An exploration of matrix equilibration

Paul Liu

Abstract

We review three algorithms that scale the infinity-norm of each row and column in a matrix to 1. The first algorithm applies to unsymmetric matrices, and uses techniques from graph theory to scale the matrix. The second algorithm applies to symmetric matrices, and is notable for being asymptotically optimal in terms of operation counts. The third is an iterative algorithm that works for both symmetric and unsymmetric matrices. Experimental evidence of the effectiveness of all three algorithms in improving the conditioning of the matrix is demonstrated.

1 Introduction

In many cases of practical interest, it is important to know if a given linear system is ill-conditioned. If the condition number of the linear system is high, then a computed solution will often have little practical use, since the solution will be highly sensitive to a perturbation in the original system. To handle ill-conditioned systems, practitioners often preprocess the matrix of the linear system by means of scaling and permutation. This preprocessing stage often betters the conditioning of the matrix and is referred to as "equilibration".

In the case of direct solvers, equilibration schemes have been devised as to make the solves more stable. For example, the scheme used in [3] computes a reordering and scaling of the matrix to eliminate the need for pivoting during gaussian elimination. In the case of iterative methods, equilibration is used to minimize the given matrix's condition number under some norm, as to decrease round-off errors in matrix vector products. For this paper, we focus on equilibration algorithms whose purpose is to minimize the condition number of a given matrix.

One way to lower the condition number is to "equilibrate" the matrix, by scaling and permuting its rows and columns. For the condition number in the max-norm, optimal scalings have been identified for various classes of matrices [1]. However, there is currently no known algorithm to optimally equilibrate a general matrix and minimize its condition number, though several heuristics have been identified that seem to work well. One such heuristic is to scale all rows and columns of a given matrix to one under some given norm. This approach works well for most ill-conditioned systems, as shown by the numerical evidence provided in [2]. This paper will present three algorithms — each interesting in its own right — that scale the max-norm of each row and column in a given matrix to one.

2 Three algorithms

For convenience, we will assume that the given matrix A is a real square matrix that only has positive entries. Since we will only consider algorithms which scale and permute A, it is clear that we can enforce positivity on A without loss of generality. We also assume that it has a non-zero diagonal, though all of the algorithms described in this section works without that restriction with minor or no modifications.

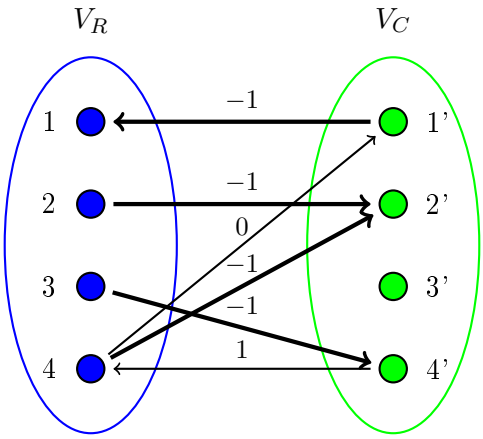


Figure 2.1: A bipartite graph on 8 nodes. Each edge (i, j') is weighted with a weight w_{ij}.

2.1 An algorithm for unsymmetric matrices

The first algorithm we present is due to Duff and Koster [5] and is implemented as algorithm MC64 in the widely used HSL Software Library. The algorithm is intended for unsymmetric matrices, and attempts to make A as diagonally dominant as possible. To be precise, the algorithm first finds a permutation matrix P such that the product of the diagonal entries of PA is maximized. Then scaling matrices R and C are found so that

$$\tilde{A} = PRAC$$

has 1's on the diagonal and all other entries less than or equal to 1.

This is achieved by modeling A as a bipartite graph, and then finding the minimum weighted matching in that graph. Before we proceed further into the details of this algorithm, we introduce the reader to the weighted bipartite matching problem.

2.1.1 Weighted bipartite matchings

Define a graph G to be bipartite if it can be partitioned into two node sets $V_R = \{1, 2, \ldots, n\}$ and $V_C = \{1', 2', \ldots, m'\}$, where every edge of the graph connects a vertex in V_R to a vertex in V_C (an example is shown in Figure 2.1). Each edge (i, j') of the graph is directed and weighted with a real number w_{ij}. In Figure 2.1, the directions are shown by arrows and the weights are shown by the edge labels.

The weighted bipartite matching problem then, is to choose a set M of edges in G, no two of which are incident to the same vertex, such that

$$\text{cost} = \sum_{(i,j')\in M} w_{ij}$$

is minimized. We refer to the optimal set M as a minimum matching.

