# The Mailman algorithm: a note on matrix vector multiplication*

## Yale university technical report #1402.

Edo Liberty
Computer Science
Yale University
New Haven, CT

Steven W. Zucker
Computer Science and Appled Mathematics
Yale University
New Haven, CT

**Abstract**

Given an $m \times n$ matrix $A$ we are interested in applying it to a real vector $x \in \mathbb{R}^n$ in less then the straightforward $O(mn)$ time. For an exact, deterministic computation at the very least all entrees in $A$ must be accessed, requiring $O(mn)$ operations and matching the running time of naively applying $A$ to $x$. However, we claim that if the matrix contains only a constant number of distinct values, then reading the matrix once in $O(mn)$ steps is sufficient to preprocess it such that any subsequent application to vectors requires only $O(mn/\log(max\{m,n\}))$ operations. Theoretically our algorithm can improve on recent results for dimensionality reduction and practically it is useful (faster) even for small matrices.

## Introduction

A classical result of Winograd ([1]) shows that general matrix-vector multiplication requires $\Omega(mn)$ operations, which matches the running time of the naive algorithm. However, since matrix-vector multiplication is such a common, basic operation, an enormous amount of effort has been put into exploiting the special structure of matrices to accelerate it. For example, Fourier, Walsh Hadamard, Toeplitz, Vandermonde, and others can be applied to vectors in $O(npolylog(n))$.

Others have focussed on matrix-matrix multiplication. For two $n \times n$ binary matrices the historical *Four Russians Algorithm* [2] (modified in [3]) gives a log factor improvement over the näive algorithm, i.e, running

*Research supported by AFOSR and NGA

time of $n^3/log(n)$. Techniques for saving log factors in real valued matrix-matrix multiplications were also found. These include Santoro and Urrutia [4] and Savage [5]. These methods are reported to be practically more efficient then naive implementations. Classic theoretical results by Strassen [6] and Coppersmith and Winograd [7] achieve better results then the above mentioned but are slower then the naive implementation for small and moderately sized matrices. The above methods, however, do not extend to matrix-vector multiplication.

We return to matrix-vector operations. Since every entry of the matrix, $A$, must be accessed at least once, we consider a preprocessing stage which takes $\Omega(mn)$ operations. After the preprocessing stage $x$ is given and we seek an algorithm to produce the product $Ax$ as fast as possible. Within this framework Williams [8] showed that an $n \times n$ binary matrix can be preprocessed in time $O(n^{2+\epsilon})$ and subsequently applied to *binary vectors* in $O(n^2/\epsilon \log^2 n)$. Williams also extends his result to matrix operations over finite semirings. However, to the best of the authors' understanding, his technique cannot be extended to real vectors.

In this manuscript we claim that any $m \times n$ matrix over a *finite alphabet*[1] can be preprocessed in time $O(mn)$ such that it can be applied to any *real* vector $x \in \mathbb{R}^n$ in $O(mn/\log(max\{m,n\}))$ operations. Such operations are common, for example, in spectral algorithms for unweighted graphs, nearest neighbor searches, dimensionality reduction, and compressed sensing. Our algorithm also achieves all previous results (excluding that of Williams) while using a strikingly simple approach.

## The Mailman algorithm

Intuitively, our algorithm multiplies $A$ by $x$ in a manner that brings to mind the way a mailman distributes letters, first sorting the letters by house and then delivering them. Recalling the identity $Ax = \sum_{i=1}^{n} A^{(i)}x(i)$, metaphorically one can think of each column $A^{(i)}$ as indicating one "address" or "house", and each entry $x(i)$ as a letter addressed to it. To continue the metaphor, imagine that computing and adding the term $A^{(i)}x(i)$ to the sum is equivalent to the effort of walking to house $A^{(i)}$ and delivering $x(i)$. From this perspective, the naive algorithm functions by delivering each letter individually to its address, regardless of how far the walk is or if other letters are going to the same address. Actual mailmen, of course, know much better. First, they arrange their letters according to the shortest route (which includes all houses) without moving; then they walk the route, visiting each house regardless of how many letters should be delivered to it (possibly none).

[1] $A(i,j) \in \Sigma$, and $|\Sigma|$ is a finite constant.

