Griffith School of Engineering

Griffith University

6002ENG – Industry Affiliates Program

Data Acquisition, Processing and Dashboard Development for Building Sensor Information

Zane Keeley, s2960235

Trimester 1, 2020

Urban Institute

Adriano Marinho

David Rowlands

A report submitted in partial fulfilment of the degree of Bachelor of Engineering (Honours) in Software Engineering

EXECUTIVE SUMMARY

This report will detail the project undertaken as part of the Griffith University Industry Affiliates Program (IAP) that was conducted in partnership with Urban Institute, who are an external commercial company. The objective of this project was to develop a system that could continuously retrieve and store the data from the collection of sensors set to be installed within the recently constructed N79 building of Griffith University, and to present the live and historical sensor data within a series of informational dashboards - which use a series of visualisation tools to present the data in a clear and meaningful way. The dashboards were required to be developed using Urban Institutes KX dashboard development tool. The information collected and presented will be used as part of a Structural Health Monitoring (SHM) strategy, which will help to monitor the performance of the N79 building, maintain the building and improve upon its lifespan.

The agile development framework was used to develop the core modules of the project, in order to perform several iterations for each part of the project and ensure each module functions as intended. Furthermore, feedback was periodically gathered regarding the user interfaces of the dashboards (which are often subjective) to continuously improve upon their designs in conjunction with the client needs.

The system developed for the project proved to be successful, resulting in a system that could successfully retrieve the N79 building accelerometer and strain gauge sensor data, and providing a total of 6 informational dashboards that were used to present the data. The final testing of the project modules was successful and approved by the client - indicating the project system functioned as intended, met the objectives for the project, and that end users could use the informational dashboards for their intended purpose (monitoring the N79 building).

ACKNOWLEDGEMENTS

I would like to start by thanking my academic supervior Dr. David Rowlands, who provided valuable support and advice throughout my professional practice and thesis. I would also like to thank Dr. Sanam Aghdamy and Dr. Dominic Ong for their support, feedback and advice throughout the project regarding the civil application of the work being undertaken.

I would also like to thank the staff of Urban Institute - including Simon Kaplan, Gabbie Lamberton, Josh Garvey, Tynan Mackay, Phil Schwake and my industry supervisor Adriano Marinho, who helped support me throughout the project and imparted valuable knowledge that I will be able to utilize within my future career.

I would like to thank both Dr. Andrew Busch and Dr. Andrew Rock for their support throughout my entire degree, both of whom have gone consistently above and beyond to support students within their learning – engaging with students throughout extra-curricular activities, investing their own free time to teach and advise students, and sharing their years of skills and experience within their respective fields.

Lastly and most importantly, I would like to thank my family for their incredible support throughout my degree. The support from my family enabled me to focus throughout my study, overcome the hardships I encountered and apply myself within my learning. Everything I have been able to learn and achieve throughout my studies is nothing more then a testament to their love and support, for which I am eternally grateful for.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

Tables

Figures

1 INTRODUCTION

This section provides an introduction to the project work that will be undertaken. Specifically, it introduces the project background, industry partner, project aims and objectives, project deliverables, project constraints, project scope and the structure of the report.

1.1 Background

1.1.1 Project Background

It is common for buildings to be regularly exposed to environmental and operational loads, including natural disasters such as earthquakes or severe weather - that lead to the deterioration of the building over its lifespan. As buildings become larger and more complex, retrieving reliable and timely information regarding the structural health of the building becomes increasingly important – as the economic, legal, performance and safety benefits become more apparent [1, 2].

The recently constructed N79 building at the Nathan campus of Griffith University is scheduled to be equipped with accelerometer, strain gauge, vibrating wire piezometer and earth pressure cell sensors - which measure the acceleration, strain, pore water pressure and total pressure at various locations of the building respectively. All earth pressure cells and vibrating wire piezometers have already been installed within the N79 building, and currently transmit their sensor data to the Griffith University PI System. However, the accelerometer and strain gauge sensors are still in the process of being installed. Photos of the sensors being installed within the N79 building can be seen in figures 1 and 2. A photo of the N79 building can be seen in figure 3.

This project will consist of collecting, processing and storing the data from these sensors, then developing an interface to present the data, which will be used as part of a Structural Health Monitoring (SHM) strategy to help with monitoring and maintaining the structural health of the N79 building. The information being collected and presented will also be used for research, marketing and educational purposes in the future.

Figure 1: An accelerometer being installed within the N79 building of Griffith University.

Figure 2: An earth pressure cell being installed within the N79 building of Griffith University.

Figure 3: The N79 building of Griffith University.

1.1.2 Company Background

The industry partner for this project is Urban Institute, who are an international company dedicated to providing tools and services for developing smart cities, with their Australian office being managed by Mr. Simon Kaplan (CEO). The work they conduct involves utilizing information and communication technologies within municipalities to achieve social, economic, organizational and environmental benefits. Urban Institute have recently been working with partners to mitigate climate change, upgrade infrastructure and improve the efficiency and sustainability of municipalities.

One of the tools Urban Institute have developed is a software platform called KX, which is a commercially available platform that provides services for storing, accessing, communicating and visualizing data. The KX platform includes a dashboard development tool that can be used to develop dashboards capable of presenting any data stored within the KX databases. Urban Institute have provided their KX platform for use within the project, which will be used to store a copy of the N79 building sensor data and develop informational dashboards to present the data.

1.2 Project Aims and Objectives

SHM is fast becoming a popular field of research within civil engineering, as indicated by the significant increase in the amount of research papers published within the field, with approxi-

mately 17,000 new papers being puplished between 2008 and 2017 [3].

As part of the vision that inspired the construction of the N79 building, Griffith University set out to install a network of sensors that could be used to monitor and analyze the buildings performance over time. As past applications of automated SHM strategies within buildings have been fairly limited, the system being developed will contribute valuable resources towards the research, marketing and education within the field.

The overarching aims of this project are to therefore provide a tool that can continuously retrieve the data being measured by the collection of sensors located at the N79 building, store a record of the data, and to provide an interface where the collected data can be easily accessed, interpreted, visualized and understood. Achieving these aims will provide a comprehensive record of the structural information associated with the N79 building, and provide tools and resources that can be utilized to monitor the building as part of an SHM strategy - which can subsequently be used to more effectively maintain the building and improve upon its lifespan. The accelerometer and strain gauge sensor data collected will need to be stored in a format which is easily compatible with the Griffith University PI System, as Griffith University are seeking to store and maintain a copy of the sensor data for their own records (the vibrating wire piezometer and earth pressure cell data being measured are already being sent to the Griffith University PI System). Lastly, this project aims to provide information and resources surrounding SHM that can be utilized within research, education and marketing in the future.

In order to achieve these aims, this project has set out the following objectives:

1. Develop software that continuously retrieves and processes the live sensor data from the N79 building accelerometers and strain gauges.

2. Develop software that continuously retrieves the live and historical sensor data for the N79 building vibrating wire piezometers and earth pressure cells from the Griffith University PI System.

3. Develop software that continuously communicates the processed sensor data with the KX server.

4. Develop software within the KX platform that continuously ingests and stores the pro-

cessed sensor data within KX databases, making the data available for use within the KX dashboards.

5. Develop a series of informational dashboards which utilize a collection of visualizations to present the processed sensor data in a user-friendly way, and are easy for users to interpret and understand.

1.2.1 Project Deliverables

In order to achieve the project objectives, the following deliverables are expected to be provided to the client at the end of the project:

1. A data retrieval software module, which continuously retrieves the N79 building accelerometer and strain gauge sensor data.

2. A data communication software module, which continuously communicates the N79 building sensor data with Urban Institutes KX server.

3. A PI System software module, which continuously retrieves the N79 building vibrating wire piezometer and earth pressure cell sensor data from the Griffith University PI System.

4. KX server software modules which ingest and store the N79 building sensor data within the KX databases.

5. A set of informational dashboards that use a series of visualisations to present the N79 building sensor data in a meaningful way, and are easy for users to interpret and understand.

1.2.2 Project Constraints

As certain hardware and software are expected to be utilized with this project, the work involved with this project will be conducted within the following constraints:

1. The sensors used for measuring the N79 building data will be provided and installed by Griffith University.

2. The computers that will be used to run the project software will be provided and installed by Griffith University.

3. The computers involved with the project will need to be connected to the Griffith University network, in order to avoid firewall issues with incoming and outgoing data.

4. A copy of all project sensor data will need to be stored within the KX databases, in order to maintain a record of the sensor data that is accessible by the KX dashboards.

5. The dashboards that will be designed for the project will be developed using the dashboard development tool of Urban Institutes KX platform.

6. Due to the coronavirus pandemic, which has lead to large scale lockdowns and social distancing - large sections of the project will need to be completed remotely. As a result, remote access and online communication tools will need to be deployed before sections of the project can be conducted, which will impact the project schedule.

1.3 Project Scope

The focus of this project is the collection, analysis, processing and communication of the raw data measured by the network of sensors that are set to be installed at the N79 building of Griffith University, as well as providing an interface that presents the processed data in a clear and meaningful way.

This project will utilize Griffith University and Urban Institute property - such as the N79 building hardware and facililties, the N79 building sensors, Urban Institutes KX software platform, the Griffith University PI System and Griffith University network. However, maintaining these components and ensuring they function as intended is outside the scope of this project.

The sensor data involved with the project will be stored into KX databases. Although Griffith University are seeking to store a copy of the N79 building accelerometer and strain gauge sensor data within their PI System (for their own records), the scope of this project is only to store the sensor data in a format that is easily compatible with their PI System, so that the data can be easily transferred at a later date. Configuring the PI System to ingest and store the accelerometer and strain gauge data is not within the scope of this project, and will most likely be completed

some time after this project is completed.

Lastly, the installation of the sensors that are yet to be installed within the N79 building is not the responsibility of this project.

1.4 Structure of the Report

This document is divided into the following 7 key chapters:

1. **Introduction:** A description of the project background, company background, project aims and objectives, project deliverables, project constraints and project scope.

2. **Literature Review:** A detailed review of the research areas that are relevant to the project work being conducted, including a review of Structural Health Monitoring (SHM), building sensors, wireless technologies and informational dashboards.

3. **Existing Project Infrastructure:** A description of the existing infrastructure that the project will need to integrate with or expand upon, including the current configuration of the N79 building, the Urban Institute KX platform and the Griffith University PI System.

4. **Design:** A detailed discussion of the design and development that was conducted in order to produce the overall system for the project, including the software modules and informational dashboards that were developed.

5. **Testing and Results:** A discussion of the testing that was conducted in order to validate the project work that was completed against the project objectives, including a detailed analysis of the testing results.

6. **Discussion:** A discussion and reflection of the methods that were used during the project and the challenges that emerged.

7. **Conclusion:** A summary of the outcomes that were produced during the project, a set of future work and recommendations that can be implemented to expand upon the project work, and a summary of the project work that was completed.

2 LITERATURE REVIEW

This section provides a literature review of the research areas being utilized within the project. Specifically, it provides a basic overview of Structural Health Monitoring (SHM), the working principles of the sensors used within the project, an overview of wireless technologies, and it explores the recommended design guidelines that should be followed when developing informational dashboards.

2.1 Structural Health Monitoring

Structural Health Monitoring (SHM) involves implementing a strategy to detect damage to a structure, through gathering and analyzing information pertaining to its structural health. Damage within SHM can be defined as changes introduced to the material or geometric properties of a structure which adversely impact its current or future performance, and is often identified through comparing different states of a structure - such as an undamaged state with the current state of the structure. Once damage is detected, corrective measures are often implemented in order to repair the structure and mitigate any risk of endagerment to human safety [1, 4, 5]. A summary of the steps involved with SHM can be seen in figure 4.

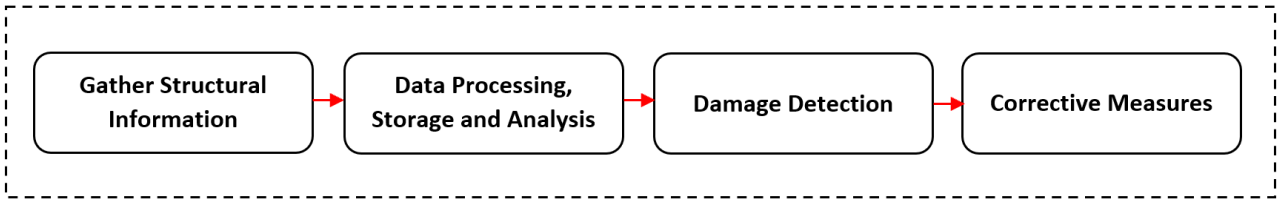


Figure 4: Workflow for a Structural Health Monitoring (SHM) strategy.

The application of SHM has primarily been used within aerospace, mechanical and civil structures, incorporating strategies that involve automated systems, manual inspection or a combination of both [1, 6].

As the effectiveness of SHM strategies are largely determined by their timely ability to detect damage, the use of manual inspection should be avoided wherever possible. Using manual inspection to monitor and analyze structural data is often very slow, and can include human error or subjective information that can adversely impact the effectiveness of any corrective measures

that are undertaken - leading to undesirable consequences such as large economic losses or the loss of human life. In particular, when buildings that involve manufacturing experience structural damage, any unnecessary downtime caused by ineffective SHM can lead to millions of dollars being lost in lost production [1, 7]. After the 1995 Kobe earthquake in Japan, where approximately 100,000 buildings were destroyed and another 283,000 suffered significant damage (see figure 5) - several buildings had to be manually evaluated for up to 2 years before they could be reoccupied, due to a lack of damage detection systems being available within the buildings, which could of provided useful and more timely information [6, 8, 9]. Hence the emerging strategy within SHM is to supplement or replace intermittent inspection with automated systems that are highly accessible, can continuously monitor infrastructure and can provide updates in real-time [4, 6].

Figure 5: Building damage after the 1995 Kobe earthquake in Japan [10].

Due to the historical costs of automated systems being very expensive, past applications of SHM strategies within buildings have been very limited, with most buildings only being manually inspected. The direct costs involved with purchasing, installing and maintaining wired sensors were a key contributing factor to these high system costs, which often require long wires and need to be installed within the walls or foundation of a building. Furthermore, when a wired system is damaged or requires scheduled maintenace, the process of inspecting and repairing

the system is often very expensive due to the difficulties reaccessing and rewiring large sections of the building [7, 11].

The application of wireless technologies and sensors within SHM has significantly helped to reduce these system costs over time, whilst also improving the benefits provided by the SHM strategies that implement them - which has greatly increased the application of automated SHM strategies within buildings during the last decade [7, 11].

Automated SHM strategies within buildings often involves placing a collection of sensors at strategic locations within the building, which can then continuously measure data to detect changes and damage regarding its structural health. Through utilizing technology to gather and analyse building data, systems can gather information faster, autonomously detect and communicate various points of interest, reduce the need for human collection and inspection of data, and more efficiently manage a wide range of resources - which are some of the core focuses behind the development and implementation of automated systems within SHM [1, 2, 7].

The use of visualisation tools within SHM are also highly recommended, as they allow users to interpret structural information much faster and can be used to highlight points of interest quickly. When inspecting the structural information of a building, the ability to interpret large volumes of data and quickly identify any damage within the structure can significantly improve the benefits provided by the associated corrective measures, as they can be implemented within a faster timeframe. A recommended technique for visualising structural information within SHM is the use of informational dashboards, which present data using a variety of visualisation tools [12, 13].

Current research within the field of SHM is focused on improving the transfer of data from sensors (as large volumes of data are typically transferred in real time), reducing the associated costs for installing and maintaining sensor networks, and improving data processing and analysis to help make actionable information more readily available [1, 6, 7].

2.2 Building Sensors

2.2.1 Accelerometers

Accelerometers are used to measure the acceleration of an object, which can be used to monitor the vibrations that act on structures which are exposed to dynamic loads. Different types of accelerometers use different techniques for measuring acceleration. One of the techniques employed by accelerometers is to attach a mass within an object, with the inner mass attached to springs along a given axis. When a force is applied to the object along the given axis, the inner mass is displaced, which can be measured to subsequently determine the acceleration of the object [14]. The working principle for the spring and mass technique can be seen in figure 6.

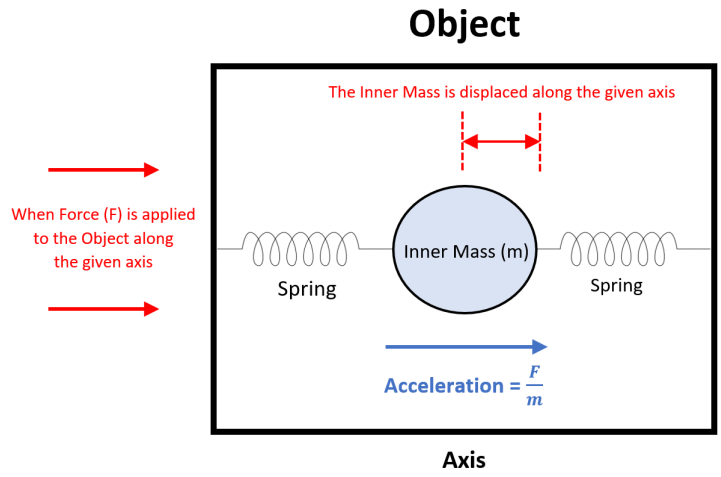


Figure 6: Conceptual diagram for an accelerometer using the mass and spring technique.

The displacement of the mass in combination with the tensile strength of the springs can be used to calculate the force (F) that was applied, which can be substituted alongside the weight of the inner mass (m) to calculate the resulting acceleration (a) along the given axis, using equation 1.

$$a = \frac{F}{m} \tag{1}$$

Using a combination of springs and mass (or an alternative technique for measuring acceleration) for each axis is a technique employed by triaxial accelerometers to measure the acceleration along all 3 axes (x, y and z axes).

2.2.2 Strain Gauge Sensors

Strain gauge sensors are used to measure the stress acting on an object. Most strain gauges are of the foil type, consisting of a piece of resistive foil and measuring the electrical resistance between 2 points. The piece of resistive foil used can come in a manner of different shapes and sizes to suit a varety of different applications [15]. A sample of the different shapes used within foil type strain gauges can be seen in figure 7.

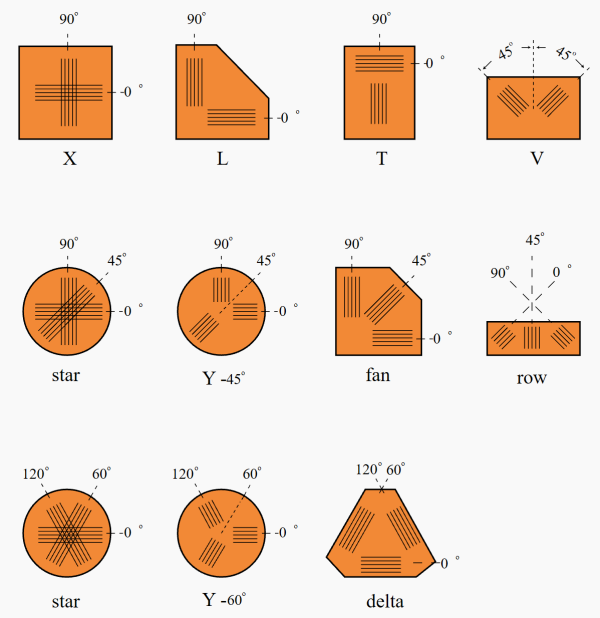


Figure 7: A sample of the shapes used within foil type strain gauges [15].

When the piece of foil is exposed to different stresses, the resistance between the 2 points being measured changes in a defined way. For example, as conductive metal is stretched, it will become skinnier and longer, increasing the electrical resistance between the two points. However if the conductive metal is compressed under force, it will broaden and shorten, reducing the eletrical resistance [16]. These properties can therefore be used to measure the stress acting on the object - see figure 8.

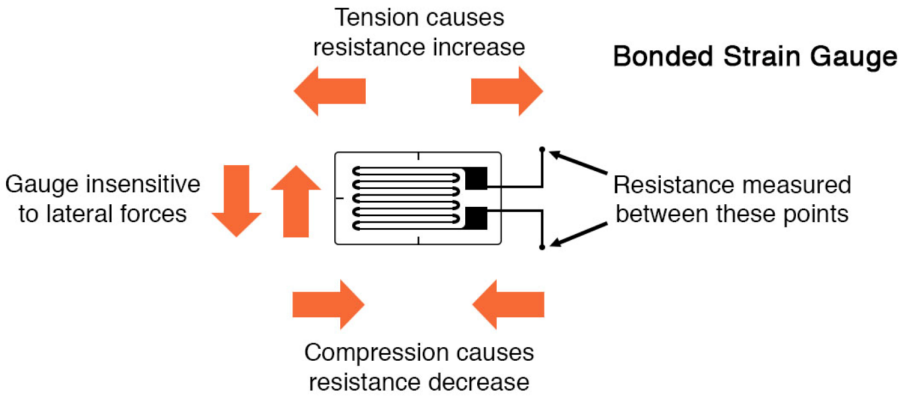


Figure 8: Operating principle for a foil type strain gauge [16].

2.2.3 Vibrating Wire Piezometer Sensors

A vibrating wire piezometer is used to measure pore water pressure within soil. By measuring the pore water pressure, the flow pattern of water within the soil can be determined, in addition to its load-bearing capacity [17].

The vibrating wire piezometer operates on the principle that when a high tension wire is plucked, it will vibrate at its resonate frequency. The vibrating wire piezometer is attached to a magnetic, high tension wire, with the piezometer acting as an anchor to keep tension on the wire. When there is a change in pressure, the tension on the wire is increased or decreased, which subsequently changes the resonate frequency of the wire when plucked. These frequencies are measured and converted into kilopascals (kPa) to determine the associated pore water pressures of the soil [17]. The structure of a vibrating wire piezometer can be seen in figure 9.

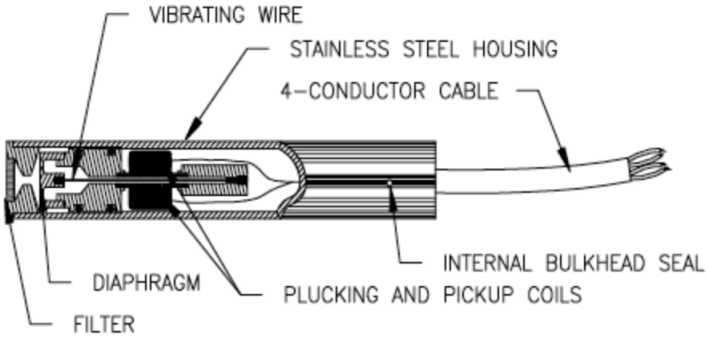


Figure 9: The structure of a vibrating wire piezometer [18].

