Why Crawl When You Can Hop?

Applying Simplex-Inspired Techniques Against a Non-Convex Problem

Jonathan Robert Perlstein
jrperl{at}cs.stanford.edu

March 24, 2018

1 Overview

1.1 Document Purpose

This document serves as both the principal summary of my research work completed during the Winter 2018 quarter and a comprehensive overview of the mathematical and programming techniques utilized in the current solution to the Blind Decoding problem. This work follows previous reports authored at the conclusion of the Summer 2017 and Fall 2017 quarters. All of these documents are meant to be read in conjunction with *Blind Joint MIMO Channel Estimation and Decoding* (hereinafter *Blind Decoding*) authored by Thomas Dean, Mary Wootters, and Andrea Goldsmith.

This first section is intended to be administrative for the purposes of documenting my work for the Winter 2018 quarter. The actual research findings begin in Section 2.

1.2 Notation

The following notation is used throughout this document.

- n: The number of antennas utilized in the MIMO channel.
- k: The number of symbols received via the MIMO channel.
- A: The unknown channel gain matrix. This document focuses on real-valued, square channel gain matrices, so $A \in \mathbb{R}^{n \times n}$. For our empirical simulations we draw A such that each entry is `iid` from a standard Gaussian $N(0, 1)$ distribution.
- X: The matrix of transmitted symbols. This document focuses exclusively on Binary Phase-Shift Keying (BPSK), so $X \in \{-1, +1\}^{n \times k}$.
- Y: The matrix of received symbols. This document focuses exclusively on the noiseless case, so $Y = AX$.
- U: The estimate for A^{-1} calculated by our algorithm. The goal of the algorithm is to calculate U such that UY matches X up to an Acceptable Transform Matrix (ATM).
- `ATM`: An Acceptable Transform Matrix is the product of a permutation matrix and a sign matrix. In the context of this problem, we seek to recover X up to an ATM, which is the best outcome we can realistically hope to achieve, and is in fact sufficient from a practical standpoint.[1]

1.3 Prior Work During Summer 2017 and Fall 2017

My research during the past three quarters has focused on the following objective: developing and implementing techniques to solve the Blind Decoding problem using a vertex hopping methodology inspired by the simplex algorithm. During the Summer 2017 and Fall 2017 quarters, I studied multiple aspects of the Blind Decoding problem and the simplex algorithm. I first focused on proving the absence of local optima in the computationally

[1] See *Blind Decoding* for a complete definition of ATM's and their significance to this problem.

challenging $n = 5$ case.[2] After confirming this conjecture through exhaustive computer simulation, I progressed to developing a vertex hopping methodology applicable to the Blind Decoding problem.[3]

At the end of the Fall 2017 quarter, we had completed a small skeleton implementation of the overall program that since has matured into a solver for the Blind Decoding problem. This implementation included the enumeration of the applicable linear constraints and generation of the corresponding simplex-style tableau. We further devised a means of vertex hopping that worked for square matrices, which applies only in the narrow and practically unrealistic case where $k = n$. Also during the Fall 2017 quarter, Tom Dean developed and implemented a novel technique for finding an initial basic feasible solution (BFS) in the context of this problem. The necessity of finding an initial BFS also exists for the classic simplex algorithm, since the origin may not be feasible; however, the technique conventionally used in simplex does not apply to our problem, since we require a BFS that additionally corresponds to a *nonsingular* matrix.[4] As described in Section 3, during the Winter 2018 quarter we substantially optimized the initial BFS finder from both an algorithmic and implementation perspective.

Our initial empirical trials during the Fall 2017 quarter showed that the BFS finder sometimes succeeds entirely in finding a vertex of the original problem space. With a probability that depends on n, k, and other factors, however, the BFS finder can become stuck. The resulting U returned by the BFS finder will be such that UY has $\tilde{k} < k$ columns that are in $\{-1, +1\}^n$, while other columns of UY are in $\{-1, 0, +1\}^n$, and potentially at rational values in $[-1, 1]$ as well.[5] During Fall 2017 we conjectured that we could leverage the same vertex hopping machinery in the reduced problem space provided by the $\tilde{k}$ good columns returned by the initial BFS finder, but we did not have any sort of functional prototype.

At the end of the Fall 2017 Quarter, we had the code infrastructure in place to run the initial BFS finder, to identify the valid columns, and to generate the reduced problem instance. We also had completed the general tableau generation code, to include the ability to pivot for the case where $k = n$. We did not have any means of leveraging the gradient to choose pivots effectively, nor could we support pivoting in the essential instance where the number of signals exceeds the number of antennas (i.e. $k > n$), which as described in Section 6 is in fact essential for accurate signal recovery to be feasible. We additionally did not have a means of determining when we had arrived at a global optimum (see Section 5), which is necessary for $n > 5$ since local optima exist.

Beyond all of the missing algorithmic aspects, we also did not have a realistic implementation. When we finished our prototype code in late February 2018, it was several orders of magnitude too slow to be practically feasible, which required a complete from-scratch code port during the last few weeks of the Winter 2018 quarter as described in Section 7.

1.4 Statement of Collaboration

My work throughout the Winter 2018 quarter was performed in direct and continuous collaboration with Thomas Dean of the Stanford Electrical Engineering Department, who is the principal researcher and architect of the Blind Decoding technique. I also attended biweekly meetings with Professor Mary Wootters of the Stanford Computer Science Department, who provided specific guidance concerning the mathematical underpinnings of this work. For the remainder of this document, I will not draw any distinction between the work for which I was the principal driver and the work primarily done by my colleagues. Since this document serves a secondary role as my statement of work for my Winter 2018 independent research project, I am providing the following rough overview of my responsibilities.

- Primary responsibility for code implementation and optimization in both the prototype and release versions.
- Primary architect of the vertex hopping solution and adaptation of simplex-style techniques to Blind Decoding, with the exception of the initial BFS finding algorithm, where I worked in a secondary role focused on efficient implementation.
- Primary responsibility for generating and running simulation code to verify or disprove mathematical conjectures.
- Secondary role in development of the underlying theory.

[2] Details are contained in my Summer 2017 report.
[3] Details of the initial underlying theory and implementation are contained in my Fall 2017 report.
[4] The gradient in this problem is equal to $(U^{-1})^T$, which can only be evaluated at a nonsingular U. See Section 3 for further detail.
[5] See Section 3.3 for detail concerning this issue.

1.5 Structure of this Report

In the subsequent sections of this document we describe the specific steps we took during the implementation and expansion of our Blind Decoding solver. This includes some amount of software engineering and/or mathematical detail where necessary for understanding. We do not provide substantial detail on topics previously covered in either our research reports or the *Blind Decoding* paper.

In Section 2 we provide a brief problem statement and overview. In Section 3 we describe the process of finding an initial BFS with the particular restrictions imposed by this problem. In Section 4 we detail the process of generating and utilizing a simplex-style tableau to encode the linear constraints present in the Blind Decoding problem, including functionality beyond traditional simplex that is necessary due to the non-convexity of the problem space. Section 5 covers our methodology for determining when we have reached a global optimum, which is nontrivial due to the presence of local optima when $n > 5$. In Section 6 we describe our progress in finding conditions necessary to guarantee that all global maxima of the optimization problem are in fact solutions to the underlying Blind Decoding problem. Section 7 details the algorithmic insights and software engineering optimizations that we have implemented to bring our solver within one order of magnitude of real-time signal decoding in the $n = 8$ case. Section 8 delineates our planned improvements that we expect to close the remaining speed gap for $n = 8$. Section 9 concludes.

2 Problem Statement

2.1 Mathematical Definition

$$\begin{aligned} &\underset{\mathbf{U}}{\text{maximize}} && \log|\det \mathbf{U}| \\ &\text{subject to} && |\mathbf{U}\mathbf{y}_j|_\infty \leq 1,\ j = 1, \ldots, k \end{aligned}$$

The gradient of the objective function has the following closed form: $(U^{-1})^T$.

2.2 Motivation for Using Vertex Hopping

The central insight of our algorithm comprises applying the efficient vertex hopping functionality present in the simplex method to a non-convex problem. The motivation behind attempting a simplex-like solution stems directly from the mathematical definition of the problem itself.[6]

- The problem involves $2kn$ constraints, all of which are linear. The closed form of the objective function gradient is computable in worst case $O(n^3)$ time; furthermore, when U is sparse or structured, faster computation of the gradient may be possible.
- All solutions to the problem lie at vertices of the feasible region. Our early solutions to this problem utilized interior point methods, which require use of a barrier function. Since barrier functions by design repel away from boundaries, interior point methods are slow when the optimum lies on the edge of the feasible region and particularly slow when the optimum is at a vertex. In this document we demonstrate simplex-style vertex hopping to be a plausible and fast alternative.

