# Data Format for Wind Energy Applications

Table of Contents

[Data Format for Wind Energy Applications 1](#_Toc170379814)

[1. Introduction 1](#_Toc170379815)

[2. Organizing data 2](#_Toc170379816)

[3. Formatting Data for Programming Languages 2](#_Toc170379817)

[3.1. Importance of Proper Data Formatting 3](#_Toc170379818)

[3.2. Best Practices for Data Formatting 3](#_Toc170379819)

[4. Exporting Data 9](#_Toc170379820)

[5. Quality check 10](#_Toc170379821)

[5.1. Data Management and Organization 10](#_Toc170379822)

[5.2. Metadata and Documentation 11](#_Toc170379823)

[5.3. Data Storage and Preservation 11](#_Toc170379824)

[5.4. Data Quality Assurance 12](#_Toc170379825)

[6. Data Sharing Protocol 13](#_Toc170379826)

[6.1. Data Sharing and Access 13](#_Toc170379827)

[6.2. Ethical and Legal Considerations 16](#_Toc170379828)

[6.3. Roles and Responsibilities 17](#_Toc170379829)

[6.4. Data Sharing Workflow 17](#_Toc170379830)

[6.5. Support and Training 17](#_Toc170379831)

[6.6. Monitoring and Evaluation 18](#_Toc170379832)

[7. Git for controlling version 18](#_Toc170379833)

[7.1. Git settings 19](#_Toc170379834)

[7.2. Git help manual 19](#_Toc170379835)

[7.3. Creating a Git Repository 20](#_Toc170379836)

[7.4. Remotes in GitHub 21](#_Toc170379837)

[7.5. Connect Local to Remote Repository 22](#_Toc170379838)

[7.6. SSH background and setups 22](#_Toc170379839)

[7.7. Push Local Changes to a Remote Repository 24](#_Toc170379840)

[7.8. Pulling Changes from the Remote Repository: 25](#_Toc170379841)

[7.9. Exploring History 25](#_Toc170379842)

## Introduction

Proper data management and sharing are crucial for enhancing the reproducibility, reliability, and collaborative potential of research in wind energy applications. This document outlines best practices for organizing, formatting, and sharing data to ensure seamless integration with computational tools like Python and R.

## Organizing data

Effective data organization is essential for maximizing the utility of datasets in computational analyses. While spreadsheets are commonly used for data entry, their role often extends to creating tables for publications, generating summary statistics, and producing figures. However, these tasks can be challenging in spreadsheets due to the need for special formatting and the potential for unpredictable results when transferring data to document software. Additionally, the graphical interface of spreadsheets makes it difficult to replicate complex calculations, increasing the risk of inconsistencies.

Command-line-based programs like Python or R offer more reliability by ensuring uniform application of calculations across datasets. This section will outline best practices for organizing and formatting your data for seamless use in these programming environments.

## Formatting Data for Programming Languages

Proper data formatting is crucial when working with programming languages like Python to avoid issues related to data manipulation and analysis. Consistent and clean data formatting ensures that your code runs smoothly, without errors related to data types or mismatched formats.

In this section, we will explore how to format your data in text format for use in Python, including tips on data cleaning and preparation. Ensuring your data is properly formatted before importing it into Python can save a significant amount of time and prevent potential errors.

We will guide you through the process of preparing your data in a text-based format ready for analysis. This includes:

* Data Cleaning: Removing any inconsistencies, duplicates, and errors in your dataset to ensure accuracy.
* Data Formatting: Ensuring all data is in a consistent format, such as CSV, which is easily readable by Python.
* Data Validation: Checking for and correcting any issues related to data types, missing values, and formatting errors.
* Preliminary Analysis: Generating summary statistics to understand the basic properties of your dataset and identify any potential issues early on.

By following these best practices, you can enhance the reliability and efficiency of your data analysis workflows, ultimately leading to more robust and reproducible results.

### Importance of Proper Data Formatting

A common mistake is treating data files like notebooks, relying on context, notes, and spatial layout to convey information. While humans can interpret these cues, computers require explicitly defined structures. Properly formatted data allows you to harness the power of programming languages for effective and efficient data analysis. Consistently formatted data ensures that computational tools can accurately and reliably process your information, making your analyses more robust and reproducible.

### Best Practices for Data Formatting

#### Save each data file separately

The data should be presented in table format where the first row of the data file should contain variables. In data files saved from commercial software, the first several lines contain information about the date, site, software, etc. This metadata is very useful for the audience to check; however, it is not programming-friendly. In the raw text file, the data should start with the first row containing column headers, such as variable names.

Ensure each text file contains only one table. Each row should represent a distinct observation, and each column should represent a variable.

If the data is saved in spreadsheet format with many tabs, combine data from all tabs into a single text file, adding additional columns to differentiate data where necessary. Alternatively, separate each tab into individual text files.

#### Including Metadata in Data Table

Adding metadata directly into the data table disrupts data parsing. Store metadata in a separate file. For example, the first 9 lines in the raw file *Turbine\_Data\_Kelmarsh\_1\_2022-01-01\_-\_2023-01-01\_228.csv* include metadata that can create issues when loading the data in Python. Please refer to the RRPythonTutorial.ipynb to check the error.

Example:

# This file was exported by Greenbyte at 2023-08-15 14:16:56. Please see https://www.greenbyte.com for more information about Greenbyte.

#

# Turbine: Kelmarsh 1

# Turbine type: Senvion MM92

# Time zone: UTC

# Time interval: 2022-01-01 00:00:00 - 2023-01-01 00:00:00 (365 days)

#

# Data that is missing or is erroneous has been marked with the value "NaN"

#

#### Not Filling in Zeros

Leaving cells blank instead of filling in zeros can lead to misinterpretation of data as missing rather than zero. Explicitly enter zeros where applicable to indicate a measured value of zero.

Example:

Date\_and\_time,Wind\_speed,Wind\_speed\_Standard\_deviation

2022-01-01 00:00:00,0,0

2022-01-01 00:10:00,6.936052060127258,1.2872216897689304

#### Using Problematic Null Values

Using arbitrary values like -999 or 999 to indicate missing data can cause issues in analysis. Use a consistent and appropriate null indicator like NA or leave cells blank.

Example:

Date\_and\_time,Wind\_speed,Wind\_speed\_Standard\_deviation

2022-01-01 00:00:00,6.781222248077393,1.182439280264067

2022-01-01 00:10:00, ,1.2872216897689304

#### Consistent Delimiters:

When working with text data, it's crucial to use consistent delimiters to separate fields within your dataset. Delimiters are characters used to define the boundary between different data elements. Common delimiters include commas, tabs, semicolons, and spaces.

Common Delimiters and Their Uses:

* Commas (,): Used in CSV (Comma-Separated Values) files, one of the most widely accepted formats. CSV files are supported by most data processing tools and spreadsheet applications.
* Tabs ( ): Used in TSV (Tab-Separated Values) files. Tab delimiters are useful when data contains commas, as tabs are less likely to be present in the actual data values.
* Semicolons (;): Sometimes used in regions where commas are used as decimal points. They provide an alternative to commas in CSV files.
* Spaces: Rarely used because spaces can easily be part of the data, making it hard to distinguish between delimiters and actual data content.

