Data scientist, 1e jaar

Thesis

Data aggregation and visualisation

Grietus Mulder

Introduction

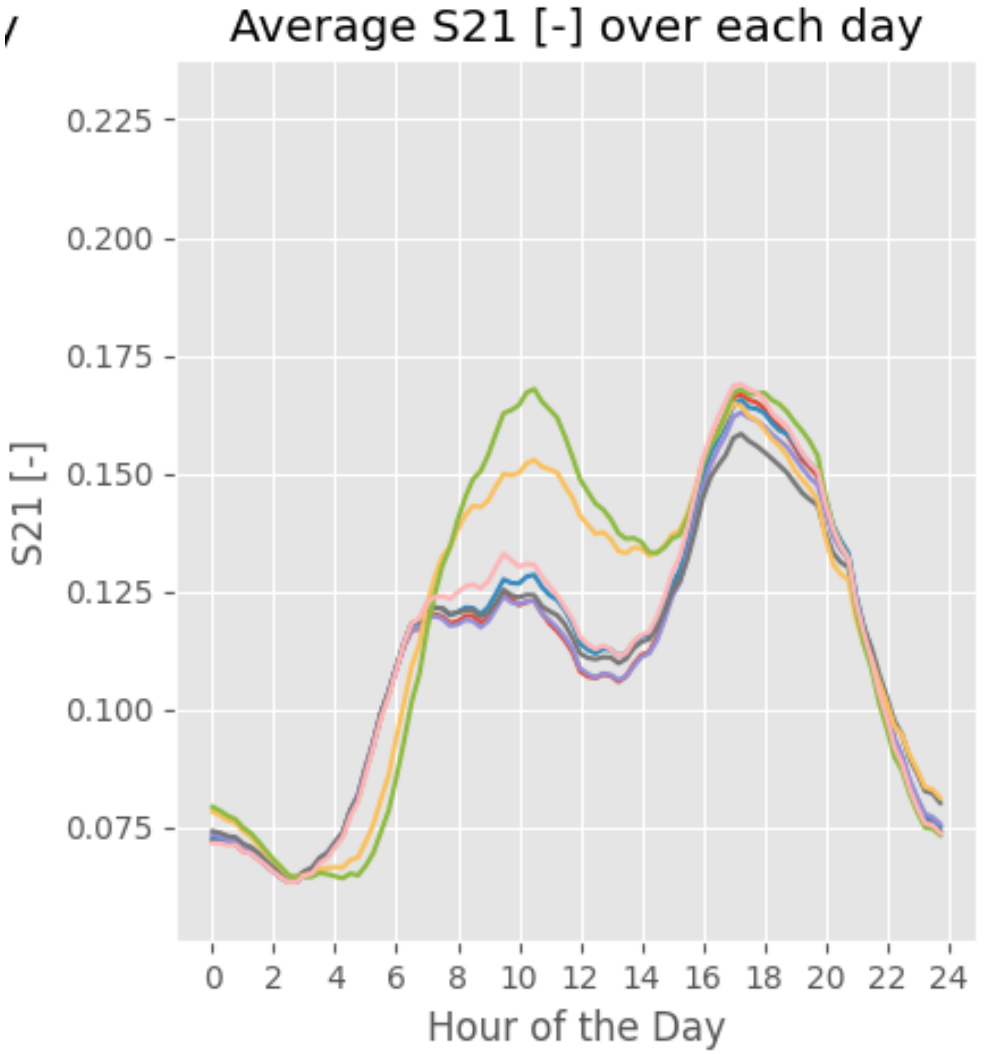
Objective

Get insight in measurement data

Emphasis in finding patterns during the day over a year

Methods to include:

- Data import (CSV and SQL)
- Data cleaning
- Data aggregation
- Data visualisation
- Preferably: adaptable looks



