Snapshots of modern mathematics
from Oberwolfach

№ 12/2018

Number theory in quantum computing

Sebastian Schönnenbeck

Algorithms are mathematical procedures developed to solve a problem. When encoded on a computer, algorithms must be "translated" to a series of simple steps, each of which the computer knows how to do. This task is relatively easy to do on a classical computer and we witness the benefits of this success in our everyday life. Quantum mechanics, the physical theory of the very small, promises to enable completely novel architectures of our machines, which will provide specific tasks with higher computing power. Translating and implementing algorithms on quantum computers is hard. However, we will show that solutions to this problem can be found and yield surprising applications to number theory.

1 Computing

In this section we briefly introduce the notion of computing in general. We then review classical computing and introduce the reader to quantum computing. Computing is a vast and fascinating topic which we do not attempt to cover or introduce here. We point the interested reader to [3].

1.1 Classical computing

A classical computer (by which we here mean a Turing machine, see [5]) is a machine that employs the principles of classical physics (in particular the principles of *electrodynamics*)[1] to perform operations on numbers (e.g. adding or multiplying two numbers) much faster than any human can hope to do.

A classical computer stores information in *bits*. A bit is always in one of two possible states, it is either "on" or "off" and we think of "on" as a 1 and "off" as a 0. If our computer can store information using n bits, it can thus be in any one of 2^n possible states (we have two possible choices for each bit). A modern personal computer, for example, usually has at least a few billion bits of memory, also known as *Giga-bytes*.[2]

In general, we want our computer not only to store information but also to process it and to do computations with it. To that end, we think of the two possible states of a single bit as the two elements of the finite field $\mathbb{F}_2$. An introduction to the concept of field can be found in the box below.

We can then think of the states of our computer, meaning all possible ways to encode information, as elements of the n-dimensional $\mathbb{F}_2$-vector space which we denote by $\mathbb{F}_2^n$. A simple way to visualize an element of this set, or a state of the computer, is the well known “string of bits”. For example, two elements x_1 and x_2 of $\mathbb{F}_2^9$, which are both nine-dimensional strings of bits, are $x_1 = 110011010$ and $x_2 = 100000110$. Can you see that there are $2^9 = 512$ such different strings, or elements, of $\mathbb{F}_2^9$?

Many of the computations that we can do on the states (strings of bits) of the computer can be represented by $n \times n$-matrices with elements of $\mathbb{F}_2$ (matrices with 0 or 1 as entries) that operate on a string of bits and map it to another string of bits. An operation can, in general, map any string of bits, to any other string of bits. For example, there will be a 9×9 matrix that maps the string x_1 to the string x_2 of the example above.

A simple example is the following. Imagine we have two strings of bits $x_1 = 100$ and $x_2 = 001$ and we wish to map x_1 to x_2 using a 3×3 matrix. Once one knows how to apply matrices to vectors, it is easy to check that one matrix $\boldsymbol{M}$ that does the job is

$$\boldsymbol{M} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

[1] Electrodynamics studies all phenomena that involve charged particles, materials and the electromagnetic field.

[2] Information capacity of computers is commonly measured in bytes, which correspond to eight-bits, see https://en.wikipedia.org/wiki/Byte.

In fact, $x_2 = \boldsymbol{M}\, x_1$.

Fields: a brief introduction

Definition
Formally, a *field* is a set $\mathbb{F}$ together with two operations, the operation $+$ called addition and the operation $\cdot$ called multiplication.

Properties of the operations
The operations $+$ and $\cdot$ are required to satisfy the following properties, referred to as *field axioms*. In the sequel, a, b, and c are arbitrary elements of $\mathbb{F}$.

i) Associativity of addition and multiplication: $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
ii) Commutativity of addition and multiplication: $a+b = b+a$ and $a \cdot b = b \cdot a$.
iii) Additive and multiplicative identity: there exist two different elements 0 and 1 in $\mathbb{F}$ such that $a + 0 = a$ and $a \cdot 1 = a$.
iv) Additive inverses: for every a in $\mathbb{F}$, there exists an element in $\mathbb{F}$, denoted $-a$, called additive inverse of a, such that $a + (-a) = 0$.
v) Multiplicative inverses: for every $a \neq 0$ in $\mathbb{F}$, there exists an element in $\mathbb{F}$, denoted by a^{-1}, $1/a$, or $\frac{1}{a}$, called the multiplicative inverse of a, such that $a \cdot a^{-1} = 1$.
vi) Distributivity of multiplication over addition: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

Example: $\mathbb{R}$
As an example, we remind the reader that the set of real numbers $\mathbb{R}$ is a field together with the standard multiplication and addition operators. Each property above can be independently checked. We leave this as a simple exercise for the reader.

