# qdb: Inserted Tomography for Breakpoint Debugging in pyQuil

Ellis Hoag, Michael Zhu, Sean Decker

June 6, 2019

## 1 Introduction

Over the last decade, quantum computing has quickly grown from academic pipe dream to an exciting reality which presents itself as an imminent revolution in coming. It continues to grow more powerful as a practical computing paradigm as qubits grow in number and fidelity. Companies like Rigetti are streamlining the process of running computation on real Quantum Processing Units (QPUs) so that quantum computing benefits from the ease of access afforded by the cloud.

As QPUs and their programs become larger and more complex, it becomes more important for programmers to have the tools to explore the behavior of programs and find bugs. However, debugging on a QPU presents a unique challenge – characterizing the state of quantum memory requires measurements, yet measurements collapse the states of the qubits we would like to characterize. This tension makes quantum computing fundamentally incompatible with classical breakpoint debugging, which relies on the assumption that we can freely examine memory without affecting the execution of the program.

By consequence, there are no existing tools for breakpoint debugging on a physical QPU. Code must be run to completion and only the final output is available to the programmer. On a quantum virtual machine (QVM), the programmer is able to access the simulated wavefunction, but QVMs are limited in the number of qubits they are able to simulate. To address this, we propose a library for automatically performing state tomography on a subset of qubits to reconstruct their state at arbitrary breakpoints in a pyQuil program.

Our solution is qdb (https://github.com/mzhu25/qdb), an extension of pdb to step through the construction of a quantum program in pyQuil, allowing the user to make queries to arbitrary qubits at intermediate stages. When a set of qubits is queried, the program is first simplified to only include the instructions necessary to characterize the state of those qubits. Then we run state tomography and display the estimated density matrix.

### 1.1 Tomography

Quantum tomography is the process of reconstructing the quantum states, measurements, or processes of some source of quantum systems though measurements on quantum systems produced by the source. Because quantum states are inherently probabilistic, it is impossible to determine a state simply given one copy of it. Quantum tomography thus relies on repeated generations of the quantum state of interest which then is measured in varying basis and then combined using Börn's rule.

qdb relies on quantum state tomography in order to reproduce the density function of a quantum program of interest at the breakpoint specified by treating the the quantum program until the breakpoint as the source of quantum states.

# 2 Implementation

qdb is an extension of pdb that also owns the Program object that is being constructed. Users can step through their code until they want to see the resulting wavefunction of the program they have constructed so far. Our main work is to run tomography, but our implementation allows users to query the entangled set of qubits and to see the control flow graph of the program. More detail about usage can be found in the README section of our repository.

## 2.1 Program Trimming

It is beneficial to remove instructions, qubits, and control flow that have no affect on the wavefunction that the user wants to query. Simpler programs with fewer instructions or qubits can be run more quickly and more accurately on QPUs which makes for more accurate wavefunctions. Instructions can be optimized out by looking at the entangled graph of qubits and the dependency graph of classical bits, and how they both propagate through control flow. If a basic block has no instructions left, it can be removed from the control flow graph.

## 2.2 Wavefunction Ensemble

The quantum tomography returns a density matrix, but wavefunction formulas are often more readable. The wavefunction that corresponds to a given density matrix can be derived by finding the eigenvectors of the matrix. These eigenvectors then have a probability given by their eigenvalue of being the perceived state of the quantum system. qdb thus returns these possible wavefunctions along with their probability to make debugging easier for the user.

# 3 Classical Control Flow

Implementing classical control flow instructions on quantum architecture remains a major engineering challenge due to slow readout times relative to the lifetime of qubits. Although classical control flow is not yet supported on QPUs, we believe that developing debugging solutions for classically controlled quantum programs is still valuable, both as a theoretical exercise and in anticipation of future hardware support.

Consider the following motivating example to show the complexities of running tomography on programs with control flow.

```
H 0
MEASURE 0 b
JUMP-WHEN @END b
X 1
JUMP @END
```

Figure 1: Simple example of classical control flow in Quil.

At the end of the program, we can say that we're in a mixed state: the two possible wavefunctions are $|00\rangle$ if the branch is not taken, and $|11\rangle$ if it is. But if we were to put a breakpoint right after the `X 1` instruction, we would expect to be in a pure state $|11\rangle$ which only arises if we happen

to have measured qubit 0 to be in the excited state. In other words, our breakpoints need to be "context-aware", in that our estimated state must be conditioned on some subset of execution paths, determined by the location of the breakpoint.