Average electricity consumption in Flanders for every weekday

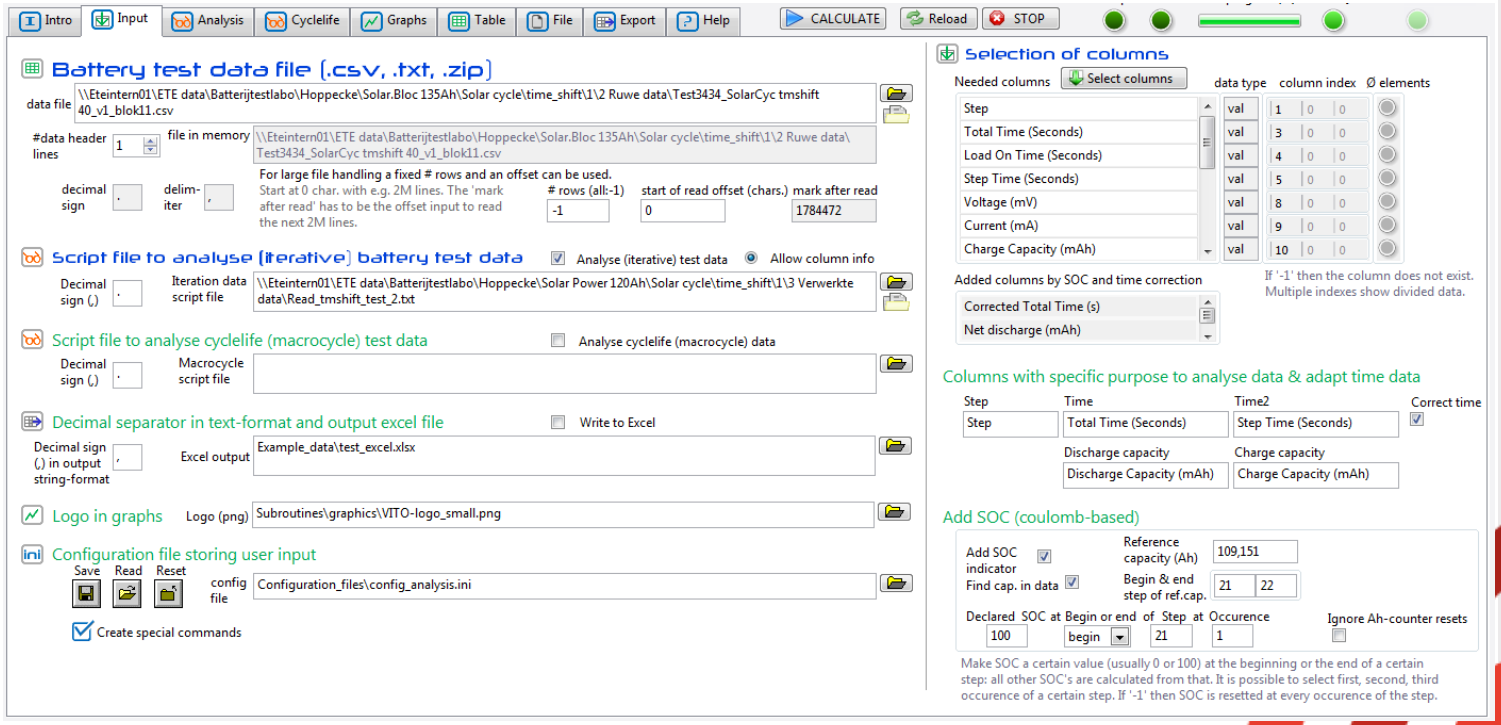
Introduction

Background

Grietus has a long history in data analysis and visualisation.

He created the BATAL suite.

This was created with LabVIEW.