Choosing the right delimiter is crucial for ensuring the accuracy and ease of data processing. It's essential to select a delimiter that does not appear in your data values to avoid confusion. For instance, if your dataset includes commas, it’s better to use tabs or semicolons as delimiters.

Consistency is also key; using the same delimiter throughout the dataset prevents parsing errors and incorrect data interpretation. Additionally, using appropriate file extensions that indicate the delimiter type, such as .csv for comma-separated values and .tsv for tab-separated values, helps in clearly identifying the file format and ensures compatibility with various data processing tools.

Example:

Date\_and\_time,Wind\_speed,Wind\_speed\_Standard\_deviation

2022-01-01 00:00:00,6.781222248077393,1.182439280264067

2022-01-01 00:10:00,6.936052060127258,1.2872216897689304

#### Uniform Data Types:

Ensure each column contains data of the same type. For example, a Wind\_speed column should only have numeric values. Consistent data types within columns are critical for accurate analysis and processing. Mixing data types, such as text and numbers in the same column, can lead to errors and misinterpretations during data analysis. Therefore, it is essential to validate and clean your data to maintain uniform data types across all columns. This practice ensures that computational tools can process the data efficiently and correctly, facilitating more reliable and meaningful analysis results.

Example:

Date\_and\_time,Wind\_speed,Wind\_speed\_Standard\_deviation

2022-01-01 00:00:00,6.781222248077393,1.182439280264067

2022-01-01 00:10:00,6.936052060127258,1.2872216897689304

#### Avoid Special Characters:

Avoid using spaces, special characters, or starting field names with numbers. Remove or escape special characters that might interfere with data parsing. Use descriptive, alphanumeric field names with underscores or camel case within text fields. This practice ensures that field names are clear, consistent, and compatible with various data processing tools.

Special characters and spaces can cause issues in data parsing and processing, leading to errors and misinterpretations. By adhering to these naming conventions, you can maintain the integrity of your data and facilitate easier data handling and analysis. For example, instead of using a field name like "Wind speed (m/s)," use "Wind\_speed\_mps" or "windSpeedMps" to ensure compatibility and readability.

Example:

# Date and time,Wind speed (m/s),"Wind speed, Standard deviation (m/s)"

2022-01-01 00:00:00,6.781222248077393,1.182439280264067

2022-01-01 00:10:00,6.936052060127258,1.2872216897689304

#### Standardize Date Formats:

Storing dates in a single column might initially seem convenient, but it can introduce significant ambiguities and compatibility issues. Different software and programming environments may interpret the same date format differently, leading to potential errors and inconsistencies in data processing. For instance, the date "03/04/2021" might be interpreted as March 4th, 2021, in one context and April 3rd, 2021, in another, depending on regional settings and software defaults.

Separate Year, Month, and Day: To eliminate ambiguity and make date manipulation easier programmatically, it is recommended to store dates as separate fields for the year, month, and day. This approach ensures that each component of the date is clearly defined and less prone to misinterpretation by various software tools.

Example:

Year,Month,Day

2015,03,24

Use Unambiguous Date Formats: When storing dates as strings, use a consistent and sortable format like YYYYMMDD for dates and YYYYMMDDhhmmss for date-times. This format is both unambiguous and easily sortable, which is crucial for maintaining data integrity across different systems and applications.

Examples:

Date

20150324

Handle Missing Dates Properly: Missing dates should be clearly indicated to avoid misinterpretation. Use a consistent placeholder such as NA or leave the cells blank. This practice ensures that the absence of data is explicitly recognized as missing rather than misinterpreted as a default or erroneous value.

Example:

Year,Month,Day

2015,03,24

NA,NA,NA

#### By following the best practices, you can ensure that your dates are handled correctly and consistently in programming environments, minimizing errors and enhancing the reliability of your data analysis.

#### Treat Dates as Multiple Pieces of Data: Store year, month, and day as separate fields rather than a single entry to reduce ambiguity.

#### Use Consistent, Unambiguous Formats: Adopt formats like YYYYMMDD for dates and YYYYMMDDhhmmss for date-times to ensure clarity and consistency.

#### Clearly Indicate Missing Dates: Use placeholders like NA or leave cells blank to clearly denote missing values.

#### Utilize Built-in Functions for Date Handling: In programming environments, leverage built-in functions to extract and manipulate date components effectively.

#### Maintain Raw Data Integrity:

Never modify your raw data directly. Always create a copy for any cleaning or analysis to ensure reproducibility. This practice safeguards the original dataset from accidental changes or corruption, maintaining its integrity. By working on a copy, you can track modifications and ensure that any changes made during data cleaning or analysis are documented and reversible.

This approach also facilitates reproducibility, as others can replicate your work using the unaltered raw data. Proper documentation of all steps taken in data processing, along with the use of version control systems, further enhances transparency and reliability in your research.

#### Keep Track of Data Cleaning Steps:

Document all data cleaning steps in a plain text file stored in the same folder as the data file. This practice ensures that your analysis is reproducible and transparent.

By maintaining a detailed record of each modification, transformation, or correction applied to the data, you enable others to follow the same process and verify results. This documentation should include descriptions of the actions taken, the rationale behind each step, and any scripts or commands used during the cleaning process. Clear and thorough documentation not only supports reproducibility but also aids in troubleshooting and refining data processing workflows.

#### Using Formatting to Convey Information

Using cell formatting (e.g., color, bold text) to convey information that isn't captured in the data values themselves can be problematic. Use additional columns to encode any extra information.

Formatting changes are not recognized by most data analysis software, which rely solely on the actual data values. By including additional columns for any extra information or annotations, you ensure that all necessary details are captured in a format that is both machine-readable and human-readable. This approach enhances data transparency and prevents the loss of important context during the data import and analysis stages. For instance, instead of using color to highlight rows with errors, create a new column labeled "Status" and use descriptive entries such as "checked" or "unchecked" to indicate the status.

Example:

Date\_and\_time,Wind\_speed,Wind\_speed\_Standard\_deviation, Status

2022-01-01 00:00:00,6.781222248077393,1.182439280264067, checked

2022-01-01 00:10:00,6.936052060127258,1.2872216897689304, unchecked

#### Placing Comments or Units in Cells

Including comments or units directly in data cells can cause issues. Use separate columns for comments and ensure all measurements in a column are in the same unit, specifying units in the column header. This ensures that the data remains clean and interpretable by analysis software, which may not correctly parse mixed content in a single cell. For instance, rather than embedding comments within data cells, create a dedicated comments column. Similarly, ensure all measurements in a column use the same unit and indicate the unit in the column header. Additionally, consider saving the units for each column in a separate raw text file, such as *‘./data/Kelmarsh\_SCADA\_2022\_clean/data.txt’*, to provide clear documentation of the measurement units used. This practice maintains the integrity and clarity of your dataset, facilitating more accurate analysis.

