Quantum Tranversals

Chams Rutkowski, David Castells-Graells, Eleftherios Tselentis

January 15, 2019

Abstract

In this project we present a code that implements a quantum circuit in *Qiskit* that generates a quantum transversal, either by entering the specific transversal, or by inputting the symmetry group we want the transversal from. Our program uses an algorithm to generate the desired final states based on control^n single-qubit unitary gates. Such gates are implemented with a construction optimized for small numbers of qubits, although the roadmap to extend the code for optimal large number of qubits is presented. The results are fully satisfactory and the code successful. A GUI has been implemented for a better accessibility. New ideas to take the code further are presented in the outlook.

1 Introduction

In this report we present an algorithm that implements a quantum circuit in *Qiskit* that generates an equal superposition of the elements of a transversal of a given symmetry group. The report is structured as follows. We first make some insights onto the concept of the transversal, how to obtain it classically, and how to be efficient about its generation. Then we tackle the problem of generating such state. We present the algorithm and the specificities on how to implement its most complex parts, and how to use it for the specific problem of the transversals. Such code is implemented and the main results presented. We have designed a GUI such that any external user can make use of our code. Its documentation can be found in the appendices.

2 The transversal of a symmetric group

Being G the set of all the possible combinations of n-bit strings and S a symmetric group, a transversal is defined as the set containing one element for each of the subsets invariant under the action of a permutation in S in which G can be partitioned. The brute-force approach to generate the partition of G from which to pick a transversal would be to generate the set G, take one of its elements and generate all the possible permutations in S. Then, these strings would be removed from G and this same process would be repeated until all the elements of G were assigned to a subset. This process can get highly inefficient for large n, given the large size of G (equal to 2^n), and S (equal to $n!$ for the fully symmetric group), over which we will have to do search, and

apply all permutations to each element taken from G (which will lead to many redundancies), respectively.

However, we can take a short-cut to the job of creating the set of partitions focusing first on the fully symmetric group. In the fully symmetric group, any element can be exchanged by any other. Therefore, we can trivially see that all the strings with the same number of 1s must be part of the same subset. This means that for a fully symmetric group of n elements, G will have $n+1$ partitions that can be generated as all the possible ways to arrange $0 \leq m \leq n$ identical elements in n, which is computationally inexpensive. For this job we use Python combinatorics module. For easier handling of the strings in future manipulations, we calculate the decimal values out of the binary strings.

The partitions of a general symmetric group, can be readily obtained combining the ones of the fully symmetric subgroups that form it. Since permutation subgroups are closed, a subset of a fully symmetric subgroup will stay an independent subset from the rest of its partition regardless of what strings you might append. Hence, two partitions of two fully symmetric groups form the partition of the combined subgroups by doing a *tensor product* between them. For instance, for the S_2xS_2 group with subpartitions $\{\{00\},\{01,10\},\{11\}\}$ would form:

$$\{\{00\},\{01,10\},\{11\}\} \otimes \{\{00\},\{01,10\},\{11\}\} = \\ \left\{ \begin{array}{ccc} \{0000\} & \{0001,0010\} & \{0011\} \\ \{0100,1000\} & \{0101,0110,1001,1010\} & \{0111,1011\} \\ \{1100\} & \{1101,1110\} & \{1111\} \end{array} \right\} \quad (1)$$

With all these considerations we have a quick way to generate symmetric partitions of G from which to extract transversals. Next we study how to make a quantum circuit to create the so-called quantum transversals, or uniform quantum superpositions of the elements of a transversal.

3 Implementation of the quantum circuit

To build the circuit that generates the quantum transversal we use the Qiskit *quantumcircuit* module in Python [3], which is especially convenient since it integrates the IBM QASM simulator [4] that we will use to test the outcome of our circuit.

In the course of this section, we describe the components necessary to implement a suitable quantum circuit. A scheme to generate any arbitrary superposition of n-qubit states available in the literature is discussed in section(3.1). With this circuit, implementing the algorithm to generate the desired superimposed state reduces to the implementation of one-qubit unitary operations controlled by an arbitrary number of estates. We delve into the decomposition of these control^n gates into a universal set of single and two-qubit gates in section(3.2). Finally, we make some notes on how to best implement the designed circuit to generate transversals.

3.1 Generating an arbitrary superposition of n-qubit states

The algorithm proposed by [1] can generate on-demand any superposition of n-qubit states with a claimed complexity of $O(Nn^2)$, where N is the number of non-zero amplitudes and n the number of qubits. This complexity, however, highly depends on the complexity of the subroutines, mainly the implementation of n-qubit controls, as discussed later.

The algorithm to construct the circuit is described in Box 1.