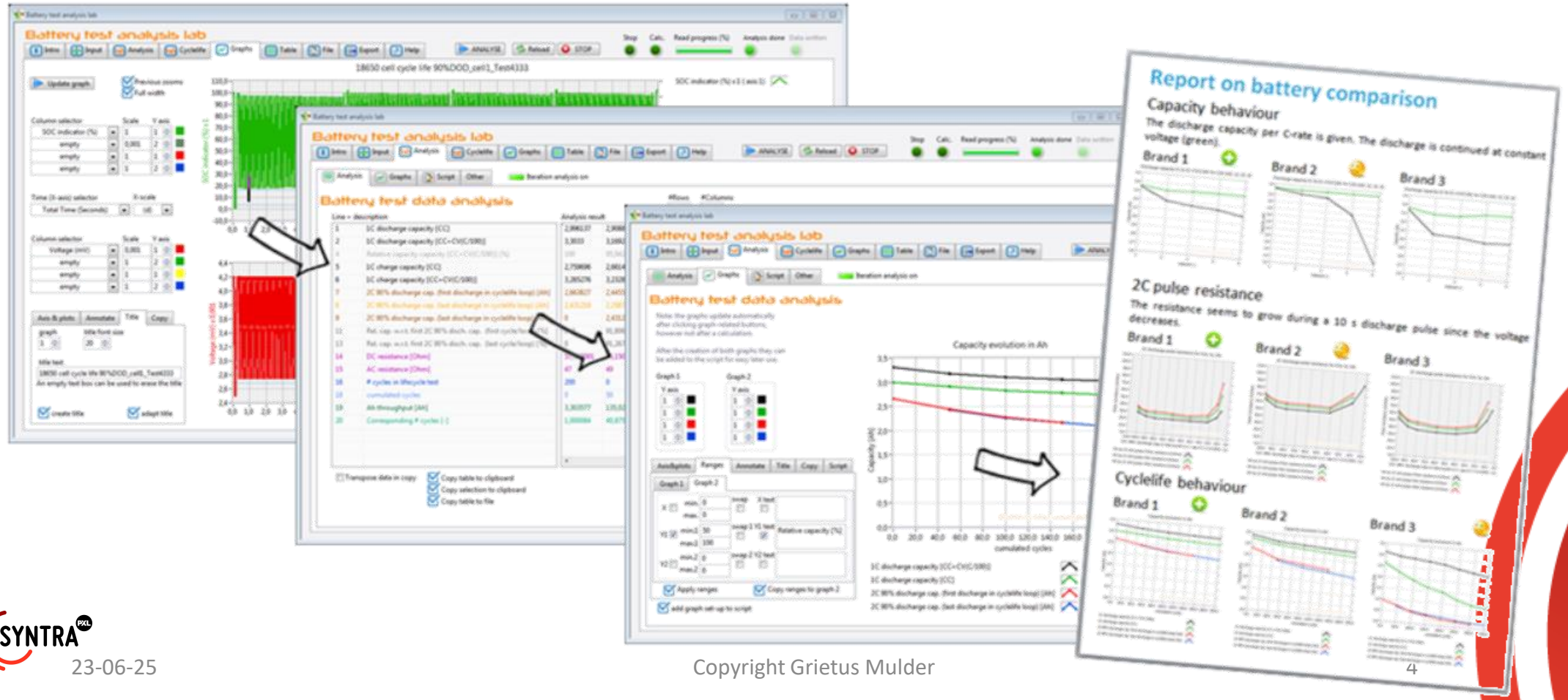


Introduction

Background

Battery test analysis lab: get quickly the figures you look for

From data graph to report with help of BATAL

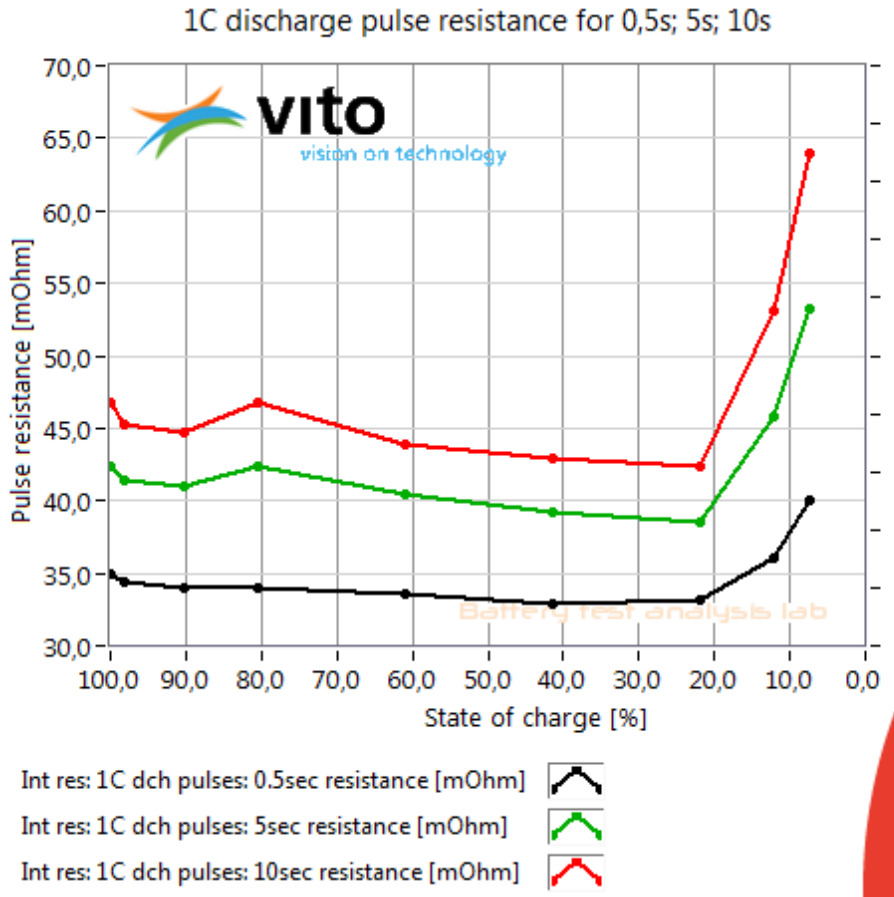


Introduction

Background

LabVIEW is powerful for creating graphical user interfaces.

