Dissertation

Supervisor: Carlos A. Pérez-Delgado
Programme: MSc Computer Science (Artificial Intelligence)
Word Count: 7,536

December 3, 2024

Acknowledgments

Abstract

List of Figures

List of Tables

List of Abbreviations

- AI - Artificial Intelligence
- CD - Class Diagram
- C-SE - Classical Software Engineering
- COBYLA - Constrained Optimization by Linear Approximation
- IBM - International Business Machines Corporation
- MSc - Master of Science
- NISQ - Noisy Intermediate-Scale Quantum
- OMG - Object Management Group
- OOP - Object-oriented programming
- POP - Procedure-oriented programming
- PUB - Primitive Unified Blocs
- RSA - Rivest-Shamir-Adleman
- SD - Sequence Diagram
- SDK - Software Development Kit
- SU(2) - Special Unitary Group of Degree 2
- Q-UML - Quantum Unified Modelling Language
- Q-SE - Quantum Software Engineering
- UML - Unified Modelling Language
- VQE - Variational Quantum Eigensolver

Contents

1 Introduction

1.1 Problem Description and Incentive

Quantum Software Engineering (Q-SE) is an emerging research field seeking to develop and standardise software engineering principles in quantum technologies. Many of these standards have been adapted from classical software engineering (C-SE), attempting to keep them as familiar as possible to C-SE whilst being able to distinguish that quantum and classical systems are two fundamentally different hardware.

One area of interest in Q-SE is the adaptation of Unified Modeling Language (UML) to model quantum systems and illustrate their communication with classical systems. Two notable papers have introduced adaptations of UML for this purpose: *"A Quantum Software Modeling Language"*[**Prez-Delgado2022**] and *"Design of classical-quantum systems with UML"*[**Prez-Castillo2022**]. Both propose methodologies for incorporating quantum technologies into UML, aiming to broaden the scope of professionals who can contribute to the Q-SE field and promote early adoption of a standardised quantum modelling language whilst quantum technologies remain in relative infancy.

The critical question is which quantum UML adaptation offers the best solution. This involves evaluating their effectiveness for modelling quantum systems, their suitability in real-world applications and their potential for widespread adoption. Additionally, should the industry favour full-scale UML modelling or another simplified modelling approach, and which of the two adaptations best accommodates an alternative design preference?

1.2 Goals and Objectives

This project aims to create and contrast UML diagrams using these two quantum UML approaches.

The first objective is to choose an appropriate real-world example as a starting point for the initial diagrams. The example chosen is the Variational Quantum Eigensolver (VQE), a hybrid quantum/classical algorithm. VQE is ideal for exploring how communication between quantum and classical machines can be represented in a modelling language, as it requires iterative interaction between a classical optimiser and a quantum circuit to find its solution.

The second objective is to choose the most suitable UML diagrams to model the algorithm and compare quantum UML methods. A sequence diagram was selected because it is one of the most commonly used UML diagrams and illustrates the communication between quantum and classical elements throughout the VQE algorithm. In addition, class diagrams were included to provide a structural view of the algorithm, offering insights into the relationships and attributes of the algorithm's components.

The third objective is to determine a robust method for comparison and analysis. This will involve combining the author's observations and following established guidance on good diagram design.

The final objective is to create multiple UML diagrams and explore other quantum applications. A key area of exploration will involve modelling fundamental quantum properties, such as entanglement and superposition, as suggested by the authors of the quantum UML literature: *"Although the scope of the paper focuses on extending UML for quantum software, it could be explored how to represent in UML other fundamental properties of quantum computers (e.g., superposition, entanglement, etc.)"*[**Prez-Castillo2022**]. This approach will enable the application of existing UML diagrams in new contexts or using additional diagram types to represent these properties.

2 Literature Review

2.1 Background

2.1.1 Q-SE 2020

The Quantum Software Engineering (Q-SE) 2020 workshop marked the first international gathering aimed at fostering a community around quantum software engineering, focusing on *"devising methods, approaches, and processes to develop software for quantum programs efficiently and to ensure their correctness"*[**QSE2020**]. Originally scheduled to be held in Seoul, South Korea, it was adapted to a virtual format due to the COVID-19 pandemic, running over two days in July 2020. This workshop was co-located with the 42nd International Conference on Software Engineering (ICSE 2020), providing a platform for interdisciplinary collaboration within quantum and software engineering communities.

At the close of Q-SE 2020's first day, Carlos A. Pérez-Delgado presented, *"Towards a Quantum Software Modeling Language"*[**Perez-Delgado2020**], a paper co-authored with Héctor G. Pérez-González. The collaboration began when Pérez-González, an expert in software engineering, approached Pérez-Delgado about submitting a paper to the Q-SE 2020 workshop. By combining Pérez-Delgado's expertise in quantum computing and quantum information with Pérez-González's background in software engineering, the duo developed the concept of Q-UML, which applies UML diagrams to quantum technologies[**Towards**].

2.1.2 UML

Unified Modelling Language (UML) is a general-purpose modelling language that is an industry standard for designing and documenting software systems. It is *"a consolidation of the best practices that have been established over the years in the use of modelling languages"*[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**].

Modelling languages use graphical or textual notation to abstract complex technical information, facilitating communication between technical and non-technical professionals involved in information systems. Just as a blueprint simplifies the construction process by visually representing a building's structure, a modelling language provides an organised framework to convey the structure and interactions of software and hardware systems that work together to perform a task[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. Their primary function is establishing a high-level language governed by a defined set of frameworks and rules[**Modelinglanguagemean**].

The Unified Modeling Language (UML) framework is rooted in the object-oriented programming (OOP) paradigm. OOP is *"a computer programming model that organises software design around data, or objects, rather than functions and logic."*[**TechTargetOOP**] Whereas Procedure Orientated Programming (POP) breaks down a task into smaller functions, executing them in a strict order [**OOPPOP**], OOP encapsulates elements in a system as classes and objects which contain data (attributes) and behaviour (methods)[**MediumOOPPOP**]. The OOP approach facilitates a more natural, modular understanding of computing tasks inspired by real-world concepts[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. As a small example, a `person` could be considered an instance (the object) of the class `Human`, inheriting shared attributes such as `Age` and `HairColour` with each instance holding a specific attribute value such as `Age = 32` and `HairColour = "blonde"`.

