CS 269Q Final Project: Inserted Tomography for Breakpoint Debugging.

Michal Adamkiewicz, Jorge Cueto, Kent Vainio

June 6, 2019

1 Introduction

Our project implements an inserted tomography algorithm for breakpoint debugging, as outlined in the CS 269Q class document "Final Project Guidelines." In this report, we describe the design and implementation of our inserted tomography algorithm, present the data and analysis methods we leveraged to test our implementation, and discuss potential future extensions of our work. The code we developed during this project can be found here:

`https://github.com/kenzomenzo/tomography-debugger/`

Please follow the README for instructions. We also submitted a pull request so that our code can become part of the forest-benchmarking library. You can find the pull request here:

`https://github.com/rigetti/forest-benchmarking/pull/137`

2 Design and Implementation

Our project implements an inserted tomography algorithm for breakpoint debugging. We implement a new `tomographize` function. The `tomographize` method takes in a program, a list of qubit indices to tomographize, the number of times to run each tomography experiment to get the expected value (`num_shots`), the type of the tomography algorithm (e.g. "compressed_sensing" or "lasso"), and the number of pauli matrices to use in the tomography (`pauli_num`). The method returns the corresponding density matrix. The `tomographize` function works by generating an experiment that holds all 2^n possible Pauli matrices for the given number of qubits and sampling `pauli num` Pauli matrices without replacement.

In order for our `tomographize` function to be leveraged by quantum programmers using the Rigetti Forest platform, we hope to push our function into the tomography.py file in the forest-benchmarking library. In this way, developers can call the `tomographize` method at any given point in their program to determine the state tomography of the program at that specific breakpoint.

The `tomographize` method returns the density matrix `rho`, which describes the state of the system for the specified set of qubits at the specified breakpoint, based on the program that has been passed in as a parameter. As Vandersypen and Chuang assert in their 2004 publication, "The density matrix `rho` completely describes our knowledge of the state of a system. Measurement of the density matrix is therefore extremely helpful when testing or claiming the preparation of specific quantum states" [1]. Taking into account the reality that different tomography algorithms work better in different scenarios, our `tomographize` method allows the user to specify which tomography algorithm they want to use to produce a density matrix. We allow the user to select from 3 distinct tomography algorithms, each of which have their own pros and cons. The two distinct tomography algorithms we have implemented for this project are the "Quantum State Tomography Via Compressed Sensing" algorithm using Pauli measurements for matrix recovery, described in Gross et al.'s 2010 publication [2] and the "matrix Lasso selector" algorithm described in Equation 3 of Flammia et al.'s 2012 publication [3]. The third algorithm is the linear inversion method already present in the Forest benchmarking tomography.py file. Users can select the tomography algorithm that best fits their needs, based on the composition of their program and the key metrics they want to optimize for (e.g. efficiency of the algorithm vs. accuracy).

We implemented Gross et al.'s algorithm first because we found it to be the most simple tomography algorithm to understand and implement. According to Gross et al., this approach is specialized for quantum states that are fairly pure, and it offers a notable performance improvement on large quantum systems. This algorithm, in other words, offers very efficient performance at the cost of being optimized more narrowly for relatively pure states, so it may not be as accurate in determining the density matrix of states that are not very pure. Gross et al.'s approach is able to reconstruct an unknown density matrix of dimension d and rank r using $O(rdlog^2d)$ measurement settings, compared to standard methods that require d^2 settings [2]. Their method proceeds as follows:

Step 1: Measurement - Choose m integers at random in the range $[1, d^2]$, corresponding to a choice, A_i, of one of the d^2 Pauli matrices for a system with n qubits (where $d = 2^n$). Then measure the expectation value $tr\rho\omega(A_i)$.

Step 2: Optimization - One then solves the following convex optimization problem:

Minimize $||\sigma||_{tr}$ subject to $tr(\sigma) = 1$ and $tr\omega(A_i)\sigma = tr\omega(A_i)\rho$.