Although Figure 1 is simple enough, it quickly becomes difficult to reason about a program as its control flow becomes more complicated – a program might have many `JUMP-WHEN` instructions, each of which depends on the measurement of multiple qubits. This motivates the following paradigm, which abstracts and generalizes the previous example.

## 3.1 Quil Control Flow Graph

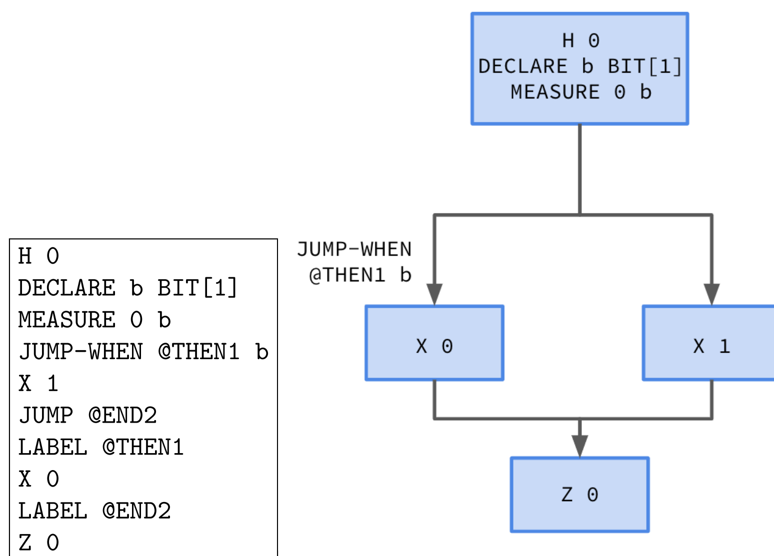


Figure 2: Quil program and corresponding control flow graph.

We can construct a control flow graph (CFG) for a Quil program, where nodes correspond to "basic blocks". We define a basic block to be a sequence of instructions such that if the first instruction is executed, then all are guaranteed to execute. We can then add edges based on `JUMP` instructions. See Figure 2 for an illustration. Observe that we could also have multiple levels of branching (induced by nested `JUMP-WHEN`s) and more than two out-edges for a single node (induced by consecutive `JUMP-WHEN`s).

We have implemented a linear time CFG construction algorithm in qdb (see `control_flow_graph.py`). For simplicity, we are only considering acyclic graphs (DAGs) for the time being, i.e. we disallow `while_do` in pyQuil.

With this paradigm in mind and a working implementation in hand, we explored several potential solutions to the classical control flow problem which leverage these new tools. We describe two of them below; a more comprehensive list can be found at https://github.com/mzhu25/qdb/issues/6.

### 3.2 Solution 1: Branch Rewiring

A simple strategy for dealing with breakpoints in a particular execution branch is to rewire the branches to effectively force the execution down the branch we care about through trial and error. This can be done by replacing the other branches with `RESET` and `JUMP` instructions, to clear the qubits and restart execution from the beginning when the wrong branch is taken. See Figure 3 for a simplified illustration.

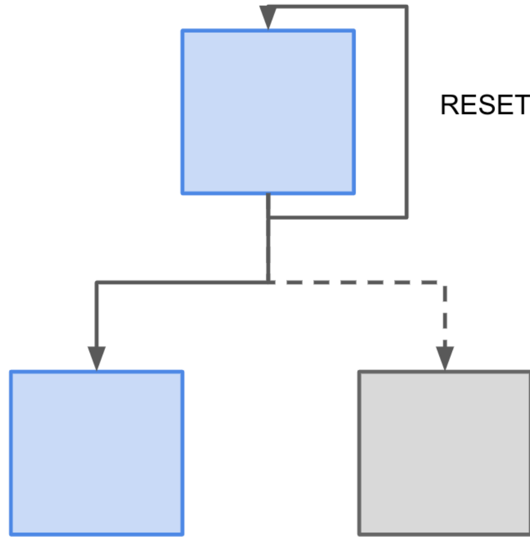


Figure 3: Branch rewiring.

While elegant, this solution cannot be used yet because the tomography suite in forest.benchmarking currently does not support classical control flow at all. Another problem is that this solution is intractable if the breakpoint is in an unlikely execution path.

### 3.3 Solution 2: Hybrid Tomography

The key observation underlying this solution is that, while we are limited in the amount of computation we can reasonably do on both the QPU and the QVM, these are entirely separate resources. If we can effectively split the debugging work between QPU and QVM, we have more computational power to work with. We term this class of solutions *hybrid tomography*.

