Geometric Approach to Digital Quantum Information. Quantum Entanglement. –Senior Project–

Dan Andrei Iancu '04
Department of Electrical Engineering
and Computer Science
Yale University
Adviser: Professor Michel Devoret
APHY/PHY

April 20, 2004

Abstract

The purpose of the project was to attempt an interpretation of the nature of digital quantum information from a geometrical perspective. This lead to designing an algorithm for building the equivalents of Platonic solids in a 2^n dimensional Hilbert space, structures called **uniform Hilbertian polytopes**. The long-term goal was to better understand quantum entanglement, and possibly find a measure for its degree in a given state.

1 Introduction

The motivation behind the need to find new approaches for the study of quantum information has a lot to do with the nature of the Hilbert space $\mathcal{H}_n$ for n qubits. This is a complex, 2^n-dimensional vector space; any state in the space can be written as:

$$|\Psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle \tag{1}$$

where the state $|i\rangle$ can be expressed:

$$|i\rangle = \bigotimes_{b_j=b_{n-1}}^{b_0} b_j = b_{n-1} \otimes b_{n-2} \otimes \cdots \otimes b_1 \otimes b_0$$

and $b_{n-1}b_{n-2}\cdots b_1 b_0$ represents the binary notation for the value i ($i = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \cdots + b_{n-1} \cdot 2^{n-1}$). In equation (1), the complex coefficients α_i have to satisfy the normalization

condition, which imposes that:

$$\sum_{i=0}^{n-1} |\alpha_i|^2 = 1$$

In addition to this, the arbitrariness of the overall phase removes yet another constraint from the system. Hence, an n-qubit state can be parameterized by exactly $2^{2n} - 2$ real coefficients - still a very large number!

The continuous structure of the space, combined with the necessity of achieving reliable computation, make it very hard to design and implement circuits operating on qubits. When one thinks of its classical counterpart - the configuration space for n bits - one realizes that the discrete nature is an essential ingredient in being able to not only reliably distinguish, but also manipulate the states of the system. [i]

Therefore, when trying to design, implement, test and optimize a simple quantum computing circuit, it would be useful to first limit ourselves to a number of discrete states, which we can characterize and manipulate in an easier fashion. This simplification does seem to come at a high price, because we lose an element that seemed necessary for the long-sought exponential speedup in quantum algorithms: the continuous nature of H_n! But the situation is not all that disastrous! Discrete sets of states have already provided marvellous results in the field of quantum error correction codes [Got97] and it seems that limiting the number of states and the number of operations performable on those states does not lead to losing generality.

The solution for this type of geometric approach was initially proposed by Michel Devoret and Chad Rigetti [DevRig03] - it involves searching for sets of states (dubbed *Hilbertian polytopes*) that form structures equivalent to the Platonic solids. The purpose of the current project was to find a way to implement the search algorithm, which would allow a complete automation of the process and would provide the first step in designing a quantum compiler for the language of generalized $\pi/2$ rotations.

2 Theoretical framework

For the purposes of both consistency and coherence, I shall first present the theoretical framework developed in [DevRig03], and then introduce the algorithms that achieve the desired behavior.

[i] If our classical *bit* had several stable states - let alone a continuum of them! - then we would not only have to change all the circuits implementing it or operations on it, but we would be forced to rethink the very algebra that we use in analyzing and designing these circuits!

2.1 The 1-qubit scenario

It is very interesting to examine the case of 1-qubit, because we have a clear understanding of the attached geometric representation of the Hilbert space $\mathcal{H}_1$. The states to consider are sketched in **Figure 1**; they represent the 6 intersections of the Bloch sphere with the Ox, Oy and Oz axes. Let the set of these states be denoted by[ii] $\mathcal{D}_1$

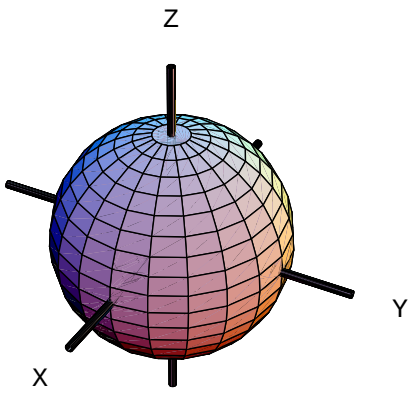

Figure 1: Important states on the Bloch sphere

$$\begin{aligned}
|+z\rangle &= |0\rangle \\
|-z\rangle &= |1\rangle \\
|+x\rangle &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \\
|-x\rangle &= \frac{|0\rangle - |1\rangle}{\sqrt{2}} \\
|+y\rangle &= \frac{|0\rangle + i\,|1\rangle}{\sqrt{2}} \\
|-y\rangle &= \frac{|0\rangle - i\,|1\rangle}{\sqrt{2}}
\end{aligned}$$

It can be noticed that these states construct an octahedron inscribed in the Bloch sphere,

[ii] $\mathcal{D}$ for $\mathcal{D}$iscrete

denoted $\mathfrak{h}_1$ and representing the Uniform Hilbertian Polytope for the 1-qubit Hilbert space.