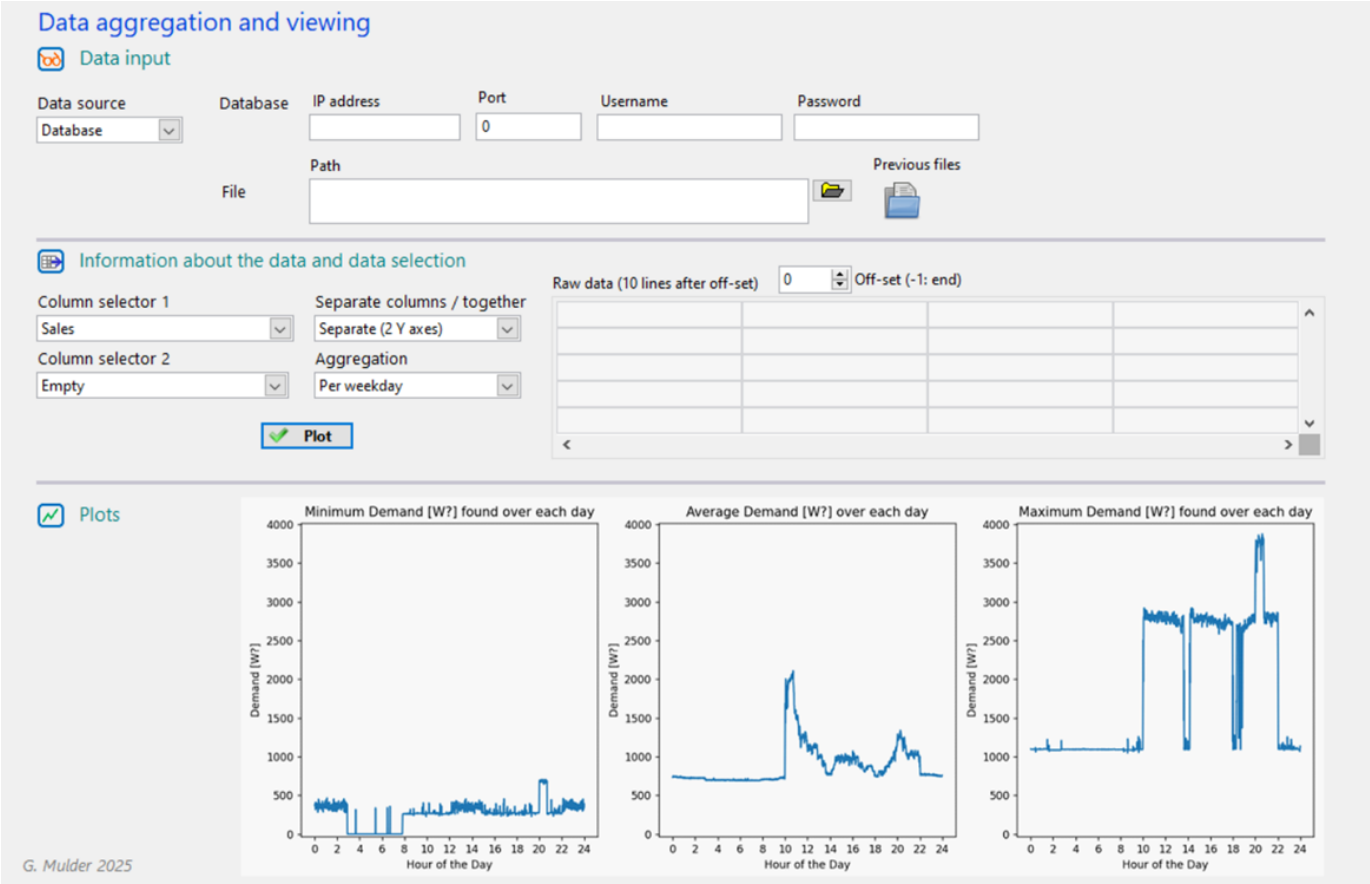
However, python is stronger in data analysis



Introduction

Background

Aimed result like it could be in LabVIEW



Core approach

One file a starting point

One file that contains :

- Datapaths
- The needed options
- The cleaning
- The aggregation
- The plotting
- Data export

Selection by uncommenting and by adding new information

Only 100 lines pure code

- Pandas is powerful

```python
'''
Read measurement data from Suriname and create plots with the average day per week
'''
import pandas as pd
import matplotlib.pyplot as plt
import os
import numpy as np
from datetime import datetime
from scipy.signal._savitzky_golay import savgol_filter

# FileWithPath = fr"C:\Users\mulderg\Downloads\1_January--GM--short.csv"
# FileWithPath = fr"C:\Users\mulderg\Downloads\1_January--GM--intermediate_360.csv"
# FileWithPath = fr"C:\Users\mulderg\Downloads\1_January--GM--intermediate_750.csv"
# FileWithPath = fr"C:\Users\mulderg\Downloads\1_January--GM--intermediate_1500.csv"
# FileWithPath = fr"C:\Users\mulderg\Downloads\1_January--GM--daytest.csv"
FileWithPath = fr"C:\Users\mulderg\Downloads\1_January.csv"
# FileWithPath = fr"C:\Users\mulderg\Downloads\10_October.csv"
# FileWithPath = fr"C:\Users\mulderg\PycharmProjects\Plot_weekday_avg_from_yeardata\data\family_solar_economic.csv"
# FileWithPath = fr"C:\Users\mulderg\Downloads\temperature_data_genk.csv"
column_name_datetime = 'Date'
# column_name_datetime = 'Datetime'
column_name_plot = 'Total_power' #'SOC' #'PVP'
# column_name_plot = 'Family [W]'
time_format = "%Y-%m-%d %H:%M:%S"
# time_format = "%d/%m/%Y %H:%M"
f_window_length = 16
f_polyorder = 4
nr_days_to_discern = 1 # For averaging daily data: set to 7 for each weekday, or 1

def printdata(name, data, nr_lines, position):
    nr_lines_corr = min(nr_lines, len(data) - position)
    # print("Lengte data: ", len(data))
    # print("nr_lines_corr: ", nr_lines_corr)
    print(f"{name}")
    print(data.iloc[position:nr_lines_corr+position])
    # print(data.iloc[position:position+nr_lines_corr])

def write_data(FileWithPath, data):
    # --- Write the cleaned data to file for later verification ---
    # Split the path and filename
    path, filename = os.path.split(FileWithPath)
    # Extract the file extension
    file_extension = os.path.splitext(filename)[1]
    # Add '--export' to the filename
    new_filename = filename.replace(file_extension, f'--export{file_extension}')
    # Create the new file path
    new_file_path = os.path.join(path, new_filename)
    # Write the data to the new CSV file with the updated filename
    data.to_csv(new_file_path, index=True, sep=';', header=True)
    print(f"Data has been written to the new file: {new_file_path}")

# --- Read the CSV file and rework the data ---
data = pd.read_csv(FileWithPath, delimiter=';')
# printdata("Original data: ", data, 10, 0)
# Convert the date column to datetime format
data[column_name_datetime] = pd.to_datetime(data[column_name_datetime], format=time_format)
# data[column_name_datetime] = pd.to_datetime(data[column_name_datetime])
```