Box 1. Algorithm to generate an arbitrary superposition of n qubits [1]

We are given a list of elements $\{T_i\}$ of size $N \in [1, 2^n]$. We use a quantum circuit of n qubits with initial state $|0\rangle^{\otimes n}$
1. Apply a single qubit gate U_1 on the first qubit
2. Apply two control[1]-gates $U_{2,0}, U_{2,1}$ on the second qubit (controlled on the fisrt qubit)
3. Apply four control[2]-gates $U_{3,00}, U_{3,01}, U_{3,10}, U_{3,11}$ on the third qubit (controlled on the first and second qubits)
$\vdots$
n. Apply 2^{n-1} control[n-1]-gates $\left\{U_{n,i_1,i_2,...,i_{n-1}}\right\}$ on the n qubit (the control qubits are the previous $n-1$)

And the equivalent circuit is drawn in fig(1).

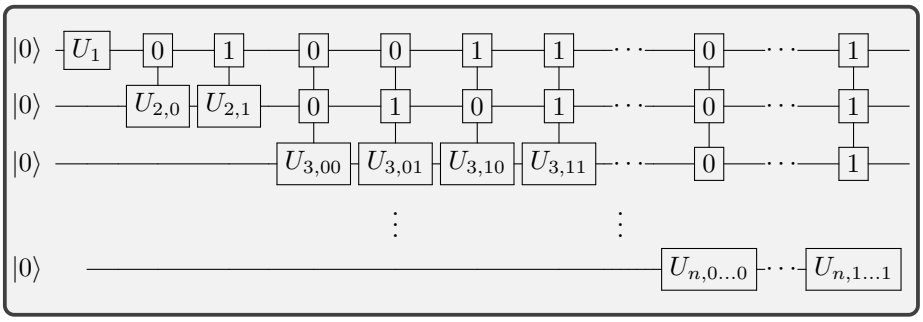

Figure 1: Quantum circuit that generates an arbitrary on-demand superposition of n qubits

Note that each qubit is sequentially acted on once in the circuit. The overall unitary acting on the initial state is:

$$|\psi_{final}\rangle = \Big(\sum_{i_1,...,i_{n-1}} U_{n,(i_1,...,i_n)} \, |i_1, ..., i_{n-1}\rangle \langle i_1, ..., i_{n-1}| \Big) \cdots \\ \cdots (U_{2,0} \, |0\rangle \langle 0| + U_{2,1} \, |1\rangle \langle 1|) \, U_1 \, |0\rangle^{\otimes n} \tag{2}$$

The single unitary operations acting on the j qubit conditioned on the $1, ..., j-1$ qubits are defined as

$$U_{j,(i_1,...,i_j-1)} = \begin{bmatrix} \cos\alpha_{j,(i_1,...,i_j-1)} & \sin\alpha_{j,(i_1,...,i_j-1)} \\ \sin\alpha_{j,(i_1,...,i_j-1)} & -\cos\alpha_{j,(i_1,...,i_j-1)} \end{bmatrix} \tag{3}$$

with α given by

$$\alpha_{j,(i_1,i_2,...,i_{j-1})} = \arctan \sqrt{\frac{\sum_{i_{j+1}}^{i_n} \left|a_{i_1,i_2,...,i_{j-1},1,i_{j+1},...i_n}\right|^2}{\sum_{i_{j+1}}^{i_n} |a_{i_1,i_2,...,i_{j-1},0,i_{j+1},...i_n}|^2}} \ , \tag{4}$$

where $a_{i_1,i_2,...,i_n}$ are the coefficients of the desired output state.

To gain some insight into the algorithm we briefly analyze what happens at each step. As we can see, the angle[1] $\alpha_{j,(i_1,i_2,...,i_{j-1})}$ that defines the unitary $U_{j,(i_1,...,i_j-1)}$ depends basically on two sums over coefficients of the target final state matching the specified control, which we will label

$$\text{sum}_1 \equiv \sum_{i_{j+1}}^{i_n} \left|a_{i_1,i_2,...,i_{j-1},1,i_{j+1},...i_n}\right|^2 \tag{5}$$

$$\text{sum}_0 \equiv \sum_{i_{j+1}}^{i_n} \left|a_{i_1,i_2,...,i_{j-1},0,i_{j+1},...i_n}\right|^2 , \tag{6}$$

for conciseness.

If there exists no element in the target state, with the first j qubits in the state $(i_1, i_2, ..., i_{j-1}, 1)$, then $\text{sum}_1 = 0$, $\alpha = 0$ and $U_{j,(i_1,...,i_j-1)} = \sigma_z$. Thus, the jth qubit (previously $|0\rangle$) is left unchanged and the $(i_1, i_2, ..., i_{j-1}, 0, ...)$ branch of fig.(2) survives, while the $(i_1, i_2, ..., i_{j-1}, 0, ...)$ vanishes.

