Sun Algorithm Implementation for Quantum State Preparation with Ancillary Qubits

High Performance Computing Course Project 2024-25

Student: Andrea Bersellini

Tutor: Giacomo Belli

Professor: Michele Amoretti

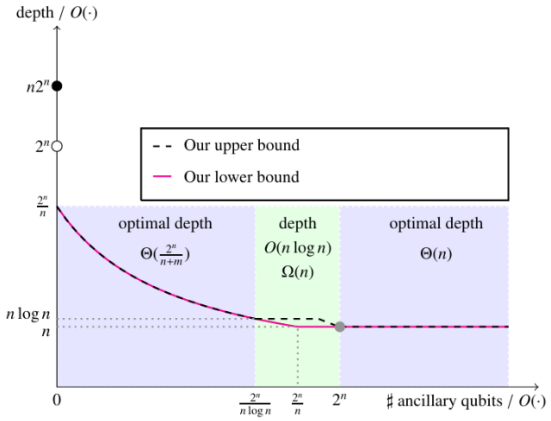
This project benefits from the High Performance Computing facility of the University of Parma, Italy (HPC.unipr.it)

1 Introduction

This report aims to present and discuss a Python implementation of Sun's algorithm [1] with $O(\frac{2^n}{n \log n})$ ancillary qubits for the *Quantum State Preparation (QSP)* problem, using the PennyLane library.

The goal of the QSP problem is to prepare a desired quantum state $|\psi_v\rangle = \sum_{k=0}^{N^2-1} v_k |k\rangle$ starting from an n-qubit quantum circuit with the initial state $|0\rangle^{\otimes n}$ and a vector $v = (v_0, v_1, v_2, ..., v_{2^n-1})^T \in C^{2^n}$ such that $\sqrt{\sum_{k=0}^{N^2-1} |v_k|^2} = 1$.

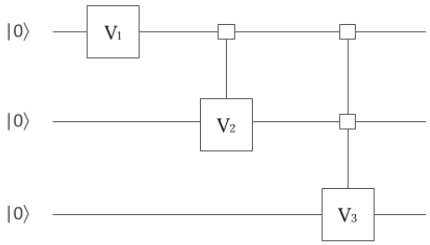
Since the problem is generally approached without the use of ancillary qubits, introducing a different number of them can lead to different upper bounds on circuit depth. This implementation focuses on the first range, where $m \in [2n, O(\frac{2^n}{n \log n})]$ qubits are used.



The proposed algorithm claims to achieve a solution to the QSP problem with the best trade-off between the number of ancillary qubits used and the depth of the implemented circuit.

1.1 State-of-the-Art Method

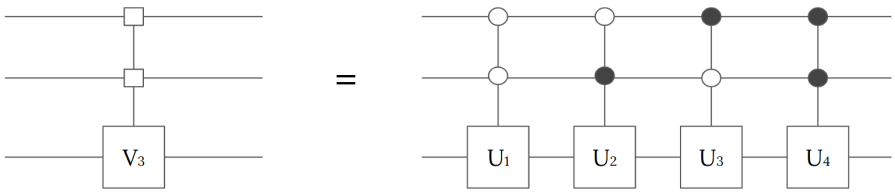
As presented in [2], it is possible to prepare an n-qubit quantum state by implementing a circuit V_j on each qubit, where the circuit applies a unitary operation conditioned on the previous $j-1$ qubits, as shown below.



Since V_j is a block-diagonal matrix with diagonal elements $U_k \in C^{2\times 2}$, its matrix form is given by

$$V_j = \text{diag}(U_1, U_2, \ldots, U_{2^{j-1}}).$$

The most common method for implementing V_j is by using a circuit composed of 2^{j-1} controlled gates, which operate on the current qubit and are conditioned on all possible computational basis states of the $j-1$ qubits.



By implementing V_j using this methodology, the general QSP problem can be solved with a circuit of depth $\Theta(n2^n)$. However, this approach is not optimal when compared to the bound of $\Theta(2^n/n)$ shown in (1.1).

1.2 Sun's Algorithm

The proposed algorithm aims to solve the QSP problem by decomposing each *uniformly controlled gate* (UCG) V_j into the following form

$$U_k = e^{i\alpha} \cdot R_z(\beta_k) \cdot S \cdot H \cdot R_z(\gamma_k) \cdot H \cdot S^\dagger \cdot R_z(\delta_k),$$

as demonstrated in (2.2).