Core approach

GUI approach

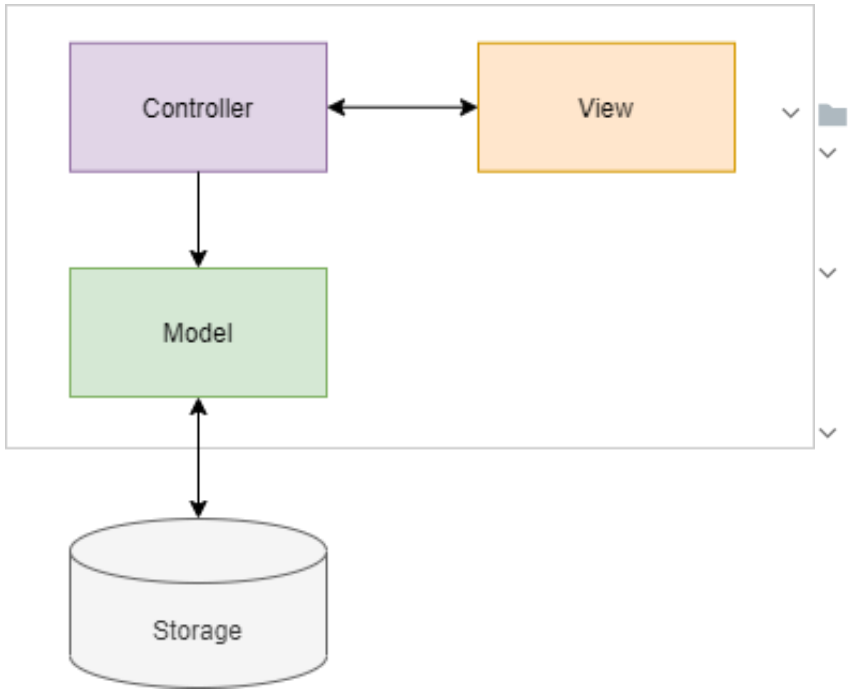
GUI is based on Custom TKInter

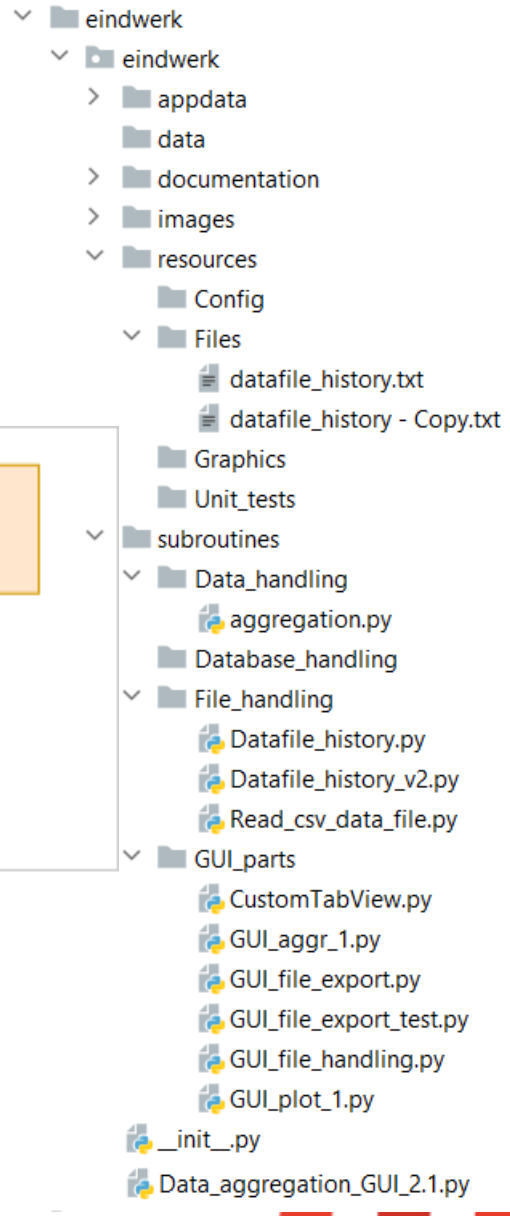
This appears to be well developed with a lot of help on the internet

