

WIND TWIN: A DIGITAL TWIN FOR WIND TURBINE

*A Main project report submitted in the partial fulfillment of
the requirement for the award of the degree for VIII semester.*

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

by

MEENUGU SIVAJI (20135A0514)

MD FIROZ KHAN (20135A0516)

B.V.S.S.L GAYATHRI (19131A05P1)

MATCHA SAIRAM (20135A0513)

Under the esteemed guidance of

Dr.D.N.D. Harini

(Associate Professor and Head of the Department)

Department of Computer Science and Engineering



**GAYATRI VIDYA PARISHAD COLLEGE OF
ENGINEERING (AUTONOMOUS)**

(Affiliated to JNTU, Kakinada, AP)

VISAKHAPATNAM-530048

2022-2023

GAYATRI VIDYA PARISHAD COLLEGE OF ENGINEERING
(AUTONOMOUS)
VISAKHAPATNAM-530048



CERTIFICATE

This is to certify that the main project report entitled **“WIND TWIN: A DIGITAL TWIN FOR WIND TURBINE”** being submitted by

MEENUGU SIVAJI	(20135A0514)
MD FIROZ KHAN	(20135A0516)
B.V.S.S.L GAYATHRI	(19131A05P1)
MATCHA SAIRAM	(20135A0513)

in their VIII semester in partial fulfillment of the requirements for the Award of Degree

Bachelor of Technology
in
Computer Science and Engineering

During the academic year 2022-2023

Guide

Dr.D.N.D.HARINI
Associate Professor and
Head of the Department
Dept. of CSE, GVPCOE(A)

Head of Department

Dr.D.N.D.HARINI
Associate Professor and
Head of the Department
Dept. of CSE, GVPCOE(A)

Project Viva-voce held on _____

External Examiner

DECLARATION

We hereby declare that this main project entitled “**WIND TWIN: A DIGITAL TWIN FOR WIND TURBINE**” is a bonafide work done by us and submitted to the Department of **Computer Science and Engineering, Gayatri Vidya Parishad college of engineering (autonomous) Visakhapatnam**, in partial fulfillment for the award of the degree of B. Tech is of own and it is not submitted to any other university or has been published any time before.

PLACE:VISAKHAPATNAM

MEENUGU SIVAJI (20135A0514)

MD FIROZ KHAN, (20135A0516)

B.V.S.S.L GAYATHRI (19131A05P1)

MATCHA SAI RAM (20135A0513)

ACKNOWLEDGEMENT

We thank **Dr. A.B.KOTESWARA RAO** principal, **Gayatri Vidya Parishad College of Engineering (Autonomous)** for extending his utmost support and cooperation in providing all the provisions for the successful completion of the project.

We consider it our privilege to express our deepest gratitude to **Dr. D.N.D. HARINI, Associate professor and Head of the Department of Computer Science and Engineering**, for her valuable suggestions and constant motivation that greatly helped the project work to get successfully completed.

We are extremely thankful **Dr. D.N.D. HARINI, Associate professor and Head of the Department of Computer Science and Engineering** for giving us an opportunity to do this project and providing us support and guidance which helped us to complete the project on time.

We also thank our coordinator, **Dr. CH. SITA KUMARI, Associate Professor, Department of Computer Science and Engineering**, for the kind suggestions and guidance for the successful completion of our project work.

We also thank all the members of the staff in Computer Science and Engineering for their sustained help in our pursuits. We thank all those who contributed directly or indirectly in successfully carrying out his work.

MEENUGU SIVAJI (20135A0514)

MD FIROZ KHAN (20135A0516)

B.V.S.S.L GAYATHRI (19131A05P1)

MATCHA SAI RAM (20135A0513)

ABSTRACT

A digital twin is a virtual replica of a physical asset or system that can be used to simulate, analyze, and optimize its performance. In the case of wind turbines, a digital twin can be used to monitor and control the turbine's operation, as well as to identify and diagnose issues that may arise.

A digital twin for a wind turbine typically includes a detailed model of the physical turbine, along with data from sensors and other sources that provide real-time information on its operation. This data can be used to simulate different operating scenarios and to predict the turbine's behavior under various conditions.

One of the key benefits of a digital twin for a wind turbine is that it can be used to optimize the turbine's performance over its entire lifetime. By using data from the digital twin to identify areas of improvement, operators can adjust the turbine's operation to maximize its energy output and minimize maintenance costs. In addition to optimizing performance, a digital twin for a wind turbine can also be used to improve safety and reduce downtime. By monitoring the turbine's operation in real time, operators can identify potential issues before they become major problems and take corrective action to prevent downtime and minimize the risk of accidents. Overall, a digital twin for a wind turbine is a powerful tool for optimizing performance, improving safety, and reducing maintenance costs, making it an essential component of modern wind power systems.

KeyWords:

- Wind turbine, Virtual model, Performance optimizing, Predicting turbines behaviour.

TABLE OF CONTENTS

CERTIFICATE	ii
DECLARATION	iii
ACKNOWLEDGEMENTS	v
ABSTRACT	vi
CONTENTS	vii
CHAPTER 1. INTRODUCTION	1
CHAPTER 2. SOFTWARE REQUIREMENT ANALYSIS	2
2.1 ERLANG	2
2.1.1 INTRODUCTION	2
2.1.2 OPERATIONS USING ERLANG	3
2.2 ELIXIR	4
2.2.1 INTRODUCTION	4
2.2.2 OPERATIONS USING ELIXIR	4
2.3 PHOENIX	4
2.3.1 INTRODUCTION	4
2.3.2 OPERATIONS USING PHOENIX	5
2.4 MODEL-VIEWER	5
2.4.1 INTRODUCTION	5
2.5 PYTHON	6
2.5.1 INTRODUCTION	6
2.6 C++	7
2.6.1 INTRODUCTION	7
2.7 SENSORS AND BOARDS	7
CHAPTER 3. SOFTWARE SYSTEM DESIGN	17
3.1 PROCESS FLOW DIAGRAM:	18
CHAPTER 4. SRS DOCUMENT	18
4.1 FUNCTIONAL REQUIREMENTS	19
4.2 NON FUNCTIONAL REQUIREMENTS	20
4.3 MINIMUM HARDWARE REQUIREMENTS	21

4.4 MINIMUM SOFTWARE REQUIREMENTS	21
CHAPTER 5. OUTPUT	21
5.1 SYSTEM IMPLEMENTATION	21
5.1.1 INTRODUCTION:	21
5.2 PROJECT MODULES:	22
5.3 SOURCE CODE:	30
5.4 OUTPUT SCREENS:	45
CHAPTER 5. TESTING	46
6.1 Testing Strategies	46
6.2 Testing WindTwin on Different Scenarios	48
6.3 WindTwin Hardware Connection Testing	48
6.4 WindTwin Hardware Connection Testing	49
6.5 WindTwin ADXL345 Sensor Connection Testing	50
CHAPTER 7. CONCLUSION	52
CHAPTER 8. REFERENCES	53

1. INTRODUCTION

A digital twin for a wind turbine is a powerful technology that enables the creation of a virtual replica of a physical wind turbine. The digital twin is created by collecting data from a variety of sources, including sensors installed on the wind turbine itself, as well as weather data and other environmental factors. This data is then fed into a computer model, which creates a digital replica of the wind turbine.

Once the digital twin is created, it can be used to monitor and optimize the performance of the physical turbine in real-time. This technology is particularly useful in the wind energy industry, where wind turbines are often located in remote and harsh environments that can be difficult to access. By creating a digital replica of the turbine, operators can remotely monitor and optimize its performance, reducing downtime and increasing overall efficiency.

Digital twins for wind turbines also enable predictive maintenance, allowing operators to identify potential problems before they occur and schedule maintenance and repairs accordingly. This can help to minimize downtime and reduce maintenance costs, while also improving the overall reliability of wind turbines.

Overall, digital twins for wind turbines are a powerful tool for optimizing the performance and reliability of wind farms. By providing a real-time simulation of the physical turbines, digital twins can help operators and engineers to monitor and optimize their performance, reduce downtime, and improve overall efficiency.

- A wind turbine is a device that converts the kinetic energy of wind into electrical energy. The energy in the wind turns two or three propeller-like blades around a rotor. The rotor is

connected to the main shaft, which spins a generator to create electricity.

- Wind turbines can be used as stand-alone applications, or they can be connected to a utility power grid or even combined with a photovoltaic (solar cell) system.
- Wind turbines require maintenance such as regular inspections, repairs, and servicing to ensure the wind turbine works properly.
- Apart from regular maintenance a person visits the turbine manually and observes turbine speed, voltage fluctuations and wind turbine vibrations to ensure proper operation.

2. SOFTWARE REQUIREMENT ANALYSIS

2.1 ERLANG

2.1.1 INTRODUCTION

Erlang is a programming language designed for building scalable and fault-tolerant distributed systems. It was originally developed in the late 1980s by Ericsson, a Swedish telecommunications company, for use in their telephone switching systems. Since then, Erlang has gained popularity in various industries for its ability to handle concurrent processes, manage high-volume network traffic, and maintain system reliability in the face of failures. Erlang is a functional programming language that emphasizes immutability and message-passing between processes instead of shared memory. It has a simple syntax and is particularly well-suited for writing concurrent, parallel, and distributed software. One of the key features of Erlang is its lightweight process model, which enables the language to handle thousands of processes simultaneously while maintaining low latency and high throughput.

2.1.2 OPERATIONS USING ERLANG

Some common use cases for Erlang in operations include

- Building and managing telecommunications infrastructure, such as switches, routers, and gateways.
- Developing messaging and chat applications with real-time communication capabilities.
- Building distributed databases and data storage systems with high availability and fault tolerance.
- Developing high-performance web servers and other network applications that require concurrency and scalability.