Example:

Date\_and\_time,Wind\_speed,Wind\_speed\_Standard\_deviation, Failure

2022-01-01 00:00:00,6.781222248077393,1.182439280264067,

2022-01-01 00:10:00,6.936052060127258,1.2872216897689304, “Gearbox error”

#### Entering Multiple Pieces of Information in One Cell

Combining multiple pieces of information within a single cell can complicate data analysis. Separate data into distinct columns. This approach enhances the clarity and usability of the dataset, allowing each piece of information to be accessed and analyzed independently. For instance, instead of combining failure status and component details within a single cell, use separate columns for each attribute.

Example:

Date\_and\_time,Wind\_speed,Wind\_speed\_Standard\_deviation, Failure, Component

2022-01-01 00:00:00,6.781222248077393,1.182439280264067, ,

2022-01-01 00:10:00,6.936052060127258,1.2872216897689304, Yes, “Gearbox”

This practice ensures that each data point is clearly defined and easily interpretable by analysis software, improving the accuracy and efficiency of data processing.

#### Structuring Data for Programming

The cardinal rule for data structure is to keep it "tidy." This means:

* Columns = Variables: Each column should represent a variable (e.g., Wind\_speed, Power).
* Rows = Observations: Each row should represent a single observation or record.
* Cells = Values: Each cell should contain a single data value.

Maintaining this structure ensures that your data is easy to manipulate and analyze using programming languages like Python or R. Well-structured, or "tidy," data simplifies the process of data cleaning, visualization, and analysis, allowing for more efficient and accurate workflows.

Example of well-structured data:

Date\_and\_time,Wind\_speed,Wind\_speed\_Standard\_deviation

2022-01-01 00:00:00,6.781222248077393,1.182439280264067

2022-01-01 00:10:00,6.936052060127258,1.2872216897689304

#### Key Points

By following these practices, you ensure that your data is ready for analysis in programming languages like Python, avoiding common pitfalls related to data formatting and quality:

* Never modify your raw data directly. Always work on a copy.
* Use consistent formatting and delimiters.
* Clearly indicate missing values.
* Maintain uniform data types within columns.
* Document all data cleaning steps for reproducibility.

By adhering to these guidelines, you can significantly enhance the reliability and efficiency of your data analysis workflows, ultimately leading to more robust and reproducible results.

## Exporting Data

When preparing data for analysis in programming environments such as Python, exporting it from spreadsheets in a way that is both universal and reliable is crucial. This guide will help you understand why using common spreadsheet formats is problematic and how to properly export your data to formats like CSV or TSV to ensure compatibility and accuracy.

Issues with Proprietary Formats: Storing your data in proprietary formats such as Excel's default (\*.xls or \*.xlsx) can lead to several issues:

* Proprietary Nature: Future technology might not support these formats.
* Compatibility: Other software might not open these files correctly.
* Version Differences: Different Excel versions might handle data differently.
* Repository Requirements: Most data repositories do not accept proprietary formats.

To avoid these issues, use universal, open, and static formats like CSV (Comma-Separated Values) or TSV (Tab-Separated Values). These formats are plain text files that can be opened and read by almost any software, ensuring maximum portability and longevity.

Exporting Data to CSV from Excel: To save a file in CSV format from Excel:

* Open your file in Excel.
* Select "File" > "Save As".
* Choose "Comma Separated Values" (\*.csv) from the format list.
* Check the file name and save location, then click "Save".

You can open CSV files in Excel, but ensure compatibility with other programs by verifying the format after saving.

Example of problematic CSV data:

Date\_and\_time,Wind\_speed,Wind\_speed\_Standard\_deviation

2022-01-01 00:00:00,6.781222248077393,1.182439280264067

2022-01-01 00:10:00,6.936052060127258,1.2872216897689304

TSV as a solution:

Date\_and\_time Wind\_speed Wind\_speed\_Standard\_deviation

2022-01-01 00:00:00 6.781222248077393 1.182439280264067

2022-01-01 00:10:00 6.936052060127258 1.2872216897689304

By following these practices, you ensure your data is in a robust format for analysis in programming environments, minimizing errors and maximizing compatibility.

## Quality check

### Data Management and Organization

#### **Data Collection Methods**

The author of the data should explain how data is collected. Data is collected using various instruments, such as anemometers and weather stations, strategically placed at different locations. These instruments measure wind speed, direction, temperature, and other relevant meteorological parameters. Data is recorded at regular intervals (e.g., every 10 minutes) and transmitted to a central database. Detailed procedures for setting up and calibrating instruments, as well as data logging protocols, are documented in the instrument manuals and data collection SOPs.

For instance, in the file./data/Kelmarsh\_SCADA\_2022/SCADA/Turbine\_Data\_Kelmarsh\_1\_2022-01-01\_-\_2023-01-01\_228.csv, the below preamble provides some metadata about the data, such as the turbine type, time zone, and the time interval of data recording. However, it lacks crucial details on how the data was collected, including the specific locations and positions of the turbines, and the precise intervals at which data was recorded. This omission can lead to uncertainties about the data's accuracy and applicability. For robust data collection and sharing protocols, it is essential to include comprehensive metadata that describes not only the time and duration of data collection but also the exact methods and instruments used, their calibration procedures, and the geographic and positional context of the data sources. This detailed documentation helps in verifying the data's integrity and ensures it can be effectively used for further analysis and research.

# This file was exported by Greenbyte at 2023-08-15 14:16:56. Please see https://www.greenbyte.com for more information about Greenbyte.

#

# Turbine: Kelmarsh 1

# Turbine type: Senvion MM92

# Time zone: UTC

# Time interval: 2022-01-01 00:00:00 - 2023-01-01 00:00:00 (365 days)

#

"# Data that is missing or is erroneous has been marked with the value ""NaN"""

#

#### **Data Structure and Format**

Data files should be structured in a consistent format to facilitate analysis and sharing. Use formats such as CSV or TSV, adhering to the tidy data principles where each column represents a variable, each row represents an observation, and each cell contains a single value. This ensures compatibility with analysis tools like Python and R.

#### **File Naming Conventions**

File names should be descriptive and consistent to allow easy identification and retrieval. A recommended format is: ProjectName\_Location\_Instrument\_DateRange.csv. For example, WindEnergy\_Kelmarsh\_Anemometer\_20220101-20221231.csv. Avoid using spaces, special characters, or overly long names.

### Metadata and Documentation

#### **Metadata Standards**

Metadata should be included to provide context and additional information about the data. Required metadata fields include the collection date, location, units of measurement, and any preprocessing steps performed. Use standardized templates to ensure consistency.

For example, the previous data preamble in section 5.1.1, while it provides some metadata like the time zone and turbine type, it lacks standardized metadata fields such as specific location details, units of measurement for each parameter, and descriptions of any preprocessing steps taken. This highlights the necessity of adhering to metadata standards to provide a complete picture of the dataset.

#### **README Files**

Each dataset should be accompanied by a README file that explains the contents, structure, and any preprocessing steps. The README file should include details such as the purpose of the dataset, data collection methods, variable descriptions, and any known issues or limitations.

