DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Classical Reinforcement Learning using Quantum Algorithms

Ayse Kotil

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Classical Reinforcement Learning using Quantum Algorithms

Klassisches Reinforcement Learning mittels Quantenalgorithmen

Author:	Ayse Kotil
Supervisor:	Prof. Dr. Christian B. Mendl
Advisor:	Prof. Dr. Christian B. Mendl
Submission Date:	15.01.2021

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.01.2021 Ayse Kotil

Acknowledgments

I would like to thank my advisor, Prof. Dr. Christian B. Mendl, for always kindly taking the time for answering my questions, guiding me through theoretical and practical challenges, and for inspiring me to pursue the field of quantum computing.

Abstract

This study investigates how the Harrow-Hassidim-Lloyd (HHL) Algorithm, the quantum algorithm for solving linear systems of equations (LSE), performs within the Policy Iteration of model-based Reinforcement Learning (RL). The HHL Algorithm offers an exponential speed-up over the best known classical algorithm for solving LSE. We simulate the algorithm numerically with Python and conduct an error analysis for an example RL application, by examining the Policy Iteration outputs produced by the HHL Algorithm. We address the prerequisites of the HHL Algorithm and how they affect our RL problem, and finally present problem mappings for our example application.

Contents

1 Introduction

1.1 Insights into Quantum Mechanics and Quantum Computing

The concept of quantum computing is based on quantum mechanics and possesses features that differ from classical computing. The most distinguishing features of quantum mechanics include superpositions, entanglement and uncertainty.

Superposition stands for the ability of a quantum system to be in any of the infinitely many combinations of different states. Any linear combination of quantum states represents a valid quantum state as long as the probability space is preserved. In the classical world, no such 'mixed-states' exist.

The principle of *entanglement* characterizes quantum systems which cannot be described as a composite system of individual components; in other words, it is not possible to obtain complete information about one component without considering the whole system. The existence of such a system was first pointed out by Einstein, Podolsky, and Rosen [1]. They showed that a qubit belonging to an entangled pair of qubits (called a EPR-pair), when it interacts with the environment through a measurement, immediately leads to a collapse of the other qubit, even with a long distance between them. They did so without the underlying concept of entanglement. In fact, they questioned the integrity of quantum mechanics, as the behaviour of the qubit pair was not explainable, in Einstein's words, "a spooky action at distance" [2]. Later on, Bell [3] showed that a theoretical justification for the observations conducted by Einstein, Podolsky and Rosen cannot be made with *local hidden variables* to 'complete' quantum mechanics.

Last but not least, the *uncertainty* principle, introduced by Heisenberg [4], addresses the uncertainty regarding obtaining information about physical quantities of a particle. For instance, it is not possible to gain information both on a particle's momentum and position with 100% certainty. The more information obtained about the momentum of the particle, the less certain one's knowledge becomes about the position of the particle, and vice versa.

First ideas about a quantum computer emerged much later in the 1980s, by the works of Feynman [5], Deutsch and Jozsa [6], Bernstein and Vazirani [7], and many other scientists. With their discoveries, and as quantum information theory started to revolve around the features of quantum mechanics, the capabilities of a quantum computer attained a stronger interest.

Making concrete statements about the capabilities of a quantum computer is not always trivial. Quantum computing is suitable for specific computational tasks; a quantum computer cannot outperform classical computers in every computational problem. In fact, a problem needs to have a particular mathematical structure in order to be able to offer significant speed-ups in its quantum-equivalent over its classical version [8]. The work of Shalev *et al.* [9] extends the results shown by Aaronson and Ambainis [8] and further puts symmetry of functions in relation with large quantum speed-ups. There exist quantum algorithms which are proven to offer polynomial (Grover's search algorithm [10], search via Quantum Walk [11]) and exponential speed-ups (Shor's factorization algorithm [12]), and some of which are believed to offer a speed-up (Variational Quantum Eigensolver [13]), but are not mathematically proven to do so, called *heuristic* quantum algorithms.

Implementing quantum algorithms on a quantum computer is highly challenging due to the fragile nature of quantum states, which are prone to lose information due to outside inferences from the environment as the number of qubits grows. Quantum error correction [14] is the concept of encoding quantum systems in entangled states, so that the environment cannot damage the system. This requires a large number of qubits, which makes it challenging to construct an error-corrected quantum computer. Babbush *et al.* [15] argue that unless error rates in quantum computers improve with increasing number of qubits, the overhead induced by error correction will make the quantum advantage of algorithms with only a quadratical speed-up infeasible. This realization brings attention to quantum algorithms with an exponential speed-up over their classical equivalents, which is one of the key motivation points of our study.

1.2 Problem Statement

Machine learning (ML) is one of the research fields of great potential to change the way we operate in a broad scope of applications, and comes with a strong computational power demand. This raises the question whether quantum computing can be used for tackling challenging ML problems, and there are already significant advances in that matter. We approach the ML sub-field reinforcement learning (RL), in particular, the Policy Evaluation (PE) step of the Policy Iteration Algorithm which is used to find an optimal state-action mapping for a model-based RL. We take advantage of the special form of the PE, which is a linear system of equations (LSE) and replace it with the quantum solver for LSE, the Harrow-Hassidim-Lloyd (HHL) Quantum Algorithm [16]. We address the necessary problem mappings for the new hybrid algorithm and investigate how the quantum equivalent of solving LSE affects the RL problem.

1.3 Related Work

There are several studies which put RL in context with quantum mechanics and quantum computing. Dunjko *et al.* [17] suggest a quantum setting for RL with actions and percepts as orthogonal quantum states, where quantum registers are used to store the history of the environment. Dong *et al.* [18] show an approach for representing states and actions as quantum states in superpositions, and ties agent's decision making with measuring quantum states; as well as transition probabilities with amplitudes of these quantum states. A more physical construction was introduced by Lamata [19] with super-conducting circuits and by Chen *et al.* [20] with variational quantum circuits for deep reinforcement learning.

Compared to the existing studies, our study follows a different approach. Instead of looking at the learning environment of the agent in a quantum mechanical way, we aim to substitute parts of classical algorithms for solving RL problems with their 'computationally' equivalent quantum algorithms. Thus the nature of the problem remains to be classical, however, the computation of the problem solution is expected to be accelerated by quantum computing. In addition, addressing the challenges of such a substitution (which is quite problem-specific) is also a side-goal of our study.

1.4 Outline of the Study

In the light of our problem statement, we aim to give the reader sufficient theoretical background about the two key concepts of our study: Chapter 2 focuses on the mathematical definitions and representations of the main components of RL, including the central algorithms used for solving a model-based RL. In Chapter 3, we show how quantum information and transformations are represented by qubits in a mathematical sense. Chapter 3 also examines relevant quantum algorithms which form the basis for the HHL Algorithm described in Chapter 4. Finally, we present a way of simulating the HHL Algorithm with Python, which will be used to solve an example RL application, in Chapter 5.

2 Overview of Classical Reinforcement Learning

2.1 Introduction

Reinforcement learning (RL), in contrast to other machine learning procedures, requires a tight interaction between an environment and an agent. Typical examples include self-driving cars, worker robots and industrial controllers. It is both a research subject and an experimental field since *dynamic programming*[1] became an important part of self-learning processes. In this section, we are going to introduce the basic components of RL including the algorithms which will be later relevant for our quantum computing application.

In the following, we focus on the discrete, *model-based* RL, where the stochasticity of the environment takes advantage of the time-invariant nature of homogeneous Markov Decision Processes (MDP).

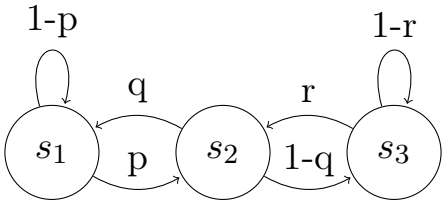


Figure 2.1: A simple example of a Markov Chain with three states

A discrete stochastic process $(X_n)_{n\in N_0}$ is called a *Markov process* with random variables $\{X_i, i \in \mathbb{N}_0\}$ taking values from a countable state space $\mathcal{S}$, if the following property holds $\forall i_0, \dots i_{k+1} \in \mathcal{S}$ [23, p. 7]:

$$\mathbb{P}(X_{k+1} = j | X_0 = i_0, X_1 = i_1, \dots, X_k = i_k) = \mathbb{P}(X_{k+1} = j | X_k = i_k) \tag{2.1}$$

The probability of transiting to the state X_{k+1}, given the history of states $X_0, \dots, X_k$, is equal to the probability of transiting to it when the transition only depends on the current state. In other words, a stochastic process is Markovian if it has a memory

[1] Introduced by Richard Bellman in 1957 [21], dynamic programming divides an optimization problem into sub-problems to bring them together for a solution [22, p. 5].

length of 1; the future of the process just depends on the present, not the past. The process is in addition *homogeneous*, if the transition probabilities are time-invariant, i.e. $\mathbb{P}(X_{k+1} = j | X_k = i) = p \, \forall k$.

2.2 Mathematical Model Definition

RL is a machine learning problem defining a mapping of situations to actions in order to maximize a numerical reward signal [24, p. 1]. The reward is acknowledged by an *agent*. Like in other machine learning applications, the learning process is realized implicitly by the agent. The *environment*, in which the agent operates, is defined as an MDP consisting of the following elements [25, pp. 1–2]:

- $\mathcal{S}$: State space, i.e. a set of states which the environment can evolve to
- $\mathcal{A}_s$: Action space, i.e. a set of actions defined for a given state $s \in \mathcal{S}$
- $\mathcal{P}_a \in [0,1]^{\mathcal{S} \times \mathcal{S}}$: Transition probability matrix. A matrix entry p_{ij} corresponds to $\mathbb{P}(s_j | s_i, a)$, i.e. the probability of transiting to state $s_j \in \mathcal{S}$ given a state $s_i \in \mathcal{S}$ and an action a
- $\mathcal{R} : \mathcal{S} \to \mathbb{R}$: Reward function mapping from a state to its corresponding real value
- $\pi : \mathcal{S} \to \mathcal{A}$: Policy function from the state space to the action space, i.e agent's decision mapping

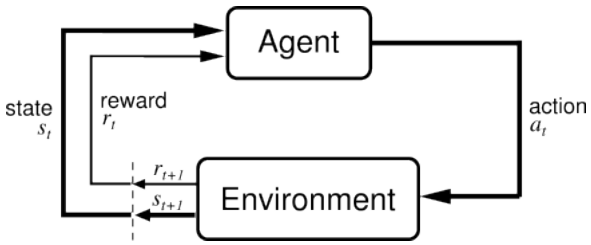


Figure 2.2: The agent-environment interaction in reinforcement learning [26].

The agent chooses in every state an available action by following a certain policy. It then receives a real valued reward which can be positive or negative. A negative reward is often associated with a 'punishment'; as the goal of the agent is to maximize the (expected) total reward sum. By receiving a negative feedback from the environment, the agent is motivated to choose the 'correct' actions in next iterations.

2.3 Markov Decision Process Components

2.3.1 Utility Function

In order to model the RL environment, a proper definition of the goal of the agent is needed. The agent is encouraged to consider the long-term consequences of its actions, since the complexity of the system can vary with each use case. In most cases, the agent may have to make decisions which do not lead to an immediate positive reward but result in a positive feedback in the long-run. This decision-making process relies on the *utility function* defined as follows [27]:

$$U(\boldsymbol{s}) = \sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{R}(s_t) \tag{2.2}$$

with $\boldsymbol{s} = \{s_1,\ s_2,\ \dots\},\ s_i \in \mathcal{S}\ \forall i$ and $0 < \gamma \leq 1$ being the *discount factor*. The infinite set $\boldsymbol{s}$ represents the (infinite) sequence of states the environment evolves to at discrete time steps t_i during the exploration phase. The received utility is the (infinite) sum of reward function values assigned to the states, which can be considered a measure of success for the agent in the learning process. The goal is to maximize the overall received rewards.

The discount factor controls the weight of the future rewards, i.e. the feedback received in later time instances. For values < 1, the future rewards become less relevant than those in the early stages of the exploration. The agent can be motivated to find the optimal solution as fast as possible by reducing the weight on future rewards. The trade-off is that the smaller γ is, the shorter 'the horizon' becomes; meaning that the agent gets closer to disregarding its future actions. This may be inconvenient in cases where future decisions are of great importance. On the other hand, γ being closer to 1 results in longer learning processes which will be discussed in 2.4.1. Therefore one should decide on a reasonable γ in accordance with the corresponding use-case.

From a mathematical point of view, the usage of discount factor leads to a convergence of the infinite sum introduced in (2.2) when $\gamma < 1$ [28, p. 650]:

Proof. Let the reward function $\mathcal{R}(s)$ be bounded from above, i.e. $\mathcal{R}(s) \leq R_{max}\ \forall s$. Then it holds:

$$\sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{R}(s_t) \leq \sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{R}_{max} = \mathcal{R}_{max} \frac{1}{1-\gamma} \quad \square \tag{2.3}$$

The agent starts exploring the environment with an initial policy, i.e. concrete actions are assigned to every state of the environment. For the agent to be able to make an assessment of the decisions made, a reference measurement is needed, such that the agent can compare its own decision process with an optimal value to eventually reach it. Every

state is evaluated by the slightly altered utility function by means of assigning a single state $s \in \mathcal{S}$ a measurable *value*:

$$V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t) \middle| \pi^*\right] \tag{2.4}$$

where $\pi^* = \arg\max_{\pi} U^{\pi}(\boldsymbol{s})$ is the optimal policy which maximizes the utility function starting from the state s, i.e. $\boldsymbol{s} = \{s, s_1, s_2, \dots\}$.