The development of the Object-Oriented Programming (OOP) paradigm is credited to the programming language SIMULA, which was introduced in 1962[**Simula**]. OOP quickly gained traction in the following decades, leading to the development of widely used programming languages today, such as C++, Java, and Python[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. As OOP programming languages became more popular, there was an increasing demand for modelling languages to aid in their analysis and development. This surge in demand led to the creation of numerous modelling languages, each with its own notation, resulting in confusion and compatibility issues[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. In response, the need for a unified modelling language arose, culminating in the creation of UML. Developed through collaborative efforts by experts in the field, UML aimed to integrate the best practices from various modelling languages[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. Released in 1997, UML has since undergone several revisions, with the latest version, UML 2.5.1, released in 2017 and maintained by the Object Management Group (OMG)[**OMG˙UML**].

There are 14 types of UML diagrams, classified into two main categories: structural and behavioural diagrams [**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. Structural diagrams provide a static view of a system's architecture, illustrating its components' composition and relationships. Behavioural diagrams represent the dynamic interactions between system components at runtime, demonstrating how elements communicate and evolve throughout their lifecycle.

Two of the most widely used diagrams are the sequence diagram, which falls under the behavioural category, and the class diagram, which is part of the structural category.

2.1.2.1 Sequence Diagram

Sequence diagrams belong to a sub-category of interaction diagrams, which includes communication, timing, and interaction overview diagrams[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙201**

Sequence diagrams model communication protocols between human and non-human entities. The horizontal axis represents the sequence of communication messages, while the vertical axis shows the timing of interactions. Each element in a sequence diagram is represented by a lifeline extending vertically, which may terminate if it is no longer required in the system. Messages are represented by arrows connecting elements at various stages of their lifecycle.

2.1.2.2 Class Diagram

The class diagram defines the data and object structures within a system[**Seidl˙Scholz˙Huemer˙Kappel˙Duff**
Each class is depicted as a rectangular element containing information about the attributes it holds and the operations it can perform or share. Interconnected edges represent the associations between different classes in the system, defining their specific relationships and interactions. The class diagram provides an overview of the modelled system's structural architecture.

2.1.2.3 Profile Diagram

The profile diagram is another structural diagram used to modify general-purpose UML diagrams to become domain-specific.

Class diagrams define the architecture of the 14 UML diagram types, forming the UML meta-model. In this context, each class diagram represents the structure and composition of the different UML diagram types, including itself[**Seidl˙Scholz˙Huemer˙Kappel˙Duf**

There are three methods for tailoring a general-purpose UML meta-model to a specific domain, with the appropriate approach chosen based on the required level of specificity. These methods, in order of increasing intensity, are creating an entirely new meta-model for the domain, modifying the existing UML meta-model by introducing new meta-classes and meta-associations or leveraging UML's built-in extension mechanisms—specifically, the UML profile diagram[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**].

The profile diagram uses stereotypes, a type of meta-class, to modify existing meta-classes in the meta-model. For example, the meta-classes **Association** and **Class** are part of the class diagram meta-model, while **Message** and **Lifeline** are used in the sequence diagram meta-model. Stereotypes are denoted by the tag `<<Stereotype>>`, followed by their name. These stereotypes are connected to the meta-classes they modify through an extension relationship pointing from the stereotype to the meta-class it's modifying[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**].

A stereotype can include meta-attributes or carry a note, represented as a rectangular box with dashed edges connected to the stereotype. Notes typically provide additional context or clarification in natural language. The stereotype name, meta-attributes, and notes impose constraints on the meta-class being extended, thus refining the system's structure and behaviour by introducing domain-specific constraints[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**].

2.1.3 Q-SE 2021

The second Q-SE workshop held virtually in June 2021 included a segment focused on modelling quantum systems[**QSE2021**]. During this session, Luis Jiménez-Navajas presented the paper *"Modelling Quantum Circuits with UML"*[**Prez-Castillo2021**], co-authored with Ricardo Pérez-Castillo and Mario Piattini. The paper introduced the concept of creating a UML profile diagram that could incorporate a quantum domain, using the example of an activity diagram to model a quantum circuit[**Prez-Castillo2021**].

Pérez-Castillo and Piattini further expanded on this work and, in 2022, published the paper *"Design of Classical-Quantum Systems with UML"*[**Prez-Castillo2022**] in the Springer May 2022 *Computing* journal[**Computing2022**]. They also served as editors for *"Quantum Software Engineering"*[**serrano2022quantum**], also published by Springer, which included a revision of Pérez-Delgado's Q-UML work. The follow-up paper, *"A Quantum Software Modeling Language"*[**Prez-Delgado2022**], further developed Q-UML by introducing additional pictorial elements for modelling quantum components and establishing fundamental axioms and core design principles.

"Design of Classical-Quantum Systems with UML"[**Prez-Castillo2022**] built upon earlier efforts to develop a quantum UML profile diagram by extending its application to multiple UML diagram types.

Pérez-Delgado's work with Q-UML modelled Shor's algorithm in the context of a quantum computer exhibiting quantum advantage, while Pérez-Castillo and Piattini modelled hybrid information systems. These hybrid systems, which combine classical and quantum processors, are already operational and can be executed on existing quantum hardware.

2.1.4 NISQ

As the introduction mentions, classical and quantum computers differ fundamentally in their underlying hardware. Classical computers operate in a binary, deterministic state where bits are clearly defined as 0 or 1. In contrast, quantum computers leverage the principles of quantum mechanics, using qubits instead of bits. A qubit can exist in a superposition of 0 and 1, with its final state determined probabilistically when measured.

