A genomic tree based sparse solver

Timothy A. Davis and Srinivas Subramanian

Abstract This article is concerned with improving an existing application of compressive sensing to metagenomics — the Quikr method, where nonnegative sparse recovery is performed using the Lawson Hanson algorithm. To enhance the computational speed of this algorithm, we offer GETS: a GEnomic Tree based Sparse solver. We exploit the inherent structure of the genomic problem to uncover an evolutionary family tree type relationship between the species. This genomic tree enables us to obtain a sparse representation of the problem which is created in the offline stage of GETS, a one time computation. This allows for reduced storage and asymptotic speed ups for our solver via sparse matrix computations. We conclude the article with the results of computational experiments performed with genomic datasets. These experiments illustrate the significant speed ups obtained by GETS over MATLAB's implementation of the Lawson Hanson algorithm.

1 Introduction

Metagenomics is the study of microbial communities from the content of their sequenced DNA or RNA. Compressive sensing techniques for metagenomics were developed in [5] and [6]. The Quikr method of [5] is a k-mer based approach that solves the problem of reconstruction of concentrations of bacterial communities from an environmental sample. Prior methods worked by classifying samples in a read-by-read fashion which is time consuming for large datasets. Compressive sensing techniques enable a quick classification of an entire dataset simultaneously, thus

Timothy A. Davis
Texas A&M University, College Station
e-mail: davis@tamu.edu

Srinivas Subramanian
Texas A&M University, College Station
e-mail: srini2092@tamu.edu

achieving an order of magnitude reduction in the execution time. Essentially, Quikr measures the frequency of k-mers in a database of 16S rRNA genes for known bacteria, calculates the frequency of k-mers in the given sample, and then reconstructs the concentrations of the bacteria in the sample by solving an underdetermined system of linear equations under a sparsity assumption. This method was extended in the WGSQuikr method of [6] to enable the accurate analysis of very large whole-genome shotgun datasets. Both methods utilize a novel technique for sparse recovery by solving an ℓ_1-squared nonnegative regularization problem. The article [3] showed that this optimization problem can be recast as a conventional nonnegative least squares problem, which can be solved using the Lawson Hanson algorithm [7, Chapter 23], an iterative active set algorithm which is well suited for a fast implementation.

Improvements to the Lawson Hanson algorithm were proposed specifically for use in Chemometric applications in [1] and [8]. The ideas are mainly applicable to problems involving matrices which have a much larger number of rows than columns. This is in contrast to the compressive sensing setting of the genomic problem that we consider here, which involves a measurement matrix with the number of columns being much larger than the number of rows. For instance, [1] involves precomputing and storing the product of the transpose of a matrix with itself, for use in each iteration of the algorithm. This would be prohibitively expensive for the dimensions involved in the genomic problem. Likewise, in [8], the idea of grouping and computing common pseudoinverses is only applicable for problems with a large number of right hand sides and utilizes a combinatorial approach that might only be suitable for matrices with a few columns.

We introduce GETS: a GEnomic Tree based Sparse solver, which improves the computational speed of the Lawson Hanson algorithm by exploiting the inherent structure of the genomic problem. This is primarily done by establishing an evolutionary family tree type relationship between the columns of the k-mer matrix of DNA sequences that allows for a terse representation of the data. This representation is created in the offline stage of GETS, a one time computation which enables reduced storage and allows for large asymptotic speed ups whenever the solver is used. The problem setting, the optimization formulations and the algorithm are discussed in Section 2. An overview of the ideas to speed up the algorithm are provided in Section 3, while the details are covered in Sections 4 and 5. GETS is primarily written in C, with an interface for use in MATLAB through its MEX functions. The computational experiments described in Section 6 illustrate the significant speed improvements obtained by GETS over MATLAB's implementation of the Lawson Hanson algorithm. These experiments may be reproduced by running the MATLAB scripts present in the GETS package [1].

[1] GETS can downloaded from the second author's GitHub page: https://github.com/SrinivasSubra.

2 Solving the problem via Lawson Hanson algorithm

2.1 Problem setting

Consider a database of 16S rRNA gene sequences for a number of N known bacterial organisms. For a fixed integer k, the number of occurrences of DNA subwords of length k in each gene sequence are measured to obtain a matrix $\mathbf{A}$ of k-mer counts. Its columns are indexed by organisms in the database and its rows are indexed by all 4^k possible DNA subwords of length k . The $(i, j)^{\text{th}}$ entry of $\mathbf{A}$ counts the number of occurrence of the i^{th} subword in the DNA sequence of the j^{th} organism. For instance, in one of our datasets, we have an integer matrix of 6-mer counts for a database of 188,326 genomes. With $k = 6$, the integer matrix $\mathbf{A} \in \mathbb{Z}^{m\times N}$ has $m = 4^k = 4096$ rows, and $N = 188{,}326$ columns. By normalizing the columns of $\mathbf{A}$, we obtain a frequency matrix $\mathbf{C}$ whose columns sum to one. It has nonnegative entries $\mathbf{C}_{i,j} \geq 0$ with $\sum_{i=1}^{m} \mathbf{C}_{i,j} = 1$, for all $j = 1, \ldots, m$. Given a sample of 16S rRNA reads, the frequency of all k-mers is calculated to obtain a vector $\mathbf{y}$, the sample k-mer frequency vector.

The problem of reconstructing the concentrations of bacterial communities from a given sample is expressed in the form of a compressive sensing problem. The k-mer frequency matrix of DNA sequences is the measurement matrix $\mathbf{C} \in \mathbb{R}^{m\times N}$. The sample k-mer frequency vector obtained from the DNA reads is the measurement vector $\mathbf{y} \in \mathbb{R}^m$. The composition of the bacterial sample is represented by a nonnegative vector $\overline{\mathbf{x}} \in \mathbb{R}^N$, where $\overline{\mathbf{x}}_j$ is the concentration of j^{th} organism in the database. Here $\overline{\mathbf{x}}$ is assumed to be sparse, since there are expected to be relatively few different organisms in a given sample. The number of measurements $m = 4^k$ is much smaller than N, the number of organisms in the database. Therefore, the problem entails the recovery of sparse nonnegative $\overline{\mathbf{x}}$ from the underdetermined system $\mathbf{y} = \mathbf{C}\overline{\mathbf{x}}$.