2.3.2 Bellman Equation

The value of a state shown in (2.4) depends directly on the next states of the environment, since the received rewards in the subsequent time instances are attached to next states. The following equation makes use of a recursive structure, using the fact that the agent's expected sum of rewards under an optimal policy is equal to the sum of the current received reward and the expected sum of next states rewards (Bellman Equation) [28, p. 652]:

$$V(s) = R(s) + \gamma \max_{a \in A} \sum_{s'} \mathbb{P}(s'|a) V(s') \tag{2.5}$$

The Bellman Equation attempts to answer the following question: *What is the value of the state s, given all the other values of next-states which the agent would visit, by maximizing the sum of next-state values, i.e. by always choosing the action that would lead to the most rewarding (remaining) state set?*. The Bellman Equations describe a system of non-linear equations, since the max operator is non-linear. Finding a solution to these equations is therefore non-trivial. In the following subsection, we examine two different approaches for solving the Bellman Equation.

2.4 Markov Decision Process Algorithms

Having defined the tools to work with, we proceed to the RL algorithms used in optimizing the agent's behaviour. They are both iterative approaches and have different strengths and weaknesses which will be discussed in each case.

2.4.1 Value Iteration Algorithm

The first approach, called Value Iteration Algorithm (VI), requires randomly assigned values for each state. That way the environment is initialized with non-optimal state

values, which are updated iteratively as follows:

$$V_{i+1}(s) = R(s) + \gamma \max_{a \in A} \sum_{s'} \mathbb{P}(s'|a) V_i(s') \tag{2.6}$$

At every iteration instance i, every state value is updated such that the overall utility increases each time. The algorithm terminates if $\boldsymbol{V}_{i+1}(\boldsymbol{s}) - \boldsymbol{V}_i(\boldsymbol{s}) < \epsilon(1-\gamma)/\gamma$ with $\boldsymbol{s}$ being the state vector and $\boldsymbol{V}$ the corresponding state value vector [28, p. 653]. ϵ is the desired error constant.

Convergence. VI converges when $\gamma < 1$ as shown in the following [28, pp. 654–655]:

Proof. The convergence of VI relies on the fact that updating every state in each iteration is a *contraction*, i.e. the Bellmann update mapping [29], denoted $\mathcal{B} : \mathcal{V} \to \mathbb{R}$, satisfies the following property:

$$||\mathcal{B}(\boldsymbol{V}_i) - \mathcal{B}(\boldsymbol{V}'_i)|| \leq \gamma ||\boldsymbol{V}_i - \boldsymbol{V}'_i|| \tag{2.7}$$

with $||\boldsymbol{V}_i|| = \max_s V(s)$. The Bellman function thus takes the (non-optimal) value of a state s and returns a new value for s. Thus the difference between two possible values of the state s after applying the Bellman operator is always less than or equal to the difference of the values themselves, without applying the Bellman operator. Let $\boldsymbol{V}^*$ be the optimal state value vector. One can replace $\boldsymbol{V}'_i$ in (2.7) to obtain:

$$||\mathcal{B}(\boldsymbol{V}_i) - \boldsymbol{V}^*|| \leq \gamma ||\boldsymbol{V}_i - \boldsymbol{V}^*|| \tag{2.8}$$

(2.8) shows that applying the Bellman operator reduces the difference between a value vector at an arbitrary iteration point and the actual value vector by γ. In fact, the convergence is exponential, since

$$\begin{aligned} ||\mathcal{B}(\boldsymbol{V}_i) - \boldsymbol{V}^*|| \leq \gamma ||\boldsymbol{V}_i - \boldsymbol{V}^*|| &= \gamma ||\mathcal{B}(\boldsymbol{V}_{i-1}) - \boldsymbol{V}^*|| \\ &\leq \gamma^2 ||\boldsymbol{V}_{i-1} - \boldsymbol{V}^*|| \leq \cdots \leq \gamma^i ||\boldsymbol{V}_0 - \boldsymbol{V}^*|| \quad \square \end{aligned} \tag{2.9}$$

The minimum number of required iterations N for a desired error rate ϵ can also be derived with the help of (2.3):

$$\begin{aligned} &||\boldsymbol{V}_0 - \boldsymbol{V}|| \leq \mathcal{R}_{max} \frac{2}{1-\gamma} \Rightarrow ||\boldsymbol{V}_N - \boldsymbol{V}|| \leq \mathcal{R}_{max} \frac{2\gamma^N}{1-\gamma} \overset{!}{\leq} \epsilon \\ &\Rightarrow N = \left\lceil \log \frac{\epsilon(1-\gamma)}{2\mathcal{R}_{max}} \frac{1}{\log \gamma} \right\rceil \end{aligned} \tag{2.10}$$

The lower limit for N shows that as γ gets closer to 1, the number of iterations needed for a maximum error rate ϵ grows exponentially, resulting in a longer learning process as mentioned in 2.3.1.

The agent can determine its optimal policy according to the state values resulting from the converged Value Iteration Algorithm:

$$\pi(s) = \arg\max_{a \in \mathcal{A}} \sum_{s'} \mathbb{P}(s'|s,a)V(s) \tag{2.11}$$

Thus, for a state s, the agent chooses the action which maximizes the expected return it would get from a possible next state s'. The Value Iteration Algorithm can be summarized as follows:

Algorithm 1: Value Iteration

Initialization

$\boldsymbol{V}_0 \leftarrow 0$, i.e. initial states values are 0

$i \leftarrow 0$

while *True* **do**

  foreach $s \in \mathcal{S}$ **do**

    $V_{i+1}(s) = R(s) + \gamma \max_{a \in A} \sum_{s'} \mathbb{P}(s'|a)V_i(s')$

  if $||\boldsymbol{V}_{i+1} - \boldsymbol{V}_i|| < \epsilon(1-\gamma)/\gamma$ **then**

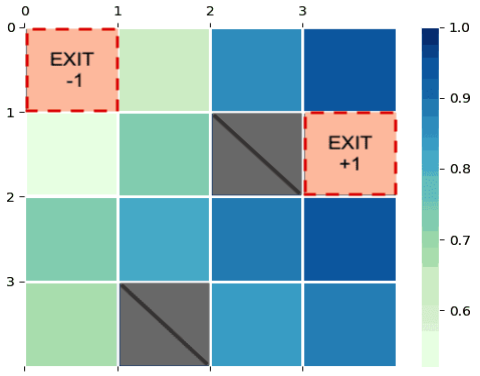
    return $\boldsymbol{V}_{i+1}$

  $i \leftarrow i + 1$

In the following, we will present a simple example of a grid environment (adapted from [28, p. 646]), where the agent is tasked to find the right exit with the shortest path possible.

Example. The example grid environment consists of a 4x4 matrix. The action set $\mathcal{A}$ consists of 4 actions; *up*, *down*, *left* and *right*. The state set $\mathcal{S}$ consists of the 16 fields and one additional 'game-over' state, which is an absorbent state, i.e. the agent stays in this state forever once the state is arrived. There are two possible exit fields which directly lead to the game-over state, one with a positive and one with a negative reward. The agent is expected to end its walk at one of these two exit fields, and optimally at the field with the positive reward. For every non-terminal state, the agent receives -0.05.

The only constraints the agent has are the blocked fields $(1,2)$ and $(3,1)$; the agent cannot move to these fields. When the agent chooses an action towards them, it stays in its current state.



(a) Utilities represented as a color gradient

EXIT -1	0,574	0,821	0,915
0,459	0,678		EXIT +1
0,670	0,767	0,845	0,919
0,608		0,789	0,843

(b) Utilities represented as numerical values

Figure 2.3: State values of the example grid world resulting from the converged value iteration

With $\mathcal{S} = \{0, \ldots, n^2 - 1\}$, $\mathcal{A} = \{-n, +n, -1, +1\}$, the transition probabilities within the environment are defined as follows:

$$\mathbb{P}(s_j|s_i, a) := \begin{cases} 0.8 & \text{if } s_i + a = s_j \\ 0.2 & \text{if } s_i + a \neq s_j \text{ and } s_i \text{ is a corner state} \\ 0.1 & \text{if } s_i + a \neq s_j \text{ and } s_i \text{ is an edge state} \\ 0.2/3 & \text{if } s_i + a \neq s_j \text{ and } s_i \text{ is a middle state} \\ 0 & \text{otherwise} \end{cases}$$

When the agent intends to move towards a direction, the probability that the intended action actually occurs is 0.8. Every other action that is not in compliance with the intended action is evenly distributed. The agent cannot move between non-adjacent states (non-neighbours), hence every other transition has 0 probability of taking place.

Figure 2.3 shows the resulting utility mapping from the converged value iteration algorithm with $\gamma = 1$. The color range indicates the different utility values, ranging from 0.5 to 1.0. One can observe that the states around the positive exit field have a high value, whereas the ones around the negative exit field have the lowest values amongst all states.

According to the policy extraction (2.11), the agent would follow the ascending value path/the darkening color direction. The agent's start state does not matter, since the agent's decision making is independent from any particular state; actions are chosen with respect to the next states only. The agent thus chooses to get as far away from the false exit as possible, as it gets closer to the right exit with every step. Since the goal of the agent was to find the right exit through the shortest path possible, VI can be considered a successful tool for the grid environment example given the transition

probabilities as described above and $\gamma = 1$. This particular example can be extended with other parameter values as well.

Pros and Cons. VI provides an optimal policy based on the premise that the iterations will eventually converge to states' optimal values. The convergence still holds even though the random initialized utilities at the beginning are not near their actual values. However, if a given environment can evolve to many different states and if the agent has a broad freedom concerning the number of available actions, the complexity of value iteration can cause long execution times, since the computational time of value iteration is $\mathcal{O}(|\mathcal{S}|^2 \times |\mathcal{A}|)$ [30]. The complexity grows quadratically in state space and linearly in action space.

2.4.2 Policy Iteration

The second approach to be presented is the policy iteration algorithm. This algorithm has the same objective as the value iteration, i.e. finding the optimal policy. However, policy iteration achieves this goal slightly differently. As opposed to the random initialization of state values in the value iteration, the policy iteration requires a randomly initialized policy, corresponding a random walk of the agent. The algorithm consists of the following two steps [28, pp. 656–657]:

Algorithm 2: Policy Iteration

Initialization
$\pi_0 \leftarrow$ randomly initialized state-action mapping
changed $\leftarrow$ True
$i \leftarrow 0$
while *changed* **do**
  Policy Evaluation
  foreach $s \in \mathcal{S}$ **do**
    $V_i(s) \leftarrow \mathcal{R}(s) + \gamma \sum_{s'} \mathbb{P}(s'|s, \pi_i(s)) V_i(s')$
  Policy Improvement
  foreach $s \in \mathcal{S}$ **do**
    $\pi_{i+1}(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s'} \mathbb{P}(s'|s, a) V_i(s)$
  if $\pi_{i+1} = \pi_i$ **then**
    changed $\leftarrow$ False
  else
    $i \leftarrow i + 1$
return π

Policy evaluation is the first step of the algorithm, where state values are obtained by a given policy, as described in (2.4). The difference to (2.4) is that the given policy is not optimal until convergence of the algorithm. The second step, called *policy improvement*, corresponds to the policy extraction from state values as introduced in (2.11).

Convergence. A policy iteration is guaranteed to return a strictly better policy than the previous iteration if the policy is not already optimal [24, pp. 76–79][31]. As there are finitely many policies (due to discrete modelling and finitely many states), the policy iteration algorithm eventually converges to an optimal policy.

An important difference between the policy evaluation and the Bellmann Equation (2.5) is the absence of the *max* operator, which makes the policy evaluation a linear operation. In fact, the utility extraction can be transformed to a linear system of equations:

$$\begin{aligned} V_i(s) &= \mathcal{R}(s) + \gamma \sum_{s'} \mathbb{P}(s'|s, \pi_i(s)) V_i(s') \; \forall s \in \mathcal{S} \\ &\Leftrightarrow (\mathbb{I} - \gamma \mathcal{P}) \boldsymbol{V} = \boldsymbol{\mathcal{R}} \end{aligned} \tag{2.12}$$

where $\mathcal{P} = (p_{i,j}) \in \mathbb{R}^{n \times n} = \mathbb{P}(s_j|s_i, \pi(s_i))$ is the probability matrix dependent on the policy π, with $n = |\mathcal{S}|$.

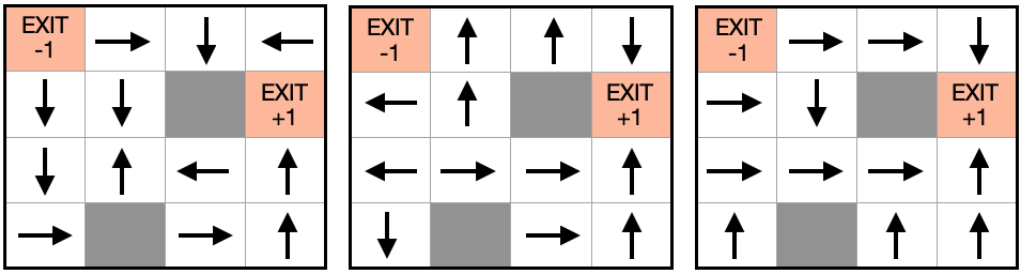


Figure 2.4: The policy mapping of the agent. From left to right: Initial random policy, 3rd iteration, last iteration

Example. The example grid environment set-up is the same as in the previous example, except that $\gamma = 0.99$. Figure 2.4 shows three iterations of the policy iteration algorithm. It can be observed that the agent's decisions converge to the optimal policy, as the action mapping in the last iteration results in the shortest possible path to the right exit state at $(1, 3)$. The crucial field in the grid environment is $(1, 1)$, where the agent has to decide on which path to take to the right exit. Both going up and down would result in the same path length to $(1, 3)$, however, the agent decides to go down to avoid the possibility of landing in the wrong exit. This 'behaviour' also coincides with the findings from the value iteration algorithm, as the lower field from $(1, 1)$ had a higher value than the upper field.