The core of the algorithm relies on the efficient implementation of a diagonal operator

$$\Lambda_n = \text{diag}(1, e^{i\theta_1}, e^{i\theta_2}, \ldots, e^{i\theta_{2^n-1}}) \in C^{2^n \times 2^n},$$

which enables the implementation of the circuit V_j in the following form, as shown in (2.2):

$$V_n = \text{diag}(e^{i\alpha_1}, \ldots, e^{i\alpha_{2^{n-1}}}) \otimes I_1 \cdot \text{diag}(R_z(\beta_1), \ldots, R_z(\beta_{2^{n-1}})),$$

$$\cdot I_{n-1} \otimes (SH) \cdot \text{diag}(R_z(\gamma_1), \ldots, R_z(\gamma_{2^{n-1}})) \cdot I_{n-1} \otimes (HS^\dagger) \cdot \text{diag}(R_z(\delta_1), \ldots, R_z(\delta_{2^{n-1}})).$$

By implementing Λ_n using a circuit with depth $D(n)$ and width $S(n)$, it is consequently possible to prepare the n-qubit quantum state with a circuit of depth

$$3\sum_{k=1}^{n}(3D(k)+2)+1,$$

and width

$$3\sum_{k=1}^{n}(3S(k)+2)+1.$$

As shown, V_i can be derived through a multiplexer of controlled gates $(U_1, \ldots, U_{2^{i-1}})$, which can be decomposed into the form

$$U_k = e^{i\alpha_k} R_z(\beta_k) SH R_z(\gamma_k) HS^\dagger R_z(\delta_k).$$

Since the author does not consider the calculation of the phase $e^{i\alpha_k}$, as it is regarded as a well-established process and does not affect the measurement of the quantum system, we will focus solely on developing the rotations and unitary gates.

Since this decomposition is valid for any unitary, and there are no constraints on the choice of U_k, it becomes possible, by appropriately adjusting the angles, to adopt an R_y unitary matrix in the form

$$SHR_z(\gamma_k)HS^\dagger.$$

By proceeding in this way, the entire UCG V_i simplifies to

$$V_i = I_{i-1} \otimes (SH) \cdot \text{diag}(R_z(\gamma_1), \ldots, R_z(\gamma_{2^{i-1}})) \cdot I_{i-1} \otimes (HS^\dagger),$$

which corresponds to only the R_y rotation component, significantly reducing both dimensional and computational complexity of the circuit and facilitating the generation of the parameters that we will see in Section 3.

2 Preliminaries and Notations

This section provides a summary of the notations and theoretical considerations that have facilitated the understanding, application, and implementation of the algorithm.

2.1 $R_y(\gamma)$ Decomposition

Starting from the definition of the following single-qubit gates, it can be shown that $R_y(\gamma)$ can be obtained from the composition of $SHR_z(\gamma)HS^\dagger$, for every $\gamma \in R$.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \quad S^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}, \quad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad R_z(\gamma) = \begin{bmatrix} e^{-i\frac{\gamma}{2}} & 0 \\ 0 & e^{i\frac{\gamma}{2}} \end{bmatrix}$$

$$R_y(\gamma) = \begin{bmatrix} \cos\frac{\gamma}{2} & -\sin\frac{\gamma}{2} \\ \sin\frac{\gamma}{2} & \cos\frac{\gamma}{2} \end{bmatrix} = SHR_z(\gamma)HS^\dagger$$

$$= \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdots$$

$$= \frac{1}{2} \begin{bmatrix} 1 & 1 \\ i & -i \end{bmatrix} \begin{bmatrix} e^{-i\frac{\gamma}{2}} & 0 \\ 0 & e^{i\frac{\gamma}{2}} \end{bmatrix} \cdots$$

$$= \frac{1}{2} \begin{bmatrix} e^{-i\frac{\gamma}{2}} & e^{i\frac{\gamma}{2}} \\ ie^{-i\frac{\gamma}{2}} & -ie^{i\frac{\gamma}{2}} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdots$$

$$= \frac{1}{2} \begin{bmatrix} e^{-i\frac{\gamma}{2}} + e^{i\frac{\gamma}{2}} & e^{-i\frac{\gamma}{2}} - e^{i\frac{\gamma}{2}} \\ ie^{-i\frac{\gamma}{2}} - ie^{i\frac{\gamma}{2}} & ie^{-i\frac{\gamma}{2}} + ie^{i\frac{\gamma}{2}} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}$$

$$= \frac{1}{2} \begin{bmatrix} e^{-i\frac{\gamma}{2}} + e^{i\frac{\gamma}{2}} & -ie^{-i\frac{\gamma}{2}} + ie^{i\frac{\gamma}{2}} \\ ie^{-i\frac{\gamma}{2}} - ie^{i\frac{\gamma}{2}} & e^{-i\frac{\gamma}{2}} + e^{i\frac{\gamma}{2}} \end{bmatrix}$$