2.2.4 Earth Pressure Cells

Earth pressure cells are used to measure total pressure, particulary within soil or rockfill structures. Earth pressure cells are constructed using 2 stainless steel plates, which are welded together around the periphery. The gap between the plates is filled with a deaired oil such as hydraulic fluid, which is supplied via a tube that is connected to the steel plates. When pressure is applied to the plates, the pressure of the internal fluid increases, which can be measured to determine the total pressure acting on the plates within the ground, see figure 10. As the temperature of the fluid within the cell can impact the pressures measured, earth pressure cells are often equipped with thermometers in order to adjust the pressure measured in relation to the temperature of the fluid [19].

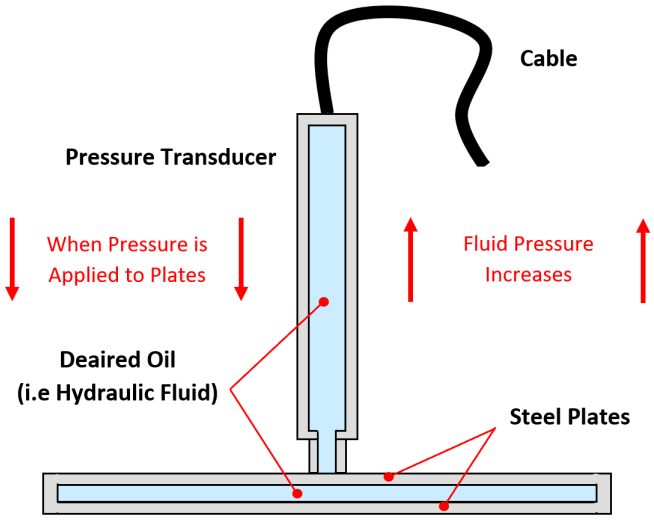


Figure 10: Conceptual diagram for an earth pressure cell.

2.3 Wireless Technology

Research has been invested into developing sensors that are wireless, which have greatly helped to reduce system costs often associated with wired sensors [7].

Wireless sensors can be organized into Wireless Sensor Networks (WSN) where the sensors communicate with each other, with the input data typically being communicated until it reaches a central repository where the data is stored. WSNs are often organized into various topologies, depending on the needs of the system [7]. A sample of common topologies used within WSNs

can be seen in figure 11.

Using a WSN offers the following key benefits:

1. Scalability. Any new sensor that is added to the WSN only has to be in range of communicating with one other sensor of the network.

2. Redundancy. Through providing several forms of input and communication, a damaged sensor(s) will typically not break the network. This becomes increasingly important in situations where there are higher risks of damage or the information being collected is highly important, such as the state of a building or an airplane - where damage can endanger human life or cause significant legal, economic and environmental consequences.

3. Low Cost. By removing all or the majority of the wires that have historically lead to expensive sensor networks, WSNs are often a cheaper alternative to wired systems.

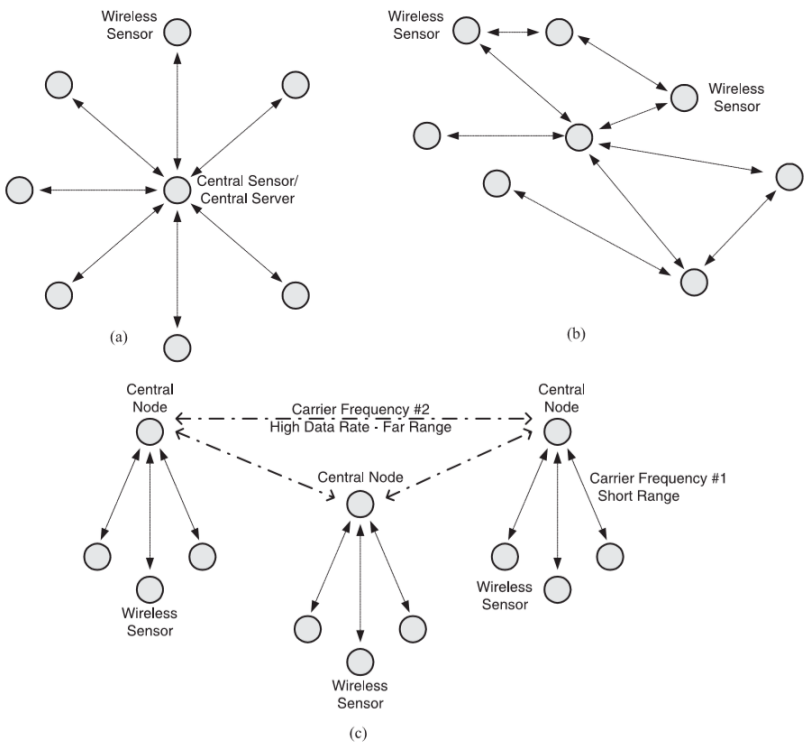


Figure 11: A Sample of the topologies used within Wireless Sensor Networks (WSN). (a) Star, (b) Peer-to-Peer and (c) Two-Tier network topologies [20].

An example of a WSN being applied to multiple floors of a building can be seen in figure 12.

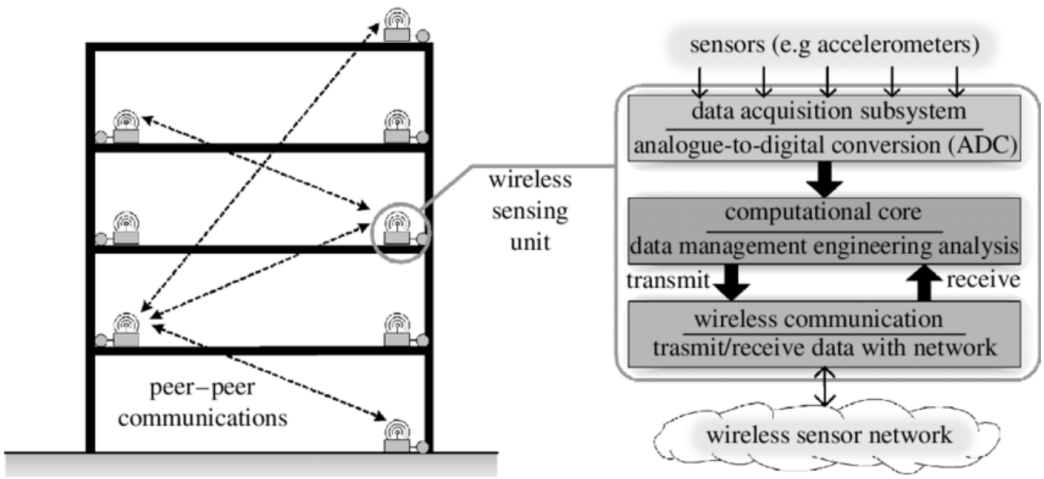


Figure 12: A building with multiple floors utilizing a Wireless Sensor Network (WSN) [7].

Studies testing wireless systems have provided results indicating the performance and accuracy of wireless sensors are capable of replacing traditional wired systems [7].

However, a common challenge that is introduced when moving to wireless technology is the need for a constant power supply. Significant research has been invested to try circumvent this issue - including developing technologies which provide more efficient power consumption, improving the electrical capacity of batteries and providing automated methods for recharging the batteries. Some examples include harvesting energy from the ambient environment, employing a mobile robot to periodically recharge the sensors, developing more power efficient algorithms, and developing systems where the wireless sensors only perform power-efficient tasks, outsourcing the less power-efficient tasks to a central repository or to a higher tier node that uses a wired connection [7, 21, 22].

Although wireless sensors offer a number of benefits, certain types of sensors still require wired cables to operate, such as earth pressure cells which require a tube to supply fluid. In addition, wireless sensors that are placed in highly inaccessible locations (such as underground or within solid structures), can be difficult to replace or maintain. In such cases, the decision to use wired sensors may become a more favourable alternative (or a requirement due to the technology) when deploying sensors within a system [7, 19].

2.4 Informational Dashboards

Dashboards are visualisation tools used to communicate large volumes of data at a glance. Through visualising data, most users find the information presented easier to comprehend and can interpret the information more quickly, allowing them to draw faster conclusions and enhancing their decision making [23, 24, 25].

2.4.1 Recommended Design Features

In order to maximize the information exchange between dashboards and end users, and make dashboards easy to use - dashboards should be designed so that they are clear, accurate, simple, meaningful and consistent [26, 27].

Keeping dashboards simple is critical for conveying information to a user. Specific examples include minimizing the amount of text, visualizing data whenever possible and restricting the amount of elements on a dashboard to a maximum of 5-7. As there is often a very limited amount of time to grab and hold a users attention (typically between 5-10 seconds), too much information or too many visualisations can quickly dissuade end users from using a dashboard - due to the user percieving the information as taking too long to interpret or feeling overwhelmed [26, 28]. Furthermore, the accuracy of the information presented within a dashboard is highly important - as inaccurate data can lead end users to assume there are further inaccuracies, resulting in a rapid loss of credibility and users quickly becoming disinterested in any further information presented [29].

To convey different levels of importance with an end user, more important information should be made more prominent by taking up more space within a dashboard, whilst less important information should consume less, and redundant information should not be displayed at all. Using comparison can also help to add context to a dashboard, such as comparing current and historical data, which can help end users distinguish between good and bad performance [26, 27].

Lastly, consistency is a key feature within dashboards. As users begin to use dashboards, they start to familiarize themselves with any patterns that emerge and build inherent expectations - such as the use of colour, the features used to convey various meaning, and the location of various information and features. Common examples include the color red to indicate poor

performance or danger, the color green to indicate good performance or results, and having the menu buttons located at the top of screen. Inconsistency within dashboards can therefore lead to difficulties navigating the dashboard or misinterpretion of the data, and should therefore be avoided [26, 27].

2.4.2 User Centred Design

Adopting an agile development methodology is a recommended approach for developing dashboards, as it offers opportunities to gather end user feedback and adapt outputs as client needs change. User Centred Design (UCD) is an agile development framework where end user needs, wants and limitations are given explicit consideration thoughout the entire design process. Collaboration with potential end users can provide valuable contextual information, such as how they interact with the dashboards and how they intend to use them - which can be incorprated within the designs to strengthen and shorten the information exchange between the dashboards and the end user, help end users navigate the dashboards more easily and support their decision making [30, 31].

3 EXISTING PROJECT INFRASTRUCTURE

This section provides an overview of the existing tools and infrastructure that this project will need to integrate with. Specifically, it will provide details of the sensors planned to be installed within the N79 building and how they will be configured, an overview of the KX platform software used within the project, and an overview of the Griffith University PI System software.

3.1 N79 Building

The N79 building located at the Nathan campus of Griffith University is scheduled to have sensors installed which collect structural information of the building. The N79 building can be seen in figure 3. The sensors that will be used within the N79 building include accelerometers, strain gauges, vibrating wire piezometers and earth pressure cell sensors. An overview of how the N79 building sensors will be configured can be seen in figure 13.

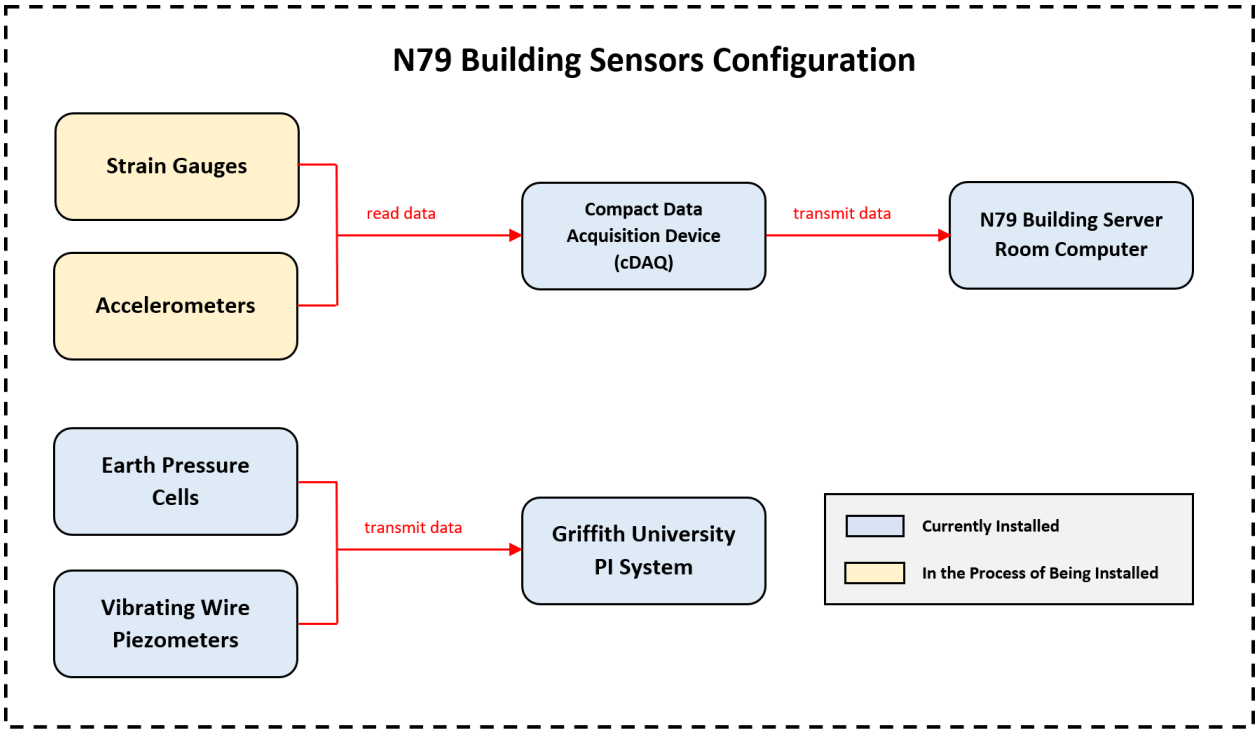


Figure 13: An overview of the N79 building sensor configuration.

The accelerometer and strain gauge sensors used within the N79 building are currently in the process of being installed, and will be connected to a series of National Instruments cDAQ-9171 Compact Data Acquisition devices (cDAQ), which are devices used for acquiring sensor

data. The cDAQ-9171 connects to a computer via a USB or ethernet connection, and can be communicated with to retrieve input data from any of the sensors that are connected to it [32]. The National Instruments cDAQ-9171 can be seen in figure 14.

Figure 14: A National Instruments Compact Data Acquisition device (cDAQ), Model: cDAQ-9171 [32].

The N79 building is set to have 12 Wilcoxon 731-207 Ultra Low Frequency Accelerometers installed, which have a nominal sensitivity of 10 V/g and a resonance frequency of 2.4 KHz [33]. 3 of the Wilcoxon Accelerometers have currently been installed on levels 1, 3 and 5 of the N79 building respectively, however there have been delays installing the remaining 9 accelerometers due to complications with the coronavirus pandemic. The Wilcoxon 731-207 Ultra Low Frequency Accelerometer can be seen figure 15.

Figure 15: A Wilcoxon Ultra Low Frequency Accelerometer, Model: 731-207 [33].

The locations of the accelerometers that are set to be installed within the N79 building can be

seen in figure 16.

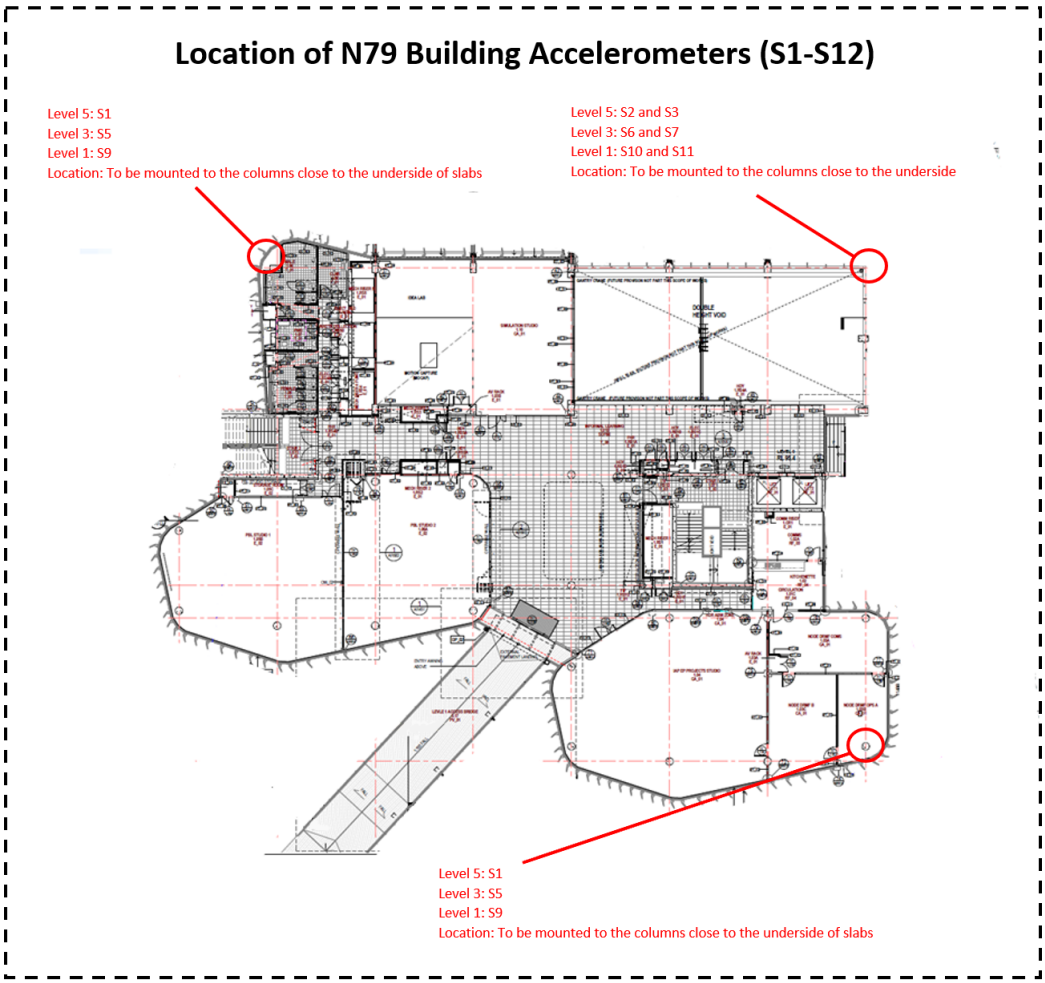


Figure 16: The locations of the N79 building accelerometers, as indicated by the red circles.

The N79 building is scheduled to have 15 PCB Piezotronics 740B02 Strain Gauges installed within the building, which have a sensitivity of 50 mV and an adjustable sampling rate [34]. However these sensors are yet to be installed due to complications with the coronavirus pandemic. The PCB Piezotronics 740B02 Strain Gauge can be seen in figure 17.

Figure 17: A PCB Piezotronics Strain Gauge, Model: 740B02 [34].

The locations of the strain gauges that are set to be installed within the N79 building can be seen in figure 18.

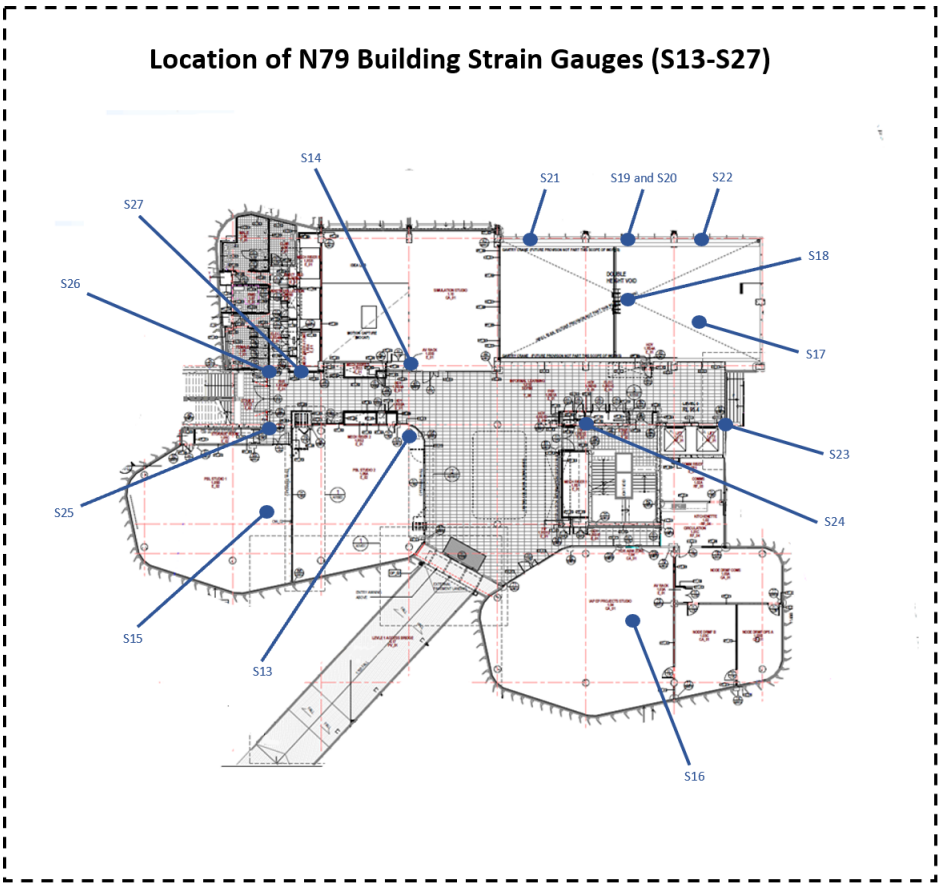


Figure 18: The locations of the N79 building strain gauges, as indicated by the blue circles.

All vibrating wire piezometers and earth pressure cells have been installed within the environment surrounding the N79 building, and currently transmit their data to the Griffith University PI System - from which a copy of the data will have to be retrieved during this project. All

vibrating wire piezometers and earth pressure cells sample data once every 15 minutes.

The vibrating wire piezometers that were installed within the N79 building included 2 Sisgeo PK20S Vibrating Wire Piezometers, which have a capacity limit of 350 kPa [17]. The Sisgeo PK20S Vibrating Wire Piezometer can be seen in figure 19.

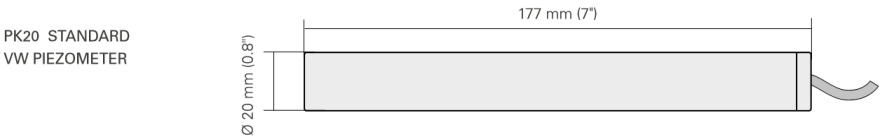


Figure 19: A Sisgeo Vibrating Wire Piezometer, Model: PK20S [17].

The locations of the Sisgeo PK20S Vibrating Wire Piezometers that were installed can be seen in figure 20.