If $|V_R| = |V_C|$, one can also impose the restriction that each vertex in $V_R \cup V_C$ is incident to an edge in M. This is called a minimum perfect matching. Note that M provides a 1-to-1 and onto map between elements of V_R and V_C. In Figure 2.1, the minimum perfect matching is denoted by the bolded edges.

An interesting fact about weighted bipartite matchings is that if there is a minimum perfect matching M, then there exists a set of *dual variables* $\{u_i\}$ and $\{v_j\}$ such that for all edges (i, j'),

$$\begin{cases} u_i + v_j = w_{ij}, & \text{if } (i, j') \in M \\ u_i + v_j \leq w_{ij}, & \text{if } (i, j') \notin M \end{cases} \tag{2.1}$$

This theorem, as well as a polynomial time algorithm of computing the minimum perfect matching along with the dual variables, is outlined in [6].

2.1.2 Application to matrix equilibration

Suppose A has dimensions $n \times n$, nnz non-zeros, and follows the restrictions we outlined at the start of Section 2.

Then MC64 solves the equilibration problem in the following manner:

1. A bipartite graph G is created from A, with vertex $i \in V_R$ representing row i of A and vertex $j' \in V_C$ representing column j of A. An edge (i, j') is then added to the graph if $A_{ij} \neq 0$. The weight of the edge is set to $-\log(A_{ij})$.
2. The minimum perfect matching M is computed. From the matching, we determine a permutation matrix P by setting $P_{ji} = 1$ for each edge (i, j') in the matching, and $P_{ij} = 0$ everywhere else. Note that since the matching M is 1-to-1 and onto, P will have the effect of permuting row i to row j. Hence the diagonal of the permuted matrix PA will be exactly the non-zeros of A that the edges in M correspond to.
3. Using the dual variables $\{u_i\}$ and $\{v_j\}$ from computing M, the scaling matrices R and C are computed by setting $R = \text{diag}(r_1, r_2, \ldots, r_n)$ and $C = \text{diag}(c_1, c_2, \ldots, c_n)$, where $r_i = \exp(u_i)$ and $c_i = \exp(v_i)$.

By weighing G in the manner described in step 1, we see that finding the minimum perfect matching will be equivalent to minimizing

$$\begin{aligned} \text{cost} &= \sum_{(i,j')\in M} w_{ij} \\ &= \sum_{(i,j')\in M} -\log(A_{ij}) \\ &= -\log\left(\prod_{(i,j')\in M} A_{ij}\right) \end{aligned}$$

which is the same as maximizing the product $\prod_{(i,j')\in M} A_{ij}$.

By the permutation computed in step 2, the product we maximized in step 1 will be permuted to the diagonal of PA, hence maximizing the product of the diagonal of PA over all possible permutations P.

By setting R and C as described in step 3, we see from Equation 2.1 that for each non-zero element α_{ij} in RAC, we'll have

$$\begin{aligned} \alpha_{ij} &= \exp(u_i) A_{ij} \exp(v_j) \\ &= \exp(u_i + v_j + \log(A_{ij})) \\ &\leq \exp(-\log(A_{ij}) + \log(A_{ij})) \\ &= 1 \end{aligned}$$

with equality when $(i, j') \in M$.

Hence $\tilde{A} = PRAC$ will have 1's on the diagonal and all other entries less than or equal to 1.

Since the most practical algorithm to compute a minimum perfect matching runs in $\mathcal{O}(VE)$ time on a graph with V vertices and E edges, this scaling algorithm runs in $\mathcal{O}(n \cdot nnz)$ time in the worst case. As noted in [5] however, the running time behaves more like $\mathcal{O}(n + nnz)$ for most matrices.

Note that the key insight to MC64 is the reduction of matrix equilibration to minimum bipartite matching. By choosing different weight functions for the edges of G, we open MC64 to a wide variety of modifications. There are variants of MC64 which maximizes different quantities,

such as the smallest element on the diagonal [5], or the ratio of the largest and smallest element in A [7]. Hence MC64 is interesting in that it provides a framework on which further algorithms can be built. Furthermore, it introduces numerical analysts to interesting algorithms from graph theory. As we'll see in section 3, it also works well in practice.

2.2 An algorithm for symmetric matrices

In this section we assume that A is symmetric.