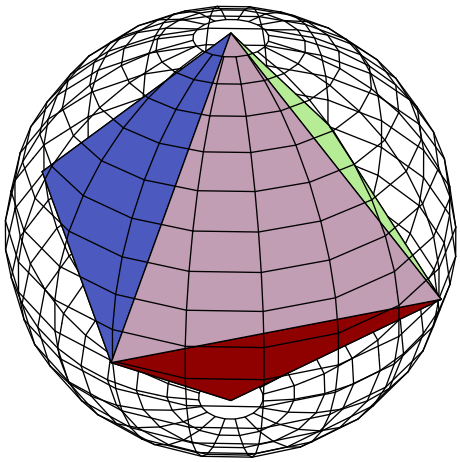

Figure 2: The Uniform Hilbertian Polytope of order 1 - octahedron

What is interesting is that starting with any of these states, all the others are accessible via a sequence of standard NMR pulses performing $\pi/2$ rotations. Therefore, in the spirit of stabilizer theory, it has been suggested [DevRig03] that instead of studying the set of states, it would be more instructive to focus on the transformations that leave $\mathfrak{h}_1$ invariant.

The beautiful part about this theory is that the geometrical description provides a very clear intuition of what these transformations might be. Instead of thinking about the qubits and their states, we can simply discuss the geometrical features of the octahedron. Therefore, the transformations that leave the set $\mathcal{D}_1$ invariant are the same transformations that leave the octahedron invariant: any sequence of $\pi/2$ rotations around Ox, Oy or Oz.

2.2 Identification and description of the uniform Hilbertian polytope $\mathcal{H}_n$

Similar to the 1-qubit scenario, the discrete set $\mathcal{D}_n$ is formed by the eigenstates of the generalized Pauli matrices. These matrices are members of the Pauli group $\mathcal{G}_n$, which contains all the tensor products of n Pauli matrices with multiplicative factors $\pm 1, \pm i$.

$$\mathcal{G}_n = \left\{ \bigotimes_{k=0}^{n-1} \Sigma_k, \quad \Sigma_k \in \{I, X, Y, Z\} \otimes \{\pm 1, \pm i\} \right\}$$

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

However, since for $n > 1$, the Pauli matrices have degenerate eigenvalues, additional explanations are required. The trick that has been employed in [DevRig03] was to lift the degeneracy by choosing the states as the common eigenvectors of sets of mutually commuting generalized Pauli matrices.[iii]

Let $\mathcal{S}_n$ be the set of 4^n linearly independent generalized Pauli matrices on n qubits Σ_j satisfying the following properties:

$$\begin{aligned} \Sigma_j^\dagger &= \Sigma_j \\ \Sigma_j^2 &= I \\ Tr(\Sigma_j^\dagger \Sigma_k) &= 2^n \delta_{jk} \end{aligned}$$

It can be seen that $\mathcal{S}_n \subset \mathcal{G}_n$. [iv] In this set, we can distinguish all the maximal subsets s_n^a of mutually commuting elements from $\mathcal{S}_n$.

$\mathfrak{h}_\mathfrak{n}$, as described in [DevRig03], has the following properties:

1. It contains all the states $|b_{n-1}b_{n-2}\ldots b_1 b_0\rangle$ corresponding to classical bit configurations
2. Each state ("vertex") in $\mathfrak{h}_\mathfrak{n}$ is geometrically equivalent to all the others
3. There is the concept of the *distance* between the states, defined as $d_{jk} = 2 \cdot \arccos \langle \Psi_j | \Psi_k \rangle$
4. It is the largest set satisfying 1-3

Therefore, in order to meet all the above criteria, [DevRig03] concludes that we have to adopt the following construction rule:

Each vertex of $\mathfrak{h}_\mathfrak{n}$ is a common eigenvector of a maximal subset $s_n^a \subset \mathcal{S}_n$ of 2^n mutually-commuting generalized Pauli matrices on n qubits. That is, if Σ_j is a generalized Pauli matrix on n qubits belonging to the subset s_n^a, $|\Psi_j\rangle$ is an n-qubit state vector, and λ_j is an eigenvalue of Σ_j belonging to the vector $|\Psi_j\rangle$,

$$|\Psi_j\rangle \in \mathfrak{h}_\mathfrak{n} \Longleftrightarrow \Sigma_j |\Psi_j\rangle = \lambda_j |\Psi_j\rangle, \quad \forall \Sigma_j \in s_n^a$$

Some of the conclusions that result from this are [DevRig03]:

[iii] The theory is very similar to the stabilizer theory. There is, however, a slight difference between the sets (of matrices) that we are considering and the stabilizers. The latter are proper groups (i.e. closed under matrix multiplication), whereas this restriction is not imposed on the former.

