2, 1, 2)
data = [[item[4] for item in simulate_running]]
arr = plt.hist(data, bins=bins, log=True, range=[-1, 120])
for i in range(bins):
    plt.text(arr[1][i], arr[0][i], str(arr[0][i]))
plt.title('ESTIMATED: MEAN LOAD PER ENGINE')
plt.show()
```

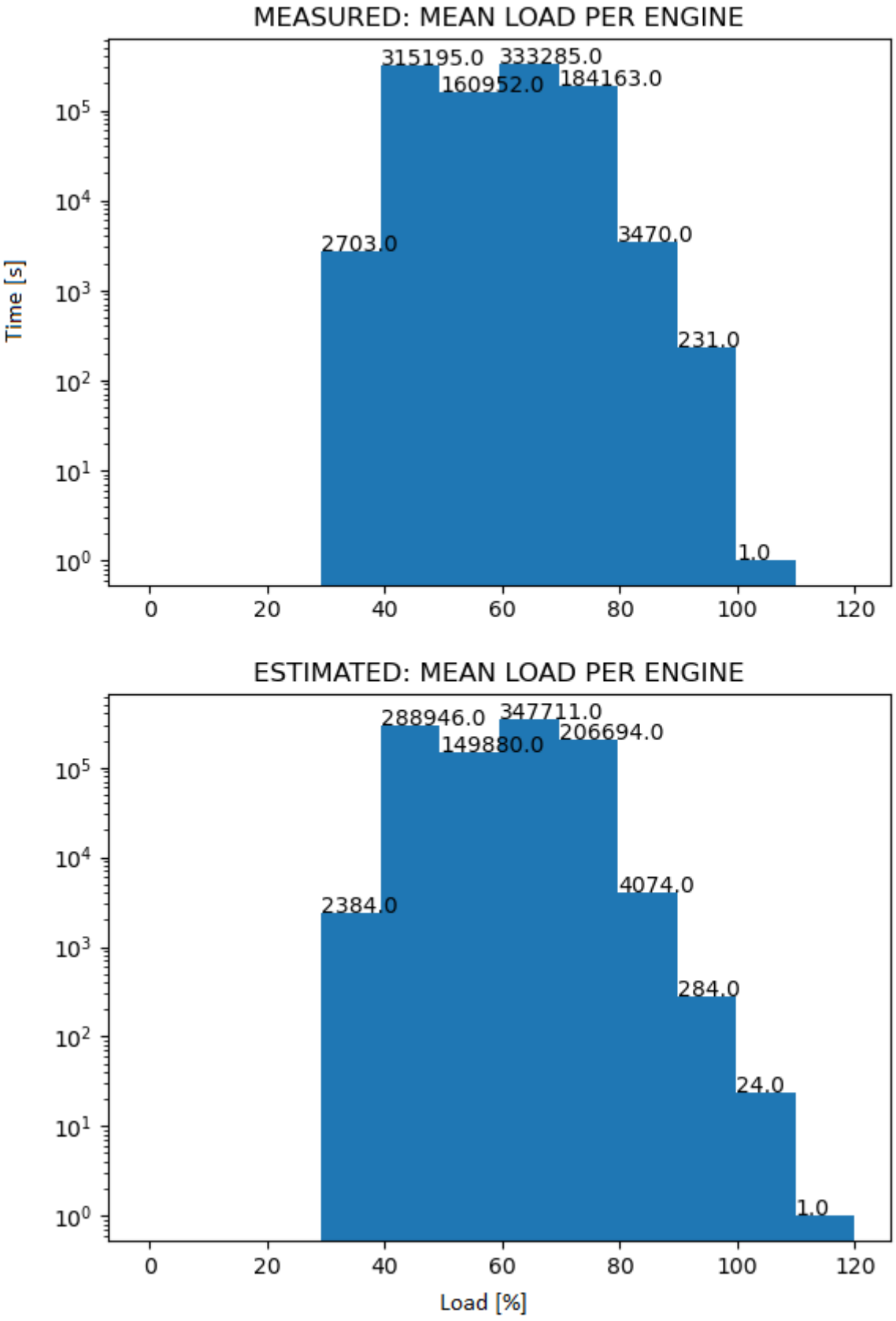

Figure 23. Comparison of measured and estimated mean load per engine for a period of 1 million seconds

Source: Created by the author using the matplotlib library

For the first time, a significant deviation between measured and estimated values is observed. The histogram of measured values presents only one second of work at a load higher than 100%, while the model presents a total of 25 seconds spent in load higher than 100%. The fact that the recorded values for the number of running engines are incremented by one as soon as the command to start is executed does not explain the difference in loads greater than 100% on the histograms.

If a non-logarithmic y axis were chosen, the shortest bars would be insignificant in size compared to the highest bars of the histogram and could not be seen by the naked eye. Nonetheless, by choosing a logarithmic axis and researching the difference between the histograms, an insight into a specific problem associated with the offshore oil and gas drilling operations, particularly the *pipe tripping* and POOH (*pulling out of the hole*) operations is reached.

This directly leads us into the most important part of this study where the author explains what *pipe tripping* and *POOH* is, why does the developed model estimate the tripping operation differently than that which was recorded and how does the tripping operation affect the performance of the PMS and the drilling facility in particular.

A solution to the specific problem of pipe tripping in the form of the change of operational parameters from their commissioning values to the optimised values will be proposed.

6. THE IMPACT OF PIPE TRIPPING AND POOH OPERATIONS ON THE PMS AND THE DRILLING FACILITY

In the following chapter, the particulars of pipe tripping and POOH and their impact on the drilling rig are explained.

Note that the term tripping in this study is not associated with the term tripping in context of electrical engineering, such as the tripping of the circuit breakers due to an overload.

6.1. EXPLANATION OF TRIPPING AND POOH

Pipe tripping (or "making a round trip" or simply "making a trip") is the physical act of pulling the drill string out of the wellbore and then running it back in [13]. The movements are achieved by moving the traveling block up and down the derrick by using drawworks to hoist the traveling block up or to lower the travelling block down.

POOH is the physical act of pulling the drill string out of the wellbore in a controlled manner [14].

Both tripping and POOH are done in accordance to well control practices to minimise the possibility of inducing a kick [15] from the wellbore.

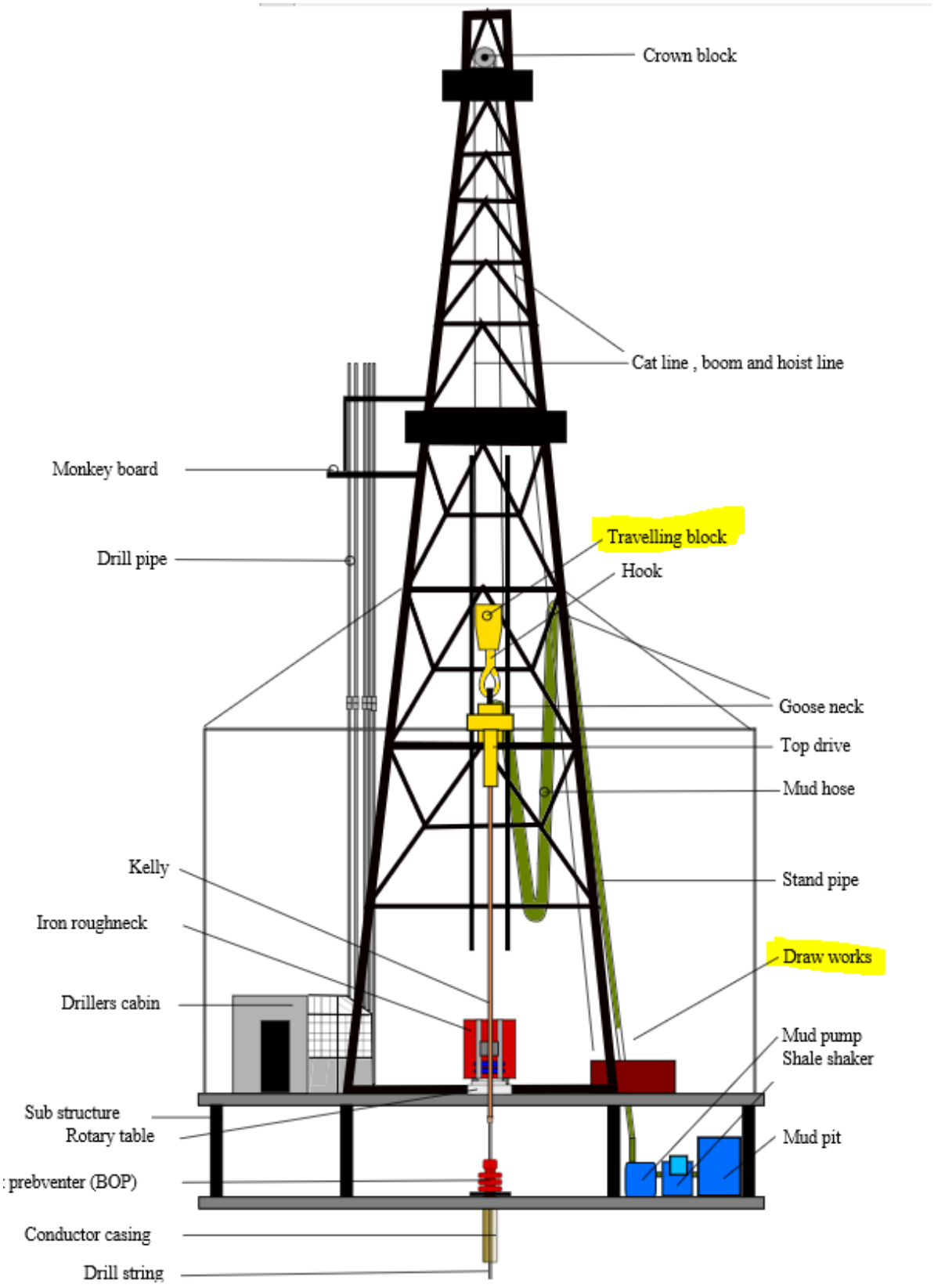

Figure 24. Representation of a drilling derrick. Drawworks and the traveling block are highlighted in yellow

Source: Drilling equipment (2020, June.) *Energy Faculty.* Available at: https://energyfaculty.com/drilling-equipment/. (23. 04. 2020.)

Figure 24 shows a typical drilling derrick of a modern offshore drilling rig. The traveling block is connected to the drawworks drum by a wire rope. Spooling of the drawworks drum results in the travelling block movement along the derrick.