Pros and Cons. The policy iteration works with entirely randomized initial policies, that way no prior knowledge of the environment is required. One advantage of the algorithm is that the optimal policy can be obtained even though the state values are not exact, or not even good approximations of their real values. On the other hand, the computation time complexity of the policy iteration is $\mathcal{O}(|\mathcal{S}|^2 \times |\mathcal{A}| + |\mathcal{S}|^3)$ [30]. In comparison to the time complexity of value iteration, the policy iteration performs worse in state space. However, the policy iteration algorithm usually requires a small number of iterations, and in most cases less than the number of iterations it takes the value iteration algorithm to converge [32]. It comes down to the characteristics of the environment, such as the state space, to determine which of the iteration algorithms might be suitable for the task at hand.

3 Overview of Quantum Computing

3.1 Introduction

The term *quantum*, as it is used today, was first mentioned by Max Planck in his work in 1901 as 'elementary quantum of electricity, i.e. the electric charge of a single positively charged ion or electron' [33]. He postulated that electromagnetic radiation is emitted in quanta; in *blocks of energy*.

The information instance of a quantum computer is a *qubit*, as opposed to a classical bit in a classical computer. Handling quantum states as the smallest entity of information originates from Schumacher's work in 1996 [34]. Feynman pointed out the impossibility of simulating quantum systems probabilistically in a classical computer; one needs a computer built on quantum mechanical elements [5, pp. 474–477]. This thought brought life to the concept of *quantum supremacy* [35], where the goal is to show that a computational problem, which is impossible for a classical computer to solve in a feasible time, is proven to be solved by a state-of-the-art quantum computer. Recent researches in quantum supremacy include [36, 37].

Without the underlying physics of quantum computing, we aim to introduce important building blocks for many quantum computational operations, including the HHL Quantum Algorithm which will be discussed in the next chapter. Starting with some notations and representations, we will build upon these fundamentals to show more complex structures.

3.2 Qubit Representation

A special representation for qubits was needed in order to combine various ways of dealing with quantum mechanics [38]: One that requires complex vectors and linear operators defined on them, and one that requires abstract calculations between the vectors and operators. Also a third way, which enables calculations with representatives of these elements. To overcome the inconvenience of switching between different notations, the Bra-Ket notation was introduced.

Bra-Ket. Quantum mechanics make use of a complex and complete[1] vector space $\mathcal{H}$ with an inner product, called Hilbert space. A vector ψ in $\mathcal{H}$ is denoted as a *ket*, i.e.

[1] A vector space is complete if every Cauchy sequence of vectors converges to a vector with respect to the norm defined in the space [39].

the vector is accompanied by $|\cdot\rangle$: $|\psi\rangle \in \mathcal{H}$. Every ket comes with its dual, namely its conjugate-transpose vector denoted as a *bra*, i.e. the vector is accompanied by $\langle\cdot|$ [38]:

$$|\psi\rangle^\dagger = \begin{pmatrix} \psi_0 \\ \vdots \\ \psi_{n-1} \end{pmatrix}^\dagger = \begin{pmatrix} \psi_0^* & \dots & \psi_{n-1}^* \end{pmatrix} = \langle\psi| \in \mathcal{H}^\dagger \tag{3.1}$$

where we assumed for simplicity that the Hilbert space is finite and n-dimensional. $\dagger$ denotes the conjugate transpose of a vector.

Inner product. The bra-ket notation is quite convenient when denoting the complex inner product. One brings the two elements of the inner product together by enclosing them with a *bra-ket* [40, p. 65]:

$$\langle\varphi, \psi\rangle = \sum_{i=0}^{N-1} \varphi_i^* \psi_i = \langle\varphi|\psi\rangle \tag{3.2}$$

Kronecker product. The Kronecker product $\otimes$ is used to represent a vector space by expanding two vector spaces, n-dimensional V and m-dimensional W, to create a $n \cdot m$-dimensional vector space $V \otimes W$:

$$\varphi \in V,\ \psi \in W \Longrightarrow \varphi \otimes \psi = \begin{pmatrix} \varphi_0\psi_0 \\ \varphi_0\psi_1 \\ \vdots \\ \varphi_0\psi_{m-1} \\ \varphi_1\psi_0 \\ \vdots \\ \varphi_{n-1}\psi_{m-1} \end{pmatrix} \tag{3.3}$$

In general, the Kronecker product is defined on arbitrary matrices. Let $A \in \mathbb{C}^{n\times m}$, $B \in \mathbb{C}^{k\times l}$, then [40, p. 74]:

$$A \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1m}B \\ \vdots & \ddots & \vdots \\ a_{n1}B & \dots & a_{nm}B \end{pmatrix} \in \mathbb{C}^{nk\times ml} \tag{3.4}$$

Hilbert Space Basis. A n-qubit is defined in the 2^n-dimensional Hilbert space. This Hilbert space can be generated through 2^n linearly independent basis vectors. Every

vector belonging to the Hilbert space can be represented with the linear combination of these basis vectors:

$$\psi \in \mathcal{H} \Longrightarrow \psi = \sum_{i \in \{0,1\}^n} \gamma_i \left| i \right\rangle \tag{3.5}$$

where $\gamma_i \in \mathbb{C} \, \forall i \in [n]$ and $\left| i \right\rangle = \left| q_0, \ldots, q_{n-1} \right\rangle = \left| q_0 \right\rangle \otimes \cdots \otimes \left| q_{n-1} \right\rangle$ with $q_j \in \{0,1\} \, \forall j \in [n]$. The standard generating vector set for one-qubit systems is the so called *computational basis*: $\{\left|0\right\rangle, \left|1\right\rangle\} = \{\binom{1}{0}, \binom{0}{1}\}$. The computational basis can be extended through Kronecker product to generate an arbitrary n-qubit system. For example, a two-qubit system can be generated by $\left|00\right\rangle, \left|01\right\rangle, \left|10\right\rangle, \left|11\right\rangle\} = \{\left|0\right\rangle \otimes \left|0\right\rangle, \left|0\right\rangle \otimes \left|1\right\rangle, \left|1\right\rangle \otimes \left|0\right\rangle, \left|1\right\rangle \otimes \left|1\right\rangle\} = \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right\}$.

We have now covered important mathematical properties of the space where quantum mechanics live. When it comes down to what a *qubit* really is, we will continue to treat it as an abstract object belonging to the Hilbert space. In reality, qubits correspond to real physical *states*. Therefore, we are going to refer to the 'vectors' in the Hilbert space as quantum states. However, there are still some disagreements about what a real quantum state is; whether or not the infinitely many possibilities are merely to be observed as information or a real physical state [41]. Examples for a 2-dimensional quantum state in the real world include the position of electron in a hydrogen molecule, a spin one-half particle in a magnetic field and a spinning electron in a magnetic field [42].

3.2.1 Measurement

We distinguish between the pre-measurement state and and post-measurement state of a qubit. Before measurement, a single qubit state $\left|\psi\right\rangle = \alpha \left|0\right\rangle + \beta \left|1\right\rangle$ in the computational basis can be anywhere in an infinitely large space. To illustrate this, Figure 3.1 shows the position of the state $\left|\psi\right\rangle$ in a 3-dimensional sphere, called the Bloch sphere. The 3-dimensional visualization is derived from several instruments, including polar coordinates, unit circle and Euler's identity [43]. For simplicity, we omit the derivation and merely show the equivalent expressions:

$$\left|\psi\right\rangle = \alpha \left|0\right\rangle + \beta \left|1\right\rangle = \cos\frac{\theta}{2} \left|0\right\rangle + e^{i\varphi} \sin\frac{\theta}{2} \left|1\right\rangle \tag{3.6}$$

This representation indicates that a qubit can indeed be in infinitely many states, called *superpositions* of basis states. However, once the qubit is observed, its state *collapses* with probability $|\alpha|^2$ and $|\beta|^2$ to the basis states $\left|0\right\rangle$ and $\left|1\right\rangle$, respectively.

Generally, the post-measurement state of a n-qubit system can only be in one of the 2^n basis states. Since the square absolute values of the amplitudes for the basis states

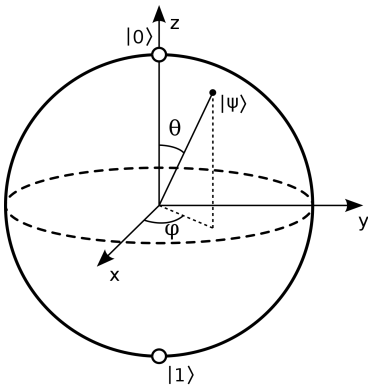


Figure 3.1: An arbitrary state $|\psi\rangle$ on the Bloch sphere [44]

represent the probabilities of an occurrence of a basis state, they sum up to 1:

$$|\psi\rangle = \sum_{i \in \{0,1\}^n} \alpha_i |i\rangle \Longrightarrow \sum_{i \in \{0,1\}^n} |\alpha_i|^2 = 1 \tag{3.7}$$

In an n-qubit system, each qubit measurement will change the state irreversibly, leaving the system in a post-measurement state. For instance, measuring the basis state 1 in the first qubit in a 2-qubit system would result in the following post-measurement state [40, p. 16]:

$$\begin{aligned} |\psi\rangle &= \alpha_{00} |00\rangle + \alpha_{01} |01\rangle + \alpha_{10} |10\rangle + \alpha_{11} |11\rangle \\ &\longrightarrow \frac{\alpha_{10} |10\rangle + \alpha_{11} |11\rangle}{\sqrt{|\alpha_{10}|^2 + |\alpha_{11}|^2}} \end{aligned} \tag{3.8}$$

Dividing the terms by the amplitudes, i.e. the *normalization* of the state, ensures that the remaining probabilities sum up to 1.

We have now covered the most important aspects of representing a qubit system. In the next sections, we are going to define operations on qubits and present relevant algorithms building upon these operations, which will be crucial in understanding our quantum computing application.

3.3 Quantum Gates

A qubit system consists of qubits as inputs, which can be transformed into an output. In other terms, an input state can be processed by means of some operations, called quantum gates, into an output state. Applying these gates are represented as matrix operations.

Quantum gates are unitary transformations[2], often denoted as U. The unitarity of quantum gates is based on the probability condition (3.7). In fact, every unitary matrix stands for a valid quantum gate [40, p. 18]. Unitary transformations are also characterised by being reversible.

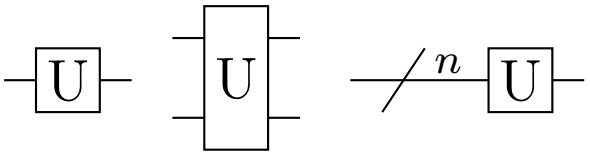


Figure 3.2: A unitary operation action on a single qubit (left), on two qubits (middle) and on n qubits (right)

Figure 3.2 shows how a unitary operation is denoted. A single line (or *wire*) represents one qubit as input, which is then processed by U. 2 or more qubits can be connected through multiple wires. However, it is often more convenient to bundle the wires and note how many qubits are to be transformed through them. A system of a sequence of qubit transformations is called a quantum circuit.

Quantum gates can be categorized as single-qubit gates and multiple-qubit gates. We first show the most relevant single-qubit gates, followed by multiple-qubit gates and how gates can be grouped together to represent a whole circuit.

3.3.1 Single-Qubit Gates

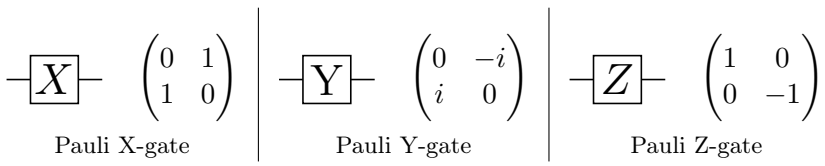


Figure 3.3: Pauli gates and their matrix equivalents [40, p. xxx]

Pauli Gates In the following, the qubit transformations corresponding to Pauli gates are listed:

1. Pauli X-gate reverses the qubit, which corresponds to a rotation of the qubit around the x-axis of the Bloch sphere by π: $|0\rangle \xrightarrow{X} |1\rangle$ and $|1\rangle \xrightarrow{X} |0\rangle$.

2. Pauli Y-gate rotates the qubit around the y-axis of the Bloch sphere by π: $|0\rangle \xrightarrow{Y} i\,|1\rangle$ and $|1\rangle \xrightarrow{Y} i\,|0\rangle$.

[2] A matrix U is unitary if $U^\dagger U = UU^\dagger = \mathbb{I}$.

3. Pauli Z-gate rotates the qubit around the z-axis of the Bloch sphere by π: $|0\rangle \xrightarrow{Z} |0\rangle$ and $|1\rangle \xrightarrow{Z} -|1\rangle$.

Pauli gates form a basis for the single qubit system together with the identity matrix $\mathbb{I}$, i.e., any single qubit gate can be generated with the matrix space spanned by the Pauli gates and $\mathbb{I}$.

Hadamard Gate The Hadamard gate puts a qubit into a superposition with equal probabilities on the basis states $|0\rangle$ and $|1\rangle$. Figure 3.4 shows how the Hadamard gate transforms a single qubit.

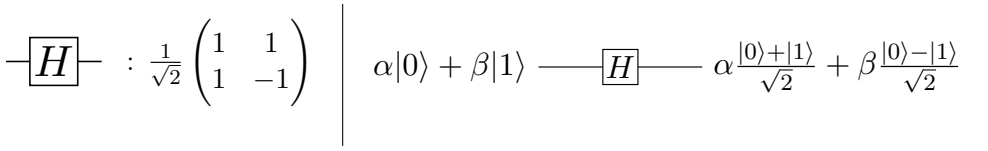


Figure 3.4: Hadamard gate representation and its application on an arbitrary qubit input [40, pp. xxx, 19]

3.3.2 Multiple-Qubit Gates

For an n-qubit system, an eligible quantum gate for a simultaneous operation on n-qubits lies in $\mathbb{C}^{2^n \times 2^n}$, since the gate is applied on a 2^n-dimensional vector. However, an n-qubit system may consist of arbitrary gates, not necessarily defined for all of the qubits; the gates may apply on an arbitrary subset of qubits.

