Decompiling quantum algorithms through sonification

Joseph T. Iosue
Massachusetts Institute of Technology, Cambridge, Massachusetts USA
(Dated: March 9, 2019)

We present the idea of using sonification as a tool to build abstractions from quantum algorithms, and discuss how this could be used to develop new quantum algorithms. We propose a simple mapping in order to create a sonic representation of a quantum algorithm, and developed a small Python package *qSonify* [1] to aid in the study using this mapping.

I. INTRODUCTION

Quantum music has been discussed as a possible future art form [2], giving listeners an acoustic experience that could not be realized classically. Creating music with quantum computers has been explored as both an educational tool as well as a means to spark interest into quantum computing in those outside of the field [3]. There has also been work done on using Grover's algorithm to speed up algorithmic rule-based music composition [4]. These endeavors illustrate a quantum approach to music, and motivate the search for a music technology approach to quantum computing.

Sonification is a tool to represent data in an auditory form. The central task of sonification is the find a mapping from data of interest to sound in such a way that we can more easily recognize patterns or interesting features of the data. This is analogous to the familiar task of finding an optimal visual mapping of data, most commonly via plots and graphs. For many data sets, a visual mapping is sufficient; however, auditory mappings have been shown to be more effective in some cases for representing data in a manner that is useful to researchers, and sometimes auditory patterns are easier to recognize than visual patterns. We refer to [5] for a review of sonification and some of its use cases.

We present the idea of sonifying quantum algorithms in order to find useful abstractions. A quantum algorithm which is written in terms of an elementary gate set can be represented in many different ways, with different rotation angles, orderings, etc. But in order to visually recognize patterns in the algorithm and/or understand the function of the algorithm, it must be written in terms of larger unitaries which have convenient meaning. As an example, many algorithms use the Quantum Fourier Transform and its inverse. The Fourier transform's function is well understood, and when included in circuit diagrams is written as a single block transformation even though its actual implementation on quantum hardware contains a combination of elementary gates depending on the hardware. Another example of when building abstractions could be useful is when trying to scale machine learned quantum algorithms. In [6], the authors use classical machine learning techniques to learn the algorithm to compute state overlap. In [7], the authors compile algorithms with machine learning while using calls to the quantum processor to speed up the training. Some methods of learning a quantum algorithm result in a non-optimal structure or simply yield unitary matrices which represent the desired transformation. But if we cannot conceptually understand the structure or function of the circuit and instead consider it almost as an oracle, then scaling the algorithm to systems with more qubits is nontrivial. However, if we can find ways to represent the learned algorithms in terms of conceptually understood transformations, then we can hopefully see a general structure and determine how to scale that structure to larger systems.

Decompiling a quantum algorithm is the process of taking an algorithm which is not represented in a convenient form, and abstracting a more conceptual structure from it. We discuss one possible method of decompiling quantum algorithms by sonifying them and listening for patterns and themes.

II. CONCEPT

The central idea of the concept to be presented is shown in the iPython notebook in the *qSonify* GitHub repository [1], and there are relevant sound files in the `outputs` folder.

Let G be a quantum gate sequence, then

- $S(G)$ is the audio result of sonifying G, and
- $E(G)$ is a gate sequence which is the result of embedding G inside another gate sequence.

Consider that we have two quantum circuits represented by gate sequences G_1 and G_2 respectively. Assume that we have access to the function S (in particular, we design it), but not to the function E. If we are given a circuit C and are told that C is either $E(G_1)$ or $E(G_2)$, how can we determine which it is? The idea is to listen to $S(C)$, $S(G_1)$, and $S(G_2)$ and see if we can hear common patterns. Given a good choice of S, one can hear distinct patterns and differences between $S(G_1)$ and $S(G_2)$ that remain even when the gate sequences are embedded, thus helping us to determine whether C contains G_1 or G_2.

The results shown in the notebook are for very small, low-depth circuits, and the embedding performed is also quite small. We find the procedure to be, as one might expect, much less effective for larger circuits, which of course are the circuits of interest. We believe that the

MIDI mapping we have chosen (see Section IV) fails to capture important aspects of quantum algorithms. The central problem in sonification is to design S to be a useful mappings from an input to sound. This paper is meant to present and motivate the use of sonification for quantum algorithms, but a better mapping must be found before this idea may be practically useful.

One can see how this concept could potentially aid in the construction of black-box unitaries, and in the following section we discuss how it could be used to develop new quantum algorithms.

