mat2qubit: A lightweight pythonic package for qubit encodings of vibrational, bosonic, graph coloring, routing, scheduling, and general matrix problems

Nicolas P. D. Sawaya
Intel Labs, Santa Clara, California
(Dated: May 23, 2022)

Preparing problems for execution on quantum computers can require many compilation steps. Automated compilation software is useful not only for easier and faster problem execution, but also for facilitating the comparison between different algorithmic choices. Here we describe mat2qubit, a Python package for encoding several classes of classical and quantum problems into qubit representations. It is intended for use especially on Hamiltonians and functions defined over variables (e.g. particles) with cardinality larger than 2. More specifically, mat2qubit may be used to compile bosonic, phononic/vibrational, and spin-s problems, as well as classical problems such as graph coloring, routing, scheduling, and classical linear algebra more generally. In order to facilitate numerical analyses and ease of programmability, a built-in computer algebra system (CAS) allows for fully symbolic preparation and manipulation of problems (with symbolic operators, symbolic coefficients, and symbolic particle labels) before the final compilation into qubits is performed. We expect this code to be useful in the preparation and analysis of various classes of physics, chemistry, materials, and optimization problems for execution on digital quantum computers.

I. INTRODUCTION

When using a qubit-based quantum computer, it is necessary to map problem instances and other primitives to a Pauli representation. This is true for both quantum Hamiltonians and classical cost functions. Some problems classes are trivial to map to qubits, for example quantum spin models (since distinguishable spin-half particles are the same as qubits), and classical optimization problems defined over binary variables (e.g. MaxCut). Other problems require more sophisticated encoding procedures; for instance, fermionic problems such as the electronic structure problem in chemistry require the Jordan-Wigner or other encodings [1–5].

Notably, there are many scientifically and industrially relevant problems that are not most naturally defined over qubit, fermionic, or classical binary degrees of freedom. Such problems, for which the subsystems often have cardinality greater than 2, include bosonic [6, 7], vibrational/phononic [8–10], spin-s [11], graph coloring [12–15], routing [16, 17], and scheduling [18–23] problems, as well as more general classical linear algebra problems. mat2qubit is a Pythonic software package for encoding these Hamiltonians and cost functions into qubit operators; previously, software had been developed mainly for compiling fermionic problems [24–26] and classical problems over binary variables [25–28].

The task of interest is to perform mappings of the following class:

$$\sum_i b_i M^{(i)}_0 \otimes M^{(i)}_1 \otimes \cdots \rightarrow \sum_j c_j \bigotimes \{I, \sigma_x, \sigma_y, \sigma_z\}$$  

(1)

where $M^{(i)}_\alpha$ are square matrix operators of arbitrary size, $I$ is the identity, and $\{\sigma_x, \sigma_y, \sigma_z\}$ are the Pauli matrices. Each subsystem on the left-hand side maps to a disjoint set of qubits; the compilation procedure is described in detail in previous work [20, 30]. The current version of mat2qubit allows only for trivial (equivalent to bosonic) commutation relations, i.e. $[M^{(i)}_\alpha, M^{(j)}_\beta] = 0$. This requirement is obeyed by the wide range of quantum and classical problems discussed in this work, but excludes problems such as many-body Hamiltonians over fermions [1], anyons [31], or parafermions [32].

The mapping of equation (1) is not unique; for each subsystem one may in principle choose any integer-to-bitstring mapping, some of which require more space (qubits) than others. One purpose of this code is to simplify and automate the implementation of a wide range of encodings. Several previous works have studied the resource tradeoffs (for example depth-space tradeoffs) between different encodings [8, 16, 24, 31, 32, 33, 36]. mat2qubit was developed primarily for the specific task of preparing and compiling various problems into qubit representations; for other aspects of the compilation pipeline and for running the resulting qubit-based problems on quantum computers, we suggest the use of one of the many actively maintained quantum simulators [25, 26, 28, 37, 47] and compilers [26, 28, 37, 45, 47, 51].