$$= \frac{1}{2} \begin{bmatrix} \cos\frac{\gamma}{2} - i\sin\frac{\gamma}{2} + \cos\frac{\gamma}{2} + i\sin\frac{\gamma}{2} & -i\cos\frac{\gamma}{2} - \sin\frac{\gamma}{2} + i\cos\frac{\gamma}{2} - \sin\frac{\gamma}{2} \\ i\cos\frac{\gamma}{2} + \sin\frac{\gamma}{2} - i\cos\frac{\gamma}{2} + \sin\frac{\gamma}{2} & \cos\frac{\gamma}{2} - i\sin\frac{\gamma}{2} + \cos\frac{\gamma}{2} + i\sin\frac{\gamma}{2} \end{bmatrix}$$

$$= \frac{1}{2} \begin{bmatrix} 2\cos\frac{\gamma}{2} & -2\sin\frac{\gamma}{2} \\ 2\sin\frac{\gamma}{2} & 2\cos\frac{\gamma}{2} \end{bmatrix}$$

$$= \begin{bmatrix} \cos\frac{\gamma}{2} & -\sin\frac{\gamma}{2} \\ \sin\frac{\gamma}{2} & \cos\frac{\gamma}{2} \end{bmatrix}$$

$$= R_y(\gamma)$$

2.2 Single-Qubit Gates Decomposition

From the decomposition property of a generic unitary matrix $U = e^{i\alpha} \cdot R_z(\beta) \cdot R_y(\gamma) \cdot R_z(\delta)$, and using the result demonstrated in (2.1), it can be deduced that for any single-qubit operation U, there exist $\alpha, \beta, \gamma, \delta \in R$ such that:

$$U = e^{i\alpha} R_z(\beta) SHR_z(\gamma) HS^\dagger R_z(\delta)$$

Since $V_n = \text{diag}(U_1, U_2, \ldots, U_{2^{n-1}}) \in R^{2^n \times 2^n}$, as previously shown in (1.1), each UCG can be decomposed as follows:

$$V_n = \text{diag}(e^{i\alpha_1}, \ldots, e^{i\alpha_{2^{n-1}}}) \otimes I_1 \cdot \text{diag}(R_z(\beta_1), \ldots, R_z(\beta_{2^{n-1}})),$$
$$\cdot I_{n-1} \otimes (SH) \cdot \text{diag}(R_z(\gamma_1), \ldots, R_z(\gamma_{2^{n-1}})) \cdot I_{n-1} \otimes (HS^\dagger) \cdot \text{diag}(R_z(\delta_1), \ldots, R_z(\delta_{2^{n-1}})).$$

2.3 Dimensionality of V_j and Λ_n

$V_j = \text{diag}(U_1, U_2, \ldots, U_{2^{j-1}})$

$\Lambda_n = \text{diag}(1, e^{i\theta_1}, e^{i\theta_2}, \ldots, e^{i\theta_{2^n-1}})$

Example with 3 qubits:

$U_k \in C^{2\times 2}$

$$V_3 = diag(U_1, U_2, U_3, U_4) = \begin{bmatrix} U_1 & 0 & 0 & 0 \\ 0 & U_2 & 0 & 0 \\ 0 & 0 & U_3 & 0 \\ 0 & 0 & 0 & U_4 \end{bmatrix} \in C^{8\times 8}$$

$$\Lambda_3 = diag(1, e^{i\theta_1}, e^{i\theta_2}, ..., e^{i\theta_7}) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & e^{i\theta_1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & e^{i\theta_2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & e^{i\theta_3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & e^{i\theta_4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & e^{i\theta_5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & e^{i\theta_6} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & e^{i\theta_7} \end{bmatrix} \in$$

$C^{8\times 8}$

2.4 State Copying

One of the advantages of Sun's algorithm is the ability to use a copy register to parallelize certain operations and reduce the overall depth of the circuit. Since the input register states are all basis states, and not arbitrary (where the *No-Cloning Theorem* would apply), and the states of the copy register are all initialized to $|0\rangle$, it is possible to copy them using simple CNOT gates with control on the input state and target on the copy qubit.