1.2 Quantum computing

A quantum computer (by which we here mean a quantum Turing machine, see [4]) is a computer that employs properties of small quantum systems, described by the theory of quantum mechanics, to perform computations.[3] These devices can handle specific problems (for example finding the prime factors of a given – typically large – number as often needed in encryption) much better than classical computers, since they behave fundamentally different from their classical counterparts.

At the moment there is a lot of exciting work going on with a lot of companies competing in building bigger and better versions of these devices. Most recently

[3] Whereas a classical computer is based in classical physics.

IBM announced their success in building the largest quantum computer capable of being programmed to perform arbitrary tasks yet.[4] In addition, the American company D-Wave is already promoting their quantum computer which is (by their own accord) two orders of magnitude more powerful than IBM's,[5] see Figure 1. However, it should be noted that these devices are single purpose-built and thus do not quite count as quantum computers in the sense of this snapshot. Finally, it is not only the private sector who invests in this field. The European Union financed a quantum technology flagship project with a billion dollars in 2016[6] and similar projects are underway in the US, Canada, and China.

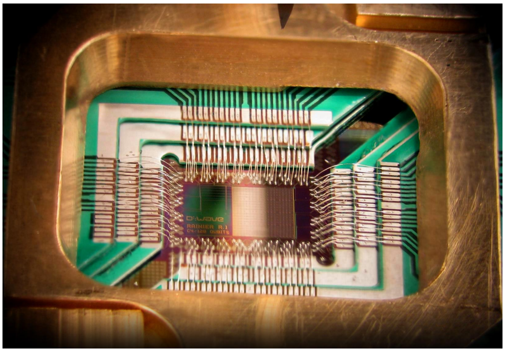

Figure 1: Photograph of a chip constructed by D-Wave Systems Inc., mounted and wire-bonded in a sample holder. The D-Wave processor is designed to use 128 superconducting logic elements that exhibit controllable and tunable coupling to perform operations.

In this snapshot we will not explain the theoretical background of a quantum computer. Neither will we consider the multitude of problems that arise when scientists actually try to build one.[7] Instead we want to think about how we would practically work with a quantum computer if we had one and how we would handle the problem of telling a quantum computer what to do.

We have already argued that classical computers operate using bits of information. In a quantum computer the situation is a little more complicated. These devices no longer store information in bits, but rather use *qubits*. A qubit does not always have to be either 0 or 1 but can be in a "combination" of the

[4] See, https://www.ibm.com/blogs/research/2017/11/the-future-is-quantum/.

[5] This statement can be found at https://www.dwavesys.com/d-wave-two-system.

[6] More information to be found at https://www.nature.com/news/europe-plans-giant-billion-euro-quantum-technologies-project-1.19796.

[7] An introduction to both of these topics can be found in [3].

two called a *superposition.* We can think of such a superposition by employing vectors. The superposition of two vectors is a (complex) linear combination of the two vectors. In the same way, we model the superposition of the possible states 0 and 1 of a classical bit as a qubit, that is, an element of the vector space $\mathbb{C}^2$.

We can denote two elements of $\mathbb{C}^2$ using the "bra-ket" notation $|0\rangle$ and $|1\rangle$.[8] If we choose $|0\rangle$ and $|1\rangle$ conveniently, we obtain a generic superposition $|\psi\rangle$ of the two by $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$, where a, b are complex numbers. Therefore, $|\psi\rangle \in \mathbb{C}^2$ is an arbitrary element of $\mathbb{C}^2$. This notation has a simple and intuitive pictorial representation. We can imagine that $|0\rangle$ and $|1\rangle$ are the unit vectors in a two-dimensional plane, where $|0\rangle$ is aligned along the x axis and $|1\rangle$ along the y axis. If a and b where real numbers, then $|\psi\rangle$ would just represent an arbitrary vector in the plane, with (signed) length a along the x axis and (signed) length b along the y axis. Its total length L would be given by the standard Pythagoras rule $L = \sqrt{a^2 + b^2}$. In the present case, since a and b are complex, the pictorial representation needs some work of imagination, but the extension should be conceptualizable with some training.