For example, the data collection methods should detail how data was collected using anemometers and weather stations, recorded at regular intervals, and transmitted to a central database. Although not explicitly stated in the provided example, this information can be inferred from the context. For completeness, explicitly include: "Data was collected using anemometers and weather stations placed at strategic locations. Data was recorded every 10 minutes and transmitted to a central database." The file lists all the variables, such as wind speed, direction, and energy metrics, along with their units, like "Wind speed (m/s), Wind speed, Standard deviation (m/s), Energy Export (kWh), etc." Preprocessing steps are mentioned as "This folder contains data which has been preprocessed to make clean header files," including removing "#" from date and time, removing quotes from feature names, replacing spaces with underscores, and listing units separately. While the provided text does not list known issues or limitations, it is important to include such details to alert users of potential data quality issues or gaps, for example, "Known issues: Some data points may be missing or marked as 'NaN' due to sensor errors or maintenance periods."

# README for Kelmarsh SCADA Data

## Purpose

Scada data collected saved for learning purposes.

## Data Collection Methods

Data was collected using anemometers and weather stations placed at strategic locations. Data was recorded every 10 minutes and transmitted to a central database.

## Preprocessing Steps

- Removed "#" at Date and time

- Removed quotes in the names of data features

- Replaced spaces in data feature names with underscores

- Removed the units of each feature and listed them below

## Variable Descriptions and Units

- Date and time

- Wind speed (m/s)

- Wind speed, Standard deviation (m/s)

- Wind speed, Minimum (m/s)

- Wind speed, Maximum (m/s)

- Long Term Wind (m/s)

- Wind speed Sensor 1 (m/s)

- Wind speed Sensor 1, Standard deviation (m/s)

- Wind speed Sensor 1, Minimum (m/s)

- Wind speed Sensor 1, Maximum (m/s)

- Wind speed Sensor 2 (m/s)

- Wind speed Sensor 2, Standard deviation (m/s)

- Wind speed Sensor 2, Minimum (m/s)

- Wind speed Sensor 2, Maximum (m/s)

- Density adjusted wind speed (m/s)

- Wind direction (degrees)

- Nacelle position (degrees)

- Wind direction, Standard deviation (degrees)

- Wind direction, Minimum (degrees)

- Wind direction, Maximum (degrees)

- Nacelle position, Standard deviation (degrees)

- Nacelle position, Minimum (degrees)

- Nacelle position, Maximum (degrees)

- Vane position 1+2 (degrees)

- Energy Export (kWh)

- Energy Export counter (kWh)

- Energy Import (kWh)

- Energy Import counter (kWh)

- Lost Production (kWh)

- Lost Production to Curtailment (kWh)

- Power (kW)

- Power, Standard deviation (kW)

- Power, Minimum (kW)

- Power, Maximum (kW)

- Apparent power (kVA)

- Apparent power, Max (kVA)

- Apparent power, Min (kVA)

- Apparent power, StdDev (kVA)

- Front bearing temperature (°C)

- Rear bearing temperature (°C)

- Stator temperature 1 (°C)

- Nacelle ambient temperature (°C)

- Transformer temperature (°C)

## Known Issues or Limitations

- Some data points may be missing or marked as 'NaN' due to sensor errors or maintenance periods.

#### **Data Dictionaries**

A data dictionary should be provided for each dataset, defining all variables, their types (e.g., numeric, string), and allowable ranges or categories. This helps users understand the dataset and ensures correct interpretation of the data.

The dictionary data can be a text file, which includes all information about the variable, data in the dataset. For example, the dataset contains SCADA data collected from the Kelmarsh 1 turbine for the period from January 1, 2022, to January 1, 2023. The data is used for learning and analysis of wind energy performance and turbine operations.

Variables:

Date and Time

Type: DateTime

Description: The timestamp for each recorded data entry.

Format: YYYY-MM-DD HH:MM

Wind Speed

Type: Numeric

Unit: m/s

Description: The average wind speed recorded at the turbine.

Allowable Range: 0 - 50 m/s

Wind Speed, Standard Deviation

Type: Numeric

Unit: m/s

Description: The standard deviation of the wind speed measurements.

Allowable Range: 0 - 15 m/s

Wind Speed, Minimum

Type: Numeric

Unit: m/s

Description: The minimum wind speed recorded during the interval.

Allowable Range: 0 - 50 m/s

Wind Speed, Maximum

Type: Numeric

Unit: m/s

Description: The maximum wind speed recorded during the interval.

Allowable Range: 0 - 50 m/s

Wind Direction

Type: Numeric

Unit: Degrees

Description: The average wind direction recorded at the turbine.

Allowable Range: 0 - 360°

Nacelle Position

Type: Numeric

Unit: Degrees

Description: The position of the nacelle relative to a reference direction.

Allowable Range: 0 - 360°

Energy Export

Type: Numeric

Unit: kWh

Description: The total energy exported by the turbine to the grid.

Allowable Range: 0 - 10,000 kWh

Energy Import

Type: Numeric

Unit: kWh

Description: The total energy imported from the grid for turbine operations.

Allowable Range: 0 - 500 kWh

Lost Production (Time-based IEC B.2.2)

Type: Numeric

Unit: kWh

Description: Energy lost due to time-based curtailments classified under IEC B.2.2.

Allowable Range: 0 - 10,000 kWh

Front Bearing Temperature

Type: Numeric

Unit: °C

Description: The temperature of the front bearing.

Allowable Range: -40 - 100°C

Rear Bearing Temperature

Type: Numeric

Unit: °C

Description: The temperature of the rear bearing.

Allowable Range: -40 - 100°C

Nacelle Ambient Temperature

Type: Numeric

Unit: °C

Description: The ambient temperature inside the nacelle.

Allowable Range: -40 - 50°C

Grid Voltage

Type: Numeric

Unit: V

Description: The voltage of the grid connection.

Allowable Range: 0 - 400 V

Power Factor (cosphi)

Type: Numeric

Description: The power factor of the turbine's electrical output.

Allowable Range: -1 to 1

This data dictionary provides a clear and structured definition for each variable, making it easier for users to understand the dataset and use it appropriately. It ensures that all variables are well-defined, and any potential ambiguities are addressed, promoting accurate analysis and interpretation.

### **Data Storage and Preservation**

#### **Storage Solutions**

Data should be stored in secure and reliable storage solutions, such as institutional repositories or cloud storage services, ensuring data integrity and accessibility. These solutions should support regular backups and provide access controls to safeguard data.

#### **Backup Procedures**

Implement regular backup procedures to prevent data loss. Backups should be performed daily, and copies should be stored in different locations to mitigate the risk of physical damage or technical failures.

#### **Data Retention Policy**

Define a data retention policy that specifies how long data will be retained and the criteria for archiving or deleting data. Typically, raw data should be kept for at least five years, while processed data should be retained for as long as it remains relevant to ongoing research.

### **Data Quality Assurance**

#### **Data Validation Procedures**

Data validation procedures should be established to ensure accuracy and completeness. This includes automated checks for data consistency, range checks, and manual reviews to identify and correct errors.