To extend this idea to matrix-vector multiplication, our algorithm decomposes $A$ into two matrices, $P$ and $U$, such that $A = UP$. $P$ is the "address-letter" correspondence matrix. Applying $P$ to $x$ is analogous to arranging the letters. $U$ is the matrix containing all possible columns in $A$. Applying $U$ to $(Px)$ is analogous to walking the route. Hence, we name our algorithm after the age-old wisdom of the men and women of the postal service.

Our idea can be easily described for an $m \times n$ matrix $A$, where $m = \log_2(n)$ (w.l.o.g assume $\log_2(n)$ is an integer) and $A(i,j) \in \{0,1\}$. (Later we shall modify it to include other sizes and possible entrees.) There are precisely $2^m = n$ possible columns of the matrix $A$, by construction, since each of the $m$ column entrees can be only 0 or 1. Now, define the matrix $U_n$ as the matrix containing *all* possible columns over $\{0,1\}$ of length $\log(n)$. By definition $U_n$ contains all the columns that appear in $A$. More precisely, for any column $A^{(j)}$ there exists an index $1 \leq i \leq n$ such that $A^{(j)} = U_n^{(i)}$. We can therefore define an $n \times n$ matrix, $P$, such that $A = UP$. In particular the matrix $P$ contains one 1 entree per column such that $P(i,j) = 1$ if $A^{(j)} = U_m^{(i)}$.

## Applying $U$

Denote by $U_n$ the $\log_2(n) \times n$ matrix containing all possible strings over $\{0,1\}$ of length $\log(n)$. Notice immediately that $U_n$ can be constructed using $U_{n/2}$ as follows:

$$U_1 = \begin{pmatrix} 0 & 1 \end{pmatrix}, \quad U_n = \begin{pmatrix} 0,\ldots,0 & 1,\ldots,1 \\ U_{n/2} & U_{n/2} \end{pmatrix}.$$

Applying $U_n$ to any vector $z$ requires less then $3n$ operations, which can be shown by dividing $z$ into its first and second halves, $z_1$ and $z_2$, and computing the product $U_n z$ recursively.

$$U_n z = \begin{pmatrix} 0,\ldots,0 & 1,\ldots,1 \\ U_{n/2} & U_{n/2} \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 0 \cdot \sum z_1 + 1 \cdot \sum z_2 \\ U_{n/2}(z_1 + z_2) \end{pmatrix}$$

Computing the vector $z_1 + z_2$, and the sums $\sum z_1$ and $\sum z_2$ requires $3n/2$ operations. We get the recurrence relation

$$T(2) = 2, \quad T(n) = T(n/2) + 3n/2 \qquad \Rightarrow \qquad T(n) \leq 3n.$$

The sum over $z_1$ is of course redundant here. However, it would not have been if the entrees in $U$ were $\{-1,1\}$ for example.

### Constructing $P$

Since $U$ is applied implicitly (not stored) we are only concerned with constructing $P$. An entry $P(i,j)$ is set to 1 if $A^{(j)} = U_n^{(i)}$. Notice that by our construction of $U_n$ its $i$'th column encodes the binary value of the index $i$. Thus, if a column of $A$ contains the binary representation of the value $i$, it is equal to $U^{(i)}$. We therefore construct $P$ by reading $A$ and setting $P(i,j)$ to 1 if $A^{(j)}$ represents the value $i$. This clearly can be done in time $O(mn)$, the size of $A$. Since $P$ contains only $n$ nonzero entries (one for each column of $A$), it can be applied in $n$ steps.

Although $A$ is of size $\log(n) \times n$, it can be applied in linear time ($O(n)$) and a $\log(n)$ factor is gained. If the number of rows, $m$, in $A$ is more then $\log(n)$ we can section $A$ into at most $\lceil m/\log(n) \rceil$ matrices each of size at most $\log(n) \times n$. Since each of the sections can be applied in $O(n)$ operations, the entire matrix can be applied in $O(mn/\log(n))$ operations.