Now, if our quantum computer operates with n qubits, its possible states are the complex linear combinations of the states of a classical computer with n usual bits. For example, a particular state can be $|\psi_{part.}\rangle = |x_1\rangle + \frac{i}{3}\,|x_2\rangle$, where each "ket" is labeled by a string of n classical bits and $x_1 = 00001001011\ldots$, $x_2 = 1010010101\ldots$ This leads us to think of these states as elements of the 2^n dimensional $\mathbb{C}$-vector space $\mathbb{C}^{2^n}$. In general, we write the generic element $|\psi\rangle$ as

$$|\psi\rangle = \sum_m a_{x_m}\,|x_m\rangle,$$

where the sum is over all possible strings x_m of n bits and the coefficients a_{x_m} are complex numbers. In our previous example, we had $a_{x_1} = 1$ and $a_{x_2} = i/3$, while all other a_{x_m} coefficients were zero.

Computations with these states become matrix multiplication with elements from $\mathbb{C}^{2^n \times 2^n}$. This means that a quantum computation algorithm can be encoded in $2^n \times 2^n$ matrix g with complex elements g_{mp}. The algorithm acts on the state $|\psi\rangle$ mapping it to a state $|\psi'\rangle = \sum_m a'_{x_m}\,|x_m\rangle$ with elements $a'_{x_m} = \sum_p g_{mp}\,a_{x_p}$. This compact notation allows us to write $|\psi'\rangle = g\,|\psi\rangle$. Finally, we note that the very nature of a quantum computer restricts us to matrices g that preserve the length of each vector, so-called unitary matrices. We denote the set of all unitary matrices by $\mathrm{U}_{2^n}(\mathbb{C})$. This requirement comes as a consequence of the probabilistic interpretation of the vectors $|\Psi\rangle$[9].

[8] The bra-ket notation was introduced by the great physicist and Nobel Prize laureate P. A. M. Dirac.

[9] Footnote added We require such a vector to have unit length in order to be able to

2 Exact synthesis

In this snapshot we focus on one specific problem that we face when working with a quantum computer. Imagine that we want to run an algorithm on our (quantum) device, an operation that is implemented by applying a specific unitary matrix g to the vector $|\psi\rangle$ that represents the current state. Our quantum computer, however, does not a priori know how to apply this operation (read, matrix). Instead it only knows how to perform a handful of basic operations that we call $g_1, g_2, g_3, \ldots$. I have added a small analogy here. This problem is equivalent to learning, say, to perform a piece on the flute. It is virtually impossible to learn the whole piece without knowing the basic moves and notes. The student (computer) can perform basic notes (operations). The basic notes (operations) in the right sequence form together the whole piece (operation g). The composition (list of basic operations) tells the student (computer) how to practically perform (implement) the piece (operation g).

Therefore, we first need to tell the quantum computer how to build up the matrix we want to apply from those basic operations it is familiar with. This is known as the *exact synthesis* problem and we formally phrase it as follows:

Problem 1 (Exact synthesis) *Let $G \subset \mathrm{U}_{2^n}(\mathbb{C})$ be the set of* fundamental operations *we can perform on our n-qubit quantum computer (called the* gate set *of the quantum computer) and let $g \in \mathrm{U}_{2^n}(\mathbb{C})$ be the matrix we actually want to apply.*[10] *Then, exact synthesis is the task of finding fundamental operations $g_1, g_2, \ldots, g_r \in G$ such that*

$$g = g_1 \cdot \ldots \cdot g_r.$$

In other words we ask for a series of fundamental operations that we can apply one after the other to achieve the same result as applying g at once.