Conversely, if there exists no element with the first j qubits in the state $(i_1, i_2, ..., i_{j-1}, 0)$, then $\text{sum}_0 = 0$, $\alpha = \frac{\pi}{2}$ and $U_{j,(i_1,...,i_j-1)} = \sigma_x$, flipping the jth qubit to $|1\rangle$. The surviving branch is now $(i_1, i_2, ..., i_{j-1}, 1, ...)$.

If for both $(i_1, i_2, ..., i_{j-1}, 0)$ and $(i_1, i_2, ..., i_{j-1}, 1)$, there exists at least one element in target matching the control string, then both sum_1 and $\text{sum}_0 \neq= 0$, and the unitary gate superimposes the two states. Both branches survive.

Consequently, the algorithm ensures that after step j is completed, for any surviving superimposed state exists at least one element in the target state matching the state of the first j qubits.

This feature of the algorithm is especially relevant to address the case $\text{sum}_1 = \text{sum}_0 = 0$, which would lead to an undetermined α, an issue that is not well addressed in the original paper [1] in which they propose a case-by-case analysis.

[1]It is convenient to use the word *angle* referring $\alpha_{j,(i_1,...,i_j-1)}$. However, note that the unitary matrix $U(\alpha)$ is not an actual rotation

Such requirement would mean a major difficulty for our quantum circuit, as we want a code that can work unsupervised for any target state. However, we have shown that an undetermined α never happens (see Appendix A). In short, we prove that the controls of the problematic unitary gates are never activated, as any possible controlled state have been eliminated in the previous steps with the process described above.

We can go a bit further into the analysis of the algorithm to understand how, apart from generating a specific set of states, it also assigns them the right weights. Fig.(2) is very enlightening for this matter. Two split branches will never recombine (for the simple reason that we act on each qubit only once). Therefore, at the moment of splitting we already now the total weight of the states at the end of the branch. In one line, the algorithm subsequently splits the weight in a controlled way towards the target states. In essence, eq.(4) compares the weights of the two branches (sum_1 and sum_0) and splits the weight of the controlled state accordingly.

After the analysis described in this section we have verified that the proposed algorithm is suitable for the task of making circuits that generate quantum transversals in an automated way.

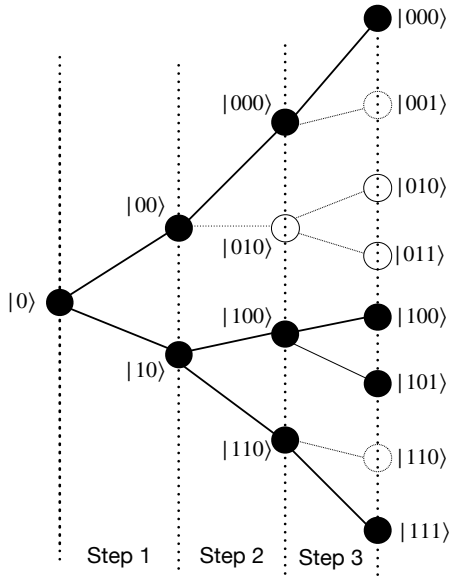

Figure 2: Tree diagram to exemplify how the branching from the $|0\rangle^{\otimes 3}$ to the target superposition of states $\{T\} = \{|000\rangle, |100\rangle, |101\rangle, |111\rangle\}$ in uniform superposition is realized. For instance, from eq.(9) and eq.(4), $U_{2,0} = \hat{\sigma}_z$, thus leaving $|00\rangle$ unchanged, so the branch $|01...\rangle$ is extinguished. This is expected from the arguments made in the text, since there exists no element in T having its first two qubits in the state $|01\rangle$

3.2 The control^n gate

To decompose the control^n gates required by our quantum state generator into a universal set of single and two-qubit gates, such that it becomes physically implementable, we use the constructions proposed in [5]. In this work they propose different methods to implement such gate. On the one hand, they introduce a construction that, to the best of their knowledge, is the most efficient for a small number of qubits involved (up to 8). However, it becomes inefficient for large number of qubits, with its complexity scaling as 2^n.

To deal with larger numbers of qubits efficiently, in [5] they also propose schemes for control^n gates with quadratic complexity, and some approximate methods (or exact for a set of special cases) to achieve sub-quadratic performance. Another option is the use of ancilla qubits, like in the scheme proposed in [6]. In such case, reduced complexity come in expense of a larger system (doubling of the number of qubits), which is not preferred and, thus, the other options are pursued.

For this project we choose the first option because it is the most clear to visualize and is optimal for small number of qubits, the cases that we might want to test in a demonstrative manner. To allow for future extensions of our program, such as optimizing the outcome circuit for large number of qubits, we have designed the code in a highly modular way such that new constructions for the control^n gate can be easily implemented and let the user choose among the different options.

In the rest of this section we explain first the scheme of the chosen control^n gate, and second how we implement it to satisfy our specific needs.