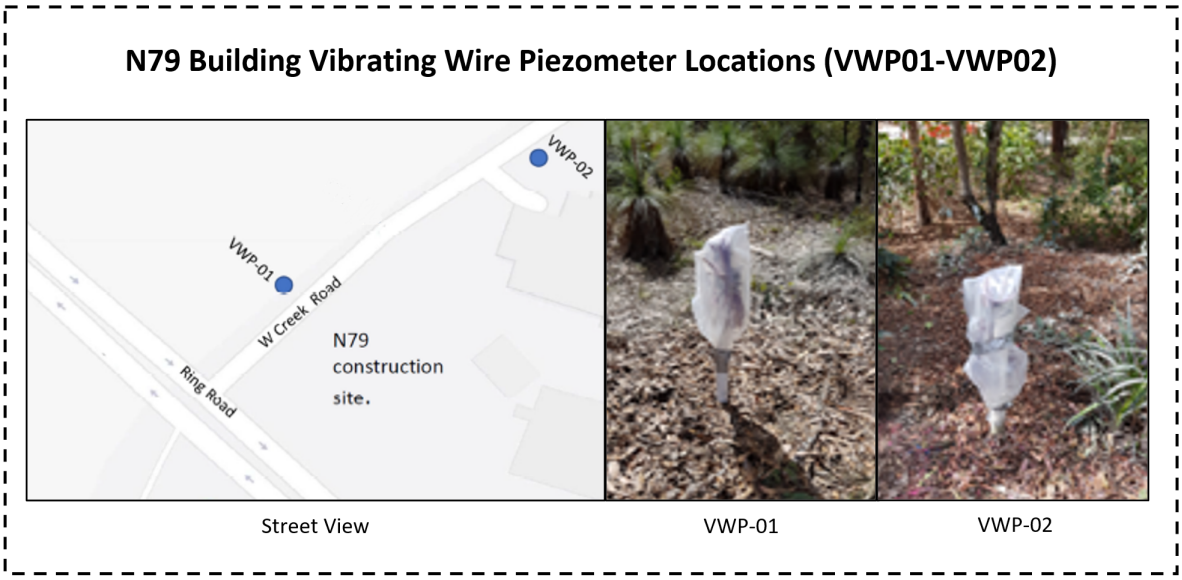


Figure 20: The locations of the N79 building vibrating wire piezometers, as indicated by the blue circles.

The earth pressure cells that were installed within the N79 building included 7 Sisgeo L143 Earth Pressure Cells, which have a capacity limit of 1000 kPa [19]. The Sisgeo L143 Earth Pressure Cell can be seen in figure 21.

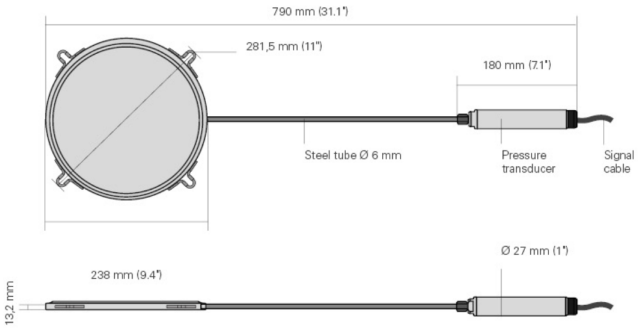


Figure 21: A Sisgeo Earth Pressure Cell, Model: L143 [19].

The locations of the Sisgeo L143 Earth Pressure Cells that were installed can be seen in figure 22.

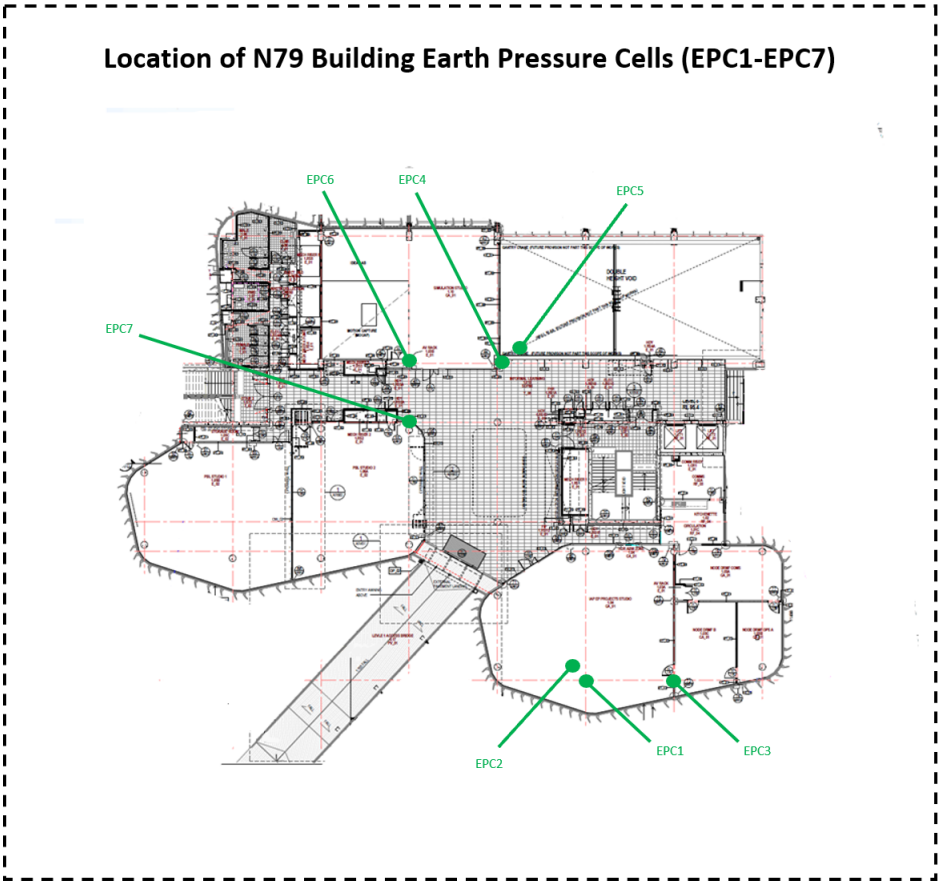


Figure 22: The locations of the N79 building earth pressure cells, as indicated by the green circles.

3.2 Urban Institute KX Platform

The KX platform is a commercially available software tool developed by Urban Institute, and provides tools for storing, accessing, communicating, analyzing and visualizing data. As part of the KX platform, a dashboard development tool is provided that can be used to develop dashboards capable of presenting any data stored within the KX databases.

The KX platform is developed primarily using the Q programming language (a scripting type language), and is regularly updated to meet Urban Institutes changing clients needs. The Q programming languages is very effective at working with data that is organized into tables, which are the primary data structures of the KX platform. The table data structures used by Q provide a format where data can be searched and returned from the KX databases quickly. Software processes can also be developed within the KX platform to perform a variety of tasks - including ingesting, processing, storing and communicating data.

3.3 Griffith University PI System

The Griffith University PI System is a software service provided by OSISoft, which provides a scalable architecture for storing and retrieving data [35].

The PI System consists of 3 core components, which are the PI Interface, PI Server and PI Client. The PI Interface provides services for organizing data into formats that are compatible with the PI Server, the PI Server stores the data within a series of data archives and the PI Client provides tools and services for organizing the data into formats that are easily ingested by users - such as spreadsheets or various charts and visualisations [35]. A summary of the PI System architecture can be seen in figure 23.

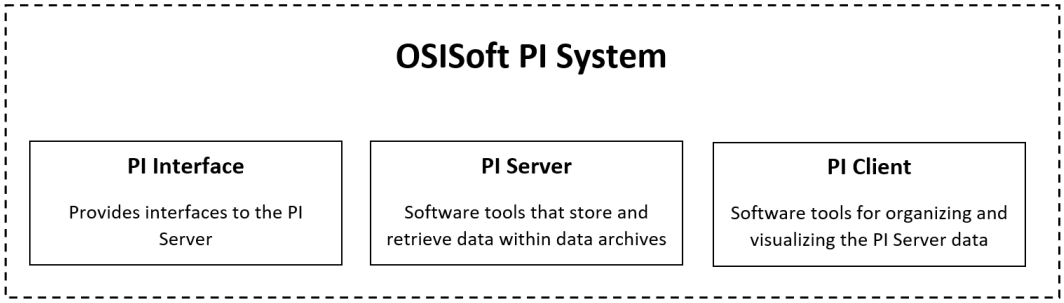


Figure 23: A summary of the OSISoft PI System architecture.

The PI System was designed to run on Windows operating systems, and is largely focused on storing large volumes of time-series and event-based data in real time. The system is capable of ingesting data from several sources simultaneously, supporting its capabilities for storing large groups of data [35].

OSISoft provide a PI Web API (Application Programming Interface), which is a RESTful (Representational State Tranfer) interface to the PI System, and allows data to be easily stored or retrieved from the PI Server. The PI Web API supports HTTP (HyperText Tranfer Protocol) requests for sending and retrieving data from the PI Server, and is optimized for searching and analysis - including the use of structured queries as URL (Uniform Resource Locator) parameters, which allows users to easily retrieve specific data sets [35].

4 DESIGN

This section provides details of the design process that was used to produce the required outcomes for the project and achieve its associated objectives. Specifically, it will provide details of the design methodology that was used for the project, details of how the project system was designed and subdivided into more manageable tasks, the design process used to develop the project software, and the design process used to develop the informational dashboards for the project.

4.1 Methodology

The agile development framework incorporates regular feedback and fast iterations that allow project milestones to be adjusted periodically to help ensure that client needs are being met [31]. As the work that was being undertaken within this project explored concepts that were fairly new, involved risks that could have impacted the project schedule, and involved the development of user interfaces that are often subjective - the agile software development approach was deemed highly suitable for the project work. The User Centred Design (UCD) development framework was also used to help ensure end user needs were being prioritized throughout the design process, which included regular meetings with civil engineers involved with the project to gather feedback (who are the intended end users for the project dashboards). Utilizing the agile development framework to subdivide the project work into sprints provided more flexibility within the project work, provided iterative feedback to regularly improve upon the project outcomes and helped reduce the risk of investing large amounts of work that didn't satisfy the client requirements.

4.2 System Design

When designing the system for the project, the system was subdivided into sprints according to the major tasks that were involved with the project.

A total of 7 sprints were allocated to the project, with 1 sprint being allocated to each of the core software modules for the project (the data retrieval, data communication, PI System and KX server software modules), and 3 sprints being allocated for the project dashboards. The

dashboards were allocated 3 sprints due to the subjective nature of user interfaces, which would allow several iterations to be completed in order to finalize designs that the clients were happy with.

The 7 sprints that were allocated for the project were as follows:

1. Develop the data retrieval software module to communicate with the N79 building cDAQ and continuously retrieve input data for the accelerometer and strain gauge sensors.

2. Develop the PI System software module to continuously retrieve the N79 building vibrating wire piezometer and earth pressure cell sensor data from the Griffith University PI System.

3. Develop the data communication software module to continuously process and communicate the N79 building sensor data with the Urban Insitute KX server.

4. Develop the KX server software module to continuously ingest sensor data and store the associated data within the KX databases.

5. Develop initial designs within Microsoft Excel for the informational dashboards, which present the sensor data of the N79 building using a series of visualisations.

6. Develop secondary designs within Microsoft Excel for the informational dashboards, which present the sensor data of the N79 building using a series of visualisations.

7. Develop final designs within the KX platform for the informational dashboards, which present the sensor data of the N79 building using a series of visualisations.

A summary of the project system design can be seen in figure 24.

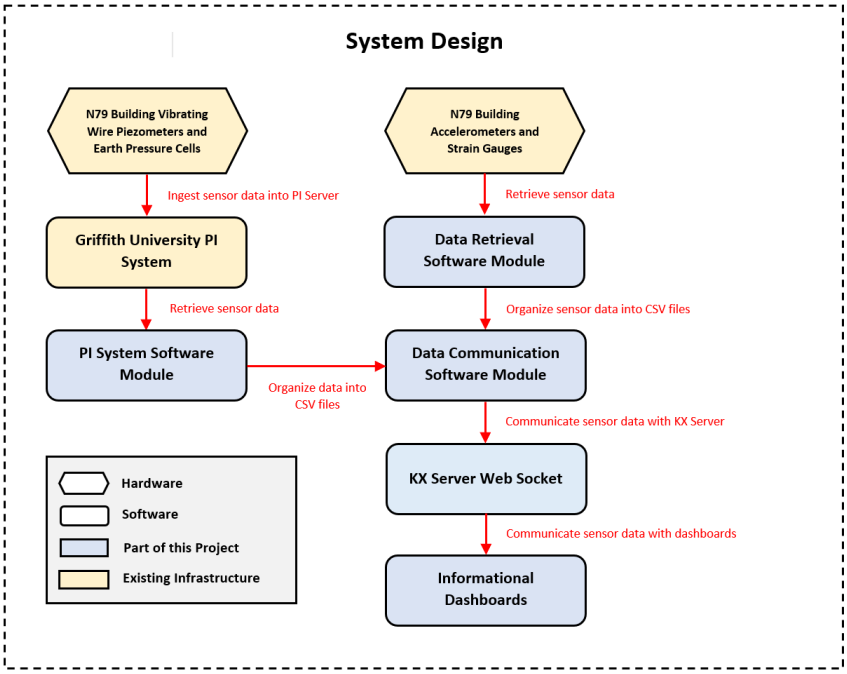


Figure 24: The project system design.

.

Each sprint was allocated a 1 week time frame, with the exception of the final dashboard designs, which were allocated 2 weeks due to the increased development associated with the KX platform. The original project schedule showing the initial time frames allocated to each sprint can be seen in Appendix A.

4.3 Software Design

When subdividing the software for the system, the system software was divided into the following 4 key modules:

1. A data retrieval software module continuously retrieves the accelerometer and strain gauge sensor data from the N79 building cDAQ.

2. A PI System software module continuously retrieves the N79 building vibrating wire piezometer and earth pressure cell sensor data from the Griffith University PI System.

3. A data communication software module continuously reads and processes sensor data stored within files and communicates the associated data with the KX server.

4. A KX server software module continuously ingests sensor data sent from the data communication software module, and stores the associated data within the KX databases, making the data available to the KX dashboards.

Dividing the system software into 4 modules decoupled the software, which is a recommended software engineering practice and also provided the following key advantages:

- The 4 software modules can run independantly.
- The data communication software module can process data that is retrieved from external sources (such as from the Griffith University PI System), provided that the data is organized into a correct format.
- The data retrieval software module keeps a local record of the sensor data if an external error occurs (such as if the KX server goes down), which can still be communicated by the data communication module at a later date once the error has been resolved.
- New software modules can be developed in the future to communicate the N79 building sensor data with new destinations (such as with the Griffith University PI System), without having to compromise the existing software modules.
- The KX server software can ingest and store sensor data from new sources, provided that the data is organized into a correct format.

When exploring what programming language was going to be suitable for developing the software involved with the project, the programming language would need to be capable of performing the following tasks:

1. Communicating with the cDAQ that houses the N79 building accelerometers and strain gauges.
2. Generating timestamps for the sensor data with microsecond precision (due to the accelerometer and strain gauge sensors sampling data at rates between 1000-2500 Hz).
3. Providing good string manipulation for generating unique file names, converting timestamps into strings and organizing data.

4. Providing good file manipulation for storing sensor data within files, deleting files and moving the associated files between a directory where the data is stored.

Upon conducting research to explore what programming language would be appropriate for the project software, only 2 programming languages provided software libraries that contained methods for easily communicating with the cDAQ - which where C and Python. When deciding which of the 2 programming languages would be more appropriate for the project software, they were compared against their capabilities for performing the required programming tasks in table 1.

As the Python programming language provided software libraries that would be easier to use for writing the project software (when compared to the C programming language), it was determined to be the best programming language for developing the software modules of the project.

As Griffith University are intending to store a copy of the N79 building accelerometer and strain gauge data within their PI System for their own records, part of this project involved storing the accelerometer and strain gauge data in a format that was easily compatible with the PI System. Although developing the software to store the sensor data within the PI System was not within the scope of this project, by storing the data in a format that was highly compatible with the system - it allows the associated sensor data to be stored within the system more easily at a later date. As Mr. Brian Hobby is the Technical Lead for IT and Engineering at Griffith University, and has extensive experience dealing with the Griffith University PI System - a meeting was scheduled with Brian to discuss what methods were available for transferring large volumes of sensor data with the PI System. Brian mentioned that Comma Seperated Value (CSV) files were highly compatible with the system, and that they could be ingested into the system using the File Transfer Protocol (FTP) or processed using software - which would therefore allow a copy of the sensor data to be easily transferred into the system. Furthermore, as CSV files are often used to store large volumes of data - the CSV file format was determined to be an appropriate choice for storing the sensor data of the project.

Required Task	Python	C
Communicating with the N79 building cDAQ to continuously retrieve sensor data for the accelerometers and strain gauges	The Python Nidaqmx software library provides methods for easily interacting with the cDAQ and retrieving data from any sensors that are connected to it	The C Nidaqmx software library provides methods for easily interacting with the cDAQ and retrieving data from any sensors that are connected to it
Generating timestamps for sensor data with microsecond precision	The Python datetime standard library can be used to easily generate timestamps with microsecond precision	The C time.h standard library can only generate timestamps with second precision
Providing good string manipulation for generating unique file names, converting timestamps into strings and organizing data	Python automatically adjusts the length of strings, allowing for easy string concatenation. Pythons datetime library can easily convert timestamps into strings	C strings have a fixed length, making string concatenation difficult. C can easily convert timestamps into strings, however the timestamps do not have microsecond precision
Providing good file manipulation for storing sensor data within files, deleting files and moving the associated files between a directory where the data is stored	Pythons standard os and system libraries provide methods to easily delete files and move files between directories	C can easily delete files. However C software libraries are limited in their ability to move files between directories, and may require system calls
Reading and writing sensor data to CSV files	Pythons standard csv library provides methods to easily read and write data from CSV files, including advanced features such as automatic header detection	C writes and reads CSV files as normal text files, requiring manual parsing and making it a lot harder to read and write CSV files

Table 1: Comparison of the programming languages that were considered for developing the project software.

When exploring how to store the N79 building sensor data within the KX server - a meeting with Urban Institute staff was conducted to discuss what methods were available for ingesting the data. Urban Institute staff recommended the use of a web socket, which are highly scalable and can handle large volumes of data. A web socket was therefore configured as part of the software processes developed within the KX server, which listens to a specific port of the KX server address and ingests any sensor data that it recieves.

4.3.1 Data Retrieval Software Module

The data retrieval software module is responsible for configuring the N79 building cDAQ, retrieving the accelerometer and strain gauge sensor data from the N79 building, generating timestamps for the data, deriving the raw outputs into their associated engineering units and organizing the data into CSV files that are compatible with the data communication software module.

The data retrieval software module starts by configuring a task that will be used to connect to the cDAQ of the server room, identifying which of the currently connected cDAQ(s) will be communicated with, and which channel(s) from each cDAQ will be communicated with to retrieve sensor data. The software is configured to create a map between which channels(s) of the cDAQ(s) are connected to which sensors (both sensor type, and a unique id which allows for each sensor within the N79 building to be uniquely identified). The unique id used for each sensor is important for uniquely identifying which measurements came from which sensor when storing the sensor data.

Once the cDAQ is configured, the data retrieval software module will continuously retrieve data from the channels that were configured within the tasks, until either an error occurs or the software is terminated by a user. As the sensor data retrieved from the cDAQ does not provide timestamps for the data, the timestamps for the sensor data are generated within the software. Furthermore, as the accelerometers and strain gauges sample data at rates between 1000-2500 Hz, the timestamps that are generated provide microsecond precision. Each time the data retrieval software module retrieves data from the cDAQ, the sensor data is processed into its associated engineering units (i.e converting voltage to acceleration), and the data is organized into a CSV file. Each CSV file is given a unique name using a microsecond timestamp, in order to ensure files with the same name are not stored within the spool directory. Once a set

limit of sensor data has been stored within each CSV file, it is moved into the spool directory monitored by the data communication module (which reads and communicates the data with the KX server), and a new CSV file is created to continue storing sensor data until the data limit is reached again. A summary of the design for the data retrieval software module can be seen in figure 25.

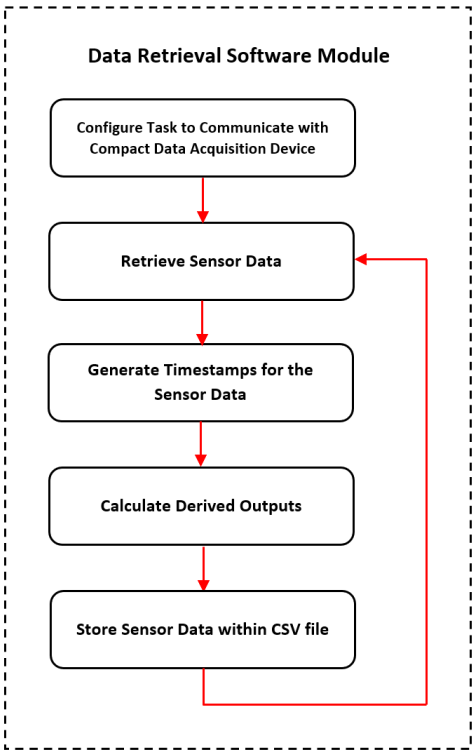


Figure 25: Data retrieval software module design.

The source code for the data retrieval software module can be seen in appendix B.

4.3.2 PI System Software Module

The PI System software module is responsible for retrieving the vibrating wire piezometer and earth pressure cell sensor data from the Griffith University PI System, and organizing the data into CSV files that are compatible with the data communication software module.

The CSV files used to store the sensor data are stored within a spool directory, which is monitored by the data communication software module. A summary of the design for the PI System software module can be seen in figure 26.

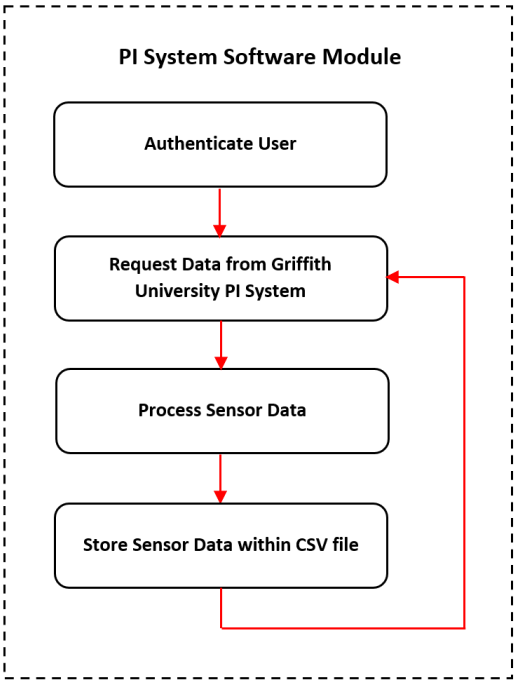


Figure 26: PI System software module design.