- (in contrast with Kivy unfortunately)

Modular approach:

- Three levels:
  - Model-View-Control
  - All functionality in separate modules
    - Distributed GUI modules
      - Callback function from submodule to main module
    - This keeps code short (200 lines)
    - Also helpful if several data sources are used
  - No parameters in middle of code





Core approach

GUI approach

GUI is based on Custom TKInter

This appears to be well developed with a lot of help on the internet

(in contrast with Kivy unfortunately)

Modular approach:

- Three levels:
  - Model-View-Control
  - All functionality in separate modules
    - Distributed GUI modules
      - Callback function from submodule to main module
    - This keeps code short (200 lines)
    - Also helpful if several data sources are used
  - No parameters in middle of code

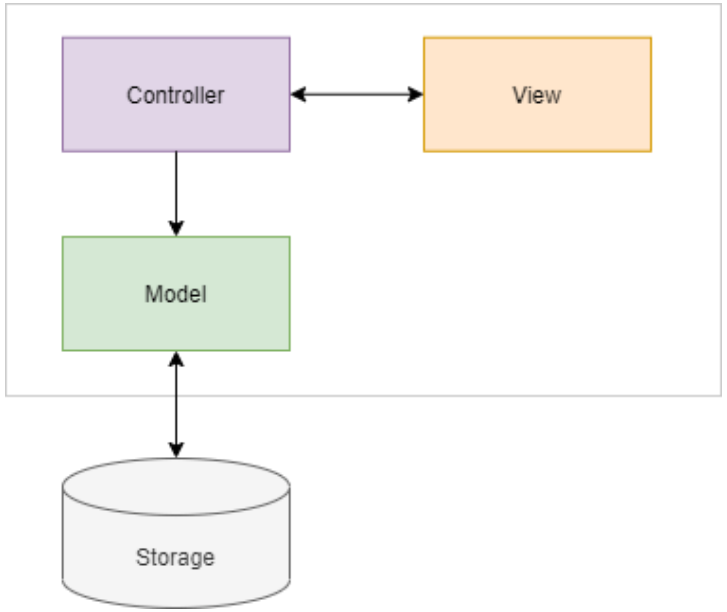
```python
# Libraries
import customtkinter as ctk

# Own routines
import DataScientist_1.GUIs.Distributed_GUI_methods.Distributed_GUI.Frame1_simple_1 as frame1

# Callback function to handle data that comes out of subframe!
def receive_data(data):
    print("data received:", data)
    data_label.configure(text=f"data: {data}")

# Call the function to place a subframe in the submodule and
# pass the callback function to the frame
def open_subframe(masterframe, callback):
    frame1.Place_frame1(masterframe, callback)

# Create the GUI
root = ctk.CTk()
root.title('Data handling and visualisation')
root.geometry("1000x600")  # width x height
root.minsize(400, 400)
```



Core approach

Data cleaning

Depending if datetime is available

Remove empty columns

Merge lines with same timestamp

Fill empty data

Add weekday

```python
def clean_data_without_datetime(data):
    # Drop rows where all columns except 'Date' are NaN
    # print("data before cleaning")
    # print(data)
    # print()
    data = data.dropna(how='all', subset=data.columns.difference(data.index))
    # printdata("Cleaned DataFrame: ", data, 10, 0)
    # Fill empty elements with previous data
    data.ffill(inplace=True)
    return data

def clean_data_with_datetime(data):
    global data_info
    # Drop rows where all columns except 'Date' are NaN
    data = data.dropna(how='all', subset=data.columns.difference([data_info[
    'datetime_column']]))
    # printdata("Cleaned DataFrame: ", data, 10, 0)
    # Merge lines with identical datetime:
    # Group by Date and aggregate the values using 'max' for all columns
    data = data.groupby(data_info['datetime_column']).max().reset_index()
    # Fill empty elements with previous data
    data.ffill(inplace=True)
    # --- add information about the weekday ---
    # data['Day'] = data[datetime_column].dt.day_name()
    data['Day_nr'] = data[data_info['datetime_column']].dt.day_of_week
    # Set the date column as the index. This only can after the .dt operations
    data.set_index(data_info['datetime_column'], inplace=True)  # inplace=True:
    Necessary to calculate avg. timestep
    # print(f"cleaned data : \n{data} \n ")
    return data
```