In the 3-qubit system example given in Figure 3.5, the gate A is being applied on two input qubits $|00\rangle$, $A \in \mathbb{C}^{4\times4}$. Moreover, the quantum circuit has two single-qubit gates applied on the 3rd qubit, i.e. C, $B \in \mathbb{C}^{2\times2}$; $|0\rangle \to B|0\rangle \to CB|0\rangle$. The whole circuit can be described as $U \in \mathbb{C}^{8\times8} = A \otimes (CB)$, i.e. $|000\rangle \to A|00\rangle\, B|0\rangle \to A|00\rangle\, CB|0\rangle$. The Kronecker product between kets are often omitted for better readability, as is also the case in the example representation.

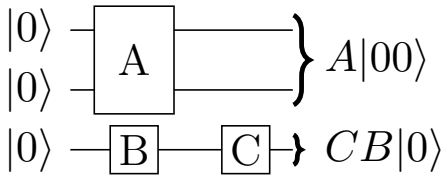
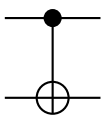


Figure 3.5: A 3-qubit circuit example

Instead of combining the operations separately by the Kronecker product, one can 'transform' every gate in the example circuit into a 3-qubit gate. Since the first two

qubits are left unaffected by the operations realized by the gates B and C, they can be thought as if they would undergo an identity element; namely the identity matrix. This way the gates B and C can now be transformed as $\mathbb{I} \otimes \mathbb{I} \otimes B$ and $\mathbb{I} \otimes \mathbb{I} \otimes C$, respectively. Analogously, the gate A can be extended to $A \otimes \mathbb{I}$. The circuit representation is therefore $U = (\mathbb{I} \otimes \mathbb{I} \otimes C)(\mathbb{I} \otimes \mathbb{I} \otimes B)(A \otimes \mathbb{I})$ and is equal to the representation derived above.

CNOT Gate The CNOT-gate (*controlled*-not gate) is applied on 2 qubits. One of the qubits is the control qubit, denoted with a black point, the other is the target qubit, denoted with an `xor`-symbol. Whenever the control qubit is $|1\rangle$, the target qubit is flipped: $|ab\rangle \xrightarrow{\texttt{CNOT}} |a, a \oplus b\rangle$.



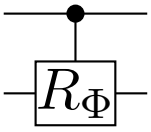
$$: \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 3.6: CNOT gate representation [40, p. xxx]

The CNOT-gate is a controlled-gate, which is a term used to describe operations on qubits that are controlled by other qubits. The control qubits may each be a $|0\rangle$ or a $|1\rangle$, depending on the desired operation.

Controlled-R_Φ Gate This controlled operation extends the single-qubit R_Φ-Gate, which is commonly referred to as *phase shifting gate*, since the operation shifts the qubit around the z-axis in the Bloch sphere (3.6):

$$R_\Phi(\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle) = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi+\Phi}\sin\frac{\theta}{2}|1\rangle \tag{3.9}$$



$$: \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\Phi} \end{pmatrix}$$

Figure 3.7: Controlled rotation gate representation [45]

The controlled-R_Φ Gate acts only if the input state is at the basis state $|11\rangle$: $|11\rangle \xrightarrow{R_\Phi} e^{i\Phi}|11\rangle$.

3.4 Quantum Algorithms

3.4.1 Discrete Quantum Fourier Transformation

Discrete Fourier Transformation maps the time domain of a set of discrete values, often referred to as signals or amplitudes, into a frequency domain, giving a different but useful representation for the same information [46]. The classical discrete Fourier Transformation returns the frequencies y_k, $k \in \{0 \dots N-1\}$ given N signal values $x_0, \dots, x_{N-1}$:

$$y_k := \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i j k / N} \tag{3.10}$$

The quantum equivalent of Fourier Transformation is the basis change between the computational basis $|x\rangle$ and the 'Fourier basis' $|y\rangle$, i.e.

$$|x\rangle \longrightarrow \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{2\pi i x y / N} |y\rangle \tag{3.11}$$

Assuming $N = 2^n$ for some $n \in \mathbb{N}$, the input $|x\rangle$ is transformed as follows [40, p. 218]:

$$\begin{aligned}
|x\rangle \longrightarrow & \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{2\pi i x y / 2^n} |y\rangle \\
\overset{*}{=} & \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{2\pi i x (\sum_{k=1}^{n} y_k / 2^k)} |y_1 \dots y_n\rangle \\
= & \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} \prod_{k=1}^{n} e^{2\pi i x y_k / 2^k} |y_1 \dots y_n\rangle \\
= & \frac{1}{\sqrt{N}} \sum_{y_1=0}^{1} \cdots \sum_{y_n=0}^{1} \prod_{k=1}^{n} e^{2\pi i x y_k / 2^k} |y_1 \dots y_n\rangle \\
\overset{**}{=} & \frac{1}{\sqrt{N}} \bigotimes_{k=1}^{n} (|0\rangle + e^{2\pi i x / 2^k} |1\rangle)
\end{aligned} \tag{3.12}$$

$(*)$ uses the binary representation of y, i.e. $y = (y_1 \dots y_n)_2 = 2^{n-1} y_1 + 2^{n-2} y_2 + \cdots + y_n 2^0$, $y/2^n = 2^{-1} y_1 + 2^{-2} y_2 + \cdots + 2^{-n} y_n$. Hence, the N Fourier basis vectors are represented by 2^n-dimensional computational basis vectors.

At the last step of the equivalence conversion, $(**)$ uses the fact that $e^{2\pi i x y_k / 2^k}$ is only effective whenever $y_k = 1$, otherwise e^0 is 1.

Figure 3.8 shows the circuit representation of quantum discrete Fourier Transformation. The intermediate states with respect to the circuit slices depicted in Figure 3.8 are as follows [40, p. 219]:

1. The input $|x_1 \dots x_n\rangle$ corresponds to the computational basis for the binary representation of one 'signal' which is to be transformed.

2. The Hadamard gate is applied to the first qubit: $|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + (-1)^{x_1} |1\rangle) |x_2 \dots x_n\rangle$. We can rewrite $(-1)^{x_1}$ as $e^{\pi i x_1}$, since the new expression evaluates to 1 when $x_1 = 0$ and to -1 when $x_1 = 1$ due to $e^{\pi i} = -1$: $|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{\pi i x_1} |1\rangle) |x_2 \dots x_n\rangle$.

3. The controlled rotation operator is applied to the first qubit. R_k is a more convenient representation of the rotation operator introduced at 3.7 with $\Phi = 2\pi i/2^k$. The circuit state after the rotation is: $|\psi_2\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(\frac{x_1}{2} + \frac{x_2}{2^2})} |1\rangle) |x_2 \dots x_n\rangle$.

4. After $n-1$ controlled rotation operators applied to the first qubit: $|\psi_3\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(\frac{x_1}{2} + \frac{x_2}{2^2} + \dots + \frac{x_n}{2^n})} |1\rangle) |x_2 \dots x_n\rangle$.

5. The end state of the circuit, after $x_2, \dots, x_n$ have been transformed through the Hadamard and rotation gates, is:

$$\begin{aligned} |\psi_4\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i x/2^n} |1\rangle) \otimes \dots \otimes \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i x/2} |1\rangle) \\ &= \frac{1}{\sqrt{2^n}} \bigotimes_{k=1}^{n} (|0\rangle + e^{2\pi i x/2^k} |1\rangle) \end{aligned} \tag{3.13}$$

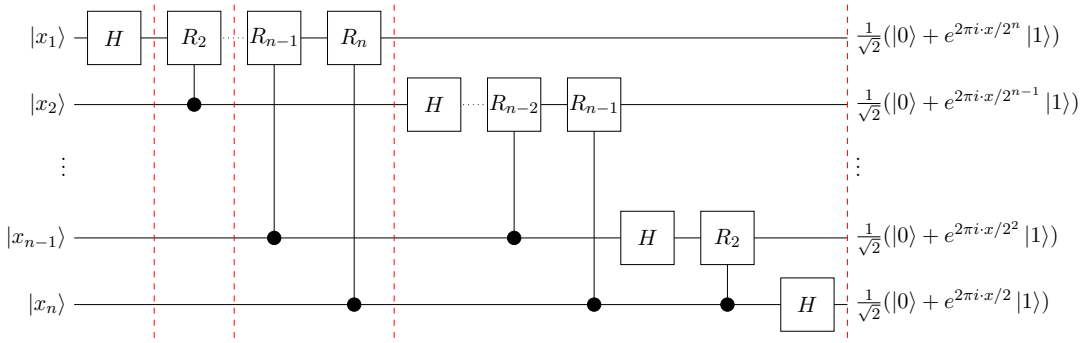


Figure 3.8: QDFT Circuit [40, p. 219]

The end state ψ_4 is thus equivalent to the transformation derived at (3.12).
The inverse discrete Quantum Fourier Transformation is analogous:

$$|y\rangle \longrightarrow \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} e^{-2\pi i x y/N} |x\rangle \tag{3.14}$$

3.4.2 Phase Estimation

A use case of Quantum Fourier Transformation (QFT) is the Phase Estimation, where the problem statement is to find the eigenvalue of a unitary operator U with eigenvector ψ, or more precisely, to find the phase φ s.t. $U\psi = e^{2\pi i\varphi}\psi$, since for the eigenvalues of U it holds:

$$\begin{aligned} &U\psi = \lambda\psi \iff \psi^\dagger U^\dagger = \lambda^*\psi^\dagger \\ &\implies \psi^\dagger U^\dagger U\psi = \lambda^*\psi^\dagger\lambda\psi \implies |\lambda|^2 = 1 \end{aligned} \tag{3.15}$$

where we used that U is unitary and $\psi^\dagger\psi = 1$.
The Phase Estimation circuit in Figure 3.9 transforms the input as following [40, p. 222]

$$\begin{aligned} |0\rangle^{\otimes t}|\psi\rangle &\xrightarrow{H} \frac{1}{\sqrt{2^t}}(|0\rangle+|1\rangle)^{\otimes t}\otimes|\psi\rangle = \frac{1}{\sqrt{2^t}}\Big((|0\rangle+|1\rangle)^{\otimes t-1}\otimes|\psi\rangle\Big)\otimes(|0\rangle|\psi\rangle+|1\rangle|\psi\rangle) \\ &\xrightarrow{U^{2^0}} \frac{1}{\sqrt{2^t}}\Big(|0\rangle+|1\rangle)^{\otimes t-1}\otimes|\psi\rangle\Big)\otimes(|0\rangle|\psi\rangle+|1\rangle U|\psi\rangle) \\ &= \frac{1}{\sqrt{2^t}}(|0\rangle+|1\rangle)^{\otimes t-1}(|0\rangle+e^{2\pi i\varphi}|1\rangle)\otimes|\psi\rangle \\ &\xrightarrow{U^{2^2}} \frac{1}{\sqrt{2^t}}(|0\rangle+|1\rangle)^{\otimes t-2}(|0\rangle+e^{2\pi i\varphi 2}|1\rangle)(|0\rangle+e^{2\pi i\varphi}|1\rangle)\otimes|\psi\rangle \\ &\longrightarrow \dots \\ &\xrightarrow{U^{2^{t-1}}} \frac{1}{\sqrt{2^t}}(|0\rangle+e^{2\pi i\varphi 2^{t-1}}|1\rangle)(|0\rangle+e^{2\pi i\varphi 2^{t-2}}|1\rangle)\otimes\dots\otimes(|0\rangle+e^{2\pi i\varphi 2^0}|1\rangle)\otimes|\psi\rangle \end{aligned} \tag{3.16}$$

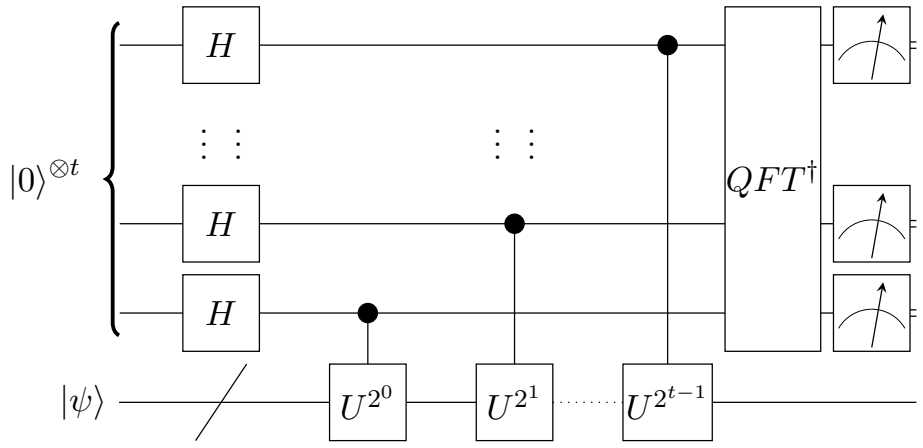


Figure 3.9: Phase Estimation Circuit [40, p. 222]

The derived end-state (3.16) of the Phase Estimation circuit is equal to the end-state of the QFT (3.13), if a 2^t-dimensional QFT-input x is replaced with $2^t\varphi$. Hence, if φ can be represented with exactly t qubits, one can obtain the exact representation of the eigenvalue of U for the eigenstate ψ [40, p. 222] by reversing the QFT (3.14).

The Phase Estimation procedure is a *subroutine*, meaning that it can be completed with other circuits, as the procedure is not responsible for preparing the unitary operations U^{2^j}.