III. DEVELOPING NEW ALGORITHMS

To describe the method, we will step through a theoretical procedure. We would like the emphasize that we have not performed a procedure of this form due to the weaknesses mentioned in Section II. Nonetheless, we discuss the method since it may be useful once a better sonification mapping has been devised, and hope that it motivates research in the area.

Consider that we desire to create an algorithm which maps some inputs to known outputs. We could consider running an optimization routine to learn the parameters of a gate ansatz that performs the desired transformation; however, this is only feasible for small circuits with few qubits. Moreover, if we learn the parameters such that the ansatz performs the transformation for a small circuit, then the gate sequence looks seemingly random and impossible to generalize. The goal is to take such a gate sequence and decompile to build abstractions such that the algorithm is human-readable and hopefully straightforward to generalize to more qubits and larger circuits.

Let us assume that we have a set I of $n \leq 16$ inputs and a set $\mathcal{O}$ of n outputs, where each element of these sets is a pure state in a 16-dimensional Hilbert space (four qubits). Let $\left|\psi^i_{\text{in}}\right\rangle$ and $\left|\psi^i_{\text{out}}\right\rangle$ represent the i^{th} element in I and $\mathcal{O}$ respectively. First we fix a parameterized ansatz circuit structure to the form of Figure 1 called $A(\boldsymbol{\alpha})$. We then run an optimization routine over the parameters $\boldsymbol{\alpha}$ attempting to find $\boldsymbol{\alpha}^*$, the parameter vector which implements the desired transformation. One choice for the optimization could be

$$\boldsymbol{\alpha}^* = \arg\min_{\boldsymbol{\alpha}} \left(n - \sum_{i=1}^{n} \left| \left\langle \psi^i_{\text{out}} \right| A(\boldsymbol{\alpha}) \left| \psi^i_{\text{in}} \right\rangle \right|^2 \right), \qquad (1)$$

where $A(\boldsymbol{\alpha}^*)$ exactly implements the transformation if and only if the term in parentheses is zero. Note that the ansatz $A(\boldsymbol{\alpha})$ cannot implement an arbitrary four qubit unitary transformation, however finding an algorithm which approximately implements the transformation may be sufficient for our method.

After finding $\boldsymbol{\alpha}^*$, we have the desired circuit for the four qubit implementation, but there is little that can be learned from it. The circuit is a collection of single qubit rotations with a few CNOT gates; there is no straightforward way to generalize the algorithm to more qubits (assuming that the input I and output $\mathcal{O}$ data can be generalized to more qubits – for example, $I = \{|x\rangle\}$ and $\mathcal{O} = \{|x+3 \mod 16\rangle\}$ for $x \in \{0, ..., 15\}$ has a clear generalization from four qubits to any number of qubits). We attempt to do this by sonifying $A(\boldsymbol{\alpha}^*)$ and comparing to a reference *toolbox* (see Section V). By listening to all (as well as sections) of $A(\boldsymbol{\alpha}^*)$, we can hopefully hear some subroutines from the toolbox embedded within $A(\boldsymbol{\alpha}^*)$. Using the knowledge that the transformation contains these well-known routines, we can attempt to decompose the matrix of $A(\boldsymbol{\alpha}^*)$ into matrices of understood subroutines. If this can be done, then the generalization of the quantum algorithm to more qubits is simple since the function of each subroutine is known on any number of qubits.

$\|000\rangle$	$\leftrightarrow$	no notes played
$\|001\rangle$	$\leftrightarrow$	the note E
$\|010\rangle$	$\leftrightarrow$	the note D
$\|011\rangle$	$\leftrightarrow$	the chord DE
$\|100\rangle$	$\leftrightarrow$	the note C
$\|101\rangle$	$\leftrightarrow$	the chord CE
$\|110\rangle$	$\leftrightarrow$	the chord CD
$\|111\rangle$	$\leftrightarrow$	the chord CDE

TABLE I. The relationship between computational basis states and notes/chords for a three qubit example using the Fermionic mapping described in [2] and IV.

IV. MAPPING ALGORITHMS TO SOUND

The most important aspect of sonification is finding a map from the data of interest to useful sound. In the qSonify GitHub repository we take a simple approach of using Musical Instrument Digital Interface (MIDI) to characterize our sound. Thus, aspects of the algorithm get mapped to discrete musical notes rather than a continuous spectrum of sound. A similar method of mapping between basis states in the computational basis and notes is given in [2], where it is suggested that each qubit can represent whether or not a note is included in a chord. Consider, for example, three qubits, where the first is mapped to middle C on a piano, the second is mapped to the D above, and the third to the E above. Then we assume the mapping in Table I.