2.3 Graph of Vertices

At a vertex of the feasible region, we will have $UY = \{-1, +1\}^{n \times k}$. The *Blind Decoding* paper defines the notion of the Maximal Subset Property (MSP),[7] which holds for a matrix $M \in [-1, +1]^{n \times k}$ if there exists some matrix V composed of columns of M such that all of the following hold: $V \in [-1, +1]^{n \times n}$; $cols(V) \subseteq cols(M)$; and $|det(V)| = \max_{W \in [-1,+1]^{n \times n}} |det(W)|$.

Since $det(V)$ is linear in the columns of V, the last criterion above corresponds with the maximum on $\{-1, +1\}^{n \times n}$. We are seeking to maximize $det(U)$ and thus $det(UV)$, and we know that all optima will occur on the boundary

[6] See author's Fall 2017 research report for further detail on the mathematical structure. See the *Blind Decoding* paper for proofs of all claims listed below.

[7] The MSP is a fundamental notion in the Blind Decoding method. The description here is borrowed in substance from *Blind Decoding*; refer to the paper for greater detail.

of the feasible region, and some such optimum will occur at a vertex.[8] We thus focus on vertices of the feasible region, noting that there are 2^{n^2} possible vertices UY where $U \in \mathbb{R}^{n \times n}$ and $Y \in \mathbb{R}^{n \times k}$ for $k \geq n$. If $k > n$ strictly, then the vertex is determined entirely by the first n linearly independent columns of Y. We model the problem space as a graph, where each of the 2^{n^2} matrices represents a vertex. An edge exists between two matrices *iff* they are at Hamming distance 1 from each other, meaning that exactly one entry of UY (among the first n linearly independent columns of Y) is flipped between ± 1. We thus have a graph with 2^{n^2} nodes and $n^2 \cdot 2^{n^2}$ edges.

3 Finding an Initial Basic Feasible Solution

3.1 Why This is Necessary

Like the standard simplex algorithm, our technique requires starting from an initial basic feasible solution (BFS). Unlike simplex, our algorithm possesses an additional restriction: the gradient is only defined at vertices which correspond to *nonsingular* matrices. The technique used in conventional simplex to find an initial BFS when the origin is infeasible thus cannot be applied to our problem. We developed a new methodology for finding an initial BFS that is applicable in this context.

3.2 Algorithm Overview

The algorithm begins by choosing a random point in the feasible region. It then performs up to n^2 iterations of the following step: compute the gradient, project the gradient onto the nullspace of active constraints, and then move along this line to the boundary of the feasible region, thus activating at least one more constraint.[9] We encountered two principal concerns in implementing this technique.

- When $n > 3$ the algorithm can run into a face of the feasible region such that the gradient is orthogonal to the nullspace of active constraints. Since the projection of the gradient is negligible, the algorithm is essentially stuck. When $n = 4, k = 5$ this occurs with probability ≈ 0.5, and for all values $k > n \geq 4$ that we have tested, this has occurred with probability at least 0.5. We address the issue in Section 3.3.
- The projection of the gradient onto the nullspace of active constraints can represent a very expensive operation unless specific optimizations are incorporated. Our initial implementation involved performing the singular value decomposition of the active constraints matrix and then taking the right singular vectors that correspond to negligible singular values. This technique proved to be prohibitively expensive, and as described in Section 3.5 we implemented our own substantially optimized algorithm.

3.3 Finding Sufficient Good Columns / Vertex Hopping In Modified Space

As previously mentioned, the BFS finding algorithm with significant likelihood will become stuck on a face. Since we cannot make further progress with our primary technique of finding a BFS, we switch to a secondary methodology when this situation occurs. Let $cols(\tilde{Y}) \subset cols(Y)$ be the set of columns of Y such that every entry is ± 1, or more formally: $\{y : y \in \{-1, +1\}^n\}$. We temporarily drop the other columns from consideration and create a simplex-style tableau using solely the columns of $\tilde{Y}$. We proceed to vertex hop in this space using the same internal machinery described in Section 4 that we use to solve the full problem once we have an initial BFS, but with the following differences.

- Our goal is to find a BFS of the original problem, not necessarily a globally optimal BFS, though we do preferentially move toward larger objective function values and with very high likelihood find a globally optimal BFS in the process. We fully address finding a global optimum with another complete tableau once we successfully have found a BFS.[10] Thus, we are done with this tableau once we have found a BFS U such that $UY = \{-1, +1\}^{n \times k}$.
- At every hop, we must check that none of the constraints represented by $Y \setminus \tilde{Y}$ are violated; in other words, we must have that $UY \in [-1, +1]^{n \times k}$. For the columns of $\tilde{Y} \subset Y$, this is handled by the linear constraints present in the tableau. The other columns $Y \setminus \tilde{Y}$ must be checked at every hop since the linear constraints

[8] *Blind Decoding* provides a proof.

[9] See author's Fall 2017 research report for further detail.

[10] While the underlying technical internals are very different in our approach, the use of a vertex-hopping algorithm on a slightly modified problem space is also how conventional simplex finds an initial BFS when the origin is not feasible.

in the tableau by construction do not include these columns. By removing these columns from the tableau, we are able to utilize vertex hopping to make further progress; however, we still must respect the boundaries of the feasible region imposed by these constraints.[11]

Once the BFS finder has discovered a vertex of the full problem, we proceed to create a simplex-style tableau reflecting all of the constraints of the full Y matrix and vertex hop to a global optimum using the techniques described throughout the remainder of this document. The procedure from this point is identical regardless of whether we found the initial BFS entirely using our primary technique of moving along the gradient in the nullspace of active constraints or whether we at some point switched to vertex hopping in the space created by the reduced $\tilde{Y} \subset Y$.

3.4 When the BFS Finder Becomes Trapped

3.4.1 Definition

It is possible that the BFS finder will arrive at a vertex of the modified problem space that is a trap. By *trap* we mean that the BFS finder arrived at a face of the original problem but a vertex of the reduced problem (with the problematic $Y \setminus \tilde{Y}$ columns temporarily removed but still checked for feasibility) such that there is nowhere possible to move from this current vertex in the modified problem space to a vertex of the full problem space. The combination of being unable to move through vertices of determinant 0, where the gradient is undefined, with the narrowing of the feasible region imposed by having $k > n$ signals leads to problem geometries where traps are possible. Note that the vertex in the trapped space with the highest objective function value is not a true local optimum; rather, it is a local optimum only in the restricted geometric space created when our solver enforces the feasibility of the dropped constraints.

For illustration purposes consider the case of $n = 5$, where there are three positive determinant values $\{16, 32, 48\}$. Consider the initial BFS finder getting stuck on a constant face of the original problem space where the objective function value is 16. Our technique temporarily ignores the columns of Y that are not entirely $\{-1, +1\}^n$; in this modified space the current point is a vertex. This vertex will have $n^2 = 25$ neighbors, some of which will have either equal or greater objective function value and thus represent legitimate hops (recall that we cannot hop to a vertex of determinant 0). The dropped columns of Y must necessarily be considered when any vertex hop is executed, since due to our dropping this column, we could be executing a hop that renders the corresponding column of UY infeasible. Therefore, not all 25 neighbors will be feasible hops, but some subset will be. We consider our solution to be trapped when we are stuck at a vertex in the modified problem space such that there is no path to a vertex of the original problem space. In other words, we have examined all steps (we use a depth-first-search exploration style) that are both feasible for all columns of the full Y and do not involve stepping to a determinant 0 vertex, and none of these paths have led to a vertex of the full problem.

3.4.2 Solution

The solution to being trapped is quite simple: start the BFS finding algorithm again with a different random initial point. For larger n, which have many intermediate determinant values, it possible to be trapped in a large subspace, and in these instances we will achieve better performance by simply quitting after some finite number of hops, which should depend on n and k. Once our overall implementation is more settled, we will determine the correct limits.

3.4.3 Problem Geometry

The solution proposed above makes sense on one condition: that certain specific problem geometries are not so inherently intractable that restarting is likely to lead into another trap. To test this hypothesis, we conducted many runs of the algorithm, capturing all instances that led to being trapped. We then re-ran each of the trapped instances many times with different random initial points. We discovered that, while some specific geometries appear to be worse than others, the difference does not seem to be prohibitive. We thus have found restarting to be an effective technique.

To explore the effect of k on the likelihood of becoming trapped, we performed the following experiment.

- Set $n = 5$. Begin with a specific 5×6 matrix X that guarantees that all global optima permit recovery of X up to an ATM.

[11] See Section 4.3 for further detail.