Due to maintenance being conducted on the Griffith University PI System during the project, which contributed to long periods where the PI System was unavailable - the PI System software module was not completed during the project. As the structure of the data retrieved from the PI System will heavily influence the sections of software that need to be developed, particularly with regards to how the data is processed - it was decided by the client that the PI System software module will be completed some time after the project is completed instead.

4.3.3 Data Communication Software Module

The data communication software module is responsible for processing sensor data stored within CSV files, organizing the sensor data into packets and communicating the data with the KX server.

The data communication software module continuously checks the spool directory configured within the software for CSV files. The software will continue to monitor the spool directory indefinitely, even if the directory is empty. Each time a CSV file is found, the data communication module reads each row of sensor data contained within the file, and organizes the sensor data into network packets - which are communicated with a web socket located on the KX server.

The web socket of the KX server where data is sent is dedicated for storing the N79 building sensor data within the KX databases. Each network packet that is created by the data communication module is configured to send a set amount of sensor data before a new network packet is created - which helps to ensure the network packets maintain a reasonable size.

Once all the sensor data within a CSV file has been communicated with the KX server, the associated file is deleted. An extra level of redundancy was added to the data communication software module, where CSV files are only deleted once all the associated data has been communicated (i.e the network packets have been sent). This ensures that if an unexpected error occurs whilst a network packet is being created, the associated CSV file will not be deleted - and can be recommunicated once the error is resolved to help ensure no loss of data occurs. A summary of the design for the data communication software module can be seen in figure 27.

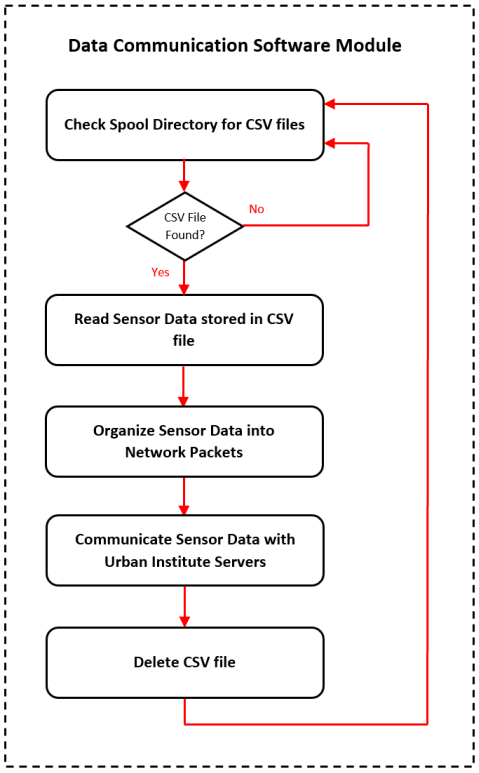


Figure 27: Data communication software module design.

The source code for the data communication software module can be seen in appendix C.

4.3.4 KX Server Software Module

As part of this project, software processes were developed within the KX platform for ingesting and storing the N79 building sensor data. The software processes are divided into a Ticker Plant (TP), Real-Time Engine (RTE), Real-Time Database (RDB) and Historical Database (HDB). An overview of the software processes that were developed within the KX platform can be seen in figure 28.

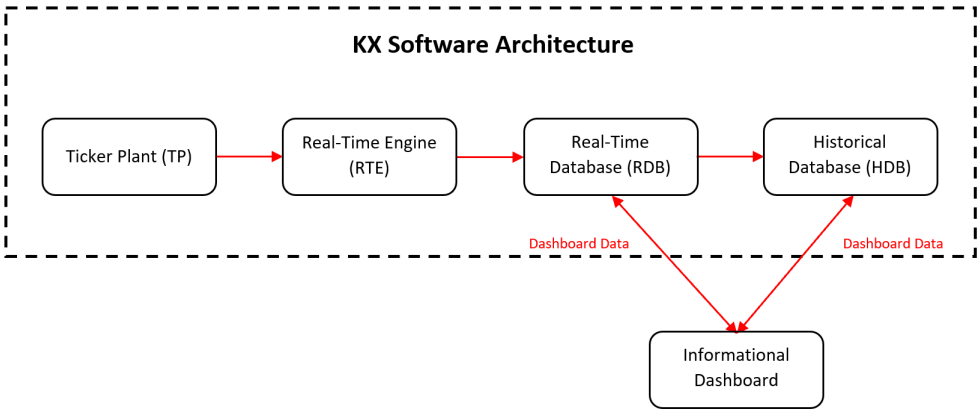


Figure 28: An overview of the software processes used for storing sensor data within KX databases.

The RTE acts as the web socket for storing all the N79 building sensor data, which listens for data sent to a specific port of the KX server address. When the RTE recieves data, it organizes the data into tables before storing it into the RDB, which stores the live data (data for the current day). The tables used help to query the databases when retrieving data, which is a feature utilized extensively when retrieving data for the KX dashboards.

The TP handles all the timing for the system - such as the End of Day (EOD) process, and triggering the RTE to read data from the port it listens to. The EOD process is triggered by the TP at the end of each day, which triggers all existing data stored within the RDB (the current days data) to be transferred to the HDB (which holds a record of all data stored before the present day). Once the data stored within the RDB has been transferred to the HDB, the RDB is cleared in preparation for the next days data.

The dashboards developed using the KX dashboard development tool can retrieve and present data from both the RDB (live data) and HDB (historical data).

4.4 Dashboard Design

As the User Centred Design (UCD) development framework was adopted to develop the dashboards for the project, regular meetings were conducted with Dr. Sanam Aghdamy and Dr. Dominic Ong, who are key clients for the project, and are senior professors at Griffith University who teach and conduct research in the field of civil engineering. As Sanam and Dominic are representative of the target end users for the dashboards (civil engineering staff who monitor buildings) - regularly collecting their feedback helped to continuously improve upon the dashboard designs, strengthen the information exchange between the dashboards and end users, and helped to ensure they were satisfied with the final designs that were being developed.

When designing each of the dashboards, recommended guildelines for keeping the dashboards clear, accurate, simple, meaningful and consistent were followed - such as minimizing the number of visualizations, using size to emphasize the most important visualizations, and maintaining a consistent interface layout and colour scheme between the designs.

Microsoft Excel was used to produce initial and secondary designs for the dashboards, in order to develop designs that could be used to gather feedback quickly - mitigating the risk of investing large amounts of work developing dashboards within the KX platform that didn't meet the clients needs.

The final designs for the dashboards were developed using the KX dashboard development tool. The dashboard development tool works by adding various 'elements' into a dashboard (such as graphs, charts, text and user inputs), then supplying data to each element through writing structured queries - which return data from the databases stored within the KX platform. The queries utilize user input elements as parameters, which allows users to influence which data is returned.

When designing the dashboards, each graph was designed to record time along the X axis (the date and time a sensor measurement was recorded), whilst the Y axis is used for presenting the associated value that was measured by the sensor.

When using the KX dashboard development tool, a number of key constraints with the tool were discovered, including:

- Each element of a dashboard can only present up to 10,000 rows of data, as exceeding this

limit will result in only a portion of the data being returned.

- The radio list element (a list of checkboxes) was not returning the expected data when used as a parameter for querying the database.
- The line chart element of the dashboards is only able to plot a maximum of 3 lines per chart, as exceeding this limit results in the data not being displayed correctly.

These constraints heavily influenced the design processes that were used when developing the dashboards for each of the sensors, requiring the original designs to be progressively changed in order to adapt to these constraints. The design changes that were used to accomodate these constraints will be discussed during the following sections.

4.4.1 Accelerometer and Strain Gauge Dashboards

Iteration 1: Initial Design

The original intention for the accelerometer and strain gauge dashboards was to include a feature that would allow users to view sensor data measured between a user supplied time frame, with seperate graphs used for live and historical data. The graphs for live and historial data were designed to appear side by side, allowing users to perform side by side comparisons for different sensors, and between different time periods as per the date range provided by the user.

The accelerometer and strain gauge dashboards that were designed during the first iteration can be seen in figures 29 and 30 respectively.

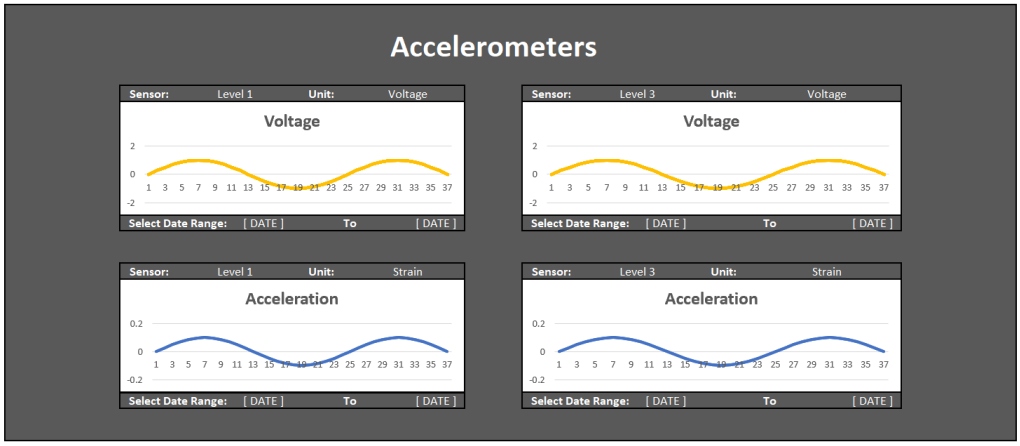


Figure 29: Initial design for the accelerometer dashboard.

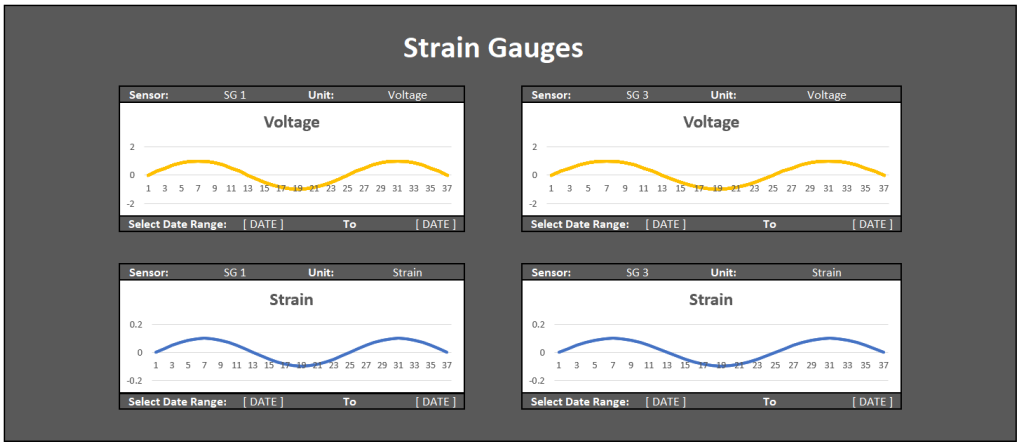


Figure 30: Initial design for the strain gauge dashboard.

When discussing these designs with the client, they provided the following feedback:

- The client wanted the sensors to be displayed within the same graph - which would allow sensors to be compared within the same graph (as opposed to side by side comparison).
- The client liked the idea of users being able to select which sensors they wanted to view within the graph, but wanted to use radio buttons for selecting which sensors to view within the graphs.
- The client wanted to include a photo of the N79 building within the accelerometer and strain gauge dashboards.

Iteration 2: Secondary Design

As the accelerometer and strain gauge sensors of the N79 building sample data at frequencies between 200-2500 Hz, each of the dashboard elements would have only been able to display approximately 4-50 seconds worth of the associated sensor data (as per the 10,000 rows of data constraint). The user supplied time frame could have also lead to misinterpretation of the data, such as a user providing a time frame covering a one day period, but only 4-50 seconds worth of the data being presented - which they could potentially misinterpret as being representative for the one day period they supplied.

When exploring alternative solutions that could be used to address this issue, downsampling the data was considered - which is a technique commonly utilized within mainstrain dashboards

when presenting large volumes of data [36]. However, as the sensor data for the accelerometers and strain gauges were sampling at such high frequencies (approximately 86,400,000 - 216,000,000 points of data per sensor, per day), the loss of data that would occur as a result of downsampling the data into 10,000 points was deemed too great.

Instead, the designs for the strain gauge and accelerometer dashboards were adjusted to include a data filter (as opposed to a date range), which filters data that is not within the bounds of a user supplied range. When analyzing sensor data for strain gauges and accelerometers, typically there are large volumes of noninteresting data - such as the data measured during long periods of low strain or low acceleration. Therefore, the ability to filter out the noninteresting data provided the following 2 key benefits:

1. Users are able to easily isolate interesting values for further analysis, such as values indicative of high strain or high acceleration.

2. Users are able to view periods of data that are longer then 4-50 seconds, due to most of the noninteresting data not being returned from the database, with the length of the time for which data is presented increasing relative to the range of data that is filtered out.

Using the data filter, users supply upper and lower thresholds for the data alongside a time and date. Only the sensor data preceding the given time and date, and which occurs between the user provided range of values are returned, returning data for an extended time period until a total of 10,000 rows is reached.

The client initially wanted to use a series of interactive checkboxes within the dashboards, providing the feature that would allow users to select which sensors they would like to view - with one graph for live data and another for historical data. However, due to the constraints within the development tool regarding the radio list element not returning data for the sensors selected, and the line charts not displaying correctly when more than 3 lines are plotted on the same chart - the original design for this feature could not be implemented, and had to be adjusted in order to accomodate these constraints. In order to adapt the dashboard designs, a series of drop down lists were implemented instead. As only one input can be selected from each drop down list, drop down lists were used to enforce a maximum of 3 sensors being selected - preventing the risk of the display issue associated with the line charts. Furthermore, as the drop down lists

were returning the expected data when used to select sensors, they were utilized to bypass the constraint associated with the radio checkboxes.

As the client wanted the sensors to be viewed within the same graphs, with one for live and another for historical data - the graphs were placed on top of one another, with the width of the graphs being expanded to the length of the dashboards. A further reason for having the 2 graphs appear on top of one another (as opposed to side by side), and expanding the width of the graphs was that it made the data within the graphs easier to view, due to providing a relatively longer spread of the data along the horizontal axis.

Lastly, an image of the N79 building was added to the strain gauge and accelerometer designs, with the goal of helping users more easily understand the context of the data that is presented, such as where the data is being recorded from.

The accelerometer and strain gauge dashboards that were designed for the second iteration can be seen in figures 31 and 32 respectively.

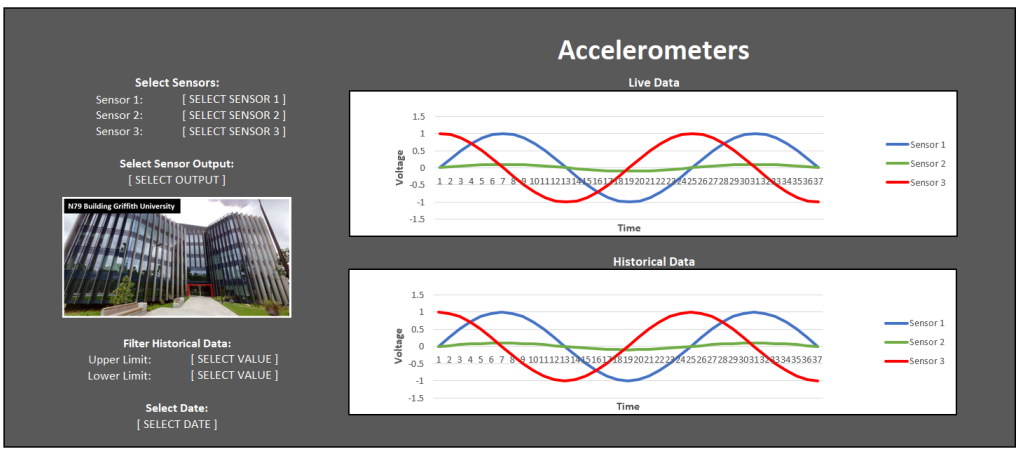


Figure 31: Secondary design for the accelerometer dashboard.

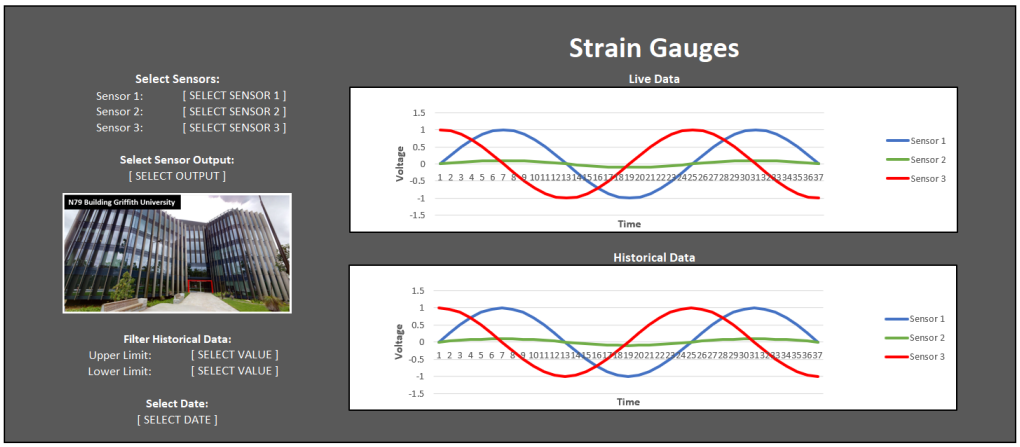


Figure 32: Secondary design for the strain gauge dashboard.

When presenting these designs to the client, they were happy with the designs and the features that were included. One piece of feedback they offered for the secondary designs was to include labels within the accelerometer dashboard which indicate the locations of the currently installed accelerometers (levels 1, 3 and 5 respectively).

Iteration 3: Final Design

The final design for the accelerometer dashboard was adjusted to include labels indicating the floors of the N79 building where each of the currently installed accelerometers are located.

The final design for the strain gauge dashboard was implemented according to its associated secondary design, which the clients were happy with.

The final designs for the accelerometer and strain gauge dashboards that were developed using the KX dashboard development tool can be seen in figures 33 and 34 respectively.

Figure 33: Final design for the accelerometer dashboard.

Figure 34: Final design for the strain gauge dashboard.

4.4.2 Vibrating Wire Piezometer and Earth Pressure Cell Dashboards

Iteration 1: Initial Design

The original intention for the vibrating wire piezometer and earth pressure cell dashboards was to be able to view and compare data between the associated sensors. User input was added to allow users to select which sensors they wanted to view, which units they wanted to view, and for which time and date they wanted to view sensor data (until 10,000 rows of data was reached, as per the 10,000 rows of data constraint).

The initial designs that were developed for the vibrating wire piezometer and earth pressure cell dashboards can be seen in figures 35 and 36 respectively.

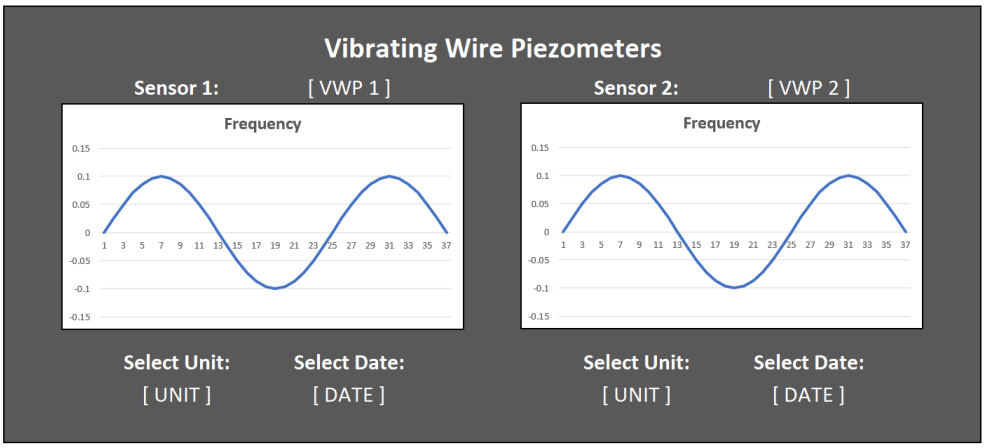


Figure 35: Initial design for the vibrating wire piezometer dashboard.

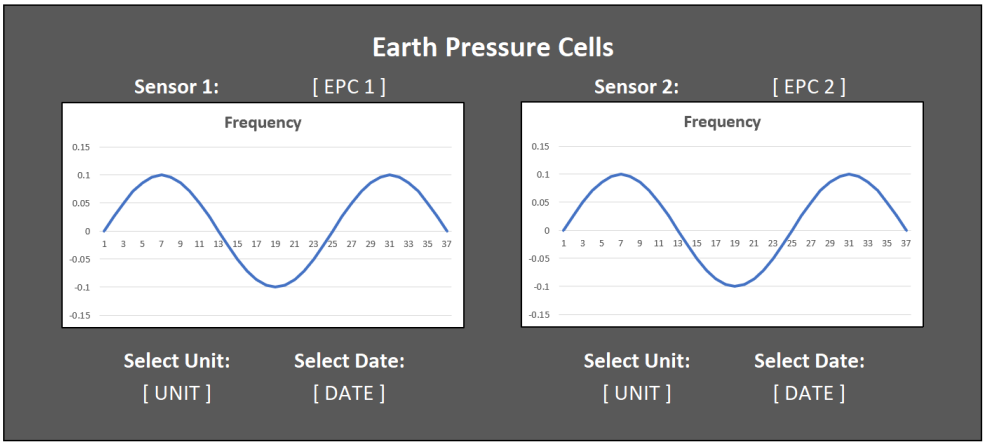


Figure 36: Initial design for the earth pressure cell dashboard.

When presenting the initial designs for the vibrating wire piezometer and earth pressure cell dashboards to the client, they provided the following feedback:

- The client wanted the sensors to be displayed within the same graph, with one graph for live data and another for historical data. The client wanted users to be able to select which sensors to view using radio checklist buttons - which would allow sensors to be compared using the same graph (as opposed to side by side comparison).

- The client wanted 2 designs for each of the vibrating wire piezometer and earth pressure cell dashboards. The first design would include line charts to present the sensor data, whereas the second design would present the sensor data using tables. Both designs would include visualisations for live and historical data.

- The client did not require users to be able to select units for the vibrating wire piezometer and earth pressure cell data, stating that displaying time and frequency would be the priority for the associated sensors.

- The client wanted to include an image of the N79 building within each of the vibrating wire piezometer and earth pressure cell dashboards.

- The client wanted to include a date range feature within each of the vibrating wire piezometer and earth pressure cell dashboards (as opposed to just a single time and date).