Erlang provides several built-in features and libraries that make it well-suited for operations, including

- OTP (Open Telecom Platform) - a collection of libraries, design principles, and best practices for building highly concurrent and fault-tolerant systems.
- Process-based concurrency - Erlang processes are lightweight and isolated, allowing for highly concurrent systems with minimal overhead.
- Fault tolerance - Erlang's "Let it crash" philosophy encourages designing systems to gracefully recover from failures rather than trying to prevent them.
- Distributed architecture - Erlang supports distributed systems out of the box, with built-in support for messaging, networking, and distributed process management.

2.2 ELIXIR

2.2.1 INTRODUCTION

Elixir is a dynamic, functional programming language used to develop applications that are scalable and maintainable. It runs on the Erlang VM, which is known for creating fault-tolerant, distributed, and low-latency systems. Developers can use Elixir tools in several areas, including web development, embedded software, data pipelines, and multimedia, across a wide range of industries. Elixir primarily relies on those features to guarantee that your software is operating inside the expected constraints. Elixir has been designed to be extensible, letting developers naturally extend the language to particular domains. As an extensible language, Elixir allows developers to extend the language to particular domains in order to increase productivity. The Erlang VM gives Elixir developers full access to Erlang's ecosystem, which is used by many companies including Heroku, WhatsApp, Klarna, and many others.

2.2.2 OPERATIONS USING ELIXIR

Using Elixir, a developer can perform the following operations

—

- Mathematical and logical operations.
- Provides a great set of tools to ease development.
- Platform to provide auto complete, debugging tools, code reloading, as well as nicely formatted documentation.

2.3 PHOENIX

2.3.1 INTRODUCTION

Phoenix is a web development framework written in Elixir which implements the server-side Model View Controller (MVC) pattern. Many of its components and concepts will seem familiar to those of us with experience in other web frameworks like Ruby on

Rails or Python's Django. Phoenix provides the best of both worlds - high developer productivity *and* high application performance. It also has some interesting new twists like channels for implementing real time features and pre-compiled templates for blazing speed.

2.3.2 OPERATIONS USING PHOENIX

- Interact with users and push events, across one or dozens of nodes, by using our built-in Pub Sub, and Channels.
- Build scalable Graph QL apps with Absinthe, or use our built-in JSON support for world class APIs.
- Built-in instrumentation and a live dashboard gives you insight into your applications. Monitor performance and diagnose issues right within your app.

Phoenix runs on the Erlang VM with the ability to handle millions of WebSocket connections alongside Elixir's tooling for building robust systems.

2.4 MODEL-VIEWER

2.4.1 INTRODUCTION

Model-Viewer is an open-source JavaScript library that allows developers to display 3D models and render them in real-time on the web. It was developed by Google and provides a simple and efficient way to display 3D content on websites without the need for plugins or special software. With Model-Viewer, users can rotate and zoom in on models, change materials and lighting, and even interact with them using JavaScript events. This library supports a variety of 3D file formats such as OBJ, GLTF, and FBX. It also includes features such as annotations, animations, and hotspots which can be used to enhance the user experience. Model-Viewer is a powerful tool

for developers who want to showcase their 3D models online in an interactive and engaging way.

2.4.2 OPERATIONS USING MODEL-VIEWER

Here are some of the most common operations that can be done using Model-Viewer:

- **Rotation:** Users can rotate the model in real-time using the mouse or touch gestures. This allows them to view the model from different angles and get a better understanding of its shape and structure.
- **Zooming:** Users can zoom in and out on the model to inspect details and see the model up close.
- **Changing materials:** Model-Viewer allows developers to define multiple materials for a model and switch between them in real-time. This allows users to see the model in different colors or textures.
- **Annotations:** Annotations can be added to the model to provide additional information or highlight specific parts of the model. These annotations can be text or images and can be triggered by clicking on the relevant part of the model.
- **Hotspots:** Hotspots can be added to the model to allow users to interact with it. For example, clicking on a hotspot can trigger an animation or display additional information about the model.
- **Animations:** Model-Viewer supports animations, allowing developers to create dynamic 3D content that can be viewed in real-time on the web.

2.5 PYTHON

2.5.1 INTRODUCTION

Python 3.0 (a.k.a. "Python 3000" or "Py3k") is a new version of

the language that is incompatible with the 2.x line of releases. The language is mostly the same, but many details, especially how built-in objects like dictionaries and strings work, have changed considerably, and a lot of deprecated features have finally been removed. Also, the standard library has been reorganized in a few prominent places.

2.6 C++

2.6.1 INTRODUCTION

C++ (pronounced "C plus plus") is a high level general purpose programming language created by Danish computer scientist Bjarne Stroustrup as an extension of the C programming language, or "C with classes ". The language has expanded significantly over time, and modern C++ now has object oriented, generic, and functional features in addition to facilities for low level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free software foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms. C++ was designed with systems programming and embedded, resource-constrained software and large systems in mind, with performance, efficiency, and flexibility of use as its design highlights. C++ has also been found useful in many other contexts, with key strengths being software infrastructure and resource-constrained applications, including desktop applications, video games, servers (e.g. ecommerce, web search, or databases), and performance-critical applications (e.g. telephone switches or space probes).

2.7 SENSORS AND BOARDS

2.7.1 PHOTO INTERRUPTER

Photo Interrupter sensor of working voltage 4.5V to 5.5V, transmit tube pressure drop 1.6V and Transmit tube current 20 mA,

Measurement frequency, 100 kHz. This speed measuring sensor is a wide voltage, high resolution, short response time, and the switch output speed measurement module. It can test the motor's rotational speed with a black encoder. Photo Interrupter sensor helps in determining the speed of a rotating object.

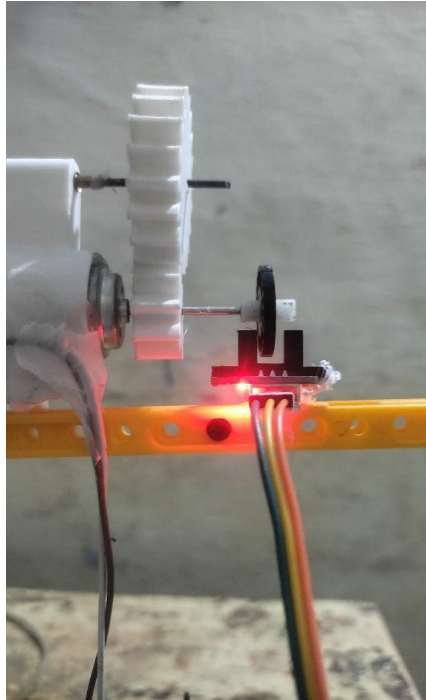


Fig 2.7.1 Photointerrupter sensor

2.7.2 VOLTAGE DETECTION SENSOR

Voltage Detection Sensor Module 25V is a precise low-cost sensor for measuring voltage. It is based on the principle of resistive voltage divider design. It can make the red terminal connector input voltage 5 times smaller. Arduino analog input voltages up to 5V, the voltage detection module input voltage not greater than $5V \times 5 = 25V$ (if using 3.3V systems, input voltage not greater than $3.3V \times 5 = 16.5V$). Arduino AVR chips have 10-bit AD, so this module simulates a resolution of 0.00489V ($5V/1023$), so the minimum voltage of the input

voltage detection module is $0.00489V \times 5 = 0.02445V$. It helps in measuring voltages using Arduino by interfacing a Voltage Sensor with Arduino.

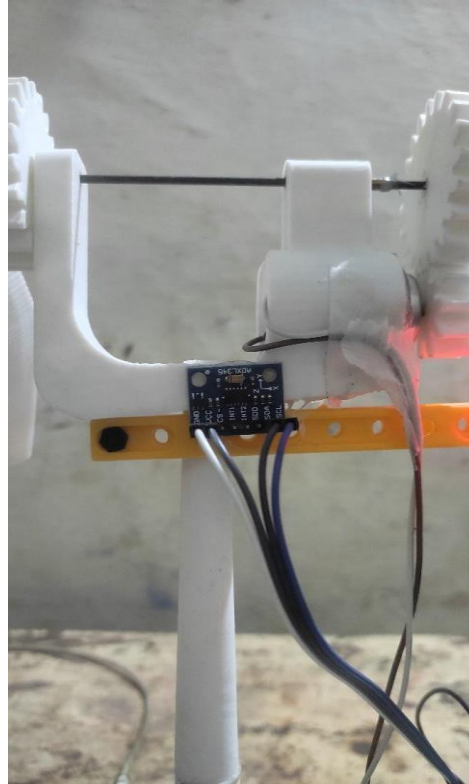


Fig 2.7.2 Voltage detection sensor

2.7.3 ACCELEROMETER GYROSCOPE SENSOR

Accelerometer gyroscope sensor is a type of sensor that combines both an accelerometer and a gyroscope into a single device. These sensors are commonly used in a variety of applications, including consumer electronics, aerospace, automotive, and robotics. An accelerometer is a sensor that measures the acceleration or change in velocity of an object. It is typically used to measure the orientation, position, and motion of an object, and is often found in devices such as smartphones, fitness trackers, and gaming controllers. Accelerometers use micro-electromechanical systems (MEMS) technology to detect changes in acceleration, and typically measure

acceleration along three axes: X, Y, and Z.