- For each value of $k \in \{6, 7, \ldots, 20\}$, perform 1000 repetitions where, for each rep, we randomly fill all extra columns of X with `iid` ± 1 entries, then run the algorithm and determine if we become trapped. The percentage of trapped instances by k is shown in Figure 1. Note that no value of k led to a probability of being trapped greater than 0.06, and while the probability of being trapped initially increases with k, it appears to level off.

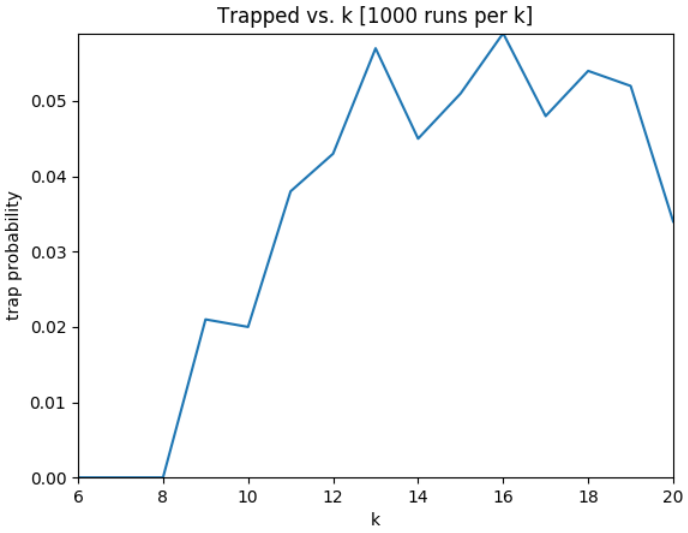

Figure 1: Each run begins with a $\{-1, +1\}^{5\times 6}$ matrix chosen such that recovery up to an ATM is possible. For each run the remaining $k - 6$ columns are filled with `iid` random ± 1 entries. This figure shows the probability of becoming trapped, per value of k.

3.5 Optimizations

We determined that our initial technique of using the SVD to find the nullspace of active constraints incurred an unreasonable performance penalty. Initial simulations with the release version of our program indicated that, for $(n = 8, k = 16)$, over the course of 1000 trials our algorithm spent over 90% of its total running time solely computing the nullspace of active constraints and projecting the gradient. The 90% figure covers the entire algorithm from receipt of the signals Y through recovery of the initial X up to an ATM. In terms of just the initial BFS finder, the projection step required over 95% of the computation time.

We devised and implemented an alternative technique that does not require performing an SVD at each of the n^2 steps. Instead, each time we add a vector to the matrix of active constraints, we then orthogonalize the matrix through a methodology similar to the Gram-Schmidt process. Using the orthogonal basis of active constraints, we then compute the projection of the gradient onto the nullspace of this basis through vector rejection. We have a more complete description of this algorithm in a separate document.

This new technique improved the performance of the projection by a factor between n and n^2. We are continuing to optimize this critical part of the implementation through further improvements to the underlying algorithm as well as software engineering techniques as described in Section 7.

4 Creating and Leveraging a Modified Simplex Tableau

4.1 Structure of the Modified Simplex Tableau

The Blind Decoding problem involves $2kn$ linear constraints. Specifically, we need to maintain that $||UY||_\infty \leq 1$ where $U \in \mathbb{R}^{n\times n}$ and $Y \in \mathbb{R}^{n\times k}$. Consider each individual row $u_i \in U$. We must have: $\forall y_j \in Y : \langle u_i, y_j \rangle \leq 1 \wedge -\langle u_i, y_j \rangle \leq 1$. Since there are k columns of Y and n rows of U, this provides the $2kn$ linear constraints.

We create a simplex-style tableau using a conventional Gauss-Jordan process among each block of $2k$ tableau constraints that correspond to a given row $u_i \in U$. Note that in our implementation, as in conventional simplex implementations, all constraint variables must be nonnegative. An input variable that can take on any real value,

such as the U variables in the Blind Decoding problem, is simulated by using the difference of two nonnegative variables. Formally, the variable $u_{i,j}$ for $i, j \in \{0, \ldots, n-1\}$ becomes two variables in the tableau, which we label as $x_{2(in+j)}$ and $x_{2(in+j)+1}$. Specifically $u_{i,j} = x_{2(in+j)} - x_{2(in+j)+1}$, where $u_{i,j} \in \mathbb{R}$ and all $x_i \in \mathbb{R}^+$. We label these variable as x_i to be consistent with the custom used in traditional simplex for maximization linear programs. This overloaded notation is not to be confused with the X matrix of transmitted symbols that we are trying to recover in the overall problem.

We process tableau constraints two at a time, since each constraint in a pair is simply a mirror image as per the standard linear programming technique used to represent an absolute value constraint. Specifically, one constraint in a pair is $\langle u_i, y_j \rangle \leq 1$ and the other is $-\langle u_i, y_j \rangle \leq 1$. As a general overview, a simplex tableau contains exactly one basic x-variable per constraint, where a basic variable is allowed to take a positive value, while all nonbasic variables must be 0. To achieve this, the simplex algorithm uses a Gauss-Jordan style elimination process, selecting a pivot row for each variable of nonzero value and eliminating that variable from all other rows through adding a multiple of the pivot row. Our tableau generation methodology closely matches that of traditional simplex. When we process a given pair of constraints, we select one of the remaining U variables (that has not already been eliminated) and perform a standard Gauss-Jordan elimination step. We first set the coefficients of the two corresponding x_i to 1 and -1, and we then eliminate these x_i from each of the other $2(k-1)$ constraints ($k-1$ constraint pairs) in the set of $2k$ constraints corresponding to this overall row of U.

We categorize the constraints resulting from this process as being of three different types. To illustrate, below $(\star)$ is an example tableau for $n = 2, k = 3$.

$$
\begin{aligned}
-1.00x_2 + 1.00x_3 + 1.30x_8 - 0.07x_{11} &= 1.23 \\
1.00x_8 + 1.00x_9 &= 2.00 \\
1.00x_{10} + 1.00x_{11} &= 2.00 \\
-1.00x_0 + 1.00x_1 - 0.25x_8 + 0.69x_{11} &= 0.44 \\
1.00x_{12} + 1.00x_{13} &= 2.00 \\
1.00x_8 + 1.00x_{13} &= 2.00 \\
1.00x_4 - 1.00x_5 + 0.25x_{14} + 0.69x_{16} &= 0.93 \\
1.00x_{14} + 1.00x_{15} &= 2.00 \\
-1.00x_6 + 1.00x_7 + 1.30x_{14} + 0.07x_{16} &= 1.38 \\
1.00x_{16} + 1.00x_{17} &= 2.00 \\
1.00x_{18} + 1.00x_{19} &= 2.00 \\
1.00x_{14} + 1.00x_{19} &= 2.00
\end{aligned}
\qquad (\star)
$$

$$
\begin{aligned}
x_0 &= 0.00 & x_1 &= 0.44 & x_2 &= 0.00 & x_3 &= 1.23 \\
x_4 &= 0.93 & x_5 &= 0.00 & x_6 &= 0.00 & x_7 &= 1.38 \\
x_8 &= 0.00 & x_9 &= 2.00 & x_{10} &= 2.00 & x_{11} &= 0.00 \\
x_{12} &= 0.00 & x_{13} &= 2.00 & x_{14} &= 0.00 & x_{15} &= 2.00 \\
x_{16} &= 0.00 & x_{17} &= 2.00 & x_{18} &= 0.00 & x_{19} &= 2.00
\end{aligned}
$$

$$
U = \begin{bmatrix} -0.44 & -1.23 \\ 0.93 & -1.38 \end{bmatrix} \qquad Y = \begin{bmatrix} -0.08 & 1.49 & -0.08 \\ -0.78 & 0.28 & -0.78 \end{bmatrix} \qquad UY = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \end{bmatrix}
$$

The three types of constraints are each described in turn.

4.1.1 Type 1 Constraints

A type 1 constraint is most analogous to a "classic" simplex constraint, where there is exactly one basic variable, which takes the value of the right hand side of the equation. Note that each constraint of this type has both the $x_{2(in+j)}, x_{2(in+j)+1}$ variables corresponding to one specific u-variable, one of which will have coefficient 1 and the other -1. These will be the only x-variables that are derived from variables of U; all other x-variables in

the constraint will be slack variables (this is immediately discernible from the fact that there are $2n^2$ x-variables derived from u-variables, so all such variables for $n = 2$ will have subscripts less than $2n^2 = 8$). All of the slack variables in a constraint of this type will be 0, and exactly one of $x_{2(in+j)}, x_{2(in+j)+1}$ will be nonzero. An example of this type of constraint is $-x_2 + x_3 + 1.30x_8 - 0.07x_{11} = 1.23$. From this constraint we see that the U variable $u_{0,1} = x_2 - x_3 = -1.23$; therefore, as in standard simplex, this type of constraint dictates the value of a specific free variable in the optimization problem. There are n^2 such constraints: one for each value of U.