The drawworks uses several 3PH AC motors driven by the Drilling VFDs to hoist the traveling block. Each hoisting sequence results in a surge of power, followed by sharp ramping of the engine load. Once the block is hoisted to the top of the drilling derrick the demand for power plummets as the equipment stands still, until a new tripping sequence is initiated. Note that the time required for hoisting of the traveling block is much lower than the time defined in the START TIME parameter.

This creates two undesirable situations for the DGs, PMS and the rig:

- The surge in power is so quick and sharp that the START TIME parameter rarely gets enough time spent in the START LIMIT zone to achieve a successful countdown to zero. Countdown to zero is never reached because the time required for hoisting the block to the top of the derrick is shorter than the time defined by the START TIME parameter. In other words, there is no time for a load-dependent start. Instead, an immediate high-load start is initiated because the engines are ramped to more than 100% load. By the time the newly started engine is synchronised, the traveling block is already hoisted to the top of the derrick and there is no requirement for that additional power.
- As soon as the traveling block is positioned at the top of the drilling derrick, most of the electrical equipment is at standstill. An additional engine is now running due to a high-load start, so the countdown to load-dependent stop is initiated. Should the equipment stay at standstill for long enough the countdown to stop will be reached and the engine that was just started will be shut down.
- To simplify – the system starts an engine only to compensate for a few seconds of peak demand. By the time the engine is synchronised the peak demand is already gone. As the peak demand is gone, the mean load per engine gets undesirably low so the load-dependent stop is initiated.
- Once the crew is ready to handle the next stand of the drill string [17], the same cycle repeats, meaning that for each stand of the drill string one engine will be started and then stopped.

The mentioned situations result in several problems:

- Instability of the Main SWBD voltage and frequency due to numerous sudden surges in required power.
- Ramping of the engines to +100% load, causing intermittent starting and stopping of the engines.
- Overheating, using of excess starting air, increased vibrations and noise.
- Stress on the Main SWBD switchgear.

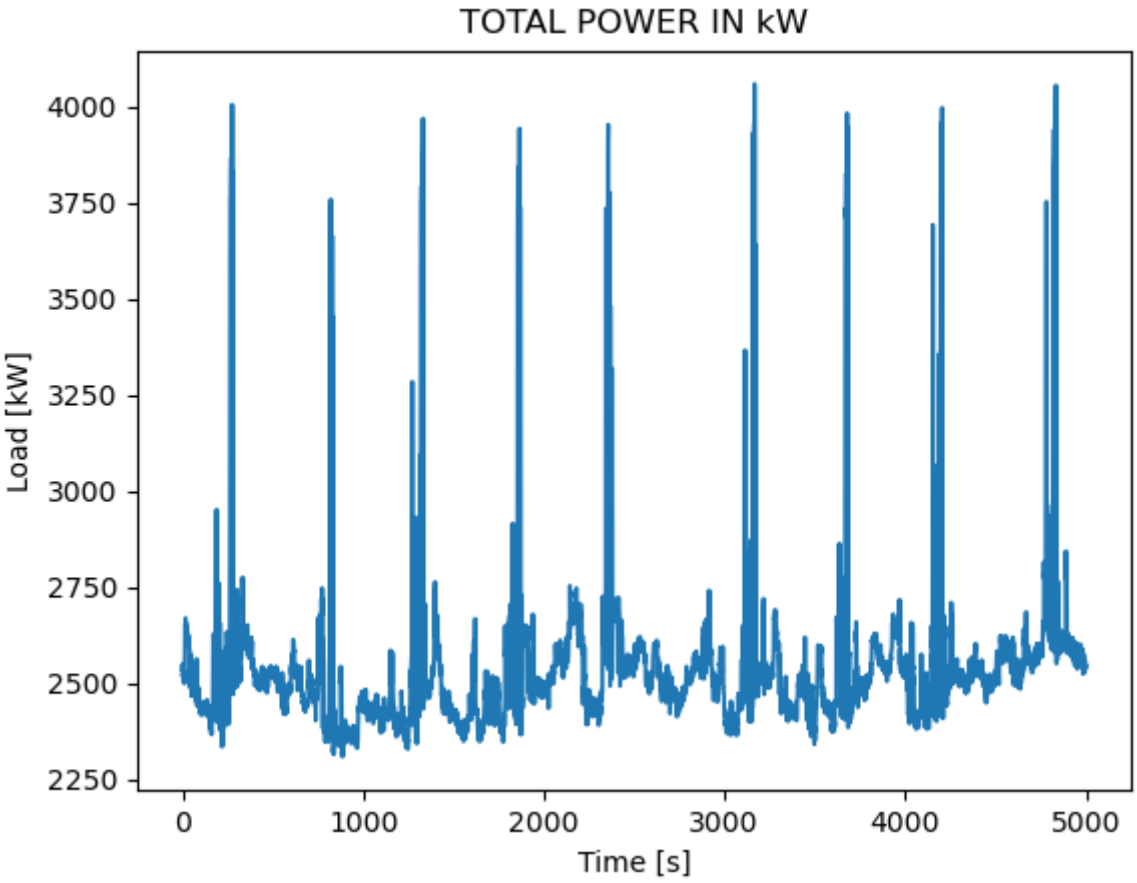

Figure 25. Change in total power during tripping operations over a period of 5000 seconds

Source: Created by the author using the matplotlib library

Figure 25 represents the sudden surges in total power associated with hoisting of the traveling block. Base load is established around 2500 kW, while the peak demand surges to 4000 kW.

Evaluation on how the change in the load would affect the engines in case only two engines were running during the operations:

For baseload:

- $\frac{\frac{2500}{2}}{1830} * 100 = 68.3\ \%$

For peak load:

- $\frac{\frac{4000}{2}}{1830} * 100 = 109.2\ \%$

This demonstrates that the peak load is sufficient to ramp the DGs to loads higher than 100%, resulting in an immediate high-load start.

Measurement of the average time between the peak demands:

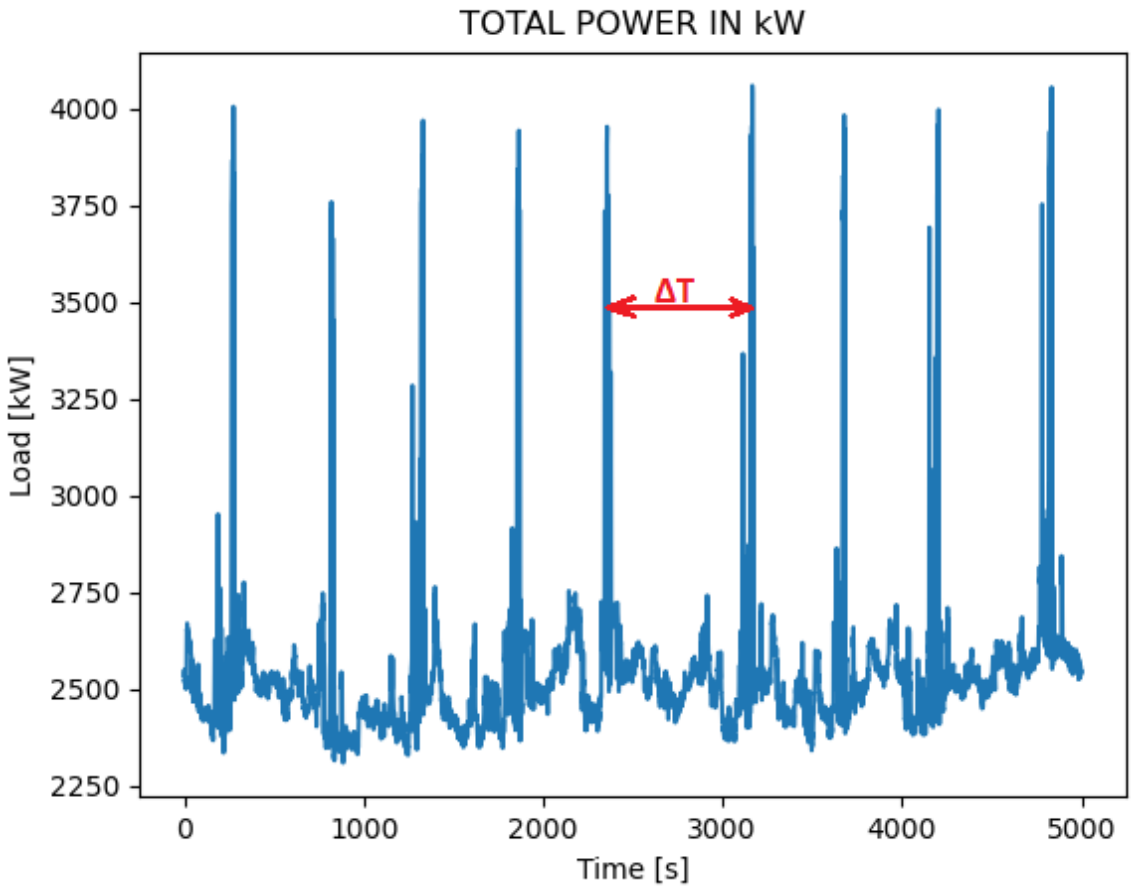

Figure 26. Time between two peak loads during tripping operations over a period of 5000 seconds

Source: Created by the author using the matplotlib library

Figure 26 presents the time difference between two peak loads, approximated at 650 seconds. This value plays an important role in the later part of the study where simulations are run under different operational parameters.

6.1.1. Examples of tripping and POOH operations on the performance of the DGs

Author demonstrates the effect by running simulations with the commissioning parameters.