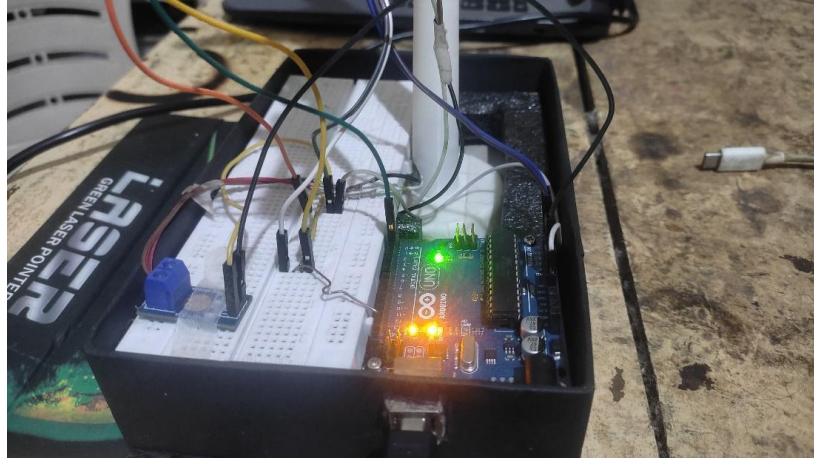


Fig 2.7.3 ACCELEROMETER GYROSCOPE SENSOR

2.7.4 ARDUINO UNO

The Arduino UNO is a standard board of Arduino. Here UNO means 'one' in Italian. It was named as UNO to label the first release of Arduino Software. It was also the first USB board released by Arduino. It is considered as the powerful board used in various projects. Arduino.cc developed the Arduino UNO board. Arduino UNO is based on an ATmega328P microcontroller. It is easy to use compared to other boards, such as the Arduino Mega board, etc. The board consists of digital and analog Input/Output pins (I/O), shields, and other circuits. The Arduino UNO includes 6 analog pin inputs, 14 digital pins, a USB connector, a power jack, and an ICSP (In-Circuit Serial Programming) header. It is programmed based on IDE, which stands for Integrated Development Environment. It can run on both online and offline platform.

2.7.5 RASPBERRY PI

Raspberry Pi, developed by Raspberry Pi Foundation in association with Broadcom, is a series of small single-board computers

and perhaps the most inspiring computer available today. From the moment you see the shiny green circuit board of Raspberry Pi, it invites you to tinker with it, play with it, start programming, and create your own software with it. Earlier, the Raspberry Pi was used to teach basic computer science in schools but later, because of its low cost and open design, the model became far more popular than anticipated. It is widely used to make gaming devices, fitness gadgets, weather stations, and much more. But apart from that, it is used by thousands of people of all ages who want to take their first step in computer science. It is one of the best-selling British computers and most of the boards are made in the Sony factory in Pencoed, Wales.

2.7.6 GENERATOR MOTOR

A generator motor is a type of electric motor that is designed to work in reverse, converting mechanical energy into electrical energy instead of the other way around. It operates by using a magnetic field to generate an electric current, which can then be used to power other electrical devices or systems.

Generator motors are commonly used in a variety of applications, including power generation, wind turbines, and hybrid or electric vehicles. In these applications, the generator motor is typically connected to a rotating shaft or turbine, which spins the motor and generates an electrical current. This current can then be stored in batteries or used to power other electrical devices.

2.8 TURBINE PARTS

2.8.1 TURBINE STAND BASE:

The turbine stand base part is a crucial component of a wind turbine, as it provides a stable foundation for the turbine tower and

blades. The base is typically made of reinforced concrete or steel and is designed to support the weight of the turbine tower, nacelle, and blades, as well as withstand the forces of wind and other environmental factors.

In addition to providing a stable foundation, the turbine stand base part may also include features such as cable trenches, anchor bolts, and access hatches, to facilitate the installation and maintenance of the turbine. The base may also be designed to incorporate vibration damping and noise reduction measures, to minimize the impact of the turbine on the surrounding environment.

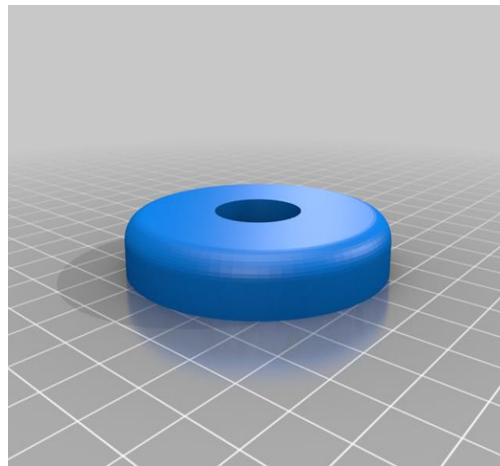


Fig 2.8.1 TURBINE STAND BASE

2.8.2 TURBINE STAND:

The turbine stand is typically comprised of two main parts: the base and the tower. The base is the foundation of the stand and is typically buried several feet into the ground to provide maximum stability. The tower is the vertical structure that rises above the base and supports the turbine blades.

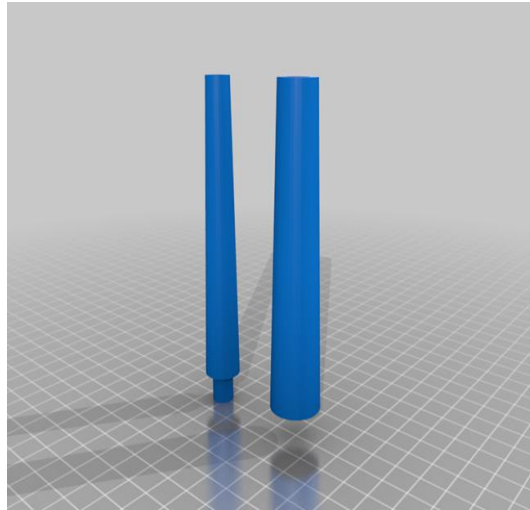


Fig 2.8.2 TURBINE STAND

2.8.3 TURBINE MOTOR MOUNT:

A turbine motor mount is a component of a wind turbine that holds the generator or motor in place. It is a crucial part of the turbine's drivetrain, which converts the kinetic energy of the wind into electrical energy. In addition to providing support for the generator or motor, the motor mount may also incorporate features such as vibration dampening and noise reduction measures, to minimize the impact of the turbine on the surrounding environment.

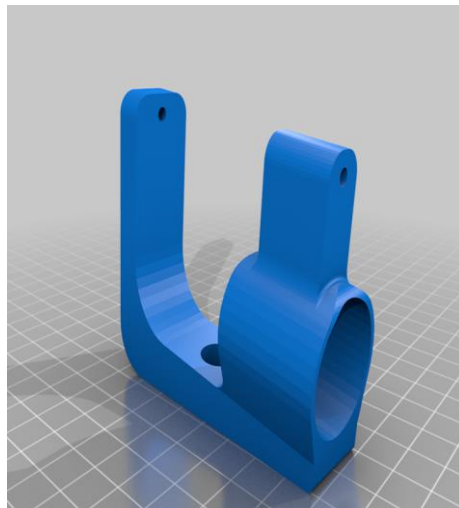


FIG 2.8.3 TURBINE MOTOR MOUNT

2.8.4 BLADE HUB:

The turbine blade hub is a critical component of a wind turbine, as it connects the rotor blades to the rotor shaft, and allows them to capture the kinetic energy of the wind and convert it into rotational energy. The hub is typically made of high-strength steel or aluminum alloy, and is designed to withstand the forces of wind and other environmental factors. The hub is typically mounted at the end of the rotor shaft, and the blades are attached to the hub using a series of bolts or other fasteners.

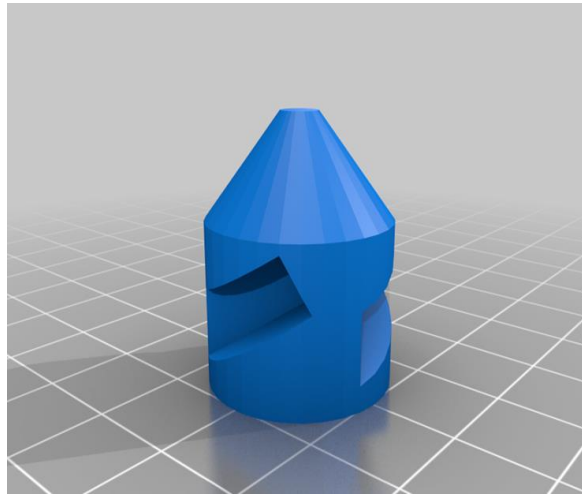


FIG 2.8.4 BLADE HUB

2.8.5 TURBINE BLADE:

The turbine blade is a crucial component of a wind turbine, as it captures the kinetic energy of the wind and converts it into rotational energy that drives the generator or motor. Turbine blades are typically made of composite materials such as fiberglass, carbon fiber, or Kevlar, which provide a high strength-to-weight ratio and durability in the face of harsh weather conditions and high winds. The blades are designed to be aerodynamic, with a curved shape that allows them to efficiently capture the energy of the wind and transfer it to the rotor shaft.

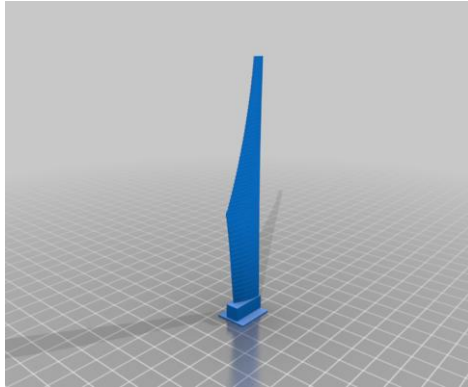


FIG 2.8.5 TURBINE BLADE

2.8.6 TURBINE BIG GEAR:

The "turbine big gear" most likely refers to the large gear that is typically used in steam turbines. These turbines are commonly used in power generation and other industrial applications. The purpose of the big gear in a turbine is to transmit the rotational energy generated by the turbine rotor to the generator or other equipment that is connected to it. The big gear is usually made of high-strength steel and is designed to withstand the extreme loads and stresses that are placed on it during operation.

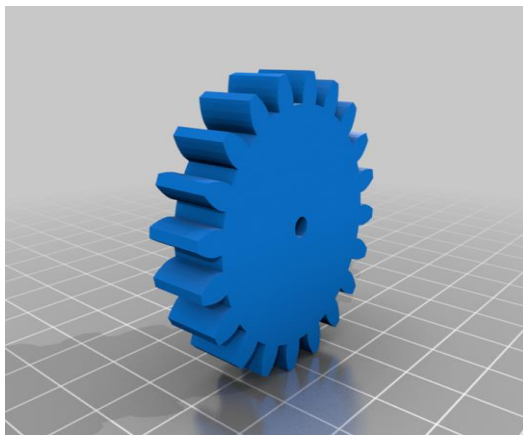


FIG 2.8.6 TURBINE BIG GEAR

2.8.7 TURBINE SMALL GEAR:

The "turbine small gear" is another important component in steam turbines, which work in conjunction with the big gear to

transmit the rotational energy generated by the turbine rotor. The small gear is typically located at the end of the turbine rotor shaft and meshes with the big gear to transmit the torque to the generator or other equipment.