The ability for qubits to exist in superpositions allows quantum computers to accelerate the speed at which certain types of problems can be solved exponentially[**knowledgeacademy**]. A well-known example is factoring large prime numbers, a computational task that classical computers find difficult to solve within a reasonable time frame. Peter Shor's development of Shor's algorithm demonstrates how a sufficiently powerful quantum computer could theoretically perform prime factorisation significantly faster than its classical counterpart[**Shor˙1997**][**minutephysics**]. The Rivest-Shamir-Adleman (RSA) encryption algorithm is a widely used encryption protocol which relies on classical computers' difficulty in factoring large prime numbers[**encryptionconsulting**]. A powerful enough quantum computer could pose a risk to modern-day security protocols as we currently know them.

Quantum computing aspires to reach the same level of ubiquity as classical computing while achieving quantum supremacy and quantum advantage. Quantum supremacy would enable quantum computers to solve tasks that classical computers cannot complete within a reasonable time frame[**quera**]. Quantum advantage would allow quantum systems to solve real-world problems *"faster than any classical algorithm running on any classical computer"* [**quera**]. To reach this point, quantum computers must scale up the number of qubits required for complex calculations—a critical challenge they face today.

Qubits are susceptible to environmental decoherence and noise, which can lead to errors and affect their ability to retain their quantum state[**futurumcareers**]. Despite notable progress in increasing qubit counts, significantly more qubits are needed to enable error-correction methods for fault-tolerant quantum computing required for quantum advantage and quantum supremacy.[**NewScientist**].

The Noisy Intermediate Scale Quantum (NISQ) era, representing near-term quantum computing, refers to hardware with qubit counts ranging from tens to hundreds [**NISQ**]. The quantum systems available via IBM's Qiskit Runtime Service all employ NISQ hardware [**ReleaseSummary**]. Despite the limited qubit numbers, these systems can perform computations using hybrid algorithms, which distribute tasks between quantum and classical machines. One such computation achievable through this approach is the Variational Quantum Eigensolver.

2.1.5 VQE

The Variational Quantum Eigensolver (VQE) is a hybrid quantum-classical algorithm that can be used to find the ground state, the lowest energy, of a given physical system[**wikivqe**]. VQE can aid in quantum chemistry simulations where determining the ground state of a given molecule or atom *"provides essential information about the system's stability, reactivity, and other chemical properties"*[**queragroundstate**]. VQE has many potential applications but can be considered a viable candidate for optimisation problems; it can be used to find a global or best local minimum (or maximum) solution within a search space[**vqeqiskit**][**VQEMax**].

VQE belongs to a class of near-term algorithms designed to run on NISQ devices. As the depth of a quantum circuit increases, error rates also rise due to decoherence, requiring more qubits for error correction. Near-term algorithms seek techniques for solving non-trivial problems using limited qubit count whilst negating decoherence[**Huang˙2023**]. VQE uses a shallow quantum circuit and delegates tasks between the quantum circuit and a classical optimiser to find its solution[**vqeqiskit**][**Peruzzo2014**].

The VQE algorithm contains the following core components:

Hamiltonian - A matrix that represents the total energy of a physical system[**hamiltonian**]. The eigenvalues of the Hamiltonian correspond to the system's energy levels, with the lowest eigenvalue representing the ground state[**vqeqiskit**].

Ansatz - A trial state for the Hamiltonian that represents an educated guess for finding the Hamiltonian's approximate ground state. The ansatz is a parametrised quantum circuit, with its parameters iteratively updated and the energy estimate evaluated against the Hamiltonian to find the lowest energy estimate[**ansatz**] [**wikiansatz**] [**Tutorial**].

Cost Function - A function that defines the objective of the problem, whether it involves minimisation or maximisation. The cost function executes the parametrised ansatz quantum circuit to estimate the energy of the Hamiltonian[**Tutorial**][**vqeqiskit**].

Optimiser - A classical optimiser that evaluates the output of the cost function and iteratively adjusts the ansatz parameters, seeking to find the optimal set of parameters for the problem solution[**Tutorial**][**vqeqiskit**].

The pseudocode for the VQE algorithm is as follows:

Algorithm 1 Variational Quantum Eigensolver (VQE)

```
1: Define Hamiltonian $\mathcal{H}$
2: Prepare ansatz $\mathcal{A}$ with $k$ parameters $\overrightarrow{\theta}$
3: Define cost function $\mathcal{C}$ as either minimisation ($\bigwedge$) or maximisation ($\bigvee$)
4: Initialise a classical optimizer $O$
5: while convergence criterion not met and max iterations not exceeded do
6:     Calculate expectation value $E$ by evaluating $\mathcal{A}(\overrightarrow{\theta})$ with $\mathcal{H}$
7:     Update $\overrightarrow{\theta}$ using $O$ to optimise $E$
8: return optimal parameters $\overrightarrow{\theta}$ and solution $E$
```

The VQE algorithm can be implemented and executed using quantum programming languages, with IBM's Qiskit SDK being a popular choice.

2.1.6 Qiskit

Qiskit is an open-source software development kit (SDK) created by IBM to access and utilise their cloud-based quantum computing services. Written in Python, Qiskit offers tools and libraries for quantum programming and experimentation.

IBM Quantum Learning offers online tutorials[**Tutorial**] that guide users through implementing quantum algorithms with the Qiskit SDK. The UML diagrams created for this project are based on the IBM tutorial demonstrating the VQE algorithm in Qiskit[**IBM2024**].

2.1.6.1 Qiskit and VQE

This section aims to provide the reader with a comprehensive overview of the VQE algorithm's implementation in IBM's Qiskit tutorial, helping the reader better understand the associated UML diagrams. The code from the tutorial is provided at the end of this section.