4.1.2 Type 2 Constraints

A type 2 constraint enforces that the two slack variables corresponding to a single pair of constraints add up to 2. The constraint $x_8 + x_9 = 2$ in $(\star)$ is an example. Note that the underlying mathematical constraint is $|\langle u_0, y_0\rangle| \leq 1$. We then perform the following three steps to linearize this constraint and convert it to a form usable by simplex.

1. Turn the absolute value constraint into two linear constraints.
2. Replace each $u_{i,j}$ by $u_{i,j} = x_{2(in+j)} - x_{2(in+j)+1}$.
3. Add a slack variable to each constraint.

This gives us the following:

$$\begin{aligned}\langle u_0, y_0\rangle + x_8 &= 1\\ -\langle u_0, y_0\rangle + x_9 &= 1\end{aligned}$$

Once we perform all of the Gauss-Jordan elimination steps, we are left with n^2 constraints (n per corresponding row of U) of this second type, which simply enforce the equality immediately derived by adding the two constraints displayed above. This type of constraint also conforms to the standard simplex requirement that each constraint have exactly one basic variable, which will be whichever of the two slack variables is equal to 2, while the other slack variable will be 0 (this holds at any vertex of the feasible region). As explained further below, we actually implement vertex hopping from the perspective of these n^2 constraints, which capture all 2^{n^2} degrees of freedom that we have in choosing a vertex.

4.1.3 Type 3 Constraints

Type 3 constraints exist if and only if $k > n$. These additional symbols impose further constraints on the feasible region, blocking some subset of the 2^{n^2} vertices. The constraint $x_8 + x_{13} = 2$ in $(\star)$ is an example of this type. Notice the distinction between $x_8 + x_{13} = 2$, which is a type 3 constraint, and $x_{12} + x_{13} = 2$, which is a type 2 constraint. x_{12} and x_{13} are slack variables from the opposite equations in the same pair, so $x_{12} + x_{13} = 2$ simply enforces the sum of the two equations corresponding to $|\langle u_0, y_2\rangle| \leq 1$. By contrast, the constraint $x_8 + x_{13} = 2$ enforces that $\langle u_0, y_0\rangle = \langle u_0, y_2\rangle$.

As previously mentioned, when $k > n$ we still have an apparent full 2^{n^2} degrees of freedom in that we can select any of the n^2 entries of UY' (where $cols(Y') \subseteq cols(Y)$ are the first n linearly independent columns of Y) and flip that entry between ± 1. We may discover, however, that performing this flip violates one of the type 3 constraints. This is more clearly illustrated in the tableau excerpt below $(\star\star)$ from a 3×4 case.

$$\begin{aligned}-1.00x_2 + 1.00x_3 - 0.92x_{18} + 11.01x_{20} + 2.31x_{23} &= 12.40\\ 1.00x_{18} + 1.00x_{19} &= 2.00\\ -1.00x_0 + 1.00x_1 - 0.40x_{18} + 1.63x_{20} + 1.10x_{23} &= 2.33\\ 1.00x_{20} + 1.00x_{21} &= 2.00\\ 1.00x_{22} + 1.00x_{23} &= 2.00\\ 1.00x_4 - 1.00x_5 - 0.79x_{18} + 12.86x_{20} + 3.46x_{23} &= 15.53\\ 1.00x_{18} - 1.00x_{20} + 1.00x_{23} + 1.00x_{24} &= 2.00\\ 1.00x_{24} + 1.00x_{25} &= 2.00\\ -1.00x_6 + 1.00x_7 - 0.40x_{26} + 1.63x_{28} - 1.10x_{30} &= 0.12\\ 1.00x_{26} + 1.00x_{27} &= 2.00\end{aligned} \qquad (\star\star)$$

$$U = \begin{bmatrix} -2.33 & -12.40 & 15.53 \\ -0.12 & -7.79 & 8.61 \\ -3.13 & -14.24 & 17.12 \end{bmatrix} \qquad Y = \begin{bmatrix} 1.84 & 0.29 & 0.67 & -0.87 \\ 1.86 & -0.11 & -0.84 & -2.81 \\ 1.83 & 0.02 & -0.64 & -2.44 \end{bmatrix} \qquad UY = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Consider the type 3 constraint: $x_{18} - x_{20} + x_{23} + x_{24} = 2$, which enforces the following: $\langle u_0, y_0 \rangle + \langle u_0, y_3 \rangle = \langle u_0, y_1 \rangle + \langle u_0, y_2 \rangle$. Examining the top row of UY demonstrates that this constraint holds for the given value of U. Note, however, that the constraint imposes requirements on the first row of UY; for example, the row $[1, -1, 1, 1]$ would violate the constraint.

4.2 Vertex Hopping

4.2.1 Technique Overview: $k = n$

To introduce our technique for vertex hopping, we will begin with the simpler case when $k = n$. At a vertex of the feasible region, the current value of U will be such that $UY = \{-1, +1\}^{n \times n}$, signifying that exactly one constraint of every pair of constraints generated by $|\langle u_i, y_j \rangle| \leq 1$ is tight, so the slack variable corresponding to the tight constraint is 0 while the slack variable corresponding to its mirror constraint is 2. Hopping to an adjacent vertex involves flipping exactly one of the values of UY between its two possible extrema $\{-1, +1\}$. To flip the value of $UY_{i,j}$ we must modify each entry of the row u_i such that $\langle u_i, y_j \rangle$ flips between $\{-1, +1\}$ while the other values in the same row of UY, specifically $\langle u_i, y_k \rangle : k \in \{0, \ldots, n-1\}, k \neq j$, remain the same. The simplex tableau form allows us to perform the necessary calculations efficiently.

Consider the tableau shown in $(\star\star\star)$ for an instance where $n = k = 2$. Note that we would never actually hop from the vertex presented in this tableau since it is a global optimum, as are all vertices of nonzero determinant for $n = 2$. We choose this example to maximize simplicity in demonstrating the vertex hopping process, which mechanically is unaffected by the fact that we are hopping away from a global optimum.

$$\begin{aligned} -1.00x_0 + 1.00x_1 + 2.27x_8 - 0.08x_{10} &= 2.20 \\ 1.00x_8 + 1.00x_9 &= 2.00 \\ 1.00x_2 - 1.00x_3 + 1.21x_8 - 0.43x_{10} &= 0.78 \\ 1.00x_{10} + 1.00x_{11} &= 2.00 \\ 1.00x_{12} + 1.00x_{13} &= 2.00 \\ 1.00x_4 - 1.00x_5 + 2.27x_{13} + 0.08x_{14} &= 2.35 \\ -1.00x_6 + 1.00x_7 + 1.21x_{13} + 0.43x_{14} &= 1.64 \\ 1.00x_{14} + 1.00x_{15} &= 2.00 \end{aligned} \qquad (\star\star\star)$$

$$\begin{aligned} x_0 &= 0.00 & x_1 &= 2.20 & x_2 &= 0.78 & x_3 &= 0.00 \\ x_4 &= 2.35 & x_5 &= 0.00 & x_6 &= 0.00 & x_7 &= 1.64 \\ x_8 &= 0.00 & x_9 &= 2.00 & x_{10} &= 0.00 & x_{11} &= 2.00 \\ x_{12} &= 2.00 & x_{13} &= 0.00 & x_{14} &= 0.00 & x_{15} &= 2.00 \end{aligned}$$

$$U = \begin{bmatrix} -2.20 & 0.78 \\ 2.35 & -1.64 \end{bmatrix} \qquad Y = \begin{bmatrix} -0.49 & -1.36 \\ -0.09 & -2.55 \end{bmatrix} \qquad UY = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$$

Focus on the constraint $x_8 + x_9 = 2$. Recall how we generate the tableau, creating slack variables x_8 and x_9 to ensure:

$$\begin{aligned} \langle u_0, y_0 \rangle \leq 1 &\implies \langle u_0, y_0 \rangle + x_8 = 1 \\ -\langle u_0, y_0 \rangle \leq 1 &\implies -\langle u_0, y_0 \rangle + x_9 = 1 \end{aligned}$$

At a vertex, precisely one of x_8 and x_9 will be 0, and the other will be 2. If $\langle u_0, y_0 \rangle = -1$, then $x_8 = 2$; whereas if $\langle u_0, y_0 \rangle = 1$, then $x_9 = 2$. This is reflected above in that $x_8 = 0$, $x_9 = 2$, and $UY_{0,0} = 1$.