The second algorithm we present is due to Bunch [2] and is implemented in the soon-to-be-released software package SYM-ILDL [8]. The algorithm is intended for symmetric matrices, and computes a diagonal matrix D such that

$$\tilde{A} = DAD$$

has max-norm 1 on every row and column.

Before we dive into the details of Bunch's algorithm, we first prove a helpful lemma.

Lemma 1. *Let L be the lower triangular part (not including the diagonal) of A and let $\triangle$ be the diagonal of A. If there exists a diagonal matrix D such that every row in $D(L+\triangle)D$ has max-norm 1, then every row and column in DAD will have max-norm 1.*

Proof. If $D(L+\triangle)D$ has max-norm 1 in every row, then $D(L^T+\triangle)D$ will have norm 1 in every column. Note that every entry in DL^TD and DLD will be less than or equal to 1. Hence

$$\begin{aligned} DAD &= D(L+\triangle+L^T)D \\ &= D(L+\triangle)D + DL^TD \\ &= DLD + D(L^T+\triangle)D. \end{aligned}$$

The second equality shows that DAD has norm 1 in every row, and the third equality shows that DAD has norm 1 in every column. □

Let $T = L + \triangle$. In Bunch's algorithm, we greedily scale one row and column of T in order, starting from the first row to the last row of A. Let $D = \mathrm{diag}(d_1, d_2, \ldots, d_n)$ be our scaling matrix. Bunch's algorithm is simply the following:

- For $1 \leq i \leq n$, set
$$d_i := \left(\max \left\{ \sqrt{T_{ii}}, \max_{1 \leq j \leq i-1} d_j T_{ij} \right\} \right)^{-1}.$$

The correctness of the algorithm is also easy to see: since $d_i \leq (\max_{1 \leq j \leq i-1} d_j T_{ij})^{-1}$, scaling row i by d_i ensures that

$$d_i T_{ij} d_j = (DTD)_{ij} \leq 1 \tag{2.2}$$

for every element on the i-th row of DTD except possibly $(DTD)_{ii}$. The diagonal element is taken care of by ensuring $d_i \leq \left(\sqrt{T_{ii}}\right)^{-1}$, where the square root comes from the constraint

$$(DTD)_{ii} = d_i^2 T_{ii} \leq 1. \tag{2.3}$$

As d_i is chosen to be the maximum possible value bounded by constaints 2.2 and 2.3, there will always be at least one entry of magnitude 1 on the i-th row after we are done. Since we scale all rows from 1 to n, DTD will have max-norm 1 in every row. Hence DAD will have max-norm 1 in every row and column by our lemma.

Though there doesn't seem to be any obvious extensions to Bunch's algorithm, it is interesting due to its simplicity and its asymptotically minimal runtime of $\mathcal{O}(n + nnz)$, where n is the dimension of A and nnz is the number of non-zeros in A. As far as the author can tell, it is the only equilibration algorithm for symmetric matrices that can be described by a single line of code.

2.3 An algorithm for general matrices

The final algorithm we present is the author's own creation, though it shares many similarities with the one presented in [9].[1] For convenience, we refer to the final algorithm as ALG3. As with Duff's algorithm, this algorithm attempts to find matrices R and C such that

$$\tilde{A} = RAC$$

has max-norm 1 in every row and column. When A is symmetric, the algorithm is designed to give $R = C$.

Let $r(A, i)$ and $c(A, i)$ denote the i -th row and column of A respectively, and let $D(i, \alpha)$ to be the diagonal matrix with $D_{jj} = 1$ for all $j \neq i$ and $D_{ii} = \alpha$. Using this notation, ALG3 is shown in 2.3.

Algorithm 1 ALG3 for equilibrating general matrices in the max-norm.

1: $R := I$
2: $C := I$
3: $\tilde{A} := A$
4: **while** R **and** C have not yet converged **do**:
5: **for** $i := 1$ to n
6: $\alpha_r := \frac{1}{\sqrt{||r(\tilde{A},i)||_\infty}}$
7: $\alpha_c := \frac{1}{\sqrt{||c(\tilde{A},i)||_\infty}}$
8: $R := R \cdot D(i, \alpha_r)$
9: $C := C \cdot D(i, \alpha_c)$
10: $\tilde{A} := D(i, \alpha_r)\tilde{A}D(i, \alpha_c)$
11: **end for**
12: **end while**

Given a tolerance ϵ, the convergence criterion for ALG3 is that $|1 - \alpha_r| < \epsilon$ and $|1 - \alpha_c| < \epsilon$ for every row and every column. When this criterion is reached, we terminate the algorithm.

