Optimiser Based Pauli Decoder

Robert J. Harris[1,*]

[1]*ARC Centre for Engineered Quantum Systems, School of Mathematics and Physics, The University of Queensland, St Lucia, QLD Australia*

(Dated: August 5, 2021)

PAULI DECODER

In contrast to erasure decoders, optimally correcting Pauli errors for general stabiliser codes is a #P-complete problem [**?**]. For some specific codes, the code properties allow simplification of this problem [**? ? ?**]. Here we use general purpose integer optimisation software to perform maximum likelihood correction for general stabiliser codes, albeit with a high computational cost. This is useful for evaluating candidate error correcting codes, even if it is impractical for real-time quantum computing.

The algorithm can be applied to any stabiliser code, with suitable adjustments from the CSS variant described here, which we use to show numerical thresholds for the heptagon code, the reduced rate SCF code and the zero rate HaPPY code [**?**].

We represent operators as binary support vectors (BSV) [**? ?**] for X and Z components defined for n-qubit operator $\hat{A}$ as

$$\hat{A} = \bigotimes_j \hat{X}^{(\underline{a}_X)_j} \hat{Z}^{(\underline{a}_Z)_j} \equiv \hat{X}^{\otimes \underline{a}_X} \hat{Z}^{\otimes \underline{a}_Z} \tag{1}$$

where both $\underline{a}_X$ and $\underline{a}_Z$ are binary vectors of length n.

For CSS codes either $\underline{a}_X$ or $\underline{a}_Z$ will be zero for a given stabiliser or logical operator. Further, for a self-dual CSS codes such as the heptagon code $\underline{a}_X = \underline{a}_Z$.

To characterise the code performance we assume that X and Z type errors are I.I.D. For clarity we drop explicit Pauli labels and describe a decoder for a CSS code with Z errors.

We consider a generic dephasing error given by

$$\hat{\mathcal{E}} = \hat{Z}^{\otimes \underline{\varepsilon}}, \tag{2}$$

where $\underline{\varepsilon} \in \mathbb{Z}_2^n$. We define the parity check matrix $\underline{\underline{S}}$, where each stabiliser BSV, $\underline{s}_j$, forms a row of the matrix, so that $\underline{\underline{S}}$ is a $\frac{n-k}{2} \times n$ dimensional matrix.

The length $\frac{n-k}{2}$ syndrome vector y can be found by multiplying the parity check matrix by the error BSV

$$\underline{y} = \underline{\underline{S}} \cdot \underline{\varepsilon}. \tag{3}$$

A common decoding approach is to correct using the maximum likelihood error that gives the same syndrome. For an IID error model, the maximum likelihood error is the lowest weight operator that is consistent with the syndrome.

From the syndrome we employ an inverse syndrome former (ISF) to find a correction that returns us to the code space. The $n \times \frac{n-k}{2}$-dimensional ISF matrix is the pseudoinverse (mod 2) of the parity check matrix, satisfying

$$\underline{\underline{F}}^T \cdot \underline{\underline{S}}^T = \mathbb{I}_{\frac{n-k}{2} \times \frac{n-k}{2}}. \tag{4}$$

That is, each row of the ISF defines an operator that anticommutes with only the corresponding stabiliser. We use $\underline{\underline{F}}$, along with the syndrome to find a *pure error* BSV, $\underline{e} = \underline{\underline{F}}\,\underline{y}$, that satisfies the syndrome, $\underline{y} = \underline{\underline{S}} \cdot \underline{\varepsilon} = \underline{\underline{S}} \cdot \underline{e}$.

Since logical operators and stabilisers commute with all the stabilisers, if we multiply any stabiliser or logical operator with a pure error $\hat{E} = \hat{Z}^{\otimes \underline{e}}$, it will still satisfy the syndrome. Further, the complete set of errors that satisfy the syndrome is generated by the product of all combinations of stabilisers and logical operators with $\hat{E}$, i.e.

$$\underline{e}' = \underline{e} + \sum_{\ell=1}^{\frac{n-k}{2}} \lambda_\ell \underline{s}_\ell + \sum_{m=1}^{k} \mu_m \underline{l}_m \qquad \lambda, \mu \in \mathbb{Z}_2, \tag{5}$$

will also generate the same syndrome for any λ and μ. A maximum likelihood correction, for an IID error model, is found by minimising the hamming weight of e' over λ and μ, i.e.

$$\underline{c} = \arg \min_{\lambda,\mu \in \mathbb{Z}_2} \mathrm{wt}\left[\underline{e}'\right], \tag{6}$$

where wt is the Hamming weight of the vector.

Finding λ and μ that minimise eq 6 is an integer optimisation problem. We implement this optimisation problem using the Gurobi optimisation package [**?**].

IMPLEMENTATION

I'll briefly outline exactly how the optimisation is implemented.

From eq 5 we combine the stabilisers and logicals into a single matrix, sl. Now eq 5 is rewritten

$$\underline{e}' = \underline{e} + \sum_{j=1} x_j \underline{sl}_j. \tag{7}$$

Now relabelling variable to match the code, and going from vectors to index notation,

$$y_i = c_i + \sum_j x_j sl_{i,j}, \tag{8}$$

where c and sl are binary inputs, and the optimisation is findin the minimum of $\sum_i y_i$. As gurobi doesn't like optimising over binary variables, we define y as an integer and add another condition

$$y_i = 2t_i + z_i, \qquad y_i, t_i \in \mathbb{Z} \qquad z_i \in \mathbb{Z}_2. \tag{9}$$

Now z is the binary vector of the optimal correction.

BRANCH AND BOUND IN GUROBI

Gurobi is a commonly used package for solving mixed integer programming (MIP) problems.

To solve integer programming problems is uses the branch and bound method, which I will describe now.

For a general MIP system we have a system, which is generally a matrix of numbers, variables to solve for, conditions on those variables and a function to minimise or maximise.

For the branch and bound method we first relax all integer conditions and find the optimal solution if all the variables were floats. This solution forms a lower bound on our constrained solution. We then find some solution with integer values which forms our best solution so far, for our system this is the pure error.

Then we choose a variable that has a non-integer optimal solution, $c_i = x$ and add an additional condition that $\text{floor}(x) \leq c_i \geq \text{ceil}(x)$. We then solve for each of these new problems.

For the branch and bound method, initially we need an integer solution to the problem, in our case this is the pure error. This solution is defined as the best bound, b. We then relax all integer conditions to create our problem P_0, which is a linear problem, and solve this analytically. The solution to this problem is our optimal solution, o. If this solution has all integer variables then we update our best bound solution b. If $b = o$ then we have found the optimal integer solution, otherwise we continue to construct our *tree* to find a solution.

Next we choose one of the non-integer variables, $c_i = x$ in the the solution P_0 to *branch* on. We then create two new problems P_1 and P_2 where one has the addition condition that $c_i \leq \text{floor}(x)$ and the other $c_i \geq \text{ceil}(x)$. If either of these problems has a solution $o_{1,2} \geq o$ then we update o, similarly if either of the solutions is all integers, $< b$, we update b. Alternatively if one of the optimal solutions $o_{1,2} > b$ then no we know that no better integer solutions exist in that branch, hence we can stop further searches there, *pruning* the branch.

This process is repeated on each non-pruned branch repeatedly until we find a solution such that $o = b$ at which point the process is terminated with a globally optimal solution.

* rjh2608@gmail.com