If we wanted to flip $UY_{0,0} = \langle u_0, y_0 \rangle$ from 1 to -1, the tableau constraints in the form shown above provide us a ready means of doing so efficiently. This flip will set $x_8 = 2$ and $x_9 = 0$. Now, the two type 1 constraints present in the first set of $kn = 4$ constraints must be modified, since they now have two x-variables with nonzero value, and each constraint must have only one basic variable. To solve this, we can add a multiple of the row $x_8 + x_9 = 2$ (which is a type 2 constraint) to each of the two relevant type 1 constraints.

Consider the constraint $-x_0 + x_1 + 2.27x_8 - 0.08x_{10} = 2.20$. We essentially want to replace the x_8 term with an x_9 term, since x_9 now will be nonbasic and thus can have a nonzero coefficient in a type 1 constraint. We simply multiply the constraint $x_8 + x_9 = 2$ by -2.27 and add the result to the type 1 constraint under consideration. This effectively replaces the x_8 term with an x_9 term, with a coefficient of equal absolute value but opposite sign. Doing so will change the right hand side of this type 1 constraint, and in order to make the equation balance, we will have to modify the value of $-x_0 + x_1$ appropriately. Recalling that $u_{0,0} = x_0 - x_1$, this process demonstrates how we leverage the tableau to compute the values of U that will cause the vertex flip to occur. We repeat the same process with the other type 1 constraint that contains x_8, thus also changing the value of $u_{0,1} = x_2 - x_3$. We then will have a first row of U, denoted u_0, such that $\langle u_0, y_0 \rangle = -1$ and $\langle u_0, y_1 \rangle = 1$, thereby completing the vertex hop by flipping $UY_{0,0}$. The resulting tableau after this flip is shown below in $(\star\star\star')$.

$$
\begin{aligned}
-1.00x_0 + 1.00x_1 - 2.27x_9 - 0.08x_{10} &= -2.35 \qquad (\star\star\star')\\
1.00x_8 + 1.00x_9 &= 2.00\\
1.00x_2 - 1.00x_3 - 1.21x_9 - 0.43x_{10} &= -1.64\\
1.00x_{10} + 1.00x_{11} &= 2.00\\
1.00x_{12} + 1.00x_{13} &= 2.00\\
1.00x_4 - 1.00x_5 + 2.27x_{13} + 0.08x_{14} &= 2.35\\
-1.00x_6 + 1.00x_7 + 1.21x_{13} + 0.43x_{14} &= 1.64\\
1.00x_{14} + 1.00x_{15} &= 2.00
\end{aligned}
$$

$$
\begin{array}{rrrr}
x_0 = 2.35 & x_1 = 0.00 & x_2 = 0.00 & x_3 = 1.64\\
x_4 = 2.35 & x_5 = 0.00 & x_6 = 0.00 & x_7 = 1.64\\
x_8 = 2.00 & x_9 = 0.00 & x_{10} = 0.00 & x_{11} = 2.00\\
x_{12} = 2.00 & x_{13} = 0.00 & x_{14} = 0.00 & x_{15} = 2.00
\end{array}
$$

$$
U = \begin{bmatrix} 2.35 & -1.64 \\ 2.35 & -1.64 \end{bmatrix} \qquad Y = \begin{bmatrix} -0.49 & -1.36 \\ -0.09 & -2.55 \end{bmatrix} \qquad UY = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}
$$

4.2.2 Hopping when $k > n$

When $k > n$ we follow the same general procedure. Recall that when $k > n$, for every row of U, we will have n type 1 constraints, n type 2 constraints, and $2(k - n)$ type 3 constraints. Even though $UY \in \{-1, +1\}^{n \times k}$, we are flipping vertices in a $\{-1, +1\}^{n \times n}$ space defined by UY' where $cols(Y') \subset cols(Y)$ comprises the first n linearly independent columns of Y. We maintain a data structure that tells us, for any of the n^2 possible values of UY', which x-variable we must flip from 0 to 2 in order to flip the corresponding vertex in UY'. The remaining vertices in UY'', where $Y'' = Y \setminus Y'$, are entirely determined by the values of UY' due to linear dependence. As noted in Section 4.1.3, some of the 2^{n^2} possible vertices will be infeasible due to type 3 constraints. Section 4.3 discusses how we recognize infeasible vertices generally, and how we ensure that we reject hops to infeasible vertices.

4.3 Recognizing Infeasible Vertices

As described throughout Section 4.2, regardless of whether $k = n$ or $k > n$, the tableau will contain n^2 type 1 and and n^2 type 2 constraints, which allow hopping among the full set of 2^{n^2} vertices. Some of these 2^{n^2} vertices, however, are infeasible due to one of three possible reasons which we describe here. When we encounter an infeasible vertex, we backtrack as described in Section 4.4.

Vertices with determinant 0 The gradient of our objective function is $(U^{-1})^T$, which is undefined when U is singular. Through tens of thousands of sample runs, we observed no legitimate objective function value less than -10.0, so we consider this to indicate a determinant 0 vertex. Specifically, we have not seen any nonsingular U, where the nonsingularity is legitimate and not simply an artifact of floating point imprecision, such that $ln\,|det(U)| < -10.0$. The values of $ln\,|det(U)|$ that we have observed for truly singular vertices, where the objective function would be $-\infty$ but for floating point imprecision, are usually less than -30.0. We additionally consider the maximum possible jump from the current vertex to any neighboring vertex. In this context we are considering the ratio of the objective function at the current vertex U versus that at a neighboring vertex V on a linear scale, so $|det(V)| : |det(U)|$. We note that for $n \leq 8$ the maximum ratio between any two vertices is $32 : 1$, and our observations have not indicated any pair of neighbors with an objective function ratio anywhere close to this value. We therefore also consider a calculated ratio increase of more than 50.0 to indicate a determinant 0 vertex. We are continuing to work through this latter technique, and our interpretation of the first-order approximation of the gradient in predicting the value at a neighboring vertex may be inaccurate, especially for $n > 5$. The overall heuristic, however, does appear to function properly.

Vertices Rendering Dropped Columns of Y Infeasible This concern only applies when utilizing vertex hopping during the BFS-finding phase, and we presented our methodology for handling this issue in Section 3.3.

Vertices Violating Type 3 Constraints As described in Section 4.1.3, when the tableau contains more than n columns, these extra columns form type 3 constraints, which are reflected directly in the tableau itself. We use a specialized structure for type 3 constraints, which efficiently can verify that any vertex satisfies the underlying equation through arithmetic and comparison operations.

4.4 Backtracking

The general applicability of the traditional simplex technique extends only to linear programs. As emphasized throughout our research, the principal underlying insight of our technique comprises applying the vertex hopping methodology of simplex to a problem that is not only nonlinear but non-convex. While the geometry of our problem does suggest that vertex hopping could be both efficient and effective as described in Section 2, applying the underlying machinery of simplex to a non-convex problem does require substantive additions to the standard algorithm. Crucial among these additions is the ability to backtrack when the algorithm encounters either an infeasible vertex or a local optimum.

As described throughout Section 4.3, our problem does have infeasible vertices, a restriction not present in linear programs. Additionally, for $n > 5$ our problem space includes local optima, which are not present in convex spaces. To allow backtracking our implementation maintains a concept of state, which includes the current value of U and all other data directly dependent on U. We keep this state in its own data structure, and at every hop, we save a snapshot of the current state before jumping to the selected neighboring vertex. This methodology allows us to explore the vertex space in a depth-first-search fashion, following the most likely path to an optimum based on the gradient (see Section 4.5) while maintaining the ability to backtrack when necessary. Minimizing computational and memory cost while supporting the ability to backtrack continues to be one of our software engineering focus areas.

Throughout our experiments with $n \leq 8$, we have noted almost no backtracking during the vertex hopping phase after we have found a BFS. In fact, in almost all instances when $n \leq 8$, the BFS finder arrives immediately at a global optimum with no further vertex hopping whatsoever. The significant application of our vertex hopping and backtracking technique occurs during the BFS finding phase when the primary algorithm becomes stuck and requires vertex hopping to arrive at a BFS. Developing a more sophisticated understanding of how many hops are typically necessary in the BFS finder, and how we can optimize the overall performance, remains a critical area of further research as we improve our algorithm.