The instance **hamiltonian** is initialised from the class **SparsePauliOp** by calling the *.from_list()* method in its construction. The **hamiltonian** object is a classical representation of Pauli operators, where each operator is given as a string (e.g., "X", "Y", "Z", or "I" for identity). Pauli operators are 2×2 matrices corresponding to spin measurements along the x, y, and z axes[**DJORDJEVIC201229**]. Each operator string specifies actions on individual qubits. The VQE tutorial uses a 2-qubit system, acting on pairs of Pauli strings referred to as operator terms. The tensor product of these pairs is assigned a coefficient (represented as a complex number in Python) that defines the strength of each operator term. The linear combination of these terms represents the system's total energy, the Hamiltonian. Only non-zero operators and coefficients are stored, resulting in a sparse representation of operator terms and the Hamiltonian as a whole to reduce computational expense.

The instance **ansatz** is initialised from the class **EfficientSU2** with the number of qubits **hamiltonian** holds passed to it in its construction. The **EfficientSU2** class provides a hardware-efficient classical representation of a quantum circuit capable of creating parametrised quantum states. The circuit comprises layers of single-qubit operations along with C-NOT gates, which entangle the qubits. The Qiskit documentation defines SU(2) as *"the special unitary group of degree 2, its elements are 2×2 unitary matrices with determinant 1, such as the Pauli rotation gates"*[**EfficientSU2**], meaning that the circuit includes layers of operations that rotate the states of individual qubits, specifically using Pauli rotation gates. Each Pauli rotation gate holds a parameter which will be iteratively adjusted to find the lowest energy state of **ansatz**.

The instance **backend** is initialised from the class **QiskitRuntimeService**. The **QiskitRuntimeService** class interacts with the IBM Qiskit Runtime Service that provides cloud-based access to quantum hardware and quantum simulators. Creating an IBM account and giving a token when executing the code to access the service is necessary. The parameter **ibm_quantum** is given to **backend** during its construction to access the quantum computing platform available on the IBM cloud service. The method *least.busy()* is used to select the next available quantum hardware with the parameter **simulator** set to false to access the quantum hardware as opposed to a quantum simulator.

QiskitRuntimeService will instantiate an **IBMBackend** object which interacts with the selected quantum hardware. The attribute **target** of the **IBMBackend** object is accessed and passed as a parameter to the instance **pm** of the **StagedPassManager** class, created using the *generate_preset_pass_manager()* method. This process allows the pass manager to receive information regarding the constraints of the selected quantum hardware.

The pass manager will *"define a typical full compilation pipeline from an abstract virtual circuit to one that is optimized and capable of running on the specified backend"*[**StagedPassManager**]. The *.run()* method is executed on **pm** to transform **ansatz** to be compatible with the selected quantum hardware, with the newly transformed ansatz stored in a new variable **ansatz_isa**. The *.apply_layout()* method is then called on **hamiltonian**, with **ansatz_isa** passed as a parameter, to modify its layout to be compatible with the selected quantum hardware. The modified Hamiltonian is then stored as a new variable **hamiltonian_isa**.

A cost function method, defined as **cost_func**, is constructed to facilitate access to quantum hardware. It accepts an estimator and the components of a primitive unified bloc (PUB) as its parameters. The PUB object comprises an array of initial guesses for the parameters of the ansatz, the **hamiltonian_isa** given as a list, and the quantum circuit **ansatz_isa**. These components form a tuple and are assigned to the variable **pub**, which is given as a parameter for the estimator object's *.run()* method.

When *estimator.run()* is executed, the classical, parametrised **ansatz_isa** circuit is transpiled into a quantum circuit that is compatible with the selected quantum hardware. It is then executed on the quantum computer provided through the IBM cloud service, preparing a quantum state based on its current parameters. The **hamiltonian_isa** list, which represents the system's Hamiltonian, is applied to this prepared quantum state. This guides the estimator in computing the energy expectation value of the system being modelled through the **ansatz_isa** circuit. Once the operation is complete, the *.result()* method returns a container of PUB results[**PrimitiveResult**], with slicing used to access the estimated energy of the **ansatz_isa**, which is stored in the variable **energy**.

A dictionary named **cost_history_dict** stores the parameter, iteration, and energy estimate each time the estimator's *.run()* method is executed. Its initial values consist of a placeholder for the parameters, iteration set to zero and an empty list for the energy estimate, with each energy estimate being appended to the list per execution. Finally, the cost function method returns the value of the **energy** variable as its result after being called.

A random array of initial guess parameters, assigned to the variable **x0**, is constructed using NumPy's constant π and its *random.random()* method. The code generates an array of random floating-point numbers, scaled to the range of [0,π2], to account for every quantum state that can be represented on the Bloch sphere. The **ansatz** attribute **num_parameters** sets the size of the array, having been stored earlier in the variable **num_params**, to match the number of parameters to assign to each of the Pauli rotation gates in the ansatz circuit. The **x0** array will be given to **cost_func** as the parameters required for the **pub** variable and updated iteratively to find the set of parameters that produce the lowest energy estimate of the system.

The instance **session** is initialised from the class **Session** with **backend** passed as a parameter to configure it to the selected quantum hardware. The instance **estimator** is then initialised from the class **EstimatorV2** with **session** passed to the estimator's **mode** attribute. This estimator, which operates within **cost_func**, uses **session** to execute computations on the specified quantum backend. Assigning a **Session** object as the estimator's **mode** facilitates the grouping of iterative calls to the quantum computer[**Session**] when *estimator.run()* executes, efficiently managing the allocation of jobs to quantum resources.

The **estimator** is used to interact with Qiskit Runtime Estimator primitive service[**EstimatorV2**]. Qiskit offers two primary primitives—"Estimator" and "Sampler"—designed to simplify foundational quantum tasks[**QiskitRuntime**]. The estimator primitive is required to complete the VQE algorithm's purpose of estimating the energy of a system.