[iv] In addition, $\mathcal{G}_n$ allows each n-fold tensor product to contain the multiplicative factors $\pm 1, \pm i$, which are suppressed here.

1. For each s_n^a with $2^n - 1$ elements different from the identity, 2^n different discrete states separated by a distance of π can be found. Each state corresponds to a unique pattern of $\lambda_j = \pm 1$.

2. Any two states of $\mathfrak{h}_\mathfrak{n}$ are equivalent - i.e. there exists a transformation that can associate the $2^n - 1$ eigenvectors of 2 neighboring groups s_n^a and s_n^b.

3. The set $\mathcal{S}_n$ contains exactly $s = \prod_{k=0}^{n-1} (2^{n-k} + 1)$ maximal mutually commuting subsets $s_n^a \subset \mathcal{S}_n$. Since each subset contributes exactly 2^n (distinct) eigenvectors to $\mathfrak{h}_\mathfrak{n}$, then the uniform Hilbertian polytope on n qubits has exactly

$$V_n = 2^n \cdot \prod_{k=0}^{n-1} (2^{n-k} + 1)$$

vertices (states).

2.3 The 1-qubit case

By reanalyzing the 1-qubit scenario, we get a very good intuition of what these results mean. The standard 2×2 Pauli matrices are:

$$\sigma_w = \sigma_0 = I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \sigma_x = \sigma_2 = X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
$$\sigma_y = \sigma_3 = Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \sigma_z = \sigma_1 = Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

The commutation relations between these matrices are:

$$\sigma_x \sigma_y = -\sigma_y \sigma_x = i\sigma_z$$
$$\sigma_y \sigma_z = -\sigma_z \sigma_y = i\sigma_x$$
$$\sigma_z \sigma_x = -\sigma_x \sigma_z = i\sigma_y$$
$$\sigma_k \sigma_w = \sigma_k, \forall k \in \{w, x, y, z\}$$
$$\sigma_x^2 = \sigma_y^2 = \sigma_z^2 = \sigma_w$$

The maximal sets s_n^a of mutually commuting matrices are represented in **Table** 1. It can be seen that each of these sets contributes exactly 2 states to $\mathfrak{h}_1$. To obtain these states, we need to simultaneously diagonalize all the matrices in each subset s_n^a, one subset at a time. However, since in this simple example, each s_n^a only contains 2 matrices, one of which is the identity, the process of simultaneous diagonalization becomes banal: simply take the eigenvectors of σ_x,σ_y,and σ_z!

Set	Matrices	States that the set is contributing
s_1^1	$\{\sigma_w, \sigma_z\}$	$\{\|+z\rangle = \|0\rangle\,,\ \|-z\rangle = \|1\rangle\}$
s_1^2	$\{\sigma_w, \sigma_x\}$	$\{\|+x\rangle = \frac{\|0\rangle+\|1\rangle}{\sqrt{2}},\ \|-x\rangle = \frac{\|0\rangle-\|1\rangle}{\sqrt{2}}\}$
s_1^3	$\{\sigma_w, \sigma_y\}$	$\{\|+y\rangle = \frac{\|0\rangle+i\|1\rangle}{\sqrt{2}},\ \|-y\rangle = \frac{\|0\rangle-i\|1\rangle}{\sqrt{2}}\}$

Table 1: Commuting sets and the vertices that they contribute to $\mathfrak{h}_1$

In [DevRig03], it had been suggested that a more general method would be to diagonalize the 'seeds' of $\mathfrak{h}_1$. These seeds are:

$$\begin{aligned}\Sigma_{01}^1 &= \tfrac{1}{\sqrt{2}}(\sigma_0 + i\sigma_1)\\ \Sigma_{02}^2 &= \tfrac{1}{\sqrt{2}}(\sigma_0 + i\sigma_2)\\ \Sigma_{03}^3 &= \tfrac{1}{\sqrt{2}}(\sigma_0 + i\sigma_3)\end{aligned}$$

