Variational Quantum Linear Solver

Implementation in Qiskit

Bachelorarbeit in Physik

von
Alexander Cornelius Mühlhausen

**Angefertigt im Helmholtz-Institut für Strahlen- und Kernphysik
an der Rheinischen Friedrich-Wilhelms-Universität Bonn.**

**Diese Arbeit wurde im August 2021 der Mathematisch-
Naturwissenschaftlichen Fakultät der Universität Bonn vorgelegt.**

1. Gutachter: Herr Professor Dr. Carsten Urbach

2. Gutachter: Herr Professor Dr. Thomas Luu

Abstract

The *Variational Quantum Linear Solver*, an algorithm by Bravo-Prieto et al., is designed to provide, in the near future, an efficient solution to the Quantum Linear Systems Problem using *noisy intermediate-scale quantum* computers [1]. The purpose of the algorithm is to find an approximation to a vector $\mathbf{x}$ that satisfies $A\mathbf{x} = \mathbf{b}$ for a given A and $\mathbf{b}$ by minimizing a cost function [1]. The algorithm is a hybrid that utilizes both quantum and classical hardware to achieve its purpose [1]:

- The cost function's value is estimated using short-depth quantum circuits called **Hadamard Tests** [1].
- Another quantum circuit, called the **Ansatz** $V(\mathbf{alpha})$, is used to prepare a quantum state for $\mathbf{x}$ that is used in the Hadamard Tests [1].
- Based on the cost functions value and the parameters **alpha** a minimization algorithm on a classical CPU variationally alters the parameters **alpha** until the value of the cost function is minimal or boundary conditions are reached. [1].

In this thesis the functionality and operating principles as well as the quantum circuits required by the algorithm are presented, explained and discussed. Important mathematical contexts and interactions between the elements of the algorithm are provided and deduced.

Based on this foundation an implementation in the quantum Software Development Kit `Qiskit` is developed and discussed. The primary focus is the simulation of various problems to proof the algorithm's functionality and estimate its potential. The secondary focus is the execution on an IBM Quantum `Falcon` quantum processor.

The respective results show the potential of the algorithm as it can find decent solutions for different problems, both in simulation and executed on real quantum hardware. But they also show limitations concerning the implementation – both in soft- and hardware for the time being – that complicate obtaining satisfactory results for some other problems. Possible solutions are illustrated and discussed and optional improvements are outlined.

With this it is established that the algorithm and other *Variational Hybrid Quantum Classical Algorithms* like it have a lot of potential as soon as a better availability of quantum computers with a medium amount of qubits is given and some software features are more accessible or evolved. Especially the up-to-exponentially shorter runtime compared to classical computers, that Bravo-Prieto et al. found [1], combined with basic executability on present day quantum computers, holds promises for significant opportunities in the near future.

Contents

1 Introduction & Motivation

This thesis is about the *Variantional Quantum Linear Solver* algorithm (from now on: *VQLS*), first presented by Bravo-Prieto et al. in 2019, as of June 2, 2020. An implementation of VQLS in the quantum SDK `Qiskit 0.26.2` in `Python 3.9` is presented (cf. section 4) and the results of the execution on real quantum hardware are discussed (cf. section 5.3). The VQLS is a "variational hybrid quantum-classical algorithm"[1, p. 1] (VHQCA), i.e. an algorithm that

- uses quantum hardware to "employ a short-depth quantum circuit to efficiently evaluate a cost function"[1, p. 1], whose value "depends on the parameters of a quantum gate sequence"[1, p. 1].
- uses classical optimization algorithms to vary the parameters of the quantum gate sequence to minimize the cost function [1].

The purpose of VQLS is to efficiently solve problems of the form $A\mathbf{x} = \mathbf{b}$ by finding the vetor $\mathbf{x}$[1], for given matrix A and vector $\mathbf{b}$ [1]. More precisely: the VQLS algorithm uses variational optimization to find a quantum state $|x\rangle$ with $|x\rangle \approx |x_0\rangle = \mathbf{x}/||\mathbf{x}||_2$. Boundary conditions for given $|b\rangle = \mathbf{b}/||\mathbf{b}||_2$ and operator A are that the following equation holds true [1]:

$$A\,|x_0\rangle \propto |b\rangle \Rightarrow A\,|x\rangle \underset{\sim}{\propto} |b\rangle$$

It is required that A can be decomposed linearly into n_A unitary operators A_l with complex coefficients c_l

$$A = \sum_{l=0}^{n_A-1} c_l A_l.$$

1.1 Importance of the problem

Linear problems are relevant to "many areas of science and technology, including machine learning [...], solving partial differential equations [...], fitting polynomial curves [...], and analyzing electrical circuits" [1, p. 1].

Therefore, speedups in solving the underlying linear problems can have extensive benefits. On a classical computer solving a system of size $N \times N$ "scales polynomially in N" [1]. For quantum computers, however, algorithms with a logarithmic scaling in N are known (e.g. one introduced by Harrow-Hassidim-Lloyd (HHL) in 2009, [2] also for the Quantum Linear Systems Problem (QLSP)) [1]. This indicates that – at least for special problems – quantum hardware can provide up to an exponential speedup over classical systems [1, 2].

In the context of the QLSP – with A and $|b\rangle$ *sparse*[2] – the *condition number* κ is "the ratio of the largest to the smallest singular value" of A and ϵ is a *fixed precision* of the solution $|x\rangle$ (i.e. the desired precision for the solution) [1]. With these parameters, for the VQLS algorithm as a quantum-classical hybrid Bravo-Prieto et al. found indicators "of (at worst) linear scaling in κ, logarithmic scaling in $1/\epsilon$, and polylogarithmic scaling in N" [1]. One can compare this to HHLs algorithm whose runtime was originally estimated as a "polynomial of log (N) and κ" [2] and was later on (with some constraints) improved to a linear dependence on κ and polylogarithmic scaling in $1/\epsilon$ [1].
However, even though this comparison is favorable to the VQLS algorithm one has to notice that it, as a VHQCA (unlike HHL), is a "heuristic algorithm[s]" and that the above runtime estimations for the VQLS are based on numerical simulations of certain problems [1]. Nonetheless, with these results it appears plausible that the VQLS-algorithm can provide a noticable speedup compared to classical algorithms.

1.2 Advantages of VHQCAs

Unlike exclusive quantum algorithms like the one from HHL, VHQCAs have the potential of delivering tangible results on near-time *noisy intermediate-scale quantum* (NISQ) computers [1]. Bravo-Prieto et al. compared current HHL-implementations for systems of size up to 8×8 (limited by quantum noise) to their algorithm [1] and found that they could solve "a particular linear system of size 1024×1024" on present-time quantum hardware [1]. Therefore, VHQCAs might provide an efficient intermediary solution and up to an exponential speedup compared to classical algorithms in the near future – unlike noise-susceptible full-quantum algorithms that require a higher circuit depth [1].

1.3 Classification in the research field

The "variational eigenvalue solver" by Peruzzo et al. [4], dealing with eigenvalues in the context of quantum chemistry, can be seen as a predecessor or inspiration to the VQLS-algorithm introduced by Bravo-Prieto et al. [1, 4]. This paper was released in 2014 and ranks at the older end of VHQCAs referenced by Bravo-Prieto et al., which deal with topics like quantum chemistry, data compression, code compilation or metrology [1].

Many of the referenced papers were published around 2019 [1], which shows that VHQCAs are an upcoming research topic.

1.4 Implementing the VQLS in Qiskit

In this thesis the way the algorithm functions, the constituent parts and primary quantum circuits and their interaction from a mathematical point of view are presented. On this foundation an implementation in `Qiskit` will be developed and noteworthy problems will be thematized. Simulations and executions on real quantum hardware are conducted and discussed.
The complete implementation is published on github[3].

[1] i.e. the algorithm deals with the Quantum Linear Systems Problem [1].
[2] A matrix is called sparse if so many of its entries are zero or their distribution is in a way that this allows for significant computational optimization, e.g. by not storing the entries [3].
[3] `https://github.com/muehlhausen/vqls-bachelor-thesis`

2 The VQLS-Algorithm – definitions & quantum circuits

In this chapter the core aspects and elementary blocks of the VQLS-algorithm as defined in reference [1] will be presented as well as their interactions and the respective quantum circuits. In the following n_q denotes the number of global qubits (ancilliary qubits added in specific circuits not counted) and n_A denotes the number of coefficients (terms) in the decomposition of A.

2.1 Preparation of the state $|x\rangle$

The basic principle of the VQLS is to generate a quantum state $|x\rangle$, calculate a cost function and variationally change parameters **alpha** to generate a new $|x(\mathbf{alpha})\rangle$ that is a better approximation to $|x_0\rangle$.
To generate the state $|x\rangle$ from the $|0\rangle$ state, an operator V is used. This operator depends on the variation parameters **alpha** and fulfills $V(\mathbf{alpha})\,|0\rangle = |x(\mathbf{alpha})\rangle := |x\rangle$. In the context of this thesis only the so-called "fixed-structured layered Hardware-Efficient Ansatz"[1, p. 5] is used as V. The quantum circuit for this Ansatz consists of a *fixed* set of gates that can be divided into multiple, identical[4] *layers*. Each layer consists of two sublayers.

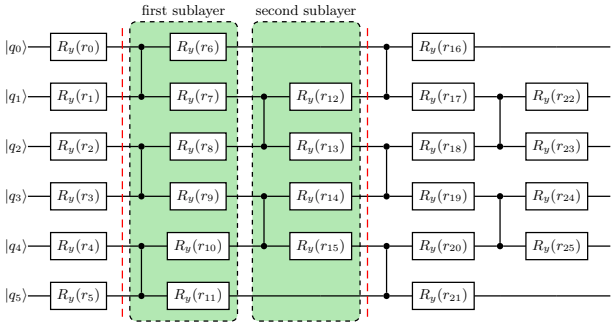

Figure 1: The fixed-structured layered Hardware-Efficient Ansatz for an even number of qubits. This is an examplary $V(\mathbf{alpha})$.
The first layer is located between the red lines, its two sublayers are marked by boxes.
This figure is based on a diagram in ref. [1].