Core approach

Data cleaning

Depending if datetime is available

Remove empty columns

Merge lines with same timestamp

Fill empty data

Add weekday

	A	B	C	E	F	G	H	M	N	O	P	Q	U	V	W	X
1	Date	PWR	PF_LT01	DS_LT01	FQDistr_D\	FQDistr_M	PV1V	PVP	SOC	BattP	BattV	NF_PWP0:	DS_PWP01	DS_PWP02	NF_FT03	DS_FT01
338	01/01/2025 04:00:38						0.04	0	87	-473	51.4					
339	01/01/2025 04:00:48			72.77971	0.04	0.04							127.8124	128.5449		0.015649
340	01/01/2025 04:00:58			72.77971	0.04	0.04							126.7138	128.1787		0.015649
341	01/01/2025 04:01:08			72.77971	0.04	0.04							128.5449	128.1787		0.015649
342	01/01/2025 04:01:08						0.03	0	87	-474	51.4					
343	01/01/2025 04:01:18			72.77971	0.04	0.04							126.3475	126.7138		0.015649
344	01/01/2025 04:01:28			72.77971	0.04	0.04							130.0098	128.5449		0.015649
345	01/01/2025 04:01:38			72.77971	0.04	0.04							127.8124	126.7138		0.015649
346	01/01/2025 04:01:38						0.03	0	87	-475	51.4					
347	01/01/2025 04:01:48			72.77971	0.04	0.04							126.3475	128.5449		0.015649
348	01/01/2025 04:01:58			72.77971	0.04	0.04							126.3475	128.1787		0.015649
349	01/01/2025 04:02:08			72.77971	0.04	0.04	0.03	0	87	-474	51.4		126.7138	128.1787		0.015649
350	01/01/2025 04:02:38						0.03	0	87	-471	51.4					
351	01/01/2025 04:02:39			72.77971	0.04	0.04							129.6436	128.1787		0.015649
352	01/01/2025 04:02:49			72.77971	0.04	0.04							126.7138	126.7138		0.015649
353	01/01/2025 04:03:09						0.03	0	87	-516	51.4					
354	01/01/2025 04:03:09			72.77971	0.04	0.04							211.311	126.7138		1.451216
355	01/01/2025 04:03:19			72.77971	0.05	0.05							350.8416	128.5449		1.392621
356	01/01/2025 04:03:29			72.75066	0.05	0.05							1.465888	126.3475		0.015649
357	01/01/2025 04:03:39						0.03	0	87	-652	51.3					
358	01/01/2025 04:03:39			72.75066	0.05	0.05							324.1074	127.8124		1.61235
359	01/01/2025 04:03:49			72.7216	0.06	0.06							350.8416	124.5164		1.363324
360	01/01/2025 04:04:00			72.7216	0.06	0.06							105.4729	128.1787		0.015649
361	01/01/2025 04:04:01	0.365224														
362	01/01/2025 04:04:09						0.03	0	87	-625	51.3					
363	01/01/2025 04:04:10			72.75066	0.06	0.06							241.7074	126.7138		0.015649
364	01/01/2025 04:04:20			72.7216	0.06	0.06							223.0301	128.5449		0.015649
365	01/01/2025 04:04:30			72.7216	0.06	0.06							0.001	128.1787		0.015649

Example

Sample data fom the file # rows: 283989 # columns: 26

```
Date;PWR;PF_LT01;NF_LT01;DS_LT01;FQDistr_DV;FQDistr_MV;PV1V;PV2V;P
2025-01-01 03:00:01;;;;72.9540329;0;0;;;;;100;;;;;;;126.713768;128.1786652;;0
2025-01-01 03:00:11;;;;72.9540329;0;0;;;;;;;;;;;;;126.3475494;128.1786652;;0.01
2025-01-01 03:00:18;0.323109388;;;;;;;;;;;;;;;;;;;;;;;;
2025-01-01 03:00:21;;;;72.9540329;0;0;;;;;;;;;;;;;128.1786652;128.1786652;;0.01
```

Cleaned data: # rows: 277197 # columns: 26

avg. step: 00:00:10 std. dev.: 00:00:07 max.step: 00:07:29

Core approach

Data aggregation

Identify the weekdays

Identify the same hours, minutes

Aggregate by groupby

and indexes

Determine minimum, average and maximum

```python
# Aggregate the data
def aggregate(data, data_info, nr_days_to_discern):
    aggregation = {'average': None, 'minimum': None, 'maximum': None}
    #--- Calculate minimum/average/maximum per (week)day (depending on
    'nr_days_to_discern')

    # Get start date. This is helpful for later operations when averaging days.
    # base_date = data[data_info['datetime_column']][0].date()  # Get first datetime
    base_date = data.index[0].date()  # Get first datetime
    base_date = pd.to_datetime(base_date)  # Ensures that the date starts at 00:00
    average_profiles = data.groupby([data.index.hour, data.index.minute, data['Day_nr']
    % nr_days_to_discern]).mean().unstack()
    average_profiles['time_minutes'] = average_profiles.index.get_level_values(0)*60 +
    average_profiles.index.get_level_values(1)
    average_profiles['Datetime_(day)'] = base_date + pd.to_timedelta(average_profiles[
    'time_minutes'], unit='m')
    minimum_profiles = data.groupby([data.index.hour, data.index.minute, data['Day_nr']
    % nr_days_to_discern]).min().unstack()
    maximum_profiles = data.groupby([data.index.hour, data.index.minute, data['Day_nr']
    % nr_days_to_discern]).max().unstack()
    aggregation['average'] = average_profiles
    aggregation['minimum'] = minimum_profiles
    aggregation['maximum'] = maximum_profiles
    return aggregation
```