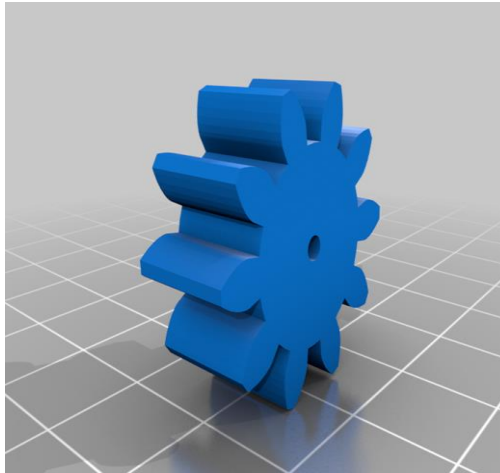


FIG 2.8.7 TURBINE SMALL GEAR

2.9 ASSEMBLING PARTS



FIG 2.9 ASSEMBLING PARTS

After assembling all the turbine parts and sensors, the final prototype of Wind Twin is ready

3. SOFTWARE SYSTEM DESIGN

System design is the process of designing the elements of a system such as the architecture, modules and components, the different interfaces of those components and the data that goes through that system. System Analysis is the process that decomposes a system into its component pieces for the purpose of defining how well those components interact to accomplish the set requirements.

The purpose of the System Design process is to provide sufficient detailed data and information about the system. The purpose of the design phase is to plan a solution of the problem specified by the requirement document. This phase is the first step in moving from problem domain to the solution domain. The design of a system is perhaps the most critical factor affecting the quality of the software, and has a major impact on the later phases, particularly testing and maintenance.

The design activity is often divided into two separate phase-system design and detailed design. System design, which is sometimes also called top-level design, aims to identify the modules that should be in the system, the specifications of these modules, and how they interact with each other to produce the desired results.

A design methodology is a systematic approach to creating a design by application of set of techniques and guidelines. Most methodologies focus on system design. The two basic principles used in any design methodology are problem partitioning and abstraction. Abstraction is a concept related to a problem.

3.1 PROCESS FLOW DIAGRAM:

A process flowchart is a graphical representation of a business process through flowchart. It's used as a means of getting a top-down understanding of how a process works, what steps it consists of, what events change outcomes, and so on.

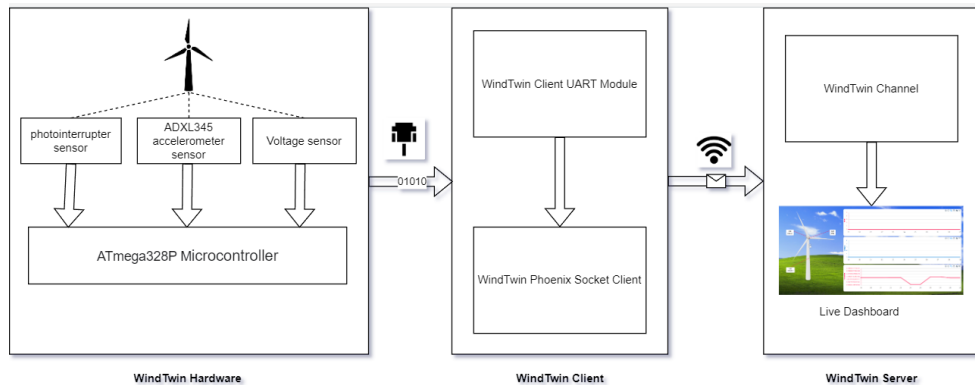


Fig 3.1 Process Flow Diagram

4. SRS DOCUMENT

SRS is a document created by a system analyst after the requirements are collected from various stakeholders. SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

The requirements received from clients are written in natural language. It is the responsibility of the system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team. The introduction of the software requirement specification states the goals and objectives of the software, describing it in the context of the computer-base system. The SRS includes an information description, functional description, behavioral

description, validation criteria.

The purpose of this document is to present the software requirements in a precise and easily understood manner. This document provides the functional, performance, design and verification requirements of the software to be developed.

After requirement specifications are developed, the requirements mentioned in this document are validated. Users might ask for illegal, impractical solutions or experts may interpret the requirements incorrectly. This results in a huge increase in cost if not nipped in the bud.

4.1 FUNCTIONAL REQUIREMENTS

The functional requirements for a digital twin of a wind turbine may include the following

- Data acquisition and storage: The digital twin should be able to collect and store real-time data from sensors installed on the wind turbine, such as Rpm of the turbine, vibration of the turbine body, and voltage generating.
- Simulation modeling: The digital twin should be able to create a simulation model of the wind turbine, which can be used to predict its behavior and performance under different operating conditions.
- Predictive maintenance: The digital twin should be able to detect any anomalies or defects in the wind turbine and predict the maintenance requirements to prevent any potential failures.
- Performance monitoring: The digital twin should be able to monitor the performance of the wind turbine in real-time and compare it to the expected performance based on the simulation model.
- Control optimization: The digital twin should be able to optimize the control settings of the wind turbine to achieve maximum efficiency and reduce wear and tear on the equipment.
- Visualization and reporting: The digital twin should be able to

visualize the performance and status of the wind turbine in a user-friendly way and generate reports for further analysis and decision-making.

4.2 NON-FUNCTIONAL REQUIREMENTS

Some Non-Functional requirements for a digital twin for a wind turbine may include:

- **Reliability:** The digital twin should be reliable, accurate and consistent in its performance, and provide dependable results for decision-making.
- **Scalability:** The digital twin should be able to handle large amounts of data and be scalable enough to accommodate future upgrades or changes in the system.
- **Security:** The digital twin should have robust security measures in place to protect sensitive data and prevent unauthorized access.
- **Performance:** The digital twin should have high performance and fast response time, to support real-time decision-making and analysis.
- **Interoperability:** The digital twin should be able to integrate with other systems and platforms in the wind turbine's ecosystem, such as SCADA systems or maintenance management systems.
- **Usability:** The digital twin should be user-friendly and easy to navigate, with an intuitive interface that allows users to access the required information quickly.
- **Maintainability:** The digital twin should be easy to maintain, with well-defined procedures for upgrades, backups, and troubleshooting.

4.3 MINIMUM HARDWARE REQUIREMENTS

HARDWARE REQUIREMENTS: Minimum hardware requirements for this project are

1. Processor -Intel Core
2. Duo or Higher 2.Hard Disk – 1 GB
3. RAM - 2 GB

4.4 MINIMUM SOFTWARE REQUIREMENTS

SOFTWARE REQUIREMENTS: Minimum software requirements are

1. ASDF Version Manager
2. Erlang/OTP 24.0.3
3. Elixir language support
4. Visual Studio Cod

5. OUTPUT

5.1 SYSTEM IMPLEMENTATION

5.1.1 INTRODUCTION:

- WindTwin is a software platform designed to monitor and optimize the performance of wind turbines in real-time. The system implementation process for WindTwin involves a series of steps to ensure that the software is installed, configured, and integrated with your wind turbine correctly.
- This includes identifying the scope of the project, setting objectives, establishing timelines.

- A digital twin for a wind turbine typically includes a detailed model of the physical turbine, along with data from sensors and other sources that provide real-time information on its operation.
- This data can be used to simulate different operating scenarios and to predict the turbine's behavior under various conditions..
- Digital twins for wind turbines also enables predictive maintenance, allowing operators to identify potential problems before they occur and schedule maintenance and repairs accordingly. This can help to minimize downtime and reduce maintenance costs, while also improving the overall reliability of wind turbines.

5.2 PROJECT MODULES:

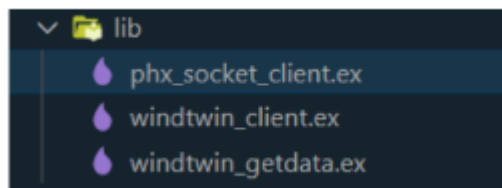
5.2.1 WindTwin Client:

The WindTwin Client is responsible for receiving data from the Arduino Uno via serial communication and then transmitting this information to the WindTwin Server. This allows the WindTwin software to collect and analyze data from the wind turbine in real-time, providing insights into the turbine's performance and identifying potential issues before they become more significant problems. The WindTwin Client acts as a bridge between the hardware components of the wind turbine and the software platform, enabling seamless communication and integration between the two.

In addition to receiving data from the Arduino Uno, the WindTwin Client also has the responsibility of managing and displaying the data received from the WindTwin Server. This includes displaying real-time performance data, providing alerts and notifications when issues are detected, and

generating reports on the turbine's performance over time. The WindTwin Client is a critical component of the WindTwin system, enabling operators to monitor and optimize the performance of their wind turbines with ease and efficiency.

WindTwin Client is a elixir mix project consists of 3 important files



Phx_socket_client.ex:

```
defmodule WindtwinClient.PhoenixSocketClient do
  use GenServer alias PhoenixClient.{Socket, Channel, Message}
  def connect_server do
    socket_opts = [url: "ws://192.168.1.3:4000/socket/websocket"]
    {:ok, socket} = PhoenixClient.Socket.start_link(socket_opts)
    wait_until_connected(socket)
    {:ok, _response, channel} = PhoenixClient.Channel.join(socket, "twin:lobby")
    IO.puts("Connected..")
    if !Enum.member?(Process.registered(), :con) do
      Process.register(channel, :con)
    else
      Process.unregister(:con)
      Process.register(channel, :con)
    end
    IO.puts("Registered..")
    {:ok, _response, channel}
  end

  defp wait_until_connected(socket) do
    if !PhoenixClient.Socket.connected?(socket) do
      Process.sleep(100)
      wait_until_connected(socket)
    end
  end

  def send_status(data) do
    PhoenixClient.Channel.push_async(:con, "updatereal", %{data: data})
    IO.puts("Data: #{data}")
  end
end
```

This is an Elixir module defining a Phoenix socket client using the

GenServer behavior. It allows a client

to connect to a Phoenix server using a WebSocket connection and join a channel named `twin:lobby`.