Each sublayer consists of a) a series of controlled-Z-gates, that entangle the qubits pairwise, and b) a series of R_y-Gates, with one gate acting on each entangled qubit. The sublayers are arranged in a way that within a layer each sublayer entangles different pairs of qubits (cf. fig. 1). Each R_y-gate has its own rotation parameter r_i that is varied between the iterations. The vector **alpha** is the list of all rotation parameters: $\mathbf{alpha} = (r_0, r_1, r_2 \dots r_i \dots r_{d-1})$.

There is also a *0th layer*[5], that consists of one R_y gate acting on each qubit. It is applied first. Thus, there are $d = n_q + (n_q - 1) \cdot n_{\text{layers}} \cdot 2$ R_y-gates in total.

[4] Except for parameters.
[5] This was extrapolated from figure 3 in ref. [1, p. 5].

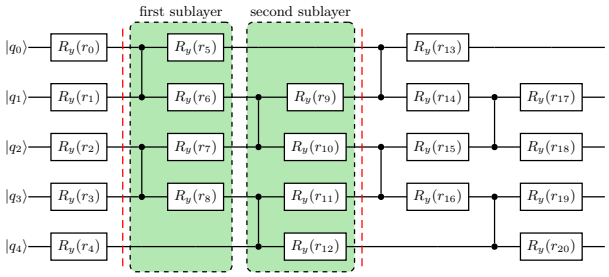

Figure 2: The fixed-structured layered Hardware-Efficient Ansatz for an odd number of qubits. This is an examplary $V(\mathbf{alpha})$.
The first layer is located between the red lines, its two sublayers are marked by boxes.
This figure is based on a diagram in ref. [1].

This Ansatz can be used to generate the desired quantum states and has the benefit that only the rotational parameters need to be modified in the variational iterations. However, it is important to notice that this Ansatz can only create real $|x(\mathbf{alpha})\rangle$, it cannot be used to prepare an complex $|x\rangle$. Thus, using this Ansatz, the algorithm only supports real $|x_0\rangle$.

Complex $|x_0\rangle$ could be supported by also adding R_x gates but this approach is not followed in this thesis as it caused problems with the used minimizer and is of limited relevance to many real-world problems.

2.2 The cost function

Another core aspect of the VQLS is the cost function that is to be minimized (ideally: $C \approx 0$). Bravo-Prieto et al. define four cost functions for the VQLS with different benefits. To define these cost functions, at first some intermediaries need to be defined:

$$|\psi\rangle := AV\,|0\rangle = A\,|x\rangle \tag{1}$$
$$U : |0\rangle \mapsto U\,|0\rangle = |b\rangle \tag{2}$$

U is an "efficient gate sequence"[1, p. 2] to prepare $|b\rangle$.

2.2.1 The local cost function

With these definitions one can now define the so-called *(normalized) "local cost function"* [1, p. 3]

$$C_L := \frac{\langle x|\,\hat{H}_L\,|x\rangle}{\langle\psi|\psi\rangle}. \tag{3}$$

Hereby $\hat{H}_L$ is the so-called *"effective Hamiltonian"* [1, p. 3]. In equation 4 the one proposed by Bravo-Prieto et al. is shown:

$$\hat{H}_L := A^\dagger U \left(\mathbb{I} - \frac{1}{n} \sum_{j=0}^{n_q-1} |0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A\,. \tag{4}$$

But applying the Hamiltonian to equation 3, expanding all terms and minimizing the resulting term proved to deliver incorrect results (cf. section 3.1 on page 7). However, a simple modification to the Hamiltonian resolves this issue:

$$\hat{H}_L := A^\dagger U \left(\frac{1}{n} \sum_{j=0}^{n_q-1} |0_j\rangle \langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A \,. \quad (5)$$

In the following the modified function from equation 5 is considered the canonical option.

In it $|0_j\rangle$ denotes "the zero state on qubit j and $\mathbb{I}_{\bar{j}}$ the identity on all qubits except qubit j"[1, p. 3]. The relevant terms can be calculated as follows:

$$\langle\psi|\psi\rangle = \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \beta_{lm} \quad (6)$$

with

$$\beta_{lm} = \langle 0| V^\dagger A_m^\dagger A_l V |0\rangle \,. \quad (7)$$

Moreover it is:

$$\langle x| \hat{H}_L |x\rangle = \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \left(\frac{1}{n_q} \sum_{j=0}^{n_q-1} \delta_{lm}^{(j)} \right) \quad (8)$$

with

$$\delta_{lm}^{(j)} = \langle 0| V^\dagger A_m^\dagger U \left(|0_j\rangle \langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A_l V |0\rangle \quad (9)$$

$$= \beta_{lm} + \langle 0| V^\dagger A_m^\dagger U \left(Z_j \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A_l V |0\rangle \,. \quad (10)$$

Z_j is the Z-Gate acting on qubit j. One can define

$$\delta_{lm} = \beta_{lm} + \frac{1}{n_q} \sum_{j=0}^{n_q-1} \langle 0| V^\dagger A_m^\dagger U \left(Z_j \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A_l V |0\rangle \quad (11)$$

and so

$$\langle x| \hat{H}_L |x\rangle = \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \delta_{lm}. \quad (12)$$

The terms β_{lm} and $\delta_{lm}^{(j)} - \beta_{lm}$ can be evaluated with the *Hadamard Test*, a quantum circuit.

2.2.2 The global cost function

Another cost function, called the *(normalized) global cost function*, can be defined as

$$C_G := \frac{\langle x| \hat{H}_G |x\rangle}{\langle\psi|\psi\rangle} = 1 - \frac{|\langle b|\psi\rangle|^2}{|\langle\psi|\psi\rangle|}$$

with the Hamiltonian

$$\hat{H}_G = A^\dagger \left(\mathbb{I} - |b\rangle \langle b| \right) A.$$

The term $|\langle b|\psi\rangle|^2$ can be calculated (and then evaluated analogously with the Hadamard Test) with:

$$|\langle b|\psi\rangle|^2 = |\langle 0| U^\dagger A V |0\rangle|^2$$

$$= \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \gamma_{lm}$$

$$= \sum_{l,m=0} c_l c_m^* \Big[\langle 0| U^\dagger A_l V |0\rangle \langle 0| V^\dagger A_m^\dagger U |0\rangle \Big]$$

Non-normalized versions of these two cost functions exist respectively. Their definitions do not include the $1/\langle\psi|\psi\rangle$ term and as such might return results near zero even though the approximation may be unprecise.

2.2.3 Short Comparison of the cost functions

In [1] it is found that the local cost function is vastly superior to the global cost function for large n_q. This is because it requires significantly less iterations of the minimizer to reach a small value of the cost function. Bravo-Prieto et al. even found that they could not "significantly lower C_G below a value of one" [1, p. 3] for $n = 50$. As this result matches the simulation results[6], especially regarding the overall minimization perfomance, the local cost function is used in the thesis.

2.3 The Hadamard Test

To calculate the cost functions the Hadamard Test can be used. The circuit is then evaluted multiple times (e.g. ten thousand times, this argument is called *shots*) with the same set of input parameters **alpha** and the ancilliary qubit is measured each time [5]. From the probabilities measured for 0 and 1 the values of $\delta_{lm}^{(j)}$, β_{lm} and γ_{lm} can be derived.

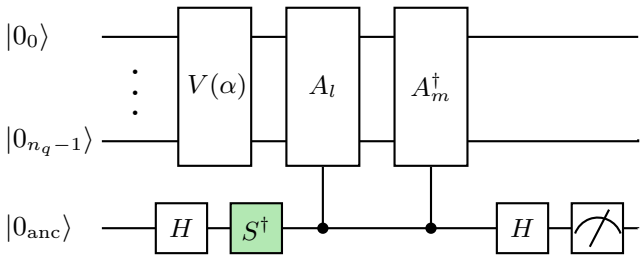

Figure 3: The circuit to calculate β_{lm}.
$|0\rangle^a$ is the ancilliary qubit. $S^\dagger$ is the adjungated phase gate and only included when the imaginary part of β_{lm} is to be obtained.
This diagram is a modified replica based on a figure shown in ref. [1].

It shall be $\xi \in \{\beta_{lm}, \delta_{lm}^{(j)} - \beta_{lm}, \gamma_{lm}\}$.

As one can see in figures 3 and 4 the Hadamard Test adds an ancilliary qubit to which a Hadamard gate is applied. If the imaginary part of ξ is wanted, an adjugated phase gate $S^\dagger$ is included. To calculate the value of a given ξ the Hadamard test is used as follows:

All gates defining ξ act on the original qubits. If a gate acts twice – once unmodified, once adjungated – and symmetrically in reference to its position (order)

[6] Both cost functions were simulated.

in ξ , it is included without modification (e.g. V, U). If it acts only once (e.g. A_l, A_m for $m \neq l$ or Z_j), it is included but as a controlled version with the ancilliary acting as the control qubit. When all gates in ξ are applied, one Hadamard gate is applied to the ancilliary qubit $|0\rangle^a$ before it is measured (allowing for 0 and 1 as results).
With $P(t) = \frac{n_t}{n_{\text{shots}}}$ it is:

$$\text{Re}(\xi) = P(0) - P(1),\ S^\dagger\text{not included} \tag{13}$$

$$\text{Im}(\xi) = P(0) - P(1),\ S^\dagger\text{included.} \tag{14}$$

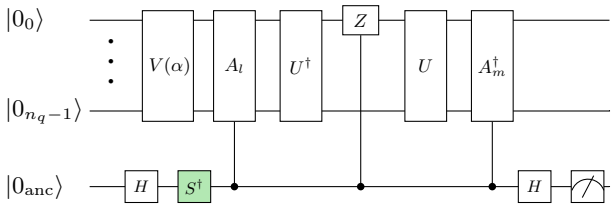

Figure 4: The circuit to calculate $\delta_{lm}^{(0)} - \beta_{lm}$.
$|0_{\text{anc}}\rangle$ is the ancilliary qubit. $S^\dagger$ is the adjungated phase gate and only included when the imaginary part is searched. Z is the Z-Gate acting on qubit 0 (thus $\delta^{(0)}$).
This diagram is a modified replica based on a figure shown in ref. [1].