2.4 The n-qubit case

The natural question that emerges is how to extend this simple algorithm for an arbitrary number of qubits - n. The same source [DevRig03] suggests that for $\mathfrak{h}_2$, we can simply form a mixed linear combination of the first two non-identity matrices in each subset s_2^a.

$$\Sigma_{kl}^a = \frac{1}{\sqrt{2}}(\Sigma_k + i\Sigma_l);\quad \{\Sigma_k, \Sigma_l\} \subset s_2^a;\quad \sigma_w^{\otimes 2} \notin \{\Sigma_k, \Sigma_l\}$$

Upon diagonalizing these combinations, we obtain all the vertices of the polytope $\mathfrak{h}_2$. However, the problem is that the method is not immediately generalizable to n qubits. Simply forming linear combinations of the first 2 non-identity matrices would not yield enough information to generate all the eigenvectors (some of the eigenvalues are going to be degenerate).

The solution would be to reconsider the type of linear combination that we are using. The first thing that comes to mind is to take a linear combination of all the matrices in the set, with coefficients in a geometric progression.[v]

$$s_n^a = \{\Sigma_0, \Sigma_1, ..., \Sigma_{2^n-1}\}$$

$$\Sigma_{all} = \sum_{i=0}^{2^n-1} c^i \cdot \Sigma_i,\quad c = ct.$$

The solution does work - the degeneracy of the eigenvalues is lifted, and when we diagonalize Σ_{all}, we obtain all the eigenvectors that contribute to the polytope $\mathfrak{h}_n$.

However, this construction contains a lot of redundant information. Intuitively, n non-identity matrices properly chosen from the set s_n^a should be enough: a linear combination of

[v]The reason for this geometric progression is to maintain a certain distance between the matrices. Including the identity among them only shifts the spectrum with a constant amount, without changing the eigenvectors, which are what we really care about.

these would represent the proper seed for the $\mathfrak{h}_\mathfrak{n}$. The problem is that not *any* n matrices work!

The answer comes - yet again - from the theory of stabilizers. If we allow our set to accommodate $\pm 1, \pm i$ factors, we obtain the corresponding Pauli group.

$$g_n^a = s_n^a \otimes \{\pm 1, \pm i\}$$

We can then apply the theory of stabilizers to this group: its entire structure is described by its *generators*. Therefore, if we choose a linear combination of generators for the set s_n^a, then this would contain sufficient information and would also be minimal (i.e. minimum number of terms).

This result gives us a very simple rule for forming the linear combination, once we have the maximally commuting subset s_n^a: simply choose n non-identity matrices that do not form a sub-sub set in s_n^a (that is to say, if we allowed $\pm 1, \pm i$, the resulting sub-subset formed by the n matrices would not be closed under matrix multiplication).

3 Implementation

With this theoretical framework, we have set out to implement the algorithms that would be able to automatize the process of finding the vertices of $\mathfrak{h}_\mathfrak{n}$. Only a brief overview of the algorithm is included, with some considerations of performance; the code will be pasted in the end, for perusal.

The first task was to generate the maximally commuting subsets s_n^a for a given n (the number of qubits). Since we need all the possible solutions, the only algorithm that can be employed is a version of optimized backtracking, building one solution from the previous one. This was implemented in C language, in order to achieve a reasonable speed of execution.[vi]

Afterwards, for each maximally commuting subset s_b^a we formed a linear combination of matrices, according to the formula above (including only the generators as terms). The resulting matrix contains all the information needed from the respective set: upon diagonalization, it yields all 2^n vertices that s_n^a is contributing to the Hilbertian polytope.

For testing purposes, these last two stages were implemented in Mathematica, both due to its flexibility with tensor product operations, but also due to its ability to extract eigenvalues and eigenvectors in a symbolic form. Implementation in C would be much faster, but several problems would first have to be overcome:

- The lack of the imaginary symbol i

[vi] Translating the code into assembler would surely speed up the execution even further, but the task is cumbersome, due to the dynamical allocation of the memory for all the matrices, and the multiple procedure calls.

- The lack of a diagonalization operation
- The lack of symbolic evaluation of expressions

Simply building data-structures for complex numbers and operating on them would surely work, but the amount of memory allocated to each subset would then be huge, and the algorithm will become unstable for a very small n.

3.1 Performance of the algorithms

The backtracking-based implementation of the search for commuting subsets has proven to be quite costly. **Table** 2 represents the average execution times obtained when running our algorithm for different numbers of qubits.[vii] The exponential characteristic can be easily noticed. Furthermore, the demands on the memory are colossal. A test run for 15 qubits has completely blown the system (segmentation fault reported while allocating memory for the matrices). It is interesting that considering even such a small number of discrete states can put such tremendous computational pressure!