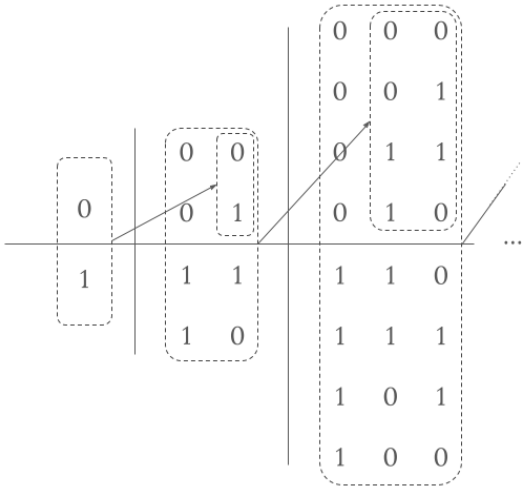
$$|0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} : \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = |00\rangle$$

$$|1\rangle \otimes |0\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} : \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = |11\rangle$$

$$|\psi\rangle \otimes |0\rangle = \begin{bmatrix} \alpha \\ 0 \\ \beta \\ 0 \end{bmatrix} : \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ 0 \\ \beta \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ 0 \\ \beta \end{bmatrix} = \alpha\,|00\rangle + \beta\,|11\rangle \rightarrow (No{-}Cloning)$$

2.5 Gray Code

To correctly assign the values of α_s to the corresponding Phase Shift gates in the implementation of Λ_n, it is necessary to use two types of Gray code, Gray-1 and Gray-2, which enable the parallelization of operations. A simple way to generate these bit sequences is by using the following pattern:



Starting with the vector $[0, 1]^T$, at each subsequent step "n", the upper part of the matrix is constructed by vertically splitting the first column of the previous step "n-1" and using it as the first column of step "n", while the entire matrix of the previous step becomes the remainder of the current step's matrix, concatenated horizontally with the first column vector. This process is repeated symmetrically for the lower part at each step, resulting in the order of the vectors that form the Gray-2 code at step "n". To obtain the Gray-1 code, the procedure is identical, except that it is executed from left to right. In practice, this means horizontally mirroring the entire matrix obtained during the Gray-2 procedure to generate the desired Gray-1 code sequence.

Gray-1 and Gray-2 Code

```
up_prefix = np.array([[0]])
down_prefix = np.array([[1]])
matrix = np.concatenate((up_prefix, down_prefix))

match n_gray:
    case 1:
        for _ in range(n - 1):
            up_prefix = np.concatenate((up_prefix, up_prefix))
            down_prefix = np.concatenate((down_prefix,
                down_prefix))
            matrix = np.concatenate((np.concatenate((matrix,
                np.flip(matrix, 0))), np.concatenate((up_prefix,
                down_prefix))), axis = 1)
    case 2:
        for _ in range(n - 1):
            up_prefix = np.concatenate((up_prefix, up_prefix))
            down_prefix = np.concatenate((down_prefix,
                down_prefix))
            matrix = np.concatenate((np.concatenate((up_prefix,
                down_prefix)), np.concatenate((matrix,
                np.flip(matrix, 0)))), axis = 1)
```

3 Generation of $\theta(x)$ and α_s

As seen in (1.2), using R_y instead of a generic U_K has simplified the QSP circuit implementation significantly. This advantage is also reflected in the computation of the angles $\theta(x)$, as they can be generated from the initial vector v using a simple Binary Search Tree (BST) with a traditional algorithm. The generation of the θ parameters is done simultaneously for each UCG by placing the coefficients of the initial vector v at the 2^n terminal nodes of the tree. Then, adjacent branches are connected pairwise to a node that contains their 2-norm, and this operation is repeated until the tree is complete, with the root node having a value of 1. Finally, by traversing the tree in reverse, each θ_i can be obtained using the following formula applied after each bifurcation:

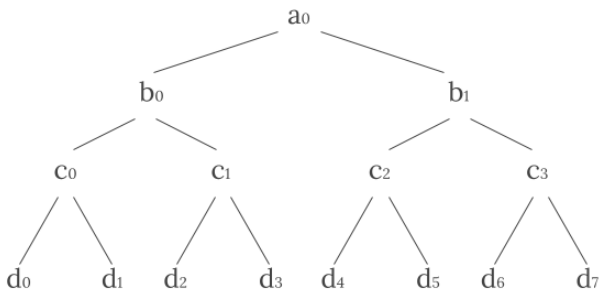
$$\theta = \arccos\left(\frac{node_{\mathrm{ROW}(j)_{\mathrm{left}}}}{node_{\mathrm{ROW}(j-1)}}\right)$$

where $node_{\mathrm{ROW}(j)_{\mathrm{left}}}$ represents the value of the left node of each bifurcation, and $node_{\mathrm{ROW}(j-1)}$ represents the value of the node that connects the two branches in the previous layer. This procedure can be also parallelized for a large number of coefficient to reduce the overall time of the computation.