It should be mentioned that Bravo-Prieto et al. define another quantum circuit, called Hadamard-Overlap-Test, that requires less control operations to calculate γ_{lm}. As this thesis focuses on the local cost function it will not be discussed.

2.4 The algorithm's steps

In the previous section all core aspects of the VQLS-algorithm were defined. The algorithm (or, more precisely, one interpretation of it) runs through the following steps to variationally find a solution:

1. Chose a random **alpha** and use it to prepare $V(\mathbf{alpha})$.
2. Use this V in the Hadamard Test circuit to calculate β_{lm}. For each combination of l and m measure n_{shots} times to calculate $\text{Re}(\beta_{lm})$. Repeat, this time with the adjungated phase gate active. Now set together β_{lm} from the real and the imaginary part.
3. Execute the Hadamard Test for $\delta_{lm}^{(j)} - \beta_{lm}$. Do this for each qubit j. For each combination of l and m iterate and sum over all j. Divide the sum by n_q and add β_{lm} to the sum. This gives δ_{lm}.
4. Calculate $\langle\psi|\psi\rangle$, $\langle x|\hat{H}_L|x\rangle$ and then calculate the cost function $C_L = \langle x|\hat{H}_L|x\rangle / \langle\psi|\psi\rangle$.
5. Insert C_L and **alpha** into the classical optimizer. It will have the following options:
   a) If a given precision is reached (the value of the cost function has fallen below a certain limit): halt and return **alpha** and C_l.
   b) If the precision cannot be improved over multiple iterations (e.g. precision limited by noise) or the maximal number of minimization iterations has been reached: halt and return results.
   c) Else: Generate a new **alpha** and prepare $V(\mathbf{alpha})$, then return to step 2. Compare with previous results to further improve the incoming results.

Use the resulting **alpha** to prepare the quantum state $V(\mathbf{alpha})|0\rangle = |x(\mathbf{alpha})\rangle \approx |x_0\rangle$.

2.5 Precision and cost functions

In their work Bravo-Prieto et al. provide a lot of information on the aspects of the fixed precision ϵ and the condition number κ. As this thesis deviates from the original algorithm in this aspect only a comprehension of the original matter will be provided.

Where the original version of VQLS minimizes the function until the given precision ϵ is reached [1], the implementation provided in this thesis makes use of a `scipy` library to minimize the cost function. This library minimizes the cost function until either a fixed number of iterations has been reached or the precision cannot be improved further (e.g. because the simulation results are too noisy). Thus, section 4 will focus on the number of simulation shots (setting the measurements precision) and the number of iterations. This corresponds to using computation time instead of precision borders [1] as the termination criterium but is only a change to the classical parts of the algorithm and does not inflict the deeper logic.

Bravo-Prieto et al. use the precision parameter γ as the termination criterium and optimize until the value of the cost function is lower than γ [1]. The following terms were used for γ [1]:

$$\gamma(C_G) = \frac{\epsilon^2}{\kappa^2 \langle\psi|\psi\rangle};\quad \gamma(C_L) = \frac{1}{n}\frac{\epsilon^2}{\kappa^2 \langle\psi|\psi\rangle}$$

Introducing a trace distance for a Hermitian M to provide another option to interpret and assess ϵ [1]:

$$D(M) = |\langle x|M|x\rangle - \langle x_0|M|x_0\rangle|$$

$$\epsilon \geq \frac{D(M)}{2||M||}$$

Therefore, this alternative provides one with the means to select a precision based termination condition for the minimizer: γ. Furthermore, the meaning of the precision parameter ϵ is clarified.

3 Mathematic proofs and deductions

This chapter covers important mathematical aspects of the algorithm and will deduce relevant terms. Interactions within the local cost function, especially the steps from equations 3 to 10, and a mathematical explanation of the Hadamard Test will be featured.

3.1 Mathematics of the local cost function

To understand the local cost function it is important to examine the difference between the Hamiltonian from the paper and the modified version. Furthermore, an understanding of the transformations of the coefficients is required.

Starting with the steps from equation 9 on page 5 to equation 10. According to ref. [6] (almost the same approach is presented in ref. [1] too):

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = |0\rangle\langle 0| - |1\rangle\langle 1|$$

Thus one can rewrite:

$$\begin{aligned}
\delta_{lm}^{(j)} &= \langle 0| V^\dagger A_m^\dagger U \left(|0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}}\right) U^\dagger A_l V |0\rangle \\
&= \langle 0| V^\dagger A_m^\dagger U \Big(\left(Z_j + |1_j\rangle\langle 1_j|\right) \otimes \mathbb{I}_{\bar{j}} \Big) U^\dagger A_l V |0\rangle \\
&= \langle 0| V^\dagger A_m^\dagger U \left(Z_j \otimes \mathbb{I}_{\bar{j}} + \mathbb{I}\right) U^\dagger A_l V |0\rangle \\
&= \langle 0| V^\dagger A_m^\dagger U \left(Z_j \otimes \mathbb{I}_{\bar{j}}\right) U^\dagger A_l V |0\rangle \\
&\quad + \langle 0| V^\dagger A_m^\dagger U \mathbb{I} U^\dagger A_l V |0\rangle \\
&\underset{U^\dagger U = \mathbb{I}}{=} \langle 0| V^\dagger A_m^\dagger U \left(Z_j \otimes \mathbb{I}_{\bar{j}}\right) U^\dagger A_l V |0\rangle + \beta_{lm}
\end{aligned}$$

Continue with $\langle x| \hat{H}_L |x\rangle$ and eq. 4. Rewrite the latter:

$$\begin{aligned}
\hat{H}_L &= A^\dagger U \left(\mathbb{I} - \frac{1}{n_q} \sum_{0}^{n_q-1} |0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A \\
&= A^\dagger U \mathbb{I} U^\dagger A - A^\dagger U \left(\frac{1}{n_q} \sum_{j=0}^{n_q-1} |0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A \\
&= A^\dagger A - A^\dagger U \left(\frac{1}{n_q} \sum_{j=0}^{n_q-1} |0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A
\end{aligned}$$

Therefore it is:

$$\begin{aligned}
&\Rightarrow \langle x| \hat{H}_L |x\rangle \\
&= \langle x| A^\dagger A |x\rangle - \\
&\quad \langle 0| V^\dagger A^\dagger U \left(\frac{1}{n_q} \sum_{j=0}^{n_q-1} |0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A V |0\rangle \\
&= \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \beta_{lm} \\
&\quad - \langle 0| V^\dagger A^\dagger U \left(\frac{1}{n_q} \sum_{j=0}^{n_q-1} |0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A V |0\rangle
\end{aligned}$$

One can move the sum in the latter term:

$$\begin{aligned}
&\langle 0| V^\dagger A^\dagger U \left(\frac{1}{n_q} \sum_{j=0}^{n_q-1} |0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A V |0\rangle \\
&= \frac{1}{n_q} \sum_{j=0}^{n_q-1} \langle 0| V^\dagger A^\dagger U \left(|0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A V |0\rangle \\
&= \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \frac{1}{n_q} \sum_{j=0}^{n_q-1} \delta_{lm}^{(j)}
\end{aligned}$$

Thus – according to the equation for $\hat{H}_L$ stated in the paper – it is:

$$\begin{aligned}
&\langle x| \hat{H}_L |x\rangle \\
&= \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \Big(\beta_{lm} - \frac{1}{n_q} \sum_{j=0}^{n_q-1} \delta_{lm}^{(j)} \Big) \\
&= \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \Big(\beta_{lm} - \\
&\quad \frac{1}{n_q} \sum_{j=0}^{n_q-1} \left(\langle 0| V^\dagger A_m^\dagger U \left(Z_j \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A_l V |0\rangle + \beta_{lm} \right) \Big) \\
&= - \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \Big(\\
&\quad \frac{1}{n_q} \sum_{j=0}^{n_q-1} \langle 0| V^\dagger A_m^\dagger U \left(Z_j \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A_l V |0\rangle \Big)
\end{aligned}$$

But, if one minimizes the cost function using this last term, the results were non-satisfactory. If instead the following equation – that corresponds to the global cost function proposed by Bravo-Prieto – is used in the local cost function, adequate results are achieved:

$$\langle x| \hat{H}_L |x\rangle = \sum_{l=0}^{n_A-1} \sum_{m=0}^{n_A-1} c_l c_m^* \left(\frac{1}{n_q} \sum_{j=0}^{n_q-1} \delta_{lm}^{(j)} \right)$$

This term is found if the Hamiltonian is reduced to the following form and then inserted into $\langle x| \hat{H}_L |x\rangle$:

$$\hat{H}_L = A^\dagger U \left(\frac{1}{n_q} \sum_{0}^{n_q-1} |0_j\rangle\langle 0_j| \otimes \mathbb{I}_{\bar{j}} \right) U^\dagger A \tag{15}$$

Thus, this slightly modified Hamiltonian from equation 15 is considered canonical in the context of this thesis[7].

[7] It is assumed that a small typing mistake occured in the paper and thus one term more than neccessary was included.

3.2 Mathematics of the Hadamard Test

In this short subchapter it shall be derived how and why the Hadamard test works in general. This deduction is almost identical to and taken from the one presented in reference [5].

In general the Hadamard circuit used to calculate $\langle\phi|G|\phi\rangle$ takes the form shown in figure 5. This is similar to e.g. figure 3 as $V(\mathbf{alpha})$ is absored in $|\phi\rangle$ as it transforms $|0\rangle$.

It is important to notice that the control operation is commutative (for unitaries): applying control(B), control(C) and control(D) after each other is identical to applying control(BCD) = control(G)[8]. So it is appropriate to consider a single gate G in the following calculations.

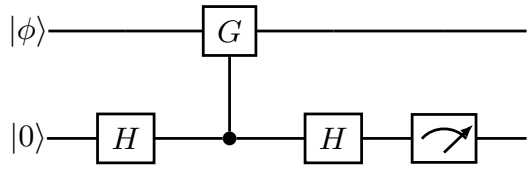

Figure 5: The circuit for a general Hadamard circuit to measure $\langle\phi|G|\phi\rangle$.
This circuit is a modified replica of a figure shown in ref. [5].