This can maybe be better understood with our flute student example. Given a set of basic notes (gates in G) and a flute piece (element g), we ask which is the sequence, or composition, $g_1 \cdot \ldots \cdot g_r$ of notes (the exact synthesis) that allows us to play the piece. This analogy is quite useful to picture in our mind the mathematical passages involved. We emphasize that it should not be taken literally as an exact representation of what happens in an algorithm for a quantum computer.

interpret the coefficients a_{x_m} as probabilities. In fact, to require that the vector $|\Psi\rangle$ has unit length is completely equivalent to the requirement that $\sum_m |a_{x_m}|^2 = 1$, which is the standard condition for probabilities of an event. For a discussion on the probabilistic nature of a wave function see [6]. A Unitary matrix $\boldsymbol{M}$ is defined by the condition $\boldsymbol{M}\boldsymbol{M}^\dagger = \mathbb{1}$ where $\mathbb{1}$ is the identity matrix. Unitary transformations preserve the unit "length" of the vector and, therefore, the total probability.

[10] The set $\mathrm{U}_{2^n}(\mathbb{C})$ is the set of $2^n \times 2^n$ unitary matrices.

We now proceed to explain this problem and its implications with an illustrative example.

Example 1 *Let us assume our* 1*-qubit quantum computer can perform the basic operations*

$$g_1 := \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \text{ and } g_2 := \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}.$$

This means that $G = \{g_1, g_2\}$*.*

Our aim is to teach the computer how to apply the operation (matrix)

$$g := \begin{pmatrix} -i & 0 \\ 0 & i \end{pmatrix}.$$

It is easy to check that one answer to this exact synthesis problem is

$$g = g_1 \cdot g_1 \cdot g_2.$$

However, another answer exists[11] *and it reads*

$$g = g_1 \cdot g_2 \cdot g_1 \cdot g_1 \cdot g_1.$$

Our Example 1 shows that there is usually more than one answer to an exact synthesis problem. However, not every answer is of the same "quality". In fact, we all know that it costs energy to perform any physical operation, such as lifting a weight or operating a computer at home. The energy that the computer consumes is predominantly spent for computational tasks, such as the fundamental operations it needs to perform to produce an image on the screen, or to save data on the hard drive. It goes without saying that different operations might cost a different amount of energy. Therefore, we usually aim to find an answer to our exact synthesis problem that minimizes the energy we have to use. Concretely, in Example 1 we would choose the first answer over the second one, since it will for sure employ less energy (The operations in the first answer all appear in the second one as well. However, the second answer requires the computer to perform two extra steps. Therefore, more energy).

3 The 1-qubit case

Let us now see how we can solve our problem in the simplest case imaginable, namely on a quantum computer that only has one qubit as Example 1 above. While computations on a classical computer with one bit are not hard to do (we can only switch the one bit on or off a couple of times), on a quantum computer it is already not easy to handle one single qubit.

[11] Did you find other examples?

3.1 Clifford and T-gates

The first question we have to ask ourselves is which is the set of fundamental operations that our quantum computer should be able to perform. We cannot arbitrarily put together this set of operations, since we want to choose elementary operations that can be easily physically built in a real quantum computer. Therefore, let us take the opposite approach and ask the computer engineers what fundamental operations they can actually build, and then proceed to find a way to perform exact synthesis with these operations.

It would lead too far to explain the details here, so we will just explain the outcome. A good reference would be [2].

The set of gates that we end up with by following this program is called the Clifford+T gate set and consists of the following gates:

- The two *basic* gates are the two *Clifford* gates (or operations) c_1 and c_2 which read
$$c_1 = \begin{pmatrix} 0 & i \\ 1 & 0 \end{pmatrix} \text{ and } c_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} -i & i \\ 1 & 1 \end{pmatrix}.$$
- It turns out that the two Clifford gates alone are not enough to build a useful quantum computer. In fact, if you had a lot of time on your hands and started writing down all possible combinations of c_1 and c_2 (or, if you had less time, you could ask a computer to do it for you) you would find that there are only 192 different matrices that you can find this way. This is not enough and one way to improve this is to add one more gate to what your quantum computer can do which is usually called T-gate and reads
$$T = \frac{1}{\sqrt{2}} \begin{pmatrix} 1+i & 0 \\ 0 & \sqrt{2} \end{pmatrix}.$$
The names T-gate and Clifford-gates have historical reasons and moreover there is no reason one should expect T to behave fundamentally different from c_1 and c_2 (in fact, $T \cdot T$ is a Clifford-gate). However, it is possible to show that it is not possible to write down every possible combination of c_1, c_2 and T, since it turns out that there are infinite such different combinations and therefore our quantum computer can now perform infinitely many different operations.

We will call *Clifford gates* those operations that our quantum computer can perform by only using c_1 and c_2 (possibly multiple times).