*nicolas.sawaya@intel.com
II. PROGRAM FLOW

Here we provide an overview of the mat2qubit program flow, focusing primarily on the most useful front-end classes and subroutines. The code’s dependencies are scipy, sympy, and openfermion; the main component used from the latter package is the convenient QubitOperator class. Our primary goal was usability, and as such there are several potential improvements in terms of computational performance. For instance, the encoding procedure could be sped up by using linear algebra subroutines related to the Walsh-Hadamard transform [52]; further, the code could be parallelized using common pythonic packages [53, 54]. Notably, despite the omission of such optimizations, we have used the code to compile problems up to thousands of qubits and published results up to hundreds of qubits [55], meaning that the current version may be used for analyses well beyond what existing quantum hardware is capable of simulating.

A. Operators over $d$-level subsystems.

Three of the fundamental front-end classes of mat2qubit are dLevelSubsystem, compositeDLevels, and compositeOperator (the fourth, qSymbOp, is discussed below). The first of these is used to define a quantum particle (i.e. “subsystem”). For an operator to be eventually converted into a qubit representation, the parameters $d$ (cardinality of the variable) and enc (the chosen encoding) must be specified. The following code defines a subsystem with a truncation of 8 levels.

```python
import mat2qubit as m2q
bosonic_mode_1 = m2q.dLevelSubsystem(d=8, enc = "stdbinary")
bm1_pauli_operator = bosonic_mode_1.opToPauli("ad")
```

The third line converts the creation operator ($a^\dagger \equiv ad$) to a qubit representation.

Collections of subsystems are organized using the compositeDLevels class. However, the user will most often interact with its child class compositeOperator. Subsystems are added using appendSubystem(), and terms are added using addHamTerm(). The following example implements a two-site Hamiltonian using creation (ad) and annihilation (a) operators.

```python
ss1 = m2q.dLevelSubsystem(4, "unary")
twoSiteOperator = m2q.compositeOperator()
twoSiteOperator.appendSubystem(ss1) # Site 1
twoSiteOperator.appendSubystem(ss1) # Site 2
twoSiteOperator.addHamTerm(4.2 , [(1 ,"ad"), (0,"a")])
twoSiteOperator.addHamTerm(4.2 , [(0,"a"), (1,"ad")])
twoSiteOperator.addHamTerm(0.5 , "ident" ) # Identity
pauli_operator = twoSiteOperator.opToPauli()
```

The final line converts the operator to a qubit (Pauli) representation. There is no requirement that subsystems have the same size $d$; hence heterogeneous quantum systems are allowed. For instance, one may implement a spin-boson model using a collection of two-level particles and bosonic particles [56]. Nor do subsystems need to use same encoding; indeed, there may be cases when using different encodings for different subsystems leads to a compiled operator with fewer Pauli terms [29].

B. Compiling into qubit operators

Once an operator has been defined, the goal is to implement the mapping of Equation 1. mat2qubit has built-in options to compile the higher-level multi-particle operator into a qubit operator. The final step of this procedure is to run the opToPauli() method in the last line of the previous section. The built-in encodings are unary (also called one-hot), standard binary (SB), Gray code, and block unary. The latter is in fact a class of encodings, for which one must specify the local encoding and the block size $g$ [29, 30, 33]. For a $d$-level particle, unary requires $d$ qubits, Gray and SB require $\lceil \log_2 d \rceil$ qubits, and block unary requires $\lceil g \lceil \log_2 (g + 1) \rceil$, where $\lceil \cdot \rceil$ is the ceiling function.

Note that each particle in the original space is mapped to its own disjoint set of qubits. In principle, more complex mappings are possible, for example different $d$-level particles could map to the same subset of qubits (perhaps in order to reduce space requirements), though the current code is not capable of such embeddings.