2.2 Nonnegative least squares (NNLS)

Since $\mathbf{C}$ is a frequency matrix and $\overline{\mathbf{x}}$ is nonnegative, recovery can be performed by solving the nonnegative least squares problem (see [3])

$$\min_{\mathbf{v}\in\mathbb{R}^N} \|\mathbf{y} - \mathbf{C}\mathbf{v}\|_2 \quad \text{subject to } \mathbf{v} \geq \mathbf{0} \tag{NNLS}$$

via the active set algorithm of Lawson and Hanson. Throughout this article, we use the following notation:

- $\mathbf{z}_{\mathcal{P}}$ is the restriction of a vector $\mathbf{z}$ to an index set $\mathcal{P}$. In MATLAB notation, $\mathbf{z}_{\mathcal{P}} = \mathbf{z}(\mathcal{P})$;
- $\mathbf{C}_{\mathcal{P}}$ is the column submatrix of a matrix $\mathbf{C}$ consisting of the columns indexed by $\mathcal{P}$. In MATLAB notation, $\mathbf{C}_{\mathcal{P}} = \mathbf{C}(:, \mathcal{P})$;
- $\mathbf{C}_{\mathcal{P}}\mathbf{z}_{\mathcal{P}} \cong \mathbf{y}$ denotes the least squares problem $\min \|\mathbf{y} - \mathbf{C}_{\mathcal{P}}\mathbf{u}\|_2$ with the solution at $\mathbf{u} = \mathbf{z}_{\mathcal{P}}$.

The Lawson Hanson algorithm (Algorithm 1) takes as input a matrix $\mathbf{C} \in \mathbb{R}^{m \times N}$, a vector $\mathbf{y} \in \mathbb{R}^m$, and a tolerance $\epsilon \in \mathbb{R}$ for the stopping criterion.

Algorithm 1 Lawson Hanson NNLS($\mathbf{C}, \mathbf{y}, \epsilon$)

1: **Initialize:** $\mathcal{P} := \emptyset$, $\mathcal{R} := \{1, 2, \ldots, n\}$, $\mathbf{x} := \mathbf{0}$, $\mathbf{z} := \mathbf{0}$, $\mathbf{w} := \mathbf{C}^\mathsf{T}\mathbf{y}$.
2: **while** $\mathcal{R} \neq \emptyset$ and $\max(\mathbf{w}_\mathcal{R}) > \epsilon$ **do**
3: Find an index $t \in \mathcal{R}$ such that $\mathbf{w}_t = \max(\mathbf{w}_\mathcal{R})$
4: Move t from $\mathcal{R}$ to $\mathcal{P}$
5: Compute $\mathbf{z}_\mathcal{P}$ as the solution to the least squares problem $\mathbf{C}_\mathcal{P}\mathbf{z}_\mathcal{P} \cong \mathbf{y}$
6: Set $\mathbf{z}_\mathcal{R} := \mathbf{0}$
7: **while** $\min(\mathbf{z}_\mathcal{P}) \leq 0$ **do**
8: Set $\alpha := \min \left\{ \dfrac{x_i}{x_i - z_i} : z_i \leq 0, i \in \mathcal{P} \right\}$
9: Set $\mathbf{x} = \mathbf{x} + \alpha(\mathbf{z} - \mathbf{x})$
10: Move from $\mathcal{P}$ to $\mathcal{R}$ all indices $j \in \mathcal{P}$ for which $x_j < \epsilon$
11: Compute $\mathbf{z}_\mathcal{P}$ as solution to the least squares problem $\mathbf{C}_\mathcal{P}\mathbf{z}_\mathcal{P} \cong \mathbf{y}$
12: Set $\mathbf{z}_\mathcal{R} := \mathbf{0}$
13: **end while**
14: Set $\mathbf{x} := \mathbf{z}$
15: Set $\mathbf{w} := \mathbf{C}^\mathsf{T}(\mathbf{y} - \mathbf{C}\mathbf{x})$
16: **end while**

Index sets $\mathcal{P}$ and $\mathcal{R}$ are modified in the course of the algorithm. The set $\mathcal{P}$ will be the support of vectors $\mathbf{x}$ and $\mathbf{z}$ at the end of each iteration of the main loop. The vectors $\mathbf{w}$ and $\mathbf{z}$ provide working space. On termination, $\mathbf{x}$ will be the solution vector and $\mathbf{w}$ will be the dual vector satisfying the Karush–Kuhn–Tucker (KKT) optimality conditions for (NNLS), namely

$$\mathbf{x}_\mathcal{P} > \mathbf{0}, \quad \mathbf{x}_\mathcal{R} = \mathbf{0}, \quad \mathbf{w}_\mathcal{P} = \mathbf{0}, \quad \mathbf{w}_\mathcal{R} \leq \mathbf{0}.$$

The inner loop at step 7 is entered if any of the components of $\mathbf{z}_\mathcal{P}$ become negative after the least squares problem at step 5. It serves as a course correction by updating $\mathbf{x}$ using a convex combination of the current and previous iterate, until the least squares problem at step 11 yields a nonnegative solution. In our implementation, the vectors $\mathbf{x}$ and $\mathbf{z}$ are stored as sparse vectors. The index sets $\mathcal{P}$ and $\mathcal{R}$, which are complements of one another, are stored in a binary indication vector of length N. The set $\mathcal{P}$ is also implicitly available as the support of $\mathbf{x}$.