With these gates our one-qubit quantum computer can perform infinitely many different operations. We would of course like to know whether we can tell by just “looking” at an operation if our quantum computer is capable of implementing it. In other words we would like to tell from looking at a matrix if the exact synthesis problem has a solution or not. Luckily there is a positive answer to this question [1]. This is guaranteed by the following theorem.

Theorem 1 *Let g be a unitary 2×2 matrix. A quantum computer equipped with the Clifford+T gate set can perform g if and only if all entries of g are of the form*

$$2^n \cdot (a_1 + a_2\sqrt{2} + (b_1 + b_2\sqrt{2}) \cdot i) \tag{1}$$

with integers a_1, a_2, b_1, b_2 and n.

Let us call Clifford+T the set of all matrices g that can be performed by a quantum computer equipped with the Clifford+T gate set. Theorem 1 guarantees that an operation is in the C+T set precisely if it has the specific form detailed in (1), it can be obtained by applying the Clifford+T gates in an appropriate fashion. In the following, we proceed to explain how to solve exactly the problem of finding the correct sequence of the required fundamental gates.

3.2 Single qubit exact synthesis in Clifford+T

Here we solve exact synthesis problems with respect to the Clifford+T gate set. To this end, we first have to agree how we want to judge the quality of a solution to the exact synthesis problem (we already saw in a previous example that there might be plenty of different solutions). It can be shown that in experiments it is far cheaper to apply a Clifford operator (that is, c_1, c_2 or any sequence of finite length containing only c_1 and c_2) than to apply T. For this reason, we will judge the quality of a solution to an exact synthesis problem by the number of times T appears. This is called the T-count of the solution and we aim to find a solution with minimal T-count. For example, the word $c_1 \cdot T \cdot c_2 \cdot c_1 \cdot c_2$ has T-count one, while the word $T \cdot c_1 \cdot c_2 \cdot T \cdot c_2 \cdot T$ has T-count three.

Now notice that it is pretty easy to do exact synthesis for purely Clifford operators. We already mentioned that there are only finitely many of these (192 to be precise) so we can just write down a list of all strings of operators c_1 and c_2 that will realize one of these 192 Clifford gates and use it as a reference whenever we need to implement a Clifford gate. From now on, we refer to such strings of basic gates as "words". For example, a word can be $c_1 \cdot c_1 \cdot c_2 \cdot c_1 \cdot c_2$ or $c_1 \cdot c_1$.

If we wanted to do the same for operators that also include T we could start by writing down all words of T-count one, then all words of T-count two and so on. This means, that a word would have exactly one T gate, two T gates and so on. This approach has two major drawbacks. On the one hand, we would never be able to write down a complete list of such words, since there are operations with arbitrarily high T-count. On the other hand, even if we settled for only writing down all operations up to a given T-count (say maybe up to 100 or 1000), we would quickly encounter the problem of being able to store all these words, that is, the limit of our capacity to store the list in a "physical" device, such as a notebook or a computer hard-drive. This would occur since the number of operations with given T-count grows rapidly. For these two reasons we need a to study the problem at hand using a completely different approach.

4 Number theory comes to the rescue

This is the point where number theory comes to our aid. Number theory does not have an obvious connection to quantum computing. However, this field of mathematics is concerned with sets of matrices that look like those we can build out of the Clifford+T gate set.[12]

The actual algorithm for performing exact synthesis already in the 1-qubit case requires a background in number theory that we cannot sensibly provide here. However, it is still possible to present the idea behind the algorithm by explaining how it works in a very similar, but much "nicer", situation, where we do not have to deal with any imaginary numbers or square-roots of 2.

To this end, the analogue of the Clifford-gates g will be all matrices that we can obtain as combinations of z_1 and z_2, which have the expression

$$z_1 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \quad z_2 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}. \text{ Here } g = \begin{pmatrix} a & b \\ c & d \end{pmatrix}. \tag{2}$$

It turns out that these matrices are easily recognized, since any matrix g can be written as a word in the gates z_1 and z_2 if and only if all of a, b, c and d are integers and the so-called determinant $\det(g) = ac - bd$ is equal to 1.