The instance **res** of the classical **Minimize** function from the SciPy package is initialised with the Constrained Optimization by Linear Approximation (COBYLA) method, which is used to minimise a scalar function. In this case, the scalar function is the **cost_func** method, which returns a scalar energy estimate as a float. The parameters for **res** include the **cost_func**, the initial guess array **x0** for the ansatz parameters, along with **ansatz_isa**, **hamiltonian_isa**, and **estimator**, all passed from **res** to **cost_func** for execution.

During the optimisation loop, **cost_func** calculates and returns the energy estimate of **ansatz_isa**, while **res** iteratively updates the parameters in **x0**, adjusting the ansatz's parameters to minimize the energy estimate. Each call to **cost_func** invokes the *estimator.run()* method and performs 10,000 shots on the quantum circuit before returning the energy estimate. The loop continues until the energy estimate converges to the lowest achievable value, representing the ground-state energy.

Upon completion of the optimisation loop, successful termination of the process is verified by comparing the solution parameter and evaluation count against the stored solution parameter and iteration count within **cost_history_dict** maintained by the **cost_func** method. These results are then visualised using the matplotlib package, plotting a graph with the number of iterations on the x-axis and the energy estimates on the y-axis.

```python
# General imports
import numpy as np

# Pre-defined ansatz circuit and operator class for Hamiltonian
from qiskit.circuit.library import EfficientSU2
from qiskit.quantum_info import SparsePauliOp

# SciPy minimizer routine
from scipy.optimize import minimize

# Plotting functions
import matplotlib.pyplot as plt

# Runtime imports
from qiskit_ibm_runtime import QiskitRuntimeService, Session
from qiskit_ibm_runtime import EstimatorV2 as Estimator

# To run on hardware, select the backend with the fewest number of jobs
    in the queue
service = QiskitRuntimeService(channel="ibm_quantum")
backend = service.least_busy(operational=True, simulator=False)

# Define Hamiltonian
hamiltonian = SparsePauliOp.from_list(
    [("YZ", 0.3980), ("ZI", -0.3980), ("ZZ", -0.0113), ("XX", 0.1810)]
)
```

```
# Create ansatz
ansatz = EfficientSU2(hamiltonian.num_qubits)
num_params = ansatz.num_parameters

# Initalise pass manager
from qiskit.transpiler.preset_passmanagers import
    generate_preset_pass_manager

target = backend.target
pm = generate_preset_pass_manager(target=target, optimization_level=3)

# Transform ansatz
ansatz_isa = pm.run(ansatz)

# Transform Hamiltonian
hamiltonian_isa = hamiltonian.apply_layout(layout=ansatz_isa.layout)

# Define Cost Function
def cost_func(params, ansatz, hamiltonian, estimator):
    """Return estimate of energy from estimator

    Parameters:
        params (ndarray): Array of ansatz parameters
        ansatz (QuantumCircuit): Parameterized ansatz circuit
        hamiltonian (SparsePauliOp): Operator representation of
            Hamiltonian
        estimator (EstimatorV2): Estimator primitive instance
        cost_history_dict: Dictionary for storing intermediate results

    Returns:
        float: Energy estimate
    """
    pub = (ansatz, [hamiltonian], [params])
    result = estimator.run(pubs=[pub]).result()
    energy = result[0].data.evs[0]

    cost_history_dict["iters"] += 1
    cost_history_dict["prev_vector"] = params
    cost_history_dict["cost_history"].append(energy)
    print(f"Iters. done: {cost_history_dict['iters']} [Current cost:
        {energy}]")

    return energy

# Initalise cost history dictionary
cost_history_dict = {
    "prev_vector": None,
    "iters": 0,
    "cost_history": [],
}

# Initalise parameters
x0 = 2 * np.pi * np.random.random(num_params)

# Run optimization routine with COBYLA
with Session(backend=backend) as session:
    estimator = Estimator(mode=session)
    estimator.options.default_shots = 10000

    res = minimize(
        cost_func,
        x0,
        args=(ansatz_isa, hamiltonian_isa, estimator),
        method="cobyla",
    )

# Verify parameters
all(cost_history_dict["prev_vector"] == res.x)

# Verify iterations
cost_history_dict["iters"] == res.nfev

# Plot convergence
fig, ax = plt.subplots()
ax.plot(range(cost_history_dict["iters"]),
     cost_history_dict["cost_history"])
ax.set_xlabel("Iterations")
ax.set_ylabel("Cost")
plt.draw()
```

2.2 Related Work

2.2.1 Q-UML

Q-UML is a methodology for integrating quantum computing into the UML standard. The paper establishes a fundamental axiom and five core design principles to consider when designing a UML quantum extension. As the axiom asserts, while the design should adhere to classical standards as closely as possible, it must acknowledge the fundamental differences between classical and quantum hardware. The design principles describe when and how to label components in a diagram as classical or quantum.

In Q-UML, quantum elements are visually distinguished using bold font for textual labels and double lines for diagrammatic components.

2.2.2 Quantum UML Profile

A quantum UML profile is an extension of UML designed explicitly for hybrid information systems incorporating classical and quantum software.

In UML, a profile is a structural diagram that extends the language to support domain-specific custom models. The quantum UML profile introduces a set of stereotypes, tagged values and constraints to represent quantum concepts.

3 Design and Implementation

3.1 Plant UML

The initial version of the VQE sequence diagram was drafted using the Plant UML plugin for PyCharm, an open-source tool that enables users to generate UML diagrams from plain text.

To create the Q-UML version of the VQE sequence diagram, a combination of Plant UML and Lucidchart was used to include the double lines and bold text necessary to distinguish quantum components. However, for creating the quantum UML profile, Plant UML alone may suffice, as it should have the tools required to extend UML to represent quantum-specific elements.

3.2 Lucidchart

Lucidchart is a web-based application that creates general-purpose diagrams, including UML diagrams, through a graphical user interface (GUI). It is used when creating Q-UML adaptations as it allows the customisation of elements to contain bold text and double lines necessary to distinguish quantum components.