3.2.1 Gate decomposition

This gate is based in the following identity [5]:

$$\begin{aligned}&\sum_{k_1} x_{k_1} - \sum_{k_1<k_2} (x_{k_1} \oplus x_{k_2}) + \sum_{k_1<k_2<k_3} (x_{k_1} \oplus x_{k_2} \oplus x_{k_3}) - \cdots + \\ &+(-1)^{m-1}(x_1 \oplus x_2 \oplus \cdots \oplus x_m) = 2^{m-1} \cdot (x_1 \wedge x_2 \wedge \cdots \wedge x_m)\ ,\end{aligned} \tag{7}$$

where x_i is the ith control bit.

Therefore, having each +(-) term of the left-hand-side of the above identity encode a unitary operation $V(V^\dagger)$, is equivalent to implementing a gate V 2^{m-1} times, or a $V^{2^{m-1}}$ gate, if an only if all involved bits are one, i.e. such construction simulates a control^n gate. Note that the choice between $V/V^\dagger$ is given by the parity of the bits involved. Hence, for a system with n qubits and $m = n-1$ control qubits, implementing the above scheme setting $V^{2^{n-2}} = U$ simulates a gate U on the nth qubit control by the other m qubits.

To give an specific example of the circuit we use the one for $n = 4$ illustrated in [5]. The resulting circuit is shown in figure(3), and the sequence of operators

performed on the fourth bit are:

$$\begin{array}{lll} V & \text{iff } x_1 = 1 & (100) \\ V^\dagger & \text{iff } x_1 \oplus x_2 = 1 & (110) \\ V & \text{iff } x_2 = 1 & (010) \\ V^\dagger & \text{iff } x_2 \oplus x_3 = 1 & (011) \\ V & \text{iff } x_1 \oplus x_2 \oplus x_3 = 1 & (111) \\ V^\dagger & \text{iff } x_1 \oplus x_3 = 1 & (101) \\ V & \text{iff } x_3 = 1 & (001)\ . \end{array}$$

For an efficient implementation of these sequence, they propose a Gray-Code sequence of operators, as encoded by the strings on the right.

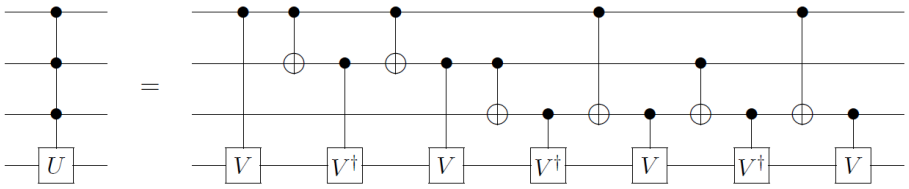

Figure 3: Implementation of a control^3 U gate, where $V^4 = U$. [5]

3.2.2 Implementation

In the next lines we briefly describe how this gate is implemented for our system and how to code it. Three main issues need to be assessed: **(i)** to find a unitary matrix V such that $V^{2^{n-2}} = U$, where U is the unitary gate in our algorithm, eq(9); **(ii)** to allow any array of control bits containing both 0's and 1's; **(iii)** and to define an algorithm to implement the sequence of operators.

(i) Although U is not a rotation, it is closely related to the R_y (rotation around y) gate:

$$U(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ \sin(\theta) & -\cos(\theta) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = R_y(\theta) \cdot Z \quad (8)$$

Note that Z is not angle dependent, so it does not depend on the specific control in the algorithm, unlike θ. Together with the fact that this gate needs to be applied once and only once on all the superimposed states, allows us to factor it out of the control. Hence, the controlled U gate is equivalent to a controlled R_y followed by a Z gate. This realization is crucial, as having a rotation under the control greatly simplifies the task of splitting the unitary matrix: $V^{2^{n-2}} = R_y(\theta) \Rightarrow V = R_y(\theta/2^{n-2})$.

The controlled R_y gate is available in Qiskit under the form of

$$u3(\theta, \phi = 0, \gamma = 0) = \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix} \quad (9)$$

Hence, $V = u3(\theta/2^{n-3}, 0, 0)$ and $V^\dagger = u3(-\theta/2^{n-3}, 0, 0)$.

(ii) Eq.(7) could be adapted to allow controls different than the traditional all-1's. However, such task is not straightforward and would prevent us from the efficiency and algorithmic convenience of the Gray code. Instead, we opted to convert the control 0's into 1's before implementing the controlled gate, and undo this operation under. For this we use X (NOT) gates on the 0 control bits.