The `connect_server` function is used to establish a connection to the Phoenix server, and it does the

following:

- Creates a Socket instance with the WebSocket URL specified in `socket_opts`.
- Calls `wait_until_connected` to wait for the socket to connect to the server.
- Joins the `twin:lobby` channel using `PhoenixClient.Channel.join`.
- Registers the channel process using `Process.register` to allow it to be referenced later.
- Returns the channel process.

The `send_status` function is used to push data to the `update_real` event on the channel. The data is

passed in as an argument and sent asynchronously to the channel using **`PhoenixClient.Channel.push_async`**.

The `wait_until_connected` function is a private function that waits until the Socket is connected to the

server before continuing. It uses recursion and sleeps for 100ms between attempts.

Overall, this module defines a simple Phoenix socket client that can be used to establish a connection to

a Phoenix server and push data to a specific channel.

Windtwin_getdata.ex:

This is an Elixir module that defines a **GenServer** process which reads data from a UART device and sends it to a Phoenix socket server using the

WindtwinClient.PhoenixSocketClient module.

The `start_link` function is called to start the **GenServer** process and it takes `opts` as an argument.

It starts a UART process using **Circuits.UART.start_link()** and configures it to use line framing with the separator `"\r\n"`. It then opens the UART connection with the device specified in the **Circuits.UART.open** function and starts the loop function in a separate process.

The `init` function initializes the **GenServer** process and it takes no arguments. It returns a tuple of `{:ok, []}` after starting the UART process and calling a `loop` function to read data from the UART device.

```
defmodule WindtwinClient.WindtwinGetdata do
  use GenServer

  def start_link(opts) do
    IO.puts("Starting Connction With Hardware..")
    GenServer.start_link(__MODULE__, [], opts)
  end

  def init(_) do
    {:ok, pid} = Circuits.UART.start_link()
    Circuits.UART.configure(pid,
      framing: {Circuits.UART.Framing.Line, separator: "\r\n"})

    Circuits.UART.open(pid, "COM5", speed: 9600, active: false)
    IO.puts("Getting Data..")
    loop(pid)
    {:ok, []}
  end

  def loop(pid) do
    IO.puts("-")

    case Circuits.UART.read(pid, 10000) do
      {:ok, data} ->
        WindtwinClient.PhoenixSocketClient.send_status(data)

      {:error, _} ->
        :ok
    end

    loop(pid)
  end
end
```

In the loop function, the `Circuits.UART.read` function reads data from the UART device, and if data is successfully read, it sends it to the Phoenix socket

server.

WindtwinClient.PhoenixSocketClient.send_status function. To send and read data continuously, the function is called recursively. Overall, this module provides a way to read data from a UART device and send it to a Phoenix socket server using a **GenServer** process.

windtwin_client.ex:

```
defmodule WindtwinClient do
  def start do
    WindtwinClient.PhoenixSocketClient.connect_server()
    WindtwinClient.WindtwinGetdata.start_link([])
  end
end
```

GenServer is started by calling the `start_link` function, which takes an argument called `opts`.

This is an **Elixir** module that defines a `start` function which starts the Phoenix socket client and UART device connection by calling the **WindtwinClient.PhoenixSocketClient.connect_server()** and **WindtwinClient.WindtwinGetdata.start_link([])** functions respectively.

When the `start` function is called, it first calls **WindtwinClient.PhoenixSocketClient.connect_server()** to establish a connection with the Phoenix socket server. It then calls **WindtwinClient.WindtwinGetdata.start_link([])** to start the **GenServer** process that reads data from the UART device and sends it to the Phoenix socket server.

5.2.2 WindTwin Hardware:

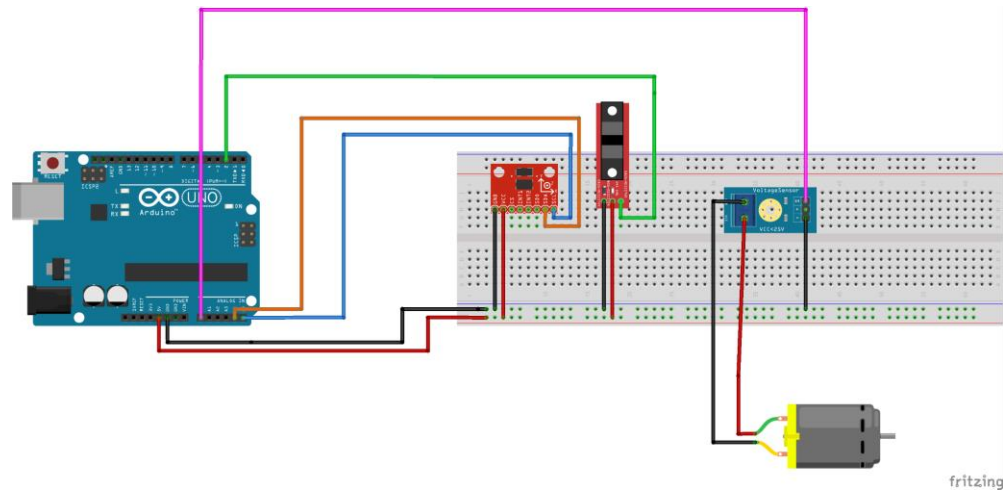


FIG 5.2.2 WindTwin Hardware

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_ADXL345_U.h>
Adafruit_ADXL345_Unified accel = Adafruit_ADXL345_Unified(12345);
#include "TimerOne.h"
const byte MOTOR1 = 2; // Motor 1 Interrupt Pin - INT 0
// Integers for pulse counters
unsigned int counter1 = 0;
// Float for number of slots in encoder disk
float diskslots = 20; // Change to match value of encoder disk
float rotation1=0.00;
//voltage Sensor
float vOUT = 0.0;
float vIN = 0.0;
float R1 = 30000.0;
float R2 = 7500.0;
// Interrupt Service Routines
// Motor 1 pulse count ISR
void ISR_count1()
{
  counter1++; // increment Motor 1 counter value
}

// TimerOne ISR
void ISR_timerone()
{
  Timer1.detachInterrupt(); // Stop the timer
```

```

Serial.print("{\\M\\":\\");
rotation1 = (counter1 / diskslots) * 60.00; // calculate RPM for Motor 1
Serial.print(rotation1);
counter1 = 0; // reset counter to zero
Timer1.attachInterrupt( ISR_timerone ); // Enable the timer
}

void setup()
{
    Serial.begin(9600);
    if (!accel.begin()) {
        Serial.println("Could not find a valid ADXL345 sensor, check wiring!");
        // while (1);
    }
    accel.setRange(ADXL345_RANGE_16_G);
    Timer1.initialize(1000000); // set timer for 1sec
    attachInterrupt(digitalPinToInterrupt (MOTOR1), ISR_count1, RISING); //
    // Increase counter 1 when speed sensor pin goes High
    Timer1.attachInterrupt( ISR_timerone ); // Enable the timer
}

void loop()
{
    //For Vibration
    sensors_event_t event;
    accel.getEvent(&event);
    float x = event.acceleration.x;
    float y = event.acceleration.y;
    float z = event.acceleration.z;
    Serial.print("\\", "\\X_Y_Z\\":\\");
    Serial.print(x);
    Serial.print("_");
    Serial.print(y);
    Serial.print("_");
    Serial.print(z);
    //For Voltage
    int sensorValue = analogRead(A0);
    vOUT = (sensorValue * 5.0) / 1024.0;
    vIN = vOUT / (R2/(R1+R2));
    Serial.print("\\", "\\V\\":\\");
    Serial.print(vIN);
    Serial.println("\\}");
    delay(1000);
}

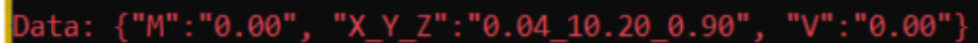
```

This code is written for an Arduino board and uses various libraries and hardware components to measure motor speed, vibration, and voltage. The code initializes the Serial communication with a baud rate of 9600. It then checks if an ADXL345 sensor is connected and sets its range to $\pm 16g$. The code also initializes TimerOne library with a timer of 1 second and attaches an interrupt to the Motor 1 interrupt pin (INT0) with the rising edge triggering.

In the **ISR_count1()** interrupt service routine, the counter for **Motor 1** is incremented each time the interrupt is triggered. In the **ISR_timerone()**

interrupt service routine, the timer is stopped, and the **RPM** for Motor 1 is calculated using the number of pulses counted by the interrupt and the number of slots in the encoder disk. The calculated RPM is then printed to the Serial monitor, and the counter for Motor 1 is reset to zero.

The loop() function gets the acceleration values for the ADXL345 sensor and prints them to the Serial monitor. It also measures the voltage at pin A0 using a voltage divider circuit and prints the measured voltage to the Serial monitor. The loop function then waits for 1 second using the delay() function before repeating the measurements.