The first completed diagram is a PNG file created in Lucidchart. An attempt was made to keep it to a format similar to the Plant UML output. Consistency must be considered to ensure a fair comparison between both diagrams; therefore, even if the VQE quantum UML profile diagram can be reproduced in Plant UML alone, it may need to be translated into Lucidchart to ensure this consistency is upheld.

3.3 UML Design Choice

UML offers users flexibility in choosing the level of detail required to represent system information within a diagram. It is essential to strike a balance between including sufficient information for clarity and avoiding over-complication, as UML diagrams should provide a high-level abstraction of the system. This section outlines the general formatting principles applied to all UML diagrams in this project. Subsequent sections on the VQE sequence and class diagrams will delve into specific design choices and their underlying rationale. Much of the reference material for constructing these diagrams has been sourced from the book UML @ Classroom[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**].

All UML diagrams consist of a content area populated by boxes and edges, which together form the specific design of each diagram type. An optional framing element was included in the UML diagrams for this project, enclosing these components within a boundary. The frame header displays the namespace of the system the diagram represents[**UMLElementFrame**]. For this project, the namespace "Variational Quantum Eigensolver" is used, with abbreviations indicating the type of diagram preceding it.

A sequence and class diagram have been created to model Qiskit's implementation of the VQE algorithm. Additional versions of these diagrams have been adapted to illustrate the QUML and Quantum UML Profile extensions.

3.4 VQE Sequence Diagram

The elements in the sequence diagram represent instances of classes from Qiskit's implementation of the VQE algorithm. Each instance is depicted as a rectangle, with a dashed, vertical lifeline extending downward from its creation to connect to a duplicated element at the bottom of the frame, where all the system elements are aligned. The naming convention follows the format *instance:Class*.

Message sequences are simplified where possible to illustrate the provision of an attribute from one instance to another or to illustrate an operation's execution. For example, the message *.num_qubits* passed from the **hamiltonian** to the **ansatz** could have been a sequence of messages where the **ansatz** first requests the **.num_qubits** attribute from the **hamiltonian** and the **hamiltonian** provides it as a response message, depicted by a dashed line and open arrowhead. This is more accurate; however, it would result in an over-complicated diagram of many messages.

All messages in the diagram are synchronous, pointing from a sender to a receiver. A synchronous message indicates that the message must be received before the sender can continue with any further instructions[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. For example, the instance **pm:PassManager** cannot execute its *pm.run(ansatz)* message until **backend:QiskitRuntimeService** passes its constraints and optimisation level. A continuous line and a filled triangular arrowhead depict a synchronous message.

A decision was made to omit the creation of an explicit **IBMBackend** element in the sequence diagram, opting instead to show messages invoking this object as originating from **backend:QiskitRuntimeService**. The naming convention effectively conveys that **QiskitRuntimeService** serves as a "backend" instance. Adding **IBMBackend** would introduce unnecessary complexity to the sequence diagram, where the emphasis is on message flow rather than object details. These specifics are more appropriately captured in the class diagram, where **IBMBackend** has been included.

The transformation of **hamiltonian** and **ansatz** to **hamiltonian_isa** and **ansatz_isa** is depicted by a destruction event; a red cross on the lifeline at the point where **hamiltonian** and **ansatz** are no longer used in the system.

Activation bars depict the activation of multiple elements within the diagram when an operation is executed[**creatley**]. This occurs when the pass manager executes its *run.()* method, the **hamiltonian** executes *apply_layout(layout=ansatz_isa.layout)* and the execution of both **cost_func** and **res**. Multiple elements within the system must be active to execute these operations.

The diagram uses two *loop* fragments and an *alt* fragment. The loop fragment expresses a sequence that is repeatedly executed[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**] with a boundary encompassing the messages involved in the repeated sequence. In the VQE sequence diagram, an outer loop fragment depicts the repeated exchange of messages between **cost_func** and **res** as they seek to find the lowest energy estimate. An inner loop fragment depicts the repeated runs of the quantum circuit when *estimator.run()* is executed. The upper right corner of the fragment contains a heading of the fragments label and a description of how long the process executes; the outer loop running until [**lowest energy estimate found**] and the inner loop running [**10,000 shots**]. The response message from the estimator is shown outside of this loop, as it does not return a result after each shot but once the circuit has completed 10,000 shots.

The alt fragment in UML represents alternative sequences[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. Here, it evaluates whether the results of the **cost_history_dict** match those from the completed minimisation routine. Based on the boolean outcome—true or false—it determines if the verification is successful or unsuccessful.

3.4.1 SD QUML

This diagram has been completed and is attached at the end of this report. It illustrates the sequence of messages between modules necessary to execute the VQE algorithm.

Multiple Pauli operators define the Hamiltonian "sparsely" instead of creating a complete matrix. It acts upon two qubits at a time, and the qubits count is passed to the ansatz object. The ansatz is a circuit that creates a trial state for the experiment.

These objects are transformed through a pass manager to be compatible with a quantum computer, the "backend". This process does not require communication with the quantum hardware, as the information is held classically; therefore, the diagram does not depict these modules as quantum.

A session is created and configured to the same backend, passing multiple circuits to an estimator object while the session handles resource management. This estimator is a quantum object, depicted as such in the diagram, as it utilises quantum hardware to estimate the values of the quantum circuits.

The methods `minimize` and `cost_func` contain the estimator object and are also depicted as quantum. The `cost_func` takes the Hamiltonian, ansatz, and parameters (initially a NumPy array of random values) and combines them into a new variable, `pub`, which is then given to the estimator object. The estimator executes 10,000 shots, defined during its creation, meaning the quantum circuit runs 10,000 times. This message is marked as a quantum message with **bold text**, indicating that the communication uses quantum hardware. The estimator then returns an energy estimate, converted to classical information before being received by the `cost_func`, therefore depicted as a classical message.

The `cost_func` method updates the parameters and records critical information in a `Dictionary` object per iteration. The `minimize` method, acting as an outer loop, receives the energy estimate and uses a classical optimiser (COBYLA) to guide the process into finding the lowest energy estimate.