## Remarks

**Remark 1 ($n < m$)** *Notice that if $n < m$ we can section $A$ vertically to sections of size $m \times \lceil \log(m) \rceil$ and argue very similarly that $P^T U^T$ can be applied in linear time. Thus a $\log(m)$ saving factor in running time is achieved.*

**Remark 2 (Constant sized alphabet)** *The definition of $U_n$ can be changed so that it encodes all possible strings over a larger alphabet $\Sigma$, $|\Sigma| = S$ .*

$$U_m = \begin{pmatrix} \sigma_1, \ldots, \sigma_1 & \ldots & \sigma_\ell, \ldots, \sigma_\ell \\ U_{n/S} & \ldots & U_{n/S} \end{pmatrix}$$

*The matrix $U_n$ of size $\log_S(n) \times n$ encodes all possible strings of length $\log_S(n)$ over $\Sigma$ and can be applied in linear time $O(n)$. If the number of rows in $A$ is larger, we divide it into sections of size $\log_{|\Sigma|}(n) \times n$. The total running time therefore becomes $O(mn \log(S)/\log(\max\{m,n\})$.*

**Remark 3 (Matrix operations over semirings)** *Suppose we supply $\Sigma$ with an associative and commutative addition operation $(+)$, and a multiplication operation $(\cdot)$ that distributes over $(+)$. If both $A$ and $x$ are chosen from $\Sigma$ the matrix vector multiplication over the semiring $\{\Sigma, +, \cdot\}$ can be performed using the exact same algorithm. This is, of course, not a new result. However, it includes all other* log*-factor-saving results.*

## Dimensionality reduction

Assume we are given $p$ points $\{x_1, \ldots, x_p\}$ in $\mathbb{R}^n$ and are asked to embed them into $\mathbb{R}^m$ such that all distances between points are preserved up to distortion $\varepsilon$ and $m \ll n$. A classic result by Johnson and Lindenstrauss [9] shows that this is possible for $m = \Theta(\log(p)/\varepsilon^2)$. The algorithm first chooses a random $m \times n$ matrix, $A$, from a special distribution. Then, each of the vectors (points) $\{x_1, \ldots, x_p\}$ are multiplied by $A$. The embedding $x_i \rightarrow Ax_i$ can be shown to have the desired property with at least constant probability.

Clearly a naive application of $A$ to each vector requires $O(mn)$ operations. Recently Ailon and Liberty [10], modifying the work of Ailon and Chazelle [11], allows $A$ to be applied in $O(n\log(m))$ operations. We now claim that, in some situations, an older result by Achlioptas can be (trivially) modified to out preform this last result. In particular, Achlioptas [12] showed that the entries of $A$ can be chosen $i.i.d$ from $\{-1, 0, 1\}$ with some constant probabilities. Using our method and Achlioptas's matrix, one can apply $A$ to each vector in $O(n\log(p)/\log(n)\varepsilon^2)$ operations (remainder $m = O(\log(p)/\varepsilon^2)$). Therefore, if $p$ is polynomial in $n$ then $\log(n) = O(\log(p))$ and applying $A$ to $x_i$ requires only $O(n/\varepsilon^2)$ operations. For a constant $\varepsilon$ this outperforms the best known $O(n\log(m))$ and matches the lower bound. Notice that the dependance on $\varepsilon$ is $1/\varepsilon^2$ instead of $\log(1/\varepsilon)$ which makes this result actually much slower for most practical purposes.

## Experiments

Here we compare the running time of applying a $log(n) \times n$ , $\{0, 1\}$ matrix $A$ to a vector of floating point variables $x \in \mathbb{R}^n$ using three methods. The first is a *naive* implementation in C. This simply consists of two nested loops ordered with respect to memory allocation. The second is an *optimized* matrix vector code. We test ourselves against LAPACK which uses BLAS subroutines. The third is, of course, the *mailman* algorithm following the preprocessing stage.

Although the complexity of the first two methods is $O(n\log(n))$ and that of the mailman algorithm is $O(n)$ the improvement factor is well below $\log(n)$. This is because the memory access pattern in applying $P$ is problematic at best, and creates many cache faults. Bare in mind that these results might depend on memory speed and size vs. CPU speed of the specific machine. The experiment below denotes the time required for the actual multiplication not including memory allocation. (Experiments were conducted on an Intel Pentium M 1.4Ghz processor, 598Mhz Bus speed and 512Mb of RAM.)