As an analogue for the T-gate gate we take the following three matrices:

$$t_1 = \begin{pmatrix} 0 & \frac{1}{2} \\ -2 & 0 \end{pmatrix}, t_2 = \begin{pmatrix} 0 & 2 \\ -\frac{1}{2} & 0 \end{pmatrix}, \text{ and } t_3 = \begin{pmatrix} \frac{1}{2} & -\frac{5}{2} \\ \frac{1}{2} & -\frac{1}{2}. \end{pmatrix} \tag{3}$$

It turns out that also the set of matrices that can be written as words in z_1, z_2 and t_1, t_2, t_3 is easily recognized and looks very similar to what we saw for Clifford+T in Theorem 1. In fact the matrix g can be written as such a word if and only if all of a, b, c and d are integers divided by a power of 2 (so of the form $\frac{m}{2^n}$ with integers m and n) and the determinant $\det(g) = ac - bd$ is still equal to 1.

[12] For an introduction to number theory aimed at beginners, see for example *A friendly introduction to number theory* by Joseph H. Silverman

As for Clifford+T we now ask ourselves how one might actually find such a word for a given matrix g (ideally using few occurrences of t_1, t_2 and t_3 as these are our analogues of the T-gate). If all entries are integers it is relatively easy to write g as a word in z_1 and z_2. Explanations on how to do this can be found all over the internet,[13] so we will not explain this step here.

If, on the other hand, at least one of the entries is not an integer, it turns out that exactly one of t_1, t_2 and t_3 has the property that the maximal power of 2 that appears in the denominator of the entries of $t_1^{-1}g$ (or $t_2^{-1}g$ or $t_3^{-1}g$) is actually smaller than the maximal power of 2 that appears as the denominator of an entry of g. This observation (which is not at all obvious) leads to the following relatively straight-forward algorithm.

Algorithm

We want to write g as a word in z_1, z_2 and t_1, t_2, t_3. We can proceed as follows:

---

1. If all entries of g are integers write g as a word in z_1 and z_2.
2. If not, find the one matrix among t_1, t_2, t_3 such that the maximal power of 2 appearing in a denominator of the entries of $t_1^{-1}g$ (or $t_2^{-1}g$ or $t_3^{-1}g$) is smaller than for g itself.
3. Continue in step 1 with $t_1^{-1}g$ (or $t_2^{-1}g$ or $t_3^{-1}g$, respectively).

After finitely many steps the maximal power of 2 that actually appears has to be $2^0 = 1$ (since it gets smaller in every step) so all the entries are now integers and we can write the resulting matrix as a word in z_1 and z_2. But the resulting matrix is just g multiplied with a couple of t_1^{-1}, t_2^{-1} and t_3^{-1} so if we bring these to the other side of the equation by multiplying with t_1, t_2 and t_3 we have written g exactly in the way we wanted.

4.1 Exact synthesis: an example

At this point we explain the abstract description above with an illustrative example.

Example 2 *Say we want to apply the above algorithm to*

$$g = \begin{pmatrix} 0 & -4 \\ \frac{1}{4} & -\frac{1}{2} \end{pmatrix}.$$

First of all we note that all entries are integers divided by powers of 2 *and that* $\det(g) = 0 \cdot (-\frac{1}{2}) - \frac{1}{4} \cdot (-4) = 1$ *so we see that this should actually be possible.*

[13] See for example: http://www.math.uconn.edu/~kconrad/blurbs/grouptheory/SL(2,Z).pdf

Also g has entries that are not already integers so we cannot (yet) write g as a word in z_1 and z_2.

Hence we are in step 2 of the algorithm. The maximal power of 2 in the denominator of an entry of g is 4 and we check

$$t_1^{-1}g = \begin{pmatrix} -\frac{1}{8} & \frac{1}{4} \\ \frac{1}{2} & -9 \end{pmatrix}, t_2^{-1}g = \begin{pmatrix} -\frac{1}{2} & 1 \\ \frac{1}{8} & -\frac{9}{4} \end{pmatrix}, \text{ and } t_3^{-1}g = \begin{pmatrix} \frac{1}{2} & 1 \\ 0 & 2 \end{pmatrix}.$$