Example with $n = 3$:

Let $v = (d_0, d_1, \ldots, d_7)^T$.

$c_0 = \|(d_0, d_1)\|_2, \quad b_0 = \|(c_0, c_1)\|_2, \quad \ldots$



$\theta_0 = \arccos(b_0/a_0), \quad \theta_1 = \arccos(c_0/b_0), \quad \theta_2 = \arccos(c_2/b_1), \quad \theta_3 = \arccos(d_0/c_0), \quad \theta_4 = \arccos(d_2/c_1), \quad \theta_5 = \arccos(d_4/c_2), \quad \theta_6 = \arccos(d_6/c_3).$

Binary Search Tree Code

```
class Node:
    def __init__(self, value, arc):
        self._value = value # Value of the node
        self._arc_val = arc # Value of the arc from node to parent
        self._children = [] # Children nodes

    def addChild(self, node: 'Node'): # Add a brach to the tree
        self._children.append(node)

    def nodeVal(self) -> float: # Value of the node
        return self._value

    def arcVal(self) -> str: # Value of the arc
        return self._arc_val

def generateBinTree(vector_v : list, bit : str, k : int):
    value = np.linalg.norm(vector_v, ord = 2)
    node = Node(value, bit)

    if k != 0:
        vector0 = vector_v[:len(vector_v)//2]
        vector1 = vector_v[len(vector_v)//2:]

        node.addChild(generateBinTree(vector0, '0', k - 1))
        node.addChild(generateBinTree(vector1, '1', k - 1))

    return node
```

Once the vector $\theta(x)$ corresponding to the QSP circuit has been generated, it is necessary to split it such that each θ_i is associated with its respective UCG and can later be used in the construction of Λ_n.
Since each UCG consists of 2^{j-1} controlled gates, which, as seen in (1.1), can be simplified to simple rotations about the y-axis, it becomes possible to progressively assign the θ_i's, in amounts of 2^{j-1}, to each UCG_j.
Finally, since $diag(R_z(\theta_n))$ is in the form

$$diag(R_z(\theta_n)) = \begin{bmatrix} e^{-i\theta_1} & 0 & 0 & 0 & 0 \\ 0 & e^{i\theta_1} & 0 & 0 & 0 \\ 0 & 0 & ... & 0 & 0 \\ 0 & 0 & 0 & e^{-i\theta_n} & 0 \\ 0 & 0 & 0 & 0 & e^{i\theta_n} \end{bmatrix}$$

while Λ_n is defined as

$$\Lambda_n = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & e^{i\theta_1} & 0 & 0 & 0 \\ 0 & 0 & ... & 0 & 0 \\ 0 & 0 & 0 & e^{i\theta_{n-1}} & 0 \\ 0 & 0 & 0 & 0 & e^{i\theta_{2^n-1}} \end{bmatrix}$$

it is necessary to collect the first element of $diag(R_z(\theta_n))$ in order to obtain the remaining $(2^n - 1)$ θ_i's from combinations such as $[(\theta_1 + \theta_1), (-\theta_2 + \theta_1), (\theta_2 + \theta_1), \dots]$, as seen in (2.3), which will be the final angles used to generate the α_s. The use of α parameters is necessary to implement rotations on the Fourier basis and consequentially use ancillary qubits to transition from the multiplexor representation to the Λ_n, by reducing the computational complexity of the circuit. After calculating all the θ angles, since the algorithm needs to implement $2^n - 1$ Phase Shift operations on each state of the computational basis, the value of the α parameters can be calculated by solving the system $\hat{A}\vec{\alpha} = \vec{\theta}$, where $\hat{A}$ represents the coefficient matrix constructed using $\langle s|x\rangle$ in F_2 for each $|x\rangle$ of the computational basis and for all binary strings s, excluding the all-zeros string.

$\langle 01|00\rangle = 0 \oplus 0 = 0; \langle 01|01\rangle = 0 \oplus 1 = 1; \langle 01|10\rangle = 0 \oplus 0 = 0; \langle 01|11\rangle = 0 \oplus 1 = 1;$
$\langle 10|00\rangle = 0 \oplus 0 = 0; \langle 10|01\rangle = 0 \oplus 0 = 0; ...$
...

$$\begin{cases} \alpha_{01} \langle 01|01\rangle + \alpha_{10} \langle 01|10\rangle + \alpha_{11} \langle 01|11\rangle = \theta_1 \\ \alpha_{01} \langle 10|01\rangle + \alpha_{10} \langle 10|10\rangle + \alpha_{11} \langle 10|11\rangle = \theta_2 \\ \alpha_{01} \langle 11|01\rangle + \alpha_{10} \langle 11|10\rangle + \alpha_{11} \langle 11|11\rangle = \theta_3 \end{cases}$$