Applying the first Hadamard Gate to the ancilla results in the following quantum state:

$$(H|0\rangle)\otimes|\phi\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}\otimes|\phi\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle\otimes|\phi\rangle + |1\rangle\otimes|\phi\rangle\right)$$

The next step is to apply the controlled unitary G:

$$\underset{\text{Controlled } G}{\Rightarrow} \frac{1}{\sqrt{2}}\left(|0\rangle\otimes|\phi\rangle + |1\rangle\otimes G|\phi\rangle\right)$$

And then the second Hadamard gate:

$$\begin{aligned}\underset{\text{ancilla } H}{\Rightarrow} \frac{1}{2}\big[&\left(|0\rangle\otimes|\phi\rangle + |1\rangle\otimes|\phi\rangle\right)\\ &+\left(|0\rangle\otimes G|\phi\rangle - |1\rangle\otimes G|\phi\rangle\right)\big]\\ \Leftrightarrow\ \frac{1}{2}\big[&|0\rangle\otimes(\mathbb{I}+G)|\phi\rangle\\ &+|1\rangle\otimes(\mathbb{I}-G)|\phi\rangle\big] =: |\Phi\rangle\end{aligned}$$

Eventually one is interested in $P(0) = P(|0\rangle) = |\langle 0|\Phi\rangle|^2$ and $P(1) = P(|1\rangle) = |\langle 1|\Phi\rangle|^2$ to calculate $\mathrm{Re}(\langle\phi|G|\phi\rangle)$:

$$\begin{aligned}P(|0\rangle) =& \frac{1}{4}\langle\phi|\left(\mathbb{I}+G\right)\left(\mathbb{I}+G^\dagger\right)|\phi\rangle\\ =& \frac{1}{4}\langle\phi|\left(2\mathbb{I}+G+G^\dagger\right)|\phi\rangle\\ \Rightarrow P(|0\rangle) =& \frac{1}{4}\big[2+\langle\phi|G^\dagger|\phi\rangle+\langle\phi|G|\phi\rangle\big]\\ =& \frac{1}{4}\big[2+\langle\phi|G|\phi\rangle^*+\langle\phi|G|\phi\rangle\big]\\ =& \frac{1}{2}\big[1+\mathrm{Re}(\langle\phi|G|\phi\rangle)\big]\end{aligned}$$

One can analogously show:

$$\begin{aligned}P(|1\rangle) =& \frac{1}{2}\big[1-\mathrm{Re}(\langle\phi|G|\phi\rangle)\big]\\ \Rightarrow P(|0\rangle) - P(|1\rangle) =& \mathrm{Re}(\langle\phi|G|\phi\rangle)\end{aligned}$$

If an adjungated phase shift gate [6] is applied to the ancilla after the Hadamard Gate and before the controlled gate G is applied [1], the state is altered to:

$$\begin{aligned}\underset{S^\dagger}{\Rightarrow}\ & \frac{1}{\sqrt{2}}\left(|0\rangle\otimes|\phi\rangle + e^{-\frac{i\pi}{2}}|1\rangle\otimes|\phi\rangle\right)\\ \underset{\text{Controlled } G}{\Rightarrow}\ & \frac{1}{\sqrt{2}}\left(|0\rangle\otimes|\phi\rangle + e^{-\frac{i\pi}{2}}|1\rangle\otimes G|\phi\rangle\right)\\ \underset{\text{ancilla } H}{\Rightarrow}\ & \frac{1}{2}\big[|0\rangle\otimes\left(\mathbb{I}+e^{-\frac{i\pi}{2}}G\right)|\phi\rangle\\ &+|1\rangle\otimes\left(\mathbb{I}-e^{-\frac{i\pi}{2}}G\right)|\phi\rangle\big]\\ \Rightarrow P_{S^\dagger}(|0\rangle) =& \frac{1}{4}\langle\phi|\left(\mathbb{I}+e^{-\frac{i\pi}{2}}G\right)\left(\mathbb{I}+e^{\frac{i\pi}{2}}G^\dagger\right)|\phi\rangle\\ =& \frac{1}{4}\langle\phi|\left(2\mathbb{I}+e^{-\frac{i\pi}{2}}G+e^{\frac{i\pi}{2}}G^\dagger\right)|\phi\rangle\\ \underset{e^{-\frac{i\pi}{2}}=-i}{\Rightarrow} =& \frac{1}{4}\big[2-\langle\phi|iG^\dagger|\phi\rangle-\langle\phi|iG|\phi\rangle\big]\\ =& \frac{1}{4}\big[2+\langle\phi|iG|\phi\rangle^*-\langle\phi|iG|\phi\rangle\big]\\ =& \frac{1}{2}\big[1+\mathrm{Im}(\langle\phi|G|\phi\rangle)\big]\end{aligned}$$

And as such it is shown that – if the adjungated phase shift gate is included according to figure 3 – the imaginary part can be calculated alike:

$$P_{S^\dagger}(|0\rangle) - P_{S^\dagger}(|1\rangle) = \mathrm{Im}(\langle\phi|G|\phi\rangle).$$

[8] This can be shown mathematically or in simulations.

4 How to implement VQLS in Qiskit

For this thesis an implementation of the VQLS-algorithm in the `Qiskit` quantum SDK (version `0.26.2`) was developed. The code was designed with the following guidelines in mind:

1. A high-level, abstract approach to code design is chosen to clearly feature the structure and logic of the algorithm instead of a small set of examples.
   This benefits simulations – this chapters focus – and enables the simulation of a broad range of problems but causes problems when executing the code on real quantum hardware (cf. section 4.7 on page 16). Nonetheless, with some modifications the code discussed here can be executed on quantum hardware.
2. As much of the code as possible was organized as functions to improve the code clarity. Helper functions, that are not strictly a part of the VQLS-algorithm but provide useful features, are indicated by a preceding underscore, e.g. `_format_alpha()`.
3. Almost all user given data – e.g. the number of shots, that the simulator shall use, or the coefficients and decomposition for A – is stored in the **class** `GlobalParameters`. An **object** of this class, `params = GlobalParameters()`, is created globally and used by all functions to access this data.
   Some helper functions are provided by the class as they are only required inside the class.
4. If possible, original Qiskit functions and classes are used as those provide almost all the needed features and are well optimized.

Almost the complete code is self-written. Some good practices and low-level inspirations are taken from the Qiskit Tutorial on the VQLS-algorithm [5], from the qiskit textbook [6] or from the IBM Quantum ressources [7], e.g. code snippets to simulate or execute quantum circuits (cf. section 4.3) or using the library `scipy.optimize.minimize`. Furthermore, the Qiskit documentation [6] proved to be a very useful source for powerful classes and objects, e.g. the `Operator` class, and contained information on how to implement them. However, all higher-level functions are a self-written adaption of the algorithm as described in ref. [1].

The complete source code for this thesis is available on Github[9]. The code structure and logic (as shown in figure 6) as well as all the parameters provided by the class `GlobalParameters` via the object `params` will be discussed here on a level required for a general understanding. However, not every single last detail of the implementation will be explained. The focus of the explanations is the simulation of the algorithm but the execution on quantum hardware is also discussed.

4.1 Required Libraries

The following libraries and modules are required as they provide the simulation backend and more.

[9] `https://github.com/muehlhausen/vqls-bachelor-thesis`

```python
from qiskit import (
    QuantumCircuit, QuantumRegister,
    ClassicalRegister, Aer, IBMQ,
    execute, transpile, assemble  )
from qiskit.circuit import Gate, Instruction
from qiskit.quantum_info.operators import (
    Operator  )
from qiskit.extensions import (
    ZGate, YGate, XGate, IGate  )

from scipy.optimize import minimize

from typing import (
    List, Set, Dict, Tuple, Optional, Union  )
import random
import numpy as np
import cmath
```

The Qiskit imports are used to build and simulate or exectute (`IBMQ`) the quantum circuits for the various Hadamard Tests and the Ansatz. `Gate` and `Operator` are used to decompose A (`qiskit.extensions` are used in the same context).

`scipy.optimize` provides a library of classical minimizers to choose from. This module effectively provides the part that – outside of simulations – is executed on a CPU instead of a QPU.

The import from `typing` is required for type annotations (`name: type`) to improve the code's readability. `Union` effectively provides a logical *or* to support multiple types. `numpy`, `cmath` and `random` are used for a mathematical background structure in `Python 3.9`.

4.2 The class: `GlobalParameters`

The class `GlobalParameters` stores almost all data the user might want to input (specific to the problem at hand) and as such can be used to control the behaviour of the algorithm. The parameters accessible to or interesting for the user are described in the next paragraph. An instance of the class, `params`, is provided and imported as a globally accessible object and then referenced by all other functions. For example, the call `params.n_qubits` gives the number of qubits n_q. The classes code is published on github as well.

4.2.1 Parameters provided via `params`

The following parameters are provided by the class:

- `n_qubits: int`
  Number of qubits, n_q.
- `n_layers: int`
  Number of layers in the Ansatz.
- `n_sublayers: int`
  `n_sublayers = 1 + 2 * n_layers`