4.5 Interpreting the Gradient

The gradient is computed in terms of U, and we must then translate it into terms of the n^2 slack variables corresponding to type 2 constraints, which is what we actually flip to hop among vertices. As previously mentioned, we maintain a data structure, which we term the `vmap`, that has n^2 entries, each of which holds the index of the current slack x-variable which must be moved from 0 to 2 to effect a flip of the corresponding UY' entry. Recall that Y' comprises the first n linearly independent columns of Y, so these are the slack variables corresponding to

the equations $|\langle u_i, y_j \rangle| \leq 1$ for the columns of $Y' \subseteq Y$. Consider the tableau $(\star\star\star\star)$ for an $(n = 2, k = 3)$ instance.

$$
\begin{aligned}
1.00x_8 + 1.00x_9 &= 2.00 \\
1.00x_2 - 1.00x_3 - 0.50x_9 + 1.09x_{10} &= 0.59 \\
-1.00x_0 + 1.00x_1 + 0.16x_9 + 0.04x_{10} &= 0.20 \\
1.00x_{10} + 1.00x_{11} &= 2.00 \\
1.00x_{10} + 1.00x_{12} &= 2.00 \\
1.00x_{12} + 1.00x_{13} &= 2.00 \\
1.00x_{14} + 1.00x_{15} &= 2.00 \\
-1.00x_4 + 1.00x_5 + 0.16x_{15} - 0.04x_{17} &= 0.13 \\
1.00x_{16} + 1.00x_{17} &= 2.00 \\
-1.00x_6 + 1.00x_7 + 0.50x_{15} + 1.09x_{17} &= 1.59 \\
1.00x_{18} + 1.00x_{19} &= 2.00 \\
1.00x_{17} + 1.00x_{19} &= 2.00
\end{aligned}
\qquad (\star\star\star\star)
$$

$$
\begin{array}{cccc}
x_0 = 0.00 & x_1 = 0.20 & x_2 = 0.59 & x_3 = 0.00 \\
x_4 = 0.00 & x_5 = 0.13 & x_6 = 0.00 & x_7 = 1.59 \\
x_8 = 2.00 & x_9 = 0.00 & x_{10} = 0.00 & x_{11} = 2.00 \\
x_{12} = 2.00 & x_{13} = 0.00 & x_{14} = 2.00 & x_{15} = 0.00 \\
x_{16} = 2.00 & x_{17} = 0.00 & x_{18} = 0.00 & x_{19} = 2.00
\end{array}
$$

```
urows:
0 => (2, 1, 0) |  1 => (1, 2, 3)
2 => (7, 5, 4) |  3 => (9, 7, 6)

vmap:
0 =>  9 |  1 => 10
2 => 15 |  3 => 17
```

The first 2 columns are linearly independent, so they contribute to the `vmap`. This specifically means that slack variable pairs $(8, 9)$, and $(10, 11)$ are part of the `vmap`, since these correspond to $|\langle u_0, y_0 \rangle| \leq 1$ and $|\langle u_0, y_1 \rangle| \leq 1$, and the first two columns of Y are linearly independent [and thus comprise Y']. For this same reason, the slack variable pairs $(15, 16)$ and $(17, 18)$, which correspond to $|\langle u_1, y_0 \rangle| \leq 1$ and $|\langle u_1, y_1 \rangle| \leq 1$, are also part of the `vmap`.

Recall from the description of our vertex hopping technique in Section 4.2 that we leverage the tableau to determine the effect that a particular hop will have on each x_i in every affected type 1 constraint. Since $u_{i,j} = x_{2(in+j)} - x_{2(in+j)+1}$, we readily can calculate the effect of any hop on each of the n relevant $u_{i,j}$ [note that any hop affects exactly one row of U]. Since we have a closed form solution for the gradient of U, we can determine the effect of any potential hop by simply multiplying the change in each relevant $u_{i,j}$ by its respective gradient entry. We therefore can compute, with n multiplies and n additions, the effect of any given potential vertex hop on our objective function. During the vertex hopping process, we select the vertex in the direction of maximal positive gradient that has not already been visited, continuing in a depth-first-search manner until a global optimum has been located or a preset hop limit has been exhausted.

5 Global Optima: How Do We Know We Are Done?

If we only examined instances where $k = n$, we would be able to determine when we are at a global optimum readily. When $k = n$, we can simply examine $|det(UY)|$, and at a global optimum the determinant of UY will have the maximal possible value of a $\{-1, +1\}^{n \times n}$ matrix.[12] When $k > n$, determining which columns of Y inherited the maximal subset property from the underlying columns of X quickly becomes computationally prohibitive from

[12] We are assuming that the underlying X possesses the MSP, as is necessary for recovery.

a combinatorial perspective. We therefore need another means of determining when we have arrived at a global optimum, since the actual numerical values of $|det(U)|$ and $|det(UY)|$ will not provide sufficient information. Our methodologies are described below.

5.1 $n = 2$ and $n = 3$

For each of $n = 2$ and $n = 3$, there is only one nonzero determinant value. Therefore, any feasible vertex that has nonzero determinant automatically constitutes a global optimum.

5.2 $n = 4$ and $n = 5$

For $n = 4$ the spectrum of possible determinants is $\{0, \pm 8, \pm 16\}$, and for $n = 5$ the spectrum of possible determinants is $\{0, \pm 16, \pm 32, \pm 48\}$. The existence of intermediate values introduces further complications beyond those present in the trivial $n = 2$ and $n = 3$ cases.

- When $k > n$, the extra constraints imposed by the additional columns can make certain global maxima infeasible, due to either dropped Y column constraints, type 3 constraints, or both.[13] The existence of these constraints means that the actual problem space has de-facto local maxima *in terms of vertices alone, not in terms of the full feasible region.* Recovering from these de-facto local optima is accomplished through backtracking.[14]

For $n = 4$ and $n = 5$, however, there are no *true* local maxima.[15] For this reason, we can always determine whether we have arrived at a global optimum by simply examining the first-order approximation of the gradient toward every neighboring vertex, since at a global optimum the gradient will be negative in all directions.

5.3 General n

The potential values of $|det(U)|$ at a vertex of the feasible region still follow a step-function, even though we do not know which subset of columns of Y inherited the maximal subset property. So while we cannot simply examine $|det(UY)|$, as UY will not even be square when $k > n$, we do know that the values of $|det(U)|$ will follow the same step-function pattern as if we were to know the columns of Y that possess the MSP. For example, the possible determinant values for $\{-1, +1\}^{6\times 6}$ are $\{0, \pm 32, \pm 64, \pm 96, \pm 128, \pm 160\}$. For any U at a *nonsingular* vertex of the feasible region, there are thus five possible values for $|det(U)|$.[16] While the actual numerical values of $|det(U)|$ are themselves dependent on the channel gain matrix A, the gradient $\nabla(U)$, when examined as described in Section 4.5 produces values with interpretable magnitudes, and the spectrum of these magnitudes does *not* depend on A. By examining $\nabla(U)$ we can do the following: for any given neighboring vertex U' of the current vertex U, we can determine the value of $ln(|det(U')|)$ as a proportion of the current objective function $ln(|det(U)|)$.

Through empirical observation we discovered that the gradient at each of the different steps of $ln\,|det(U)|$ is substantially distinct from each other step. In other words, by examining $\nabla(U)$ and calculating the first-order approximation of the effect on the objective function of moving to each of the n^2 possible neighbors, we obtain substantial information about where in the step function we currently are. Consider a set of k received symbols $Y = AX$, and let $cols(Y^*) \subset cols(Y)$ be any subset of columns of Y such that the corresponding columns of X have the MSP. We observed that the n^2 values of $\nabla(U)$ contain substantial information about the current value of $|det(UY^*)|$. This is not surprising: what we are stating is that the underlying space of $\{-1, +1\}^{n\times n}$ matrices, viewed as an undirected graph where the edge (R, S) exists *iff* the Hamming distance between R and S is 1, where the objective function value of a given matrix R is $ln\,|det(R)|$, possesses a substantial amount of geometric structure. While this structure differs for among values of n, and tends to become more complex as n increases, the underlying geometry allows us to determine the current value of $|det(UY^*)|$ with great accuracy at reasonable computational cost. Critically, the geometry pertinent to matrices of *maximal* determinant at every value of $n \in \{2, 3, 4, 5, 6, 8\}$ allows us to precisely and efficiently figure out when we have reached a global optimum.