As seen in figure 1 , the mailman algorithm operates faster then the naive algorithm even for $4 \times 16$

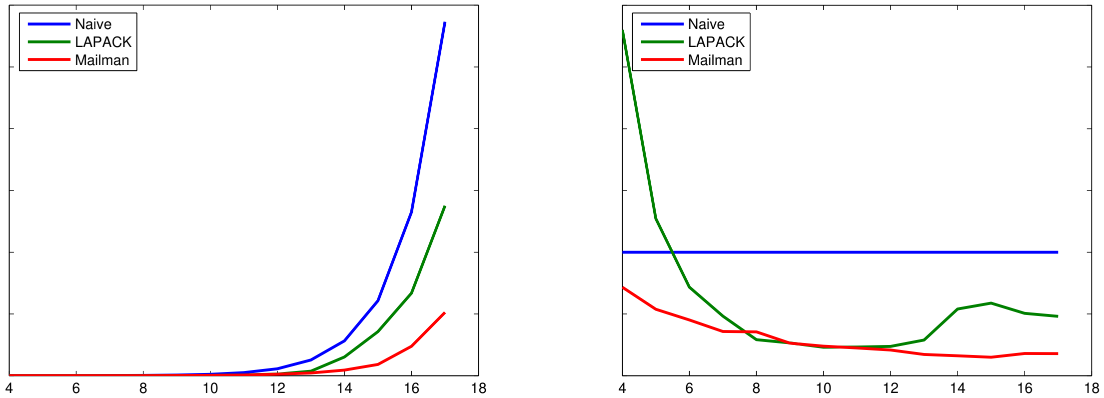


Figure 1: Running time of three different algorithms for matrix vector multiplication. Naive is a nested loop written in C. LAPACK is a machine optimized code. The mailman algorithm is described above. The matrices were of size $m$ (on the $x$ axis) by $2^m$ and they were multiplied by real (double precision) values vectors. The right figure plots the running times relative to those of the naive algorithm.

matrices. It also outperforms the LAPACK procedure for most matrix sizes. The machine specific optimized code (LAPACK) is superior when the matrix row allocation size approaches the memory page size. A machine specific optimized mailman algorithm might take advantage of the same phenomenon and outperform the LAPACK on those values as well.

## Concluding remark

It has been known for a long time that a log factor can be saved in matrix-vector multiplication when the matrix and the vector are over constant size alphabets. In this paper we described an algorithm that achieves this while also dealing with real-valued vectors. As such the idea by itself is neither revolutionary nor complicated, but it is useful in current contexts. We showed, as a simple application of it, that random projections can be achieved asymptotically faster then the best currently known algorithm, provided the number of projected points is polynomial in their original dimension. Moreover, we saw that our algorithm is practically advantageous even for small matrices.

## References


[1] Shmuel Winograd. On the number of multiplications necessary to compute certain functions. *Communications on Pure and Applied Mathematics*, (2):165–179, 1970.

[2] M.A. Kronrod V.L. Arlazarov, E.A. Dinic and I.A. Faradzev. On economic construction of the transitive closure of a direct graph. *Soviet Mathematics, Doklady*, (11):1209–1210, 1970.

[3] N Santoro and J Urrutia. An improved algorithm for boolean matrix multiplication. *Computing*, 36(4):375–382, 1986.

[4] Nicola Santoro. Extending the four russians' bound to general matrix multiplication. *Inf. Process. Lett.*, 10(2):87–88, 1980.

[5] John E. Savage. An algorithm for the computation of linear forms. *SIAM J. Comput.*, 3(2):150–158, 1974.

[6] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, (4):354–356, 08 1969.

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM.

[8] Ryan Williams. Matrix-vector multiplication in sub-quadratic time: (some preprocessing required). In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 995–1001, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[9] W.B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemp. Math.*, 26:189–206, 1984.

[10] Nir Ailon and Edo Liberty. Fast dimension reduction using rademacher series on dual bch codes. In *Symposium on Discrete Algorithms (SODA), accepted*, 2008.

[11] Nir Ailon and Bernard Chazelle. Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform. In *Proceedings of the 38st Annual Symposium on the Theory of Compututing (STOC)*, pages 557–563, Seattle, WA, 2006.

[12] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, 2003.