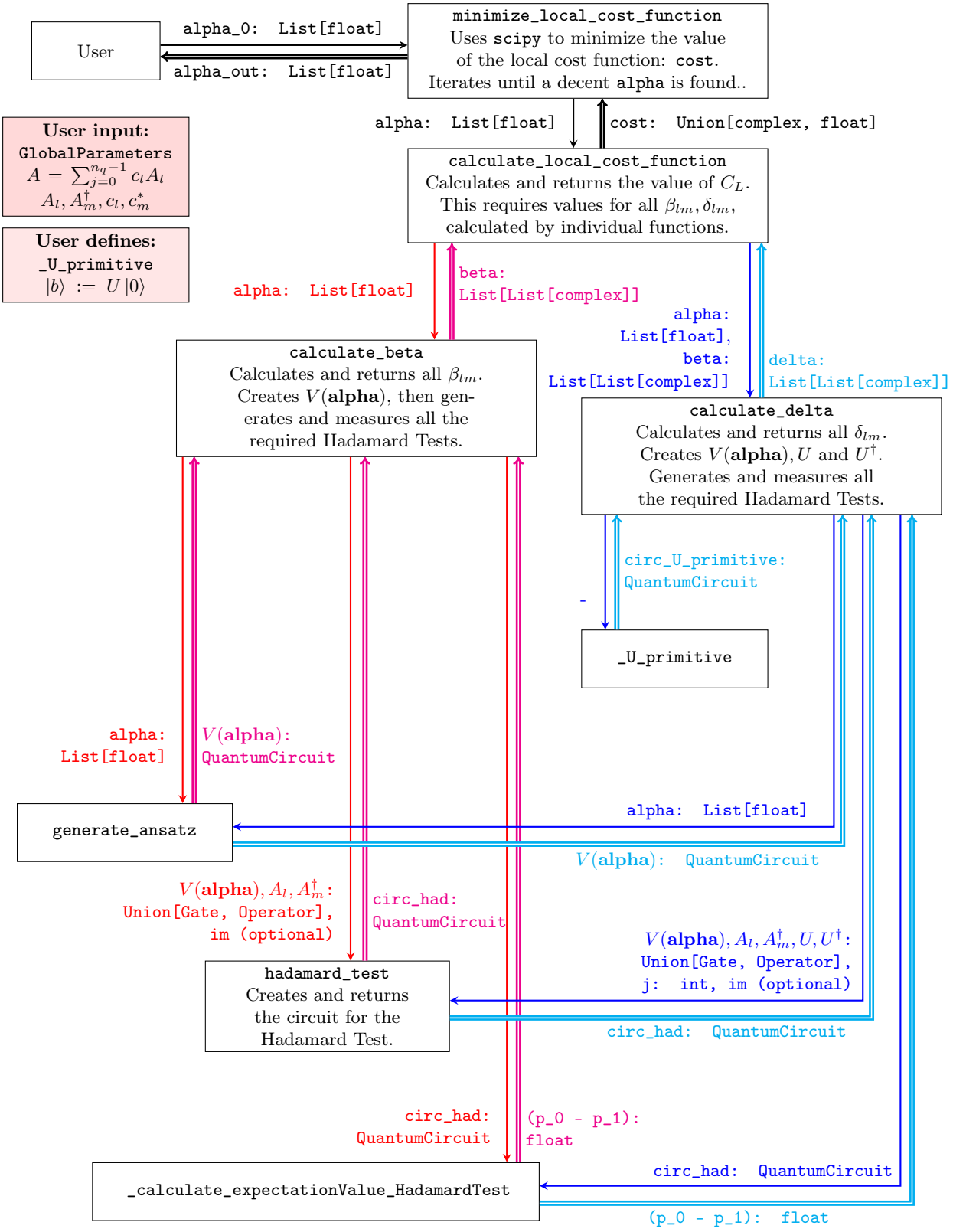

Figure 6: Code diagram for the given implementation of the VQLS-algorithm.
Single arrows mark function **calls/input**, double Arrows mark function **returns**.
Time proceeds from top to bottom, i.e. `calculate_beta` is called before `calculate_delta` is called. However, all function calls of `calculate_beta` are executed, before `calculate_delta` is executed.
Only helper function calls required for the comprehension of the algorithms structure are shown.

- `alpha_0: List[float]`
  A randomly initialized list of length d.
  `d = n_qubits + (n_qubits - 1) * 2 * n_layers`.
  Each randomly chosen element is in $[0, 2\pi)$ and used as a parameter r_i for an R_y-Gate.
  Specific implementation based on ref. [5].
- `coefficients: List[complex]`
  List of length n_A. Stores the coefficients c_l used in the decomposition of A. It is required that the resulting $A = c_l A_l$ is unitary.
- `coefficients_conjugate = \`
  `[np.conjugate(coeff) for`
  `coeff in coefficients]`
  Store the c_l^* in another list of length n_A.
- `decomposition_asGate: List[Gate]`
  The elements are the A_l of the decomposition as instances of the `qiskit.circuit.Gate` class.
  An object of the `Gate` class can be generated by (e.g.) executing the `.to_gate()` subroutine of objects of the class `qiskit.circuit.QuantumCircuit`.
  Cf. helper function `_decompose_asGate`.
  The list has length n_A.
- `decomposition_asOperator: List[Operator]`
  Its elements are the A_l of the decomposition as an instance of the `qiskit.quantum_info.Operator` class.
  Cf. helper function `_decompose_asOperator`.
  This list has length n_A as well.
- `decomposition_adjoint_asOperator = \`
  `[op.adjoint() for op`
  `in decomposition_asOperator]`
  List of length n_A. Its elements are the $A_m^\dagger$ as instances of `qiskit.quantum_info.Operator`, as this class allows for its objects to be easily adjointed.
  Alternative for quantum hardware: `decomposition_adjoint`.
- `A: Operator`
  `A` = $\sum_{l=0}^{n_A-1} c_l A_l$. The A_l are elements of the list `decomposition_asOperator`.
- `COBYLA_maxiter: int`
  Upper border for the iterations the minimization algorithm may use to find an approximation for $|x\rangle$.
  Reference value: 200 to 500.
- `qiskit_simulation_backend: str`
  This is an element of `['statevector_simulator', 'qasm_simulator']`. This parameter selects the backend used by Qiskit to simulate quantum circuits.
  Alternative for quantum hardware: `IBMQ_backend`.
- `qiskit_simulation_shots: int`
  Selects the amount of shots the simulator uses to measure the individual probabilities $P(0)$ and $P(1)$ in the Hadamard test.
  Reference value: 10^3 to 10^4.
  Alternative for quantum hardware: `IBMQ_shots`.
- `IBMQ_TOKEN: str`
  Input the token to access the IBM Quantum cloud.

Some of the above parameters (e.g. `alpha_0`, `A` or `n_sublayers`), that are provided by the class to the rest of the code, are automatically calculated upon initialization. Others are based on the input that as given to the `__init__` subroutine of `params`. User input **for simulations** is possible for the following parameters:

`n_qubits, n_layers, coefficients, COBYLA_maxiter, qiskit_simulation_shots, qiskit_simulation_backend`

Furthermore, the user has to select which gates form the decomposition of A. This is done by modifying the code of the class itself as the user might want to add helper functions to create the decomposition.

4.2.2 Class-internal helper functions

Some helper functions are implemented inside the class as they only prepare lists (e.g.). Either a possible implementation or a short documentation and implementation specifications are given here.

The first internal helper function, `_decompose_asGate`, is used to create a list of standard gates as `qiskit.circuit.Gate` objects (i.e. `List[Gate]`). The function output can be, e.g., $[X(0), X(1), X(2), \ldots, X(n_q - 1)]$. So, by mixing multiple function calls, different decompositions can be generated and stored in `decomposition_asGate`.

Another helper function, `_decompose_asOperator`, returns a `List[Operator]` of length n_q. It requires an Operator `Op` as input, generated by (e.g.) `Op = Operator(ZGate())`. It uses tensor products to generate, for example, $Z(i)$, the Z-Gate acting on qubit i. Each element is the operator acting on the qubit that resembles the list index. This function can be used to generate the list `decomposition_asOperator` (e.g. by calling it multiple times and mixing the results).
It is important to remember that an operator applied from the left side in a tensor product ($Op \otimes \ldots$) is acting on the last qubit, one applied from the right side ($\cdots \otimes Op$) is acting on the first qubit (and so on for all qubits inbetween). Note: `Op.tensor(Id)` $= Op \otimes \mathbb{I}$.

```python
def _decompose_asOperator(self, Op: Operator):
    Id = Operator(IGate())
    dec_asOperator: List[Operator] = []

    # Operator acting on the last qubit
    last = Op.copy()
    for _ in range(self.n_qubits - 1):
        last = last.tensor(Id)
    # now it is:
    # last = Op ⊗ 𝕀 ⊗ 𝕀 ⊗ ...
    dec_asOperator.insert(0, last)

```

```
    # acting on qubits 0 to n_q-2
    for x in range(self.n_qubits - 1):
        i = x+1
        temp = Id
        for j in range(i-1):
            temp = temp.tensor(Id)
        temp = temp.tensor(Op)
        for j in range(self.n_qubits - i - 1):
            temp = temp.tensor(Id)
        # temp is now of the form
        # I⊗...⊗I⊗Op⊗I⊗...⊗I
        dec_asOperator.insert(0, temp)
    return dec_asOperator
```

4.3 The global helper functions

The following *global* helper functions implement handy side features or are an example on how to solve specific problems. Either a possible implementation reference or a documentation and implementation specification are given. Full code (with minor differences) can be found on Github[10]. These helper functions are used all over in the code by various methods.

The function `_format_alpha(alpha_unformated)` is used to reorganize a vector of the length d (that is expected to be of type `List[float]`) into sublists (i.e. `List[List[float]]`). This simplifies the Ansatz.
It **returns** a list of the following form:
[[0th sublayer], [1st sublayer], [2nd sublayer], ...]
Thus for, e.g., 4 quits and 2 layers, it will return:
$\Big[[r_0, r_1, r_2, r_3], [r_4, r_5, r_6, r_7], [r_8, r_9],$
$[r_{10}, r_{11}, r_{12}, r_{13}], [r_{14}, r_{15}]\Big]$

`_calculate_expectationValue_HadamardTest(circ)` is used to simulate *(or execute)* a given quantum circuit. Its **input** `circ: QuantumCircuit` is a quantum circuit, that implements the Hadamard Test and has one classical bit for measurements.
It will use the `statevector_simulator` or the significantly faster `qasm_simulator` *(or the* `IBMQ_backend`*)*.

The code, including e.g. transpiling and assembling the circuit for the `statevector_simulator`, is based on best practices and code snippets taken from references [6] and [5] *(and [7])*. The helper function will then measure the circuit `qiskit_simulation_shots` *(or* `IBMQ_shots`*)* times and count the occurrences of 0 and 1 in the result, delivering probabilities $P(0), P(1)$. It **returns** $(P(0) - P(1))$.