```
query5 = 'SELECT "TOTAL POWER" AS TOT_POW, "Time", "START TIMER" AS ST,
"REQUIRED ENGINES" AS ENGINES from Engine_Room_Database LIMIT 25000 OFFSET
1720000'
df = pd.DataFrame(pd.read_sql_query(query5, con))
power_list = df[['TOT_POW', 'Time', 'ST']].to_numpy()

start_limit = 80  # Given in [%]
start_time = 10  # Given in [s]
stop_limit = 70  # Given in [%]
stop_time = 200  # Given in [s]

#  LINE PLOTS: MEASURED TOTAL POWER, ESTIMATED STOP TIME, ESTIMATED CHANGE IN
NUMBER OF ENGINES
plt.subplot(3, 1, 1)
plt.plot(df['TOT_POW'])
plt.title('TOTAL POWER IN kW')
plt.subplot(3, 1, 2)
plt.plot([item[-1] for item in simulate_running])
plt.title('ESTIMATED: STOP TIME')
plt.subplot(3, 1, 3)
plt.plot([item[2] for item in simulate_running])
plt.title('ESTIMATED: CHANGE IN NUMBER OF ENGINES')
plt.show()
```

Recorded total power, estimated countdown times and estimated change in the number of running engines over 25000 seconds are plotted below.

Returns:

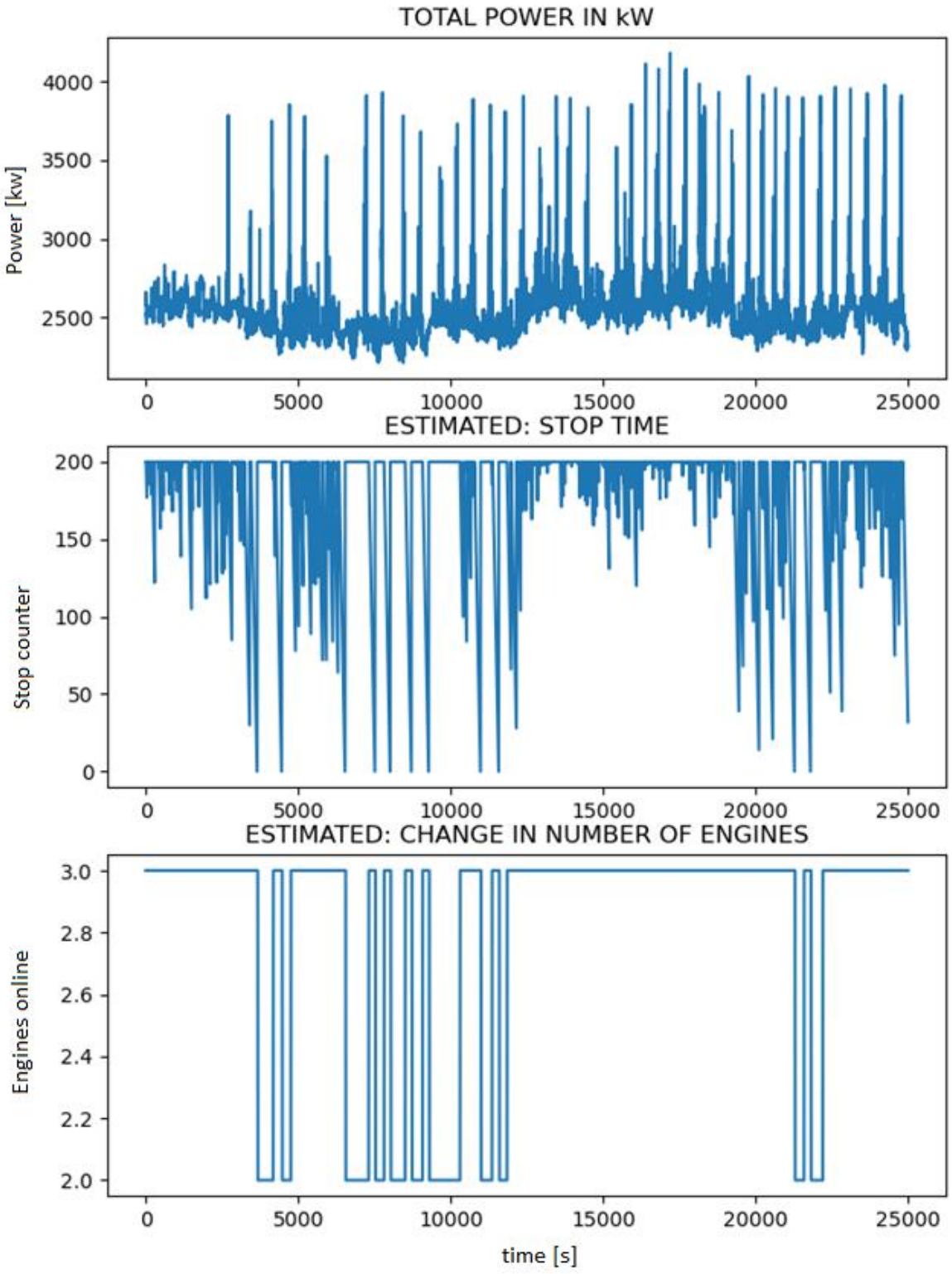

Figure 27. The effect of pipe tripping on DG behaviour

Source: Created by the author using the matplotlib library

Figure 27 presents the effect of pipe tripping on DG behaviour for a period of 25000 seconds. The top subplot displays the recorded total power in kW, the middle subplot displays a

countdown to a load-dependent stop, while the bottom subplot displays the change in the number of running engines, constantly fluctuating between two and three.

This demonstrates a strong correlation between the tripping operation and the change of the number of running engines.

In this simulation, the model estimated the total change in number of running engines to 22. Over a period of 25000 seconds, this averages down to one change every 19 minutes. A total of 86 seconds spent at load great than 100% is estimated.

Another simulation:

```
query5 = 'SELECT "TOTAL POWER" AS TOT_POW, "Time", "START TIMER" AS ST,
"REQUIRED ENGINES" AS ENGINES from Engine_Room_Database LIMIT 28000 OFFSET
3122500'
df = pd.DataFrame(pd.read_sql_query(query5, con))
power_list = df[['TOT_POW', 'Time', 'ST']].to_numpy()

start_limit = 80  # Given in [%]
start_time = 10  # Given in [s]
stop_limit = 70  # Given in [%]
stop_time = 200  # Given in [s]

#  LINE PLOTS: MEASURED TOTAL POWER, ESTIMATED STOP TIME, ESTIMATED CHANGE IN
NUMBER OF ENGINES
plt.subplot(3, 1, 1)
plt.plot(df['TOT_POW'])
plt.title('TOTAL POWER IN kW')
plt.subplot(3, 1, 2)
plt.plot([item[-1] for item in simulate_running])
plt.title('ESTIMATED: STOP TIME')
plt.subplot(3, 1, 3)
plt.plot([item[2] for item in simulate_running])
plt.title('ESTIMATED: CHANGE IN NUMBER OF ENGINES')
plt.show()
```

Recorded total power, estimated countdown times and estimated change in the number of running engines over 28000 seconds are plotted.

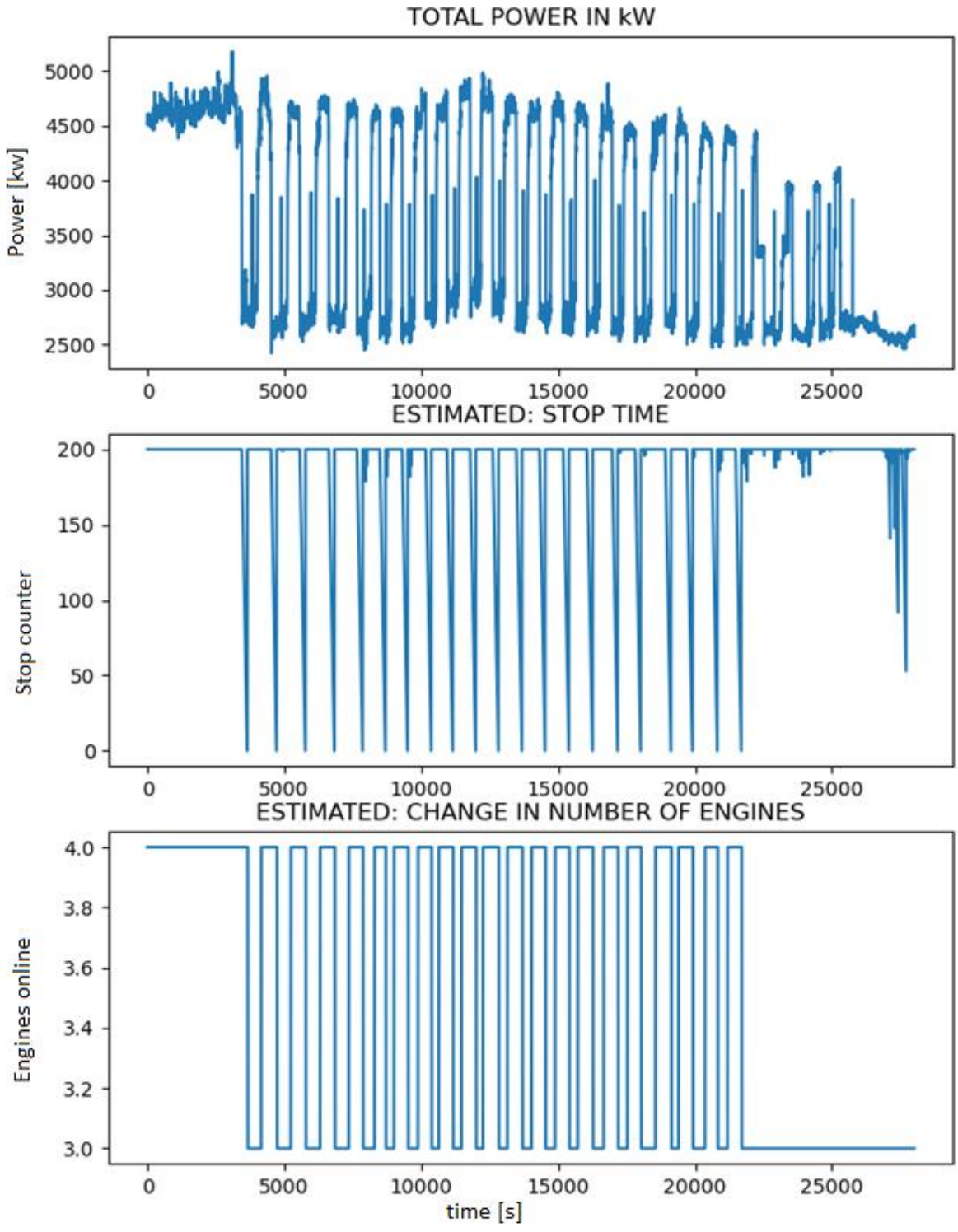

Figure 28. The effect of POOH on DG behaviour

Source: Created by the author using the matplotlib library

Again, a strong correlation between spikes in power and the change of number of running engines is observed.

In this simulation, the model estimated a total number of changes in running engines to 41, which averages down to one change every 10 minutes. This time the number of high loads remained at zero.

The above examples demonstrate that the commissioning parameters are not optimised for tripping and POOH operations due to intermittent starting and stopping of the engines, as well as the excess time spent at loads higher than 100%.

6.1.1.1. Mitigating the issues manually

As mentioned in chapter 2, during the data logging and exporting of the recorded values, the operational parameters of the PMS were altered from their commissioning values. The crew was aware of the tripping and POOH issues and had already tried to resolve the issues by adjusting the parameters to the following values:

```
start_limit = 85  # Given in [%]
start_time = 15  # Given in [s]
stop_limit = 71  # Given in [%]
stop_time = 1599  # Given in [s]
```

In this part of the study, the author explains the methodology and the results of manually mitigating the issues by two different methods:

- Changing the values of the operational parameters.
- Forcing the minimum required number of running engines to a fixed value.

6.1.1.2. Changing the value of the operational parameters

By changing the value of the STOP TIME parameter from the commissioning value of 200 seconds to a value of 1599 seconds (which is the largest possible value that the system would accept for this parameter) the system rarely gets enough time spent in the load zone required for load-dependent stop to successfully reach the countdown to zero. This resolves the problem of intermittent starting and stopping of the engines. Also, since the system effectively runs more engines than required by the calculation, a sudden surge in power does not push the engines to a load greater than 100%.

However, there is a considerable drawback in the form of reduced mean load per engine during the time when the equipment is at standstill. Furthermore, since a single second spent at a load higher than a load which satisfies the load-dependent stop countdown is sufficient to reset the counter back to the default value of 1599 seconds, a successful countdown to

zero is less likely to occur. As a result, the generator sets are underloaded, which impacts product health, operation and uptime while increasing the opportunity for unplanned events and shutdowns [18].

6.1.1.3. Forcing the minimum required number of engines

Another method of mitigation was setting the minimum required number of engines to a fixed value, effectively forcing the system to run a higher number of engines than required by the calculation. Most of the time this was set to a minimum of three engines, making the system perform load-dependent starting and stopping strictly in the range from three to five engines, never dropping below three engines.

Like the method above, this effectively lowered the mean load per engine during the times when three engines were running but two engines were required by the calculation. Furthermore, this method was subject to human error, as the crew could forget to remove the forced value once tripping and POOH operations were completed.

The author provides proof that the third engine was started by a human action rather than an automatic start by extracting the measured data from the database.

Table 3. Export from the SQLite database

DG5 POWER [%]	REACTIVE POWER [kW]	REACTIVE POWER [%]	cos ϕ	FREQUENCY [Hz]	VOLTAGE [V]	ENGINES RUNNING
75.9375	1032.0	45.125	0.802734375	59.96875	673.0	2.0
75.0	1039.0	45.40625	0.79736328125	60.0	673.5	2.0
75.4375	1041.0	45.5	0.798339843375	60.0	673.5	2.0
74.8125	1045.0	45.6875	0.794921875	60.0	673.5	2.0
76.0625	1041.0	45.5	0.80078125	59.9375	673.0	2.0
75.875	1036.0	45.28125	0.8017578125	59.9375	673.0	2.0
76.1875	1052.0	45.96875	0.798339843375	59.96875	673.0	2.0
75.0	1054.0	46.0625	0.79345703125	60.0	673.0	2.0
75.8125	1045.0	45.6875	0.798828125	60.0	673.5	2.0
75.75	1048.0	45.8125	0.7978515625	59.96875	673.0	2.0

75.9375	1043.0	45.59375	0.7919921875	59.96875	673.5	2.0
75.4375	1050.0	45.90625	0.79541015625	60.0	673.5	2.0
73.9375	1046.0	45.71875	0.791015625	60.0	673.5	2.0
74.5625	1046.0	45.71875	0.796875	60.03125	673.5	2.0
74.0	1041.0	45.5	0.7939453125	60.0	673.5	2.0
74.75	1046.0	45.71875	0.79443359375	59.96875	673.5	2.0
74.6875	1041.0	45.5	0.79541015625	59.96875	673.5	2.0
75.1875	1033.0	45.15625	0.7998046875	59.96875	673.5	2.0
74.5	1034.0	45.21875	0.796875	59.96875	673.0	2.0
75.25	1041.0	45.5	0.7978515625	60.0	673.5	2.0
74.0	1033.0	45.15625	0.794921875	60.03125	673.5	2.0
75.1875	1040.0	45.46875	0.7978515625	60.0	673.5	2.0
73.8125	1036.0	45.28125	0.79443359375	59.96875	673.0	2.0
74.9375	1038.0	45.375	0.79638671875	60.0	673.5	2.0
74.0	1040.0	45.46875	0.79296875	60.0	673.0	2.0
74.375	1037.0	45.34375	0.79541015625	59.96875	673.5	2.0
73.4375	1038.0	45.375	0.79150390625	60.0	673.5	2.0
74.3125	1029.0	45.0	0.79736328125	60.0	673.5	2.0
74.1875	1047.0	45.78125	0.7919921875	59.90625	672.5	2.0
75.125	1037.0	45.34375	0.79833984375	59.96875	673.5	2.0
73.6875	1042.0	45.5625	0.791015625	59.96875	673.5	2.0
75.375	1039.0	45.40625	0.798828125	59.875	673.0	2.0
76.8125	1051.0	45.9375	0.80078125	59.8125	672.0	2.0
74.3125	1046.0	45.71875	0.79248046875	60.09375	674.0	2.0
74.0	1049.0	45.84375	0.790527343750	60.0625	673.5	2.0
74.25	1041.0	45.5	0.787109375	60.0	673.5	2.0
73.1875	1044.0	45.625	0.788574218750	60.0	673.5	2.0
73.1875	1041.0	45.5	0.789550781250	59.96875	673.5	2.0

74.5	1042.0	45.5625	0.79443359375	59.96875	673.5	2.0
74.3125	1043.0	45.59375	0.793457031250	60.0	673.0	2.0
74.4375	1038.0	45.375	0.795410156250	60.0	673.5	2.0
72.9375	1046.0	45.71875	0.787109375	60.03125	673.5	2.0
72.8125	1031.0	45.0625	0.791015625	60.03125	673.5	2.0
71.875	1034.0	45.21875	0.7861328125	60.03125	674.0	2.0
72.5	1034.0	45.21875	0.788574218750	60.0	673.5	2.0
71.0625	1032.0	45.125	0.783203125	60.0	673.0	2.0
73.125	1036.0	45.28125	0.790527343750	60.0	674.0	2.0
72.625	900.5	39.375	0.792480468750	60.0	679.5	2.0
70.375	875.0	38.25	0.8271484375	60.0625	679.5	2.0
70.375	851.0	37.21875	0.834472656250	60.03125	679.5	2.0
69.75	835.5	36.53125	0.836425781250	60.03125	679.5	2.0
68.4375	807.5	35.3125	0.840332031250	60.03125	679.5	2.0
67.9375	796.0	34.78125	0.842285156250	60.0625	680.0	2.0
66.1875	793.0	34.65625	0.8369140625	60.09375	680.5	2.0
66.25	788.0	34.4375	0.838378906250	60.09375	680.0	2.0
63.875	770.0	33.65625	0.8349609375	60.125	679.5	2.0
64.4375	761.5	33.28125	0.83984375	60.125	680.0	2.0
62.25	746.5	32.625	0.836425781250	60.09375	679.5	2.0
62.09375	740.5	32.375	0.837890625	60.125	679.5	2.0
59.625	738.5	32.28125	0.834472656250	60.125	679.5	2.0
59.84375	735.5	32.15625	0.8310546875	60.125	678.5	2.0
58.875	725.5	31.71875	0.829589843750	60.125	678.5	2.0
57.03125	735.5	32.15625	0.812011718750	60.125	678.5	2.0

54.5625	744.5	32.53125	0.802734375	60.1875	678.5	3.0

Source: Created by the author using the DB Browser for SQLite software

Recorded data shows no indication of increased load in the moments before the change in the number of running engines. DG5 works in balanced load, never ramping higher than the value of the START LIMIT parameter.

The change in the number of running engines is a result of human action, rather than an automatic start commanded by the PMS.

6.1.1.4. Proposal of the solution

The author proposes a solution to the problem of unstable work during tripping and POOH operations in the form of a STOP TIME parameter adjustment.

Three subplots are presented, covering the period of the POOH operations:

- The top graph presents the recorded total power, in [kW].
- The middle graph presents the countdown to a load-dependent stop, in [s]
- The bottom graph displays the change in the number of running engines.

Two simulations are run. The first simulation runs under commissioning parameters, while the second simulation runs with the STOP TIME equal to 650, as approximated in the previous chapter (ΔT=650).

6.1.1.5. POOH Simulation

The simulation is run during POOH operations over a period of 28000 seconds.

```
query5 = 'SELECT "TOTAL POWER" AS TOT_POW, "Time", "START TIMER" AS ST,
"REQUIRED ENGINES" AS ENGINES from Engine_Room_Database LIMIT 28000 OFFSET
3122500'
df = pd.DataFrame(pd.read_sql_query(query5, con))
power_list = df[['TOT_POW', 'Time', 'ST']].to_numpy()

start_limit = 80  # Given in [%]
start_time = 10  # Given in [s]
stop_limit = 70  # Given in [%]
stop_time = 200  # Given in [s]

#  LINE PLOTS: MEASURED TOTAL POWER, ESTIMATED STOP TIME, ESTIMATED CHANGE IN
NUMBER OF ENGINES
plt.subplot(3, 1, 1)
plt.plot(df['TOT_POW'])
plt.title('TOTAL POWER IN kW')
plt.subplot(3, 1, 2)
plt.plot([item[-1] for item in simulate_running])
```

```
plt.title('ESTIMATED: STOP TIME')
plt.subplot(3, 1, 3)
plt.plot([item[2] for item in simulate_running])
plt.title('ESTIMATED: CHANGE IN NUMBER OF ENGINES')
plt.show()
```

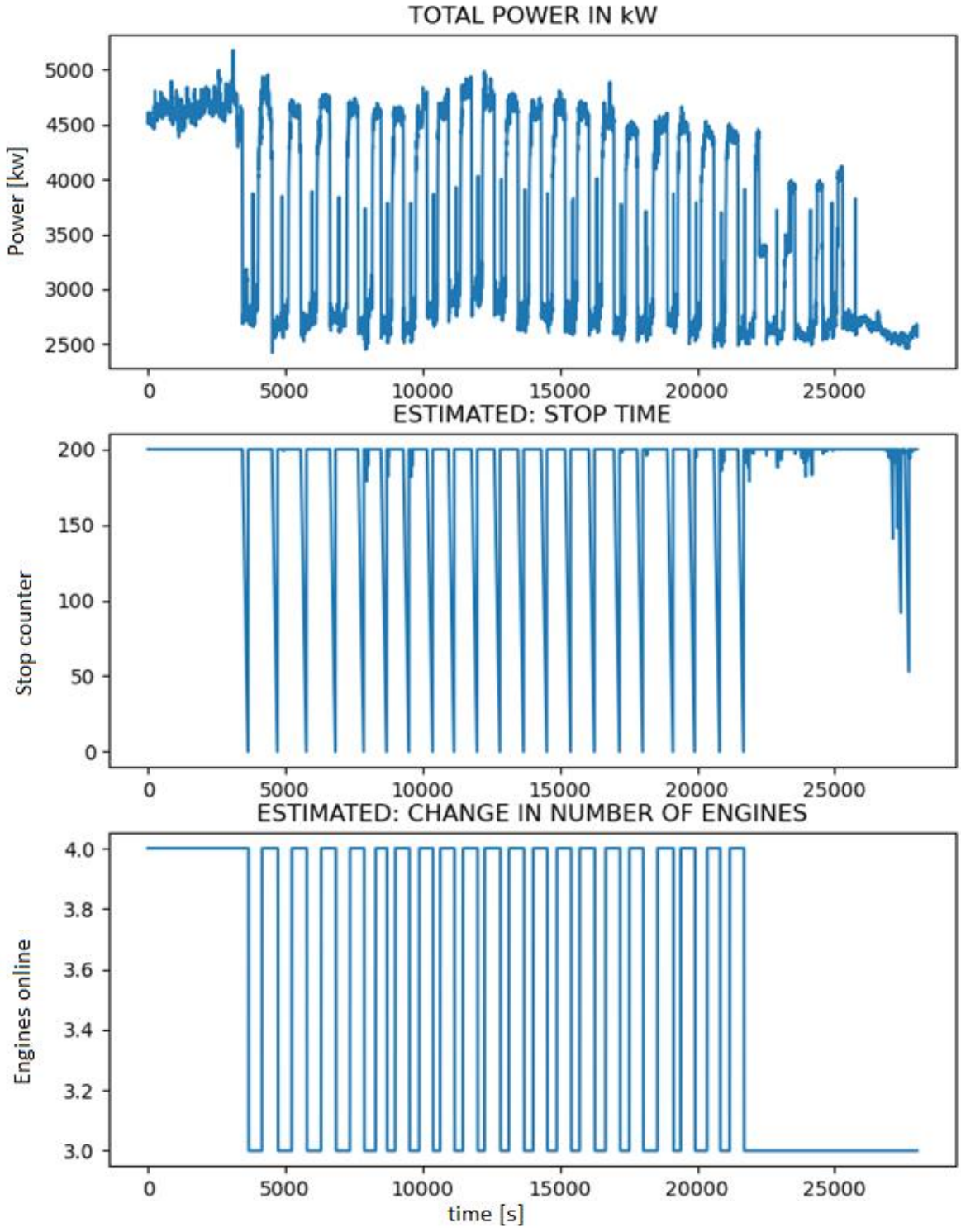

Figure 29. Simulation of POOH, period = 25000 seconds, STOP TIME = 200

Source: Created by the author using the matplotlib library

Figure 29 presents the result of the first simulation.

It is observed that the commissioning value of the STOP TIME parameter is not large enough to provide reliable performance during the POOH operations. Each surge in total power results in an engine start, followed by an immediate countdown to stop, unloading, and stopping of the engine. To eliminate a successful countdown to zero, the value of the STOP TIME parameter needs to be increased. As a result, load-dependent stop would not occur as often, and the overall stability and performance would be increased.

The model estimates the following key values over a period of 28000 seconds:

Table 3. Estimates of the first simulation under the commissioned parameters

Changes in number of running engines:	41
Total running hours [h]:	27.29
Total fuel consumption [L]:	7595.51
Total time spent at load higher than 100% [s]:	0

Source: Created by the author using the MS Word software

The same simulation is run again, but this time with the modified STOP TIME parameter. The rest of the parameters remain at their commissioned values:

```
start_limit = 80  # Given in [%]
start_time = 10  # Given in [s]
stop_limit = 70  # Given in [%]
stop_time = 650  # Given in [s]
```

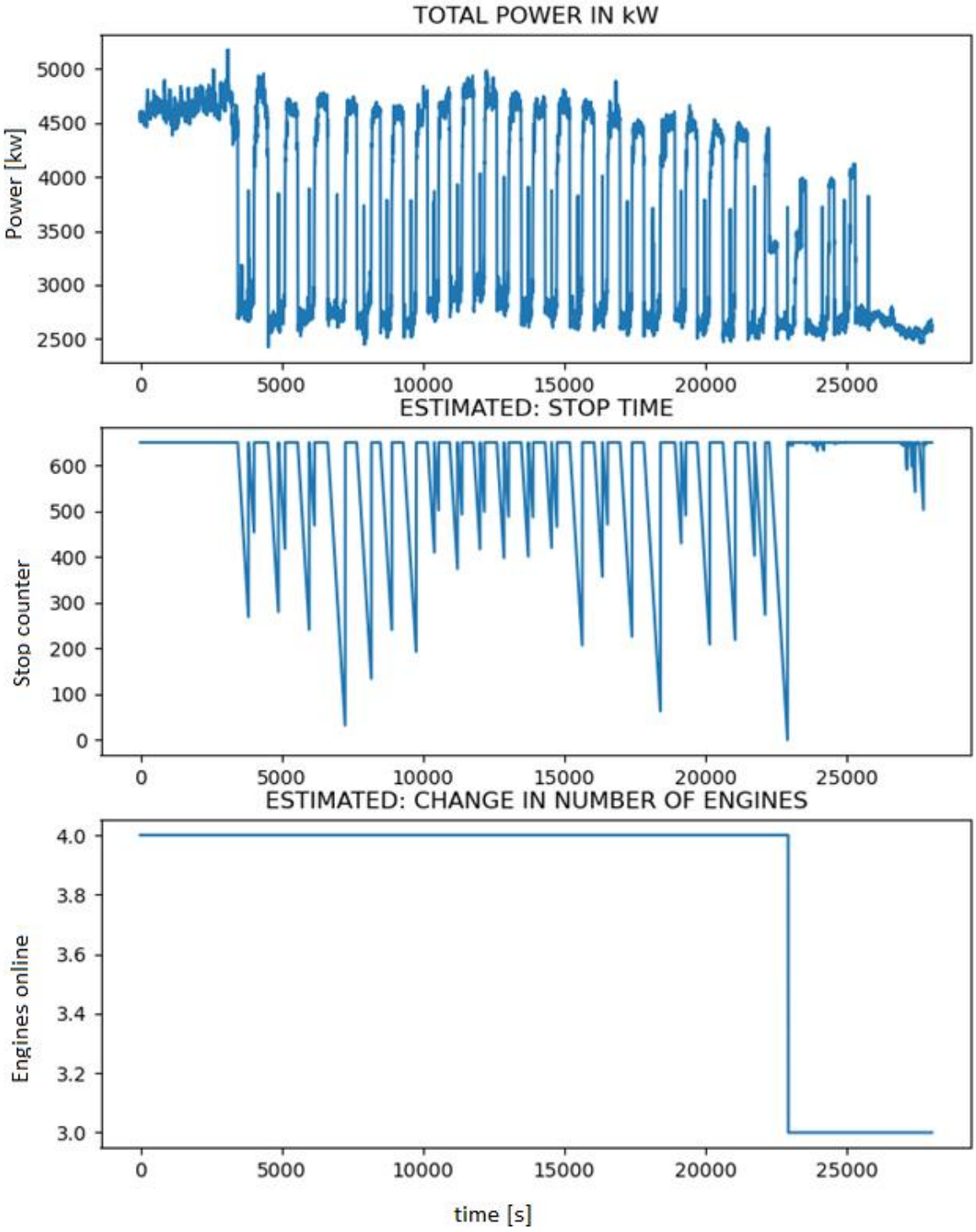

Figure 30. Simulation of POOH, period = 25000 seconds, STOP TIME = 650

Source: Created by the author using the matplotlib library

Figure 30 presents the result of the second simulation.

It is observed that the modified STOP TIME value of 650 seconds was chosen correctly and that the intermittent starting and stopping of the engines is fully eliminated. Surges in total power are frequent enough to reset the countdown to zero before it is successfully reached. As a result, no engines are started or stopped for the entire duration of the POOH operations. POOH operations are completed around 23000 seconds, resulting in the first successful countdown to zero, followed by a load-dependent stop. The system is balanced during the entire time of the simulation.

The model estimates the following key values over a period of 28000 seconds:

Table 4. Estimates of the first simulation under the modified parameters

Changes in number of running engines:	1
Total running hours [h]:	29.7
Total fuel consumption [L]:	7657.32
Total time spent at load higher than 100% [s]:	0

Source: Created by the author using the MS Word software

There is a drawback in the form of increased running hours and increased fuel consumption. Nonetheless, the drawback is acceptable since the priority on the drilling rig is stability and safety of operations, especially those operations that heavily involve well control [19].

6.1.1.6. Tripping simulation

The simulation is run for the tripping operations over a period of 25000 seconds.

```
query5 = 'SELECT "TOTAL POWER" AS TOT_POW, "Time", "START TIMER" AS ST,
"REQUIRED ENGINES" AS ENGINES from Engine_Room_Database LIMIT 25000 OFFSET
1720000'
df = pd.DataFrame(pd.read_sql_query(query5, con))
power_list = df[['TOT_POW', 'Time', 'ST']].to_numpy()

start_limit = 80  # Given in [%]
start_time = 10  # Given in [s]
stop_limit = 70  # Given in [%]
stop_time = 200  # Given in [s]

#  LINE PLOTS: MEASURED TOTAL POWER, ESTIMATED STOP TIME, ESTIMATED CHANGE IN
NUMBER OF ENGINES
plt.subplot(3, 1, 1)
plt.plot(df['TOT_POW'])
plt.title('TOTAL POWER IN kW')
plt.subplot(3, 1, 2)
plt.plot([item[-1] for item in simulate_running])
plt.title('ESTIMATED: STOP TIME')
plt.subplot(3, 1, 3)
plt.plot([item[2] for item in simulate_running])
plt.title('ESTIMATED: CHANGE IN NUMBER OF ENGINES')
plt.show()
```

Returns:

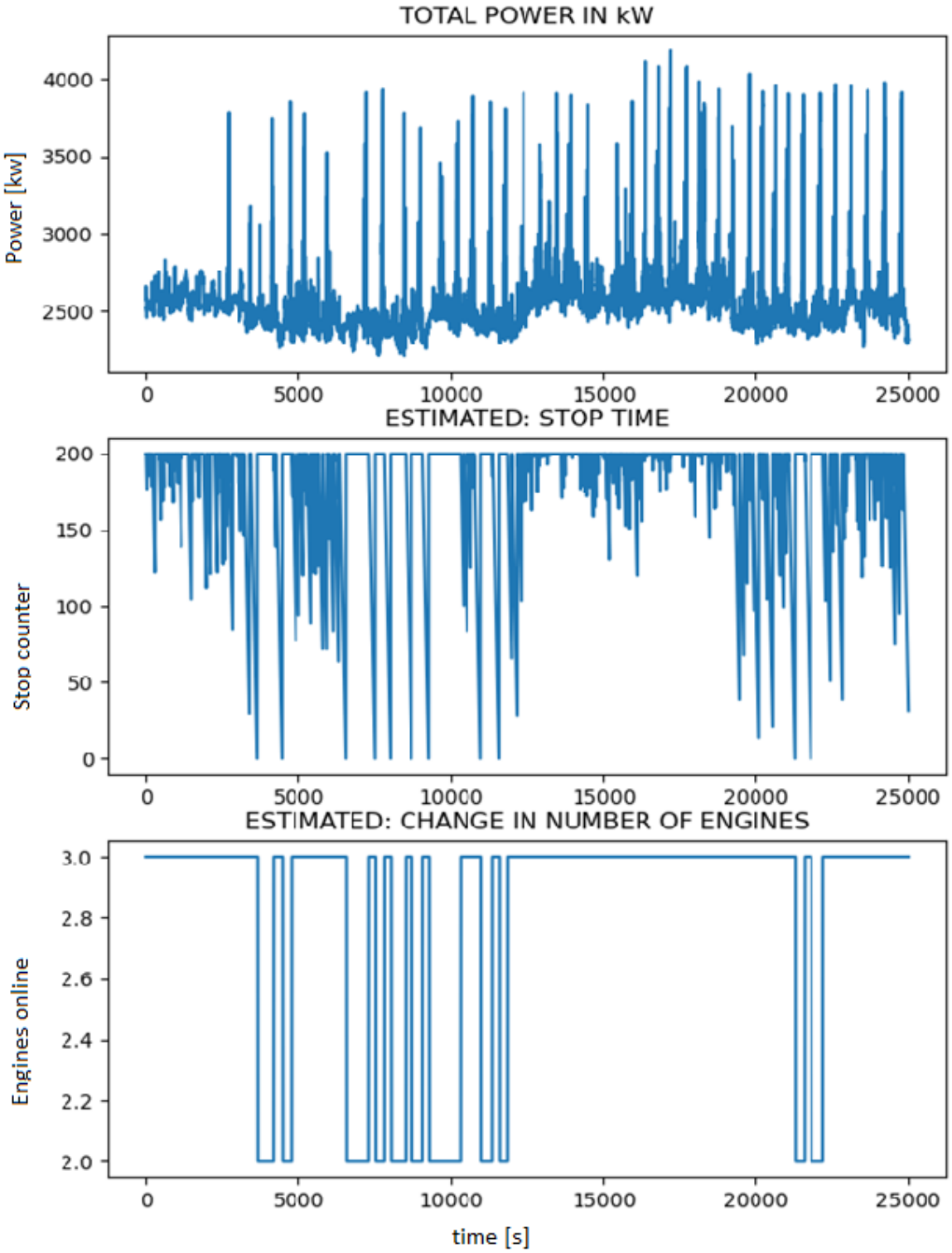

Figure 31. Simulation of tripping, period = 25000 seconds, STOP TIME = 200

Source: Created by the author using the matplotlib library

Figure 31 presents the result of the first simulation.

It is observed that the commissioning value of the STOP TIME parameter is not large enough to provide reliable performance during the tripping operations. As soon as the peak demand is over, the countdown to a load-dependent stop begins. Since the STOP TIME duration is shorter than the intervals between peak demands, a successful countdown to zero is reached approximately 50% of the time. As a result, a load-dependent stop is executed. Once the engine is stopped, the remaining engines continue load-sharing. The engines are then heavily exposed to the next sudden surge in power. As a result, engines are loaded to more than 100% of the nominal load, which induces a high-load start. The cycle then repeats.

The model estimates the following key values over a period of 25000 seconds:

Table 5. Estimates of the second simulation under the commissioned parameters

Changes in number of running engines:	22
Total running hours [h]:	19.46
Total fuel consumption [L]:	4871.62
Total time spent at load higher than 100% [s]:	86

Source: Created by the author using the MS Word software

The same simulation is run again, but this time with the modified STOP TIME parameter. The rest of the parameters remain at their commissioned values:

```
start_limit = 80  # Given in [%]
start_time = 10  # Given in [s]
stop_limit = 70  # Given in [%]
stop_time = 650  # Given in [s]
```

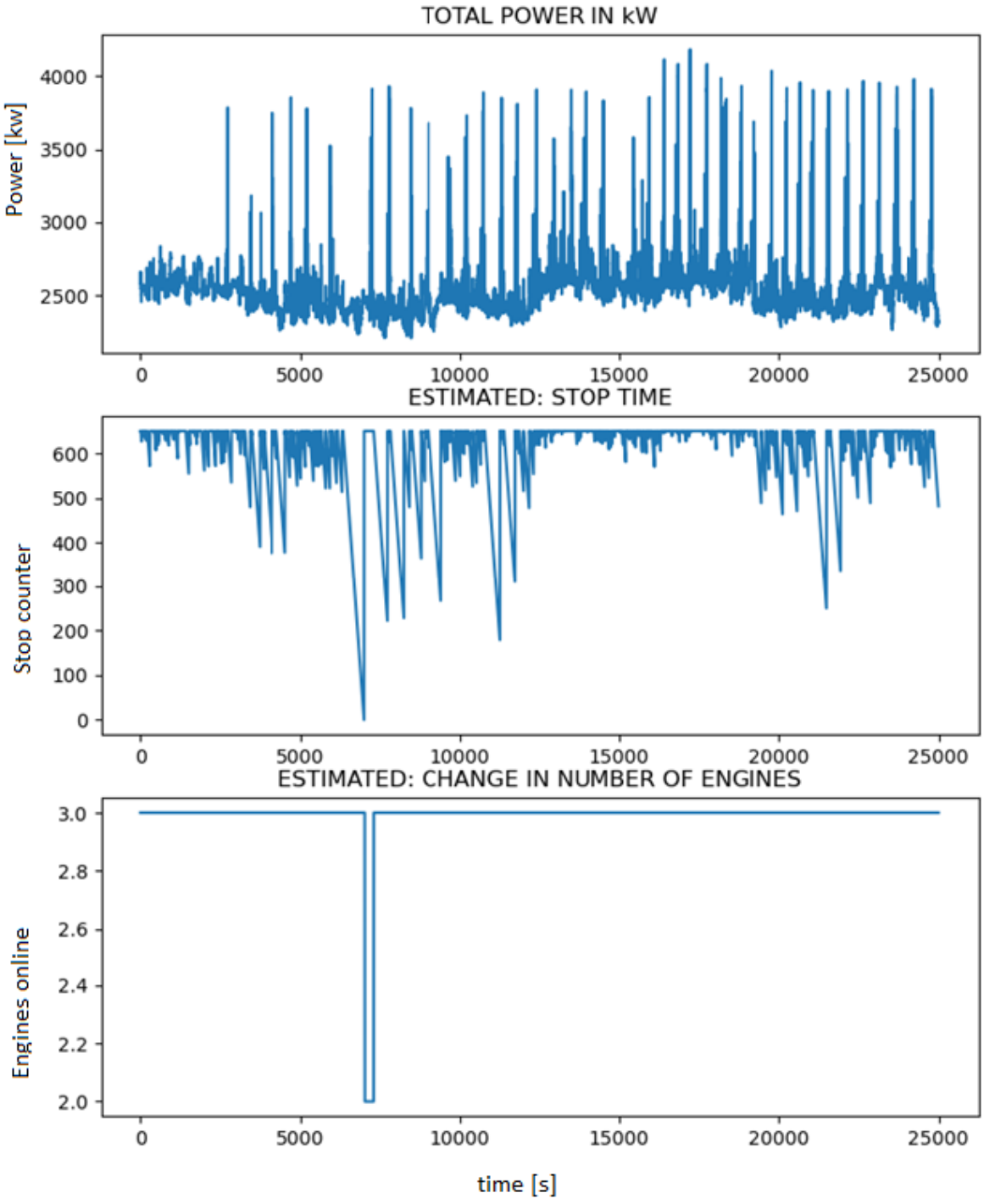

Figure 32. Simulation of tripping, period = 25000 seconds, STOP TIME = 650

Source: Created by the author using the matplotlib library

It is observed that the modified STOP TIME value of 650 seconds was chosen correctly and that the intermittent starting and stopping of the engines is greatly reduced. Surges in total power are frequent enough to reset the countdown to zero before it is successfully reached.

As a result, a single load-dependent start and stop is executed. The total time spent at load greater than 100% is reduced from 86 seconds to 14 seconds.

The model estimates the following key values over a period of 25000 seconds:

Table 6. Estimates of the second simulation under the modified parameters

Changes in number of running engines:	2
Total running hours [h]:	20.76
Total fuel consumption [L]:	4912.75
Total time spent at load higher than 100% [s]:	14

Source: Created by the author using the MS Word software

There is a drawback in the form of increased running hours and increased fuel consumption, but with the benefit of significant reduction in the number of changes in running engines and the amount of time spent under high load.

7. COMPARISON OF VALUES AND PARAMETERS

In this chapter, the simulations are run on the entire dataset. The outcomes are compared in order to find the optimal parameters for operating the automation system.

First, the recorded values are compared against the values given by the developed model while set to run in a fully automatic mode under the commissioned parameters. Then, the simulation is repeated by running the model in a fully automatic mode but under the modified parameters.

The parameters are presented in the form of comma separated values. For example, if the parameters are:

- START LIMIT = 85 [%]
- START TIME = 15 [s]
- STOP LIMIT = 71 [%]
- STOP TIME = 1599 [s]

the table shows:

- [85, 15, 71, 1599]

Table 7. Results of simulations on the entire dataset

	Measurements and Simulations	Change in number of engines	Total running hours [h]	Total fuel consumption [L]	Total time spent at load > 100% [s]
1.	Measured [85, 15, 71, 1599]	Unknown	4092	1175108	80
2.	Model (as measured) [85, 15, 71, 1599]	108	4027	1173523	194
3.	Model (as commissioned) [80, 10, 70, 200]	1084	3935	1170127	39909
4.	Model (adjusted) [85, 15, 71, 650]	278	3937	1170875	1472

Source: Created by the author using the MS Word software

The outcomes are explained:

The measured values (row 1) provided least time spent at load greater than 100%. This is expected, as the number of minimum required running engines was sometimes forced. Due to forcing of the number of running engines and unavailable data for forced inputs, it is not possible to accurately determine the actual changes in the number of running engines for the recorded data. However, the model estimates the number relatively close to 108 due to consistency of other values between row 1 and row 2.

The model as measured (row 2) is consistent with the recorded data from row 1. The model also successfully passes all benchmark test from chapter 5. It estimates slightly fewer working hours and less fuel consumption due to never being forced to run more engines than required by the calculation. Because the model never runs more engines than required by the calculation, the engines were more exposed to sudden surges in power, which results in an increase of the total time spent at loads greater than 100%.

As commissioned, the model (row 3) shows unbalanced operations and erratic behaviour in terms of starting and stopping the engines. Since engines are started and stopped more often, the change in the number of running engines is the highest, followed by low running hours and low fuel consumption. Since the engines are often stopped, they are extremely exposed to sudden surges in power, accounting to a total of 39909 seconds spent at loads greater than 100%. A total change in the number of running engines is estimated at 1084, which further stresses the switchgear [20] rated at 25000 lifetime cycles per circuit breaker.

An adjusted model (row 4) yields the best overall results across the table. The change in the number of engines is steady at 278, averaged down to five changes per day. Total running hours and the fuel consumption are lower than the values in rows 1 and 2. Total time spent at loads greater than 100% is 1472 seconds, averaged down to 26 seconds per day, or 10.4 seconds per day per engine.

Values from row 4 serve as basis for further optimisation.

Simulations are run repeatedly with different parameter values to fine-tune the overall system.

The summary of simulations that yielded viable results is presented, together with the recorded values and the values of the default simulation (last two rows).

Table 8. Summary of simulations

	Parameters	Change in number of running engines	Total running hours [h]	Total fuel consumption [L]	Total time spent at load > 100% [s]
1.	[90, 15, 70, 650]	216	3919	1170066	1878
2.	[90, 30, 70, 650]	216	3909	1169744	1945
3.	[90, 20, 70, 650]	214	3917	1170013	1864
4.	[90, 20, 65, 650]	70	4160	1176987	115
5.	[90, 20, 69, 650]	180	3950	1170989	503
6.	[90, 20, 71, 650]	254	3869	1168615	2913
7.	[88, 15, 68, 650]	136	4024	1173178	338
8.	[88, 15, 69, 650]	188	3981	1171969	486
9.	[88, 15, 70, 650]	230	3948	1171015	1847

10.	[88, 15, 71, 650]	272	3900	1169604	2925
11.	[89, 15, 68, 650]	134	4021	1173083	342
12.	[89, 15, 69, 650]	188	3976	1171826	500
13.	[89, 15, 70, 650]	224	3944	1170923	691
14.	[89, 15, 71, 650]	262	3895	1169455	2708
15.	[91, 15, 68, 650]	126	3986	1171945	345
16.	[91, 15, 69, 650]	176	3940	1170671	500
17.	[91, 15, 70, 650]	214	3908	1169729	1880
18.	[91, 15, 71, 650]	254	3860	1168335	2716
19.	[92, 15, 68, 650]	124	3977	1171601	2231
20.	[92, 15, 69, 659]	174	3931	1170335	2345
21.	[92, 15, 70, 650]	208	3901	1169466	3707
22.	[92, 15, 71, 650]	250	3854	1168070	4638
23.	Measured	Unknown	4092	1175108	80
24.	Model default	108	4027	1173523	194

Source: Created by the author using the MS Word software

A Pandas DataFrame containing the results of the simulations is created. All the simulation estimates are added, together with the measured data, where the number 90 is manually added in place of the unknown data. The outlier values from the commissioning simulation are not added, since the outlier would significantly affect the calculated mean.