4 Implementation of Λ_n

The implementation of Λ_n is a fundamental step in the algorithm, as it allows the efficient construction of the diagonal unitary matrix used in the decomposition of V_n.
The matrix Λ_n can be obtained through a quantum circuit with a size of $O(2^n)$ and depth $O(\log_2 m + \frac{2^n}{m})$, where n represents the number of qubits in the input register and m represents the number of ancillary qubits used. Each Λ_n implements a PhaseShift on each state of the computational basis and makes use of rotations in the Fourier basis to parallelize the operations.
The proposed circuit is divided into 5 stages, which define a logical structure for the operations and facilitate the scalability of the implementation. To simplify the implementation, the circuit uses $m = 2n$ ancillary qubits, that is the lower bound of the analysed range, but can be relatively easily generalized to all the possible integer values in the range.

4.1 Prefix Copy Stage

In this preliminary stage, $\lfloor m/2t \rfloor$ copies of each qubit $x_1, x_2, ..., x_t$, where $t = \lfloor \log_2(m/2) \rfloor$, are made from the input register to the copy register. Since the state of the qubits in the input register is in a basis state, the copies can be created simply using CNOT gates (2.4).
The effect of these operations on the system's state is as follows:

$$|x\rangle |0^{m/2}\rangle \rightarrow |x\rangle |x_{pre}\rangle$$

Prefix Copy Stage Code

```
k = int(floor(self._ancillaries / (2 * self._t)))

for x in range(self._t):
    control, target = x, self._qubits + x
    qml.CNOT(wires = [control, target])

for i in range(k - 1):
    for x in range(self._t):
        control, target = (self._t * i) + self._qubits + x,
            self._t + (self._t * i) + self._qubits + x
        qml.CNOT(wires = [control, target])
```

4.2 Gray Initial Stage

After initializing the copy register, the operations on the phase register are performed. The first step is to execute $m/2$ linear functions based on a sequence of n-bit strings, while the second step involves implementing a number of rotations (Phase Shift) equal to the number of qubits in the phase register that were affected by the previous step.
To compose the n-bit strings, the computational basis vectors of t-bit are generated and concatenated with the $2t$ strings of $(n-t)$-bit with '0' values.
These two steps allow the addition of a global phase to the system's state:

$$|x\rangle |x_{pre}\rangle |0^{m/2}\rangle \rightarrow e^{i \sum_{j \in [l]} f_{j,1}(x) \alpha_{s(j,1)}} |x\rangle |x_{pre}\rangle |f_{[l],1}\rangle$$

Gray Initial Stage Code

```
binary = ['0' * (self._qubits - self._t)] * pow(2, self._t)
basis = [s[::-1] for s in compBasis(self._t)]
strings = list(map(add, basis, binary))

for i, string in enumerate(strings):
    for j, bit in enumerate(string):
        if bit == '1':
            control, target = self._qubits + j, self._qubits*2 + i
            qml.CNOT(wires = [control, target])

for i, id in enumerate(strings):
    if '1' in id:
        qml.PhaseShift(self._alphas[id], wires = [self._qubits*2 +
            i])
```

4.3 Suffix Copy Stage

In this stage, the inverse of the operations on the copy register performed in the previous stage is first implemented, in order to reset the states of the register. Afterward, $t = \lfloor (m/[2(n-t)]) \rfloor$ copies are created for each of the qubits in the suffix of the input register $x_{t+1}, x_{t+2}, ..., x_n$. This allows to reuse the qubits of the copy and phase registers as before, but with the remaining qubits of the input register:

$|x\rangle |x_{pre}\rangle \rightarrow |x\rangle |0^{m/2}\rangle \rightarrow |x\rangle |x_{suf}\rangle$

Suffix Copy Stage Code

```
k = int(floor(self._ancillaries / (2 * self._t)))

for i in range(k-1):
    for x in range(self._t)[::-1]:
        control, target = (self._t * i) + self._qubits + x,
            self._t + (self._t * i) + self._qubits + x
        qml.CNOT(wires = [control, target])

for x in range(self._t)[::-1]:
    control, target = x, self._qubits + x
    qml.CNOT(wires = [control, target])

k = int(floor(self._ancillaries / (2 * (self._qubits - self._t))))

for x in range(self._qubits - self._t):
    control, target = self._t + x, self._qubits + x
    qml.CNOT(wires = [control, target])

for i in range(k - 1):
    for x in range(self._qubits - self._t):
        control, target = ((self._qubits - self._t) * i) +
            self._qubits + x, (self._qubits - self._t) +
            ((self._qubits - self._t) * i) + self._qubits + x
        qml.CNOT(wires = [control, target])
```