```
Data: {"M":"0.00", "X_Y_Z":"0.04_10.20_0.90", "V":"0.00"}
```

FIG 5.2.1 Understanding the output from the UNO

After executing the code in the UNO board we get output like:

This JSON represents the 3 parameters.

“M” => Motor Speed in RPM (More info (Usage))

“X_Y_Z” => Values generating from ADXL345 Sensor (X,YZ) coordinates (More info (ADXL345) (Usage))

“V” => Voltage generating from the wind turbine (More info (Usage))

5.2.3 WindTwin Server:

The responsibility of the WindTwin Server is to receive data from the client through websocket

communication and render the same in the dashboard.

WindTwin Server is using the following to render real time information to dashboard

1. Phoenix web development framework (refer)
2. Model Viewer (refer)
3. Apexcharts (refer)

Phoenix web development framework:

- Phoenix is an Elixir-based web development framework that follows the server-side Model View Controller (MVC) pattern.
- Phoenix offers the best of both worlds in terms of developer productivity and application performance. It also includes some unique new features, such as channels for implementing real-time features and pre-compiled templates for lightning-fast development.

Model Viewer :

- Model Viewer is a web-based tool that allows users to display 3D models on websites or in augmented reality (AR) experiences. It was developed by Google as an open-source project that supports a wide range of 3D file formats, including OBJ, FBX, GLTF, and STL.
- With Model Viewer, users can view and interact with 3D models directly in their web browser or in AR mode on their mobile devices without needing to install any additional software. This makes it a convenient and accessible tool for designers, developers, and businesses looking to showcase their products or designs in 3D.
- Model Viewer also includes features for customization and interactivity, such as the ability to add annotations, animations, and hotspots to guide viewers through the model. Additionally, it supports various lighting and shading effects to enhance the realism of the 3D models.

5.3 SOURCE CODE:

DT_HW_Code.ino:

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_ADXL345_U.h>
Adafruit_ADXL345_Unified accel =
```

```

Adafruit_ADXL345_Unified(12345);

#include "TimerOne.h"

const byte MOTOR1 = 2;  // Motor 1 Interrupt Pin - INT 0

// Integers for pulse counters
unsigned int counter1 = 0;

// Float for number of slots in encoder disk
float diskslots = 20;  // Change to match value of encoder
disk
float rotation1=0.00;

//voltage Sensor
float vOUT = 0.0;
float vIN = 0.0;
float R1 = 30000.0;
float R2 = 7500.0;

// Interrupt Service Routines

// Motor 1 pulse count ISR
void ISR_count1()
{
    counter1++;  // increment Motor 1 counter value
}

// TimerOne ISR
void ISR_timerone()
{
    Timer1.detachInterrupt();  // Stop the timer
    Serial.print("{\\"M\\":\\"");
    rotation1 = (counter1 / diskslots) * 60.00;  //
calculate RPM for Motor 1
    Serial.print(rotation1);

    counter1 = 0;  // reset counter to zero
    Timer1.attachInterrupt( ISR_timerone );  // Enable the
timer

```

```

}

void setup()
{
    Serial.begin(9600);
    if (!accel.begin()) {
        Serial.println("Could not find a valid ADXL345 sensor,
check wiring!");
        // while (1);
    }
    accel.setRange(ADXL345_RANGE_16_G);
    Timer1.initialize(1000000); // set timer for 1sec
    attachInterrupt(digitalPinToInterrupt (MOTOR1),
ISR_count1, RISING); // Increase counter 1 when speed
sensor pin goes High
    Timer1.attachInterrupt( ISR_timerone ); // Enable the
timer
}

void loop()
{
    //For Vibration
    sensors_event_t event;
    accel.getEvent(&event);
    float x = event.acceleration.x;
    float y = event.acceleration.y;
    float z = event.acceleration.z;
    Serial.print("\", \"X_Y_Z\":\");
    Serial.print(x);
    Serial.print("_");
    Serial.print(y);
    Serial.print("_");
    Serial.print(z);
    //For Voltage
    int sensorValue = analogRead(A0);
    vOUT = (sensorValue * 5.0) / 1024.0;
    vIN = vOUT / (R2/(R1+R2));
    Serial.print("\", \"V\":\");
    Serial.print(vIN);

```

```

Serial.println("\n");
delay(1000);
}

```

Twin_Channel.ex:

```

defmodule WindTwinWeb.TwinChannel do
  use WindTwinWeb, :channel

  def join("twin:lobby", _payload, socket) do
    WindTwinWeb.Endpoint.subscribe("twin:update")
    :ok = Phoenix.PubSub.subscribe(WindTwin.PubSub,
    "twin:update")

    {:ok, socket}
  end

  def handle_in("update", data, socket) do
    IO.puts(" #{inspect data["data"]} #{inspect
data["labels"]} ")
    broadcast_from(socket, "update", data)
    {:noreply, socket}
  end

  def handle_in("updatereal", data, socket) do
    IO.puts(" #{inspect data}")
    broadcast!(socket, "updatereal", data)
    WindTwinWeb.Endpoint.broadcast_from(self(),
    "twin:update", "updatereal", data)

    {:noreply, socket}
  end
end

```



```

def handle_out(e,p,socket) do
  push(socket, e, p)
  IO.puts("[RoomChannel]  handle out")
  IO.inspect(e)
  IO.inspect(p)
  {:noreply, socket}
end

def handle_info(data, socket) do
  IO.puts("[RoomChannel]  info")
  IO.inspect(data)
  socket= cond do
    data.event == "light" ->
      IO.puts("[RoomChannel]  handle_info[m]")
      Phoenix.Channel.push(socket, "light",
data.payload)
      socket

    true ->
      socket
  end

  {:noreply, socket}
end

# Add authorization logic here as required.
defp authorized?(_payload) do
  true
end
end

```

Application.ex:

```

defmodule WindTwin.Application do
  # See https://hexdocs.pm/elixir/Application.html
  # for more information on OTP Applications
  @moduledoc false

```

```

use Application

@impl true
def start(_type, _args) do
  children = [
    # Start the Telemetry supervisor
    WindTwinWeb.Telemetry,
    # Start the PubSub system
    {Phoenix.PubSub, name: WindTwin.PubSub},
    # Start the Endpoint (http/https)
    WindTwinWeb.Endpoint
    # Start a worker by calling:
    WindTwin.Worker.start_link(arg)
    # {WindTwin.Worker, arg}
  ]

  # See https://hexdocs.pm/elixir/Supervisor.html
  # for other strategies and supported options
  opts = [strategy: :one_for_one, name:
WindTwin.Supervisor]
  Supervisor.start_link(children, opts)
end

# Tell Phoenix to update the endpoint configuration
# whenever the application is updated.
@impl true
def config_change(changed, _new, removed) do
  WindTwinWeb.Endpoint.config_change(changed, removed)
  :ok
end
end

```

WindTwin_live.ex:

```

defmodule WindTwinWeb.WindTwinLive do
  use WindTwinWeb, :live_view

  def mount(_params, _session, socket) do
    WindTwinWeb.Endpoint.subscribe("twin:update")
  end
end

```

```

      :ok = Phoenix.PubSub.subscribe(WindTwin.PubSub,
"twin:update")

      socket=assign(socket, :hi, "test")

      {:ok, socket}
    end

    def render(assigns) do
      ~H"""
      <div style="display:none" class=" p-5 h-2/6 grid gap-
y-10 gap-x-6 md:grid-cols-3 xl:grid-cols-3">

        <div class="relative flex flex-col bg-clip-
border rounded-xl bg-white text-gray-700 shadow-md">
          <div
            class="bg-clip-border mx-4 rounded-xl
overflow-hidden bg-gradient-to-tr from-green-600 to-green-
400 text-white shadow-blue-500/40 shadow-lg absolute -mt-4
grid h-16 w-16 place-items-center">
              <svg xmlns="http://www.w3.org/2000/svg"
viewBox="0 0 24 24" fill="currentColor" aria-hidden="true"
                class="w-6 h-6 text-white">
                  <path d="M12 7.5a2.25 2.25 0 100 4.5
2.25 2.25 0 000-4.5z"></path>
                  <path fill-rule="evenodd"
                    d="M1.5 4.875C1.5 3.839 2.34 3
3.375 3h17.25c1.035 0 1.875.84 1.875 1.875v9.75c0 1.036-
.84 1.875-1.875 1.875H3.375A1.875 1.875 0 011.5 14.625v-
9.75zM8.25 9.75a3.75 3.75 0 117.5 0 3.75 3.75 0 01-7.5
0zM18.75 9a.75.75 0 00-.75.75v.008c0
.414.336.75.75h.008a.75.75 0 00.75-.75V9.75a.75.75 0
00-.75-.75h-.008zM4.5 9.75A.75.75 0 015.25 9h.008a.75.75 0
01.75.75v.008a.75.75 0 01-.75.75H5.25a.75.75 0 01-.75-
.75V9.75z"
                    clip-rule="evenodd"></path>
                  <path
                    d="M2.25 18a.75.75 0 000 1.5c5.4
0 10.63.722 15.6 2.075 1.19.324 2.4-.558 2.4-
1.82V18.75a.75.75 0 00-.75-.75H2.25z">

```

```

        </path>
    </svg></div>
    <div class="p-4 text-right">
        <p class="block antialiased font-sans text-sm leading-normal font-normal text-blue-gray-600">Current Speed</p>
        <h4
            class="block antialiased tracking-normal font-sans text-2xl font-semibold leading-snug text-blue-gray-900">
            15RPM</h4>
    </div>
    <div class="border-t border-blue-gray-50 p-4">
        <p class="block antialiased font-sans text-base leading-relaxed font-normal text-blue-gray-600"><strong
            class="text-green-500">+55%</strong>&nbsp;than last week</p>
    </div>
</div>

<div class="relative flex flex-col bg-clip-border rounded-xl bg-white text-gray-700 shadow-md">
    <div
        class="bg-clip-border mx-4 rounded-xl overflow-hidden bg-gradient-to-tr from-blue-600 to-blue-400 text-white shadow-blue-500/40 shadow-lg absolute -mt-4 grid h-16 w-16 place-items-center">
        <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" fill="currentColor" aria-hidden="true"
            class="w-6 h-6 text-white">
            <path d="M12 7.5a2.25 2.25 0 100 4.5 2.25 2.25 0 00-4.5z"></path>
            <path fill-rule="evenodd"
                d="M1.5 4.875C1.5 3.839 2.34 3 3.375 3h17.25c1.035 0 1.875.84 1.875 1.875v9.75c0 1.036-.84 1.875-1.875 1.875H3.375A1.875 1.875 0 011.5 14.625v-9.75zM8.25 9.75a3.75 3.75 0 117.5 0 3.75 3.75 0 01-7.5 0zM18.75 9a.75.75 0 00-.75.75v.008c0