Consider the generalization of Figure 2: we have a parent block $P$ which branches into two child blocks $X$ and $Y$, conditioned on the classical register $b$. $P$ includes some number of measurements and classical register operations, such that $b$ depends on some subset of qubits $B$ at the time of the branch. The user has placed a breakpoint at the end of $X$, and would like to query the state of some other subset of qubits $Q$ at this point. In what follows, let $\mathrm{ent}(Q)$ denote the entanglement set of $Q$, and let $\mathrm{ent}_P(B)$ denote the entanglement set of $B$ induced only by the multi-qubit gates in $P$. There are three cases to consider:

1. $\mathrm{ent}(Q)$ and $\mathrm{ent}(B)$ are always disjoint. This is the simplest case. We can characterize the branch probabilities by introducing an ancilla bit which gets flipped in one branch but not the other. We can then run $P$ many times and see how many times the ancilla bit gets flipped. We can then trim $P + X$ to only include gates involving the entanglement set of $Q$ and run state tomography to estimate the desired state.

2. $Q$ and $B$ are not entangled at any point in $P$, but become entangled in $X$. Moreover, we know the possible states of $\mathrm{ent}_P(B)$ going into $X$ – this is the case e.g. in Figure 2, where the last instruction in $P$ involving each qubit in $\mathrm{ent}(B)$ is a `MEASURE`. In this scenario, we can run $P$ many times to determine the probability of each possible post-measurement state of $B$ satisfying the branch condition. Then we can "clamp" $\mathrm{ent}_P(B)$ to one such post-measurement state, and run state tomography for $Q$ on $P' + X'$, where $P' := \mathtt{trim}(P, \mathrm{ent}_P(Q))$ and $X' := \mathtt{trim}(X, \mathrm{ent}(Q))$.

3. The general case, i.e. no assumptions on the entanglement of $Q$ and $B$. This presents a problem: on one hand, we don't know how measuring $B$ affects the state of $Q$; on the other, we need to know which branch a particular execution would have gone down.

We observe that if we had an oracle for the wavefunction of $\mathrm{ent}_P(B)$, we'd basically reduce the general case to the second case. So, we'll let the QVM play the role of this oracle.

Let $P_{\mathrm{QVM}} := \mathtt{trim}(P, \mathrm{ent}_P(B))$, and and run the QVM wavefunction simulator on $P_{\mathrm{QVM}}$ to obtain the wavefunction ensemble for $\mathrm{ent}_P(B)$ at the branching point. Then we proceed as we did in case 2: clamp to one of the wavefunctions in the ensemble, and run state tomography for $Q$ on $P' + X'$. Thus the QVM only needs to simulate a subset of qubits ($\mathrm{ent}_P(B)$) on a subset of the program ($P_{\mathrm{QVM}}$), and we only need to run state tomography experiments on the QPU for a trimmed program ($P' + X'$) without control flow. This might still be intractable, e.g. if $\mathrm{ent}_A(B)$ is too large to simulate on the QVM, but we hope that this will be a good compromise for most practical applications.

# 4 Future Work

There remain many areas of ongoing work on qdb, spanning theory and implementation. The following subsections discuss three such areas that are of particular value and interest.

## 4.1 Entanglement Set Evolution

We have implemented an undirected "entanglement graph", where nodes are qubits and we add an edge for every multi-qubit gate in a program. We then compute an (over)estimate of the entanglement set of some qubit as the connected component containing it.

With a tighter estimate of the entanglement set, we can trim programs better. One low-hanging fruit at the moment is that we currently do not take into account the fact that a qubit can be disentangled by measurement, so edges can be both added and deleted over the course of an execution path. An interesting observation is that if we would like to perform entanglement estimation queries as we evolve the graph, we arrive at the dynamic connectivity problem, a well-studied problem in algorithms.

## 4.2 Control Flow Dependencies

For the hybrid tomography approach, we need to compute the set of qubits that a particular branch condition depends on, using the observation that a classical register can only be written to by measurement or binary classical operations. We have implemented an algorithm that does this, but it currently overestimates the set (discussion here).

We can exactly compute $B$ for a particular classical register $b$ by performing a reverse pass over the program starting at the branching point, keeping track of a set `dependencies` of classical registers and qubits. If the target register of a binary operation is in the set, we add the other register to the set as well. If the target register of a `MEASURE` is in the set, we remove that register and add the measured qubit. We continue this way until we are left with only qubits.

## 4.3 Hybrid Tomography on Arbitrary Graphs

Although Section 3.3 gives a theoretical proof-of-concept for hybrid tomography, it is difficult to reason about how those rules generalize to arbitrary DAGs (and cycles introduce yet another layer of difficulty). Once we have a clearer theoretical picture of hybrid tomography, we will be able to implement it in qdb.