4.4 Gray Path Stage

This stage implements $2^n/(2t-1)$ frames in which, similarly to the Gray Initial stage, CNOT gates and Phase Shift operations are applied to the phase register. The linear functions implemented at each frame have a dependency on the previous frame, that is because every linear function is implemented through CNOT gates, corresponding to the change of some bits in the binary strings of the previous frame and the current frame. In the first frame, the previous binary strings correspond to those used in the Gray Initial stage, while in subsequent frames, those from the preceding frame are used.

To determine the binary strings of the current frame, a scheme based on Gray Code is used, where the operation strings are divided internally and split based on the position of the qubits in the phase register.

The first split is made by dividing each string into two parts; the first t elements consist of strings from the computational basis, which are then concatenated with corresponding binary values generated from Gray-1 and Gray-2 Code, depending on the ancillary qubit being used.

The second split is made horizontally across the entire phase register to de-

termine which Gray Code to use in the aforementioned step. By dividing the register into two parts, the strings associated with the first $2^t/2$ ancillary qubits use Gray-1 Code to generate the suffix of the string, while for the remaining ancillary qubits, Gray-2 Code is used instead (2.5).

$$|x\rangle |x_{suf}\rangle |f_{[l],k-1}\rangle \rightarrow |x\rangle |x_{suf}\rangle |f_{[l],k}\rangle \rightarrow e^{i \sum_{j \in [l]} f_{j,k}(x) \alpha_{s(j,k)}} |x\rangle |x_{suf}\rangle |f_{[l],k}\rangle$$

Gray Path Stage Code

```
gray_1 = grayCode(1, self._qubits - self._t)
gray_2 = grayCode(2, self._qubits - self._t)

basis = [s[::-1] for s in compBasis(self._t)]
binary = ['0' * (self._qubits - self._t)] * pow(2, self._t)

prev = list(map(add, basis, binary))

for k in range(int(pow(2, self._qubits) / pow(2, self._t) - 1)):

    curr = [] # Current phase string vector

    for i, id in enumerate(basis):
        g1 = str("".join(map(str, gray_1[k + 1])))
        g2 = str("".join(map(str, gray_2[k + 1])))
        curr.append(id + g1 if i < (len(basis) / 2) else id + g2)

        for i, id in enumerate(basis):
            curr_str = curr[i]
            prev_str = prev[i]
            change_vector = [True if b1 != b2 else False for b1,
                b2 in zip(prev_str, curr_str)]

        for bit, change in enumerate(change_vector[self._t:]):
            if change:
                control, target = self._qubits + bit, self._qubits
                    * 2 + i
                qml.CNOT(wires = [control, target])

        for i, id in enumerate(basis):
            string = curr[i]
            qml.PhaseShift(self._alphas[string], wires =
                [self._qubits * 2 + i])

        prev = curr
```

4.5 Inverse Stage

The last stage simply consists of a sequence of inverse operations corresponding to all the steps in which CNOT gates were applied, in order to return the final state of the system's qubits to the starting state net of a global phase.

$|x\rangle |x_{suf}\rangle |f_{[l],2^n/l}\rangle \rightarrow |x\rangle |0^{m/2}\rangle |0^{m/2}\rangle$