When discussing the constraints that had been discovered within the KX dashboard development tool with the client, the client mentioned that the vibrating wire piezometers and earth pressure cells only sample data once every 15 minutes. With the sensors retrieving data at such a low sampling rate, using a date range feature would allow periods of up to 104 days to be viewable within their respective dashboards (based on the constraint of 10,000 rows of data), which the client was eager to include within the designs.

The feedback from the client regarding the initial designs for the vibrating wire piezometer and earth pressure cell dashboards was used to develop secondary designs for the respective dashboards.

Iteration 2: Secondary Design

Based on the feedback from the initial designs, the secondary designs for the vibrating wire piezometer and earth pressure cell dashboards were expanded to include a total of 4 dashboards designs - 2 which use line charts to present sensor data, and another 2 which use tables to present the sensor data.

The desired goals for using the line charts were to visualise and isolate anomalies within the data quickly, whereas the tables are used to inspect specific values more easily. Both designs

include live and historical data, with the end user being able to select which sensor(s) they want to view within each of the dashboards.

The user input that allowed users to select which engineering unit was presented for the vibrating wire piezometers and earth pressure cells was removed, with only the time and frequency being returned for the associated sensors instead.

A date range feature was included within the secondary designs, allowing users to provide a time frame for which they want to view data, and only returning sensor data measured between the time frame provided. The date range feature is capable of showing vibrating wire piezometer and earth pressure cell data for periods of up to 104 days, after which only a portion of the data will be returned (as per the 10,000 rows of data constraint).

An image of the N79 building was also added to the vibrating wire piezometer and earth pressure cell dashboards, in order to provide more context regarding the sensor data being retrieved.

The client mentioned that they wanted to use radio checklist buttons for allowing users to select which sensors they wanted to view within the line charts. However the feature could not be implemented as intended due to the constraints within the KX dashboard development tool associated with the radio checklist elements not returning data for the inputs selected, and line graphs not presenting data correctly when more then 3 lines were plotted within the same chart. As drop down lists could be used to enforce a maximum of 3 sensors being selected - they prevented the risk of the display issue occuring with the line charts. Furthermore, as the drop down lists were returning the expected data for the sensors that were selected, they were used to bypass the constraint associated with the radio checkboxes not returning data. Therefore in order to accomodate the associated constraints, drop down lists were used as an alternative solution for allowing users to select which sensor(s) they would like to view within the line charts of the dashboards.

The secondary designs for the vibrating wire piezometer and earth pressure cell dashboards that use line charts to present the data can be seen in figures 37 and 38 respectively.

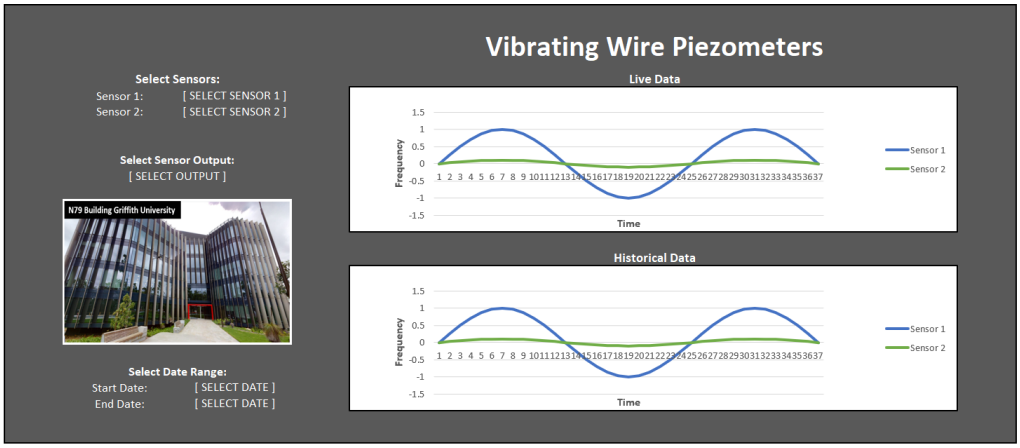


Figure 37: Secondary design for the vibrating wire piezometer dashboard using line charts.

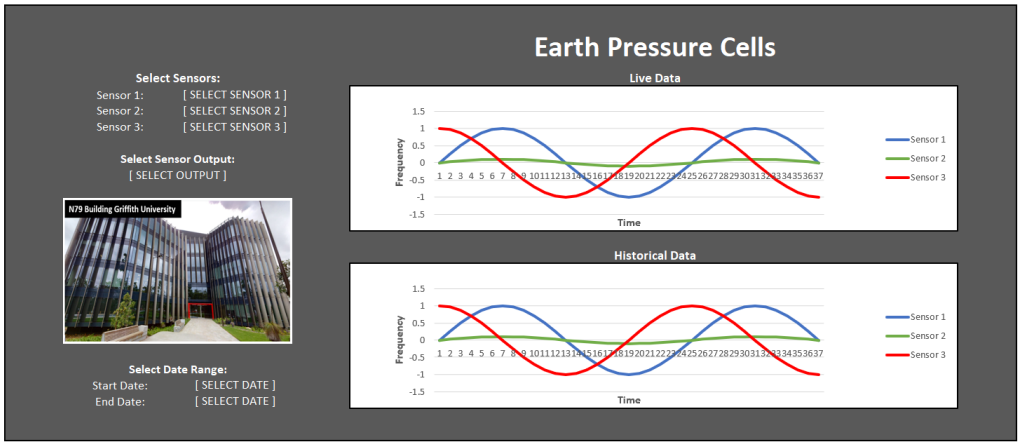


Figure 38: Secondary design for the earth pressure cell dashboard using line charts.

The secondary designs for the vibrating wire piezometer and earth pressure cell dashboards that use tables to present the data can be seen in figures 39 and 40 respectively.

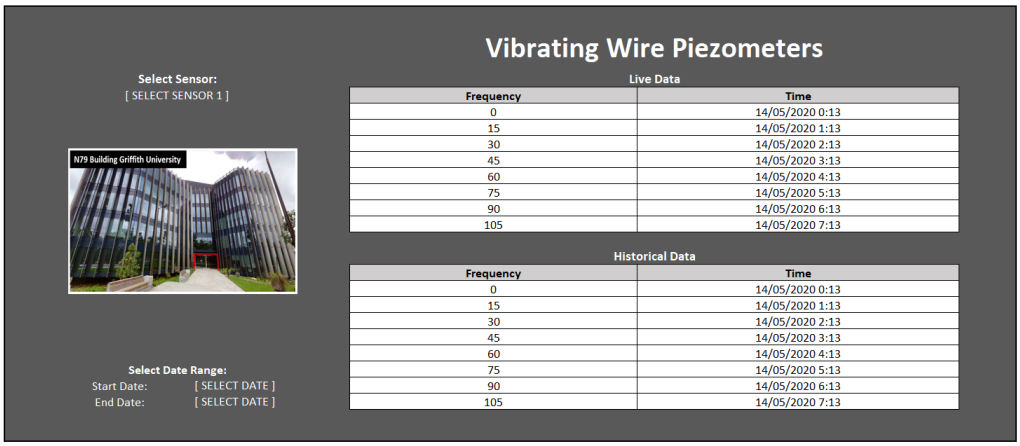


Figure 39: Secondary design for the vibrating wire piezometer dashboard using tables.

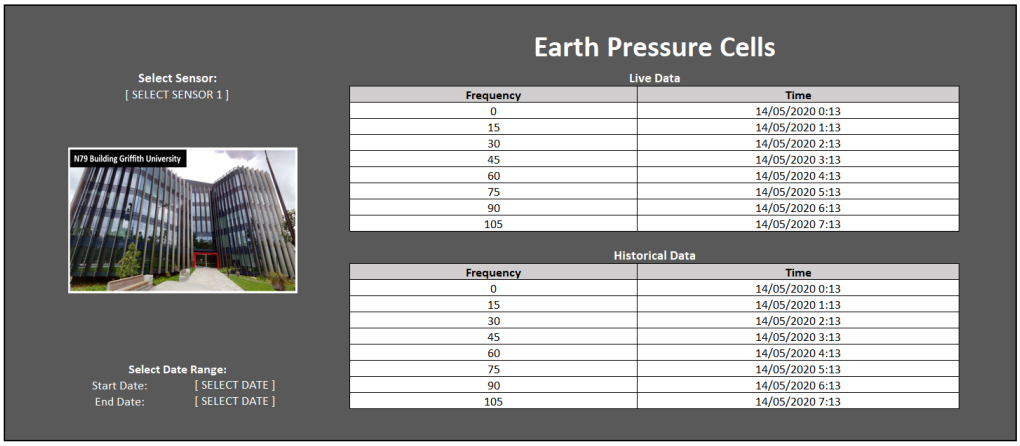


Figure 40: Secondary design for the earth pressure cell dashboard using tables.

When presenting these designs to the client, they were happy with the designs and the alternative solutions that were being used to accomodate the KX dashboard development tool constraints. The client mentioned that they wanted a label included near the date range of the dashboards, which would be used to notify users not to select date ranges that exceed 3 months (as per the 10,000 rows of data constraint).

Iteration 3: Final Design

When developing the final designs for the vibrating wire piezometer and earth pressure cell dashboards, a warning label was added underneath the time and date range in order to notify the user not to select time periods longer than 3 months. The label was added to help accomodate

the 10,000 row constraint of the dashboard development tool, as selecting periods longer than 104 days will result in only a portion of the data being presented - which could potentially lead to misinterpretation of the data by the user. 3 months was selected as the label for the recommended limit in order to provide a small level of redundancy regarding the 10,000 row constraint.

The final designs for the vibrating wire piezometer and earth pressure cell dashboards, which use line charts to present the data and which were developed using the KX dashboard development tool can be seen in figures 41 and 42 respectively.

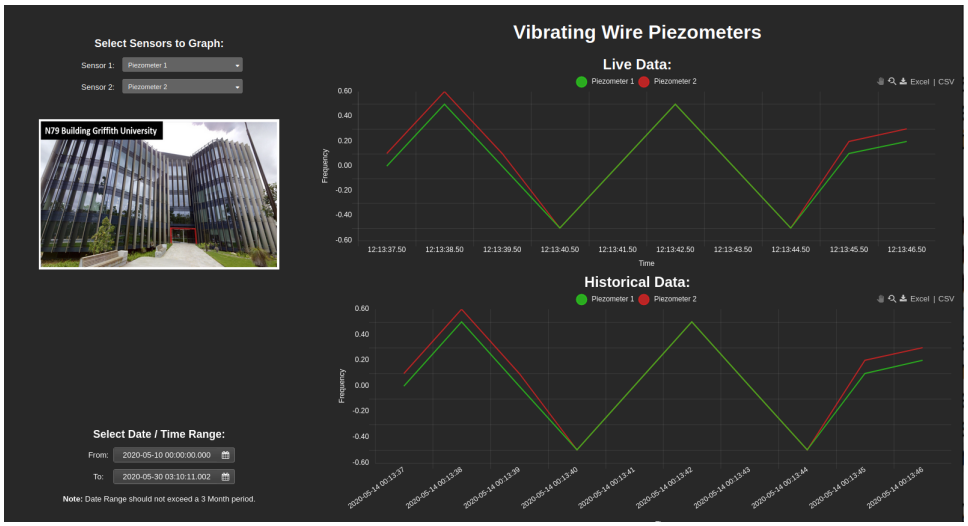


Figure 41: Final design for the vibrating wire piezometer dashboard using line charts.

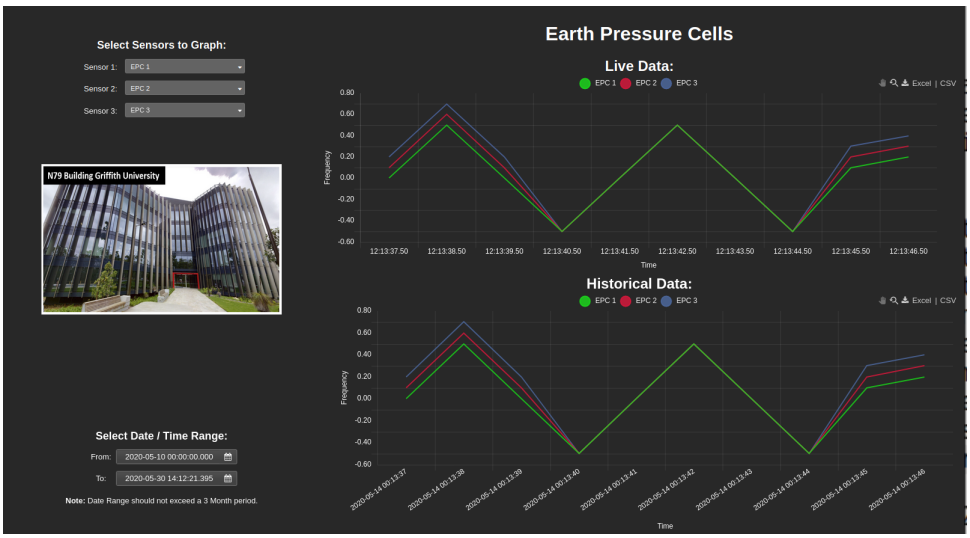


Figure 42: Final design for the earth pressure cell dashboard using line charts.

The final designs for the vibrating wire piezometer and earth pressure cell dashboards, which

use tables to present the data and which were developed using the KX dashboard development tool can be seen in figures 43 and 44 respectively.

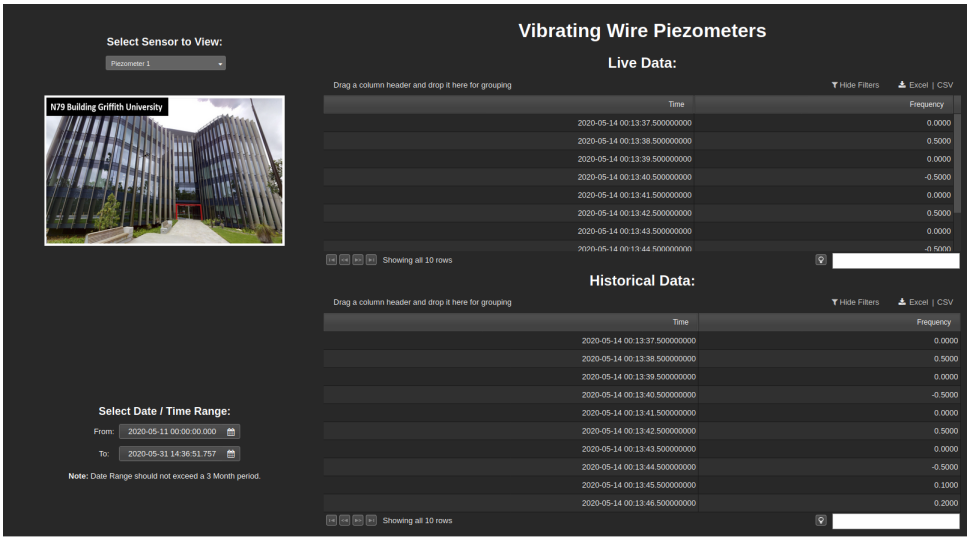


Figure 43: Final design for the vibrating wire piezometer dashboard using tables.

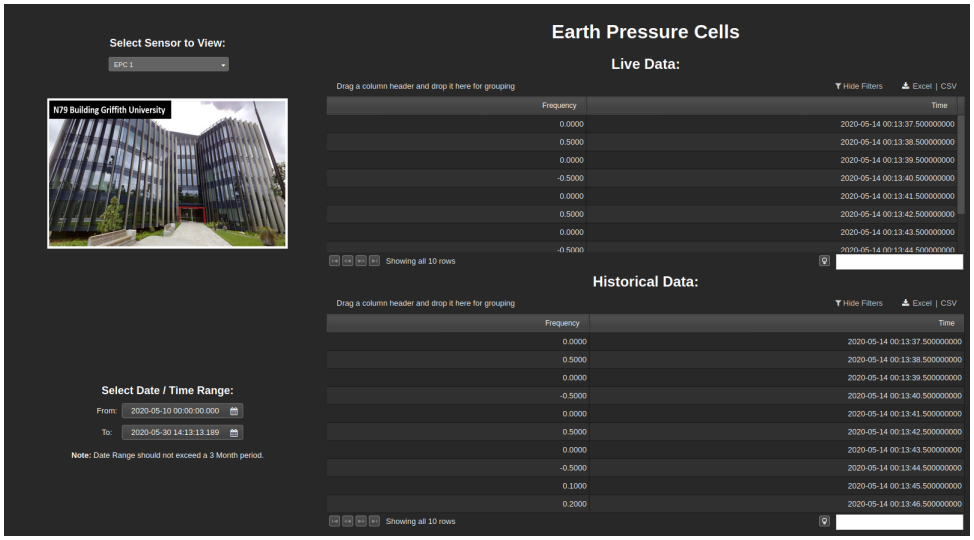


Figure 44: Final design for the earth pressure cell dashboard using tables.

5 TESTING AND RESULTS

This section provides details of the testing that was conducted to validate the functionality of the work developed during the project, and to verify that the project outcomes met the objectives that were set out for the project. Specifically, this section will provide an overview of the testing methodology used for the project, provide details of the software testing, runtime testing, user testing and system acceptance testing that was conducted, and discuss the testing results.

5.1 Methodology

The testing methodology used for the project involved 4 types of testing, consisting of software testing, runtime testing, user testing and system acceptance testing. The software testing was used to validate the functionality of the software developed during the project and ensure it worked as intended. Runtime testing was used to verify the project system is functional, stable and can run continuously for extended periods without interruption. User testing was used to ensure the dashboards that were developed during the project convey the intended information, are user-friendly and adhere to the recommended design features for informational dashboards. Lastly, system acceptance testing was conducted using a combination of testing, visual inspection and client approval to ascertain that each of the project objectives were being met.

5.1.1 Software Testing

Unit testing was selected as the model for our software testing, using the Python unittest software library to develop the test cases. The decision to use unit testing was made due to its ability to validate the functionality of the software, increase client confidence regarding the quality of the software, and help identify the cause and location of detected bugs through testing specific lines of code.

The test cases developed were used to help ensure the data retrieval software module can retrieve accelerometer and strain gauge sensor data from the N79 building sensors and store the data within the CSV file format, that the PI System software module can retrieve the vibrating wire piezometer and earth pressure cell sensor data from the Griffith University PI System and store it within the CSV file format, and that the data communication software module can process

the sensor data stored within the CSV files.

A further goal of the software testing was to help minimize the risk of future bugs being introduced when software changes are made, which can be achieved through running the test cases to re-validate the software functionality whenever maintenance is performed. The Python unit tests can also be run automatically whenever the project software is recompiled.

5.1.2 Runtime Testing

Runtime testing was conducted on the project system to validate the stability and functionality of the system, and ensure the software could run for extended periods of time without interruption. These tests were important as the system is expected to continuously retrieve and store the N79 building sensor data, in order to regularly monitor and analyze the performance of the N79 building.

When performing runtime testing, the project system was set to run continuously using the N79 server room computer for an extended period, where any errors that occurred were recorded. Load testing was conducted on the computer during the runtime testing, in order to measure the demands that are placed on its components and provide results regarding its performance. The results gathered from load testing were used to verify that the N79 server room computer can run the project system continuously without any existing or expected future hardware issues being experienced on the machine.

5.1.3 User Testing

User testing was conducted to gather feedback pertaining to the user experience of the dashboards, focusing on how well the dashboards met the features outlined within the recommended guidelines (dashboards that are clear, accurate, simple, meaningful and consistent), whether users found the dashboards easy to use and whether they could easily interpret and understand the information being presented within the dashboards.

To achieve the user testing goals, 10 people were asked to use a project dashboard and complete a survey rating different parts of their experience. When selecting the users who took part in the survey, effort was invested to find people who have relatively diverse backgrounds - which helped to mitigate any bias from the feedback. Furthermore, a civil engineer was included

within the sample of users selected in order to verify the performance of the dashboards against the intended end users (civil engineers who monitor buildings). Each users professional background was recorded as part of the survey, in order to maintain a record of how different career backgrounds may influence a users experience. When designing the surveys, questions were developed asking users to rate different aspects of their experience on a scale from 1 to 10. The lower end of the scale was used to denote qualities that are not well suited to informational dashboards - such as clutteredness, difficulty navigating the dashboards or difficulty with interpreting the information. The higher end of the scale was used to denote qualities that are desirable within informational dashboards - such as simplicity, ease of use and ease of interpretation.

A sample of the survey used for user testing can be seen in appendix D.

5.1.4 System Acceptance Testing

System acceptance testing was conducted using a combination of software testing, runtime testing, user testing, visual inspection and client approval to verify that the core objectives for the project were met.

The completion of each objective for the project was measured against a series of tests designed to provide evidence of their completion, with the client (Adriano Marinho) inspecting the results of the tests and signing off on each objective he approved as being met.

5.2 Testing Results

5.2.1 Software Testing

The software unit testing that was completed for the project software involved 30 test cases for the data retrieval and data communication software modules, which all passed when running the test cases - see figure 45.

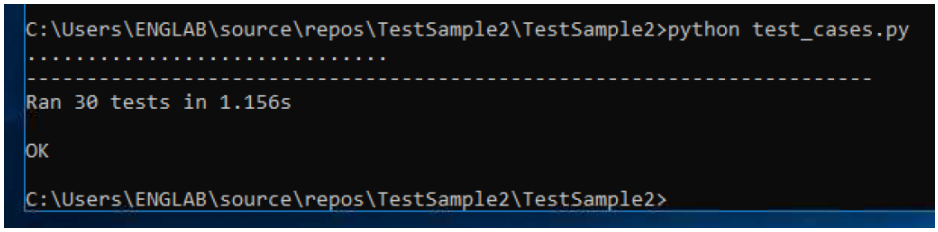


Figure 45: Software unit testing results.

Test cases for the PI System software module were not completed, due to maintenance being performed on the Griffith University PI System that prevented access to the vibrating wire piezometer and earth pressure cell data stored on the system - which resulted in the PI System software module not being completed during the project.

The source code for the test cases that were developed for the data retrieval and data communication software modules can be seen in Appendix E.

5.2.2 Runtime Testing

The computer that was used to run the project system during runtime testing was a Dell Latitude 5580 computer, which uses a 64 bit operating system and consists of a dual core i5-7300U Central Processing Unit (CPU) running at 2.6 GHz with 32 GB of Random Access Memory (RAM).

Runtime testing was conducted on the machine for a period of 48 hours, where the system was continuously retrieving sensor data from the cDAQ-9171 for the 3 accelerometers currently installed within the N79 building.

The system ran continuously throughout the 48 hour period without interruption, measuring a total of 446,342,000 samples from each of the 3 accelerometers installed within the N79 building (based on 892,684 CSV files written during the period, which each currently store 500 samples of data for each accelerometer). These results indicated an average sampling rate for the accelerometers of 2583 Hz - which was close to the expected result of 2500 Hz.