(iii) *Sympy combinatorics* module generates Gray-code sequences. After some slight modifications a sequence like the one in the 4-qubit example above is obtained. The virtue of following the Gray-code, which makes it efficient for our job, is that one can iterate over the subsequent elements using only one C-NOT gate at a time. How the Gray-code encodes $V/V^\dagger$ and the C-NOT in between might not be always obvious. To streamline its implementation in our code we think of the Gray-code arrays as arrays of qubits and define the following algorithm. It is important to note that in such sequence the rightmost 1 never retrocedes, i.e. $(\ldots 10)$ will never come after $(\ldots 01)$ or $(\ldots 11)$:

Box2. Algorithm to build a control^n gate iterating over a Gray-code

1. Starting from $(10\ldots0)$, apply V controlled on the first qubit (define parity = 1)
2. Take the next element of the Gray code and compare it to the previous one (re-define parity = -parity)
3. If a new rightmost 1 is created, apply on such qubit a X gate controlled by the qubit at its left. If a 1 have been created or converted back to 0 in any other position, apply a X gate on the current rightmost 1 qubit controlled on the qubit at the modified position
4. Apply a V gate controlled on the rightmost 1 qubit with the sign of the angle given by parity = ±1
5. Repeat steps 2-4 until the end of the Gray code

3.3 Generating the quantum transversal

Following on the discussion on section(2), the problem of finding the transversal of any symmetric group on N elements $\prod_m S_{\nu_m}$, with $\sum_m \nu_m = N$, reduces to finding the transversals of each of the fully symmetric subgroups that form it. A direct consequence of this is the fact that the tensor product of m of such *sub-transversals* is a transversal of the full group. This realization much simplifies the quantum circuit required to generate a quantum transversal, as the cost of generating an on-demand superposition state of n qubits is highly non-linear with n, as have already discussed in section(3.1) and (3.2). Hence, the full transversal would be the product state of the *sub-transversals* generated for each fully symmetric subgroup.

The resulting transversal from acting with any permutation in the symmetry group on the transversal described above could still be generated in the same way. However, not any transversal can be expressed as the tensor product of *sub-transversals*, as can be very easily seen in the following example. Take the group

S_2xS_2, which have a partition: $\{\{0\},\{1\}\} \otimes \{\{00\},\{01,10\},\{11\}\}$. A product state can generate, for instance, $T = \{0,1\} \otimes \{00,10,11\} = \{000,100,010,110,011,111\}$, but it cannot do $T = \{000,100,0\mathbf{01},1\mathbf{10},011,111\}$. In other words, a transversal obtained as a tensor product between *sub-transversals* can only use at the same time one element of each subset into which a subgroup is partitioned.

Therefore, if we have do not have a preference on the specific quantum transversal to generate, the computational complexity can be much reduced by generating sub-transversals, with complexity corresponding to the one of generation the transversal for the largest fully symmetric subgroup. On the other hand, if we have a preferred transversal, we might generate it at a higher computational cost. Based on this decision, our code generates the transversal in sub-blocks or with one full block of qubits.

4 Results and interpretation

Putting together all the work up until this point, we have written a program that generates transversals (any or the user's preferred), apply permutations on them, generates a suitable quantum circuit and simulates it with QASM. For a more detailed documentation see Appendix B.

We have also programmed a simple GUI to make it readily accessible for any interested user. The program is also equipped with some metrics to evaluate whether the quantum transversal was created successfully (see Appendix B). In this report, thus, we focus on some simple illustrative cases, but it has been widely tested with far more demanding requests. Namely we show the results of the simulation of one of our outputs circuits, and then show two examples of the obtained circuits both for the case of a *separable* (as a product of sub-transversals) and *non-separable* transversal.

In fig.(4) the results obtained for the fully symmetric group S_3 are presented. The obtained values are fully compatible with a uniform superposition of the desired transversal elements.

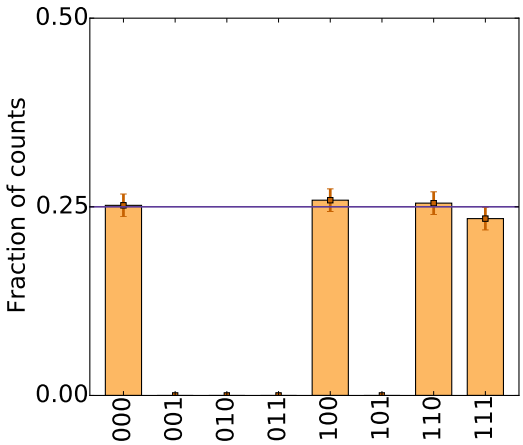

Figure 4: Simulation results for a quantum transversal of the S_3 fully symmetric group. Namely, $T = \{000, 100, 110, 111\}$. The results are normalized by the total number of experiments (1024) and compared to the expected value for a uniform superposition (solid line). The error bars are calculated theoretically as discussed in Appendix B, and indicate that the miss-match of the peaks is compatible with statistical fluctuations

In fig.(5) the circuit to generate a transversal of the S_1xS_2 group. In this case we chose a *separable* transversal, so the circuit is split in two blocks, qubit1 and qubit2-3. This makes the circuit much simpler as compared to the one to simulate a *non-separable* transversal of the same symmetry group, fig.(6).