Average runtime	**Number of qubits**
69.7 μs	2
53.61 s	3
1069 s	4
1.5 days	5

Table 2: Average execution times for different configurations

Several improvements that could be brought would be to ease the requirements of the backtracking algorithm, asking it only to determine the matrices that are needed for our linear combination: the n generators, as opposed to the whole subset s_n^a.

One more thing that should be said about the C implementation: one of the key operations in the algorithm was to decide whether 2 generalized Pauli matrices commute. Each was specified as an array of coefficients from the set $\{0,1,2,3\}$[viii]. For instance:

$$\Sigma_1 = [102103] \quad when \quad \Sigma_1 = Z \otimes I \otimes X \otimes Z \otimes I \otimes Y$$
$$\Sigma_2 = [013131] \quad when \quad \Sigma_2 = I \otimes Z \otimes Y \otimes Z \otimes Y \otimes Z$$

In deciding whether they commute, the algorithm simply computed the coefficient:

$$g = \prod_{i=0}^{n-1} g_i, \quad g_i = \begin{cases} +1, & [\Sigma_1[i], \Sigma_2[i]] = 0 \\ -1, & \{\Sigma_1[i], \Sigma_2[i]\} = 0 \end{cases}$$

[vii] The tests have been run on the *rattlesnake, scorpion, viper, hare* and *leopard* machines in the "ZOO"-3rd floor AKW. Each is a dual-processor, Pentium IV machine, with 1GB of RAM.

[viii] According to the Gottesman notation, 0 was used for I, 1 for Z, 2 for X and 3 for Y.

A result $g = 1$ would mean that Σ_1 and Σ_2 commute. This solution was chosen for reasons of simplicity, as well as to ease memory and computational requirements.

A much more elegant solution would have been to represent the two matrices using the *check matrix*[ChuNie00]. In this language, our two matrices would become:

$$\begin{aligned} \Sigma_1 &\implies r_1 = [001001|\ 100101] \\ \Sigma_2 &\implies r_2 = [001010|\ 011111] \end{aligned}$$

Then, the test for commutation simply becomes

$$r_1 \cdot \Lambda \cdot r_2^T = 0, \Lambda = \begin{pmatrix} 0 & I \\ I & 0 \end{pmatrix}$$

Although mathematically more interesting and convenient, this would have been more expensive to implement in terms of memory (all generalized Pauli matrices would have required arrays of size $2n$), and the operations would have been of roughly the same complexity.

However, one big advantage of the check matrix is that it only needs bits (i.e. 0 and 1) - so we could use binary number representations, instead of arrays, to store the n needed generators. Furthermore, the operation of matrix multiplication (with the exception of $\pm 1, \pm i$ factors) becomes extremely simple in the check-matrix notation: an addition of the 2 check-matrix representations modulo 2!

With these considerations, the check-matrix representation might be much easier to implement on a vector machine, and - with a proper chip coding the transition operations - might be the key feature in building a quantum compiler.

4 Conclusion

A quantum computer that operates only on states of the uniform Hilbertian polytope is not universal. In fact, even if we allow $\pi/4$ rotations[ix], we still do not harness all the true of quantum information processing. According to the Gottesman-Knill theorem, any quantum computer using only the $H, S, C-NOT$ and T gates, with preparations in the computational basis and measurements of observables in the Pauli group, can be simulated on a classical computer![x]

The correct way to interpret this result is that digital quantum computation can be simulated on a classical machine. Genuine quantum algorithms, which achieve the long-sought exponential speedup (like Shor's factoring), seem to require the continuous structure

[ix] Allowing $\pi/4$ rotations means achieving the set of Hadamard, phase, C-NOT and $\pi/8$ gates, which is enough to simulate any unitary transformation to an arbitrary precision.

[x] The way to achieve this is simply to track the generators of the stabilizer, as the different operations are performed in the computation.

of the Hilbert space $\mathcal{H}_n$.

However, even with this in mind, digital quantum computing on the Hilbertian polytope still has a lot to offer! The entire plethora of error-correction is achievable by operating only on this discrete subset of states. Furthermore, since much of the processing power of quantum computation seems to lie hidden in the phenomenon of entanglement, we do not lose much with a digital framework: the polytope still contains entangled states, and - in addition - provides a better chance to measure and describe the degree of entanglement.

Ultimately, beyond the theoretical considerations, a digital framework for quantum computing simplifies the existence of the experimentalist: from designing and implementing gates, all the way to testing for errors and adding control.