Novel encodings beyond those mentioned above may be added. To do so, the user must define functions for both the integer-to-bit mapping and the bitmask subset, which determines the set of qubits over which a particular matrix element operates. Previous work discusses this concept more thoroughly [29, 30]. Once these functions are defined they may be introduced by directly modifying integer2bit.py. Note that the current infrastructure is built to handle...
encoding parameters as well—for instance, parameters are required in order to fully define a block unary encoding, and other user-defined parametrized encodings are allowed.

C. mat2qubit’s symbolic computer algebra system

For ease of programmability, we have included a computer algebra system (CAS) that simplifies operator algebra. Our qSymbOp class allows for construction and algebraic manipulation of fully symbolic operators, where local operators, subsystem (particle) labels, and coefficients may all be defined symbolically. Allowing all three of these entities to be labelled symbolically allows for improved programmability and readability. For example, one may use symbolic coefficients and then pass the symbolic object to various functions that implement different numerical parameters. Further, the ability to label subsystems with arbitrary character strings can prevent confusion, for example clarifying when different particles are of different types; if one were to model a chain of spins each coupled to a bosonic bath, one might label spin subsystems with \( \_\text{sp}0, \_\text{sp}1 \), etc. and bosonic modes with \( \_\text{b}0, \_\text{b}1 \), etc.

The following is an example of the use of qSymbOp, with operators defined over subsystems A, B, and C.

```python
sympy_op1 = m2q.qSymbOp("k [n_A] ++ j [p_B] ++ r [l_C]")
sympy_op2 = m2q.qSymbOp("k [q_A] ++ -r [p_C]")

# Product of operators
sympy_product = sympy_op1*sympy_op2

# Substitute numbers in for variables

sympy_product.scalar_subs({"k":2.1, "j":np.pi, "r":3.2})
```

Multiplication and addition operations may be implemented directly; the background data structures are appropriately updated to ensure there are no duplicate terms. Local operators are stored in order of multiplication as they may not commute. The final line of the above code fragment is used to convert the symbolic coefficients to numerical values, a step that must be done before the operator is compiled into a qubit representation. The final mapping is performed as follows, where three inputs must be given: the subsystem labels, \( d \) values, and encodings.

```python
ssid_order = ['A', 'B', 'C']
sid_to_d = [4,4,6]  # Heterogeneous \( d \) values
ssid_to_enc = ['stdbinary', 'gray', 'stdbinary']  # Mixed mappings

dlev_obj = m2q.symbop_to_dlevcompositeop(sympy_product, ssid_order, ssid_to_d, ssid_to_enc)
pauli_operator = dlev_obj.opToPauli()
```

D. Built-in and user-defined local operators

There are many built-in operators, which are enumerated in the builtInOps list of dLevelSystemEncodings.py. Some are duplicate; for instance both qhoPos and q may be used for standard bosonic position operator, while qhoMom and p are both for the momentum operator. Also notable are \( q_2, q_3, \) etc., which denote powers \( q^2, q^3, \) etc. of the position operator, used especially in molecular vibrational Hamiltonians \([8–10]\). \( S_x, S_y, \) and \( S_z \) are the standard spin-\( s \) operators for the three Cartesian directions.

Further, there are built-in operators that can effectively be used to implement the discrete quantum intermediate representation (DQIR) \([30]\). The DQIR is an intermediate representation that was previously introduced in order to facilitate compilation and analysis of classical optimization problems defined over discrete variables. Two “primitives” of the DQIR are the indicator primitive \( P^{(k)}_\alpha \equiv |k\rangle \langle k|_\alpha \), and the transfer primitive \( T^{(k-l)}_\alpha \equiv |k\rangle \langle l|_\alpha \). They are implemented in mat2qubit as \( P r[i] \_\alpha \) and \( k[i] \_b \{j\} \_\alpha \), respectively. For instance,

```python
Pr3_ss
```

denotes the projector (indicator primitive) \( |3\rangle \langle 3| \) and

```python
k2b3_ss
```