The loop is completed once the energy estimate returned by `cost_func` converges to the lowest energy estimate, and the parameters and iterations are verified with the `Dictionary` object. The final step involves plotting each iteration and energy estimate using the Matplotlib Python library.

3.4.2 SD Quantum UML Profile

This has yet to be completed and will be the next step in the process.

3.5 VQE Class Diagram

Packages group the imported Python libraries used to execute the VQE algorithm in Qiskit. Although the Qiskit SDK is more than a library, encompassing a collection of libraries alongside other utilities[**SheriefAbul-Ezz**], it is depicted as a single package for simplicity. Packages in UML are typically used for larger, more complex software systems[**VisualParadigm**]; however, it was elected as a design choice to provide a clear visual of the required imports to run the algorithm. The design of a package element in UML resembles the structure of loop and alt fragments in sequence diagrams, grouping related elements within defined boundaries. However, unlike loop and alt fragments, which display their names in the upper-left corner inside the boundary, a package's name appears in a smaller rectangle positioned outside of the boundary. The package **NumPy** is embedded within the Qiskit package—a design choice to prevent overlapping edges rather than imply a nested structure.

The class diagram elements, depicted as rectangles, illustrate the classes used in Qiskit's VQE algorithm implementation, with each rectangle divided into three sections: namespace, attributes, and operations. A design choice was made to keep the same notation of *instance:Class* for the namespace, as used in the sequence diagram. This notation is typically used in object diagrams. Class diagrams provide a static system view and typically only include the class name. In contrast, object diagrams capture the system at runtime and use either *instance*, *instance:Class* or *:Class* as its naming convention[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. The decision to retain the instance in the notation is for clarity's sake, providing a minimal but effective indication of which class represents what aspect of the VQE algorithm by the instance name. Without an in-depth knowledge of Qiskit, it may be difficult to determine that **EfficentSU2** represents the ansatz or **SparsePauliOp** represents the Hamiltonian.

A design choice was also made to create a separate diagram for the **Minimize** class, distinct from the **CostFunction** class. Attempts to depict all class connections in a single diagram led to overlapping edges and clutter. The resulting diagram remains clear and readable by isolating **Minimize** and connecting it with simplified classes from the Qiskit package. The simplified classes are enclosed within the Qiskit package to indicate the part of the system to which they belong. The only simplified class not included is **CostFunction**, as it caused visual alignment issues. Since **CostFunction** is a user-defined method rather than a Qiskit class, it was decided to keep it outside the package in the minor diagram. Its meaning should still be evident within the complete diagram's context.

Only the attributes and methods relevant to the VQE algorithm are shown; additional attributes and methods documented in these classes have been omitted for clarity.

Some elements in the diagram, such as **Numpy.ndarray**, **CostFunction**, and **Dict**, are user-defined classes and therefore lack documentation detailing their attributes and operations. The variables and methods within these objects are presented directly as the attributes and operations of these elements in the diagram.

Each class attribute and method contains a visibility marker and it's name with the format "visbility**name:**". The visibility of all attributes and methods within the system is public, distinguished by the character +, meaning they are accessible to all other objects within the system[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. This was determined by none of the attributes or methods containing a leading underscore, which is the convention used in Python to define private attributes and methods[**Privacy**].

Class attributes contain their datatype with the format "visbility**name:** data type". Some class attributes also include multiplicities, which can be seen in the classes **SparsePauliOp**, **EfficientSU2**, **QiskitRuntimeService**, **EstimatorV2**, **Dict**, and **Pyplot**. Attribute multiplities are used in these classes to convey how many values the attribute can hold[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**], with [0...1] indicating either none or one value or [1...*] to indicate one or many values. Attributes where multiplicities are omitted are considered to have only one value. Attributes with multiplicities also have additional information as to whether the values can be duplicated with the terms "unique" or "non-unique" and whether the values must be in a fixed order with the terms "ordered" or "unordered"[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. The format for an attribute which contains all of these details is "visbility**name:** data type[multiplicity]{duplication,order}". When an attribute's data type is a tuple, such as the **pub** attribute in **CostFunction**, each item's name and data type are specified, with square brackets enclosing the entire tuple to indicate its structure. Attributes can contain multiple data types. For example, the **mode** attribute of **Estimatorv2** can be assigned other data types than just Session. The name of the data type used in the specific algorithm is used for clarity. Including attributes is optional, with some classes not requiring this information, such as **StagedPassManager** and **Session**.

Class methods are shown with either empty parentheses if no parameter information is needed or with detailed parameter information listed within the parentheses. The data type of the returned value is provided after the parentheses. The typical notation for a method with parameter information is "visbility**name**(**parameter name:** parameter data type): returned data type. For methods returning tuples—such as the *from_list()* method in **SparsePauliOp**, the *run()* method in **Estimator**, and the *res()* method in **Minimize**—each item's name and data type are specified and enclosed in square brackets, mirroring the method's format in code. The methods *plot()*, *set_xlabel()*, and *set_ylabel()* in the **Pyplot** class do not have a return data type, as they do not return a value. Instead, they update a Figure object by assigning relevant information. The final output, of type Figure, is produced when the *draw()* method is executed. Some classes, like **CostFunction** and **Minimize**, are methods themselves. These classes include methods with the class instance name, indicating that the class can be executed as an operation. Including operations is optional, with some classes not requiring this information, such as **IBMBackend** and **Dict**.

Relationships, known as associations, between classes, are depicted by the edges that join the rectangles of the diagram. Edges with a solid line and filled triangular arrowhead pointing from one class to another indicate a binary association, where one class can view the visible attributes and operations of another class, but not the other way round[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. For example, **Session** can view the attributes and operations of **QiskitRuntimeService**, but **QiskitRuntimeService** cannot view the attributes and operations of **Session**. The edges between **SparsePauliOp** and **EfficentSU2** do not have arrowheads, meaning they are bidirectional[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]; each class can view the other's attributes and operations. As we established earlier, all attributes and operations in the system are public and can technically be accessed by all objects. In the context of the VQE algorithm, the associations indicate which objects share and which objects receive information. Specifically, **SparsePauliOp** and **EfficientSU2** provide each other with information throughout the VQE algorithm. In contrast, **QiskitRuntimeService** provides information to **Session**, but **Session** does not provide information to **QiskitRuntimeService**.