2.3 ℓ_1-squared nonnegative regularization (NNREG)

In practice, we need to account for error in the measurement because the acquisition of the vector $\overline{\mathbf{x}}$ is not perfect. The measurement vector $\mathbf{y} = \mathbf{C}\overline{\mathbf{x}} + \mathbf{e}$ now contains a noise term $\mathbf{e} \in \mathbb{R}^m$. There are several strategies possible for the approximate recovery of $\overline{\mathbf{x}}$, typically involving an optimization problem with a nonnegative constraint. As demonstrated in [3], a novel technique is to solve an ℓ_1-squared nonnegative

regularization problem, namely

$$\min_{\mathbf{v}\in\mathbb{R}^N} \|\mathbf{v}\|_1^2 + \lambda^2\|\mathbf{y} - \mathbf{C}\mathbf{v}\|_2^2 \quad \text{subject to } \mathbf{v} \geq \mathbf{0} \tag{NNREG}$$

for some large $\lambda > 0$, a regularization parameter. The squared ℓ_1-norm along with the nonnegative constraint enables a recasting of this problem into a nonnegative least squares problem. Indeed, with $\mathbf{v} \geq 0$, we have $\|\mathbf{v}\|_1 = \sum_{j=1}^N v_j$. This observation allows (NNREG) to be written as

$$\min_{\mathbf{v}\in\mathbb{R}^N} \|\widetilde{\mathbf{y}} - \widetilde{\mathbf{C}}\mathbf{v}\|_2^2 \quad \text{subject to } \mathbf{v} \geq \mathbf{0} \tag{$\widetilde{\text{NNLS}}$}$$

where $\widetilde{\mathbf{y}} \in \mathbb{R}^{m+1}$ and $\widetilde{\mathbf{C}} \in \mathbb{R}^{(m+1)\times N}$ are defined as

$$\widetilde{\mathbf{y}} = \left[\frac{\lambda\mathbf{y}}{0}\right] \quad \text{and} \quad \widetilde{\mathbf{C}} = \left[\frac{\lambda\mathbf{C}}{1 \ \ldots \ 1}\right].$$

Therefore, we can use the Lawson Hanson algorithm to solve (NNREG), since ($\widetilde{\text{NNLS}}$) is equivalent to (NNLS) with ($\mathbf{C}$, $\mathbf{y}$) replaced with ($\widetilde{\mathbf{C}}$, $\widetilde{\mathbf{y}}$). For our implementation in GETS, it is useful to reformulate the algorithm computations involving the matrix $\widetilde{\mathbf{C}}$ in terms of computations involving the matrix $\mathbf{C}$. With vectors $\mathbf{x} \in \mathbb{R}^N$, $\widetilde{\mathbf{r}} \in \mathbb{R}^{m+1}$, and defining $\mathbf{r} = \widetilde{\mathbf{r}}(1:m)$, we observe that

$$\widetilde{\mathbf{C}}\mathbf{x} = \left[\frac{\lambda\mathbf{C}\mathbf{x}}{\sum_{j=1}^N x_j}\right] \quad \text{and} \quad \widetilde{\mathbf{C}}^\mathsf{T}\widetilde{\mathbf{r}} = \left[\lambda\mathbf{C}^\mathsf{T}\middle|\begin{matrix}1\\ \vdots \\ 1\end{matrix}\right]\left[\frac{\mathbf{r}}{\widetilde{r}_{m+1}}\right] = \lambda\mathbf{C}^\mathsf{T}\mathbf{r} + \widetilde{r}_{m+1}\begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix}.$$

Then, setting $\widetilde{\mathbf{r}} = \widetilde{\mathbf{y}} - \widetilde{\mathbf{C}}\mathbf{x}$ allows us to compute $\mathbf{w} = \widetilde{\mathbf{C}}^\mathsf{T}(\widetilde{\mathbf{y}} - \widetilde{\mathbf{C}}\mathbf{x})$, since

$$\widetilde{\mathbf{r}} = \widetilde{\mathbf{y}} - \widetilde{\mathbf{C}}\mathbf{x} = \left[\frac{\lambda(\mathbf{y} - \mathbf{C}\mathbf{x})}{-\sum_{j=1}^N x_j}\right] \quad \text{yields} \quad \mathbf{w} = \widetilde{\mathbf{C}}^\mathsf{T}\widetilde{\mathbf{r}} = \lambda^2\mathbf{C}^\mathsf{T}(\mathbf{y} - \mathbf{C}\mathbf{x}) - \sum_{j=1}^N x_j\begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix}.$$

For the least squares step, we solve the problem

$$\widetilde{\mathbf{C}}_\mathcal{P}\mathbf{z}_\mathcal{P} = \left[\frac{\lambda\mathbf{C}_\mathcal{P}}{1 \ \ldots \ 1}\right]\mathbf{z}_\mathcal{P} \cong \left[\frac{\lambda\mathbf{y}}{0}\right] = \widetilde{\mathbf{y}}$$

by forming the submatrix $\widetilde{\mathbf{C}}_\mathcal{P}$ on the fly. Therefore, we don't need to store or work directly with the matrix $\widetilde{\mathbf{C}}$ (in fact, in the subsequent sections we will see that we don't actually need to store the matrix $\mathbf{C}$ either). Since the implementation of (NNREG) is a straightforward extension of (NNLS), in the rest of this article, we will describe the conceptual details with reference to (NNLS) only. The computational experiments for solving (NNREG) problems are described in Section 6.

3 Exploiting the structure of the problem

In MATLAB, the Lawson Hanson algorithm is implemented as the function `lsqnonneg`. However, this implementation is not efficient for our purpose, since it does not exploit the structure of the problem. In our implementation in GETS, we exploit the algorithm structure along with the genetics of the problem. In terms of time consumed during each iteration of the algorithm, the following steps are significant:

- Computing $\mathbf{Cx}$;
- Solving $\mathbf{C}_{\mathcal{P}}\mathbf{z}_{\mathcal{P}} \cong \mathbf{y}$;
- Computing $\mathbf{C}^{\mathsf{T}}\mathbf{r}$, where the residual $\mathbf{r} = \mathbf{y} - \mathbf{Cx}$.