Inverse Stage Code

```
# INVERSE GRAY PATH STAGE
gray_1 = grayCode(1, self._qubits - self._t)
gray_2 = grayCode(2, self._qubits - self._t)

basis = [s[::-1] for s in compBasis(self._t)]
binary = ['0' * (self._qubits - self._t)] * pow(2, self._t)
strings = [list(map(add, basis, binary))]

for k in range(int(pow(2, self._qubits) / pow(2, self._t) - 1)):

    s = []

    for i, id in enumerate(basis):
        g1 = str("".join(map(str, gray_1[k + 1])))
        g2 = str("".join(map(str, gray_2[k + 1])))
        s.append(id + g1 if i < (len(basis) / 2) else id + g2)

    strings.append(s)

strings = strings[::-1]

for k in range(int(pow(2,self._qubits) / pow(2,
    self._t)-1))[::-1]:

    curr = strings[k]
    post = strings[k + 1]

    for i in range(len(basis))[::-1]:
        curr_str = curr[i]
        post_str = post[i]
        change_vector = [True if b1 != b2 else False for b1, b2 in
            zip(post_str, curr_str)]
        for bit, change in enumerate(change_vector[self._t:]):
            if change:
                control, target = self._qubits + bit, self._qubits
                    * 2 + i
                qml.CNOT(wires = [control, target])
```

```
# INVERSE SUFFIX COPY STAGE
k = int(floor(self._ancillaries / (2 * (self._qubits - self._t))))

for i in range(k - 1):
    for x in range(self._qubits - self._t)[::-1]:
        control, target = ((self._qubits - self._t) * i) +
            self._qubits + x, (self._qubits - self._t) +
            ((self._qubits - self._t) * i) + self._qubits + x
        qml.CNOT(wires = [control, target])

for x in range(self._qubits - self._t)[::-1]:
    control, target = self._t + x, self._qubits + x
    qml.CNOT(wires = [control, target])

k = int(floor(self._ancillaries / (2 * self._t)))

for x in range(self._t):
    control, target = x, self._qubits + x
    qml.CNOT(wires = [control, target])

for i in range(k - 1):
    for x in range(self._t):
        control, target = (self._t * i) + self._qubits + x,
            self._t + (self._t * i) + self._qubits + x
        qml.CNOT(wires = [control, target])

# INVERSE GRAY INITIAL STAGE
strings = list(map(add, basis, binary))

for i, string in enumerate(strings[::-1]):
    for j, bit in enumerate(string[::-1]):
        if bit == '1':
            control, target = self._qubits + len(string) - 1 - j,
                self._qubits * 2 + len(strings) - 1 - i
            qml.CNOT(wires = [control, target])

# INVERSE PREFIX COPY STAGE
for i in range(k-1):
    for x in range(self._t)[::-1]:
        control, target = (self._t * i) + self._qubits + x,
            self._t + (self._t * i) + self._qubits + x
        qml.CNOT(wires = [control, target])

for x in range(self._t)[::-1]:
    control, target = x, self._qubits + x
    qml.CNOT(wires = [control, target])
```

5 Simulation and Testing Using HPC Facility

To simulate the implementation, a CPU node from the HPC facility at the University of Parma was used, which provided much faster machines for testing the system.

Tests are being conducted on both the implementation of Λ_n and the entire system for QSP with $n = 2$ qubits as input. The implementation of Λ_n for $n = 2, 3, 4, 5$ was able to consistently reproduce the correct circuit according to Sun's algorithm and implement the diagonal matrix described in (1.2).

From the circuit representation, it was observed that in some cases, some ancillary qubits were not used. Thus, it may be possible to further reduce the size of the circuit for particular values of n.

Regarding the implementation of the full circuit, tests confirmed the system's ability to produce the desired quantum state, whether starting from real or complex coefficients. One consideration that was made was to compare the resulting state with the initial vector, in the case of complex coefficients, using the l_2-norm of the values. This is because, when generating the parameters through the Binary Search Tree (BST), it uses the norm of the coefficient vector to produce the θ_i values.

6 Conclusioni

As a final result, we successfully implemented both a system to generate a Λ_n circuit with a variable n as well as the entire Quantum State Preparation circuit, at net of a global phase, for two qubits initialized in state $|0\rangle$, following Sun's algorithm. Knowing how the algorithm works and how the subcircuits are concatenated, and utilizing the current implementation as a foundation, a promising starting point for a future project on this topic would be to enhance this implementation and develop a system with a variable value of n to prepare an arbitrary state.

The generation of all parameters, particularly θ and α, was carried out using a classical computing algorithm, scalable for n-qubit circuits.

It was observed that, during the generation of the α parameters, some of them consistently take the value 0 depending on n. Consequently, in the implementation of Λ_n, some PhaseShift gates reduce to identity matrices, thus contributing nothing to the circuit. In light of this, a possible future development could involve refining the algorithm to exclude these parameters a priori, thereby reducing both the computational cost of coefficient generation and the number of gates used in the circuit.

On a personal note, this project was also useful for gaining hands-on experience with the HPC system of the University of Parma and for exploring in detail some topics not covered in the course, as well as for learning how to use the PennyLane library for quantum computing purposes.

7 References

[1] Xiaoming Sun, Guojing Tian, Shuai Yang, Pei Yuan and Shengyu Zhang, "Asymptotically Optimal Circuit Depth for Quantum State Preparation and General Unitary Synthesis", 2023, arXiv:2108.06150v3 [quant-ph]

[2] Lov Grover and Terry Rudolph. Creating superpositions that correspond to efficiently integrable probability distributions. arXiv preprint quant-ph/0208112, 2002.

This project benefits from the High Performance Computing facility of the University of Parma, Italy (HPC.unipr.it)