4.3.1 Implementation of U with $U|0\rangle := |b\rangle$

The last helper function is `_U_primitive`. Though, strictly speaking, it is rather part of the actual algorithm than a true ancilliary function. Its purpose is to provide an "efficient gate sequence" [1, p. 2] to prepare $|b\rangle$. However, the "`primitive`" in the name is derived from the way how this helper function achieves this:
It does **not** generate a gate sequence for a given vector $|b\rangle$. Instead it generates a state $|x_0\rangle$ (e.g. by applying some Hadamard Gates to $|0\rangle$) and then applies A to this state. Thus, this function **defines** a $|b\rangle$ as $A|x_0\rangle$ and *knows* the true solution $|x_0\rangle$.
This is not a problem as this thesis purpose is to demonstrate the potential of the VQLS-algorithm. The algorithm can still work in a proper way, even if a subroutine already *knows* the answer. The algorithm still executes all steps to find a good approximation $|x\rangle$. Thus, this ancilliary function can be used for demonstrational purposes and needs to be replace for *real* problems. However, this is beyond the scope of this thesis.

Below, one can find an example for a *possible* implementation of `_U_primitive` **for simulations** defining a specific $|x_0\rangle$. In this example three Hadamard gates were chosen. The function returns a `QuantumCircuit` object that, applied to $|0\rangle$, implements $|b\rangle$.

```
def _U_primitive() -> QuantumCircuit:
    qr_U_primitive = \
        QuantumRegister(params.n_qubits)
    circ_U_primitive = \
        QuantumCircuit(qr_U_primitive)

    # define |x_0⟩ by applying
    # Hadamard gates on qubits
    # 0 to 2 (example).
    circ_U_primitive.h(0)
    circ_U_primitive.h(1)
    circ_U_primitive.h(2)

    # avoid problems due to copy
    # by reference or inconsistent
    # usage of Operators/Gates
    A_copy = params.A.copy()
    if isinstance(params.A, Operator):
        A_copy = A_copy.to_instruction()
    circ_U_primitive.append(A_copy,
                            qr_U_primitive)

    return circ_U_primitive
```

4.4 Implementation of the algorithm

In this subchapter the primary functions are defined.

4.4.1 Implementation of the minimizer

`minimize_local_cost_function` is the only function the user has to call (besides initializing `params`). It calls all other functions in the correct order.
This function implements `scipy.optimize.minimize` (based on references [5, 8]). Its **input** `alpha_0: List[float]` is a user-given initial guess for **alpha** (i.e.

[10] https://github.com/muehlhausen/vqls-bachelor-thesis

e.g. a randomly initialized list). It **returns** `alpha_out`, also a `List[float]`, a best guess for **alpha** that fulfills:

$$AV(\textbf{alpha\_out}\,|0\rangle) \underset{\sim}{\propto} |b\rangle$$

In the code shown in figure 7 only the `COBYLA` method is implemented. This method, that is also recommended in [5], proved to be by far the best method for this sort of problem. It significantly outperformed multiple other methods like `Nelder-Mead` or `Powell` for various VQLS-problems. A version of this code supporting different methods is published on github.

The `scipy` function starts with the list `alpha_0` and uses it as input for `calculate_local_cost_function`. Based on the resulting cost function values it varies the entries of `alpha` and then executes `calculate_local_cost_function` with this new, variationally changed `alpha` and searches for a minimum. It stops upon exceeding `COBYLA_maxiter` iterations or if it is unable to improve the perfomance.

Therefore, this function implements the minimization that - outside of simulations - would be executed on a classical CPU, whereas (most of the parts of) `calculate_local_cost_function` would be executed on quantum hardware [1].

When the minimization algorithm has finished the complete result (including metadata like, e.g., the number of iterations) is printed and the vector **alpha** that delivers the best solution is returned.

```
1 def minimize_local_cost_function() -> List[float]:
2     min = minimize(calculate_local_cost_function, x0=params.alpha_0, method='COBYLA'
3                    options={'maxiter': params.COBYLA_maxiter})
4      print(min)
5      alpha_out = min['x']
6      return alpha_out
```

Figure 7: The source code for the minimizer: `minimize_local_cost_function`.

4.4.2 Implementation of the local cost function C_L

The *real magic* happens within the function `calculate_local_cost_function`, that **returns** the **absolute of**[11] the value of the local cost function C_l, `cost: float`. Its **input** is the variationally altered `alpha: List[float]`. Recall equations 3, 6 and 12

$$C_L := \frac{\langle x|\,\hat{H}_L\,|x\rangle}{\langle\psi|\psi\rangle}$$

$$\langle\psi|\psi\rangle = \sum_{l=0}^{n_A-1}\sum_{m=0}^{n_A-1} c_l c_m^* \beta_{lm}$$

$$\langle x|\,\hat{H}_L\,|x\rangle = \sum_{l=0}^{n_A-1}\sum_{m=0}^{n_A-1} c_l c_m^* \delta_{lm}.$$

As one can see, the values of β_{lm} and δ_{lm} are required for all combinations of l, m with $c_l \neq 0, c_m^* \neq 0$. Thus, all of these β_{lm} and δ_{lm} are calculated and then saved in the lists `beta` and `delta` respectively. The lists type is `List[List[complex]]`. Each list is of length n_A.

Each item of the top-level lists `delta` and `beta` is another list of length n_A (type: `List[complex]`). Each of these lower-level lists represents one specific `l`. The items within each of these lower-level lists are then sorted by `m`.
This means that `beta[2]` is the list in which all combinations of $\beta_{2,m}$ are stored and `beta[2][3]` is $\beta_{2,3}$.

The calculation of the values of β_{lm} and δ_{lm} is moved into two other functions, `calculate_beta` and `calculate_delta`.

When all those terms are calculated using the Hadamard test (on quantum hardware, if not simulated), the terms $\langle\psi|\psi\rangle$ and $\langle x|\,\hat{H}_L\,|x\rangle$ are calculated. Afterwards, the absolute of C_l, $|\langle x|\,\hat{H}_L\,|x\rangle / \langle\psi|\psi\rangle|$, is returned to the minimizer.

As the minimizer does not provide an adequate option, this subfunction would be the right place for a subroutine to implement the precision-based termination approach from the original paper (cf. section 2.5). A workaround would be that – as soon as $C_L < \gamma$ – the same cost value would be returned upon each iteration until the classical minimizer stops by itself.

4.4.3 Implementation: calculation of β_{lm}

`calculate_beta(alpha: List[float])` is used to estimate β_{lm} for all combinations of l and m. `beta: List[List[complex]]`, as defined above, is **returned**.

The code initializes the nested list `beta`, creates a `Gate` object for $V(\textbf{alpha})$ by calling `generate_ansatz(alpha)` and converting the resulting `QuantumCircuit`. Then all β_{lm} are calculated.

The function `hadamard_test` is used to create all the neccessary Hadamard Tests. Its inputs are $V(\textbf{alpha}), A_l, A_m^\dagger$, generated by nested for-loop, as instances of the `Gate` or `Operator` classes. It then returns a `QuantumCircuit` implementing the Hadamard Test that is measured with the help of the `_calculate_expectationValue_HadamardTest` function. The parameter `im` is used to signal that the circuit for the imaginary part of β_{lm} is requested. cf. section 4.4.5 on the next page.

[11] This solves that `scipys` minimizer dismisses the imaginary part of a complex number but is not completely true to the definitions in [1].

An important aspect is that β_{lm} is only calculated if c_l and c_m^* are not 0. Even though the parameter might not be 0 itself, it (and δ_{lm}, that includes β_{lm}) is never needed on its own but used only in sums over these coefficients. Thus the outcome is not altered but the perfomance increased (often significantly).

The complete code for `calculate_beta(alpha)` can be found in figure 9 on the following page.

```python
def calculate_local_cost_function(alpha: List[float]) -> Union[complex, float]:
    beta = calculate_beta(alpha)
    delta = calculate_delta(alpha, beta)
    xHx = 0 # variable for ⟨x| Ĥ_L |x⟩
    psipsi = 0 # variable for ⟨ψ|ψ⟩

    for l, coeff_l in enumerate(params.coefficients):
        for m, coeff_m_conj in enumerate(params.coefficients_conjugate):
            xHx += coeff_l * coeff_m_conj * delta[l][m]
            psipsi += coeff_l * coeff_m_conj * beta[l][m]

    return abs(xHx / psipsi) # cost = |⟨x| Ĥ_L |x⟩/⟨ψ|ψ⟩|
```

Figure 8: The source code for `calculate_local_cost_function`.

4.4.4 Implementation of the Ansatz V(**alpha**)

`generate_ansatz` **returns** a `QuantumCircuit` object representing the Ansatz V(**alpha**). The required **input** is `alpha: List[float]`, the list storing all rotation coefficients r_i. As this function is a straightforward implementation of the Ansatz as presented in figures 1 and 2 on page 4 the source code is included in the git repository only.

In it the input `List[float]` is reformated to a `List[List[float]]` (cf. subsection 4.3), structured according to the sublayers. This step is included because the minimizer reduces all nested lists to a single-level list that is hard to comprehend for humans.

This function makes heavy usage of `Python`'s list slicing syntax to keep the internal logic readable. The syntax `list[start:stop:stepwidth]` effectively generates a new (sub-)list from `list`. In it all elements from the element with index `start` (inclusive) to the element with the index `stop` (exclusive) are stored. If `stepwidth` $\neq 1$, every 2nd, 3rd, ... element is ommited.

4.4.5 Implementation of the Hadamard Test

The function `hadamard_test` is used to generate and **return** a `QuantumCircuit` for the Hadamard Test. Most of the **input** parameters are `Gates` or `Operators`. It adds an ancilliary qubit to the circuit.
If used as function input, the respective `Gates` and `Operators` are applied to the circuit in the following order (more precisely: applied to a quantum register and, if neccessary, controlled by an ancilla):
`ansatz` → `first` → `first_uncontrolled` → Z-Gate on qubit `j` → `second_uncontrolled` → `second`.

Separate to these gates two Hadamard gates, a classical measurement and – if applicable – the adjungated phase shift gate are applied to the ancilla according to figures 3 and 4. This implementation allows to generate circuits for β, γ and δ. The full code can be found in figure 12 on page 17.
The **parameters** available on function call are:

- `im`: If `im` is not `None`, the adjungated phase shift gate is applied to the circuit. Thus, the circuit can be used to evaluate the imaginary part of ξ.
- `ansatz: Union[Gate, Operator]`
  It is an implementation of V(**alpha**), generated by (e.g.) executing `generate_ansatz` and executing the `.to_gate()` subroutine of the resulting `QuantumCircuit`.
- `first, second: Union[Gate, Operator]`
  Objects that are used to implement $A_l, A_m^\dagger$. Controlled versions of these are created with the `.control()` subroutine
- `first_uncontrolled, second_uncontrolled: Union[Gate, Operator]`
  These will be applied after `first` and before `second`. They are used to apply $U^\dagger, U$ to the circuit for the Hadamard test.
  They will not be controlled. Confer figure 4.
- `j` selects the qubit upon which to apply a Z-Gate. This is relevant for $\delta_{lm}^{(j)} - \beta_{lm}$, cf. figure 4.