In this section, we provide an overview of tackling these aspects of the problem. We begin with the basic observation that at each iteration of the algorithm, the variable vector $\mathbf{x}$ is sparse. Then, the computation $\mathbf{Cx}$ is quite simple, since $\mathbf{Cx} = \mathbf{C}_{\mathcal{P}}\mathbf{x}_{\mathcal{P}}$. This is essentially equivalent to a dense matrix-vector multiplication problem involving small dimensions, since the submatrix $\mathbf{C}_{\mathcal{P}}$ is of size $m \times s$, where s denotes the sparsity of $\mathbf{x}$ in the current iteration. Thus the complexity is just $O(ms)$ flops instead of $O(mN)$ flops which is required in a full dense matrix-vector product.

Next, we consider the least squares problem $\mathbf{C}_{\mathcal{P}}\mathbf{z}_{\mathcal{P}} \cong \mathbf{y}$ in step 5. This is an overdetermined least squares problem which normally requires $O(ms^2)$ flops. The Lawson Hanson algorithm is an active set algorithm where each iteration of the outer loop involves updating the support $\mathcal{P}$ by adding an index to it. The submatrix $\mathbf{C}_{\mathcal{P}}$ is thus updated by the addition of a column. We exploit this fact to utilize a QR update method which speeds up the least squares computations by bringing down the complexity to $O(ms)$ flops (Note: If the inner loop is entered, which does not happen often in practice, one or more columns of $\mathbf{C}_{\mathcal{P}}$ are removed in step 10. Since we keep track of the columns of $\mathbf{C}_{\mathcal{P}}$, a QR update is always possible in the outer loop. See Section 5).

The main focus of this article is in tackling the major computational bottleneck — computing $\mathbf{C}^{\mathsf{T}}\mathbf{r}$, a dense matrix-vector product of complexity $O(mN)$ flops, since the residual $\mathbf{r}$ is dense. We consider the following question:

How can we exploit the genetics of the problem to speed up the computations?

The matrix $\mathbf{C}$ was originally obtained after column normalizing an integer matrix $\mathbf{A}$, whose $(i, j)^{\text{th}}$ entry counts the number of occurrence of the i^{th} subword in the DNA sequence of the j^{th} organism. This suggests that columns corresponding to genetically similar organisms would not be very different. We establish an evolutionary family tree type relationship between the columns of the integer matrix $\mathbf{A}$ that allows for a sparse representation of the data. This representation is created in the offline stage, which is a one time computation. This enables a compact storage of the data and allows for large asymptotic speed ups in the computation of $\mathbf{C}^{\mathsf{T}}\mathbf{r}$ during each iteration of the algorithm.

Thus for the given data of a k-mer matrix $\mathbf{A}$ of DNA sequences, we obtain a significant overall speedup of the Lawson Hanson algorithm every time the solver is

called to recover any sparse vector $\overline{\mathbf{x}}$ from the measurement vector $\mathbf{y} = \mathbf{C}\overline{\mathbf{x}}$ as input. The ideas discussed above are explored in the following sections.

4 Genomic tree based computations

4.1 Offline stage

Let us consider a similarity matrix $\mathbf{\Delta} \in \mathbb{Z}^{N\times N}$ to quantify the differences in the columns of $\mathbf{A}$. Define $\mathbf{\Delta}(i,j) := \text{nnz}\left(\mathbf{A}(:,i) - \mathbf{A}(:,j)\right)$, where we use the MATLAB notation $\mathbf{A}(:,i)$ to denote the i^{th} column of $\mathbf{A}$ and $\text{nnz}(\mathbf{M})$ to denote the number of non-zeros in a matrix $\mathbf{M}$. We compute the minimum spanning tree (MST) on the graph of this matrix with edge weights $\mathbf{\Delta}(i,j)$. Since the graph of $\mathbf{\Delta}$ would be dense, we use Prim's algorithm to compute the MST. We use the adjacency matrix representation of the graph for the implementation. We do not actually store $\mathbf{\Delta}$, a large matrix requiring significant memory because N can be large. Instead, we compute the edge weights $\mathbf{\Delta}(i,j)$ implicitly, on the fly, during the course of Prim's algorithm. Computing an edge weight $\mathbf{\Delta}(i,j)$ takes $O(m)$ flops, therefore the entire computational complexity for the MST would be $O(N^2m)$ flops.

The MST gives us a tree of relationships between the organisms. The tree can be represented by a length N `parent` array. If $\texttt{parent}(i) = j$ in the tree, we would expect $\mathbf{\Delta}(i,j)$ to be small due to the genetic similarities between the organisms. Since the MST relies on genetic similarities, we can consider it to be a genomic tree somewhat reminiscent to a phylogenetic tree. In this article, we are only concerned with the computational benefits offered by this tree. The MST enables the creation of a sparse difference matrix which is formed by taking the differences between the columns of a child and its parent in the tree. Let $\mathbf{D} \in \mathbb{Z}^{m\times N}$ denote the difference matrix. Then $\mathbf{D}$ is defined as $\mathbf{D}(:,i) = \mathbf{A}(:,i) - \mathbf{A}(:,j)$, where $j = \texttt{parent}(i)$ for all nodes i (except for the root node which has no parent). Thus $\mathbf{D}$ would be a sparse matrix as the parent and child columns would differ only in few places. The sparseness of $\mathbf{D}$ was numerically confirmed for our genomic datasets (see Section 4.3). Before delving into the remaining offline computations, we first look at how this $\mathbf{D}$ matrix can be used to compute $\mathbf{C}^\mathsf{T}\mathbf{r}$ in $O(\text{nnz}(\mathbf{D}))$ flops.

4.2 Fast computation of $\mathbf{C}^\mathsf{T}\mathbf{r}$

We make use of the difference matrix $\mathbf{D}$ in computing $\mathbf{C}^\mathsf{T}\mathbf{r}$, where $\mathbf{r}$ is the residual. Let **diag** denote a diagonal matrix consisting of the reciprocals of the column sums of the matrix $\mathbf{A}$. As the matrix $\mathbf{C}$ was obtained after column normalizing $\mathbf{A}$, we have $\mathbf{C} = \mathbf{A} * \mathbf{diag}$ and $\mathbf{C}^\mathsf{T}\mathbf{r} = (\mathbf{A} * \mathbf{diag})^\mathsf{T}\mathbf{r} = \mathbf{diag} * (\mathbf{A}^\mathsf{T}\mathbf{r})$. Therefore, we only require the integer matrix $\mathbf{A}$ for the computations. We will first compute $\mathbf{w} = \mathbf{A}^\mathsf{T}\mathbf{r}$ and then scale $\mathbf{w}$ using **diag** afterwards. For each i, we need to compute $\mathbf{w}(i) = \mathbf{A}(:,i)^\mathsf{T}\mathbf{r}$.