Fundamental Observation For any given value of n such that $n \leq 8$, the shape of the objective function at any global optimum is identical. For illustration consider first $n = 4$. The maximal determinant value in $\{-1, +1\}^{4\times 4}$ is 16. We are claiming that, for any vertex of determinant ± 16, the set of determinants among all of its neighboring

[13] See Section 4.3 for full detail.
[14] See Section 4.4.
[15] We proved that this conjecture holds for $n = 5$ through computer simulation during the Summer of 2017.
[16] Our objective function is $ln(|det(U)|)$, but since ln is a monotonic function, all of the statements made in this section apply to both.

vertices is identical to that of every other vertex of determinant ± 16. Note for clarity that we are only considering the absolute values of the determinants. Continuing with the example for $n = 4$, for any matrix M such that $|det(M)| = 16$, for all neighboring matrices N of M [so $\texttt{hamming\_dist}(M, N) = 1$], it will hold that $|det(N)| = 8$. In other words, for any global optimum at $n = 4$, which necessarily will have determinant of absolute value 16, every neighboring matrix will have determinant of absolute value 8. We prove in a separate document that whenever there is only one equivalence class of maximal determinant matrices up to an ATM, there is only one geometric pattern of the determinants of neighboring vertices. For $n \leq 10$, there is only one class of maximal determinant matrices. Thus while the patterns of the determinants of neighboring matrices become more complex at intermediate values as n grows, the central aspect of this fundamental observation holds for all $n \leq 10$. We show the relevant patterns for values of $n \leq 8$ in Table 1.

Table 1: Neighbor Determinant Value for Global Optima

n	Opt Value	Neighbor Pattern
2	2	All nonzero are global optima
3	4	All nonzero are global optima
4	16	16 vertices of det 8
5	48	5 vertices of det 32; 20 vertices of det 16
6	160	18 vertices of det 128; 12 vertices of det 96; 6 vertices of det 64
7	Not Evaluated	Not Evaluated
8	4096	64 neighbors of det 3072

Technical Implementation Our algorithm computes the gradient after every vertex hop in order to determine the optimal next hop. As previously mentioned, this calculation provides us the relative value of the objective function at each neighboring vertex compared to the objective function at the current vertex. This information allows us to determine with great [in some instances, total] accuracy whether the current vertex is in fact a global optimum.

Recall that for $n \leq 5$ this issue does not apply since all optima are global. For $n = 6$ the maximal determinant value is 160, and the minimum step value is 32. Since $160/32 = 5$ and 5 is a prime number, we can detect global optima with total accuracy in the $n = 6$ case. When at a global optimum, each step value will be a rational fraction with a denominator value of 5. Since 5 has no nontrivial divisors, we can state that we are at a vertex of determinant 160 if and only if every gradient step is a rational fraction with 5 as the denominator. For $n = 8$ the maximal determinant value is 4096, and at global optima every neighbor has determinant 3072. This instance does not provide the total guarantees that the previous case furnished, since conceivably there could be a local optimum of lesser value such that every one of its 64 neighbors has a determinant precisely 3/4 the determinant of the local optimum. We have no reason to believe that such a vertex exists for $n = 8$, and our empirical observations of the geometry of $\{-1, +1\}^{n \times n}$ matrices for several values of n leads us to conjecture that the existence of such a matrix is unlikely.

We thus are able to leverage the gradient computation to determine with near-total accuracy whether our current vertex is in fact a global optimum. This allows us to define a effective stopping condition even without knowing the subset $cols(Y^*) \subseteq cols(Y)$ for which the underlying $cols(X^*) \subseteq cols(X)$ has the Maximal Subset Property.

6 Guaranteeing Recovery Up To An ATM

We utilized our simulation engine to run large numbers of trials for each set of relevant X matrices that we examined for each value of n. We further developed simple techniques to determine if our algorithm succeeded entirely, meaning it recovered the original X up to an ATM, or succeeded in finding a global optimum without recovering X up to an ATM. We ran many statistical analyses for each value of n to assist in the development and corroboration of theory concerning the properties of X necessary to guarantee that all global optima are in fact solutions to the Blind Decoding problem. We present our work for each value of n separately.

6.1 $n = 2$

For $n = 2$ we use several proofs from the *Blind Decoding* paper.

- If n is such that a Hadamard matrix exists (which is true for $n = 2$), and X has the maximal subset property, then all global optima are of the form $U = QA^{-1}$, where $Q \in \mathcal{O}(n)$.[17]
- All full-rank matrices in $\{-1, +1\}^{2\times 2}$ have the MSP.
- ATM's are the only orthogonal matrices that map among elements of the set of full-rank matrices in $\{-1, +1\}^{2\times 2}$.

We therefore conclude that every global optimum must recover X up to an ATM and thus be a solution to the problem.

6.2 $n = 3$

For $n = 3$ the following must hold to guarantee recovery.

- A collection of $k \geq 4$ samples (columns of X) contains the MSP+1 property if X has the MSP and at least one additional column which is a non-trivial linear combination of the three columns which give X the MSP. The MSP+1 property guarantees recovery up to an ATM for $n = 3$, and an example matrix is shown below.

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

6.3 $n = 4$

To guarantee recovery a matrix must have the Hadamard + 1 property. Note that there are two equivalence classes of Hadamard matrices for $n = 4$, where an equivalence class is defined as the set of Hadamard matrices that are equal to each other up to an ATM.[18] An example matrix is shown below.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 \end{bmatrix}$$

6.4 $n = 5$

There are 16 distinct equivalence classes of optimal matrices for $n = 5$. We have identified all 16 classes, and we have determined two conditions which, if either holds, we can guarantee recovery of X up to an ATM. We summarize these conditions here, and we have a more thorough description in a separate document, from which the following description borrows significantly.

Let $\mathbf{G} = \{-1, +1\}^5$ be the set of all five-dimensional ± 1-valued vectors. Let $\mathbf{G_0} \subset \mathbf{G}$ contain all vectors where each entry has the same sign, and let $\mathbf{G_1} \subset \mathbf{G}$ and $\mathbf{G_2} \subset \mathbf{G}$ contain all vectors where four and three entries have the same sign, respectively. Let $\{\mathcal{H}^{(1)}, \ldots, \mathcal{H}^{(16)}\}$ denote the 16 equivalence classes of maximal determinant matrices. Allow the notation $\mathcal{H}^{(i)} \subseteq cols(\mathbf{X})$ to mean that all columns in $\mathcal{H}^{(i)}$ up to negation are contained in the set of columns of $\mathbf{X}$; further, note that this implies that $\mathbf{X}$ has the MSP. Either of the following two conditions suffices to guarantee that all global optima recover $\mathbf{X}$ up to an ATM.

- $\mathcal{H}^{(i)} \subseteq cols(\mathbf{X})$, and there exists a column $x_k \in cols(\mathbf{X})$ such that $\exists i, j : x_k \notin \mathcal{H}^{(i)} \wedge x_k \in \mathcal{H}^{(j)}$. An example 5×6 matrix follows:

$$\begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & -1 & 1 & 1 \\ -1 & 1 & -1 & -1 & 1 & 1 \\ -1 & 1 & 1 & 1 & -1 & -1 \end{bmatrix}$$

- $\mathcal{H}^{(i)} \subseteq cols(\mathbf{X})$, and there exist two columns $x_i, x_j \in \mathbf{G_1} : x_i, x_j \in \mathbf{X} \wedge x_i \neq \pm x_j$. Since none of the 16 matrices $\mathcal{H}^{(i)}$ contain any columns in $\mathbf{G_1}$, the smallest possible matrix of this sort is 5×7, such as the

[17] This is proven as Lemma 7 in *Blind Decoding*.
[18] This is demonstrated in *Blind Decoding*.

following example:

$$\begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & 1 \\ -1 & -1 & -1 & 1 & 1 & 1 & -1 \\ -1 & -1 & 1 & -1 & 1 & 1 & 1 \\ -1 & 1 & -1 & -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 & -1 & 1 & 1 \end{bmatrix}$$

6.5 $n = 6$

There are 320 distinct equivalence classes of optimal matrices for $n = 6$, and we have identified all of them. We have not identified specific conditions necessary to guarantee recovery up to an ATM, but tens of thousands of empirical runs with $n = 6$ suggest that any reasonable number of extra columns (certainly $k = 12$) renders recovery up to an ATM very likely (see Section 7.4 for empirical results).

6.6 $n = 8$

Hadamard matrices exist for $n = 8$, and through a combination of theoretical analysis and empirical testing, we found 480 equivalence classes of Hadamard matrices. As for $n = 6$ we have not identified specific conditions necessary to guarantee recovery up to an ATM, but as shown in Section 7.4, we have found that realistic values of k do render recovery very likely.

7 Development of Release Version and Code Optimization

7.1 Port to Rust

Over the final few weeks of the Winter 2018 quarter we migrated our code base from Python to Rust. We will work primarily from the Rust code base from this point forward, and this code will eventually become the release version of our software implementation. Due to time and space efficiency requirements and the potential need to run on computationally limited devices, we considered C/C++ and Rust to be our most realistic language options. We chose Rust due to its strong memory safety guarantees. Since Rust can interoperate costlessly with arbitrary C libraries, we can both embed our software into C programs and utilize C code in our software.