```

```

.414.336.75.75.75h.008a.75.75 0 00.75-.75V9.75a.75.75 0
00-.75-.75h-.008zM4.5 9.75A.75.75 0 015.25 9h.008a.75.75 0
01.75.75v.008a.75.75 0 01-.75.75H5.25a.75.75 0 01-.75-
.75V9.75z"
        clip-rule="evenodd"></path>
    <path
        d="M2.25 18a.75.75 0 000 1.5c5.4 0
10.63.722 15.6 2.075 1.19.324 2.4-.558 2.4-
1.82V18.75a.75.75 0 00-.75-.75H2.25z">
    </path>
</svg></div>
<div class="p-4 text-right">
    <p class="block antialiased font-sans
text-sm leading-normal font-normal text-blue-gray-
600">Voltage</p>
    <h4
        class="block antialiased tracking-
normal font-sans text-2xl font-semibold leading-snug text-
blue-gray-900">
        10V</h4>
</div>
<div class="border-t border-blue-gray-50 p-4">
    <p class="block antialiased font-sans
text-base leading-relaxed font-normal text-blue-gray-
600"><strong
        class="text-green-
500">+55%</strong>&nbsp;than last week</p>
</div>
</div>

<div class="relative flex flex-col bg-clip-
border rounded-xl bg-white text-gray-700 shadow-md">
    <div
        class="bg-clip-border mx-4 rounded-xl
overflow-hidden bg-gradient-to-tr from-blue-600 to-blue-
400 text-white shadow-blue-500/40 shadow-lg absolute -mt-4
grid h-16 w-16 place-items-center">
        <svg xmlns="http://www.w3.org/2000/svg"
viewBox="0 0 24 24" fill="currentColor" aria-hidden="true"
            class="w-6 h-6 text-white">

```

```

        <path d="M12 7.5a2.25 2.25 0 100 4.5
2.25 2.25 0 000-4.5z"></path>
        <path fill-rule="evenodd"
            d="M1.5 4.875C1.5 3.839 2.34 3 3.375
3h17.25c1.035 0 1.875.84 1.875 1.875v9.75c0 1.036-.84
1.875-1.875 1.875H3.375A1.875 1.875 0 011.5 14.625v-
9.75zM8.25 9.75a3.75 3.75 0 117.5 0 3.75 3.75 0 01-7.5
0zM18.75 9a.75.75 0 00-.75.75v.008c0
.414.336.75.75.75h.008a.75.75 0 00.75-.75V9.75a.75.75 0
00-.75-.75h-.008zM4.5 9.75A.75.75 0 015.25 9h.008a.75.75 0
01.75.75v.008a.75.75 0 01-.75.75H5.25a.75.75 0 01-.75-
.75V9.75z"
            clip-rule="evenodd"></path>
        <path
            d="M2.25 18a.75.75 0 000 1.5c5.4 0
10.63.722 15.6 2.075 1.19.324 2.4-.558 2.4-
1.82V18.75a.75.75 0 00-.75-.75H2.25z">
        </path>
    </svg></div>
    <div class="p-4 text-right">
        <p class="block antialiased font-sans text-
sm leading-normal font-normal text-blue-gray-600">Body
Vibration</p>
        <h4
            class="block antialiased tracking-normal
font-sans text-2xl font-semibold leading-snug text-blue-
gray-900">
            500</h4>
    </div>
    <div class="border-t border-blue-gray-50 p-4">
        <p class="block antialiased font-sans text-
base leading-relaxed font-normal text-blue-gray-
600"><strong
            class="text-green-
500">+55%</strong>&nbsp;than last week</p>
    </div>
</div>
    "" ""

```

```

end

def handle_info(%{event: "update", payload: payload},
socket) do
  IO.puts("[Live][Test] #{inspect payload} ")
  socket=assign(socket, :hi, payload["light_status"])
  socket=push_event(socket, "update", payload)
  {:noreply, socket}
end

def handle_info(%{event: "updatereal", payload:
payload}, socket) do
  IO.puts("[Live][Real] #{inspect payload} ")
  socket=push_event(socket, "updatereal", payload)
  {:noreply, socket}
end

end

```

App.html.hex:

```

<main class="container">
  <p class="alert alert-info" role="alert"><%= get_flash(@conn, :info) %></p>
  <p class="alert alert-danger" role="alert"><%= get_flash(@conn, :error) %></p>
  <%= @inner_content %>
</main>

```

Router.ex:

```

defmodule WindTwinWeb.Router do
  use WindTwinWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_live_flash
    plug :put_root_layout, {WindTwinWeb.LayoutView, :root}
  end
end

```

```

    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", WindTwinWeb do
    pipe_through :browser

    get "/", PageController, :index

    live "/windTwin", WindTwinLive
  end

  # Other scopes may use custom stacks.
  # scope "/api", WindTwinWeb do
  #   pipe_through :api
  # end

  # Enables LiveDashboard only for development
  #
  # If you want to use the LiveDashboard in production,
  you should put
  # it behind authentication and allow only admins to
  access it.
  # If your application does not have an admins-only
  section yet,
  # you can use Plug.BasicAuth to set up some basic
  authentication
  # as long as you are also using SSL (which you should
  anyway).
  if Mix.env() in [:dev, :test] do
    import Phoenix.LiveDashboard.Router

    scope "/" do
      pipe_through :browser

      live_dashboard "/dashboard", metrics:

```



```

WindTwinWeb.Telemetry
  end
end

# Enables the Swoosh mailbox preview in development.
#
# Note that preview only shows emails that were sent by
the same
# node running the Phoenix server.
if Mix.env() == :dev do
  scope "/dev" do
    pipe_through :browser

    forward "/mailbox", Plug.Swoosh.MailboxPreview
  end
end
end
end

```

WindTwin_Web.ex:

```

defmodule WindTwinWeb do

  def controller do
    quote do
      use Phoenix.Controller, namespace: WindTwinWeb

      import Plug.Conn
      import WindTwinWeb.Gettext
      alias WindTwinWeb.Router.Helpers, as: Routes
    end
  end

  def view do
    quote do
      use Phoenix.View,
        root: "lib/windTwin_web/templates",
        namespace: WindTwinWeb
    end
  end
end

```

```

    # Import convenience functions from controllers
    import Phoenix.Controller,
      only: [get_flash: 1, get_flash: 2, view_module: 1,
view_template: 1]

    # Include shared imports and aliases for views
    unquote(view_helpers())
  end
end

def live_view do
  quote do
    use Phoenix.LiveView,
      layout: {WindTwinWeb.LayoutView, "live.html"}

    unquote(view_helpers())
  end
end

def live_component do
  quote do
    use Phoenix.LiveComponent

    unquote(view_helpers())
  end
end

def component do
  quote do
    use Phoenix.Component

    unquote(view_helpers())
  end
end

def router do
  quote do
    use Phoenix.Router

    import Plug.Conn

```

```

import Phoenix.Controller
import Phoenix.LiveView.Router
end
end

def channel do
  quote do
    use Phoenix.Channel
    import WindTwinWeb.Gettext
  end
end

defp view_helpers do
  quote do
    # Use all HTML functionality (forms, tags, etc)
    use Phoenix.HTML

    # Import LiveView and .heex helpers (live_render,
live_patch, <.form>, etc)
    import Phoenix.LiveView.Helpers

    # Import basic rendering functionality (render,
render_layout, etc)
    import Phoenix.View

    import WindTwinWeb.ErrorHelpers
    import WindTwinWeb.Gettext
    alias WindTwinWeb.Router.Helpers, as: Routes
  end
end

@doc """
When used, dispatch to the appropriate
controller/view/etc.
"""
defmacro __using__(which) when is_atom(which) do
  apply(__MODULE__, which, [])
end
end

```

5.4 OUTPUT SCREENS:

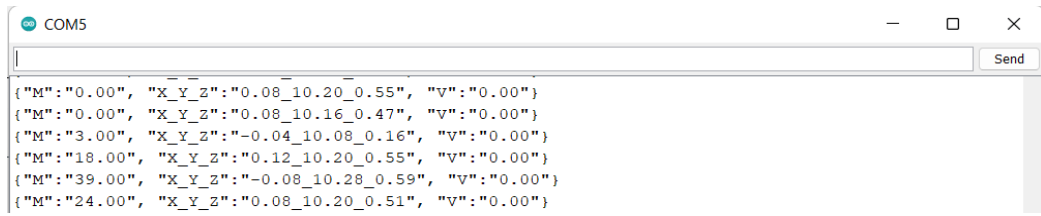


Fig 5.4.1 Output screen 1

Understanding the output from the UNO:

After executing the code in the UNO board we get output like:

```
Data: {"M": "0.00", "X_Y_Z": "0.04_10.20_0.90", "V": "0.00"}
```

Fig 5.4.1 Understanding the output from UNO

This JSON represents the 3 parameters.

“M” => Motor Speed in RPM (More info (Usage))

“X_Y_Z” => Values generating from ADXL345 Sensor (X,YZ) coordinates

“V” => Voltage generating from the wind turbine (More info (Usage))

The above generated data is sent to the WindTwin Server to display in the dashboard has shown below.



FIG 5.4.2 OUTPUT

6. TESTING

Testing is the process of detecting errors. Testing performs a very critical role for quality assurance and for ensuring the reliability of software. The results of testing are used later on during maintenance also. Purpose of Testing: The aim of testing is often to demonstrate that a program works by showing that it has no errors. The basic purpose of testing phase is to detect the errors that may be present in the program. Hence one should not start testing with the intent of showing that a program works, but the intent should be to show that a program doesn't work. Testing Objectives: The main objective of testing is to uncover a host of errors, systematically and with minimum effort and time.

6.1 Testing Strategies

In order to make sure that system does not have any errors, the different levels of testing strategies that are applied at different phases of software development are unit testing, integration testing, system testing and acceptance testing.