```
df = pd.DataFrame([
    [1  ,   216,    3919,   1170066 ,   1878],
    [2  ,   216,    3909,   1169744 ,   1945],
    [3  ,   214,    3917,   1170013 ,   1864],
    [4  ,   70,     4160,   1176987 ,   115],
    [5  ,   180 ,   3950,   1170989 ,   503],
    [6  ,   254,    3869,   1168615,    2913],
    [7,     136,    4024,   1173178,    338],
    [8,     188,    3981,   1171969,    486],
    [9,     230,    3948,   1171015,    1847],
    [10,    272,    3900,   1169604,    2925],
    [11,    134,    4021,   1173083,    342],
    [12,    188,    3976,   1171826,    500],
    [13,    224,    3944,   1170923,    691],
    [14,    262,    3895,   1169455,    2708],
    [15,    126,    3986,   1171945,    345],
    [16,    176,    3940,   1170671,    500],
    [17,    214,    3908,   1169729,    1880],
    [18,    254,    3860,   1168335,    2716],
    [19,    124,    3977,   1171601,    2231],
    [20,    174,    3931,   1170335,    2345],
    [21,    208,    3901,   1169466,    3707],
    [22,    250,    3854,   1168070,    4638],
    [23,    90,     4092,   1175108,    80],
    [24,    108,    4027,   1173523,    194]])


print(df.describe())
```

Statistics on the DataFrame are described.

Returns:

	0	1	2	3	4
count	24.000000	24.000000	24.000000	2.400000e+01	24.000000
mean	12.500000	187.833333	3953.708333	1.171094e+06	1570.458333
std	7.071068	57.240656	72.783466	2.131473e+03	1283.552170
min	1.000000	70.000000	3854.000000	1.168070e+06	80.000000
25%	6.750000	135.500000	3906.250000	1.169698e+06	450.750000
50%	12.500000	198.000000	3942.000000	1.170797e+06	1855.500000
75%	18.250000	225.500000	3982.250000	1.171951e+06	2435.750000
max	24.000000	272.000000	4160.000000	1.176987e+06	4638.000000

A Python script that helps us choose the best solution based on the function minimum is written. Each solution has drawbacks either in the form of increased fuel consumption and running hours, or in the form of increased starting/stopping and high-load times.

Python function:

```
df['Score'] = 0

def rate_parameters(df):
    for I in range(1,5):
        for index, row in df.iterrows():
            if row[i] < df[i].mean():
                df['Score'][index] += 1
    return df
```

Function takes the DataFrame and iterates through every row. It compares every value of every row with the mean of the associated column. If the value is lower than the column mean, df['Score'] is incremented by 1. Rows with the highest Score are those containing the most values under the mean.

Returns:

	0	1	2	3	4	Score
0	1	216	3919	1170066	1878	2
1	2	216	3909	1169744	1945	2
2	3	214	3917	1170013	1864	2
3	4	70	4160	1176987	115	2
4	5	180	3950	1170989	503	4
5	6	254	3869	1168615	2913	2
6	7	136	4024	1173178	338	2
7	8	188	3981	1171969	486	1
8	9	230	3948	1171015	1847	2
9	10	272	3900	1169604	2925	2
10	11	134	4021	1173083	342	2
11	12	188	3976	1171826	500	1
12	13	224	3944	1170923	691	3
13	14	262	3895	1169455	2708	2
14	15	126	3986	1171945	345	2
15	16	176	3940	1170671	500	4
16	17	214	3908	1169729	1880	2
17	18	254	3860	1168335	2716	2
18	19	124	3977	1171601	2231	1
19	20	174	3931	1170335	2345	3
20	21	208	3901	1169466	3707	2
21	22	250	3854	1168070	4638	2
22	23	90	4092	1175108	80	2
23	24	108	4027	1173523	194	2

Results are observed. Two simulations yielded results where all the values are below the mean. The one with lower values is chosen as the best option.

[91, 15, 69, 650]	176	3940	1170671	500

7.1. SAVINGS AND DRAWBACKS

Presented are the estimated differences for a period of one year by extrapolating from the model estimates. Measured data used in this study was acquired over a period of 55 days. The yearly totals are estimated as follows:

Table 9. Annual projected savings and drawbacks

	Measured	Model	Difference
Change in number of running engines	Unknown	1168	Unknown
Total running hours [h]	27 156	26 147	-1009
Total fuel consumption [L]	7 798 444	7 768 998	-29 446
Total CO_2 emissions [T]	20 899	20 820	-79
Total time spent at load > 100% [s]	530	3318	+2788

Source: Created by the author using the MS Word software

The differences are briefly discussed.

The model estimates a total change in the number of running engines to 1168. The actual number of changes in the measured data is unknown due to forcing of the minimum required number of running engines, so it is not possible to present the difference. At an annual 1168 Main SWBD circuit breaker cycles, it would take approximately 21 years per circuit breaker to reach the lifetime mechanical durability of 25000 cycles [21].

By lowering the total running hours by 1009, one 1000-hour service is avoided every year, saving man-hours and consumable parts [22]. A typical 1000-h service consists of oil and filter changes.

Fuel consumption savings of 29446 litres accounts for a price of approximately 0.64 USD per litre (as of 01.06.2020, [23] meaning that it is possible to save approximately 18845 USD annually. Furthermore, a total reduction of CO_2 emissions by 78.6 metric tonnes annually is estimated [24].

There is a drawback in total time of running the engines at loads greater than 100%. At 3318 seconds annually, the model approximates less than 10 seconds of high-loads per day, or an average of 3.6 seconds per engine daily, meaning that the drawback does not place a hard constraint on the solution.

7.1.2. Suggestions for further improvements

Four suggestions for further reduction in running hours and fuel consumption are presented.

7.1.2.1. Development of communication channel between the drilling equipment and the PMS

Since the PMS and the drilling package are both interfaced to the VMS (IAS) [25], a communication channel could be programmed with minimal or no modification to the existing physical infrastructure.

Integrated in the drilling software, a choice between “Drilling mode” and “Tripping mode” already exists and is used by the operators on the drill floor. Drilling equipment functions differently when “Tripping mode” is selected, so the change in the selection could be broadcast to the IAS and the PMS, respectively. If the existing PLC installed on the PMS could detect the change in the selected mode, a new logic for running the engines could be activated, eliminating the need to manually force the minimum required number of engines, or the need to trade off performance for increased high-load times.

The downside of this approach would be the requirement by different companies to modify the current state of the system. In this example, it would require the manufacturers of the drilling equipment, PMS and the AIS to collaborate, which could prove time consuming, expensive and challenging to organise.

7.1.2.2. Upgrade of the DG cooling system

Physical upgrade to the cooling system would allow the engines to run at higher loads for an extended amount of time.

The main downside of this proposal is the requirement for a dry dock modification or a total shutdown of the drilling facility and a changeover to the emergency generator while the work is carried out.

7.1.2.3. Development of battery powered peak-shaving system

An installation and commissioning of battery powered peak-shaving equipment on the drilling rig would allow the PMS to utilise the DGs only for base load power requirements [26]. The sudden surges in power would be met by batteries installed on the drilling facility. This would eliminate the need to run an additional engine for peak demand and would in turn save significant running hours, fuel consumption and high-load times.

This solution would require a major overhaul on the existing system and would be the most expensive and complex, but it would yield best overall results.

7.1.2.4. Development of the machine learning algorithm

A development of a machine learning algorithm that would measure all the parameters and values continuously and then fine-tune the operational parameters would be a logical next step in the advanced development of the model presented in this study. Such an algorithm could then be implemented on-board the drilling rig in the existing IT infrastructure.

8. SUMMARY OF MATPLOTLIB GRAPHS

All the matplotlib graphs written and used in this study are presented.

```
# MEASURED: LINE PLOT DG1, DG2, STOP TIMER, DG3
plt.subplot(4, 1, 1)
df['DG1'].plot()
plt.ylabel('Load [%]')
plt.title('DG1 LOAD, DG2 LOAD, STOP TIME COUNTER AND DG3 LOAD')
plt.subplot(4, 1, 2)
df['DG5'].plot()
plt.ylabel('Load [%]')
plt.subplot(4, 1, 3)
plt.plot([item[-1] for item in simulate_running])
plt.ylabel('Counter')
plt.subplot(4, 1, 4)
df['DG4'].plot()
plt.ylabel('Load [%]')
plt.show()
```

```
#  LINE PLOTS: MEASURED TOTAL POWER, ESTIMATED STOP TIME, ESTIMATED CHANGE IN
NUMBER OF ENGINES
plt.subplot(3, 1, 1)
plt.plot(df['TOT_POW'])
plt.title('TOTAL POWER IN kW')
plt.subplot(3, 1, 2)
plt.plot([item[-1] for item in simulate_running])
plt.title('MODEL: STOP TIME')
plt.subplot(3, 1, 3)
plt.plot([item[2] for item in simulate_running])
plt.title('MODEL: CHANGE IN NUMBER OF ENGINES')
plt.show()
```

```
#  START TIME LINE PLOTS
plt.subplot(2, 1, 1)
plt.plot(df['ST'])
plt.title('MEASURED: START TIME')
plt.subplot(2, 1, 2)
plt.plot([item[-1] for item in simulate_running])
plt.title('MODEL: START TIME')
plt.show()
```

```
#  DG1 HISTOGRAM
data=[[item[3][0,0] for item in simulate_running]]
arr=plt.hist(data, bins=bins, log=True, range=[-1,150])
for i in range(bins):
    plt.text(arr[1][i],arr[0][i],str(arr[0][i]))
```

```
plt.title('MODEL: DG1 LOAD')
plt.show()



# LINE SUBPLOTS OF DG1 MEASURED AND DG1 ESTIMATED LOAD
plt.subplot(2, 1, 1)
plt.plot(df['DG1'])
plt.title('MEASURED: DG1 LOAD')
plt.subplot(2, 1, 2)
plt.plot([item[3][0,0] for item in simulate_running])
plt.title('MODEL: DG1 LOAD')
plt.show()