Since $\mathbf{D}(:,i) = \mathbf{A}(:,i) - \mathbf{A}(:,j)$ and $j = \texttt{parent}(i)$, we have

$$\begin{aligned} \mathbf{w}(i) &= \big(\mathbf{D}(:,i) + \mathbf{A}(:,j)\big)^{\mathsf{T}}\mathbf{r}, \qquad \text{(TREEMULT)} \\ &= \mathbf{A}(:,j)^{\mathsf{T}}\mathbf{r} + \mathbf{D}(:,i)^{\mathsf{T}}\mathbf{r}, \\ &= \mathbf{w}(j) + \mathbf{D}(:,i)^{\mathsf{T}}\mathbf{r}. \end{aligned}$$

As the columns of $\mathbf{D}$ are sparse, we can compute $\mathbf{D}(:,i)^{\mathsf{T}}\mathbf{r}$ very quickly in $\text{nnz}(\mathbf{D}(:,i))$ flops. This allows us to obtain an asymptotic speed up in computing $\mathbf{C}^{\mathsf{T}}\mathbf{r}$ — the complexity reduces from $O(mN)$ to $O(\text{nnz}(\mathbf{D}))$ flops. The speed up is proportional to the sparsity of the difference matrix $\mathbf{D}$. Any spanning tree would give a difference matrix that can be used to compute $\mathbf{C}^{\mathsf{T}}\mathbf{r}$, but a MST would yield the sparsest difference matrix and hence the largest speed up.

In (TREEMULT), we need to compute the parent value $\mathbf{w}(j)$ prior to computing the child value $\mathbf{w}(i)$. In order to access a parent node and compute its $\mathbf{w}$ value before reaching its child node in an efficient manner, we make use of a topological ordering of the MST. A topological ordering of the tree ensures that the parent node j occurs before node i in the ordering. We obtain a topological ordering in the offline stage of our implementation, by performing a reverse post ordering of the MST via depth first search. The reverse post ordering is represented by a `post` array. One can apply `post` on a new node of the reordered tree to obtain the old node. To do the reverse, one can apply `invpost`, the inverse of the `post` array, on the old node. The parent-child relationships remain the same, so we can think of the reordered tree as the old tree with the nodes renumbered by applying `invpost`. In this reordered tree, the new 'parent' array `ipt` is given by `ipt = invpost(parent(post))`. We work with the reordered tree throughout the Lawson Hanson algorithm. For instance, $\mathbf{A}$ is now replaced by $\mathbf{A}(:,\texttt{post})$, its column reordered version. Likewise, we work with the reordered versions of $\mathbf{D}$ and **diag**. Computing $\mathbf{w}(i)$ in (TREEMULT) then becomes $\mathbf{w}(i) = \mathbf{w}(\texttt{ipt}(i)) + \mathbf{D}(:,i)^{\mathsf{T}}\mathbf{r}$. The index i can be simply be varied from 2 to N in a loop without issue, since the topological ordering ensures that $\texttt{ipt}(i) < i$ (the root node at $i = 1$ has no parent). All the reordering related computations are performed in the offline stage. If the original indices of the solution are needed, one can apply the `post` array on the support of the solution. That is, with $\mathcal{P}$ as the support of the solution vector $\mathbf{x}$ upon termination, the original support is given by $\texttt{post}(\mathcal{P})$.

4.3 Compact data representation

In the C programming language, the size of the datatype `double` is 64 bits, `short` is 16 bits, and `char` is 8 bits. The normalized matrix $\mathbf{C}$, which is stored in `double` format, is expensive for memory access in comparison with storing only an integer matrix $\mathbf{A}$. In our large dataset, we have a k-mer matrix $\mathbf{A}_{\text{large}} \in \mathbb{Z}^{m\times n}$ of dimensions $m = 4096$ and $n = 188{,}326$. We observed that $\mathbf{A}_{\text{large}}$ has entries no bigger than 50, hence we store this as a `char` matrix. Storing $\mathbf{C}$ requires 8 times more memory space

than storing **A**. Therefore, we never work directly with the matrix **C** in the entire algorithm. Instead, we only store the matrix **A** and scale on the fly. For instance, for the computation **Cx**, we compute $\mathbf{A}(\mathbf{diag} * \mathbf{x})$. We just need to scale the entries of the sparse vector **x** with **diag** before performing a dense matrix times sparse vector operation. The computation of $\mathbf{C}^{\mathsf{T}}\mathbf{r}$ and the least squares problem using only **A** and **diag**, without **C**, are noted in the corresponding sections.

The expected sparseness of the difference matrix **D** corresponding to the large dataset was confirmed numerically. We found that it comprised approximately 93% zeros, 6.5% binary non-zeros with +1s and -1s, and 0.5% non-binary values ranging between -40 and +40. We exploit this distribution of values by splitting **D** into a sum of three sparse matrices, $\mathbf{D} = \mathbf{D}_{+1} + \mathbf{D}_{-1} + \mathbf{D}_{\mathrm{nb}}$. The non-zero values of **D** are split into +1s in $\mathbf{D}_{+1}$, -1s in $\mathbf{D}_{-1}$, and non-binary values in $\mathbf{D}_{\mathrm{nb}}$. The benefit of this lies in even more compact storage — for the sparse matrices $\mathbf{D}_{+1}$ and $\mathbf{D}_{-1}$ we don't need to store the numerical values, we only need to store the location of the binary entries as opposed to both location and values in a *compressed column form* [2]. The dot products $\mathbf{D}_{+1}^{\mathsf{T}}\mathbf{r}$ and $\mathbf{D}_{-1}^{\mathsf{T}}\mathbf{r}$ require only addition operations. We store the entries of the very sparse $\mathbf{D}_{\mathrm{nb}}$ matrix as a `char` array. We create a data structure `treedata`, a C `struct`, for this purpose that encapsulates all the data required for the solver — the post ordering of the tree, parent array of the reordered tree, reordered **A**, **diag**, and the sparse components of **D**. The offline stage of GETS involves taking the original k-mer matrix **A** as input and performing all the required one time computations to generate this compact data representation. The output `treedata` thus generated is saved so that it is always available to be used as an input for the solver. This enables the fast computations previously discussed thereby providing significant speedups to the Lawson Hanson algorithm whenever the solver is used.