Load testing was conducted for the first 24 hours of runtime testing. During load testing, it was observed that the data communication software module reads and communicates the CSV files faster than the data retrieval software module retrieves and stores the data - resulting in

Average CPU Usage (Total)	Peak CPU Usage (Total)	Average RAM Usage (Total)	Peak RAM Usage (Total)
41.3%	70.1%	54.0%	54.6%

Table 2: Computer CPU and RAM usage (total) measured during the 24 hour load testing of the project system.

minimal hard disk space being required over time (as CSV files are not being built up over time, assuming the data communication software module is running). The load testing measured the total usage for the CPU and RAM of the computer, which were indicative of the usage required for typical operating conditions. The results for the load testing can be seen in table 2 and figure 46.

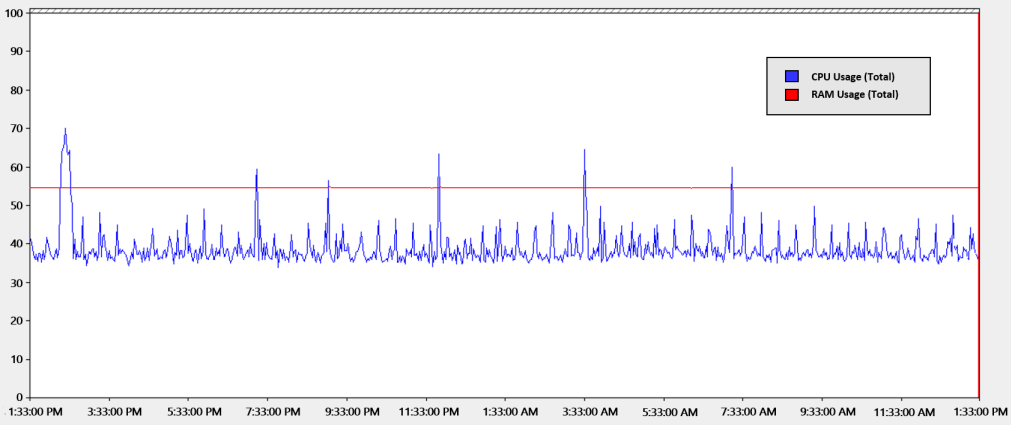


Figure 46: Computer CPU and RAM usage (Total) measured during the 24 hour load testing of the project system.

As seen in the load testing, the server room computer was highly capable of maintaining the hardware performance required to run the project software, averaging a total CPU load of 41.3% and peaking at 70.1%. The total RAM usage remained fairly consistent at 54% throughout the load testing, which was due to the Windows operating system installed on the computer automatically preallocating free RAM when not in use.

5.2.3 User Testing

User testing was conducted on 10 test users, who each tested a dashboard before completing the survey provided in Appendix D.

Overall, the results gathered from user testing were very positive, with the average user rating their overall experience using the dashboards at 8.4 out of 10 (84%), with a standard deviation of 0.66, and rating the dashboards adherance to the recommended design features at an average of 8.55 out of 10 (85.5%), with a standard deviation of 1.16. A civil engineer who tested the dashboard rated their overall experience at 9 out of 10 (90%), rating the dashboards adherance to the recommended design features at an average of 8.75 out of 10 (87.5%), with a standard deviation of 0.83 - which were particularly significant results considering civil engineers are the target end user audience for the dashboards (civil engineers who use the dashboards to monitor the N79 building). These results provided evidence that the dashboards adhered to the recommended design guidelines for dashboards, that the dashboards were capable of being used for their intended purpose, and that they met the associated client requirements for the project.

Some feedback that was gathered from user testing that can be used to improve upon the future designs was to include an incrementor to the data filters, which would allow users to tap a button to increment / deincrement the data filter values. As this feature is currently not available within the KX dashboard development tool, the feedback has been passed along to Urban Institute staff as a possible feature that can be included within the future of the tool. Furthermore, a user mentioned to provide more context to the information being shown - as they were initially not sure what accelerometers or strain gauges were, or what they measured. The full results gathered from user testing can be seen in table 3.

5.2.4 System Acceptance Testing

System acceptance testing was used to verify which objectives of the project were completed.

The core objectives for the project were to:

1. Develop software that continuously retrieves and processes the live sensor data from the N79 building accelerometers and strain gauges.

User #	Background	Navigation (/10)	Understanding (/10)	Simplicity (/10)	Usability (/10)	Experience (/10)
1	Mechanic	9	8	9	9	9
2	Electrical Engineer	9	10	10	9	9
3	Software Engineer	9	9	10	10	9
4	Radio Sales	10	8	9	8	8
5	Beautician	9	7	10	8	8
6	Civil Engineer	8	9	10	8	9
7	Security Guard	9	7	8	8	8
8	Electrical Engineer	9	8	10	9	9
9	Nurse	6	5	8	6	7
10	Truck Driver	9	8	9	8	8
Total: (/100)	N/A	**87**	**79**	**93**	**83**	**84**

Table 3: Dashboard user testing results.

2. Develop software that continuously retrieves the live and historical sensor data for the N79 building vibrating wire piezometers and earth pressure cells from the Griffith University PI System.

3. Develop software that continuously communicates the processed sensor data with the KX server.

4. Develop software within the KX platform that continuously ingests and stores the processed sensor data within KX databases, making the data available for use within the KX dashboards.

5. Develop a series of informational dashboards which utilize a collection of visualizations to present the processed sensor data in a user-friendly way, and is easy for users to interpret and understand.

The system acceptance testing that was conducted to verify these objectives were met is summarized in table 4. A copy of the system acceptance testing that was signed off on by the client (Adriano Marinho) can be seen in Appendix F.

Related Objective Number(s)	Test Description	Result
1	Software unit testing of the data retrieval module to verify accelerometer and strain gauge sensor data can be collected from the N79 Building cDAQ when connected	Passed
1	Software unit testing of the data retrieval module to verify accelerometer and strain gauge sensor data is organized into unique CSV files for temporary storage	Passed
1	Software unit testing of the data communication module to verify accelerometer and strain gauge sensor data can be retrieved from CSV files	Passed
2	Software unit testing to verify the vibrating wire piezometer and earth pressure cell sensor data can be retrieved from the Griffith University PI System	Not Completed (Refer to 4.3.2)
3	Software unit testing to verify network access to the KX server web socket address that was configured for ingesting and storing the N79 building sensor data	Passed
1,3,4	Runtime testing to verify the project system was stable, and could run continuously for an extended period without interupption	Passed
1,3	Load testing to verify the N79 building server room computer could run the project system continuously for an extended period without any existing or expected future hardware issues being experienced on the machine	Passed
3,4	Printing the output of relevant processes used within the KX server to visually verify that sensor data is being received by the KX server and being stored within the KX databases	Passed
3,4	Using KX dashboards to visualize the retrieval of sensor data stored within the KX databases, with the sensor data being presented within a series of visualisations	Passed
5	Client approval of the informational dashboards that were developed during the project	Passed
5	Dashboard user testing to verify the dashboards convey the intended information and can be used by end users for their intended purpose (monitoring the structural state of the N79 building)	Passed
5	Dashboard user testing to verify that the dashboards developed follow recommended dashboard design guidelines	Passed

Table 4: System acceptance testing results.

6 DISCUSSION

This section will provide a reflection of the methods used during the project, and discuss the challenges that were presented when conducting the project work.

6.1 Methods

The agile development framework significantly helped throughout the project through gathering regular feedback and having the project work being constantly re-evaluated. The agile development framework was particularly important considering the unpredictable nature of the coronavirus and the restrictions that were being frequently updated to help mitigate its spread. Furthermore, as their were constraints within the KX dashboard development tool that were periodically discovered throughout the course of the project, hosting regular meetings with the client helped to adapt the original dashboard designs in order to accomodate these constraints. Through several meetings with the client, the dashboard designs were able to be continuously improved in order to develop a series of dashboards that the clients were happy with.

The Python programming language provided a range of useful software libraries that were used to complete the project software. The Python Nidaqmx software library was easy to learn and proved highly effective for communicating with the National Instruments cDAQ-9171, which was used for retrieving the N79 building accelerometer and strain gauge data. The Python unittest library was also very effective for performing software unit testing, which made test cases easy to develop and run, and provided useful feedback when tests failed - which helped save substantial amounts of time when resolving software problems that emerged during the project. The Python unittest library was also very useful for revalidating the functionality of the software when changes were made, and provides tools that can be used to periodically test the software automatically in the future.

The data communication module was capable of communicating sensor data from any source, as long as the data is formatted correctly within CSV files and placed within the spool directory that the software monitors. Separating the communication of the sensor data from the retrieval of the sensor data proved to be a highly effective decision, as the software can continuously retrieve sensor data even if there are communication problems (such as if the KX server goes

down, or the network experiences issues). Furthermore, new software modules that retrieve sensor data can utilize the same data communication software module, provided that the sensor data is organized into a valid format - which could save time in the future through not having to rewrite software in order to communicate the associated data with the KX server.

When reflecting on the project, the biggest thing that would have been done differently would have been to spend more time organizing the hardware setup of the N79 building server room. In particular, the Linux computer of the server room is a very outdated machine, which performed very slowly and completely crashed several times during the project. Next time, more time would be spent organizing a more modern computer to run the KX server software, as doing so would have resulted in less frequent delays associated with the hardware of the machine.

6.2 Challenges

6.2.1 Project Management

Due to the lockdown caused by the coronavirus, a lot of resources directly involved with the project were made more inaccessible - with buildings being locked down and staff being made to work from home. Extensive project management had to be invested to help mitigate the impact of the coronavirus pandemic, such as organizing more frequent online meetings (due to social distancing / staff working from home), meetings having to be more carefully planned (due to meetings being shorter, as a lot of staff had their workloads substantially increased due to the coronavirus), setting up and managing remote access tools to ensure work could be completed remotely, and more careful planning and preparation had to be conducted in order to access resources that were required for the project - such as buildings and hardware.

When the N79 building went into a full lockdown due to the coronavirus, applications had to be submitted before anyone could be authorized to re-enter the building. This was a particularly significant delay within the project, as a lot of important hardware involved with the project was installed within the N79 building (such as the building sensors, data acquisition devices and computers), which often had to be physically maintained due to problems that occurred (such as power cuts and unexpected crashing of the hardware). When delays were experienced regarding various resources of the project, work was undertaken within the other areas of the

project wherever possible in order to help minimize any downtime.

6.2.2 Server Room Computer

The Linux computer being used with the N79 building server room presented several issues during the project - such as slow processing, software bugs, constantly freezing and unexpectedly crashing. The issues with the Linux computer contributed several delays within the project work. In particular, as a lot of the work involved with the dashboards had to be completed using this computer (as it was running the KX server software for the project, including the KX dashboard development tool), the dashboards took longer than anticipated to complete. The computer was not able to be easily replaced during the project due to the coronavirus - which included substantial product shipping delays, factories closing down, staff working from home and Griffith University buildings where alternate hardware may be located often being unaccessible.

6.2.3 Remote Access

Several times during the project, remote access to the server room computers of the N79 building were distrupted. As these computers were connected to the building sensors, and were responsible for running the project software - the loss of remote access lead to several delays within the project.

Remote access was lost on several occassions due to power cuts within the N79 building, the server room computers freezing or unexpectedly crashing, and the server room computers having to be reset in order to install critical KX software updates. As the N79 building was often unoccupied due to the coronavirus lockdown, a physical visit to the building was often required in order to fix the remote access issues each time.

A particular loss of remote access was caused due to a scheduled power cut within the N79 building. After the power had been reset and the associated computer had been restarted, the computer could still not access the internet or establish a remote access connection. During a physical visit to the building in an attempt to resolve the issue, the issue was discovered to be caused by an incorrect time and date on the machine (which were reset to default values during the power cut), which interfered with critical network protocols and subsequently prevented access to the internet. Once the error had been discovered and the time and date on the machine

had been reset to their correct values, the remote access to the machine was restored.

6.2.4 Dashboard Development Tool Constraints

During the development of the dashboards, several constraints were discovered with the KX dashboard development tool that impacted the original designs for the dashboards.

As the constraints within the KX dashboard development tool were only periodically discovered, the dashboard designs had to be continuously reworked in order to accomodate the constraints. Furthermore, as the designs had to be adjusted in a manner that the clients were still happy with - more regular meetings had to be scheduled with the clients in order to develop alternative solutions that the clients were happy with.

Although the dashboards required numerous iterations in order to circumvent these constraints, the core goals and features that the client wanted from the dashboards were able to be implemented using the alternate solutions that were proposed, resulting in a set of dashboards that the clients were happy with.

7 CONCLUSION

This section will summarize the project outcomes that were produced during the project, provide details for future work and recommendations, and provide a project summary for the work that was conducted.

7.1 Project Outcomes

The final results for the project achieved the majority of objectives that were originally laid out in section 1.2, with the exception of retrieving the N79 building vibrating wire piezometer and earth pressure cell data from the Griffith University PI System - which could not be completed due to the PI System being inaccessible for long periods during the project.

The project work that was completed produced the following outcomes:

1. Software that continuously retrieves the N79 building accelerometer and strain gauge data.
2. Software that continuously communicates processed sensor data with Urban Institutes KX server.
3. A total of 30 software test cases were developed that validated the functionality of the project software, which also allow the software to be retested if maintenance changes are introduced within the future.
4. Software within the KX server was developed to continuously ingest and store the sensor data within the KX databases, making the data accessible to the KX dashboards.
5. A total of 6 informational dashboards were developed, which present the N79 building sensor data in a clear and meaningful way using a series of visualizations.

7.2 Future Work and Recommendations

Reflecting upon the work that was conducted during the project, a set of future work and recommendations have been provided below, which can be implemented to expand upon the project work in the future.

7.2.1 Replace Linux Server Room Computer

The current Linux machine responsible for running Urban institute KX server software is an outdated machine - which often performs tasks very slowly, freezes completely or unexpectedly crashes. As this computer is responsible for running the KX server software, which ingests the sensor data into the KX databases - the successful operation of this computer forms a critical part of the project system. An important recommendation is therefore that this computer is updated with a more modern machine. The new computer should include more up-to-date hardware, run the latest Ubuntu operating system and have all of the KX server software associated with the project migrated to the new machine.

7.2.2 Server Room Maintenance

The server room of the N79 building contains the computer responsible for retrieving the accelerometer and strain gauge sensor data, in addition to the computer responsible for running the KX server software associated with the project. An image showing the current layout for the server room can be seen in figure 47.

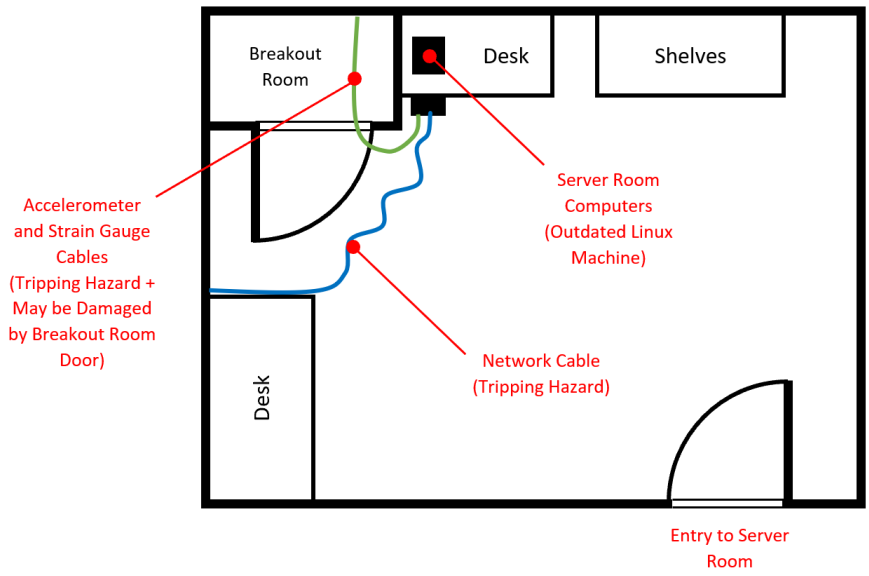


Figure 47: N79 building server room layout.

Currently the accelerometer and strain gauge sensor cables are connected into the server room through an internal breakout room, where the sensor cables traverse through the doorway of

the breakout room in order to connect to the relevant computer. The doorway of the breakout room is currently being held open by a block of wood in order to allow the sensor cables to pass through. However these sensor cables may present a tripping hazard to occupants of the server room, and may also be damaged if the block of wood is unexpectedly displaced - as the door would slam shut on the cables. There is also an ethernet cable that traverses across the floor of the server room and connects to one of the server room computers, which presents a tripping hazard.

These sensor cables should be organized into a safer manner, such that they do not present a tripping hazard and are not at risk of being damaged by the breakout room door.

I therefore recommend that the cables and computers associated with the project are all organized into the breakout room, with potentially a monitor outside the room that could be used to display the current state of the system. A rack should be used to connect all the sensors located within the server room (which will eventually connect 25+ sensors), and placed with the rest of the hardware inside the breakout room. A wireless router, or a network jack closer to the server room computers should be connected in order to remove the tripping hazard currently created by the ethernet cable running across the floor.

Lastly, the computers and cDAQ of the server room should be connected to an Uninterruptable Power Supply (UPS), as the system requires a consistent power supply in order to continuously retrieve the N79 building sensor data. A UPS will prevent any scheduled or unexpected power cuts interfering with the retrieval of the sensor data - which will be particularly important in the event of a disaster (such as an earthquake), where the sensor data could be required to evaluate the structural state of the building.

7.2.3 Automatic Email Notification for Software Errors

Currently the project software writes any errors that occur to an error log file, which can be used to help fix any software errors that occur. The error messaging system could be expanded to include a feature where error messages are automatically sent to an email address, which can be monitored by staff responsible for maintaining the project software, notifying them that an error has occurred - which will help speed up the process of resolving any errors and restoring the functionality of the software.

7.2.4 Organize Project Software into Live and Development Environments

Currently there is only one version of the project software being maintained. I recommend that the project software is organized into live and development environments, where any software changes are made within the development environment, then thoroughly tested before being used within the live environment to retrieve and store the N79 building sensor data.

This will allow software changes and testing to be conducted without interrupting the retrieval and storage of the sensor data being measured. Furthermore, previous versions of the development environment software can be used when any unexpected bugs are introduced within the live environment - providing system redundancy and helping mitigate the loss of sensor data if unexpected errors occur.

7.2.5 Scheduled Patching for Server Room Computers

I recommend that regular updates and patching for the server room computers are scheduled in order to ensure any associated software updates are maintained, and preventing any software issues that may arise due to incompatibility or outdated software - particularly with regards to the computer operating systems, the KX platform and the Python programming language.

Furthermore, once the remaining accelerometers and strain gauges are installed within the N79 building, slight adjustments will need to be made within the data retrieval software module. These changes include adding the new channels, sensor types and sensor ids within the software, which are used to specify which channels the cDAQ will communicate with and retrieve sensor data from.

7.2.6 Develop Software to Retrieve PI System Sensor Data

The PI System software that was intended to retrieve the vibrating wire piezometer and earth pressure cell data for the N79 was completed during the project, due to maintenance being conducted on the PI System that prevented access to the associated data. As this sensor data provides important structural information for maintaining the N79 building and monitoring its performance, I recommend the software is completed once the PI System is made to be accessible again.

7.3 Summary

The project conducted at Griffith University to retrieve, store and visualize the sensor data being measured by the collection of sensors set to be installed within the N79 building was considered successful. The software developed to retrieve the N79 building sensor data proved to be functional, scalable and resilient - retrieving sensor data from the existing accelerometers that had been intalled within the N79 building for a period of 48 hours without interruption, and passing the software unit test cases that were developed. A total of 6 dashboards were developed that provided interfaces and features that the clients were happy with. The dashboards incorporated designs that were simple, displayed data meaningfully and accurately, and were easy for users to navigate - which helped users visualise and interpret the sensor data presented quickly, as indicated by the user testing that was conducted during the project.

The agile development framework and User Centred Design (UCD) approaches that were used throughout the project proved to be significantly helpful. In particular, gathering regular feedback from the clients in conjunction with frequent iterations of the project dashboards help to continuously improve upon their designs, and adapt the features of the dashboards to the constraints that emerged without sacrificing on the clients needs.

Despite the challenges that were introduced with the project, additional effort was invested in order to adapt to the conditions of the coronavirus pandemic. The majority of objectives for the project were met, producing a system that could be used to continuously retrieve and visualize the sensor data from the N79 building.

8 REFERENCES

Please Note: After each reference there are links indicating each section where the reference is cited within the text.

[1] Charles R Farrar and Keith Worden. An introduction to structural health monitoring. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 365(1851):303–315, 2007. 1.1.1, 2.1, 2.1, 2.1

[2] Jennifer A Rice and BF Spencer Jr. Structural health monitoring sensor development for the imote2 platform. In *Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2008*, volume 6932, page 693234. International Society for Optics and Photonics, 2008. 1.1.1, 2.1

[3] Peter Cawley. Structural health monitoring: Closing the gap between research and industrial deployment. *Structural Health Monitoring*, 17(5):1225–1244, 2018. 1.2

[4] Daniel Balageas, Claus-Peter Fritzen, and Alfredo Güemes. *Structural health monitoring*, volume 90. 2010. 2.1, 2.1

[5] Victor Giurgiutiu. *Structural health monitoring: with piezoelectric wafer active sensors*. 2007. 2.1

[6] James MW Brownjohn. Structural health monitoring of civil infrastructure. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 365(1851):589–592, 2007. 2.1, 2.1

[7] Jerome Peter Lynch. An overview of wireless structural health monitoring for civil structures. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 365(1851):345–347, 2007. 2.1, 2.1, 2.3, 2.3

[8] George Horwich. Economic lessons of the kobe earthquake. *Economic development and cultural change*, 48(3):521–522, 2000. 2.1

[9] Charles R Farrar and Nick AJ Lieven. Damage prognosis: the future of structural health monitoring. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 365(1851):623–625, 2007. 2.1

[10] Kobe Earthquake of 1995. https://www.britannica.com/event/Kobe-earthquake-of-1995. Accessed: 2020-06-25.