5 Acknowledgement

The end of the article contains a list with some of the books and articles that I have used at various stages throughout the project. However, by far the most helpful have been the discussions with Professor Michel Devoret and Chad Rigetti, who have been incredibly kind and supportive throughout the entire endeavor, and to whom I owe a debt of immense gratitude.

6 Code

```
/* PROGRAM THAT GENERATES THE MAXIMAL MUTUALLY COMMUTING */
/* SUBSETS OF GENERALIZED PAULI MATRICES ON N QUBITS */
/* Dan Andrei Iancu ;          Class of 2004 */
/* --Senior Project-- */
/* Adviser: Professor Michel Devoret */

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<sys/time.h>

/* GOTTESMAN'S BINARY DIGIT NOTATION */
/* sigmaW = (identity) =sigma0    */
/* sigmaZ = sigma1 */
/* sigmaX = sigma2 */
/* sigmaY = sigma3 */
```

```
/* A generalized Pauli matrix on N qubits = tensor product */
/* of N Pauli matrices; the notation for a generalized Pauli */
/* matrix is:                              */
/*  Sigma_number, where "number" is a concatenation of N strings,*/
/* each representing one Pauli matrix - 2 bits for each          */

void generateMatrices(int N, long NumMatrices, int **Space);
/* function that generates all the generalized Sigma Pauli matrices */
/* for N qubits and stores the result in the pre-allocated "Space" */
/* "NumMatrices" is the number of such matrices (already computed as 4^N) */

long generateGottesmanString(int N, int *SigmaMatrix);
/* generates the Gottesman string code corresponding to the generalized  */
/* N-qubit Pauli polymatrix specified by the array SigmMatrix            */

void generateMaximalSets(int N, long NumMatrices, int **PoliMatrices);
/* generates all the maximal, mutually commuting sets of polymatrices */

void printTensorProductForm(int solutionSize, int *solution, int
N, int NumMatrices, int **Polymatrices);
/* we have a solution of size 'solutionSize', stored in 'solution'  */
/* using the values in solution, we print the tensor product corresponding */
/* to each value based on the matrices stored in 'PolyMatrices' */

void printGottForm(int solutionSize, int *solution);
/* prints the solution by writing the Gottesman value corresponding to each  */
/* matrix    */

void printLetterForm(int solutionSize, int *solution, int N, int
NumMatrices, int **Polymatrices);
/* just like the 'printTensorProductForm', just that it prints letters, not  */
numbers   */

void printNeededCommSubgroups(int solutionSize, int *solution, int
N, int NumMatrices, int **Polymatrices);
/* this prints only as many matrices as are needed to compute the combination */
/* that, when diagonalized, yields all the eigenvectors that characterize our */
/* particular commuting set; simplest way to achieve this is to pick 2 poly- */
```

```
/* matrices at random, compute their product and then include another N-2    */
/* polymatrices, making sure that we do not include the product of the first */
/* two. All matrices are printed in tensor product form -- just as the */
/* procedure printTensorProductForm() does */

void init(long index, int *Stack);
/* initializes the position "index" in the stack "Stack" with a value        */

int validate(long index, int *stack, int N, int **PoliMatrices);
/* validate the new element inserted at position "index" in the "stack"      */
/* "N" = number of qubits; "PoliMatrices" = matrix w/ the Pauli Polimatrices */

void dumpMatrices(int N, long NumMatrices, int **PoliMatrices);
/* print all the Pauli Polimatrices in the array, with the associated */
/* Gottesman value */

int main(int argc, char **argv) {
  int n;
  int **PauliMatrices;  /* the matrix containing the generalized Pauli */
                        /* matrices, one per row, given as numbers */
                        /* from 0 to 3, using Gottesman's convention  */
  int i;
  long noM;             /* the number of generalized Sigma matrices  */

  if (argc!=2) {
    fprintf(stderr,"Error in number of input arguments.\nPlease retry...\n");
    exit(1);
  }
  sscanf(argv[1],"%d",&n);      /* read the number of qubits     */

  /* the matrix of Pauli matrices contains 4^N matrices          */
  noM=(long)pow(4,n);
  PauliMatrices=(int**)malloc(noM*sizeof(int*));
  for(i=0;i<noM;i++)
    PauliMatrices[i]=(int*)malloc((n+1)*sizeof(int));

  /* generate all the 4^N matrices using an iterative algorithm  */