To sonify a quantum algorithm, we use the following procedure. Consider an algorithm Q on n qubits. We first run Q on the state $|t_0\rangle \equiv |0\rangle^{\otimes n}$ and measure the result in the computational basis. Let the result of the measurement be $|t_1\rangle$. We then prepare $|t_1\rangle$, run Q, and measure in the computational basis, giving the measure-

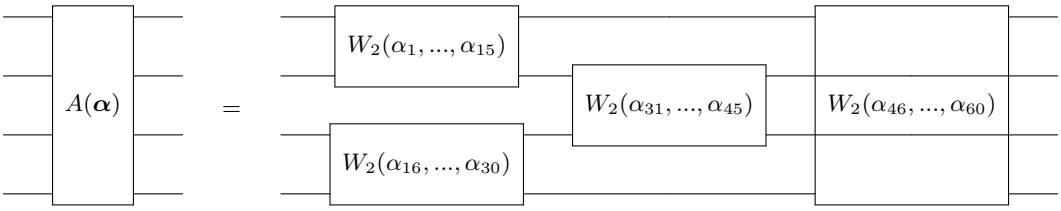

FIG. 1. Layered ansatz $A(\boldsymbol{\alpha})$, where each W_2 is a collection of 18 gates which is characterized by 15 parameters that implements an arbitrary two qubit gate, as shown in Figure 7(b) of [7]. Thus, in total, this ansatz has 60 parameters, which are all encompassed by the parameter vector $\boldsymbol{\alpha}$.

ment $|t_2\rangle$. We continue on until we have generated a set of classical bit strings $\{t_1, ..., t_N\}$. This procedure is a Markovian process and will be referred to as Markovian sampling. Markovian sampling allows us to explore the sound space of Q and hear patterns in how it evolves as we continue through the space, however it can also get stuck in a subspace of the full sound space, meaning that it explores Q by sending in basis states that are only a subset of the full basis. If we use only this Markovian sampling procedure, then there is no way to break out of this subspace. We discuss this more in the final paragraph of this section.

We have thus far mapped the quantum algorithm Q to a set T of N classical bit strings, and now must map T to sound. We chose to iterate through T and map each bit string to a note/chord. Thus we will have a song consisting of N chords. As mentioned, we chose to map to MIDI because of its simplicity, but of course there are infinitely many mappings to choose from. Some of the mappings that are implemented in qSonify are listed below. Note that each mapping below takes in a bit string $t_i \in T$ and maps it to a note/chord.

- **Fermionic** This mapping is exactly as discussed in the beginning of this section, as defined in [2]. There is a one-to-one correspondence between each bit in t_i and a chosen note. If the qubit is measured to be 1, then the note is included in the chord, otherwise, it is not.

- **Grand Piano** This method is defined by two sets of notes to be used for the bass clef and the treble clef. If the bit string t_i has n bits, then we choose $\lfloor n/2 \rfloor$ of them to be used to select which note to play in the bass clef and $\lceil n/2 \rceil$ of them to select which note to play in the treble clef (with this definition none of the bits are being used for both the bass and the treble clef, but this is also an option). In other words, part of t_i serves as an index to select notes for the bass clef and part of t_i serves as an index to select notes for the treble clef. These notes are played together as a chord.

- **Frequency$_\beta$** This method uses a more exponential scaling, which can be useful since sound is perceived exponentially (one octave is from 400Hz to 800 Hz and also from 800 Hz to 1600Hz). We first convert t_i to decimal by assuming that t_i is in base β. For example, if $\beta = 3$, then we read in t_i as if it were in base 3 (even though it is a binary number) and convert it to decimal. Call this number b. We then scale this value by some chosen factor w which has units of Hz, sending $b \to wb$, and then find the MIDI note that has a frequency closest to b.

An important note is the following. Consider an approximately classical algorithm M, meaning that basis states are approximately mapped to other basis states rather than a complex superposition of basis states when operated on by M. If this is the case, then we are very limited in the sound space that we explore when using the Markovian sampling method. For example, if $M|0\rangle^{\otimes n} = |1\rangle^{\otimes n}$ and $M|1\rangle^{\otimes n} = |0\rangle^{\otimes n}$, then we may only ever explore the sound of M limited to the subspace $\{|0\rangle^{\otimes n}, |1\rangle^{\otimes n}\}$. Therefore, we should introduce some perturbations to our starting conditions or throughout the algorithm M such that more of the sound space can be explored. Even something as simple as exploring $MH^{\otimes n}$, where H is the Hadamard gate, can expand the reachable space. This phenomena of getting stuck in subspaces of the full sound space of an algorithm is not only relevant to approximately classical algorithms but also to sonifying any arbitrary quantum algorithm. We use Markovian sampling as a way of traversing through the sound space, but it is still often necessary to manually increase the reachable space since certain algorithms restrict the Markovian sampling method.