To append the above to the quantum circuit a subfunction `append_ifExists` is defined. It checks for the existence and type of the optional parameters and creates deep copies (to avoid copy by reference errors). `Operators` are converted into `instructions` as those support, like `Gates`, the `control()` and the `append()` subroutines. If applicable, a version of the gate with one control qubit is created. Each `Gate` or `instruction` is then `appended` to the quantum circuit in the order described above.

```python
def calculate_beta(alpha: List[float]) -> List[List[complex]]:
    # preparation of the list for the results
    beta = [[complex(0, 0) for _ in params.coefficients] for _
            in params.coefficients_conjugate]
    # generate the Ansatz outside the for-loops for better perfomance
    V = generate_ansatz(alpha).to_gate()

    # A_l, (l, c_l)
    for gate_l, (l, coeff_l) in zip(params.decomposition_asGate,
                                    enumerate(params.coefficients)):
        if coeff_l == 0:
            continue # increase perfomance by ommiting unused β
        # A_m^†, (m, c_m^*) -- for quantum hardware use params.decomposition_adjoint
        for gate_m_adj, (m, coeff_m) in zip(params.decomposition_adjoint_asOperator,
                                            enumerate(params.coefficients_conjugate)):
            if coeff_m == 0:
                continue # increase perfomance by ommiting unused β
            # circuit for Re(⟨0| V^†(alpha) A_m^† A_l V(alpha) |0⟩)
            circ_had_re = hadamard_test(ansatz=V, first=gate_l,
                                        second=gate_m_adj)
            # circuit for Im(⟨0| V^†(alpha) A_m^† A_l V(alpha) |0⟩)
            circ_had_im = hadamard_test(ansatz=V, first=gate_l,
                                        second=gate_m_adj, im=1)

            # calculate Re and Im of β_lm / simulate circuits
            expV_had_re = _calculate_expectationValue_HadamardTest(circ_had_re)
            expV_had_im = _calculate_expectationValue_HadamardTest(circ_had_im)
            # set together β_lm from its real and imaginary part
            expV_had = complex(expV_had_re, expV_had_im)
            beta[l][m] = expV_had
    return beta
```

Figure 9: The source code for `calculate_beta`.

4.4.6 Implementation: calculation of δ_{lm}

The calculation of δ_{lm} is strictly similar to the calculation of β_{lm}. Only the important differences are characterized in this section. As a reminder:

$$\delta_{lm} = \beta_{lm} + \frac{1}{n_q} \sum_{j=0}^{n_q-1} \langle 0| V^\dagger A_m^\dagger U (Z_j \otimes \mathbb{I}_{\bar{j}}) U^\dagger A_l V |0\rangle$$

The function `calculate_delta` has another **input** parameter, besides `alpha: List[float]`, namely: `beta: List[List[complex]]` as it is generated by `calculate_beta`. It **returns** a similar list `delta: List[List[complex]]`, containing all relevant δ_{lm}. Furthermore, in the preparatory part of the function U and $U^\dagger$ are created with the adequate helper function.

The only other relevant difference between `calculate_delta` and `calculate_beta` is the sum over j. This is implemented with a third, nested `for` loop as the innermost loop.

4.5 Neccessary Post Correction

When one executes the code as defined above severe problems occur. The resulting state vector will appear as if it had been *mirrored*, compared to the state vector one would expect for $|b\rangle$. This behaviour is shown in table 10 for an example with four qubits. Furthermore, multiple sign errors occur.

Basis	$\|b\rangle$	**Simulation**
$\|0000\rangle$	a	0
$\|0001\rangle$	b	0
$\|0010\rangle$	0	0
$\|0011\rangle$	0	0
$\|0100\rangle$	0	0
$\|0101\rangle$	0	0
$\|0110\rangle$	0	d
$\|0111\rangle$	0	c
$\|1000\rangle$	c	0
$\|1001\rangle$	d	0
$\|1010\rangle$	0	0
$\|1011\rangle$	0	0
$\|1100\rangle$	0	0
$\|1101\rangle$	0	0
$\|1110\rangle$	0	b
$\|1111\rangle$	0	a

Figure 10: State vector coefficients.

Though it is unclear why this error arises, it is not that hard to fix it. It becomes clear that this inversion of the qubits can be fixed by applying the X-Gate to every single qubit. Studying different examples also reveals that applying the Z-Gate to each qubit can fix the sign errors. Thus, one has to implement a short function for post correction that has to be applied after the minimization. More precisely: that has to be applied after them $AV(\mathbf{alpha})\,|0\rangle$ is applied.

```
def postCorrection(qc: QuantumCircuit):
    for i in range(params.n_qubits):
        qc.x(i)
        qc.z(i)
    return qc
```

Figure 11: The source code for the post corrections.

This post correction fixes all errors except for a global sign error (that does not occur each time). This is because the **absolute of** the cost function is minimized instead of the non-absolute version. Effectively, this global sign error corresponds to a global phase shift and can be treated as such.

4.6 Executing a simulation

In the git repository a working example is included in `execute.py`. It imports the object `params` from and runs a short simulation based on the parameters and decomposition set in `GlobalParameters.py` and the definition of $|x_0\rangle$ from `_U_primitive`.

The quality of the result depends on both the number of iterations of the minimizer as well as on the simulation shots – both can be set in `GlobalParameters.py`, when initializing `params`. For example: with about 150 iterations, 2 layers and $2 \cdot 10^3$ shots the cost function can often be minimized to a value of approximately 0.01 or lower – but only for rather simple examples like $A = \mathbb{I}$ or $A = Z(2)$. Simulating slightly harder problems soon requires signifcantly more ressources and a higher accuracy.

Enhancing the accuracy requires increasing both iterations and shots. If the number of shots is too low, the simulated noise prevents a gain in accuracy. If the number of iterations is too low, the minimizer cannot find the ideal solution. Furthermore, more layers can help to improve the simulations perfomance, especially for complexer systems.

Increasing these parameters automatically increases calculation time. This increase is linear at worst. Increasing the number of qubits, however, translates to a faster than linear growth in both computation time and memory requirements. Thus, simulating systems with more than four to six qubits quickly becomes unattractive, especially due to memory constraints.

[12] This can be shown easily, either by calculation or simulation.

4.7 Modifications required for real quantum hardware

The above code is well-suited for simulations as it uses optimized libraries and supports a broad range of problems/Operators as input. However, there are some problems and aspects that require optimization before the code should be executed on real quantum hardware. They resolve around the following points:

1. The `Operator` class. Implementing its objects in a real quantum circuit would require (too) many quantum gates. However, in the context of simulations it is chosen due to the gained flexibility:
   a) Any object of the class (i.e. element of the decomposition) can be adjointed via the call of a single subroutine, no matter how complicated the operator might be.
   b) It enables one to implement quantum operators, that are a sum of different gates **with varying coefficients** (i.e. it supports coefficients in the deocmposition, that are neither 0 or 1).
2. The usage of the `.control()` subroutine for `Operators` and `Gates` can result in non-standard gates, which might be hard to implement.

The degree of difficulty for solving these problems depends on the specific aspect and the general physical problem that is considered. For some classes of problems many problems vanish quickly:

One such example are decompositions containing only single qubit **hermitian** gates and their products.

As a consequence of the hermitian property – $A_l^\dagger = A_l$ – the need for explicit adjoints and as such the need for the list `decomposition_adjoint_asOperator` vanishes. If only single qubit gates are implemented, the list can be replaced with the list `decomposition_asGate`. If products of such gates are involved, a list with the inverse order of multiplication can be used instead. In this example problem 1.a vanishes as adjungations are trivial. If only standard single qubit gates are involved, problem 2 does not exist either because for these gates controlled versions exists. This holds true for products as well, as the product of (e.g.) control(X), control(Y) and control(Z) is the same as control(XYZ).[12]

However, even this simple problem causes conflicts with 1.b. This is due to the implementation of U. Furthermore, in the end there is still A itself, which – if it is only given as a decomposition with coefficients – needs to be applied to evaluate the results.

Thus, in the end, adding a decomposition of A with multiple coefficients is the strongest blocker for applying a general version of this code to real quantum hardware. For the time being no efficient solution to implement this in Qiskit was found. However, for simple examples (arbitrarily complicated $|x_0\rangle$ but an A acting on a single qubit only) only few changes are necessary to execute the code on quantum hardware. This is discussed in section 5.2.

```python
def hadamard_test(
    *, ansatz: Union[Gate, Operator] = None, j: int = None, im=None,
    first: Union[Gate, Operator] = None, first_uncontrolled: Union[Gate, Operator] = None,
    second_uncontrolled: Union[Gate, Operator] = None, second: Union[Gate, Operator] = None
     ):
    qr_had = QuantumRegister(params.n_qubits)
    circ_had = QuantumCircuit(qr_had)
    ancilla = QuantumRegister(1, name="ancilla")
    cr_had = ClassicalRegister(1)

    circ_had.add_register(ancilla) # The ancilla to measure
    circ_had.add_register(cr_had) # Store the measurement in here

    qubits_designation_control = [i for i in range(params.n_qubits)]
    qubits_designation_control.insert(0, params.n_qubits) # [n, 0, 1, 2, ...]

    def append_ifExists(obj: Union[Gate, Operator], control=None):
        if isinstance(obj, (Gate, Operator)):
            _obj = obj.copy() # deep copy
            if isinstance(_obj, Operator):
                _obj = _obj.to_instruction() # instructions support .control() and .append()
            if control is True:
                _obj = _obj.control(1)
                circ_had.append(_obj, qubits_designation_control)
            else:
                circ_had.append(_obj, qr_had)

    # act on the ancilla
    circ_had.h(ancilla)

    # if Im(<>) shall be calculated
    if im is not None:
        circ_had.sdg(ancilla) # adjungated phase shift gate on the ancilla

    append_ifExists(ansatz) # Ansatz on qr_had

    append_ifExists(first, 1) # controlled A_l

    append_ifExists(first_uncontrolled) # U^†, if applicable

    if j is not None:
        circ_had.cz(params.n_qubits, qr_had[j]) #Z_j, if applicable

    append_ifExists(second_uncontrolled) # U, if applicable

    append_ifExists(second, 1) # controlled A_m^†

    # last operation on the ancilla & measurement
    circ_had.h(ancilla)
    circ_had.measure(ancilla, cr_had)

    return circ_had
```

Figure 12: The source code for the `hadamard_test`.