The **IBMBackend** class depends on the **QiskitRuntimeService** class. This is depicted by a dashed line with an open arrowhead pointing from the dependent object to the object it depends on. The Qiskit documentation states that IBMBackend must not be instantiated directly and instead should be interacted with using the methods in **QiskitRuntimeService**[**IBMBackend**]. The **IBMBackend** class is included in this diagram as its documentation contains the **target** attribute, which must be passed to **StagedPassManager** to access information regarding the selected quantum hardware constraints. The relationship has the stereotype `<<instantiate>>` as **IBMBackend** requires **QiskitRuntimeService** for it's full implementation[**Dependencyrelationships**]

The class **Dict** is a composition of the class **CostFunction**. A composition is a binary association signifying that one class is contained as part of another class and cannot exist without it; if the aggregate(whole) object is destroyed, the contained part is also destroyed[**UMLComposition**]. The relationship is depicted on the association edge by a filled diamond attached to the aggregate class. In this case, **Dict** is an integral part of **CostFunction** as it's written inside the method, remembering that the class **CostFunction** represents a user-defined method. If **CostFunction** were destroyed, **Dict** would also be destroyed.

Shared aggregation is similar to composition in that it signifies that one class belongs to another[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**]. However, unlike a composition, the contained classes can exist outside the aggregated class. The relationship is depicted on the assocation edge by a hollow diamond attached to the aggregate class. Shared aggregation is used to depict classes that are passed as parameters to other classes in the VQE algorithm; **Numpy:ndarray**, **EfficentSU2**, **EstimatorV2**, and **SparsePauliOp** are all passed as parameters to **CostFunction** and therefore have shared aggregation. The classes along with **CostFunction** are also passed to **Minimize** as parameters and, therefore, have shared aggregation.

Some associations may be given an association name with a reading direction indicated by a filled triangular arrowhead pointing from one class to another. This arrowhead signifies the flow of the association's action, helping to clarify the direction in which information or control is passed between the classes involved. For example, the association between **StagedPassManager** and **EfficientSU2** shows that the pass manager will transform the ansatz and not the other way around. Although the reading direction points in the same direction as the binary association in this example, it does not have to[**Seidl˙Scholz˙Huemer˙Kappel˙Duffy˙2014**].

The relationships between the classes in the diagram are defined by multiplicities, which specify how many objects are involved in each association. For instance, the binary association between **StagedPassManager** and **EfficientSU2** demonstrates that one instance of **StagedPassManager** is associated with exactly one instance of **EfficientSU2**. Multiplicities in the diagram are either **1** or **1..***, with the option to label shared attributes. In the case of the **Minimize** class, it shares between one and many values of the attribute **res.x**—which contains parameters it iteratively updates—with a single **Numpy.ndarray** object. Subsequently, **Numpy.ndarray** shares only one array object with **CostFunction**. The use of multiplicities conveys a single-instance relationship between **Numpy.ndarray** and **CostFunction** while showing that **Minimize** dynamically updates and shares multiple values of **res.x** to the **Numpy.ndarray** class.

3.5.1 CD QUML

3.5.2 CD Quantum UML Profile

4 Results and Analysis

4.1 Author Observations

My initial observations of the first diagram suggest that while the bold text and double lines used to distinguish quantum components are visible, they are more easily identifiable in examples provided in the Q-UML paper. This is likely due to the larger scope of the VQE sequence diagram and the fact that it is primarily composed of classical elements. Additional design choices could be considered to better differentiate between classical and quantum components, although these would fall outside UML standards. For example, different colours could represent classical and quantum modules. This feature would only be readily available in applications like Lucidchart but could be incorporated with Plant UML by developing an extension. Once the quantum UML profile version of the VQE sequence diagram is completed, more detailed observations and comparisons between the two diagrams will be possible.

4.2 Diagram Literature

Once both diagrams are completed, it will be crucial to consult relevant literature on effective diagram design to make an informed assessment. Research papers exploring good diagram design, particularly concerning UML, should be considered. An example of such work is the paper "Improving Information System Design: Using UML and Axiomatic Design" [**CAVIQUE2022103569**], which provides valuable insights into improving system information design.

4.3 Polling

An anonymous questionnaire should be developed to present both VQE sequence diagrams to professionals and researchers in quantum computing and software engineering. Ideally, multiple UML diagrams would be included. The questionnaire should ask participants about the clarity of the diagrams, their understanding of the content being depicted, and their overall observations on the application of UML to quantum technologies. Additionally, gathering information on the industry sectors respondents are associated with would be valuable, providing insight into the diversity of perspectives within the results.

5 Conclusions

The paper will conclude with an overall assessment of the advantages and disadvantages of each method. It will address whether the primary goal and all four main objectives were achieved. Additionally, there will be a discussion on whether UML is the most suitable approach for representing quantum systems or if a more general flowchart format might be more appropriate. This will include considering which method could be used in such a case. I assume that Q-UML would remain the most applicable, as the quantum UML profile is inherently based on UML formatting.

6 Future Work

Depending on whether the fourth objective is achieved, further exploration into modelling fundamental quantum concepts would be beneficial for evaluating the effectiveness of applying UML to quantum systems. Plant UML is currently the most convenient tool for creating UML diagrams, as Lucidchart is significantly more labour-intensive. Developing an extension for Plant UML or other open-source tools like Mermaid to incorporate Q-UML formatting would be highly advantageous for future diagram creation. Such an extension would also increase the utility of this method, making it more accessible for continued exploration and broader adoption in the QSE field.

References

7 Bibliography

8 Appendix