3.5 Hamiltonian Simulation

The time dynamics of a closed quantum system governed by a Hamiltonian H is given by the Schrödinger equation [47]:

$$i\hbar\frac{d\left|\psi\right\rangle}{dt} = H\left|\psi\right\rangle \tag{3.17}$$

The Schrödinger equation shows that once the Hamiltonian H of a system is known, it is possible to extract the state at a certain time point. Moreover, if the Hamiltonian is time-independent, the time evolution of $|\psi(t)\rangle$ may be written as [40, p. 206]

$$\left|\psi(t)\right\rangle = e^{-iHt/\hbar}\left|\psi(0)\right\rangle \tag{3.18}$$

with $|\psi(0)\rangle$ as the initial state and $e^{-iHt/\hbar}$ as the unitary operator of the system. According to [48], simulating a quantum system with the corresponding Hamiltonian on a quantum computer requires N steps, whereas it takes a classical computer exponentially many steps with respect to N. This exponential speed-up in simulation of quantum systems is considered as the main motivation of quantum computing.
Simulating the Hamiltonian of a quantum system has the following objective [49]: Finding a quantum algorithm which is able to produce a unitary operator U, given the simulation error ϵ and time $t \in \mathbb{R}$ such that $||U - e^{-iHt}|| \leq \epsilon$, with $||$ the spectral norm and e^{-iHt} the desired unitary operator.

4 The HHL Quantum Algorithm

4.1 Introduction

In Chapter 2, we have formulated the Policy Evaluation step of the Policy Iteration Algorithm as a linear system of equations (LSE). In this chapter, we are going to investigate the HHL Quantum Algorithm [16], which is named after its authors Harrow, Hassidim and Lloyd and was introduced as a quantum solver for LSE. The algorithm offers an estimation for the expected value of the solution vector of a given linear system of equations in a time complexity of $\widetilde{\mathcal{O}}(\log(N)s^2\kappa^2/\epsilon)$, where κ is the condition number of the given matrix with dimension N, s the sparsity of the matrix, i.e. the maximum number of non-zero entries in the rows of the matrix and ϵ the desired accuracy deviation. The best known classical algorithm for solving linear systems of equations with a sparse matrix A, the *Conjugate Gradient Descent*, has a time complexity of $\mathcal{O}(Ns\kappa\log(1/\epsilon))$ [50]. Hence, the HHL Algorithm performs exponentially faster regarding the matrix dimension N, given that the matrix is well-conditioned and sparse.

Starting from the classical equivalent of the quantum solution, we are going to break down the algorithm into steps and show how it works, using the tools that were covered briefly in Chapter 3. We will discuss the prerequisites of the algorithm. This short discussion will be a building block for the upcoming Chapter 5, as we analyze our reinforcement learning application.

Problem statement. The HHL algorithm solves a linear system of equations defined on a Hermitian matrix $A \in \mathbb{C}^{N\times N}$, i.e. $A^\dagger = A$:

$$Ax = b, \quad x, b \in \mathbb{C}^N \tag{4.1}$$

Since A satisfies the normality condition, i.e. $A^\dagger A = AA^\dagger$, there exists a spectral decomposition of A [40, p. 72]:

$$A = \sum_j \lambda_i \left|u_j\right\rangle \left\langle u_j\right| \tag{4.2}$$

with λ_i as eigenvalues of A and $|u_j\rangle$ the corresponding eigenvectors. The spectral decomposition results in a diagonal matrix, thus the inverse of A is obtained by inverting

the eigenvalues. The vector $|b\rangle$ can be expressed in the eigenbasis of A:

$$A^{-1} = \sum_j \lambda_j^{-1} |u_j\rangle \langle u_j| , \quad |b\rangle = \sum_j \langle u_j|b_j\rangle |u_j\rangle = \sum_j \beta_j |u_j\rangle , \quad \beta_j \in \mathbb{C} \tag{4.3}$$

Hence, one can obtain a representation for the solution vector $|x\rangle$:

$$|x\rangle = A^{-1} |b\rangle = \sum_j \lambda_j^{-1} \beta_j |u_j\rangle \tag{4.4}$$

The HHL Algorithm takes advantage of the spectral decomposition of the Hermitian input matrix A by using Phase Estimation (3.4.2) to represent $|b\rangle$ in the eigenbasis of A, inverting the resulting eigenvalues and finally reversing the Phase Estimation to obtain the solution vector $|x\rangle$.

In the next section, we go through the steps of the algorithm in detail.

4.2 The Algorithm

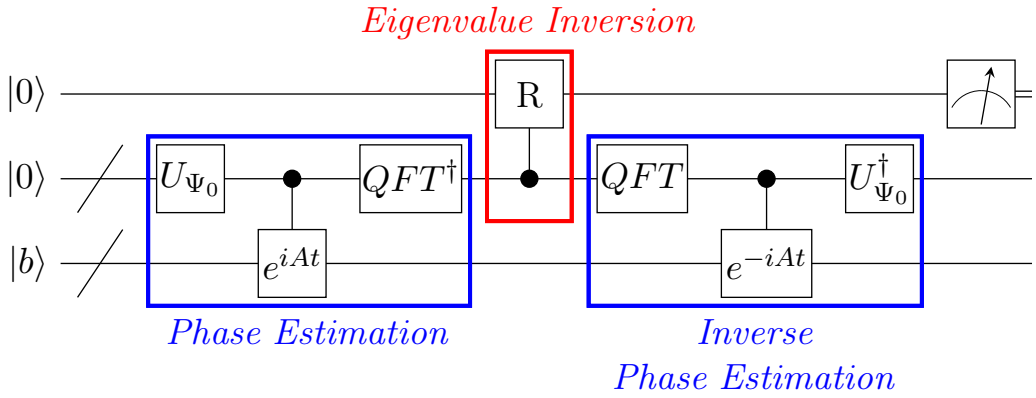


Figure 4.1: Schematic HHL quantum circuit

The circuit depicted in Figure 4.1 is a simplified circuit representation of the HHL Algorithm. The simplified circuit transformations given the input $|0\rangle |0\rangle |b\rangle$ can be summarized by the following steps [16][51, p. 6]:

1. The initial state of the HHL Algorithm is $|0\rangle^A |0\rangle^C |b\rangle^I$, where C (*control*) and I (*input*) symbolize the circuit registers for storing the $|0\rangle$-qubits and qubits for representing the vector b, respectively. The first register A (*ancilla*) contains only one qubit, which will be used to apply a measurement.

2. The C-register is transformed into a super-position state:
   $|0\rangle^C \xrightarrow{U_{\Psi_0}} \sqrt{\frac{2}{T}} \sum_{\tau}^{T-1} \sin\frac{\pi(\tau+\frac{1}{2})}{T} |\tau\rangle =: \Psi_0^C$. This super-position state is comparable with the super-position state realized by Hadamard gates in the Phase Estimation Algorithm, however, the HHL Algorithm uses a different approach in order to minimize a quadratic loss function [16].

3. The conditional Hamiltonian operator is applied to the registers C and I:
   $(\sum_{\tau=0}^{T-1} |\tau\rangle \langle\tau|^C \otimes e^{iA\tau t_0/T}) \cdot (|\Psi_0\rangle^C \otimes |b\rangle^I)$.

4. Inverse Quantum Fourier Transformation is applied to the register C to complete the phase estimation, which leaves the state in $|0\rangle^A \sum_{j=1}^{N} \beta_j |\widetilde{\lambda_j}\rangle^C |u_j\rangle^I$.

5. The rotation on the ancilla qubit controlled by the C-register yields:
   $\sum_{j=1}^{N} \beta_j |u_j\rangle^I |\widetilde{\lambda}\rangle^C \left(\sqrt{1 - \frac{C^2}{\widetilde{\lambda_j^2}}} |0\rangle + \frac{C}{\widetilde{\lambda_j}} |1\rangle\right)^A$. C is a normalizing constant which is $\mathcal{O}(1/\kappa)$. The rotation operator is $R_y(\theta) = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix}$ with $\theta = 2\arcsin(C/\widetilde{\lambda_j})$ [51].

6. The register C is uncomputed through the Inverse Phase Estimation, leaving the circuit in the state $\sum_{j=1}^{N} \beta_j |u_j\rangle^I |0\rangle^C \left(\sqrt{1 - \frac{C^2}{\widetilde{\lambda_j^2}}} |0\rangle + \frac{C}{\widetilde{\lambda_j}} |1\rangle\right)^A$. The 'uncomputation' will be explained in subsection 4.2.2.

7. Measuring the ancilla qubit and obtaining $|1\rangle$ results in the state
   $\sqrt{\frac{1}{\sum_{j=1}^{N} C^2|\beta_j|^2/|\lambda_j|^2}} \sum_{j=1}^{N} \beta_j \frac{C}{\lambda_j} |u_j\rangle$, which differs from the solution vector x only by normalization of the end-state.

4.2.1 Hamiltonian Evolution and Phase Estimation

After having prepared the inputs, the algorithm proceeds to find the eigenvalues of A. In order to do that, the phase estimation procedure requires a unitary operator and its efficient preparation. Since A is Hermitian and thus has the spectral decomposition introduced at (4.2), the unitary operator e^{iAt} has the eigenvalues $e^{i\lambda_j t}$ and eigenvectors u_j:

$$
\begin{aligned}
e^{iAt} &= \sum_{k=0}^{\infty} \frac{(iAt)^k}{k!} \\
&= \mathbb{I} + iAt - \frac{A^2t^2}{2!} - \frac{iA^3t^3}{3!} + \dots \\
&= \mathbb{I} + it(\lambda_1 \cdot |u_1\rangle \langle u_1| + \lambda_2 \cdot |u_2\rangle \langle u_2| + \dots) - t^2 \frac{(\lambda_1^2 \cdot |u_1\rangle \langle u_1| + \lambda_2^2 \cdot |u_2\rangle \langle u_2|)}{2!} - \dots \\
&= \sum_{k=0}^{\infty} \frac{(i\lambda_1 t)^k}{k!} |u_1\rangle \langle u_1| + \sum_{k=0}^{\infty} \frac{(i\lambda_2 t)^k}{k!} |u_2\rangle \langle u_2| + \dots + \sum_{k=0}^{\infty} \frac{(i\lambda_n t)^k}{k!} |u_n\rangle \langle u_n| \\
&= e^{i\lambda_1 t} |u_1\rangle \langle u_1| + e^{i\lambda_2 t} |u_2\rangle \langle u_2| + \dots + e^{i\lambda_n t} |u_n\rangle \langle u_n| \\
&= \sum_j e^{i\lambda_j t} |u_j\rangle \langle u_j|
\end{aligned}
\tag{4.5}
$$

The unitary operator e^{iAt} can be used to get the eigenvalue information from A by simulating it with Hamiltonian Evolution discussed in 3.5. In the HHL Algorithm, the time t is chosen to be t_0/T, with T being the number of computational steps regarding the simulation, and t_0 being the time interval, hence, t_0/T corresponds to a single time-step of the simulation [52, p. 32]. The HHL Algorithm requires T to be a large number, and t_0 to lie in $\mathcal{O}(\kappa/\epsilon)$.

4.2.2 Uncomputation

When constructed for obtaining a function value $f(x)$ with the given input x, quantum algorithms may produce a so called 'garbage output', which is dependent on x [53]. Most of the times, the garbage register needs to be *uncomputed* to reuse it or to prevent any unwanted interferences if the garbage register is entangled with the register containing the solution. The term uncomputation refers to the procedure of reversing the computation for obtaining $f(x)$ whilst preserving the desired output. Because of no-deleting theorem [54], one cannot simply set the garbage registers to $|0\rangle$; there is no unitary operation which can accomplish it in general. Therefore, the concept of uncomputation trick is widely used in quantum algorithms.

In the HHL Algorithm, we need to uncompute the control register. The control register corresponds to the eigenvalues of A. To elaborate on how uncomputation works, we briefly summarize the discussion in [52, pp. 22, 23]. Given a quantum algorithm which takes the input $|x\rangle |0\rangle |0\rangle$ and produces $\sum_y \alpha_y |x\rangle |y\rangle |f(x)\rangle$ with $|f(x)\rangle$ as the solution, the garbage state is $\sum_y \alpha_y |y\rangle$. An additional register added, which undergoes a CNOT-operation controlled on the $|f(x)\rangle$ register, leaving the whole state in $\sum_y \alpha_y |x\rangle |y\rangle |f(x)\rangle |f(x)\rangle$. If now the first registers are inversed, meaning that the inverse of the unitary operations which

were used to compute $|f(x)\rangle$ is applied, the state becomes $|x\rangle|0\rangle|0\rangle|f(x)\rangle$. Swapping the last two registers results in $|x\rangle|0\rangle|f(x)\rangle|0\rangle$, which is the final state without the garbage output.

4.3 Prerequisites and Caveats

The HHL Algorithm comes with some important restrictions regarding the inputs, which will be crucial in our reinforcement learning application. Therefore, we address these restrictions briefly in this chapter, in order to refer to them when we present the application simulation as well as the workarounds and input compatibility analyses in Chapter 5.

Hermitian input matrix. The input matrix has to be Hermitian, so that one can make use of its spectral decomposition. [16] suggests defining a new matrix $\widetilde{A}$, such that

$$\widetilde{A} = \begin{pmatrix} 0 & A \\ A^\dagger & 0 \end{pmatrix} \in \mathbb{C}^{2N \times 2N}, \quad \widetilde{x} = \begin{pmatrix} 0 \\ x \end{pmatrix} \in \mathbb{C}^{2N}, \quad \widetilde{A}\widetilde{x} = \begin{pmatrix} b \\ 0 \end{pmatrix} \in \mathbb{C}^{2N} \tag{4.6}$$

However, transforming the input matrix is not sufficient on its own. One also needs to make sure that the matrix can be simulated efficiently, which will be discussed in Chapter 5. When the simulation itself does not perform in $\mathcal{O}(\log(n))$, the exponential speed-up of the HHL Algorithm is lost.