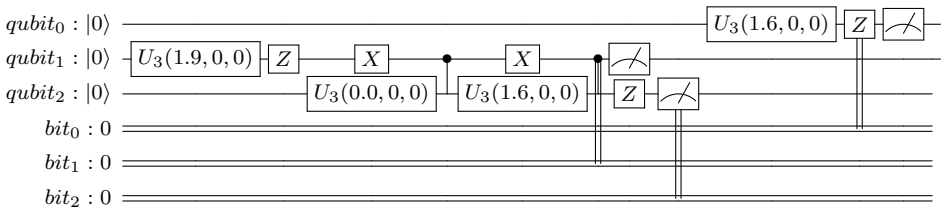

Figure 5: Quantum circuit to generate the transversal $T = \{000, 010, 011, 100, 110, 111\}$ of the S_1xS_2 group, as output by our program

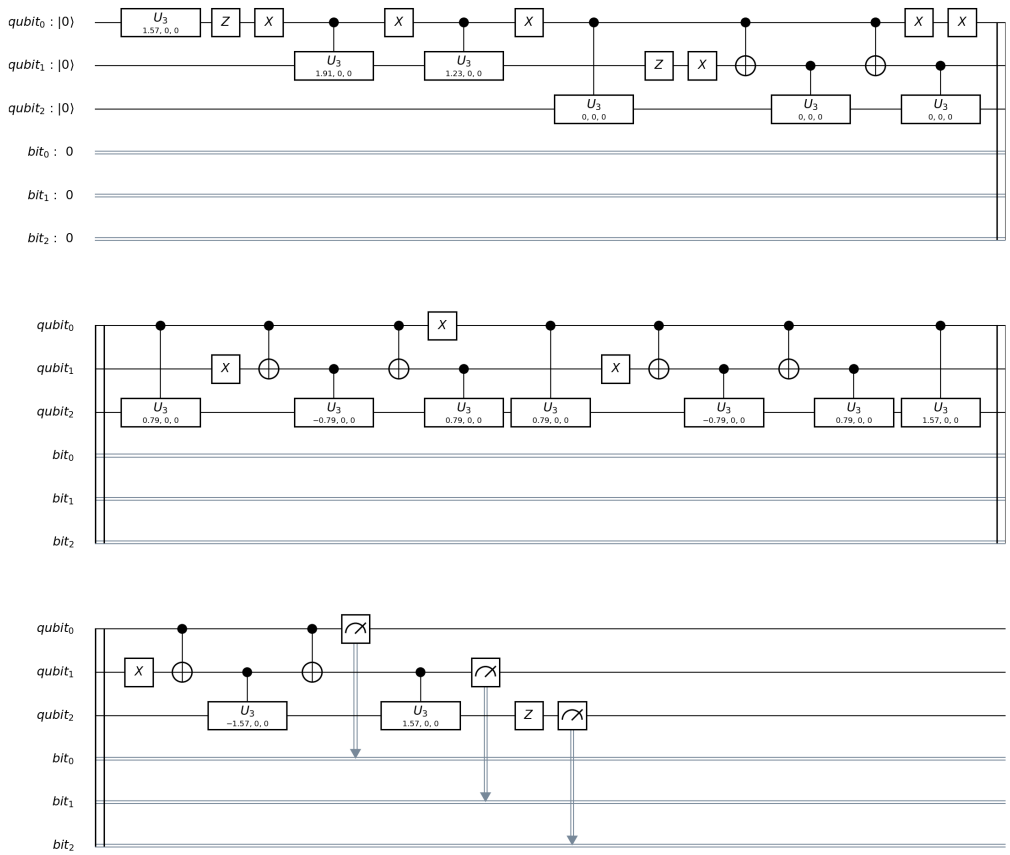

Figure 6: Quantum circuit to generate the transversal $T = \{000, 010, 011, 100, \mathbf{101}, 111\}$ of the $S_1 \mathrm{x} S_2$ group, as output by our program

5 Discussion and Outlook

We can conclude from the results that our code is able to implement a quantum circuit with *Qiskit* that successfully generates a desired transversal. All the test cases have given a positive outcome. The complexity of the code is given by the number of elements of the target state by the complexity of our control^n gates. For the time being, we have opted for a construction optimized for smaller systems, but that does not scale well with size. Nevertheless, we know how to implement other schemes optimized for larger number of qubits that can be readily integrated to our code.

It is worth noting that the specific transversal to be generated can largely affect the efficiency of the circuit for the case of non-fully symmetric circuits. A transversal that can be expressed as the tensor product of the transversals of the fully symmetric subgroups into which a group can be decomposed can be generated in a far more efficient way, as reduces the size of the system the parts of the circuit work with. The latter is very interesting for non-linear

complexities with the number of qubits, as is the case.