  generateMatrices(n,noM,PauliMatrices);
```

```
  /* generate all the maximal mutually commuting sets  */
  generateMaximalSets(n,noM,PauliMatrices);
  return 0;
}

void generateMatrices(int n, long noM, int **matrix) {
  /* the first position in the each row of the **matrix will be the       */
  /* Gottesman number corresp. to the respective matrix - this makes      */
  /* comparing the matrices a lot simpler!                                */

  int *index;             /* array with the indices corresponding         */
                          /* to each Pauli matrix in the tensor product   */
  int i,j,k,c,aux;
  index=(int*)malloc(n*sizeof(int));

  for(i=0;i<=n;i++)
    index[i]=0;

  /* simulate succesive additions of 1 in the least significant position  */
  /* and adjust the carry at each step; stop when carry is 0 or running   */
  /* index=-1 */
  c=0;
  for(k=0;c==0;) {
    /* storing the solution */

    /* as mentioned, first position is the Gottesman value */
    matrix[k][0]=generateGottesmanString(n,index);
    for(j=0;j<n;j++)
      matrix[k][j+1]=index[j];
    k++;
    /* generating new solution */
    c=1;
    for(j=n-1;(j>=0)&&(c!=0);j--) {
      aux=index[j]+c;
      index[j]=aux%4;
      c=aux/4;
    }
```

```
  }
}

long generateGottesmanString(int n, int *sigma) {
  long value;       /* value of the code  */
  long powerof2;    /* running power of 2 */
  int i;

  for(i=n-1,value=0,powerof2=1;i>=0;i--) {
    value=value+(sigma[i]%2)*powerof2+(sigma[i]/2)*powerof2*2;
    powerof2*=4;
  }
  return value;
}

void printTensorProductForm(int k, int *stack, int n, int noM, int
**matrices) {
  /* solution is an array with numbers, each denoting polymatrices  */
  /* each number is converted into a tensor product of 'n' Sigma    */
  /* Pauli matrices, based on the Gottesman binary representation   */
  /* 'k' - number of polymatrices contained in the stack            */
  /* 'stack' - Gottesman numbers corresponding to each matrix       */
  /* 'n' - number of qubits                                         */
  /* 'noM' - total number of matrices (all contained in 'matrices') */
  /* 'matrices' - containes all the generalized polymatrices        */

  int i,j;
  for(i=0;i<k;i++) {    /* for each polymatrix in the solution      */
    /* 'j' goes across all the pauli matrices that give the product */
    /*    printf("("); */
    for(j=1;j<=n;j++)
      printf("%d ",matrices[stack[i]][j]);
    /* printf(") "); */
  }
  printf("\n");
}

void printGottForm(int k, int *stack) {
```

```
  /* prints the array of numbers corresponding to a subgroup        */
  /* i.e. does NOT convert to tensor product form                   */
  int i;
  for(i=0;i<k;i++)
    printf("%d ",stack[i]);
  printf("\n");
}

void printLetterForm(int k, int *stack, int n, int noM, int
**matrices) {

  int i,j;
  for(i=0;i<k;i++) {    /* for each polymatrix in the solution      */
    /* 'j' goes across all the pauli matrices that give the product */
    for(j=1;j<=n;j++)
      switch(matrices[stack[i]][j]) {
      case 0: printf("w "); break;
      case 1: printf("z "); break;
      case 2: printf("x "); break;
      case 3: printf("y "); break;
      }
    printf(", ");
  }
  printf("\n");
}

void printNeededCommSubgroups(int k, int *stack, int n, int noM,
int **matrices) {
  int *productOfFirstTwo;    /* the product of the first two polymatrices */
  int i,j,sum,np;

  /* some simple rules to compute product : (might change later) */
  /* if m1(i)=m2(i) -> (m1*m2)(i)=0                    */
  /* else if m1(i)+m2(i)<=3 -> (m1*m2)(i)= m1(i)+m2(i) */
  /* else (m1*m2)(i)=6-m1(i)-m2(i)                     */

  productOfFirstTwo=(int*)malloc(n*sizeof(int));
```

```
  /* this computes the product of the first two NON-  */
  for(i=1;i<=n;i++)        /* IDENTITY Poly-matrices  */
    if(matrices[stack[1]][i]==matrices[stack[2]][i])
      productOfFirstTwo[i-1]=0;
    else {
      sum=matrices[stack[1]][i]+matrices[stack[2]][i];
      if(sum<=3)
        productOfFirstTwo[i-1]=sum;
      else
        productOfFirstTwo[i-1]=6-sum;
    }

  /* compute the Gottesman string associated with the */
  /* polymatrix 'productOfFirstTwo' & store in 'sum'  */

  sum=generateGottesmanString(n,productOfFirstTwo);

  /* counter 'np' keeps track of how many matrices we  */
  /* have already printed, so that we do not cross n   */
  for(i=1,np=0;(i<k)&&(np<n);i++)