Condition number κ. The condition number of a matrix is the ratio between its smallest and biggest singular value. If the matrix is normal[1], which is the case with Hermitian matrices, then $\kappa = \frac{|\lambda_{max}|}{|\lambda_{min}|}$. The HHL Algorithm performs quadratically with the condition number κ. If the matrix is ill-conditioned, then the exponential speed-up may be lost; since κ has to lie in $\mathcal{O}(\log(N))$ in order to achieve an exponential speed-up [16].

We also note that a non-Hermitian input matrix A with a condition number $\text{cond}(A)$ has the same condition number after having been transformed to A' as described in (4.6):

$$\text{cond}(A') = \frac{|\lambda_{max}^{A'}|}{|\lambda_{min}^{A'}|} = \frac{\sigma_{max}^{A}}{\sigma_{min}^{A}} = \text{cond}(A) \tag{4.7}$$

Eigenvalue range. The HHL Algorithm assumes that the singular values of the input matrix lie between $1/\kappa$ and 1. This restriction requires a priori knowledge about the range of the singular values, which may not be available with every use-case. In addition,

[1] A matrix A is normal if $AA^\dagger = A^\dagger A$.

transforming an arbitrary, non-Hermitian matrix into a Hermitian matrix as described in (4.6) results in the following:

$$\widetilde{A} := \begin{pmatrix} 0 & A \\ A^\dagger & 0 \end{pmatrix}, \quad \det(\widetilde{A} - \mathbb{I}x) = 0 \Leftrightarrow \det \begin{pmatrix} -\mathbb{I}x & A \\ A^\dagger & -\mathbb{I}x \end{pmatrix} = \det(\mathbb{I}x^2 - AA^\dagger) = 0 \quad (4.8)$$

Hence, the new matrix $\widetilde{A}$ has the square root of the eigenvalues of $AA^\dagger$, which are the positive and negative singular values of the input matrix A. However, the negative eigenvalues pose a problem for the HHL Algorithm, since the eigenvalues are mapped to $2\pi k/t_0$ before the controlled rotation. Both the index k and the time interval t_0 are positive, which indicates that the eigenvalues should be positive as well.

The output. The end-state $|x\rangle$ that the HHL-circuit is left with after the ancilla-qubit measurement contains the solution vector to the given linear system of equations. However, reading out the state would require a minimum of N measurements, which would eliminate the exponential speed-up of the HHL Algorithm. Instead, one can obtain a 'statistical' information about the solution vector, like the expectation value $\langle x| M |x\rangle$, as suggested by the authors of the HHL Algorithm. [55] shows three other ways to use the solution vector without measuring it; including checking the overlap of any vector with $|x\rangle$, a specific entry of $|x\rangle$ (i.e. $\langle j|x\rangle$ for a given j) and $\langle x| x^n |x\rangle$.

5 Policy Iteration using the HHL Quantum Algorithm

5.1 Introduction

In this chapter, we are going to present how we simulate the HHL Algorithm with Python by implementing the computational steps corresponding to quantum transformations. The simulation is followed by numerical analyses regarding the accuracy of the algorithm with respect to the number of simulation steps T, the desired accuracy deviation ϵ and the condition number κ of the input matrix A. We investigate how κ is affected by the problem dimension N and the reinforcement learning parameter γ. In addition, we address the prerequisites of the HHL Algorithm concerning the data preparation.

5.2 HHL Simulation

The Python simulation[1] aims to evaluate whether the HHL Algorithm, if executed perfectly on a quantum computer, could provide a good enough approximation for the Policy Iteration. As the Policy Iteration Algorithm requires solving a linear system of equations in every iteration, it is important that the cumulation of the expected error ϵ for each solution vector does not lead to wrong policies and consequently, to more iterations.

```
def hhl(A, b, epsilon, T):
    """
    :param A: ndarray
           2d array containing the input matrix
    :param b: ndarray
           1d array containing the right hand side
    :param epsilon: float
    :param T: int
    :return: ndarray
           1d array containing the solution vector
    """
```

Listing 5.1: The method outline corresponding to the HHL Algorithm

[1] The Python code can be found at `https://github.com/akotil/quantum-reinforcement-learning`.

For the simulation, we make use of the Python libraries NumPy [56] and SciPy [57] and import these libraries as `np` and `sp`, respectively.

The HHL Algorithm is mapped to the Python method `hhl(A, b, epsilon, T)`, as shown in Listing 5.1. The method parameters consist of the input matrix A, the right hand side b, the error constant ϵ and time evolution parameter T.

Preparing the inputs. Before running the simulation, it has to be checked whether the input matrix A fulfils the following two conditions: *i*). A is Hermitian. *ii*) A has an eigenvalue range between 0 and 1. The former can be checked with `np.allclose(A.conj().T, A)`, which returns `True` if $A = A^{\dagger}$. If A does not fulfil the Hermitian property, we apply the transformation from (4.8).

```
if not np.allclose(A.conj().T, A):
        A_dagger = np.zeros((n * 2, n * 2), dtype="complex")
        A_dagger[:n, n:] = A
        A_dagger[n:, :n] = A.conj().T
        b = np.pad(b, (0, n), "constant")
```

Listing 5.2: Transforming the input matrix

In order to bring the eigenvalues in the required range, a scaling of A is required, since multiplying A with a positive scalar $\delta < 1$ equates to the same amount of scaling for its eigenvalues:

$$Au = \lambda u \xrightarrow{\delta} (\delta A)u = \delta(Au) = \delta(\lambda u) = (\delta\lambda)u = \delta^{'}u \tag{5.1}$$

With the linear scaling of A, one arrives at the modified LSE $\delta Ax = \delta b$. Since the HHL Algorithm requires b to be a unit vector, the scaling of the solution vector x is applied after the simulation, i.e. we solve $\delta Ax = b$ and scale x thereafter by $\delta||b||$.

Scaling A brings only the *absolute* values of its eigenvalues in a desired range. As discussed in Section 4.3, the eigenvalues come with a negative range after the Hermitian transformation. We deal with the negative eigenvalues by mapping the eigenvalue range to $[-1/2, 1/2]$ and performing an 'eigenvalue flip' in the controlled rotation which will be discussed shortly.

In order to map the eigenvalues to $[-1/2, 1/2]$, we scale A with $\gamma = 1/2\lambda_{max}$.

```
eigenvalues = linalg.eigvals(A)
if np.any(eigenvalues > 0.5):
        max_eigval = max([abs(i) for i in linalg.eigvals(A)])
        A *= 1 / (2 * max_eigval)
```

Listing 5.3: Scaling the input matrix A

Running the simulation. The first step of the simulation is to initialize the system, that is, preparing the two registers containing $|\psi\rangle$ and $|b\rangle$, i.e. the state $|\psi\rangle \otimes |b\rangle$. We can store these registers conveniently as arrays, both in computational basis, and combine them via `np.kron`.

```
psi = [np.sqrt(2 / T) * np.sin(np.pi * (i + 1 / 2) / T) for i in range(T)]
registers = np.kron(psi, b)
```

Listing 5.4: Initialization of the registers

After having initialized the registers, the simulation proceeds to creating the Hamiltonian operator which will be applied to the registers. Due to the special structure of $|\tau\rangle\langle\tau| \otimes e^{iA\tau t_0/T}$ for $\tau \in [T]$, the Hamiltonian operator consists of 'exponential block matrices' on its diagonal. Hence, only these block matrices have to be stored. We therefore create a 3-dimensional array of T $n \times n$ matrices, with n being the dimension of the input matrix. Applying the Hamiltonian operator to the registers corresponds to T matrix-vector multiplications, thus one can treat the registers as a 2-dimensional array to iterate through a block of n-entries.

```
H = np.zeros((T, n, n), dtype="complex")
t_0 = k / epsilon
for i in range(T):
    H[i] = linalg.expm(1j * A * i * t_0 / T)

# Apply Hamiltonian Evolution to the registers
state = [np.dot(H[i], registers.reshape((T,n))[i]) for i in range(T)]
```

Listing 5.5: Hamiltonian Evolution

In order to complete the Phase Estimation, we now fourier transform the first register. We again assume that a quantum computer is able to perform QFT without errors, so that we can use the classical Fourier Transformation to simulate QFT.

After the Hamiltonian Evolution, `state` stores the following:

$$\begin{pmatrix} \left(e^{iA\cdot 0\cdot t_0/T}(|\psi\rangle \otimes |b\rangle)_{j\in\mathbb{N}_0,\, 0\le j<n}\right)^\top \\ \left(e^{iA\cdot 1\cdot t_0/T}(|\psi\rangle \otimes |b\rangle)_{j\in\mathbb{N}_0,\, n\le j<2n}\right)^\top \\ \vdots \\ \left(e^{iA\cdot (T-1)\cdot t_0/T}(|\psi\rangle \otimes |b\rangle)_{j\in\mathbb{N}_0,\, (T-1)n\le j<Tn}\right)^\top \end{pmatrix} = \begin{pmatrix} (|\psi\rangle_0\, e^{iA\cdot 0\cdot t_0/T}\,|b\rangle)^\top \\ (|\psi\rangle_1\, e^{iA\cdot 1\cdot t_0/T}\,|b\rangle)^\top \\ \vdots \\ (|\psi\rangle_{T-1}\, e^{iA\cdot (T-1)\cdot t_0/T}\,|b\rangle)^\top \end{pmatrix} \tag{5.2}$$

The Fourier Transformation is applied on every column of `state`, which can be achieved by specifying `axis=0` in `np.fft.fft()`. Furthermore, the parameter `norm` specifies the scaling within the transformation. By setting `norm="ortho"`, the scale factor is set to $1/\sqrt{n}$

instead of the default scaling $1/n$ of `fft`. As a consequence, the Fourier transformation becomes unitary, i.e. it is symmetric with respect to its inverse, as shown in (3.11) and (3.14).

```
state = np.fft.fft(state, axis=0, norm="ortho")
```

Listing 5.6: Fourier Transformation

After the Fourier Transformation, the controlled rotation is applied on the ancilla register. For this step, we consider the mathematical representation of the ancilla register after the rotation: $\sqrt{1 - \frac{C^2}{\widetilde{\lambda}_k}}\,|0\rangle + \frac{C}{\widetilde{\lambda}_k}\,|1\rangle$. It is not necessary to numerically add the ancilla register, one can just as well rescale `state[k]` by $\frac{C}{\widetilde{\lambda}_k}$ to extract the relevant state for measurement. One can also compute 'the other half' which carries $|0\rangle$, however, only the state half with $|1\rangle$ delivers the solution vector when measured. In reality, one may have to perform several measurements until the state is left with the solution vector, corresponding to measuring 1. We can disregard the measurement in the simulation, since we can directly extract the solution.

In order to deal with negative eigenvalues, we scale the eigenvalue range to $[-1/2, 1/2]$, following a similar approach in [58, 59]. We mirror the the eigenvalues by changing their signs from $T/2$. The specified range is necessary, since the negative numbers in range $[-1/2, 0)$ cannot be distinguished from their positive equivalents with `mod 1`.

```
C = 0.1 / k
one_state = np.zeros((T, n), dtype="complex")
for i in range(T):
    if not scaled or (scaled and i < T // 2):
        eigenvalue = 2 * np.pi * i / t_0
    else:
        eigenvalue = 2 * np.pi * (i - T) / t_0

    C_1 = C / eigenvalue if C <= abs(eigenvalue) else 0
    one_state[i] = C_1 * state[i]
```

Listing 5.7: Conditional rotation

The last stages of the simulation match the steps of reversing the Phase Estimation. These stages include the inverse Fourier Transformation, the inverse Hamiltonian Evolution and undoing the computation of $|\psi\rangle$. The inverse Hamiltonian Evolution corresponds to the conjugate transpose of `H[i]`, as the Hamiltonian operator is unitary and therefore satisfy `np.dot(H[i].conj().T, H) == np.eye(n)`.

```
# Inverse Fourier Transformation
one_state = np.fft.ifft(one_state, axis=0, norm="ortho")

# Inverse Hamiltonian Evolution
one_state = [np.dot(H[i].conj().T, one_state[i]) for i in range(T)]
```

Listing 5.8: Reversing the Phase Estimation

Undoing the computation of $|\psi\rangle$, i.e. reversing the control register back to $|0\rangle$ would correspond to a 'reverse' Kronecker Product. We can consider the `one_state` after the inverse Hamiltonian Evolution to be $|\psi\rangle \otimes |x\rangle$. This representation gives us the ability to simply reverse $\otimes$ by choosing an $i \in \{1, \ldots, T\}$ such that

$$\begin{aligned} |\psi\rangle \otimes |x\rangle &= (\psi_1 x_1,\ \psi_1 x_2,\ \ldots,\ \psi_1 x_n,\ \psi_2 x_1, \ldots,\ \psi_T x_n)^\top \\ x_j &= \frac{1}{\psi_i}(|\psi\rangle \otimes |x\rangle)_{i \times j} \quad \forall j \in \{1, \ldots, n\} \end{aligned} \tag{5.3}$$

We observed that the choice of the index i affects the accuracy of the result, so we randomly initialized linear systems of equations to extract x_1 for all values of i.

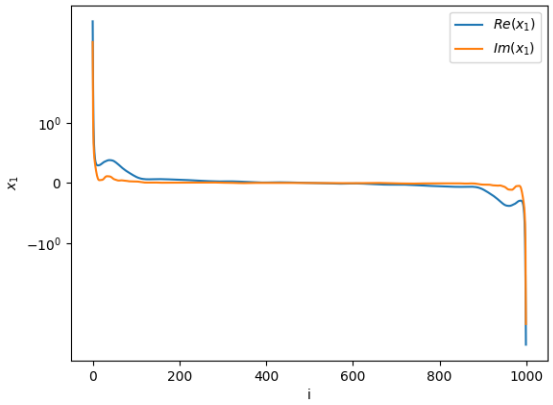


Figure 5.1: The values of x_1 in dependence of the selected index i for obtaining the x vector from $|\psi\rangle \otimes |x\rangle$ under a randomly initialized system with $T = 1000$.