denotes the ket-bra (transfer primitive) \( |k\rangle \langle l| \), both operating on subspace \( ss \). The examples/ directory of the source shows examples of these indicator primitives being used in problem instances of graph coloring, traveling salesperson, and machine scheduling. The transfer primitives may be used for example to define QAOA mixers within the DQIR \([30]\). The user is not required to use built-in operator definitions; arbitrary local \( d \)-level operators may be used with either qSymbOp or compositeOperator objects, in the following manner. For the latter, one may directly insert an operator string that uses numpy arrays instead of character strings:
For user-defined local matrix operators in qSymbOp, one may use any alphanumeric character string as long as the inpOpChars argument is specified when converting to a qubit operator. In the following example the local operators W and V are defined by the user.

```python
import numpy as np

locop1 = np.random.rand(d,d)
locop2 = np.random.rand(d,d)
twoSiteOperator.addHamTerm(4.2, [(1, locop1), (0, locop2)])
```

E. Distribution and examples

The source code and installation instructions are available at [github.com/IntelLabs/mat2qubit](https://github.com/IntelLabs/mat2qubit). The package is distributed under the Apache 2.0 license. Many examples are available in the repository, where we have implemented simple instances of the following problem types: Bose-Hubbard models, anharmonic molecular vibrations, interacting spin-\(\frac{3}{2}\) and spin-\(\frac{5}{2}\) models, random tensor trains from classical linear algebra, graph coloring, the traveling salesperson problem, and machine scheduling.

III. CONCLUDING REMARKS

We have introduced mat2qubit, a lightweight python package for compiling quantum and classical problems into qubit representations. Specifically, it is intended for non-fermionic problems like bosonic, phononic, and spin-s problems, as well as discrete optimization problems like scheduling, routing, and coloring. It may also be used to encode general matrix problems using various embeddings, as long as the matrix is first expressed as a sum of tensor trains. Theoretical details related to the code have been discussed previously [29, 30]. Before being open sourced, early versions of this package were used in numerical studies of several classes of Hamiltonian [9, 29, 30, 33, 55, 57, 58]. The current release includes a computer algebra system for defining and manipulating quantum operators for which coefficients, local operators, and subsystem labels are all symbolic; this is intended to improve programmability, usability, and interpretability when constructing complex Hamiltonians. One main purpose of the code is to easily and rapidly implement many encodings for a given problem, in order to compare resource requirements. As quantum computers become more ubiquitous and their hardware characteristics more varied, such resource comparisons will be needed in order to determine the most efficient encodings for a given problem and a given set of hardware. In conclusion, we expect mat2qubit to be useful for compiling a wide variety of problems for quantum computers.

ACKNOWLEDGEMENTS

I am grateful for useful discussions with Tim Menke, Stuart Hadfield, Gian Guerreschi, Shavindra Premaratne, Alán Aspuru-Guzik, Thi-Ha Kyaw, and Albert Schmitz.

[1] G. Ortiz, J. E. Gubernatis, E. Knill, and R. Laflamme, Quantum algorithms for fermionic simulations, Physical Review A 64, 022319 (2001).
[2] S. B. Bravyi and A. Y. Kitaev, Fermionic quantum computation, Annals of Physics 298, 210 (2002).
[3] J. T. Seeley, M. J. Richard, and P. J. Love, The Bravyi-Kitaev transformation for quantum computation of electronic structure, The Journal of Chemical Physics 137, 224109 (2012).
[4] Y. Cao, J. Romero, J. P. Olson, M. Degroote, P. D. Johnson, M. Kieferová, I. D. Kivlichan, T. Menke, B. Peropadre, N. P. Sawaya, et al., Quantum chemistry in the age of quantum computing, Chemical Reviews 119, 10856 (2019).
[58] T. H. Kyaw, T. Menke, S. Sim, A. Anand, N. P. Sawaya, W. D. Oliver, G. G. Guerreschi, and A. Aspuru-Guzik, Quantum computer-aided design: digital quantum simulation of quantum processors, Physical Review Applied 16, 044042 (2021).