Example Data Validation Procedures

Dataset: Turbine\_Data\_Kelmarsh\_1\_2022-01-01\_-\_2023-01-01\_228.csv

Automated Checks for Data Consistency:

Consistency Checks: Implement scripts to check for consistency across related data fields. For instance, if Wind Speed is recorded, ensure corresponding Wind Speed, Minimum and Wind Speed, Maximum values are logical (i.e., Minimum <= Wind Speed <= Maximum).

Timestamp Verification: Verify that the Date and Time stamps are in a continuous sequence with no missing intervals, ensuring data is recorded at the expected intervals (e.g., every 10 minutes).

Range Checks:

Wind Speed: Ensure that all values for Wind Speed fall within the allowable range of 0 - 50 m/s. Any values outside this range should be flagged for review.

Temperature Variables: Check that temperature readings (e.g., Front Bearing Temperature, Nacelle Ambient Temperature) fall within expected operational limits (e.g., -40 to 100°C for bearing temperatures).

Power and Energy Metrics: Validate that energy metrics such as Energy Export and Energy Import are non-negative and within realistic operational ranges.

Manual Reviews:

Outlier Detection: Conduct manual reviews to identify and investigate outliers. For example, if a Wind Speed value is significantly higher or lower than typical values, manually check sensor logs and maintenance records for possible explanations.

Data Completeness: Review for completeness by ensuring that no critical data fields are left empty or marked as NaN without justification. This includes checking for missing values in essential metrics such as Wind Speed, Wind Direction, and Energy Export.

Cross-Verification: Manually cross-verify a random sample of the dataset against raw data logs to ensure accuracy in data recording and preprocessing steps.

Error Correction:

Flagging and Correction: Automatically flag data points that fail consistency or range checks. Investigate these points to determine if they are genuine anomalies or errors. Correct data entry errors where possible, or provide annotations if data cannot be corrected.

Documentation: Document any corrections made during the validation process. This includes recording the original value, the corrected value, and the reason for the correction.

Validation Reports:

Automated Reports: Generate automated validation reports summarizing the results of consistency checks, range checks, and any identified anomalies.

Review Logs: Maintain logs of manual reviews and corrections for transparency and future reference. These logs should detail the steps taken during the review process and the outcomes of any investigations into flagged data points.

By implementing these data validation procedures, the dataset's accuracy and completeness can be significantly improved, ensuring reliable and high-quality data for analysis and decision-making.

#### **Error Handling**

Outline procedures for identifying, documenting, and correcting errors in the data. This includes maintaining an error log and updating the data files as necessary, ensuring that all changes are documented and traceable.

Example Error Handling Procedures

Dataset: Turbine\_Data\_Kelmarsh\_1\_2022-01-01\_-\_2023-01-01\_228.csv

Identifying Errors:

Automated Error Detection: Implement automated scripts to scan the dataset for anomalies, such as missing values, outliers, or values that fall outside the expected ranges. For instance, if Wind Speed is outside the range of 0-50 m/s, the value should be flagged.

Manual Review: Perform periodic manual reviews of the dataset to catch errors that automated scripts might miss, such as inconsistencies in time stamps or unexpected patterns in the data.

Documenting Errors:

Error Log: Maintain a comprehensive error log that records details of each identified error. The log should include:

Date and Time: When the error was detected.

Variable Affected: The specific variable(s) involved.

Error Description: A brief description of the nature of the error.

Detection Method: Whether the error was detected by automated scripts or manual review.

Status: The current status of the error (e.g., identified, under review, corrected).

Correcting Errors:

Investigation: Upon identifying an error, investigate the cause. This might involve checking the original raw data, reviewing sensor logs, or consulting with team members who were involved in data collection.

Correction Process: Correct the error in the dataset. If the error can be fixed (e.g., a typographical error in data entry), update the value. If the correct value cannot be determined, mark the value as missing (e.g., NaN).

Update Documentation: Update the error log to reflect the correction. Record the original value, the corrected value, and the reason for the correction.

Traceability:

Version Control: Use version control for the dataset to keep track of changes. Each update or correction should be documented with a new version number and a summary of changes made.

Audit Trail: Maintain an audit trail that includes all actions taken to correct errors. This ensures that every change is traceable back to the original error detection.

Regular Review:

Periodic Audits: Conduct regular audits of the error log and corrected data to ensure that error handling procedures are being followed consistently.

Feedback Loop: Implement a feedback loop where recurring errors are analyzed to improve data collection and processing methods, reducing the likelihood of similar errors in the future.

By following these procedures, the dataset's accuracy and reliability can be maintained, ensuring that all changes are documented and traceable. This systematic approach to error handling helps build trust in the data and supports robust data analysis and decision-making.

#### **Version Control**

Implement a version control system to track changes to datasets over time. This practice helps maintain a comprehensive record of modifications and ensures that everyone is using the correct version of the data. Version control systems, such as Git, allow you to document each change, revert to previous versions if necessary, and collaborate efficiently with other team members. By using these tools, you can prevent conflicts and confusion that arise from multiple versions of the same dataset being used simultaneously.

Tools and Methods:

* Git: Utilize Git for version control, which provides robust tracking and collaboration features. Create a repository for your datasets where all changes are committed and logged.
* Other Methods: If Git is not suitable, consider other version control tools or software that offer similar functionalities. Ensure that whichever tool you choose supports detailed change logs and easy access to previous versions.

By implementing a version control system, you enhance the transparency and reliability of your data management processes, ensuring that all team members are working with the most up-to-date and accurate datasets.

## Data Sharing Protocol

### **Data Sharing and Access**

#### **Data Sharing Platforms**

Identify preferred platforms for data sharing to ensure that data is securely stored, easily accessible, and meets the standards for interoperability. Various types of platforms can be used to share data, each with its own set of features and benefits.

##### Institutional Repositories:

Many universities have their own repositories where researchers can deposit their datasets. These platforms typically offer secure storage, controlled access, and compliance with institutional policies.

Examples: Harvard Dataverse, Purdue University Research Repository (PURR).

##### Public Databases:

General Repositories: These repositories accept data from a wide range of disciplines and offer broad visibility and accessibility.

* Figshare: Supports a variety of data types and provides DOIs for datasets, enhancing citation and sharing.
* Zenodo: Developed by CERN, it supports open data and provides DOIs, ensuring data preservation and discoverability.
* Kaggle:

##### Project Websites:

It’s al beneficial to use project-specific websites or platforms designed to host datasets related to a specific research project. These platforms can be tailored to meet the specific needs of the project and its stakeholders.

Examples: Project websites with dedicated data sections, bespoke data portals developed for large-scale projects.

##### Commercial and Cloud-Based Solutions:

* Amazon Web Services (AWS): Offers robust data storage solutions with tools for sharing and collaboration.
* Google Cloud Storage: Provides secure storage with easy sharing options and integration with other Google services.
* Microsoft Azure: Offers data storage with extensive compliance and security features.

##### Git Repositories:

* GitHub: While primarily used for source code, GitHub can also store datasets, especially those that require version control. GitHub provides collaborative features and version tracking, making it suitable for dynamic datasets.
* GitLab: Similar to GitHub, it offers robust version control and collaboration features, suitable for both data and code sharing.