Unit Testing

It focuses on smallest unit of software design. In this, testing an individual unit or group of inter related units will be done. It is often done by programmer by using sample input and observing its corresponding outputs. A unit may be an individual function, method, procedure, module, or object. It is a White Box testing technique that is usually performed by the developer.

Integration Testing

The testing of combined parts of an application to determine if they function correctly together is Integration testing. In this, the program is constructed and tested in small increments, where errors are easier

to isolated and correct; interfaces are more likely to be tested completely; and systematic test approach may be applied. This testing can be done by using two different methods

7. Top Down Integration Testing

8. Bottom Up Integration Testing.

System Testing

System Testing is a type of software testing that is performed on a complete integrated system to evaluate the compliance of the system with the corresponding requirements. System testing detects defects within both the integrated units and the whole system.

The result of system testing is the observed behavior of a component or a system when it is tested. System Testing is a black-box testing.

Acceptance Testing

Acceptance Testing is a method of software testing where a system is tested for acceptability. It is a formal testing according to user needs, requirements and business processes conducted to determine whether a system satisfies the acceptance criteria or not and to enable the users, customers or other authorized entities to determine whether to accept the system or not.

White Box Test

White Box Testing is a testing in which the software tester has knowledge of the inner workings, structure and language of the software, or at least its purpose. It is used to test areas that cannot be reached from a black box level.

Black Box Test

Black Box Testing is testing the software without any knowledge of the inner workings, structure language of the module being tested.

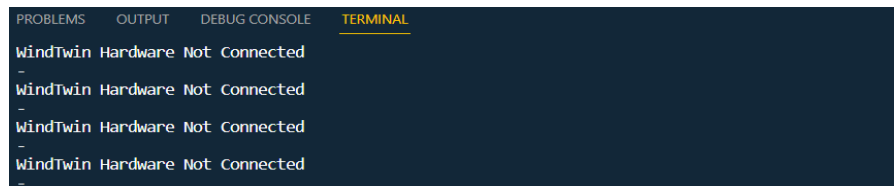
Black box tests, as most other kinds of tests, must be written from a definitive source document, such as specification or requirements

6.2 Testing WindTwin on Different Scenarios

WindTwin is a software application designed to simulate wind turbine behavior and provide insights into the performance of the turbine in different scenarios. To use WindTwin, it is necessary to connect hardware components to the software. This document will provide instructions on how to test the hardware connections to ensure proper functioning of the system.

6.3 WindTwin Hardware Connection Testing

Testing the hardware connections is an important step in using WindTwin software to simulate wind turbine behavior. By following these instructions, you can ensure that the hardware components are properly connected and functioning, allowing you to make accurate and informed decisions about the performance of your wind turbine.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
WindTwin Hardware Not Connected
-
WindTwin Hardware Not Connected
-
WindTwin Hardware Not Connected
-
WindTwin Hardware Not Connected
-
```

FIG 6.3.1 WindTwin Hardware Not Connected to WindTwin Client

Without a connection between the WindTwin hardware and the WindTwin client, the software cannot accurately simulate the wind turbine's behavior. Therefore, the above error in the terminal will appear whenever the hardware connection is not established.

```
Interactive Elixir (1.12.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> WindtwinClient.start
Database connection established
Starting Connection With Hardware..
Getting Data..
-
0.00,0.00,5.916398115971124
-
0.00,0.00,5.847167405390978
-
0.00,0.00,5.8942486091669615
-
0.00,0.00,5.871175350813498
-

```

FIG 6.3.2 If Connected with Hardware component:

6.4 WindTwin Hardware Connection Testing

To ensure that the WindTwin software is functioning properly, it is essential to test the connection to the WindTwin server. By establishing a successful connection, you can use the full capabilities of the software and gain valuable insights into the performance of your wind turbine. Testing the connection to the WindTwin server involves verifying that your internet connection is stable and active, and that you can successfully log in to the WindTwin. By regularly testing the connection to the WindTwin server, you can help ensure that your WindTwin software is always working as intended.

In the image below, you can see how the WindTwin Client displays the dashboard without the server connected.

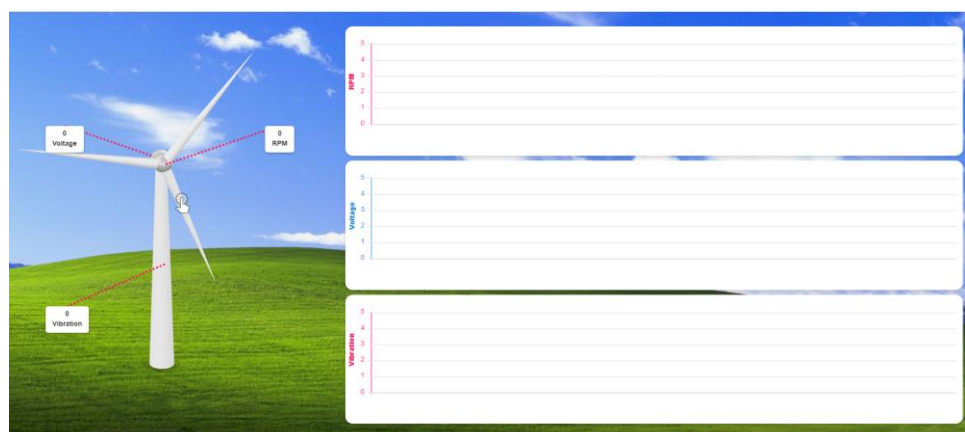


FIG 6.3.3 WindTwin Server is not Connected to WindTwin Client



FIG 6.3.4 WindTwin Server is Connected to WindTwin Client:

If the WindTwin server is connected to the WindTwin client, all data can be displayed in the dashboard. The dashboard provides a real-time view of the wind turbine's performance and includes data such as power output, wind speed, and rotor speed. By having a reliable connection between the WindTwin server and client, you can benefit from the full capabilities of the software and make informed decisions about the operation of your wind turbine.

6.4 WindTwin ADXL345 Sensor Connection Testing

The ADXL345 sensor is a critical component of the WindTwin software that is used to measure the acceleration of the wind turbine's blades. If the ADXL345 sensor is not connected, the WindTwin software will not be able to provide accurate insights into the turbine's performance. The software may display error messages or warnings indicating that the sensor is not connected, and the dashboard may show missing or incomplete data related to the turbine's performance. To ensure that the WindTwin software is functioning properly, it is essential to verify that the ADXL345 sensor is connected and communicating with the WindTwin server.

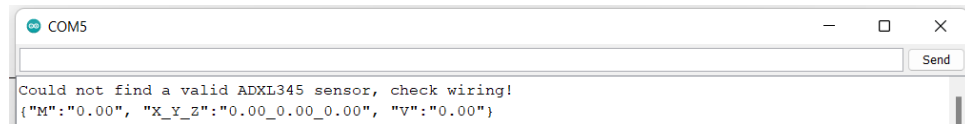


FIG 6.3.5 WindTwin ADXL345 Sensor is not Connected

These error messages indicate that the WindTwin software is unable to retrieve the acceleration data from the ADXL345 sensor, which is critical for monitoring the performance of the wind turbine.

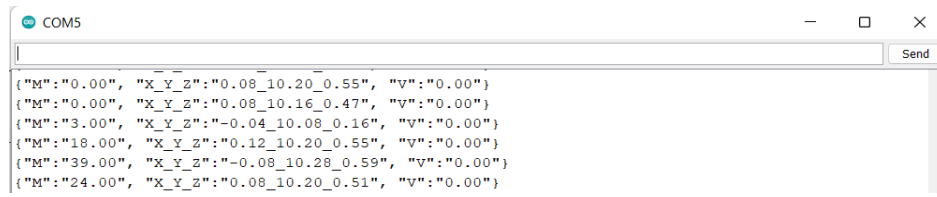


FIG 6.3.6 WindTwin ADXL345 Sensor is Connected

7. CONCLUSION

In recent years, the use of digital twin technology has become increasingly prevalent in the wind energy industry. This approach involves creating a virtual replica of a wind turbine that is continuously updated with real-time data on key parameters such as RPM, voltage, and vibration. By analyzing this data and identifying patterns and trends, operators can gain deeper insights into turbine performance and take proactive measures to ensure their continued operation.

The benefits of digital twin technology for wind turbines are significant. By monitoring key parameters and identifying potential issues before they escalate, operators can improve turbine efficiency and reliability, reduce maintenance costs, and minimize downtime. This approach can also help extend the lifespan of wind turbines, allowing operators to maximize their return on investment.

Furthermore, the use of digital twins can also help operators make more informed decisions when it comes to turbine design and maintenance. By simulating different scenarios and testing various maintenance strategies, operators can determine the most effective approaches to ensure optimal turbine performance.

the benefits of digital twin technology for wind turbines are clear. By leveraging real-time data and simulation tools, operators can gain deeper insights into turbine performance and take proactive measures to ensure their continued operation. As the wind energy industry continues to grow, the use of digital twin technology is likely to become even more prevalent, helping to drive increased efficiency, reliability, and sustainability in this critical sector.

8. REFERENCES

- [1]. Steve Parvin , Portfolio Strategy, AVEVA, "Embracing digital twin technology for engineering assets" by NREL in 2022
- [2]. Michelle Froese , "WINDTWIN PROJECT TO REVOLUTIONISE WIND TURBINE TECHNOLOGY" in 2017
- [3]. Javier Marin, "How to Build a Digital Twin in Python, Virtual systems and digital twins", in 2022
- [4]. Amin Amini,Jamil Kanfound, Tat-Hean Gan "An AI Driven Real-time 3-D Representation of an Off-shore WT for Fault Diagnosis and Monitoring" in 2019
- [5]. Umar Kangiwa Muhammad, Sadik Umar , Muazu Musa , Muhammad Mahmoud Garba, "Effects of Wind Speed Variability on Operation Parameters of an Off-Grid Wind Turbine System" in 2013