The empirical results revealed that the index interval around $T/2$ result in the most stable result for x_1. Hence, we choose $i = T/2$ for extracting the solution vector.

```
one_state /= C # undo scaling by rotation
solution = one_state[T // 2] / psi[T // 2] # undo Kronecker Product
solution *= SCALE_FACTOR # undo scaling by δ
solution *= B_NORM # undo scaling by ||b||
```

Listing 5.9: Reversing the control register

The `solution` array now stores the desired solution vector to the LSE $Ax = b$ up to the desired precision given by ϵ.

5.3 Parameter Analysis

For this and upcoming sections, we are going to focus on a concrete reinforcement learning (RL) application. Our goal is to make a solid statement about whether or not the HHL Algorithm can deliver good approximations for the Policy Evaluation (PE) step of the Policy Iteration Algorithm (PI), and how the cumulated errors affect the accuracy performance of policy iterations.

The concrete RL application[2] is a modified version of the example introduced in Chapter 2, Section 2.4.2. We expand the maze dimension to $n = 10$ and add more blocked states and one exit state. Figure 5.2 shows the grid environment, where `[(3,2), (7,3), (4,5), (9,5), (7,6), (1,7), (3,9), (5,9)]` are blocked fields, `[(1,4), (8,5)]` are exit fields with a reward of `+1` and `(1,5)` is an exit field with a reward of `-1`. Under this set-up and with $\gamma = 0.95$, the classical PI converges to the state-action mapping as depicted in the rightmost grid of Figure 5.2.

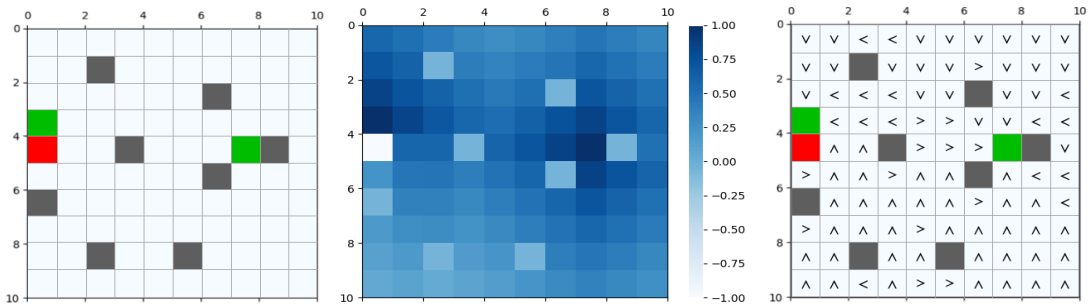


Figure 5.2: The setting of the RL application which is to be investigated (*left*), converged state values (*middle*) and converged state-action mapping (*right*).

In the following, we reproduce the results of classical PI by replacing the PE-step with the simulation of HHL Algorithm. In order to decide on a number of time evolution steps T, we first investigate the γ-κ and then the T-accuracy relationship given a matrix with a specific κ-value. The accuracy measure in the latter case is based on the residue $||Ax - b||$ with x being the output vector of the HHL Algorithm with arbitrary test inputs A and b.

[2]The maze construction is located at `module/maze.py` and a simulation of RL algorithms can be found at `module/reinforcement_learning.py` under `https://github.com/akotil/quantum-reinforcement-learning`.

γ - Condition number κ The RL parameter γ is an important factor which directly affects the condition number of the matrix of one single PE. As the condition number grows, the accuracy of the HHL Algorithm suffers. Therefore we take a close look at the $\gamma - \kappa$ relationship so that a statement can be made as to how the system matrix will behave under a certain γ value.

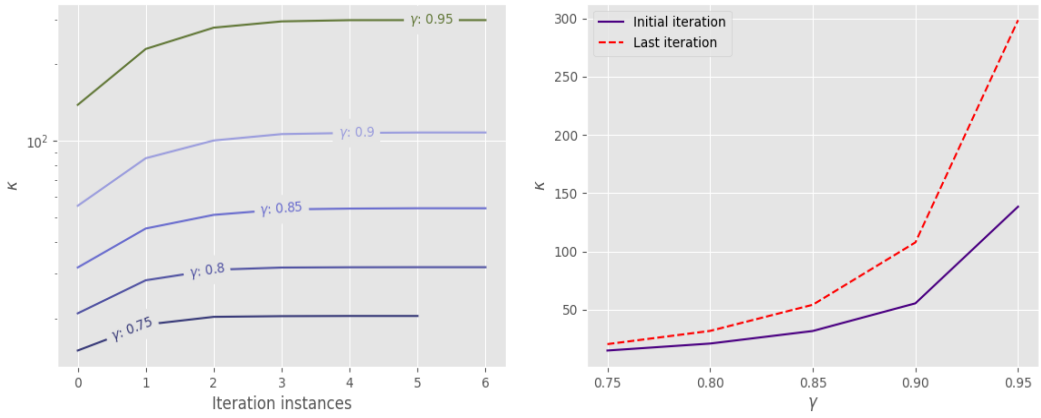


Figure 5.3: Plots showing how the condition number of the PE matrix differs by iterations given γ-value range

Figure 5.3 presents the dependence of the condition number of the matrix resulting from one PE on different γ values until convergence. For the plots, we have used the RL application introduced before. For consistency, we fixed a random policy, which was used to initialize the PI with. There are two important relations which can be extracted from the plots: One relation is the condition number increase between iterations for a given γ, depicted as a line at the left graph. The condition numbers seem to grow logarithmically as PI proceeds with further iterations. The other relation is the condition number increase (plotted with logarithmic scale) of one matrix belonging to a certain iteration (first or last) as γ is increased, depicted as points at the right graph. The condition number growth is exponential in the given γ set of `[0.75, 0.8, 0.85, 0.9, 0.99]`.

The results from γ-κ analysis show that the γ-parameter directly plays a role in the condition number of a PE matrix. As the classical results from the RL application are to be reproduced with the help of the HHL Algorithm, we note the condition number range for the γ-value `0.95`, which is $(138, 299)$.

T - Accuracy deviation ϵ The goal of the $T - \epsilon$ analysis is to find out the following: Given an input matrix with condition number κ, at which T value the HHL Algorithm is able to produce a solution vector which differs from the actual solution vector x^* by ϵ under the l^2-norm.

One can produce arbitrary matrices with a specific κ value. A possible approach for achieving this would be to produce a random matrix $A \in \mathbb{R}^{n \times n}$, extract its singular value decomposition UDV and replace the entries $(\sigma_1, \dots \sigma_n)$ of the diagonal matrix D with a uniformly spaced interval between 1 and κ. That way the ratio between the biggest and smallest singular values is guaranteed to be κ for the modified matrix $A^{'} = UD^{'}V$ with $D^{'} = \text{diag}(1, \dots, \kappa)$.

```
A = np.random.random((n, n))
u, s, v = sp.linalg.svd(A)
s = np.linspace(1, condition_number, endpoint=True, num=n)
s = np.diag(s)
A = u @ s @ v
```

Listing 5.10: Producing a random matrix with desired condition number κ

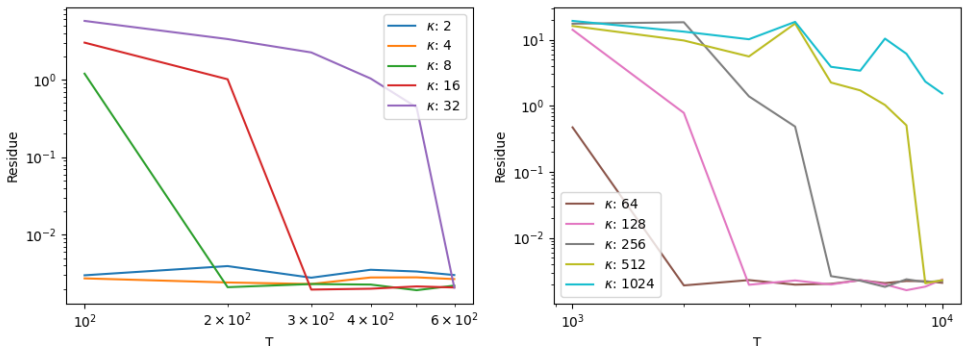


Figure 5.4: The residue $||A\widetilde{x} - b||$ in dependence of the parameter T

Figure 5.4 depicts the T - ϵ relationship for matrices with dimensions 10×10. The graphs show the residue $||A\widetilde{x} - b||$ depending on the condition numbers of the test matrices A. As the condition numbers grow, the HHL Algorithm needs a larger T-parameter to produce a solution vector with the desired accuracy derivation of $\epsilon = 0.01$. The HHL Algorithm succeeds in producing the right approximation for x at most by $T = 600$ for a $\kappa \in \{2, 4, 8, 16, 32\}$. For very small condition numbers, $T = 100$ already satisfies the accuracy deviation, as seen with $\kappa \in \{2, 4\}$. As for the other half of condition numbers, $\kappa \in \{64, 128, 256, 512, 1024\}$, a larger scaling for T is needed. Looking back at the condition number range of our RL application, a proper T value falls in the range $[4000, 5000]$ as the residues in the right graph converge to ϵ within this range for $\kappa \cong 256$.

Dimension n - Condition number κ One of the important constraints of the HHL Algorithm is the necessity of a polylogarithmic scaling of κ as the matrix dimension n grows, otherwise the quantum algorithm cannot outperform the best classical solver for

LSE. Considering the setting of our RL application, it is not clear how the application scales with the growing number of states, as there is no unique way of expanding a maze. One can expand the maze structure in numerous ways; by adding more fields and fixing the number of blocked states and keeping the exit states where they are, or by adding more blocked states and exit states as the dimension grows. Both positions and the number of such special states make up the characteristics of the matrix, and implicitly its condition number. It is therefore not trivial to find out about the relationship between the condition number of the PE matrix and its dimension. To illustrate this, Figure 5.5 shows two different scaling strategies: The left graph shows a maze growth with two fixed exit fields and three fixed blocked states. The right graph illustrates another maze growth with randomly placed and proportionally many blocked states with respect to the maze size.

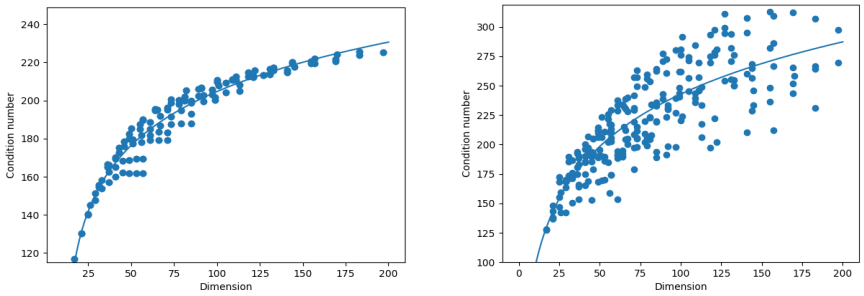


Figure 5.5: The growth of condition number as the maze dimension scales, resulting in different distributions with respect to different scaling strategies (*left*: fixed, *right*: randomly)

A detailed analysis is needed for cases where the scaling of RL applications is well-defined, in order to make a concrete statement about n-κ-dependency.

Results. Running the HHL Algorithm with $\epsilon = 0.01$ and $T = 5000$ for the example RL application yields the results in the left graph of Figure 5.6. The approximate solution vectors resulting from the simulation differ about a magnitude of 10^{-3} from the real solution in every iteration of the PI. These approximations indeed lead to the same policies in every iteration; resulting in the same converged solution from the classical version showed in Figure 5.2.

On the other hand, choosing $T = 4000$ fails to converge to the right policy. The right hand side of the Figure 5.6 shows the norm differences for the case $T = 4000$, where the HHL Algorithm fails to be a good approximation for the PI. For the first 3 iterations, the quantum algorithm results in rather small deviations from the real solution vector, however, after the third iteration, the resulting PE matrix has a higher condition number

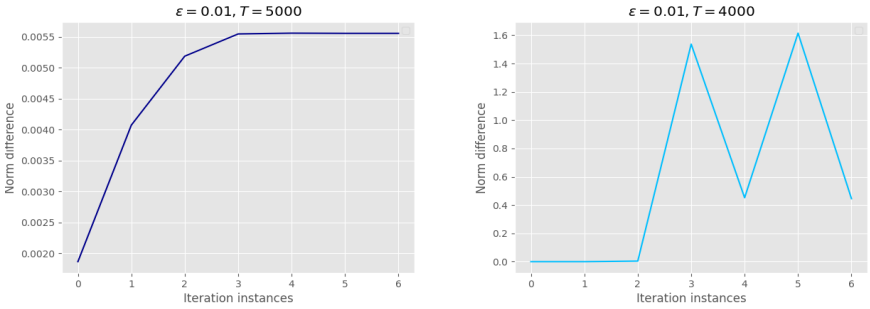


Figure 5.6: The absolute norm differences $\big|\|x\| - \|\widetilde{x}\|\big|$ between the classical solution vector x and the quantum solution vector $\widetilde{x}$ per iteration (*left*: $T = 5000$, *right*: $T = 4000$)

($\kappa \cong 279$) than the parameter $T = 4000$ can compensate for. As a result, the policy vector differs from the corresponding policy vector in the classical case, and the norm difference gets bigger. After that, the HHL Algorithm proceeds to solve for the next iterations to only fail every second iteration, because the PE matrix has almost the same large condition number from before as the policy vector is near convergence. Consequently, the quantum solutions oscillate between two policies, one of which is the nearly converged one, with the other policy being the result from a poor approximation.

Efficient Hamiltonian Simulation The HHL Algorithm assumes that the inputs are efficiently computable, meaning that the Hamiltonian operator responsible for simulating the matrix A can be efficiently prepared and applied in a quantum computer. This is crucial in our application, as the inputs change with every iteration of PI; a quantum computer needs to prepare the inputs not slower than a time complexity which grows polylogarithmic in n.