#  LINE SUBPLOTS OF MEASURED AND ESTIMATED CHANGE IN NUMBER OF ENGINES
plt.subplot(2, 1, 1)
plt.plot(df['ENGINES'])
plt.title('MEASURED: CALCULATION OF REQUIRED ENGINES')
plt.subplot(2, 1, 2)
plt.plot([item[2] for item in simulate_running])
plt.title('MODEL: CHANGE IN NUMBER OF ENGINES')
plt.show()



# HISTOGRAM DG1 MEASURED AND ESTIMATED
plt.subplot(2, 1, 1)
arr=plt.hist(df['DG1'], bins=bins, log=True, range=[-1,160])
for i in range(bins):
    plt.text(arr[1][i],arr[0][i],str(arr[0][i]))
plt.title('MEASURED: DG1 LOAD')
plt.subplot(2, 1, 2)
data=[[item[3][0,0] for item in simulate_running]]
arr=plt.hist(data, bins=bins, log=True, range=[-1,160])
for i in range(bins):
    plt.text(arr[1][i],arr[0][i],str(arr[0][i]))
plt.title('ESTIMATED: DG1 LOAD')
plt.show()



# MEAN LOAD PER ENGINE MEASURED AND ESTIMATED
plt.subplot(2, 1, 1)
df['MEAN POWER'] = df['TOT_POW'] / df['ENGINES'] / 1830 * 100
arr = plt.hist(df['MEAN POWER'], bins=bins, log=True, range=[-1, 160])
for i in range(bins):
    plt.text(arr[1][i], arr[0][i], str(arr[0][i]))
plt.title('MEASURED: MEAN LOAD PER ENGINE')
plt.subplot(2, 1, 2)
data = [[item[4] for item in simulate_running]]
arr = plt.hist(data, bins=bins, log=True, range=[-1, 160])
for i in range(bins):
    plt.text(arr[1][i], arr[0][i], str(arr[0][i]))
```

```
plt.title('ESTIMATED: MEAN LOAD PER ENGINE')
plt.show()


# HISTOGRAM 5X ENGINES ESTIMATED
for i in range(1, 6):
    plt.subplot(5, 1, i)
    data = [[item[3][0, i-1] for item in simulate_running]]
    arr=plt.hist(data, bins=bins, log=True, range=[-1,130])
    for i in range(bins):
        plt.text(arr[1][i], arr[0][i], str(arr[0][i]))
    plt.xlabel('Load [%]')
plt.title('MODEL: ALL DG HISTOGRAMS')
plt.show()
```

9. CONCLUSION

During this study, the author has demonstrated the operation of PMS on an offshore drilling rig.

The PMS simulator in the form of Python program was developed. Additional libraries to handle large sets of data, draw graphs and describe statistics on the data have been used.

Benchmark tests were run to prove that the model is viable and that its estimations are accurate.

The common issues of unreliable performance during tripping and POOH operations was explained and a proposed solution to these issues in the form of an adjustment of the operational parameters was given. Simulations under default and modified parameters were run. Different estimates were compared. Potential annual savings were presented.

Notable observations

- The commissioning parameters, particularly the STOP TIME parameter set at 200 seconds yield poor performance and lead to issues during tripping and POOH operations.
- The methods taken by the crew to mitigate the issues were focused on the change in operational parameters, but without the developed simulation model it was not possible to fine-tune and test the parameters.
- Adjusting the operational parameters to higher highs and lower lows always yields increased savings in terms of running hours and fuel consumption, but at an expense of change in the number of engines and time spent at loads greater than 100%.

REFERENCES

[1] Parallel Operation Procedure for Marine Diesel Generators in Ships (2020, June). *Marine Engineering Study Materials: Information for Marine Engineers.* Available at:

https://marineengineeringonline.com/parallel-operation-procedure-marine-diesel-generators-ships/#:~:text=Parallel%20Operation%20Procedure%20for%20Marine%20Diesel%20Generators%20in%20Ships,-May%2027%2C%202014&text=In%20order%20to%20put%20a,following%20conditions%20must%20be%20fulfilled.&text=The%20incoming%20generator%20frequency%20is,than%20the%20bus%20bar%20frequency. (15. 03. 2020.)

[2] Technical information portal (2020, June.) *CAT.* Available at https://www.cat.com/en_ZA/additional-product-information/product-families/technical-informationportal.html (26. 03. 2020.)

[3] Radan, D., Johansen, Tor A., Sørensen, A. J., Ådnanes, A. K. (2020, June) *Optimization of Load Dependent Start Tables in Marine Power Management Systems with Blackout Prevention.* [Online] Trondheim: Norwegian University of Science and Technology. Available at: http://folk.ntnu.no/torarnj/501-214.pdf?id=ansatte/Johansen_Tor.Arne/501-214.pdf (02. 04. 2020.)

[4] Delomatic 4 DM-4 Land/DM-4 Marine, Power Management Unit, Part 2, chapter 16. User manual. (2020, June.) *DEIF.* Available at: https://deif-cdn.azureedge.net/v-dj18w4yjn2du/documentation/download/%7BCBC3DB32-DC4D-49B4-AFB3-EBDB49B48663%7D (18. 03. 2020.)

[5] Diesel Generator Technical Specifications (2020, June) *United Nations global marketplace.* Available at: https://www.ungm.org/UNUser/Documents/DownloadPublicDocument?docId=539581 (29. 03. 2020.)

[6] Matplotlib: Visualization with Python. (2020, June.) *Matplotlib.* Available at: https://matplotlib.org/ (01. 05. 2020.)

[7] Pandas Dana Frame (2020, June.) *Pandas.* Available at: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html. (01. 05. 2020.)

[8] Deep Learning Performance Documentation (2020, June.) *NVIDIA.* Available at: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html. (02. 05. 2020.)

[9] DB Browser for SQLite (2020, June.) *SQLITEBROWSER.* Available at: https://sqlitebrowser.org/. (02. 05. 2020.)

[10] Functional programming. (2020, June.) *Wikipedia.* Available at: https://en.wikipedia.org/wiki/Functional_programming. (05. 05. 2020.)

[11] Numpy, The fundamental package for scientific computing with Python. (2020, June.) *Numpy.* Available at: https://numpy.org/. (05. 05. 2020.)

[12] numpy.zeros (2020, June.) *Numpy.* Available at: https://numpy.org/devdocs/reference/generated/numpy.zeros.html (11. 04. 2020.)

[13] Tripping pipe (2020, June.) *The Oilfield Glossary: Where the Oil Field Meets the Dictionary.* Available at: https://www.glossary.oilfield.slb.com/en/Terms/t/tripping_pipe.aspx. (01. 06. 2020.)

[14] Pull out of the hole (2020, June.). *The Oilfield Glossary: Where the Oil Field Meets the Dictionary.* Available at: https://www.glossary.oilfield.slb.com/en/Terms/p/pull_out_of_the_hole.aspx. (01. 06. 2020.)

[15] Well control (2020, June.). *Wikipedia.* Available at: https://en.wikipedia.org/wiki/Well_control. (22. 04. 2020.)

[16] Drilling equipment (2020, June.) *Energy Faculty.* Available at: https://energyfaculty.com/drilling-equipment/. (23. 04. 2020.)

[17] Drill String (2020, June.) *Wikipedia.* Available at: https://en.wikipedia.org/wiki/Drill_string. (26. 04. 2020.)

[18] Jabeck, B. (2020, June.) *The impact of generator set underloading.* [Online] October 2013. , *CAT.* Available at: https://www.cat.com/en_IN/by-industry/electric-power-generation/Articles/White-papers/the-impact-of-generator-set-underloading.html#:~:text=Also%2C%20a%20rich%20air%2Dto,and%20unplanned%20downtime%20or%20failure. (13. 05. 2020.)

[19] McAleese, *Operational Aspects of Oil and Gas Well Testing.* Elsevier Science, 2000.

[20][21] Detailed information for: E2B1600PR121/P-LIIn=1600A3pFHR (2020, June.) *ABB.* Available at: https://new.abb.com/products/1SDA055792R1/e2b-1600-pr121-p-li-in-1600a-3p-f-hr. (11. 05. 2020.)

[22] MAN Maintenance (2020, June.). *Marine diesel specialists.* Available at: https://marinedieselspecialists.com/man-maintenance. (02. 05. 2020.)

[23] Diesel prices, litre, 01-Jun-2020 (2020, June.). *Global Petrol Prices.* Available at: https://www.globalpetrolprices.com/diesel_prices/. (01. 06. 2020.)

[24] Annex 5: Subsidy level indicators for the case studies (2020, June.) '*Environmentally Harmful Subsidies: Identification and Assessment' A study led by IEEP, with Ecologic, IVM and Claudia Dias Soares for the European Commission, DG Environment. EUROPEAN COMMISION.* [Online]. Available at: https://ec.europa.eu/environment/enveco/taxation/pdf/Annex%205%20-%20Calculations%20from%20the%20case%20studies.pdf. (29. 05. 2020.)

[25] What is Vessel Management System? (2020, June.) *Marine Insight.* Available at: https://www.marineinsight.com/marine-navigation/what-is-vessel-management-system/#:~:text=The%20vessel%20management%20system%20uses,the%20oceanic%20a

nd%20sea%20waters.&text=The%20provision%20of%20GPS%20is,in%20a%20vessel%20management%20system. (08. 04. 2020.)

[26] Peak Shaving. (2020, June.) *Greener.* Available at: https://www.greener.nl/technology/peak-shaving/. (25. 05. 2020.)

ABBREVIATIONS AND TECHNICAL TERMS

Abbreviations:

- DB – Database
- DG – Diesel Generator
- DP – dynamic positioning
- HMI – Human Machine Interface
- PLC – Programmable Logic Controller
- PMS – Power Management System
- VFD – Variable Frequency Drive
- POOH – pulling out of the hole (*oil drilling operation*)

Technical terms:

- Python – high-level general-purpose programming language.
- SQLite – relational database management system.
- Matplotlib – plotting library for the Python programming language.
- Numpy – fundamental package for scientific computing with Python.
- Pandas – software library written for the Python programming language for data manipulation and analysis.
- DataFrame – Two-dimensional, size-mutable, potentially heterogeneous tabular data.
- DB Browser for SQLite – visual, open source tool to create, design, and edit database files
- Drawworks – primary hoisting machinery component of a rotary drilling rig.
- Topdrive – mechanical device on a drilling rig that provides clockwise torque to the drill string to drill a borehole.
- Baseload – the permanent minimum load that a power supply system is required to deliver.
- Derrick – integrated system that drills wells, such as oil or water wells, in the Earth's subsurface.
- Tripping – physical act of pulling the drill string out of the wellbore and then running it back in.
- Jackup rig – a type of mobile platform that consists of a buoyant hull fitted with a number of movable legs.

LIST OF TABLES

LIST OF FIGURES