[11] Chandrasekaran Srinivasan. *Structural health monitoring with application to offshore structures*. World Scientific, 2019. 2.1

[12] David Lehrer and Janani Vasudev. Visualizing information to improve building performance: a study of expert users. 2010. 2.1

[13] Juan Davila Delgado et al. Modelling, management, and visualisation of structural performance monitoring data on bim. In *Transforming the Future of Infrastructure through Smarter Information: Proceedings of the International Conference on Smart Infrastructure and ConstructionConstruction, 27–29 June 2016*, pages 543–549. ICE Publishing, 2016. 2.1

[14] Girish Krishnan et al. Micromachined high-resolution accelerometers. *Journal of the Indian Institute of Science*, 87(3):333, 2012. 2.2.1

[15] How to Measure Strain and Pressure With Strain Gage Sensors. https://dewesoft.com/daq/measure-strain-and-pressure. Accessed: 2020-06-25. 2.2.2

[16] Strain Gauges. https://www.allaboutcircuits.com/textbook/direct-current/chpt-9/strain-gauges/. Accessed: 2020-06-25. 2.2.2

[17] Sisgeo Vibrating Wire Piezometers. https://www.sisgeo.com/uploads/schede/schede/VW_PIEZO_EN_05_vibrating_wire_piezometer.pdf. Accessed: 2020-06-25. 2.2.3, 3.1

[18] Porfirio V Guerrero, HKM Vicente, and Agustín de Gortari. Comparison of excitement methods of vibrant wire sensors. *African Journal of Physics ISSN*, 3(4):78–80, 2016.

[19] Sisgeo Earth Pressure Cells. https://www.sisgeo.com/uploads/schede/schede/L140_EN_11_Earth_pressure_cells.pdf. Accessed: 2020-06-25. 2.2.4, 2.3, 3.1

[20] Jerome P Lynch and Kenneth J Loh. A summary review of wireless sensors and sensor networks for structural health monitoring. *Shock and Vibration Digest*, 38(2):98, 2006.

[21] Gyuhae Park et al. Energy harvesting for structural health monitoring sensor networks. *Journal of Infrastructure Systems*, 14(1):64–79, 2008. 2.3

[22] Leonardo Militano et al. Recharging versus replacing sensor nodes using mobile robots for network maintenance. *Telecommunication Systems*, 63(4):625–642, 2016. 2.3

[23] Amy Franklin et al. Dashboard visualizations: Supporting real-time throughput decision-making. *Journal of biomedical informatics*, 71:211–221, 2017. 2.4

[24] Amar Sahay. *Data Visualization, Volume II: Uncovering the Hidden Pattern in Data Using Basic and New Quality Tools*. Business Expert Press, 2017. 2.4

[25] Colin Ware. *Information visualization: perception for design*. Morgan Kaufmann, 2019. 2.4

[26] Dashboard Design. https://www.klipfolio.com/dashboard-design. Accessed: 2020-06-25. 2.4.1

[27] How to Design and Build a Great Dashboard. https://www.geckoboard.com/best-practice/dashboard-design/. Accessed: 2020-06-25. 2.4.1

[28] Taras Bakusevych. 10 Rules for Better Dashboard Design. https://uxplanet.org/10-rules-for-better-dashboard-design-ef68189d734c. Accessed: 2020-06-25. 2.4.1

[29] What Makes a Great Dashboard. https://dataschool.com/how-to-design-a-dashboard/what-makes-a-great-dashboard-aces/. Accessed: 2020-06-25. 2.4.1

[30] Joseph F Obermaier et al. *Employing User-Centered Design to Accelerate the Construction of a Business Intelligence Dashboard*. PhD thesis, 2018. 2.4.2

[31] James Shore et al. *The Art of Agile Development: Pragmatic guide to agile software development*. " O'Reilly Media, Inc.", 2007. 2.4.2, 4.1

[32] National Instruments CDAQ-9171. https://www.ni.com/en-au/support/model.cdaq-9171.html. Accessed: 2020-06-25. 3.1

[33] Wilcoxon 731-207 Compact, Ultra Low-Frequency Accelerometer. https://wilcoxon.com/wp-content/uploads/2018/11/731-207-spec-98069D.2.pdf. Accessed: 2020-06-25. 3.1

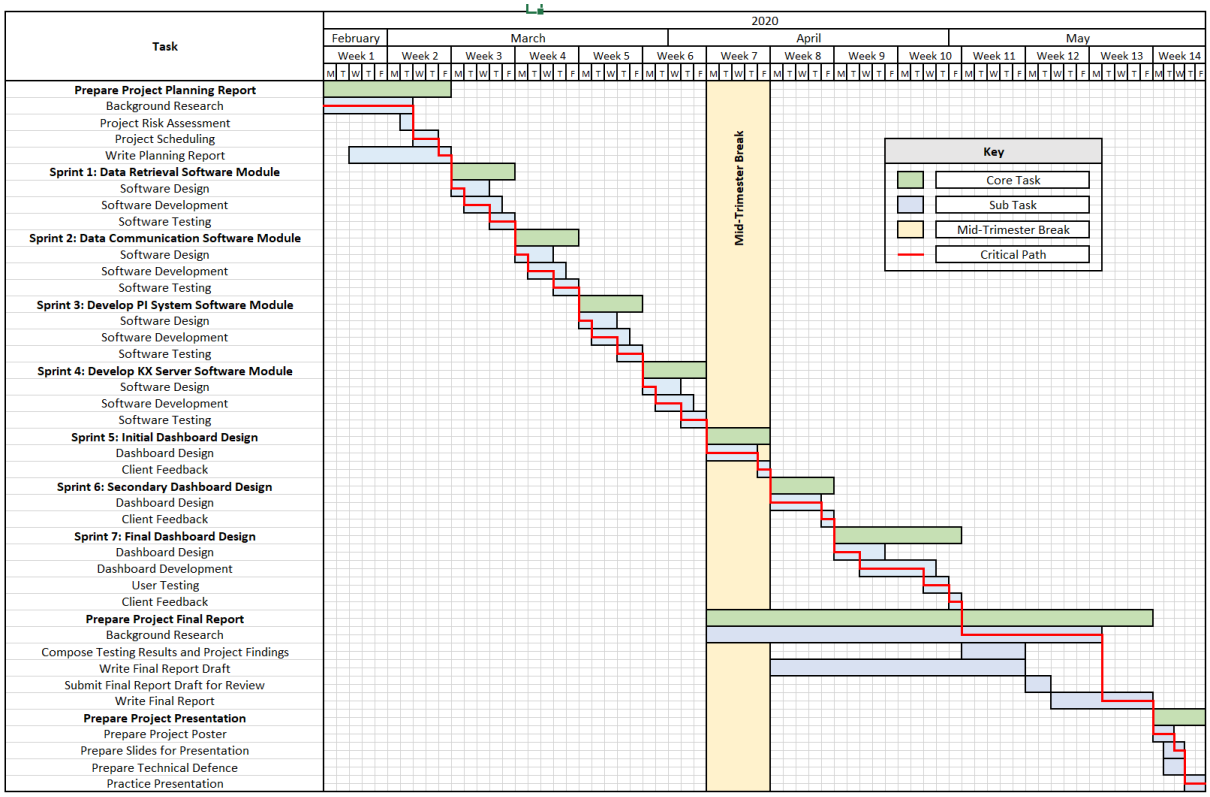
[34] PCB Pizotronics 740B02 Strain Gauge. https://www.pcb.com/products?model=740B02. Accessed: 2020-06-25. 3.1

[35] OSISoft PI System. https://www.osisoft.com/pi-system/. Accessed: 2020-06-25. 3.3, 3.3

[36] Take Advantage of Downsampling for Better Visualisation. https://www.datapine.com/documentation/downsampling/. Accessed: 2020-06-25. 4.4.1

9 APPENDICES

9.1 Appendix A: Initial Project Schedule



9.2 Appendix B: Data Retrieval Software Module

```
import nidaqmx

import csv

import os

import sys

import logging

import os.path

import time

from os import path

from nidaqmx.constants import AcquisitionType, Edge

from datetime import datetime


# setup_task is a function to configure a nidaqmx task to
```

```
    ↪ continuously read samples from the cDAQ (Data Acquitision
    ↪ Device)
# Input Parameters:
# device_name - a string containing the device name for the
    ↪ cDAQ connected to the computer
# channels - a list containing the channels that you would like
    ↪  to create virtual channels for (must be contained within
    ↪ the list ["ai0", "ai1", "ai2", "ai3"]
# clock_rate - the sample clock rate for the task. Must be a
    ↪ number greater than or requal to 1
#
# Return Parameters:
# task - nidaqmx Task that has been configured to communicate
    ↪ with the cDAQ
def setup_task(device_name, channels, clock_rate):

    # perform error checking on input parameters
    for i in channels:
        if(i not in ["ai0", "ai1", "ai2", "ai3", "ai4"]):
            raise Exception("setup_task:␣invalid␣channel␣name␣
                ↪ detected");

    if(not isinstance(device_name, str)):
        raise Exception("setup_task:␣device␣name␣must␣be␣a␣String"
            ↪ );

    if(not isinstance(clock_rate, (int, float)) or clock_rate <=
        ↪  1):
        raise Exception("setup_task:␣clock_rate␣must␣be␣a␣number␣
            ↪ greater␣than␣or␣equal␣to␣1");
```

```
    # start a task to access the National Instruments cDAQ
    task = nidaqmx.Task();

    try:
        # create a virtual channel for each of the channels
            ↪ provided
        for ch in channels:
            task.ai_channels.add_ai_voltage_chan(device_name + "/"
                ↪  + ch);

        # configure the clock rate for the task
        task.timing.cfg_samp_clk_timing(clock_rate, source="",
            ↪ active_edge=Edge.FALLING, sample_mode=
            ↪ AcquisitionType.FINITE, samps_per_chan=75);

    except Exception as e:
        task.close();
        raise Exception("Error␣configuring␣task:␣" + str(e));

    return task

# create_csv_file creates a new csv file with a unique name
    ↪ that can be written to
# Return Parameters:
# fp - file pointer to the new csv file that has been created
def create_csv_file():

    # retrieve the current time
    time = datetime.now();
```

```
    # create a new csv file to store the input data
    fp = open('N79_Sensor_Data_' + time.strftime("%d_%m_%Y_%H_%
        ↪ M_%S_%f") + ".csv", mode="w", newline='');

    return fp;

# write_data_to_csv is a function to write input data collected
    ↪  from a cDAQ (Data Acquisition Device) until the number of
    ↪  maximum rows for the file have been reached
# Input Parameters:
# fp - file pointer containing the csv file to be written to
# data - data to be written to csv file (a list, containing
    ↪ lists of [sensor_id, sensor_type, voltage, timestamp])

def write_data_to_csv(fp, data):

    seen = [];

    if(len(data) <= 0):
        return;

    # headers for the csv file
    headers = ["sensor_id", "sensor_type", "sensor_value", "
        ↪ sensor_derived_value", "sensor_timestamp"];

    # perform error checking on input parameters
    if(not os.access(fp.name, os.F_OK)):
        raise Exception("write_data_to_csv:␣file␣pointed␣to␣by␣fp
            ↪ ␣does␣not␣exist");
```

```
if(not os.access(fp.name, os.W_OK) or not os.access(fp.name,
   ↪  os.R_OK)):
   raise Exception("write_data_to_csv:␣file␣pointed␣by␣fp␣
      ↪ must␣have␣read␣and␣write␣permissions");

if(not fp.name.endswith(".csv")):
   raise Exception("write_data_to_csv:␣fp␣must␣point␣to␣a␣
      ↪ csv␣file");

csv_writer = csv.writer(fp, delimiter=',');

# write data headers to csv file
csv_writer.writerow(headers);

# write data to csv file
for row in data:

   # calculate acceleration measurement for current voltage
      ↪ value
   # As per specifications for acclerometer being used:
      ↪ acceleration = input (mV) / 10
   if(row["type"] == "Accelerometer"):
      derived_value = row["value"] / 10;
   elif(row["type"] == "Strain␣Gauge"):
      derived_value = row["value"];
   else:
      derived_value = row["value"];

   csv_writer.writerow([row["id"], row["type"], row["value"
```

```
        ↪ ], derived_value, row["timestamp"]]);

def main():

    # device name for the cDAQ
    device_name = "cDAQ3Mod1";

    # define sensor types connected to cDAQ. They should be
       ↪ arranged in the order they are connectected (i.e
       ↪ channel ai0 -> index 0)3
    # NOTE - Each sensor Type should be "Accelerometer" or "
       ↪ Strain Gauge"
    # Each index of the sensor_type list should map the channel
       ↪ of the associated sensor connected to the cDAQ
    sensor_type = ["Accelerometer", "Accelerometer", "
       ↪ Accelerometer"];

    # channel names for the cDAQ that you want to read inputs
       ↪ from
    # NOTE: the number of channels must be greater than 0 and
       ↪ max_file_rows must be evenly divisible by the number of
       ↪  channels
    # Channel names should be of the format "ai<channel number
       ↪ >", as recognized by the cDAQ
    channels = ["ai0", "ai1", "ai2"];

    # define sensor ids for the sensors currently connected to
       ↪ the cDAQ
    # NOTE: The number of ids must be equal to the number of
       ↪ channels, with each id being unique
```

```
# Each index of the sensor id list should map the channel of
   ↪ the associated sensor connected to the cDAQ
# Each sensor is should use the following format:
# Accelerometers: "Level <floor> Accelerometer <unique
   ↪ identifier>". The unique idenifier can be used when
# multiple accelerometers are located on the same floor.
# Strain Gauge: "Strain Gauge <unique identifier>". The
   ↪ unique identifier is currently a number between 1-15.
# However it can be used in future to provide further
   ↪ context, such as location.
sensor_ids = ["Level␣1␣Accelerometer", "Level␣3␣
   ↪ Accelerometer", "Level␣5␣Accelerometer"];

# define the maximum sample clock rate for onboard clock of
   ↪ the cDAQ
clock_rate = 2000.0;

# define the subdirectory where the input data files will be
   ↪ stored
# NOTE: this should be the same spool directory where the
   ↪ input files are being processed
data_directory = "Sensor␣Data";

# define the number of rows that will be written to each csv
   ↪ file (1 row = 1 input from a single sensor connected
   ↪ to the cDAQ)
# NOTE: max_file_rows should be greater than 0 and must be
   ↪ evenly divisible by the number of channels
max_file_rows = 1500;
```

```
# configure the task for the cDAQ
task = setup_task(device_name, channels, clock_rate);

while(True):

    # counter to track the amount of rows to write to csv
        ↪ file
    row_counter = 0;

    if(len(channels) < 1 or ((max_file_rows % len(channels))
        ↪ != 0)):
        raise ValueError('max_file_rows must be evenly
            ↪ divisible by the number of channels');

    if(len(sensor_ids) != len(sensor_type)):
        raise ValueError('the length of sensor_ids must match
            ↪ the length of sensor types');

    for t in sensor_type:
        if(not isinstance(t, str)):
            raise ValueError('each sensor type must be a string
                ↪ ');


    # list to collect rows of input data from the sensors
    data = [];

    while(row_counter < max_file_rows):

        # retrieve the current time
```

```
time = datetime.now();

timestamp = time.strftime("%Y-%m-%dT%H:%M:%S.%f");

# read data from the input channels
inputs = task.read();

# perform error checking
if(len(sensor_ids) != len(inputs)):
    raise ValueError('The␣number␣of␣sensor_ids␣did␣not␣
        ↪ match␣the␣number␣of␣inputs␣being␣read␣from␣the
        ↪ ␣cDAQ');

if(len(inputs) != len(channels)):
    raise ValueError('the␣number␣of␣channels␣did␣not␣
        ↪ match␣the␣number␣of␣inputs␣being␣read␣from␣the
        ↪ ␣cDAQ');



for i in range(0, len(inputs)):
    row = {
        "id": sensor_ids[i],
        "type": sensor_type[i],
        "timestamp": timestamp,
        "value": inputs[i]
    }

    data.append(row);

    row_counter += 1;
```

```
        # create new csv file to write input data to
        f = create_csv_file();

        # write the input data collected from the cDAQ until the
            ↪ csv file has reached the maximum number of rows
        write_data_to_csv(f, data);

        # close the file
        f.close();

        # move the file into the directory defined by
            ↪ data_directory
        cwd = os.getcwd();
        os.replace(os.path.realpath(f.name), cwd + '/' +
            ↪ data_directory + '/' + os.path.basename(f.name));

    # close the nidaqmx task to free up resources
    task.close();

# configure log file for writing uncaught errors and exceptions
logger = logging.getLogger(__name__);
logging.basicConfig(filename="error.log", filemode='a', level=
    ↪ logging.INFO, format='%(asctime)s␣%(levelname)-8s␣%(
    ↪ message)s', datefmt="%d/%m/%Y␣%I:%M:%S␣%p");

# handle_uncaught_exception is a function to write the uncaught
    ↪  exceptions to the log file
def handle_uncaught_exception(exc_type, exc_value,
    ↪ exc_traceback):
```

```
    if(issubclass(exc_type, KeyboardInterrupt)):
        sys.__excepthook__(exc_type, exc_value, exc_traceback);
        return
    logger.critical("Uncaught Exception", exc_info=(exc_type,
        ↪ exc_value, exc_traceback));

# configure default exception handling to use the method we
    ↪ defined
sys.excepthook = handle_uncaught_exception;

# run main function
if __name__ == '__main__':
    main();
```

9.3 Appendix C: Data Communication Software Module

```
import os
import logging
import os.path
from os import path
from datetime import datetime
from datetime import timedelta
import json
import csv
import requests
import time
import sys

# process_csv_file reads the data from a csv file, organizing
    ↪ each row of input into an array
```

```
# Input Parameters:
# directory - subdirectory location that holds the csv file to
   ↪ be processed (usually the spool directory)
# file_name - name of the csv file to be processed
#
# Return Parameter:
# csv_data - dictionary containing the rows of data from the
   ↪ csv file (1 row from csv file = 1 row in dictionary)
def process_csv_file(directory, file_name):

   # dictionary to store csv file data
   csv_data = [];

   # perform error checking on input parameters
   if(not isinstance(directory, str)):
      raise Exception("process_csv_file:␣input␣directory␣must␣
         ↪ be␣a␣String");

   if(not os.path.isdir("./" + directory)):
      raise Exception("process_csv_file:␣input␣directory␣does␣
         ↪ not␣exist");

   if(not isinstance(file_name, str)):
      raise Exception("process_csv_file:␣input␣file_name␣must␣
         ↪ be␣a␣String");

   if(not file_name.endswith(".csv")):
      raise Exception("process_csv_file:␣input␣file_name␣must␣
         ↪ be␣a␣.csv␣file");
```

```
    if(not os.path.isfile("./" + directory + "/" + file_name)):
        raise Exception("process_csv_file: input file_name does
            ↪ not exist in the input directory");

    time.sleep(0.1);

    # open file and read input into dictionary
    with open(directory + "/" + file_name, mode='r') as csv_file
        ↪ :
        csv_reader = csv.DictReader(csv_file);

        # add csv data to return dictionary
        for row in csv_reader:
            csv_data.append(row);

        # close csv file
        csv_file.close();

    return csv_data;

# process csv_data takes a dictionary of input data, converts
    ↪ it to json and communicates it to a web socket
# Input Parameters:
# data - list of dictionaries containing the input data to be
    ↪ communicated to the web socket
# web_socket - address of the web_socket to communicate the
    ↪ data to
#
# Return Parameter:
# r - response from the web socket
```

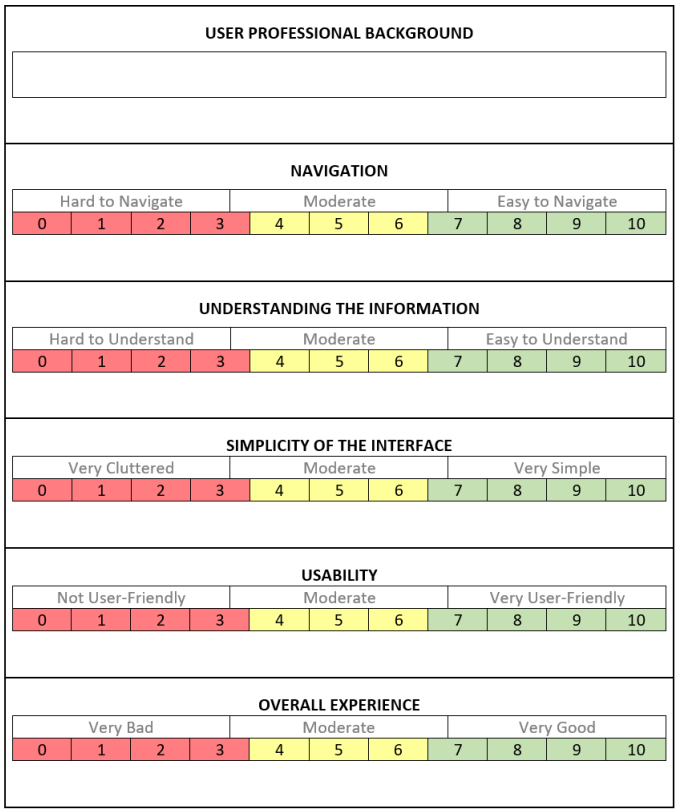
```
def process_csv_data(data, web_socket):

    # perform error checking on input parameters
    if (not isinstance(data, list)):
        raise Exception("process_csv_data: input data must be a
            ↪ list of dictionaries");
    else:
        for i in data:
            if (not isinstance(i, dict)):
                raise Exception("process_csv_data: input data must
                    ↪ be a list of dictionaries");

    if(not isinstance(web_socket, str)):
        raise Exception("process_csv_data: web socket must be a
            ↪ String");

    # post request headers (used in request to send sensor data)
    headers = { 'Content-type': 'application/json' };

    #convert sensor data to json
    json_body = json.dumps(data);

    # send data to web socket
    r = requests.post(web_socket, data=json_body, headers=
        ↪ headers);

    return r

def main():
```

```
# name of the file directory that will contain the .csv
   ↪ files containing the sensor data
fileDirectory = "Sensor Data";

# web socket url and port to send sensor data
web_socket = "http://132.234.39.78:4005";

# number of rows sent per post request to the web socket
request_limit = 120;

# perform error checking
if(not isinstance(request_limit, int) or request_limit < 1):
   raise Exception("main: request limit must be an integer
      ↪ greater than 1")

if(not os.path.isdir("./" + fileDirectory)):
   raise Exception("main: the file directory provided by
      ↪ fileDirectory does not exist");

# continuously process csv files from the spool directory
while (True):

   # process files with .csv extension
   for sensor_file in os.listdir(fileDirectory + "/"):
      if sensor_file.endswith(".csv"):

         # retrieve data from csv file
         data = process_csv_file(fileDirectory, sensor_file)
            ↪ ;
```

```
json_data = [];

while(len(data) > 0):

    json_data.append(data.pop());

    # communicate input data to web socket each time
      ↪ we reach the number of rows equal to the
      ↪ request limit (number of rows sent per
      ↪ request)
    if (len(json_data) >= request_limit or len(data)
      ↪ <= 0):

        response = process_csv_data(json_data,
          ↪ web_socket);

        print(response.status_code);

        if(response.status_code != 200):
            print("error posting data to web socket");

        json_data = [];

try:
    # delete the file after it has been read and
      ↪ communicated
    os.remove(fileDirectory + "/" + sensor_file);

except Exception as e:
    print("Exception Raised deleting file: {}, 
```

```
                    ↪ Exception:␣{}".format(sensor_file, e));
                sys.exit(1);

# configure log file for writing uncaught errors and exceptions
logger = logging.getLogger(__name__);
logging.basicConfig(filename="error.log", filemode='a', level=
   ↪ logging.INFO, format='%(asctime)s␣%(levelname)-8s␣%(
   ↪ message)s', datefmt="%d/%m/%Y␣%I:%M:%S␣%p");

# handle_uncaught_exception is a function to write the uncaught
   ↪  exceptions to the log file
def handle_uncaught_exception(exc_type, exc_value,
   ↪ exc_traceback):

   if(issubclass(exc_type, KeyboardInterrupt)):
      sys.__excepthook__(exc_type, exc_value, exc_traceback);
      return
   logger.critical("Uncaught␣Exception", exc_info=(exc_type,
      ↪ exc_value, exc_traceback));

# configure default exception handling to use the method we
   ↪ defined
sys.excepthook = handle_uncaught_exception;

# run main function
if __name__ == '__main__':
   main();
```