The maximal power of 2 in the denominator of $t_3^{-1}g$ is 2 (which is clearly less than 4) so we continue with $t_3^{-1}g$. This matrix still has an entry that is not an integer so we are again in step 2 of the algorithm and compute:

$$t_1^{-1}t_3^{-1}g = \begin{pmatrix} 0 & -1 \\ 1 & 2 \end{pmatrix}, t_2^{-1}t_3^{-1}g = \begin{pmatrix} 0 & -4 \\ \frac{1}{4} & \frac{1}{2} \end{pmatrix}, \text{ and } t_3^{-1}t_3^{-1}g = \begin{pmatrix} -\frac{1}{4} & \frac{9}{2} \\ -\frac{1}{4} & \frac{1}{2} \end{pmatrix}.$$

Now the entries of $t_1^{-1}t_3^{-1}g$ are actually integers and (applying the algorithm we looked up) we write

$$t_1^{-1}t_3^{-1}g = z_1z_2z_2.$$

If we multiply this equation from the left first by t_1 and then by t_3 we obtain

$$g = t_3t_1z_1z_2z_2$$

so we have actually succeeded in writing the matrix g as a word in z_1, z_2 and t_1, t_2, t_3 and in each step we only had to check which denominators appear in the entries of the matrix.

Since the maximal power of 2 appearing in an entry of the given matrix is lowered by a factor of 2 in each step of the algorithm we actually know from the beginning how many steps the algorithm will take. Namely, if this maximal power is 2^n, we will need to take n steps before all entries are integers. Moreover, the algorithm automatically uses as few instances of t_1, t_2, t_3 as possible which was another one of our goals. We want to conclude this section by noting that the same is true for the completely analogous algorithm that solves the actual exact synthesis problem for Clifford+T.

5 The multi-qubit case

In the section above we saw which principle allows us to decompose a 1-qubit matrix into a suitable combination of basic Clifford gates and T-gates. Even better, given that the T-gates are “expensive” to perform, we (essentially) saw how to achieve the desired combination with the minimal number of T-gates involved.

So far, we have discussed the single qubit case alone. However, a quantum computer that has only a single qubit as a resource is not of much use for our everyday purposes. Indeed, we would actually like to be able to solve interesting problems that require the use of a (much) higher number of qubits. Following the same logic as before, the set of gates that we would like to use will still be an appropriate Clifford+T set of gates. In particular, it will be a generalization of the one we used in the 1-qubit case to the n-qubit one.

The whole scheme presented above to characterize the matrices of the 1-qubit case, see Theorem 1, can be extended in a straightforward fashion to the n-qubit case. Therefore, it is still quite easy to tell which matrices we can implement on our quantum computer that operates using n-qubit registers. However, for two or more qubits the algorithm for exact synthesis sadly does not generalize as nicely. In particular, if we try to just use its naive generalization it can happen that we run into an impasse or get stuck in an infinite loop.

While a suitable generalization can still be used to help us solve the exact synthesis problem, it is significantly harder to do so for more than one qubit. For this reason, finding solutions with minimal T-count for n qubits is one of the topics of modern research.

Acknowledgements

Sebastian Schönnenbeck thanks the DFG collaborative research center TRR 195 for financial support.

Image credits

Fig. 1 "DWave 128chip.jpg". Author: D-Wave Systems, Inc. Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons, https://en.wikipedia.org/wiki/Quantum_computing#/media/File:DWave_128chip.jpg, visited on June 19, 2018.

References

[1] Brett Giles and Peter Selinger, *Exact synthesis of multiqubit Clifford+T circuits*, Phys. Rev. A **87** (2013), 032332.

[2] Michael Nielsen and Isaac Chuang, *Quantum computation and quantum information*, Cambridge University Press, 2002.

[3] Wikipedia, *Quantum computing*, https://en.wikipedia.org/wiki/Quantum_computing, Accessed: 2017-11-04.

[4] ———, *Quantum turing machine*, https://en.wikipedia.org/wiki/Quantum_Turing_machine, Accessed: 2018-06-03.

[5] ———, *Turing machine*, https://en.wikipedia.org/wiki/Turing_machine, Accessed: 2018-06-03.

[6] ———, *Wave function*, https://en.wikipedia.org/wiki/Wave_function, Accessed: 2017-11-08.

Sebastian Schönnenbeck *is a postdoctoral researcher at RWTH Aachen University.*

Mathematical subjects
Algebra and Number Theory

Connections to other fields
Computer Science, Engineering and Technology, Physics

DOI
10.14760/SNAP-2018-012-EN