The presented code works for any input and matches all the goals of the project. However, there is always room for improvement. Next we outline some of the next features that could be added to our code to enhance its performance and flexibility:

- Implement controlled^n gates with quadratic (or even linear) with n complexity, like the ones proposed in [5]. The code would then be made to choose between the one currently implemented (optimal for small n) and the new ones
- When an on-demand transversal is input by the user, the current code generates the circuit for the full system. However, the input transversal could still be separable into the product of partial transversals, which would allow a more efficient quantum circuit. Hence, a new functionality could be implemented that, before generating the quantum circuit, decomposes the transversal as much as possible
- The current program can be very easily extended to generate non-uniform states (with any weight of the target superimposed states) with some small modifications in the functions that compute the angle for the unitary single-qubit controlled gate used. Hence, we would not just be limited to transversals, but could generate any state
- Some other minor improvements can be made. For instance, in some cases our circuit presents two X subsequent on the same qubit that have no other effect than cancelling each other. They are introduced on converting a general control into a full 1's control, but sometimes can be spared

References

[1] Gui-Lu Long, Yang Sun: "Efficient Scheme for Initializing a Quantum Register with an Arbitrary Superposed State", Physical Rev. A, Vol 64, 014303 (2001)

[2] Nicolas Borie: "Generating tuples of integers modulo the action of a permutation group and applications", 2012; arXiv:1211.6261.

[3] "IBM Qiskit Terra", accessed on 2019-11-01. [Online]. Available: https://qiskit.org/documentation/_autodoc/qiskit.html

[4] "IBM Qiskit Aer", accessed on 2019-12-01. [Online]. Available: https://qiskit.org/aer

[5] Barenco, Adriano, et al.: "Elementary gates for quantum computation." Physical review A 52.5 (1995): 3457.

[6] Nielsen, Michael A., and Isaac Chuang: "Quantum computation and quantum information." (2002): 184.

A Proof of the algorithm's robustness against undetermined unitary parameters

In this section we prove that an undetermined unitary gate is never activated and, thus, they can be ignored. Following up on the discussion around fig.(2), this means that the controlled states that activate the undetermined gate are always orthogonal to the circuit state at the moment of being tested.

In the following we denote the target collection of superimposed states by T and the states by $(i_1, i_2, ..., i_n)$. We also set the notation $(i_1, ..., i_{j-1}) \in T$ to indicate that such sub-string of the full $(i_1, ..., i_n)$ is present in at least one element of T.

We make the proof by induction starting from the following statement.

Induction Statement 1: Provided that $(i_1, ..., i_{j-1}) \in T$,

$$(i_1, ..., i_{j-1}, x_j) \notin T \quad \Rightarrow \quad (i_1, ..., i_{j-1}, x_j \oplus 1) \in T \tag{10}$$

- *Base case*: For $j = 1$ we have full freedom over all possible qubit states, so $(i_1, ..., i_{j-1}, x_j) \in T$ is trivially satisfied provided that $T \neq \emptyset$
- *Step case*: Assuming that Induction Statement 1 holds for an arbitrary step j, the condition $(i_1, ..., i_{j-1}) \in T$ guarantees that $\exists\ x_j$ such that $(i_1, ..., i_{j-1}, x_j, ...) \in T$

The proof of Induction Statement 1 is trivial and follows from the basic properties of binary arrays.

The above statement is equivalent to never activating an undetermined unitary, as we can see from the following relations with the numerator and denominator of eq.(4).

$$(i_1, ..., i_{j-1}, x_j) \notin T \quad \Leftrightarrow \quad \sum_{i_{j+1}}^{i_n} \left| a_{i_1, i_2, ..., i_{j-1}, x_j, i_{j+1}, ... i_n} \right|^2 = 0 \tag{11}$$

$$(i_1, ..., i_{j-1}, x_j) \in T \quad \Leftrightarrow \quad \sum_{i_{j+1}}^{i_n} \left| a_{i_1, i_2, ..., i_{j-1}, x_j, i_{j+1}, ... i_n} \right|^2 \neq 0 \tag{12}$$

Therefore, a new statement can be made as a direct consequence of the above:

Induction Statement 2: Provided that $(i_1, ..., i_{j-1}) \in T$:

$$\sum_{i_{j+1}}^{i_n} \left| a_{i_1, i_2, ..., i_{j-1}, x_j, i_{j+1}, ... i_n} \right|^2 = 0 \Leftrightarrow \sum_{i_{j+1}}^{i_n} \left| a_{i_1, i_2, ..., i_{j-1}, x_j \oplus 1, i_{j+1}, ... i_n} \right|^2 \neq 0 \tag{13}$$

Replacing this new statement in the prove by induction set above completes the proof that a $\frac{0}{0}$ case is never activated in the algorithm.