Hamiltonian simulations are in general non-trivial, complex operations. However, a particular matrix form is shown to be efficiently simulated. If the matrix is s-sparse and efficiently row-computable, then the Hamiltonian simulation can be applied in a time period which grows almost linearly within s [60, 61]. A matrix A is called s-sparse if it has at most s non-zero values in each row. In addition, the matrix A is efficiently row-computable if there is an efficient quantum or classical algorithm which can output all non-zero entries $(j, A_{i,j})$ for a given row index i, often treated as a 'black box' operation [62].

We break down the PE matrix $A = (\mathbb{I} - \gamma\mathcal{P})$ with $\mathcal{P} = (p_{i,j}) = \mathbb{P}(s_j|s_i, \pi(s_i))$ to find out about its sparsity. Each row of $\mathcal{P}$ represents the transition probabilities to every other state from the state which the row corresponds to. In our setting, the agent can only choose to move to neighbouring states. Since the environment is chessboard-shaped,

there are only a maximum of 4 fields that the agent can move to. These states result in a non-zero transition probability; for every other state, the row is filled with zeros. Scaling $\mathcal{P}$ with γ does not change its sparsity, but substracting it from the identity matrix results in a maximum of 5 non-zero entries in each row. It is important to note that the sparsity of A is indeed constant for any dimensions of the grid environment, given that the setting of the RL problem is not changed. Furthermore, if the matrix is not Hermitian, turning it into a Hermitian matrix will result in a sparsity of $s = 5$ in a dimension of $2n$.

As for the row-computability, we consider the transition probabilities introduced in Chapter 2, Section 2.4.1. For every row i, one only needs to calculate the probability of a transition to the neighbouring states. This computation needs to be done only once, since the transition probabilities are pre-determined and remain the same throughout the application. Thus, a classical computer can output $(j, A_{i,j})$ in at most $\mathcal{O}(s)$ with the overhead of preparing the transition probability tensor at the beginning.

6 Conclusion

Summary. In this study, we introduced a way of simulating the Harrow-Hassidim-Lloyd Quantum Algorithm (HHL Algorithm) in Python and integrated the simulation in the Policy Evaluation step of the Policy Iteration Algorithm under a concrete model-based reinforcement learning application. We investigated the results, where we showed that a proper parameter initialization of the HHL Algorithm is necessary in order to achieve a satisfying approximation for the policy iterations. We also provided our estimates for proper parameters in our case; which may serve as a guide for other applications, although suitable parameter will vary with each use case. We addressed most of the prerequisites of the HHL Algorithm and presented ways of overcoming them.

Future Work. An important caveat of the HHL Algorithm is the solution read-out, which we have not addressed and which may be a starting point of a future work investigating a similar Reinforcement Learning application in context of the HHL Algorithm. More in-depth mathematical analyses about the Policy Iteration Algorithm are needed to tackle the read-out problem. One could look at possible patterns of state values to derive an approximate simplification, so that the whole solution vector does not have to be read out. Instead, a subset of the vector's entries could be sufficient to reach convergence. One could also look into possible and practical ways of using the statistical data provided by the solution vector, as it is the only way of making use of the solution without losing the quantum-advantage of the HHL Algorithm.

List of Figures

Bibliography

[1] A. Einstein, B. Podolsky, and N. Rosen. "Can Quantum-Mechanical Description of Physical Reality Be Considered Complete?" In: *Phys. Rev.* 47 (10 May 1935), pp. 777–780. DOI: 10.1103/PhysRev.47.777.

[2] A. Einstein, Hedwig Born, and M. Born. *Briefwechsel 1916-1955*. rororo Taschenbücher. Rowohlt, 1972, p. 254. ISBN: 9783499114786.

[3] J. S. Bell. "On the Einstein Podolsky Rosen paradox." In: *Physics Physique Fizika* 1 (3 Nov. 1964), pp. 195–200. DOI: 10.1103/PhysicsPhysiqueFizika.1.195.

[4] W. Heisenberg. "Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik." In: *Zeitschrift für Physik* 43.3 (1927), pp. 172–198. DOI: 10.1007/BF01397280.

[5] Richard P. Feynman. "Simulating physics with computers." In: *International Journal of Theoretical Physics* 21.6 (1982), pp. 467–488. DOI: 10.1007/BF02650179.

[6] David Deutsch and Richard Jozsa. "Rapid solution of problems by quantum computation." In: (Dec. 1992). DOI: https://doi.org/10.1098/rspa.1992.0167.

[7] Ethan Bernstein and Umesh Vazirani. "Quantum Complexity Theory." In: *SIAM Journal on Computing* 26.5 (1997), pp. 1411–1473. DOI: 10.1137/S0097539796300921.

[8] Scott Aaronson and Andris Ambainis. "The Need for Structure in Quantum Speedups." In: *Theory of Computing* 10.6 (2014), pp. 133–166. DOI: 10.4086/toc.2014.v010a006.

[9] Shalev Ben-David et al. *Symmetries, graph properties, and quantum speedups*. 2020. arXiv: 2006.12760 [quant-ph].

[10] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search." In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 212–219. ISBN: 0897917855. DOI: 10.1145/237814.237866.

[11] Frédéric Magniez et al. "Search via Quantum Walk." In: *SIAM Journal on Computing* 40.1 (2011), pp. 142–164. DOI: 10.1137/090745854.

[12] P. W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring." In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[13] Alberto Peruzzo et al. "A variational eigenvalue solver on a photonic quantum processor." In: *Nature Communications* 5.1 (2014), p. 4213. DOI: 10.1038/ncomms5213.

[14] Daniel Gottesman. *An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation*. 2009. arXiv: 0904.2557 [quant-ph].

[15] Ryan Babbush et al. *Focus beyond quadratic speedups for error-corrected quantum advantage*. 2020. arXiv: 2011.04149 [quant-ph].

[16] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. "Quantum Algorithm for Linear Systems of Equations." In: *Phys. Rev. Lett.* 103 (15 Oct. 2009), p. 150502. DOI: 10.1103/PhysRevLett.103.150502.

[17] Vedran Dunjko, Jacob M. Taylor, and Hans J. Briegel. *Framework for learning agents in quantum environments*. 2015. arXiv: 1507.08482 [quant-ph].

[18] D. Dong et al. "Quantum Reinforcement Learning." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 38.5 (Oct. 2008), pp. 1207–1220. ISSN: 1941-0492. DOI: 10.1109/TSMCB.2008.925743.

[19] Lucas Lamata. "Basic protocols in quantum reinforcement learning with superconducting circuits." In: *Scientific Reports* 7.1 (2017), p. 1609. DOI: 10.1038/s41598-017-01711-6.

[20] Samuel Yen-Chi Chen et al. *Variational Quantum Circuits for Deep Reinforcement Learning*. 2020. arXiv: 1907.00397 [cs.LG].

[21] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[22] Art Lew and Holger Mauch. *Dynamic Programming: A Computational Tool (Studies in Computational Intelligence (38))*. Springer, 2006.

[23] Ehrhard Behrends. *Introduction to Markov Chains With Special Emphasis on Rapid Mixing*. Vieweg+Teubner Verlag, 1999. ISBN: 978-3-528-06986-5.

[24] Richard S. Sutton and Andrew G. Barto. "Reinforcement learning: an introduction." In: A Bradford Book, 2018.

[25] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, 2005.

[26] Mark Lee. *The Agent-Environment Interface*. [Online; accessed November 26, 2020]. 2005. URL: http://incompleteideas.net/book/first/ebook/node28.html.

[27] Tjalling C. Koopmans. "Stationary Ordinal Utility and Impatience." In: *Econometrica* 28.2 (1960), pp. 287–309. ISSN: 00129682, 14680262.

[28] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. 2002.

[29] Eric V Denardo. "Contraction mappings in the theory underlying dynamic programming." In: *Siam Review* 9.2 (1967), pp. 165–177.

[30] H.S. Chang et al. "Simulation-based Algorithms for Markov Decision Processes." In: Communications and Control Engineering. Springer London, 2007, p. 7. ISBN: 9781846286896.

[31] Manuel S Santos and John Rust. "Convergence properties of policy iteration." In: *SIAM Journal on Control and Optimization* 42.6 (2004), pp. 2094–2115.

[32] Lucian Busoniu et al. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press, 2010, p. 113.

[33] Max Planck. "Über die Elementarquanta der Materie und der Elektrizität." In: *Von Kirchhoff bis Planck*. Springer, 1978, pp. 191–194.

[34] Benjamin Schumacher. "Quantum coding." In: *Phys. Rev. A* 51 (4 Apr. 1995), pp. 2738–2747. DOI: 10.1103/PhysRevA.51.2738.

[35] Sergio Boixo et al. "Characterizing quantum supremacy in near-term devices." In: *Nature Physics* 14.6 (2018), pp. 595–600. DOI: 10.1038/s41567-018-0124-x.

[36] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor." In: *Nature* 574.7779 (2019), pp. 505–510. DOI: 10.1038/s41586-019-1666-5.

[37] Han-Sen Zhong et al. "Quantum computational advantage using photons." In: *Science* (2020). ISSN: 0036-8075. DOI: 10.1126/science.abe8770.

[38] P. A. M. Dirac. "A new notation for quantum mechanics." In: *Mathematical Proceedings of the Cambridge Philosophical Society* 35.3 (1939), pp. 416–418. DOI: 10.1017/S0305004100021162.

[39] Sadri Hassani. *Mathematical Physics*. Springer International Publishing, 2013.

[40] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2012.

[41] Matthew F. Pusey, Jonathan Barrett, and Terry Rudolph. "On the reality of the quantum state." In: *Nature Physics* 8.6 (2012), pp. 475–478. DOI: 10.1038/nphys2309.

[42] R.P. Feynman, R.B. Leighton, and M. Sands. *The Feynman Lectures on Physics, Vol. III: The New Millennium Edition: Quantum Mechanics*. The Feynman Lectures on Physics. Basic Books, 2011. Chap. 10. ISBN: 9780465025015.

[43] Ian Glendinning. "The bloch sphere." In: *QIA Meeting TechGate*. 2005.

[44] CC BY-SA 3.0 Smite-Meister. *Bloch sphere, a geometrical representation of a two-level quantum system.* [Online; accessed December 12, 2020]. 2009. URL: https://commons.wikimedia.org/wiki/File:Bloch_sphere.svg.

[45] *Summary of Quantum Operations*. https://qiskit.org/documentation/tutorials/circuits/3_summary_of_quantum_operations.html. Accessed: 2021-01-14.

[46] D. Sundararajan. *The Discrete Fourier Transform: Theory, Algorithms and Applications*. World Scientific, 2001. ISBN: 9789812810298.

[47] E. Schrödinger. "An Undulatory Theory of the Mechanics of Atoms and Molecules." In: *Phys. Rev.* 28 (6 Dec. 1926), pp. 1049–1070. DOI: 10.1103/PhysRev.28.1049. URL: https://link.aps.org/doi/10.1103/PhysRev.28.1049.

[48] Seth Lloyd. "Universal Quantum Simulators." In: *Science* 273.5278 (1996), pp. 1073–1078. ISSN: 0036-8075. DOI: 10.1126/science.273.5278.1073.

[49] Stuart Hadfield and Anargyros Papageorgiou. "Divide and conquer approach to quantum Hamiltonian simulation." In: *New Journal of Physics* 20.4 (Apr. 2018), p. 043003. DOI: 10.1088/1367-2630/aab1ef.

[50] Jonathan R Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. USA, 1994.

[51] Bojia Duan et al. "A survey on HHL algorithm: From theory to application in quantum machine learning." In: *Physics Letters A* 384.24 (2020), p. 126595. ISSN: 0375-9601.

[52] Danial Dervovic et al. *Quantum linear systems algorithms: a primer*. 2018. arXiv: 1802.08227 [quant-ph].

[53] Scott Aaronson, Daniel Grier, and Luke Schaeffer. "The Classification of Reversible Bit Operations." In: *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*. Ed. by Christos H. Papadimitriou. Vol. 67. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 23:1–23:34. ISBN: 978-3-95977-029-3. DOI: 10.4230/LIPIcs.ITCS.2017.23.

[54] Arun Kumar Pati and Samuel L. Braunstein. In: *Nature* 404.6774 (Mar. 2000), pp. 164–165. DOI: 10.1038/35004532.

[55] B. D. Clader, B. C. Jacobs, and C. R. Sprouse. "Preconditioned Quantum Linear System Algorithm." In: *Phys. Rev. Lett.* 110 (25 June 2013), p. 250504. DOI: 10.1103/PhysRevLett.110.250504.

[56] Charles R. Harris et al. "Array programming with NumPy." In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[57] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[58] Lidia Ruiz-Perez and Juan Carlos Garcia-Escartin. "Quantum arithmetic with the quantum Fourier transform." In: *Quantum Information Processing* 16.6 (2017), p. 152. DOI: 10.1007/s11128-017-1603-1.

[59] Changpeng Shao. *Reconsider HHL algorithm and its related quantum machine learning algorithms*. 2018. arXiv: 1803.01486 [quant-ph].

[60] Dominic W. Berry et al. "Exponential Improvement in Precision for Simulating Sparse Hamiltonians." In: *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*. STOC '14. New York, New York: Association for Computing Machinery, 2014, pp. 283–292. ISBN: 9781450327107. DOI: 10.1145/2591796.2591854.

[61] Dominic W. Berry et al. "Simulating Hamiltonian Dynamics with a Truncated Taylor Series." In: *Phys. Rev. Lett.* 114 (9 Mar. 2015), p. 090502. DOI: 10.1103/PhysRevLett.114.090502.

[62] Dorit Aharonov and Amnon Ta-Shma. "Adiabatic Quantum State Generation and Statistical Zero Knowledge." In: *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*. STOC '03. San Diego, CA, USA: Association for Computing Machinery, 2003, pp. 20–29. ISBN: 1581136749. DOI: 10.1145/780542.780546.