5 Discussion of the results

In this chapter the results of the algorithm will be discussed. In the first section the simulations of the algorithm and the general knowledge gained will be reviewed. In the second section the execution of the VQLS-algorithm on a quantum computer provided by **IBM Quantum**[13] will be presented. These experiments were conducted with the 5-qubit quantum computer `ibmq_manila`, equipped with an IBM Quantum `Falcon r5.11L` processor [7].

5.1 Simulations

Simulating the VQLS-algorithm proved its functionality: by measuring the (simulated) circuits, calculating the cost function and minimizing it satisfactory results could be found. But the simulations also showed some complications and room for improvement. For simple problems the algorithm can produce very adequate results with comparatively few iterations ($\mathcal{O}(100)$), a low circuit depth (`n_layers` $= 2$) and rather few shots (`n_shots` $= 10^3$). An example for such a problem would be $A = \mathbb{I}$ or $A = Z(i)$ (i arbitrary). With the above parameters a cost function value of $\mathcal{O}(0.01)$ could be achieved often and the resulting state vector $A\,|x(\mathbf{alpha})\rangle$ would be very close to the one for $|b\rangle$[14].

However, slightly increasing the problems complexity, e.g. by using a Y- or X-gate in the decomposition of A, drastically increases the requirements. This problem appears to be caused by the switching of qubits, respectively their values. For such decompositions all parameters mentioned above need to be increased. At least 200 to 400 iterations of the minimizer, a circuit depth of 4 or 5 and a higher simulation precision (i.e. $\approx 10^4$ shots) are necessary. But even this might not be enough, because the minimization has to reach a lower magnitude of the cost functions value too as the results for $C_L \approx 0.01$ are not in line with the expectations. And these stronger parameters are already required for a decomposition involving a single coefficient $\neq 0$ – even then, in many cases, without success.

This lack of success is primarily due to the classical minimizer that is often unable to lower the cost function significantly below 0.01. Therefore, it is clear that the classical minimizer (i.e. `scipy`) is one of the primary objectives when planning optimizations (with the other being the usage of the `Operator` class). Furthermore, it becomes clear that simulating decompositions containing multiple gates is not expedient for the time being, because the results with the current COBYLA minimizer[15] would not be satisfactory.

Considering all of the above, simulation times that are often in the order of hours to days[16] and problems resolving around the implementation of non-trivial decompositions,[17] it becomes clear that simulations of the algorithm soon reach their borders. They proof the general functionality and potential of the VQLS-algorithm, deliver decent results for some problems and can be used as an implementation reference. However, they cannot provide a detailed analysis or decent results for elaborate problems – at least not with the minimizers tested.

5.2 Enabling Quantum Hardware

An approach to solve some of these problems, especially simulation time, is the execution on current-day quantum hardware. Therefore the algorithm was executed with IBM Quantums freely available `manila` system. Due to the problems discussed in section 4.7 this required some restrictions to the program:

- The dispense of the `Operator` class requires that decompositions contains only gates acting on a single qubit (only one coefficient can be $\neq 0$ and that one has to be $1 \to A = \mathbb{I}$ or $A = Z(i)$ or $A = Z(i)Y(i)$ for example).
- U and $U^\dagger$ have to be prepared manually for almost every problem.

The respective code can be found on Github[18] as well (in the folder IBMQ). It is required to insert the personal API-Key (cf. [7]) to access the IBM Quantum system when initializing `params`. The other modifications are based on the documentations in [7] and [6].

5.3 Results on Quantum Hardware

At this time, on the `manila` quantum computer the algorithm can not completely fulfil the expectations. Almost all Hadamard Tests are executed and measured in under ten seconds – this can be faster than the simulations of more-intricate problems but is slower than many of the other simulations. The real setback, however, is that as IBMs quantum computers are publicly available the execution of each quantum circuit is only possible after waiting in a queue for several minutes per circuit. Therefore, the execution of the algorithm on quantum hardware takes (with the rather limited, publicly available number of qubits) significantly longer than the simulation of a respective circuit. Furthermore, only a certain number of jobs can be executed in sequence unitl an error is raised and some time has to

[13] For a disclaimer cf. to chapter 6 on page 20.
[14] Sometimes with a global phase shift (global sign error), cf. subsection 4.5.
[15] That was already the best one of the ones tested.
[16] On powerful modern-day laptops for systems with 4-8 qubits and non-intricate decompositions.
[17] As noted in section 4 it is required that $A = \sum c_l A_l$ is unitary. For mixed decompositions implementing this with the `Operator` class can proof to be hard.
[18] `https://github.com/muehlhausen/vqls-bachelor-thesis`

exceed until new jobs are allowed. Therefore, it is necessary to manually restart the program multiple times (potentially loosing some progress due to resetting the minimizers memory each time – even though the last **alpha** can be reused).

Nonetheless, this shows that the VQLS-algorithm can be executed on current-day quantum hardware that is available to everyone. And one has to consider that these results were found with the freely-provided and therefore rather-limited quantum computers. Considering that quantum computers scale significantly better than classical computers simulating them when adding qubits, this only helps to support the potential of the algorithm – if it should be executed on a quantum computer that is exclusively available to it and provides more qubits. And such quantum devices already exist (e.g. up to 65 qubits at IBM Quantum) or are in development and expected for the near future [7, 9]. Therefore, the results required for the use in a productive enviroment are not available yet, but they might be sooner than later.

6 Conclusion

To conclude this thesis it can be stated that the VQLS-algorithm has both potential and functionality. It and other VHQCAs like it might not yet be at a point where they are superior to classical algorithms, but, as has been shown in simulations as well as on real quantum computers, it works in principle. The implementation presented here can successfully solve some classes of problems and there should be not that many steps necessary to enable solving more problems:

- An alternative to the `Operator` class, that requires less gates to implement non-intricate decompositions, has to be found or developed.
- An optimized minimizer is required to solve harder problems, especially those that include switching the values of qubits.
- Modules to automatically generate U for a given $|b\rangle$ and to decompose A (cf. ref. [1]) need to be added.

After these updates an implementation of the VQLS should be able to efficently solve elements of the Quantum Linear Systems Problems. Especially, as soon as quantum processors, that support more qubits, become generally available. On these systems and for adequatly large problems the algorithm might even be able to get close to its estimated up-to exponential performance benefit in comparison to classical systems [1]. Therefore, it can be deemed plausible that within the next years or decade ([1, 9]) quantum hardware could be integrated into productive, hybrid algorithms and be used to solve problems significantly faster than a pure classical algorithm could.

Helpful ressources

In this short section some further helpful ressources shall be mentioned. Unless explicitly quoted or otherwise referenced in the text those were not used for the final thesis as the primary source [1] provided all the neccessary information. But they provided a useful context to initially understand the topic and comprehend the interaction or functionality of various aspects.

The Qiskit tutorial on the VQLS [5] was helpful, though soon a different approach and feature set for the programing in the context of this thesis was chosen. Similar applies to a tutorial on `pennylane.ai` [10]. Furthermore, various articles in the Qiskit textbook [6] provided useful context.

A paper on a variational eigenvalue solver by Peruzzo et al. [4] and two articles on `medium.com` [11] and `towardsdatascience.com` [12] provided context to variational approaches on quantum devices and proved valuable for the basic understanding of the topic.

The foundations of quantum computing have been obtained during a lecture hold by Professor Dr. C. Urbach, the supervisor of this thesis, in the winter semester 2020/21 at the University of Bonn.

This thesis was set with LaTeX and Ti*k*Z. The `Quantikz` package [13] was used to typeset figures 1, 2 and 5.

I acknowledge the use of IBM Quantum services for this work. The views expressed are those of the author, and do not reflect the official policy or position of IBM or the IBM Quantum team.[19]

References

[1] Bravo-Prieto, C. *et al.* Variational quantum linear solver (2020). `1909.05820v2`.

[2] Harrow, A. W., Hassidim, A. & Lloyd, S. Quantum algorithm for linear systems of equations. *Phys. Rev. Lett.* **103**, 150502 (2009). URL `https://link.aps.org/doi/10.1103/PhysRevLett.103.150502`.

[3] Higham, P. N. J. What is a sparse matrix? `https://nhigham.com/2020/09/08/what-is-a-sparse-matrix/` (2020).

[4] Peruzzo, A. *et al.* A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* **5**, 4213 (2014). URL `https://doi.org/10.1038/ncomms5213`.

[5] Ceroni, J. Variational quantum linear solver. `https://qiskit.org/textbook/ch-paper-implementations/vqls.html`.

[6] Abraham, H. *et al.* Qiskit: An open-source framework for quantum computing (2019).

[7] IBM quantum. `https://quantum-computing.ibm.com/` (2021).

[8] The SciPy community. Scipy API. `https://docs.scipy.org/doc/scipy/reference/` (2021).

[9] IBM. `https://www.ibm.com/quantum-computing/` (2021).

[10] Mari, A. Variational quantum linear solver. `https://pennylane.ai/qml/demos/tutorial_vqls.html` (2021).

[11] Weaver, J. The variational quantum eigensolver. `https://medium.com/qiskit/the-variational-quantum-eigensolver-43f7718c2747` (2019).

[12] Zickert, F. The variational quantum eigensolver — explained. `https://towardsdatascience.com/the-variational-quantum-eigensolver-explained-adcbc9659c3a`.

[13] Kay, A. Tutorial on the quantikz package (2020). `1809.03842v5`.

[19] Cf. `https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/cite` .

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate kenntlich gemacht habe.

Alexander Cornelius Mühlhausen, Bonn, den 3. August 2021