##### Other Platforms:

* OSF (Open Science Framework): A free, open platform to support research and enable collaboration. It allows users to manage, share, and archive data, facilitating open and reproducible research.
* Mendeley Data: A secure cloud-based repository that allows researchers to upload and share data.

When selecting a data sharing platform, ensure that it meets relevant standards for data security, accessibility, and interoperability. The platform should support the necessary metadata standards, offer robust access controls, and comply with legal and ethical requirements for data sharing. Additionally, providing DOIs or other persistent identifiers for datasets ensures they can be reliably cited and accessed over time.

#### **Access Levels**

Define different levels of data access based on the sensitivity of the data. For example, some data may be openly accessible to the public, while other datasets may be restricted to certain users or require a data use agreement.

#### **Licensing**

Specify the licenses under which data can be shared to protect intellectual property rights while promoting reuse. Licensing is a crucial aspect of data sharing that ensures users understand the terms under which data can be used, shared, and adapted. Adhering to the principles of FAIR (Findable, Accessible, Interoperable, and Reusable) data, choosing the right license ensures that data is openly available while respecting the rights of the data creators. FAIR Data Principles:

* Findable: Data should be easy to find for both humans and computers. This includes assigning a globally unique and persistent identifier to the data.
* Accessible: Once found, data should be retrievable through well-defined protocols. This includes metadata that explains how to access the data.
* Interoperable: Data should be compatible with other datasets and tools. This involves using standardized data formats and vocabularies.
* Reusable: Data should be well-described so that it can be replicated and/or combined in different settings. Licensing plays a key role here to specify how the data can be reused.

Common licenses include Creative Commons licenses, which allow users to share and adapt the data with proper attribution.

* CC BY (Attribution): Allows others to distribute, remix, adapt, and build upon the data, even commercially, as long as they credit the original creation.
* CC BY-SA (Attribution-ShareAlike): Similar to CC BY, but new creations must be licensed under the identical terms.
* CC BY-ND (Attribution-NoDerivs): Allows for redistribution, commercial and non-commercial, as long as it is passed along unchanged and in whole, with credit to the creator.
* CC BY-NC (Attribution-NonCommercial): Allows others to remix, adapt, and build upon the data non-commercially, and although their new works must also acknowledge the original creator, they don’t have to license their derivative works on the same terms.
* CC BY-NC-SA (Attribution-NonCommercial-ShareAlike): Allows others to remix, adapt, and build upon the data non-commercially, as long as they credit the original creation and license their new creations under the identical terms.
* CC BY-NC-ND (Attribution-NonCommercial-NoDerivs): The most restrictive license, allowing others to download the data and share it with others as long as they credit the original creator, but they can’t change it in any way or use it commercially.

##### Apache License 2.0:

The Apache License 2.0 is a permissive license that allows users to freely use, modify, and distribute the licensed work, both in original and modified form. Key features include:

* Freedom to Use: Users can use the software for any purpose, including commercial applications.
* Freedom to Modify: Users can modify the software to fit their needs.
* Freedom to Distribute: Users can distribute the original or modified software under the same license.
* Patent Rights: The license provides an express grant of patent rights from contributors to users.
* Conditions: Users must include a copy of the license in any redistribution of the software and provide a notice of any changes made to the original.

##### MIT License:

The MIT License is another permissive license, similar to the Apache License, allowing users extensive freedom with minimal restrictions:

* Freedom to Use: Users can use the software for any purpose.
* Freedom to Modify and Distribute: Users can modify the software and distribute it, whether in original or modified form.
* Conditions: Users must include the original copyright notice and a copy of the license in any distribution.

##### GPL (General Public License):

The GPL is a copyleft license that requires any modified versions of the software to also be open source and distributed under the same license:

* Freedom to Use: Users can use the software for any purpose.
* Freedom to Modify: Users can modify the software.
* Freedom to Distribute: Users can distribute original or modified versions, but must keep the same license.
* Copyleft: Any derivative work must be distributed under the same GPL license, ensuring that modifications remain open source.

By clearly specifying the licenses under which data can be shared, you ensure that data is used ethically and legally, promoting reuse and collaboration while protecting intellectual property rights. This approach aligns with the FAIR principles, enhancing the utility and impact of your data.

### **Ethical and Legal Considerations**

#### **Informed Consent**

Ensure that all data collection involving human subjects has obtained appropriate informed consent for data sharing. Consent forms should clearly state how the data will be used, stored, and shared.

#### **Anonymization and Pseudonymization**

Describe methods for anonymizing or pseudonymizing data to protect privacy. This includes removing or masking identifiers that could be used to trace data back to individuals.

#### Compliance with Regulations

Ensure adherence to relevant data protection regulations, such as the General Data Protection Regulation (GDPR), and institutional policies. This includes conducting data protection impact assessments where necessary.

### **Roles and Responsibilities**

#### Data Steward

Designate a data steward responsible for overseeing data management and sharing practices. The data steward ensures compliance with the protocol, coordinates data submissions, and manages access requests.

#### Contributor Responsibilities

Define the responsibilities of researchers contributing data, including providing complete metadata, adhering to data formatting standards, and submitting data in a timely manner.

#### **User Responsibilities**

Outline the responsibilities of users accessing shared data, such as adhering to data use agreements, citing the data correctly, and reporting any errors or issues encountered.

### **Data Sharing Workflow**

#### **Submission Process**

Describe the process for submitting data to the shared repository, including the required documentation, quality checks, and metadata. Submissions should be reviewed for completeness and accuracy before being accepted.

#### **Review and Approval**

Implement a review process to ensure data meets quality standards before sharing. This includes validating metadata, checking for data integrity, and confirming adherence to the data sharing protocol.

#### **Distribution**

Detail how data will be distributed to approved users, including any necessary agreements or access controls. Ensure that users are provided with all relevant documentation and metadata.

### **Support and Training**

#### **Training Programs**

Provide training for researchers on data management best practices, tools, and protocols. Training should cover data collection, documentation, formatting, and sharing procedures.

#### **Support Resources**

Offer support resources, such as templates, guides, and help desks, to assist with data management and sharing. These resources should be easily accessible to all members of the research group.

### **Monitoring and Evaluation**

#### **Performance Metrics**

Establish metrics to evaluate the effectiveness of the data sharing protocol, such as data usage statistics, user feedback, and compliance rates. Regularly review these metrics to identify areas for improvement.

#### **Continuous Improvement**

Regularly review and update the protocol based on feedback and evolving best practices. Ensure that changes are communicated to all members of the research group and that training materials are updated accordingly.

## Git for controlling version

A version control system is a software tool designed to manage and record changes to files over time. It enables users to create multiple versions of their files, capturing the changes in records known as commits. Each commit contains valuable metadata, including details about what changes were made and who made them. The entire sequence of commits and their associated metadata form a repository. Repositories can be synchronized across different computers, making it easier for multiple individuals to collaborate on a project. This system ensures that every participant has access to the latest versions and can contribute efficiently to the shared work.