7.2 Software Engineering Techniques Implemented

By using a language that allows manual memory allocation [and due to affine typing, essentially manual memory deallocation as well], we can pre-allocate the structures that we need and prevent their continuous allocation and deallocation. We also can systematically eliminate as many unnecessary copies as possible. We have begun taking all of these steps and will continue to do so as our release code matures.

We also are using Rust's internal timing facilities to locate bottlenecks. As an example, we suspected due to asymptotic complexity analysis that the projection onto the nullspace of active constraints would pose our largest initial efficiency bottleneck. Timing measurements confirmed this suspicion, specifically showing its substantial effect for $n = 8$. We utilized this information to prioritize development of a nullspace projection technique that does not require performing any singular value decompositions.

7.3 Algorithmic Optimizations

As described in Section 3.5, the single most important algorithmic optimization we implemented is finding a better technique for projecting the gradient onto the nullspace of active constraints. While we have implemented the initial technique, which has provided tremendous time savings especially in the $n = 8$ case, we have several substantial optimizations remaining to finalize and implement. The matrix of active constraints exhibits significant sparsity. The calculations that we need to perform are quite narrow and specific in scope, and we believe that building a customized sparse matrix library optimized for exactly what we need is our best option. We intend to do this over the next few months.

We plan to optimize our vertex hopping code in several ways. At present we recompute full matrices in every step of the algorithm. Hopping to an adjacent vertex, however, only induces changes in a single row of U; therefore, rank-one updates are feasible for many of the necessary matrix operations. We additionally may be able to avoid performing the calculations to interpret the gradient in terms of every one of the n^2 neighbors at every step, instead

calculating only a limited subset. We believe that these optimizations, when combined with careful software engineering, could yield an $O(n)$ total performance gain.

In Section 3.4 we discussed that, for $n \geq 6$, the initial BFS finder can enter a substantial space with no valid exit. Starting over from a new random initial point obviously incurs significant cost. On the other side, exploring the entire massive space in which the algorithm is trapped, only to later give up (or to exhaust the space entirely) wastes time. Optimizing the projection of the gradient onto the nullspace of active constraints, which represents the vast majority of the cost of restarting from a new initial point, reduces the cost of the restart option. Overall, we need to perform substantial empirical experimentation to determine the optimal number of hops in the expanded space for which we allow the BFS finder to run before simply assuming that we are in fact trapped and starting over from a new random initial U_i.

7.4 Initial Performance Results

Our initial results confirm the expected 100x speedup from the port to Rust combined with the first set of improved algorithms. Table 2 demonstrates the current state of the technique. Note that successes are defined as recovery of the initial matrix X up to an ATM. In the tests summarized in Table 2, we utilized limits in two contexts, both of which realistically will need to exist in production versions:

1. We limited the number of overall attempts that the algorithm is allowed to make, where an attempt is defined as a full run of the algorithm beginning with selecting an initial random feasible point U_i.

2. Within any given attempt, assuming that the initial BFS finder yielded enough $\{\pm 1\}^n$ columns to begin the vertex-hopping process, we limited the number of hops before simply assuming that the algorithm was trapped.

When the latter of these two limits is exhausted, we restart from a new random initial feasible U_i, just as we would if the initial BFS finder had not yielded sufficient $\{\pm 1\}^n$ columns. When the former of these limits is exhausted, we assume that the given symbols cannot be efficiently interpreted, so in an industrial application we would request retransmission of the symbols. Such instances are reflected as `runout` in Table 2. The different result categories in Table 2 are defined as follows.

- `success` is defined as $UY = X$ up to an ATM.

- `runout` is defined as running through all allowed attempts without achieving a global optimum. This is not an error; rather, we never arrive at any vertex of the problem space whatsoever. This usually happens with small k since the BFS finder will then struggle to find n linearly independent good columns.

- $UY = \pm 1$ is defined as recovering a U such that all $n \times k$ entries of UY are ± 1, but $UY \neq_{ATM} X$. This can result from X not having the MSP, or from X having the MSP but not the extra conditions required to ensure recovery up to an ATM, or potentially from an error in our algorithm (the first two possibilities are not erroneous results). As described in Section 5, for $n \leq 5$ we have determined the criteria necessary to recover X up to an ATM. When we utilize an X matrix that does guarantee recovery, if we find a U that produces $UY = \{-1, +1\}^{n \times k}$ that we believe to be a global optimum but $UY \neq_{ATM} X$, we count this an an `error`, not as $UY = \pm 1$.

- `error` is defined as an explicit error or anything that falls into none of the above categories and thus is an unexpected result.

Table 2 demonstrates an apparent anomaly where the algorithm runs faster for larger values of k. This is most apparent in the performance difference between $(n = 8, k = 20)$ and $(n = 8, k = 28)$. Recall that for our overall solver to work, the BFS finder must return some point U such that there are n linearly independent columns of UY that are composed of entirely ± 1 entries. Without n linearly independent $\{-1, +1\}^n$ columns, we cannot transition to vertex hopping in the modified space defined by these columns, and we must start over with new random initial U_i. With $n = 8$ especially, the overhead imposed by the extra restarts when k is smaller outweighs the overhead imposed by the computations over the larger data structures when k is larger. Table 2 shows that for $n = 8$ the performance gain levels off, and if we were to test larger k we would encounter the expected performance decrease as the overhead of extra columns overtakes any savings generated by fewer restarts and fewer feasible vertices.

Table 2: Performance for Various Values of n, k

n	k	attempts	success	runout	$UY = \pm 1$	error	mean seconds / attempt
2	3	1000	1000	0	0	0	3.20×10^{-5}
3	6	1000	1000	0	0	0	9.86×10^{-5}
4	6	1000	999	1	0	0	3.47×10^{-4}
4	8	1000	996	4	0	0	4.64×10^{-4}
5	9	1000	998	1	1	0	1.71×10^{-3}
5	12	1000	1000	0	0	0	1.78×10^{-3}
5	15	1000	1000	0	0	0	1.83×10^{-3}
6	12	1000	991	4	5	0	7.51×10^{-3}
6	18	1000	993	7	0	0	6.47×10^{-3}
6	24	1000	998	1	1	0	7.10×10^{-3}
8	20	1000	945	17	38	0	1.09×10^{-1}
8	28	1000	991	5	4	0	6.32×10^{-2}
8	36	1000	996	4	0	0	5.66×10^{-2}

8 Future Work

Optimizations and Extensions We will continue to develop optimization techniques both in terms of algorithmic approaches and software engineering methodologies.

- Building on our current improvements in projecting the gradient onto the nullspace of active constraints, we plan to implement a block-matrix technique that will permit more efficient storage and computation and may leverage multithreading.
- To increase the efficiency of the vertex-hopping code, which is becoming increasingly important as we improve the initial BFS finder, we plan to implement efficient updates to the gradient at each vertex hop. Generally we will implement rank-one updates where applicable for data structures used in the vertex hopping portion of the algorithm.
- We plan to continue improving our memory management throughout the entire algorithm, in addition to improving general systems implementation throughout our code.
- We will eliminate unnecessary computations that cannot be used prior to recomputation of the underlying variables.
- We will experiment with multithreading in several contexts, including the BFS finder.

We intend these optimizations to expand the scope of realistic applications for our algorithm. If we are able to develop a sufficiently fast algorithm for $n = 8$, we may explore larger values of n to determine if our algorithm can handle these much larger problem spaces. The ratio of global optima to overall optima decreases drastically in this neighborhood of n-values; therefore, interior-point type methods become infeasible from a theoretical perspective as well as a computational perspective. Our algorithm, by contrast, may prove to be feasible for $n > 8$.

Conference and Journal Papers We plan to write at least one journal paper concerning this work, documenting both the theoretical and implementation aspects of our technique in either a single work or two separate works. We also plan to submit a conference paper in the near future.

9 Conclusion

In this document we have described our motivation for exploring a vertex hopping solution to the Blind Decoding problem, which involves decoding wireless MIMO signals in the absence of channel state information. The central theme of our work comprises combining algorithm design and software engineering techniques to adapt the principles of the simplex algorithm to the non-convex geometry presented by this problem. We described our customized technique for finding a BFS under the substantial restrictions that are not present in a linear program. We detailed our tableau construction and vertex hopping methods, both of which require departures from conventional simplex,

to include state maintenance and backtracking. We demonstrated our method for distinguishing global optima from local optima accurately and efficiently. We further described how we ensure, in instances where we have the knowledge needed to do so, that solutions to the optimization problem are in fact solutions to the underlying decoding problem. We outlined our recent port of the entire codebase to Rust, a fast low-level systems language that, with the incorporation of several major algorithm improvements, has allowed us to achieve nearly practical real-time signal processing for $n = 8$. We further outlined several planned optimizations that likely will make real-time processing for $n = 8$ possible.