Core approach

CSV reading

This appears weak in Python

No good functionality to discover data structure

- Needed:
  - Start of data
  - Column names in 1 or divided over 2 lines
  - Delimiter
  - Decimal sign

CleverCSV appears a bit better than CSV

- Correct delimiter recognition

To be done

Solve path dependency in submodules

Submodules cannot straightforwardly be tested:

Paths start from main module

(going backwards is not possible)

Data_aggregation_GUI_1.4_2.py | GUI_file_handling.py

```
18  # Own routines
19  if __name__ != "__main__":
20      import  subroutines.File_handling.Datafile_history_v2 as hist
21      import subroutines.File_handling.Read_csv_data_file as rd_csv
22  else:
23      import ..\\..\subroutines.File_handling.Datafile_history_v2 as hist
24      import ..\\..\subroutines.File_handling.Read_csv_data_file as rd_csv
25
```

To be done

Package creation

Configuration files for parameters

__init__.py application

Path encapsulation

Error capture with output to user

- Errors are now ‘invisible’: stay in compiler

Unit tests

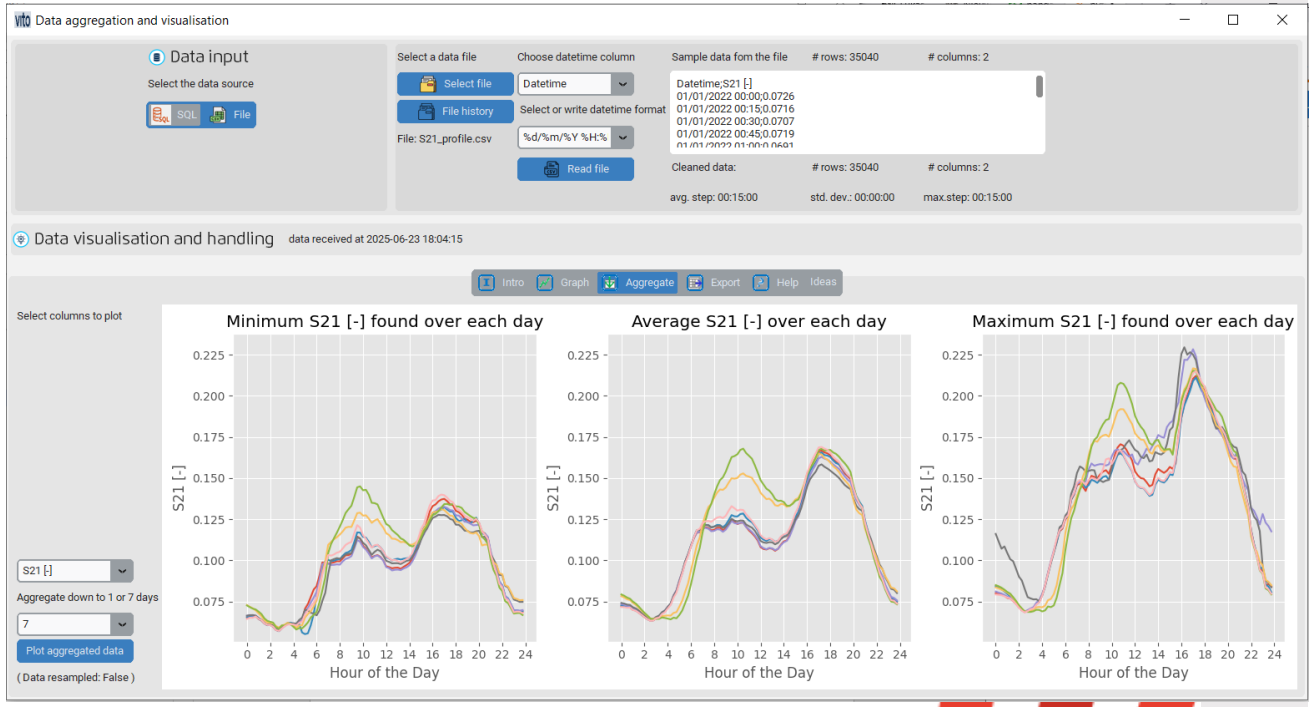
Executable

Documentation

Result

GUI creation

GUI: original code x 20 …



Take aways/ experiences

Plus

- Pandas for python is powerful in data aggregation
- Copilot can assist well
  - Certainly to find small errors
- Learning by doing
- '__name__ = " __main__' : helpful for testing and adapting modules
  - (if path dependency is solved)
- GUI distribution approach can be shared if wanted

Take aways/ experiences

Minus

- Dictionaries with parameters difficult to manage between modules
  - No single point of definition
- CSV reading not well established
- Architectures seem not common
- Coding a GUI is a real effort
  - Code exploses
- GUI module distribution with callback towards main module is necessary
  - No ‘return’ possible at end of function
- Applying the index in pandas is combination with graphs is not easy
  - The column name is lost temporarily