□

B Code Documentation

In the following we explain the main features of our code, and define the most relevant implemented functions. They follow the same structure than the submitted *Jupyter* notebook. The functions are group into purpose groups.

The last part of this section explains how to use the GUI and explains the metrics used to evaluate the degree of success of our quantum circuit on generating the expected transversal.

B.1 Generating the partition of G

The following functions generate the partition into a maximal collection of invariant subsets of G

`fullSymDecomp(n,start,end)`: Returns dictionary with FULLY symmetric group G partition (decimal values) into maximal subsets.

- `n`: total number of qubits
- `start`: position of the first element of the fully symmetric group (starting from 0)
- `end`: position of the last element of the fully symmetric group

  `combine(sys)`: Return dictionary with G partition (decimal values) into maximal subsets by combining the fully Symmetric blocks
- `sys`: list with the sizes of the permutation sub-groups

B.2 Angles for the controlled rotations

The definition of the angles for the initial and controlled rotations used in the quantum circuit to generate the transversal

`alpha1(T,n)`: Calculation of the initial rotation angle
INPUT

- `T`: list with the indices (converted to decimal) of the elements in the transversal
- `n`: total number of qubits in current fully symmetric block of the circuit

OUTPUT

- rotation angle α_1

`ControlledAlpha(T,n,j,control)`: Calculation of controlled α_j

INPUTS

- `T`: list of elements of the transversal
- `n`: total number of qubits in current fully symmetric block of the circuit

- `j`: target qubit index (starting from 0 up to n-1, reading the arrays from left to right)
- `control`: control qubits state in decimal form (calculated for the full qubit array, assuming all the rest are zeros, e.g. $|01XX\rangle \Rightarrow |0100\rangle \Rightarrow$ control = 4)

B.3 Quantum subroutines

- Controlled-n Ry gate implementation
- Convert a control from mix of zeros and ones to full ones, or unconvert full ones to desired control

`nControlledRy(start,target,theta,q,circ)`: Adds to the quantum circuit a rotation around 'y' gate controlled by any number of qubits, i.e. the rotation is performed if all the control qubits are 1. The control qubits are defined in block, e.g. from qubit 0 to qubit 4

INPUT

- `start`: first qubit used as control
- `target`: qubit over which the Ry gate is applied (therefore, the control is from qubit 'start' to 'target-1')
- `theta`: desired angle of rotation (in radians)
- `q`: quantum register with circuit's qubits
- `circ`: original circuit in where the nC-Ry gate is implemented

OUTPUT

- new circuit with the C^n-$\hat{R}$y(`theta`) gate added

B.4 Transversal quantum generator for the full symmetric group

The following function puts all together to output a circuit that generates a quantum transversal in uniform superposition

`FullSymQSuperposition(T,n,start,q,circ)`: Circuit generator to create a quantum transversal for the FULLY symmetric group

INPUT

- T: on demand fully symmetric group transversal

- n: total number of qubits in current fully symmetric block of the circuit
- start: first qubit of the fully symmetric block
- `q`: quantum register with circuit's qubits
- `circ`: original circuit in where the nC-Ry gate is implemented

OUTPUT

- Quantum circuit generating an on-demand fully symmetric group transversal

B.5 Interactive solver

- Introduce your symmetry group. If composed, separate blocks with 'x'. e.g. 3 for S3, and 2x1 for S2xS1
- Check whether you want to input a permutation (permuteT) or on-demand transversal (onDemandT) and input the corresponding information
- Choose the number of shots for the circuit simulation
- Choose the file name (or path/filename) for your output circuits (in '.tex' and '.png' format)

(Some errors are raised by the program if some of the inputs are wrong in some way. In case of facing such error, please skip the traceback part and go directly to the end where you will see: TypeError: "description of what went wrong")

OUTPUT

- Transversal and eventually the permuted transversal QASM simulations results given by the number of measurements for each superimposed state
- Result verifications
- Drawing of the circuit exported in .png and .tex format

B.5.1 Verification

The GUI outputs two verifications of the results. First it checks that there is a perfect match between the elements in the target transversal and the measurements results. Second it gives a metric to assess whether the dispersion of the results is the one expected statistically or we should have some concerns.

This metric tries to estimate what dispersion to expect around the expected number of counts. For this we take one of the output states and assume a Bernouilli between getting a count for that state with probability $p = 1/N$ and

not getting it with probability $1 - p = (N - 1)/N$. Then we calculate the standard deviation of the mean $\sqrt{p(1-p)/N}$. The value we compare to this one is the standard deviation of all the outcomes with the expected perfect equal superposition value. This last standard deviation is biased as it is not obtained from independent measurements, but it is good enough for the type of test we want to do.