The C `struct` declaration of `treedata` is provided below to illustrate the discussion here.

```
typedef struct gets_treedata_struct
{
    csi m; /* number of rows */
    csi n; /* number of columns */
    csi *bposp; /* column pointers for binary +1 entries of D */
    csi *bnegp; /* column pointers for binary -1 entries of D */
    short *bposi; /* row indices for binary +1 entries of D */
    short *bnegi; /* row indices for binary -1 entries of D */
    csi *xp; /* column pointers for non-binary entries of D */
    short *xi; /* row indices for non-binary entries of D */
    char *xx; /* numerical values for non-binary entries of D */
    char *A; /* reordered unnormalized integer matrix A */
    csi *diag; /* column sums of reordered A */
    csi *ipt; /* parent array in the reordered tree */
    csi *post; /* post ordering of the MST */
} treedata ;
```

GETS uses the CSparse package [2] for implementing sparse vectors and utilizing certain functions. The `csi` datatype from CSparse is an integer datatype of size 64 bits. In `treedata`, the integers `m` and `n` are the dimensions of the integer k-mer matrix $\mathbf{A}$. This m-by-n dense matrix is represented by the `char` array `A` (after its columns are reordered). The `csi` arrays `diag`, `ipt` and `post` are all of size n. The array `diag` stores the column sums of $\mathbf{A}$, which are used for the previously mentioned scaling operations done by **diag**. The arrays `ipt` and `post` are as defined in Section 4.2. The sparse m-by-n matrix $\mathbf{D}_{\text{nb}}$ is essentially represented in *compressed column form* by the arrays `xp`, `xi` and `xx`. The array `xi`, of type `short`, contains the row indices of non-zero entries. Its size therefore equals $\text{nnz}(\mathbf{D}_{\text{nb}})$, the number of non-zero entries in $\mathbf{D}_{\text{nb}}$, and its values will be less than or equal to m. The array `xx`, of the same size and of type `char`, contains the numerical values of the non-zero entries in $\mathbf{D}_{\text{nb}}$. The array `xp`, of size n+1, contains the column pointers for the *compressed column form* such that $\texttt{xp}(j)+1$ is the location in the `xi` and `xx` arrays of the first non-zero entry of $\mathbf{D}_{\text{nb}}$ in column j. Row indices of entries in column j are stored in $\texttt{xi}(\texttt{xp}(j)+1)$ through $\texttt{xi}(\texttt{xp}(j+1))$ for $j = 1:n$, and the corresponding numerical values are stored in the same location in `xx`. The first entry $\texttt{xp}(1)$ is 0, while the last entry $\texttt{xp}(n+1)$ is $\text{nnz}(\mathbf{D}_{\text{nb}})$. Likewise, `bposp` and `bposi` are the arrays of column pointers and row indices used to represent the sparse matrix $\mathbf{D}_{+1}$, while the arrays `bnegp` and `bnegi` correspond to the sparse matrix $\mathbf{D}_{-1}$. As mentioned before, we do not require any arrays to store the numerical values of $\mathbf{D}_{+1}$ and $\mathbf{D}_{-1}$. (Note: The C programming language actually uses *zero-based* indexing, but the above descriptions are *one-based* for the sake of consistency).

To solve larger sized problems, we might need to use bigger sized datatypes than discussed above. The values in the arrays of row indices `bposi`, `bnegi` and `xi` can range from 1 to $m = 4^k$. They are declared as `short` type in `treedata`, since we assume that $m < 2^{15}$. This would work for k-mers in the range $k = 1, \ldots, 7$. For 8-mers or larger, a datatype of size bigger than `short` would need to be chosen accordingly. For instance, declaring the arrays of row indices as 32 bit integers would work for k-mers up to $k = 15$. Further, `treedata` assumes that the values in the matrices $\mathbf{A}$ and $\mathbf{D}_{\text{nb}}$ are in the range -2^7 to $2^7 - 1$, since the arrays `A` and `xx` are declared as `char`. As discussed earlier, this was indeed the case for our datasets. If this is not the case for some given dataset, the datatype can be changed to `short` or a bigger sized integer datatype in the offline stage, according to the range of values in $\mathbf{A}$ and $\mathbf{D}_{\text{nb}}$.

5 Solving least squares using QR updates

We use Householder based QR updates for speeding up the least squares problem $\mathbf{C}_{\mathcal{P}}\mathbf{z}_{\mathcal{P}} \cong \mathbf{y}$ in step 5. Throughout the algorithm, the submatrix $\mathbf{C}_{\mathcal{P}}$ is maintained in the form of a 'VR matrix', an economical representation of its QR factorization. Let us assume that $\mathcal{P}$ contains s indices, so that $\mathbf{C}_{\mathcal{P}} \in \mathbb{R}^{m \times s}$. Let $\mathbf{H}_1, \ldots, \mathbf{H}_s$ be the Householder matrices that have previously been computed (implicitly), so