Flammia et al.'s method was not a viable starting point because, while it is an efficient algorithm, it only works for reconstructing low rank density matrices. As Flammia et al. state, their low rank methods "do not attempt to reconstruct the complete density matrix, but only a rank-r approximation, which is accurate when the true state is close to low-rank" [3]. Flammia et al.'s "matrix Lasso selector" leverages least-squares linear regression with trace-norm regularization to reconstruct the density matrix [3]. Flammia et al.'s method proceeds as follows:

Step 1: Measurement - Same measurement process as in Gross et al.'s algorithm.

Step 2: Optimization - Define the *sampling operator* to be a linear map A defined for all i. One then solves the following convex optimization problem:

$$\hat{\rho}_{\text{Lasso}} = \arg\min_{X} \tfrac{1}{2}\|\mathcal{A}(X) - y\|_2^2 + \mu\|X\|_{\text{tr}}$$

3 Results

To test the performance of the 3 tomography algorithms we implemented, we compare the results of each of our algorithms to the true output obtained using pyQuil's `wavefunction` method. We compare the density matrix produced by `wavefunction`, which will serve as our ground truth matrix, to the density matrix produced by each of our tomography algorithms, as well as the linear inversion tomography method implemented in Rigetti's `tomography.py` file. We visualize our results using Hinton plots, in which positive and negative values are represented by green and blue squares, respectively, and the size of each square represents the magnitude of each value. For each of our tomography algorithms, we also graph the matrix trace-norm against the number of measurements values taken. We also use vertical error bars to show the standard deviation of each point in each of our plots. We compare the plots corresponding to each of our tomography algorithms to the plot corresponding to the ground truth algorithm. After running numerous tests throughout the project (many of which can be found inside the **tomography test** iPython notebook that is included in our GitHub repository), we decided to report our final results on a simple 3 qubit program both with and without noise. The two programs are:

Listing 1: Program without noise

```
H 0
H 1
H 2
CZ 0 2
RX(pi/2) 1
RX(-pi/2) 2
CZ 1 2
X 2
CNOT 1 2
```

Listing 2: Program with noise

```
H 0
H 1
H 2
NOISY-CZ 0 2
NOISY-RX-PLUS-90 1
NOISY-RX-MINUS-90 2
NOISY-CZ 1 2
X 2
CNOT 1 2
```

We obtained our first set of results by running all three tomography algorithms on both the noisy and noiseless programs, using 32 Pauli matrices for the measuring expectation values. Each measurement loop was performed 1000 times. Below are the matrix norm results (calculated using `np.linalg.norm` between the ground truth density matrix produced by PyQuil's `Wavefuction` class and the density matrices produced by our tomography algorithms):

Tomography Algorithm	Noisy Program	Noiseless Program
Linear	0.80	0.71
Compressed	0.14	0.078
Lasso	0.074	0.047

We also produced Hinton plots for each experiment (where darker values indicate more negative entries in the matrix):

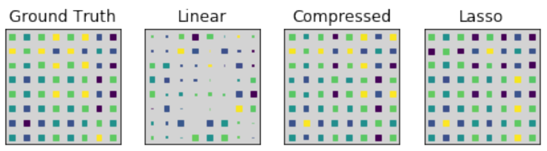

Figure 1: Noisy Hinton Diagram

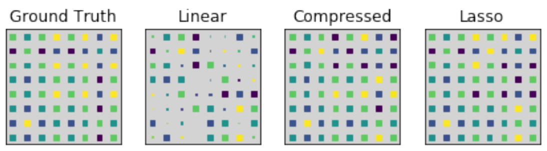

Figure 2: Noiseless Hinton Diagram

As can be seen from the table, the lasso method does the best in both cases, followed by compressed and then linear, which is very far behind the other two. We expect linear to not perform as well since it is a very simple algorithm, however it is important to include it for the sake of comparison. These results can also be seen in the Hinton diagram if you look at the overall distribution of colors (Lasso has the closest distribution to the ground truth in both cases).