    if(matrices[stack[i]][0]!=sum) {  /* avoid the product of first 2*/
      np++;
      for(j=1;j<=n;j++)
        printf("%d ",matrices[stack[i]][j]);
    }
  printf("\n");
}

void generateMaximalSets(int n, long noM, int **matrices) {
  /* Each maximally commuting subset has 2^N polymatrices          */
  /* Currently using a backtracking algorithm, because we          */
  /* have to find all the solutions                                */

  /* the index 0 will be excluded from all solutions; hence, a     */
  /* solution is reached when 2^N-1 non-identity matrices are      */
  /* found. To this, we add the identity matrix.                   */

  /* st = stack that contains indices corresponding to Pauli       */
```

```
/*      generalized matrices; each index specifies a row in the  */
/*      matrix "**matrices"                                       */
/* k = index in the stack; when a solution is generated, k will  */
/*     be equal to 2^N-2, and all 2^N-1 matrices in "st" commute */
/* solSize = 2^N-1 (computed only once)                          */


long solSize,k,numSol;    /* solSize = 2^N (size of the solution)     */
int *st;                  /* the stack in which we store the solution */
int ok;                   /* numSol = the number of total solutions   */

solSize=(long)pow(2,n);
st=(int*)malloc((solSize)*sizeof(int));
numSol=0;
st[0]=0;
k=1;
st[1]=0;

printf("%d\n",n);     /* print the number of qubits on the first line */
while(k>=1) {
  ok=0;
  while((st[k]<noM-1)&&(!ok)) {
    st[k]++;                          /* try value one unit larger */
    ok=validate(k,st,n,matrices);     /* validate value i          */
  }
  if(!ok) {
    k--;
  } else if (k==solSize-1) {

    /* we have a solution                    */
    /* which we print in tensor product form */

    /* printLetterForm(k+1,st,n,noM,matrices); */
    /* printGottForm(k+1,st); */
    printNeededCommSubgroups(k+1,st,n,noM,matrices);
    /* printTensorProductForm(k+1,st,n,noM,matrices); */
    numSol++;
```

```
    } else {
      k++;
      st[k]=st[k-1];
    }
  }
  printf("%ld solutions total.\n",numSol);
}

int validate(long k, int *st, int n, int **PoliMatrices) {

  /* validate a new element inserted in the stack                        */

  /* Conditions that must be met:                                        */
  /* 1) the matrix at the row st[k] in **PoliMatrices must commute       */
  /*    with all the matrices at rows st[0]..st[k-1];                    */
  /* 2) the matrix at row st[k] must be different from all the           */
  /*    other matrices                                                   */

  int ok,i,j,g;
  /* ok = 1 when the new matrix commutes with all the previous  */
  /*        matrices, and is different from them; 0 otherwise   */

  ok=1;
  if(k>0)
    if(st[k]<=st[k-1]) /* cannot allow combinations to repeat; hence all */
      return 0;        /* the matrix numbers must be increasing in stack */
  for(i=0,g=1;i<k;i++) {
    if(PoliMatrices[st[i]][0]==PoliMatrices[st[k]][0]) {
      ok=0;    /* the new matrix is equal to another one in the set      */
      break;
    }
    else {
      /* we have to check if the matrices given by rows # st[i] and st[k]*/
      /* commute                                                          */
      for(j=1;j<=n;j++)
        if((PoliMatrices[st[i]][j]==PoliMatrices[st[k]][j])||
             ((PoliMatrices[st[i]][j]*PoliMatrices[st[k]][j])==0))
          ;
```

```
        else
          g*=-1;
      if(g==-1) {
        ok=0;
        break;
      }
    }
  }
  return ok;
}

void dumpMatrices(int N, long noM, int **PauliMatrices) {
  /* printing all Pauli matrices with associated Gottesman value */
  int i,j;

  for(i=0;i<noM;i++) {
    for(j=0;j<=N;j++)
      printf("%d ",PauliMatrices[i][j]);
    printf("\n");
  }
}
```

References

[DevRig03] Chad Rigetti, Rémi Mosseri and Michel Devoret, *"Geometric Approach to Digital Quantum Information"*, arXiv:quant-ph/0312196v1, 2003

[Got97] Daniel Gottesman, *Stabilizer Codes and Error Correction.*, Ph.D. thesis, Caltech, 1997, http://xxx.lanl.gov/abs/quant-ph/?9705052

[ChuNie00] Isaac Chuang, Michael Nielsen, *Quantum Computation and Quantum Information*, Cambridge University Press, 2000

[Got98] Daniel Gottesman, *The Heisenberg Interpretation of Quantum Computers*, 1998 International Conference on Group Theoretic Physics, arXiv:quant-ph/9807006