that if $\mathbf{Q} = \mathbf{H}_1 \ldots \mathbf{H}_s$ then $\mathbf{Q}^\mathsf{T}\mathbf{C}_\mathcal{P} = \mathbf{R}$ is upper triangular. The matrix $\mathbf{Q}$ and the Householder matrices $\mathbf{H}_j = \mathbf{I} - \beta_j \mathbf{v}_j \mathbf{v}_j^\mathsf{T}$ are never explicitly formed, but are represented by the Householder vectors $\mathbf{v}_j$ and the corresponding scalars β_j. The 'VR matrix' is a factored form representation of the QR factorization that stores the matrix $\mathbf{R}$ in its the upper triangular part, and the essential parts of the s Householder vectors in its lower triangular part. Let the array C denote the 'VR matrix', of the same dimensions as the submatrix $\mathbf{C}_\mathcal{P}$. Then its upper triangular part $C(1:s, 1:s) = \mathbf{R}$, while $C(j+1:m, j)$ houses the essential part of the j^{th} Householder vector $\mathbf{v}_j$, for $j = 1:s$. This is enabled by the use of [4, Algorithm 5.1.1] which creates a Householder vector v that is normalized so that $v(1) = 1$ does not need to be stored — we only need to store the remaining part of v, its essential part. Therefore, in order to apply the j^{th} Householder vector $\mathbf{v}_j$ on some vector $\mathbf{c}$, we make use $C(j+1:m, j)$ and β_j to overwrite $\mathbf{c}(j:m)$ with $\mathbf{H}_j\mathbf{c}(j:m)$.

When a new index is added to $\mathcal{P}$, we need to update the submatrix by a new column. Since we never actually store the normalized $\mathbf{C}$ matrix, we access the corresponding column of $\mathbf{A}$ and normalize it on the fly using the corresponding **diag** value to obtain $C(:, s+1) = \mathbf{C}_\mathcal{P}(:, s+1)$. We perform the QR updates by first applying all the previous Householder vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$ to the new column $C(:, s+1)$. Next, we compute the Householder vector $\mathbf{v}_{s+1}$ and β_{s+1} for $C(s+1:m, s+1)$. This is done such that $C(s+2:m, s+1)$ is overwritten with the essential part of $\mathbf{v}_{s+1}$, while $C(1:s+1, s+1) = \mathbf{R}(1:s+1, s+1)$. Lastly, we apply the new Householder vector to update the right hand side vector. If the current right hand side vector is denoted by $\mathbf{b}$ (initially $\mathbf{b} = \mathbf{y}$), we apply Householder vector $\mathbf{v}_{s+1}$ on $\mathbf{b}(s+1:m)$. With this, the QR update steps for the new index are complete and the 'VR matrix' form is thus maintained for subsequent use. Since $C(1:s, 1:s) = \mathbf{R}$, the least squares solution for $\mathbf{C}_\mathcal{P}\mathbf{z}_\mathcal{P} \cong \mathbf{y}$ is obtained by simply solving the upper triangular system $\mathbf{R}\mathbf{z}_\mathcal{P} = \mathbf{b}(1:s)$. This method allows us to solve the least squares problem in step 5 with reduced complexity in $O(ms)$ flops. Without QR updates, the problem would require $O(ms^2)$ flops in each iteration.

If the inner loop is entered in step 7, which is an infrequent occurrence, one or more indices in $\mathcal{P}$ will be removed prior to solving the least squares problem in step 11. Since several columns could potentially be removed, we do a full QR factorization from scratch by using successive QR updates. Thus the 'VR matrix' form of $\mathbf{C}_\mathcal{P}$ is always maintained, and regardless of whether the inner loop is entered, step 5 in the outer loop is always solved using QR updates.

Remark 1 The outer loop condition of step 2 ensures that a new column added to $\mathbf{C}_\mathcal{P}$ will never be redundant. Indeed, once the condition of step 2 is satisfied, the outer loop continues and the new index t is chosen in step 3. Therefore, the new column satisfies $\mathbf{C}(:, t)^\mathsf{T}(\mathbf{y} - \mathbf{C}\mathbf{x}) > \epsilon$. Moreover, after the least squares step in the previous iteration, the residual $\mathbf{y} - \mathbf{C}\mathbf{x}$ will be orthogonal to the column space of $\mathbf{C}_\mathcal{P}$. This implies that $\mathbf{C}(:, t)$ cannot lie in the column space of $\mathbf{C}_\mathcal{P}$. Thus $\mathbf{C}_\mathcal{P}$ will always have full rank and the least squares problem will never be rank deficient.

6 Computational experiments

The datasets used for our numerical experiments pertain to two different databases of 16S rRNA gene sequences for known bacteria. We were provided with integer k-mer matrices (for a fixed $k \sim 6$) with $m = 4^k$ rows and N columns, and several sample k-mer frequency vectors of size m. Specifically, the dataset consists of:

- $\mathbf{A}_{\text{small}}$, an integer matrix of 6-mer counts for a database of 10,046 genomes with dimensions $m = 4096$ and $N = 10{,}046$. This corresponds to the "small database" used in the experiments of Quikr [5].
- $\mathbf{A}_{\text{large}}$, an integer matrix of 6-mer counts for a database of 188,326 genomes with dimensions $m = 4096$ and $N = 188{,}326$. This was derived from the "large database" used in the experiments of Quikr.
- A total of 211 different sample 6-mer frequency vectors $\mathbf{y}_i$, which are the measurement vectors of size $m = 4096$.

6.1 Offline stage

The terse representations of the small and large datasets, $\mathbf{D}_{\text{small}}$ and $\mathbf{D}_{\text{large}}$, were created in the offline stage. Given an integer k-mer matrix $\mathbf{A}$, GETS performs the required one time computations to generate the compact representation for the problem in the form of the `treedata` data structure. For the small dataset, with $\mathbf{A}_{\text{small}}$ as input, GETS generated $\mathbf{D}_{\text{small}}$ in a runtime of about 2 minutes. Correspondingly, for the large dataset, with $\mathbf{A}_{\text{large}}$ as input, $\mathbf{D}_{\text{large}}$ was generated in about 8 hours. They were saved to be available as inputs for the GETS solver for the experiments described below.