What is interesting to note here is that the relative gap between compressed and lasso is larger for noisy images, indicating that lasso does better in noisy contexts. This is indeed what we expected from Gross et al.'s algorithm, as the authors claim that their approach only works well on states that are especially pure. This hypothesis was confirmed by our second experiment in which we ran each of the three algorithms for multiple Pauli measurement numbers, and then plotted a graph (following the graphs in [3]). The two graphs we produced are shown below

As you can see, is both the noisy and noiseless cases lasso does at least as well as compressed, if mot better. The really striking result is that for the noisy program lasso's matrix norm goes down to almost 0 after 30 Pauli measurements, whereas both the compressed and linear methods remain at a high matrix norm, indicating significant deviation from the ground-truth density matrix. The experiments used to generate the graph ran multiple trials of each tomography algorithm per Pauli measurement number, and then took the mean and standard deviation of those values to produce the final output. We ran 5 trails for the noiseless program and 2 trails for the noisy program, given the extra time it took to compute the density matrices for the noisy program. From our results we can conclude that lasso is the best performing method out of the

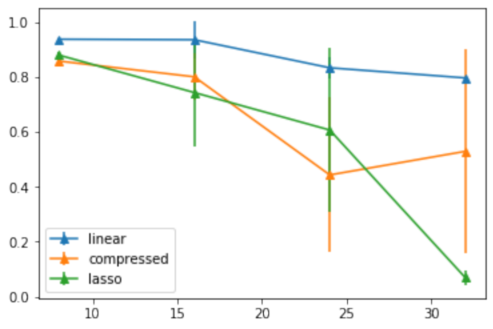

Figure 3: Noisy Program Graph

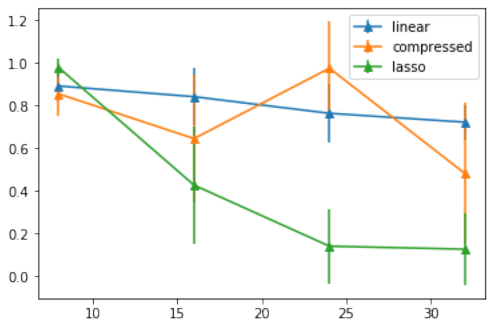

Figure 4: Noiseless Program Graph

three, however we still provide both compressed and lasso (and linear inversion, which is already contained in tomography.py) for the user to decide between.

4 Future Work

While our the tomography algorithms we implemented produced insightful results, there are several additional extensions we hope to implement in the future. For example, we would like to explore testing our algorithms on a greater number of qubits. Due to constraints related to time and computational power, we only tested our tomography algorithms on up to 3 qubits. With access to more time and more powerful computational resources, we would like to test our tomography algorithms on a greater number of qubits. We have designed our algorithms to work on an arbitray number of qubits, but in practice we are constrained by time and computational power.

In our future work, we would also like to expand the range of tomography algorithms for our `tomographize` function that we make available for users to use to determine the quantum state of their programs. For example, we can implement the "matrix Dantzig selector" algorithm described in Equation 3 in Flammia et al.'s 2012 publication. In this algorithm, the density matrix is reconstructed using constrained trace-minimization [3]. Like Flammia et al.'s "matrix Lasso selector" algorithm, the "matrix Dantzig selector" algorithm is an efficient algorithm, but it only works for reconstructing low rank density matrices. We actually started implementing the "matrix Dantzig selector" algorithm while working on our project, but we ran into problems calculating the convex optimization of complex numbers using the `abs` function. We would like to revisit implementing this algorithm in our future work. We can also implement the tomography algorithm illustrated by Cramer et al. in their 2010 publication. Cramer et al.'s approach is a very efficient tomography algorithm, but it assumes that the qubits in the system are laid out in a one-dimensional, linear formation, so it would not work on qubit infrastructures that are organized in a lattice formation [4]. Additionally, we can implement the standard, inefficient tomography algorithm outlined in CS 269Q Lecture 6. While this tomography algorithm is slow to run, we believe it is a good starting point because it covers a wide variety of real-world use cases [5], while many of the other tomography algorithms are more limited in scope. Ultimately, we hope to implement a wide variety of tomography algorithms in order to make it easier for the user to find a tomography algorithm that closely matches their specific debugging needs and goals.

5 References

[1] https://arxiv.org/pdf/quant-ph/0404064.pdf
[2] https://arxiv.org/pdf/0909.3304.pdf
[3] https://arxiv.org/pdf/1205.2300.pdf
[4] https://www.nature.com/articles/ncomms1147.pdf
[5] https://cs269q.stanford.edu/lectures/lecture6.pdf