Git is a modern version control system designed to facilitate distributed version control and collaboration. Git allows every user to have a full-fledged repository on their local machine, complete with the entire project history. This distributed nature ensures that the project is not dependent on a single server, increasing reliability and availability.

One of Git's standout features is its powerful merging tools, which enable multiple developers to work on the same files concurrently. When different versions of a file are created, Git can merge changes seamlessly, using advanced algorithms to integrate modifications made by various contributors. This minimizes conflicts and ensures that all changes are incorporated smoothly.

Additionally, Git supports a variety of workflows and branching models, making it flexible enough to handle projects of any size and complexity. Branching allows developers to work on new features, bug fixes, or experiments in isolation before merging them back into the main project, maintaining a clean and stable codebase.

Git's ability to track changes, revert to previous states, and maintain a detailed history of the project, including who made each change and why, makes it an invaluable tool for both individual developers and large teams. Its robustness and efficiency have made it the version control system of choice for reserachers, engineers, and open-source projects.

### Git settings

When setting up Git on a new computer, there are a few initial configurations that need to be done. These settings help to personalize your Git environment and ensure that your identity is associated with your work. Here are the essential configurations you will need to set up:

* User Name and Email Address: These identify your commits and changes.
* Preferred Text Editor: This sets the editor you prefer for writing commit messages.
* Global Settings: Applying these settings globally means they will be used for all your Git projects on that computer.

To configure Git, you use the command line with the syntax git verb options, where verb represents the action you want to perform and options provide additional details for that action. For example, to set up his new laptop, TCD might use the following commands:

$ git config --global user.name "TCD "

$ git config --global user.email "TCD@tcd.ie"

Make sure to replace "Vlad Dracula" and "vlad@tran.sylvan.ia" with your actual name and email address. This information will be attached to your Git activity, ensuring that all your commits and changes can be traced back to you. If you are using GitHub, it is important to use the same email address associated with your GitHub account to maintain consistency.

### Git help manual

If you ever find yourself unsure about the subcommands or options for a Git command, you can easily access help directly from the command line. By typing git <command> -h, you can view a list of options related to that command. Alternatively, you can view the detailed Git manual by typing git <command> --help. For example:

$ git config -h

$ git config --help

While viewing the manual, remember that the : character indicates a prompt waiting for your input, and you can press Qto exit the manual and return to the command line.

For a more general overview of all available Git commands and additional resources, you can use:

$ git help

This command provides a comprehensive list of Git commands and directs you to further resources within the Git manual. These built-in help tools are invaluable for quickly finding the information you need to effectively use Git.

### Creating a Git Repository

Once Git is configured, we can start using it. First, let’s create a new directory in the Desktop folder for our work and then change the current working directory to the newly created one: To start working with Git, you first need to set up a new directory and initialize it as a Git repository. Here’s how you can do it:

$ cd ~/Desktop

$ mkdir RRTutorialsGit

$ cd RRTutorialsGit

This creates a new directory called RRTutorialsGit on your desktop and navigates into it. Next, you need to initialize this directory as a Git repository:

$ git init

This command sets up a new Git repository in the RRTutorialsGit directory, which will store versions of your files. It's important to note that git init will create a repository that can include subdirectories and their files. There is no need to create separate repositories within the RRTutorialsGit repository, whether subdirectories are present initially or added later. The creation of the RRTutorialsGit directory and its initialization as a Git repository are independent steps.

If you list the contents of the directory using ls, it might appear that nothing has changed:

$ ls

However, if you use the -a flag to show hidden files, you will see that Git has created a hidden .git directory:

$ ls -a

Output:

. .. .git

The .git subdirectory is used by Git to store all the information about your project, including the tracked files and subdirectories. If you ever delete the .git directory, you will lose the project’s version history.

Next, you can change the default branch to main. Depending on your Git settings and version, main might already be the default branch. To switch to a new branch called main, use:

$ git checkout -b main

Output:

Switched to a new branch 'main'

To verify that everything is set up correctly, you can check the status of your project with:

$ git status

Output:

On branch main

No commits yet

nothing to commit

Note that the exact wording of the output may vary slightly depending on your version of Git. This command gives you a summary of the current state of the repository, indicating which branch you are on and whether there are any changes staged for commit.

### Remotes in GitHub

To create a new remote repository on GitHub, follow these steps:

Log in to GitHub: Access your GitHub account and click on the "+" icon in the top right corner of the page. Select "New repository" from the dropdown menu.

* Name Your Repository: In the repository creation form, enter " RRTutorialsGit" as the repository name. Ensure that "Initialize this repository with a README" is unchecked and leave "Add .gitignore" and "Add a license" set to "None." This is important because the repository must be empty to connect it with your local repository.
* Create Repository: Click the "Create Repository" button to finalize the setup.
* The Repository Needs to Be Empty: The repository needs to be empty because it will be linked to an existing local repository. Initializing it with files like a README or license could cause conflicts during the linking process.
* Configure Local Repository: Once the repository is created, GitHub will display a page with the repository URL and instructions on how to set up your local repository to connect with this new remote repository.
* Example URL and Instructions: The page will provide a URL similar to https://github.com/yourusername/RRTutorialsGit.git and commands to add this remote repository to your local Git setup.

Here’s an example of how to connect your local repository to this new remote repository:

$ git remote add origin https://github.com/yourusername/RRTutorialsGit.git

$ git branch -M main

$ git push -u origin main

These commands add the remote repository to your local Git setup, rename the default branch to main (if it isn't already), and push your local commits to the remote repository, establishing a connection between them.

### Connect Local to Remote Repository

To connect your local Git repository to the remote repository on GitHub, follow these steps:

* Get the Repository URL: Navigate to the home page of your newly created repository on GitHub. You will find a URL that you need to copy. This URL identifies your remote repository.

Choose SSH: Click on the ‘SSH’ link to change the protocol from HTTPS to SSH. SSH (Secure Shell) is a protocol that, although requiring additional configuration, provides a secure method for transferring data. It is widely used in many applications for its security benefits.

* Copy the SSH URL: After switching to SSH, copy the provided SSH URL. It will look something like git@github.com:yourusername/RRTutorialsGit.git.
* Add Remote Repository: Go to your local RRTutorialsGit repository and run the following command to add the remote repository:

$ git remote add origin git@github.com:yourusername/RRTutorialsGit.git

Make sure to replace yourusername with your actual GitHub username. The origin is a conventional name used to refer to the remote repository. Although you can name it anything, sticking with origin is helpful as it is commonly used by default in Git and GitHub.

* Verify the Remote Connection: To ensure that the remote repository has been added correctly, run:

$ git remote -v

* You should see an output similar to this:

origin git@github.com:yourusername/RRTutorialsGit.git (fetch)

origin git@github.com:yourusername/RRTutorialsGit.git (push)

This confirms that your local repository is now connected to the remote repository on GitHub, enabling you to push and pull changes between the two. In subsequent lessons, you will learn more about using remotes for collaboration and managing shared projects.