6.2 Speed comparison of solvers

To test how fast our solver performs, we compared GETS with MATLAB's `lsqnonneg`. We found that GETS is about 15 times faster than `lsqnonneg` for the large dataset, and about 6 times faster for the small dataset. The experiments were performed on laptop with 16GB memory and a 2.6 GHz 6-Core Intel Core i7 processor.

In our experiments, we solved 211 problems each for the small and large datasets. In each problem, given the measurement vector $\mathbf{y}_i = \mathbf{C}\overline{\mathbf{x}_i} + \mathbf{e}_i$, we seek to approximately recover the vector $\overline{\mathbf{x}_i}$, where $i = 1 : 211$. The vector $\mathbf{e}_i$ indicates the unknown measurement error. The matrix $\mathbf{C} \in \mathbb{R}^{m \times N}$ is the normalized version of the integer matrix $\mathbf{A}$, with column sums equal to one. The experiments were performed for the small and large datasets with $\mathbf{A} = \mathbf{A}_{\text{small}}, \mathbf{A}_{\text{large}}$. For each problem, the solution was obtained by solving (NNREG) via ($\widetilde{\text{NNLS}}$) as described in Section 2.3. For (NNREG), smaller the value of λ chosen, the sparser the reconstruction obtained.

The larger it is, the more closely the k-mer counts will be fit. Our 16S rRNA datasets, $\mathbf{A}_{\text{small}}$ and $\mathbf{A}_{\text{large}}$, were obtained from the Quikr datasets of [5]. Therefore, we used their choice of $\lambda = 10,000$ as the regularization parameter, since it worked well for the Quikr experiments. Changing the dimensions of the problem, for instance for problems involving k-mer matrices with larger values of k, will most likely require a change in the selection of λ. In that case, depending on the inputs, a procedure such as the one used in [6] may be considered — WGSQuikr uses an adaptive procedure which consists of running (NNREG) multiple times with different parameters.

Let $\mathbf{x}_G$ and $\mathbf{x}_M$ denote the solutions generated by GETS and MATLAB's `lsqnonneg` respectively. We compared $\|\mathbf{x}_G - \mathbf{x}_M\|_2$, the error in ℓ_2 norm between the two solutions. The maximum ℓ_2 error was found to be 2.59158e-14 for the small dataset and 3.90415e-14 for the large dataset. The solutions $\mathbf{x}_G$ and $\mathbf{x}_M$ were found to have the same support in every experiment. As expected, both MATLAB and GETS generated the same solutions within tolerance.

For the small dataset, the average time taken by MATLAB to solve a problem was 2.2446 seconds, while GETS took 0.3667 seconds. Correspondingly, for the large dataset, the MATLAB average was 66.7561 seconds, while the GETS average was just 4.3911 seconds. To illustrate the speed up for a specific problem in the large dataset, consider the problem where both GETS and MATLAB took the maximum time to solve it. MATLAB solved the problem in 229.1092 seconds, while GETS took just 16.2336 seconds.

For each problem we considered the speed up ratio, which is defined as the ratio of the time taken by MATLAB divided by the time taken by GETS. The histogram plots of the speed up ratios for the problems are provided in Figure 1. For the small dataset, the mean speed up ratio was 5.9647, while the median speed up ratio was 5.9125. For the large dataset, the mean speed up ratio was 15.2900, while the median speed up ratio was 15.3311. Thus GETS provides a significant speed improvement in both cases, with the large dataset exhibiting a greater speed up. We believe that this is mainly due to the larger dimensions involved and the sparser representation in $\mathbf{D}_{\text{large}}$ (detailed in Section 4.3). As discussed in Section 4.2, the sparser the representation, the greater the asymptotic speed up obtained in computing $\mathbf{C}^{\mathsf{T}}\mathbf{r}$ — the major computational bottleneck of the Lawson Hanson algorithm.

Acknowledgements We thank Simon Foucart for introducing the problem and for valuable discussions and suggestions. We also thank David Koslicki for providing the genomic data that was used in the computational experiments.

References

1. Bro, R., De Jong, S.: A fast non-negativity-constrained least squares algorithm. Journal of Chemometrics **11**, 393-401 (1997)
2. Davis, T. A.: Direct methods for sparse linear systems. Society for Industrial and Applied Mathematics (2006)

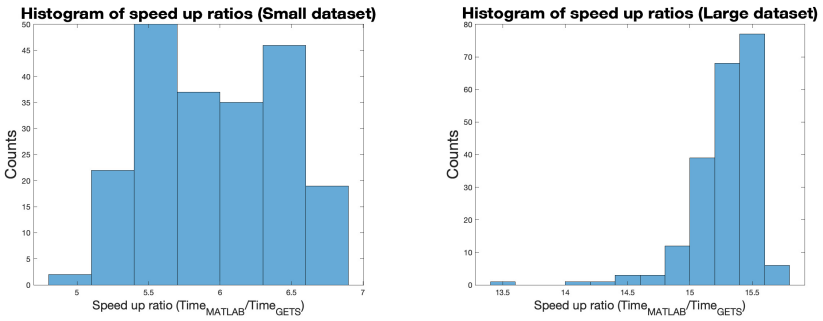

Fig. 1 Histogram of speed up ratios for the problems of small and large datasets. A total of 211 problems were solved in each case.

3. Foucart, S., Koslicki, D.: Sparse recovery by means of nonnegative least squares. IEEE Signal Processing Letters **21**, 498-502 (2014)
4. Golub, G. H., Van Loan, C. F.: Matrix computations. John Hopkins University Press (2013)
5. Koslicki, D., Foucart, S., Rosen, G.: Quikr: a method for rapid reconstruction of bacterial communities via compressive sensing. Bioinformatics **29**, 2096-2102 (2013)
6. Koslicki, D., Foucart, S., Rosen, G.: WGSQuikr: fast whole-genome shotgun metagenomic classification. PLoS ONE **9**, e91784 (2014)
7. Lawson, C. L., Hanson, R. J.: Solving least squares problems. Prentice-Hall Series in Automatic Computation (1974)
8. Van Benthem, M. H., Keenan, M. R.: Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems. Journal of Chemometrics **18**, 441-450 (2004)