V. THE TOOLBOX

The idea behind the toolbox is that common subroutines show up often in quantum algorithms – such as the preparation of GHZ and W states [8], the exponentiation of a tensor product of Pauli operators, the Grover diffusion operator, etc – and we should become familiar with how these subroutines sound. When sonifying an unknown algorithm, the hope is that we can hear these subroutines that we are familiar with from our toolbox embedded within the larger algorithm, thereby allowing us to simplify sets of compiled gate sequences to higher level subroutines that we understand.

Choosing the subroutines that are to be included in the toolbox is crucial and nontrivial. For example, from

the author's experience, the algorithm to prepare a GHZ state often has a characteristic sound whereas the Quantum Fourier Transform does not seem to have a very useful sound signature that makes it easy to identify.

As mentioned in Section IV, oftentimes we must manually introduce effects to increase the reachable space of our sonification procedure such that we are not limited to a subspace of the full sound space. This idea also applies to building our toolbox. When adding the sounds of subroutines to the toolbox, we must be sure to fully explore the subroutine.

VI. CONCLUSION

We have presented the idea of using sonification as a tool to decompile quantum algorithms by building abstractions, and we discussed how new quantum algorithms can be developed by using these abstractions to form a high-level understanding of their function. We created the qSonify GitHub repository in order to aid the study of sonification using the MIDI mapping, and include examples of sonified algorithms.

The MIDI mapping used throughout this paper is a simple method, but infinitely many more exist that remain to be studied. There are various tricks from the study of sonification in music technology that could be applied in relation to quantum algorithms, such as spacialization [9] and continuous mappings, and we emphasize that mappings better than our MIDI mapping must be found in order for our method to be practically useful. Nonetheless, it has proven useful for proof-of-principle examples and serves as a starting point for the study of using sonification to decompile quantum algorithms.

VII. ACKNOWLEDGMENTS

The author would like to acknowledge Ian Hattwick for discussion on sonification in music technology, Andrea Li for fruitful conversations on qSonify, and Patrick Coles for discussion on decompiling quantum algorithms.

[1] Joseph T. Iosue. *qSonify*. GitHub repository, github.com/jiosue/qSonify/.

[2] Volkmar Putz & Karl Svozil. *Quantum Music*. Soft Computing, March 2017, Volume 21, Issue 6, pp 1467 - 1471.

[3] James Weaver. *quantum-toy-piano-ibmq*. GitHub repository, github.com/JavaFXpert/quantum-toy-piano-ibmq.

[4] Alexis Kirke. *Application of Grover's Algorithm on the ibmqx4 Quantum Computer to Rule-based Algorithmic Music Composition*. arXiv:1902.04237, Feb 2019.

[5] Stephen Barrass & Gregory Kramer. *Using sonification*. Multimedia Systems 7: 23-31, Springer-Verlag, 1999. https://ccrma.stanford.edu/courses/120-fall-2005/using_sonification.pdf

[6] Lukasz Cincio, Yigit Subasi, Andrew T. Sornborger, & Patrick J. Coles. *Learning the quantum algorithm for state overlap*. arXiv:1803.04114, March 2018.

[7] Sumeet Khatri, Ryan LaRose, Alexander Poremba, Lukasz Cincio, Andrew T. Sornborger, & Patrick J. Coles. *Quantum-assisted quantum compiling*. arXiv:1807.00800, July 2018.

[8] D. Cruz, R. Fournier, F. Gremion, A. Jeannerot, K. Komagata, T. Tosic, J. Thiesbrummel, C. L. Chan, N. Macris, M.-A. Dupertuis, and C. Javerzac-Galy, Efficient quantum algorithms for GHZ and W states, and implementation on the IBM quantum computer, arXiv:1807.05572.

[9] Edward Childs & Ville Pulkki. *Using multi-channel spatialization in sonification: A case study with meteorological data*. International Conference on Auditory Display, July 2003. http://hdl.handle.net/1853/50506.