### SSH background and setups

Before connecting to a remote repository, it's essential to set up a way for your computer to authenticate with GitHub. This setup uses Secure Shell Protocol (SSH), a cryptographic network protocol that ensures secure communication between computers over an otherwise insecure network.

#### **Understanding SSH and Key Pairs:**

SSH uses a pair of keys for authentication: a public key and a private key. The public key can be shared openly, while the private key is kept secure and private. Think of the public key as a padlock that can be locked by anyone, but only you have the private key to unlock it. By adding your public key to your GitHub account, you instruct GitHub to lock communications using this key, ensuring that only your computer, which holds the matching private key, can access your repositories.

#### **Setting Up SSH Keys:**

#### **Check for Existing SSH Keys:**

First, verify if SSH keys already exist on your computer. SSH keys are generally set up once and then used repeatedly.

ls -al ~/.ssh

If SSH has not been set up, you might see an error like:

ls: cannot access '/c/Users/YourUsername/.ssh': No such file or directory

#### **Generate a New SSH Key Pair:**

If SSH keys do not exist, create a new key pair. Use the following command to generate the keys, where the -t option specifies the type of algorithm and -C attaches a comment to the key (such as your email address):

$ ssh-keygen -t ed25519 -C "your\_email@example.com"

For legacy systems that do not support the Ed25519 algorithm, use:

$ ssh-keygen -t rsa -b 4096 -C "your\_email@example.com"

Follow the prompts to save the key in the default location and optionally add a passphrase for added security.

#### **Save the SSH Keys:**

After generating the keys, you should see output similar to this:

Your identification has been saved in /c/Users/YourUsername/.ssh/id\_ed25519

Your public key has been saved in /c/Users/YourUsername/.ssh/id\_ed25519.pub

#### **Copy the Public Key to GitHub:**

Display the public key with the following command:

cat ~/.ssh/id\_ed25519.pub

Copy the output, which looks like:

ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDmRA3d51X0uu9wXek559gfn6UFNF69yZjChyBIU2qKI your\_email@example.com

Go to GitHub, click on your profile icon, and navigate to **Settings** > **SSH and GPG keys**. Click **New SSH key**, add a title (e.g., "My Laptop"), paste your public key into the key field, and click **Add SSH key**.

#### **Test the SSH Connection:**

Verify that your SSH setup works by running:

$ ssh -T git@github.com

If successful, you should see a message like:

Hi YourUsername! You've successfully authenticated, but GitHub does not provide shell access.

This confirms that your SSH keys are correctly configured, allowing you to securely push and pull from your GitHub repositories. With SSH setup complete, you're ready to work with your remote repositories securely.

### Push Local Changes to a Remote Repository

With SSH authentication set up, you can now push your local repository changes to the remote repository on GitHub. Use the following command to push your changes:

$ git push origin main

If you have set up a passphrase for your SSH key, you will be prompted to enter it when you execute this command. If you configured advanced authentication settings, you might not be prompted for the passphrase.

You should see output similar to this:

(base) YouComputerAddress:RRTutorials mac$ git push origin main

Enumerating objects: 6, done.

Counting objects: 100% (6/6), done.

Delta compression using up to 4 threads

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 48.06 KiB | 24.03 MiB/s, done.

Total 4 (delta 1), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (1/1), completed with 1 local object.

To git@github.com:yourusername/RRTutorials.git

ed39e78..6f30070 main -> main

This output confirms that the changes from your local main branch have been successfully pushed to the main branch in your remote GitHub repository.

If you are connected to a network that uses a proxy, you might encounter an error message like “Could not resolve hostname.” To resolve this issue, configure Git to use the proxy:

$ git config --global http.proxy http://user:password@proxy.url

$ git config --global https.proxy https://user:password@proxy.url

Replace user, password, and proxy.url with your actual proxy credentials and URL. When you switch to a network that does not use a proxy, disable the proxy settings with:

$ git config --global --unset http.proxy

$ git config --global --unset https.proxy

By correctly configuring the proxy settings, you can ensure smooth connectivity regardless of the network environment, allowing you to push changes to the remote repository without issues.

After pushing your local changes to the remote repository, your local and remote repositories are now synchronized. The remote repository is an exact copy of your local repository.

### Pulling Changes from the Remote Repository:

You can also pull changes from the remote repository to your local repository using the git pull command:

$ git pull origin main

Output:

From https://github.com/yourusername/RRTutorials

\* branch main -> FETCH\_HEAD

Already up-to-date.

In this case, pulling has no effect because the local and remote repositories are already synchronized. However, if someone else had pushed changes to the remote repository on GitHub, this command would download those changes to your local repository, ensuring both repositories are up-to-date.

This process maintains synchronization between your local and remote repositories, enabling seamless collaboration and version control.

### Exploring History

In Git, commits can be referred to by their unique identifiers, which are SHA-1 hashes. For convenience, you can refer to the most recent commit of the working directory using the identifier HEAD. This points to the tip of the current branch and is a useful reference for various Git operations.

To illustrate, let's make a change to a file named data.txt in ./data/Kelmarsh\_SCADA\_2022\_clean by adding another line. Since we've been adding lines one by one, it’s straightforward to track our progress by examining the changes referenced by HEAD.

**Example Workflow:**

* **Modify**data.txt**:** Open data.txt and add a new line of text. For example, add "End of notes."
* **Check the Current Status:** Use the git status command to see the current state of your working directory.

$ git status

On branch main

Your branch is up to date with 'origin/main'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: ../../.DS\_Store

modified: ../../DataFormatforWindEnergyApplications.docx

modified: data.txt

no changes added to commit (use "git add" and/or "git commit -a")

* Add the Change: Stage the change using git add .

$ git add .

* Commit the Change: Commit the change with a descriptive message.

$ git commit -m " Add note at the end of file."

[main e56eec2] Add note at the end of file.

1 file changed, 2 insertions(+)

* Referencing HEAD: You can view the most recent commit using HEAD. For instance, to see the details of the latest commit, you can use:

$ git show HEAD

commit e56eec2ae0f948f8f562d9adc592f6b227c81257 (HEAD -> main)

Author: yourusername <youremail>

Date: Wed Jun 26 15:45:08 2024 +0100

Add note at the end of file.

diff --git a/data/Kelmarsh\_SCADA\_2022\_clean/data.txt b/data/Kelmarsh\_SCADA\_2022\_clean/data.txt

index a131784..58eebfd 100644

--- a/data/Kelmarsh\_SCADA\_2022\_clean/data.txt

+++ b/data/Kelmarsh\_SCADA\_2022\_clean/data.txt

@@ -314,3 +314,5 @@ Drive train acceleration, Min (mm/ss),

Drive train acceleration, StdDev (mm/ss),

Tower Acceleration X, StdDev (mm/ss),

Tower Acceleration Y, StdDev (mm/ss)

+

+End of notes

\ No newline at end of file

This will display the changes introduced in the latest commit along with the commit message and other metadata.

By using HEAD, you can easily navigate and inspect the history of your project, ensuring you keep track of all modifications and their impacts on your codebase. This approach helps maintain a clear record of changes and facilitates efficient project management.