9.4 Appendix D: Dashboard User Testing Survey



USER PROFESSIONAL BACKGROUND

NAVIGATION

Hard to Navigate				Moderate			Easy to Navigate			
0	1	2	3	4	5	6	7	8	9	10

UNDERSTANDING THE INFORMATION

Hard to Understand				Moderate			Easy to Understand			
0	1	2	3	4	5	6	7	8	9	10

SIMPLICITY OF THE INTERFACE

Very Cluttered				Moderate			Very Simple			
0	1	2	3	4	5	6	7	8	9	10

USABILITY

Not User-Friendly				Moderate			Very User-Friendly			
0	1	2	3	4	5	6	7	8	9	10

OVERALL EXPERIENCE

Very Bad				Moderate			Very Good			
0	1	2	3	4	5	6	7	8	9	10

9.5 Appendix E: Software Unit Testing

```
import unittest
import retrieveData
import readCsvData
import nidaqmx
import csv
import time
import os.path
import platform
import subprocess
from nidaqmx.constants import AcquisitionType, Edge


class TestSensorData(unittest.TestCase):
```

```
#
#
# TESTS FOR CONFIGURING CDAQ
#
#

    # test configuring task with valid parameters and 3 virtual
      ↪ channels
    def test_setup_task_valid_with_3(self):
        device = "cDAQ3Mod1";
        channels = ["ai0", "ai1", "ai2"];
        clock_rate = 2000;
        task = retrieveData.setup_task(device, channels,
          ↪ clock_rate);

        self.assertTrue(isinstance(task, nidaqmx.Task));
        self.assertEqual(task.number_of_channels, 3);
        self.assertTrue("cDAQ3Mod1/ai0" in task.ai_channels.
          ↪ channel_names);
        self.assertTrue("cDAQ3Mod1/ai1" in task.ai_channels.
          ↪ channel_names);
        self.assertTrue("cDAQ3Mod1/ai2" in task.ai_channels.
          ↪ channel_names);

        task.close();

    # test configuring task with valid parameters and 2 virtual
      ↪ channels
    def test_setup_task_valid_with_2(self):
        device = "cDAQ3Mod1";
```

```
        channels = ["ai0", "ai1"];
        clock_rate = 1500;
        task = retrieveData.setup_task(device, channels,
            ↪ clock_rate);

        self.assertTrue(isinstance(task, nidaqmx.Task));
        self.assertEqual(task.number_of_channels, 2);
        self.assertTrue("cDAQ3Mod1/ai0" in task.ai_channels.
            ↪ channel_names);
        self.assertTrue("cDAQ3Mod1/ai1" in task.ai_channels.
            ↪ channel_names);

        task.close();

    # test configuring task with valid parameters and 1 virtual
        ↪ channel
    def test_setup_task_valid_with_1(self):
        device = "cDAQ3Mod1";
        channels = ["ai0"];
        clock_rate = 2000;
        task = retrieveData.setup_task(device, channels,
            ↪ clock_rate);

        self.assertTrue(isinstance(task, nidaqmx.Task));
        self.assertEqual(task.number_of_channels, 1);
        self.assertTrue("cDAQ3Mod1/ai0" in task.ai_channels.
            ↪ channel_names);

        task.close();
```

```
    # test configuring task with float clock rate
    def test_setup_task_float_clock(self):
        device = "cDAQ3Mod1";
        channels = ["ai0"];
        clock_rate = 2000.0;

        task = retrieveData.setup_task(device, channels,
            ↪ clock_rate);

        self.assertTrue(isinstance(task, nidaqmx.Task));
        self.assertEqual(task.number_of_channels, 1);
        self.assertTrue("cDAQ3Mod1/ai0" in task.ai_channels.
            ↪ channel_names);

        task.close();


    # test configuring task with maximum clock rate of 0
    def test_setup_task_clock_0(self):
        device = "cDAQ3Mod1";
        channels = ["ai0"];
        clock_rate = 0;

        with self.assertRaises(Exception): retrieveData.
            ↪ setup_task(device, channels, clock_rate);

    # test configuring task with invalid device name
    def test_setup_task_invalid_device(self):
        device = "fake_name";
        channels = ["ai0"];
```

```
        clock_rate = 2000;

        with self.assertRaises(Exception): retrieveData.
            ↪ setup_task(device, channels, clock_rate);

    # test configuring task with invalid channel
    def test_setup_task_invalid_channel(self):
        device = "cDAQ3Mod1";
        channels = ["ai9"];
        clock_rate = 2000;

        with self.assertRaises(Exception): retrieveData.
            ↪ setup_task(device, channels, clock_rate);

    # test configuring task with a one valid and one invalid
        ↪ channel
    def test_setup_task_invalid_channel_type1(self):
        device = "cDAQ3Mod1";
        channels = ["ai0", "ai6"];
        clock_rate = 2000;

        with self.assertRaises(Exception): retrieveData.
            ↪ setup_task(device, channels, clock_rate);

    # test configuring task with invalid channel type
    def test_setup_task_invalid_channel_type2(self):
        device = "cDAQ3Mod1";
        channels = [3];
        clock_rate = 2000;
```

```
        with self.assertRaises(Exception): retrieveData.
            ↪ setup_task(device, channels, clock_rate);

    # test configuring task with negative maximum clock rate
    def test_setup_task_negative_clock(self):
        device = "cDAQ3Mod1";
        channels = ["ai0"];
        clock_rate = -2000;

        with self.assertRaises(Exception): retrieveData.
            ↪ setup_task(device, channels, clock_rate);

    # test configuring task with invalid device type
    def test_setup_task_invalid_device_type(self):
        device = 345;
        channels = ["ai0"];
        clock_rate = 2000;

        with self.assertRaises(Exception): retrieveData.
            ↪ setup_task(device, channels, clock_rate);

    # test configuring task with invalid clock_rate type
    def test_setup_task_invalid_clock_type(self):
        device = "cDAQ3Mod1";
        channels = ["ai0"];
        clock_rate = "2000";

        with self.assertRaises(Exception): retrieveData.
            ↪ setup_task(device, channels, clock_rate);
```

```
    # test configuring task with channels not in a list
    def test_setup_task_channels_not_in_list(self):
        device = "cDAQ3Mod1";
        channels = "ai0";
        clock_rate = 2000;

        with self.assertRaises(Exception): retrieveData.
            ↪ setup_task(device, channels, clock_rate);

#
#
# TESTS FOR CREATING UNIQUE CSV FILES
#
#

    # test creating a csv file
    def test_creating_csv_file(self):

        fp = retrieveData.create_csv_file();

        self.assertTrue(os.path.isfile("./" + fp.name));

        fp.close();
        os.remove("./" + fp.name);

    # test creating unique csv files
    def test_creating_csv_file_unique(self):

        fp1 = retrieveData.create_csv_file();
        time.sleep(0.5);
```

```
        fp2 = retrieveData.create_csv_file();

        self.assertTrue(os.path.isfile("./" + fp1.name));
        self.assertTrue(os.path.isfile("./" + fp2.name));
        self.assertNotEqual(fp1.name, fp2.name);

        fp1.close();
        fp2.close();

        os.remove("./" + fp1.name);
        os.remove("./" + fp2.name);

#
#
# TESTS FOR WRITING DATA TO CSV FILE
#
#
    # test writing to csv file with valid parameters and 1
      ↪ virtual task reading from the cDAQ
    def test_write_data_to_csv_valid_with_1(self):

        fp = open("test_file.csv", mode="w+", newline='');

        sensor_type = "Accelerometer";
        data = [
            {
                "id": "Level_1_Acc",
                "type": "Accelerometer",
                "value": 0.5,
                "timestamp": "2020-04-20T00:23:18.481"
```

```
    },
    {
        "id": "Level_3_Acc",
        "type": "Accelerometer",
        "value": 0.9,
        "timestamp": "2020-04-20T00:23:18.481"
    },
    {
        "id": "Level_5_Acc",
        "type": "Accelerometer",
        "value": -0.4,
        "timestamp": "2020-04-20T00:23:18.481"
    }
];

retrieveData.write_data_to_csv(fp, data);

fp.close();

fp = open(fp.name, mode="r", newline='');

csv_reader = csv.reader(fp, delimiter=",");
line_count = 0;
data = [];

for row in csv_reader:
    data.append(row);
    line_count += 1;

fp.close();
```

```
        for y in data[0]:
            self.assertTrue(y in ["sensor_id", "sensor_type", "
                ↪ sensor_value", "sensor_derived_value", "
                ↪ sensor_timestamp"]);

        os.remove(fp.name);

    # test writing to csv file with invalid data format
    def test_write_data_to_csv_with_invalid_data_format(self):

        fp = open("test_file.csv", mode="w", newline='');

        sensor_type = "Accelerometer";
        data = [
            {
                "fake": "Level_1_Acc",
                "type": "Accelerometer",
                "value": 0.5,
                "timestamp": "2020-04-20T00:23:18.481"
            },
            {
                "fake": "Level_3_Acc",
                "type": "Accelerometer",
                "value": 0.9,
                "timestamp": "2020-04-20T00:23:18.481"
            },
            {
                "fake": "Level_5_Acc",
                "type": "Accelerometer",
```

```
            "value": -0.4,
            "timestamp": "2020-04-20T00:23:18.481"
        }
    ];

    with self.assertRaises(Exception): retrieveData.
        ↪ write_data_to_csv(fp, data);

    fp.close();
    os.remove(fp.name);

# test writing to csv file that doesn't exist
def test_write_data_to_csv_no_file(self):

    fp = open("test_file.csv", mode="w", newline='');
    fp.close();
    os.remove(fp.name);

    sensor_type = "Accelerometer";
    data = [
        {
            "id": "Level_1_Acc",
            "type": "Accelerometer",
            "value": 0.5,
            "timestamp": "2020-04-20T00:23:18.481"
        },
        {
            "id": "Level_3_Acc",
            "type": "Accelerometer",
            "value": 0.9,
```

```
            "timestamp": "2020-04-20T00:23:18.481"
        },
        {
            "id": "Level_5_Acc",
            "type": "Accelerometer",
            "value": -0.4,
            "timestamp": "2020-04-20T00:23:18.481"
        }
    ];

    with self.assertRaises(Exception): retrieveData.
        ↪ write_data_to_csv(fp, data);

# test writing to file that's not a csv
def test_write_data_to_csv_invalid_file_ext(self):

    fp = open("test_file.txt", mode="w", newline='');

    sensor_type = "Accelerometer";
    data = [
        {
            "id": "Level_1_Acc",
            "type": "Accelerometer",
            "value": 0.5,
            "timestamp": "2020-04-20T00:23:18.481"
        },
        {
            "id": "Level_3_Acc",
            "type": "Accelerometer",
            "value": 0.9,
```

```
            "timestamp": "2020-04-20T00:23:18.481"
        },
        {
            "id": "Level_5_Acc",
            "type": "Accelerometer",
            "value": -0.4,
            "timestamp": "2020-04-20T00:23:18.481"
        }
    ];

    with self.assertRaises(Exception): retrieveData.
        ↪ write_data_to_csv(fp, data);

    fp.close();



#
#
# TESTS FOR READING CSV FILES
#
#

    # test reading csv file with valid paramaters 1
    def test_read_csv_valid1(self):

        directory = "Sensor Data";

        fp = open("test_file.csv", mode="w", newline='');

        csv_writer = csv.writer(fp, delimiter=',');
```

```
csv_writer.writerow(["sensor_id", "sensor_type", "
   ↪ sensor_value", "timestamp"]);
csv_writer.writerow(["Level_1_Acc", "Accelerometer", 0.5,
   ↪  "12/06/1998␣13:15:23.001589"]);
csv_writer.writerow(["Level_3_Str", "Strain␣Gauge",
   ↪ 0.243, "12/06/1998␣16:25:29.538532"]);

fp.close();

cwd = os.getcwd();
os.replace(os.path.realpath(fp.name), cwd + '/' +
   ↪ directory + '/' + os.path.basename(fp.name));

input_data = readCsvData.process_csv_file(directory, fp.
   ↪ name);

self.assertEqual(len(input_data), 2);

for x in input_data[0].keys():
   self.assertTrue(x in ["sensor_id", "sensor_type", "
      ↪ sensor_value", "timestamp"]);

for y in input_data[0].values():
   self.assertTrue(y in ["Level_1_Acc", "Accelerometer",
      ↪ "0.5", "12/06/1998␣13:15:23.001589"]);

for x in input_data[1].keys():
   self.assertTrue(x in ["sensor_id", "sensor_type", "
      ↪ sensor_value", "timestamp"]);
```

```
        for y in input_data[1].values():
            self.assertTrue(y in ["Level_3_Str", "Strain Gauge", "
                ↪ 0.243", "12/06/1998 16:25:29.538532"]);

        os.remove(directory + "/" + fp.name);


    # test reading csv file with valid paramaters 2
    def test_read_csv_valid2(self):

        directory = "Sensor Data";

        fp = open("test_file.csv", mode="w", newline='');

        csv_writer = csv.writer(fp, delimiter=',');

        csv_writer.writerow(["sensor_id", "sensor_type", "
            ↪ sensor_value", "timestamp"]);
        csv_writer.writerow(["Level_5_Acc", "Accelerometer", "
            ↪ 0.21", "17/11/1999 13:15:23.000589"]);
        csv_writer.writerow(["Level_1_Str", "Strain Gauge", "
            ↪ -0.187", "20/09/20018 18:25:28.238032"]);
        csv_writer.writerow(["Level_5_Wire", "Wire Vibration", "
            ↪ -0.45", "17/10/2009 12:25:11.038532"]);

        fp.close();

        cwd = os.getcwd();
        os.replace(os.path.realpath(fp.name), cwd + '/' +
```

```
    ↪ directory + '//' + os.path.basename(fp.name));

input_data = readCsvData.process_csv_file(directory, fp.
    ↪ name);

self.assertEqual(len(input_data), 3);

for x in input_data[0].keys():
    self.assertTrue(x in ["sensor_id", "sensor_type", "
        ↪ sensor_value", "timestamp"]);

for y in input_data[0].values():
    self.assertTrue(y in ["Level_5_Acc", "Accelerometer",
        ↪ "0.21", "17/11/1999␣13:15:23.000589"]);

for x in input_data[1].keys():
    self.assertTrue(x in ["sensor_id", "sensor_type", "
        ↪ sensor_value", "timestamp"]);

for y in input_data[1].values():
    self.assertTrue(y in ["Level_1_Str", "Strain␣Gauge", "
        ↪ -0.187", "20/09/20018␣18:25:28.238032"]);

for x in input_data[2].keys():
    self.assertTrue(x in ["sensor_id", "sensor_type", "
        ↪ sensor_value", "timestamp"]);

for y in input_data[2].values():
    self.assertTrue(y in ["Level_5_Wire", "Wire␣Vibration"
        ↪ , "-0.45", "17/10/2009␣12:25:11.038532"]);
```

```
        os.remove(directory + "/" + fp.name);

    # test reading csv file with directory that doesn't exist
    def test_read_csv_no_dir(self):

        directory = "Fake";

        fp = open("test_file.csv", mode="w", newline='');

        csv_writer = csv.writer(fp, delimiter=',');

        csv_writer.writerow(["sensor_id", "sensor_type", "
            ↪ sensor_value", "timestamp"]);
        csv_writer.writerow(["Level_5_Acc", "Accelerometer",
            ↪ 0.21, "17/11/1999␣13:15:23.000589"]);

        fp.close();

        cwd = os.getcwd();
        os.replace(os.path.realpath(fp.name), cwd + '/' + "Sensor
            ↪ ␣Data" + '/' + os.path.basename(fp.name));

        with self.assertRaises(Exception): readCsvData.
            ↪ process_csv_file(directory, fp.name);

        os.remove("Sensor␣Data" + "/" + fp.name);


    # test reading csv file with file that doesn't exist
```

```
def test_read_csv_no_file(self):

    directory = "Sensor Data";

    fp = open("test_file.csv", mode="w", newline='');

    csv_writer = csv.writer(fp, delimiter=',');

    csv_writer.writerow(["sensor_id", "sensor_type", "
       ↪ sensor_value", "timestamp"]);
    csv_writer.writerow(["Level_5_Acc", "Accelerometer",
       ↪ 0.21, "17/11/1999 13:15:23.000589"]);

    fp.close();
    cwd = os.getcwd();
    os.replace(os.path.realpath(fp.name), cwd + '/' +
       ↪ directory + '/' + os.path.basename(fp.name));
    os.remove(directory + "/" + fp.name);

    with self.assertRaises(Exception): readCsvData.
       ↪ process_csv_file(directory, fp.name);


# test reading csv file that's empty
def test_read_csv_empty_csv(self):

    directory = "Sensor Data";

    fp = open("test_file.csv", mode="w", newline='');
    fp.close();
```

```
    cwd = os.getcwd();
    os.replace(os.path.realpath(fp.name), cwd + '/' +
       ↪ directory + '/' + os.path.basename(fp.name));

    input_data = readCsvData.process_csv_file(directory, fp.
       ↪ name);

    self.assertEqual(len(input_data), 0);

    os.remove(directory + "/" + fp.name);


# test reading a file that's not a csv
def test_read_csv_not_csv(self):

    directory = "Sensor Data";

    fp = open("test_file.txt", mode="w", newline='');
    fp.close();

    cwd = os.getcwd();
    os.replace(os.path.realpath(fp.name), cwd + '/' +
       ↪ directory + '/' + os.path.basename(fp.name));

    with self.assertRaises(Exception): readCsvData.
       ↪ process_csv_file(directory, fp.name);

    os.remove(directory + "/" + fp.name);
```

```
    # test reading a file that's not a string
    def test_read_csv_file_not_string(self):

        directory = "Sensor_Data";

        with self.assertRaises(Exception): readCsvData.
            ↪ process_csv_file(directory, 12345);

    # test reading a directory that's not a string
    def test_read_csv_dir_not_string(self):

        fp = open("test_file.csv", mode="w", newline='');
        fp.close();

        cwd = os.getcwd();
        os.replace(os.path.realpath(fp.name), cwd + '/' + "Sensor
            ↪ _Data" + '/' + os.path.basename(fp.name));

        with self.assertRaises(Exception): readCsvData.
            ↪ process_csv_file(123, fp.name);

        os.remove("Sensor_Data" + '/' + fp.name);

#
#
# TESTS FOR PROCESSING DATA READ FROM CSV FILES
#
#
```

```
# test processing data that is not a list of dictionaries
def test_invalid_data_format(self):

    data = [56];

    with self.assertRaises(Exception): readCsvData.
        ↪ process_csv_data(data, "112.34.56");

# test processing data with an invalid web socket type
def test_invalid_web_address(self):

    data = [
        {
            "sensor_id": "Level_1_Acc",
            "sensor_type": "Accelerometer",
            "sensor_value": 0.5,
            "sensor_derived_value": 0.05,
            "sensor_timestamp": "2020-04-20T00:23:18.481"
        },
        {
            "sensor_id": "Level_3_Acc",
            "sensor_type": "Accelerometer",
            "sensor_value": 0.9,
            "sensor_derived_value": 0.09,
            "sensor_timestamp": "2020-04-20T00:23:18.481"
        },
        {
            "sensor_id": "Level_5_Acc",
            "sensor_type": "Accelerometer",
            "sensor_value": -0.1,
```

```
            "sensor_derived_value": -0.01,
            "sensor_timestamp": "2020-04-20T00:23:18.481"
        }
    ];

    with self.assertRaises(Exception): readCsvData.
        ↪ process_csv_data(data, 112);

# test availability of Urban Institute server web socket
def test_web_socket_availability(self):

    current_os = platform.system().lower();

    if (current_os == "windows"):
        parameter = "-n";
    else:
        parameter = "-c";

    ip = "http://132.234.39.78:4005";
    exit_code = subprocess.call(['ping', parameter, '1', ip])
        ↪ ;

    self.assertTrue(True);


if __name__ == '__main__':
    unittest.main();
```

9.6 Appendix F: Client Approval of System Acceptance Testing

Related Objective Number(s)	Test Description	Result
1	Software Unit Testing of the data retrieval module to verify accelerometer and strain gauge sensor data can be collected from the N79 Building cDAQ when connected	Passed
1	Software Unit Testing of the data retrieval module to verify accelerometer and strain gauge sensor data is organized into unique CSV files for temporary storage	Passed
1	Software Unit Testing of the data communication module to verify accelerometer and strain gauge sensor data can be retrieved from CSV files	Passed
2	Software Unit Testing to verify the vibrating wire piezometer and earth pressure cell sensor data can be retrieved from the Griffith University PI System	Not Completed (Refer to 4.3.2)
3	Software Unit Testing to verify network access to the KX Server web socket address that was configured for ingesting and storing the N79 building sensor data	Passed
1,3,4	Runtime Testing to verify the project system was stable, and could run continuously for an extended period without interruption	Passed
1,3	Load Testing to verify the N79 building server room computer could run the project system continuously for an extended period without any existing or expected future hardware issues being experienced on the machine	Passed
3,4	Printing the output of relevant processes used within the KX Server to visually verify that sensor data is being received by the KX Server and being stored within the KX databases	Passed
3,4	Using KX dashboards to visualize the retrieval of sensor data stored within the KX databases, with the sensor data being presented within a series of visualisations	Passed
5	Client approval of the informational dashboards that were developed during the project	Passed
5	Dashboard User Testing to verify the dashboards convey the intended information and can be used by end users for their intended purpose (monitoring the structural state of the N79 building)	Passed
5	Dashboard User Testing to verify that the dashboards developed follow recommended dashboard design guidelines	Passed
Client Signature:		