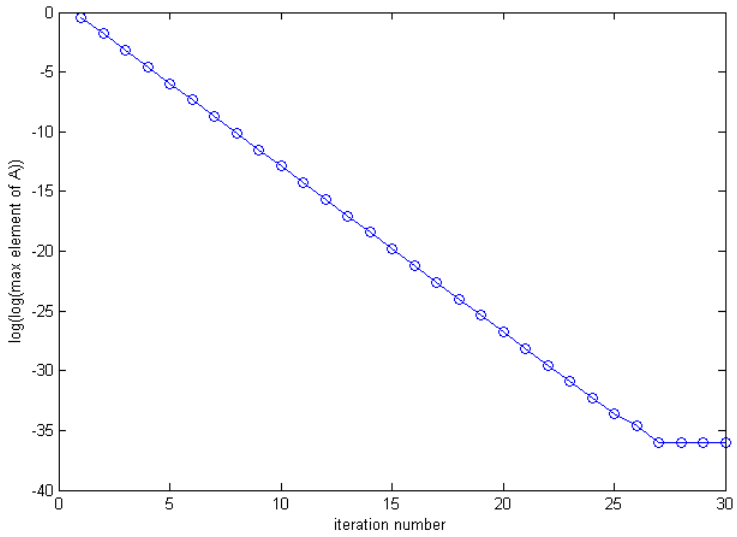
As we can see above, each iteration of the for loop (line 5) attempts to fix a single row and column of $\tilde{A}$. On iteration i, we take the maximum element α_i in row i, scale it to $\sqrt{\alpha_i}$, and then do the same for column i. If the maximum of both row and column i is on the diagonal, it is scaled to 1. Though we might undo some of the scaling we did for row and column i when we are scaling some other row and column $j > i$, it seems from experiments that this does not stop the algorithm from converging.

Heuristically, the norm of each row and column converges to 1 since on each iteration of the while loop, we take the maximum element in each row and column to approximately square root of their original value. Since the iteration $T_{n+1} = \sqrt{T_n}$ converges to 1 linearly for any initial T_0, we expect to see this in the algorithm. In fact, as we can see in Figure 2.2, the maximum element in $\tilde{A}$ seems to converge linearly as expected. On a test of 1000 random 100×100 matrices, ALG3 successfully returned R and C for which RAC had max-norm 1 on every row and column. Since each iteration of the while loop costs $\mathcal{O}(nnz)$ operations, this algorithm seems to take $\mathcal{O}(nnz \cdot \log \epsilon)$ time if we assume linear convergence.

Furthermore, the algorithm seems to work for any norm, not just the infinity norm. That is, if we switched the norm on lines 6 and 7 to the p-norm, then ALG3 returns matrices R and C such that RAC has p-norm 1 in every row and column. This is also the behaviour of the algorithm given in [9], for which they prove convergence results for the infinity norm and p-norm for any finite $p \geq 1$.

[1] This algorithm can be seen as a variant of Ruiz's algorithm, as the same proof of correctness will apply with the same convergence rate.

Figure 2.2: Convergence of ALG3, with iteration plotted against log log of the maximum of A. Each 'iteration' in the graph above is one iteration of the while loop. The flat part of the plot past iteration 26 is the algorithm reaching the limits of machine precision.



Another note of interest is that we did not need to use the square root function for this algorithm. In fact, any function f with the property that the two sequences

$$y_{n+1} = y_n/f(y_n) \tag{2.4}$$
$$x_{n+1} = f(x_n) \tag{2.5}$$

both converge to 1 for any initial x_0 and y_0 seems to work in place of the square root. We do not even have to use the same f for different iterations of the for loop, though this seems a bit extreme. Equation 2.4 comes from the constraint that we when scale a row or column of $\tilde{A}$ during the course of ALG3, we want the elements $\tilde{A}_{ij}$ of that row or column to converge to 1. For the max-norm, the largest element α_i of row i will get scaled to $\alpha_i/f(\alpha_i)$. So for the algorithm to work in the max-norm, $\alpha_i/f(\alpha_i)$ must approach 1. Equation 2.5 comes from the constraint that α_r and α_c must converge to 1 for R and C to converge. Hence $1/f(x_n)$ must approach 1, which implies that $f(x_n)$ approaches 1. Though the convergence of 2.4 and 2.5 to 1 seem to be necessary conditions on f, they are actually sufficient. The proof of this follows straightforwardly from the proof of the algorithm given in given in [9].

3 Results

3.1 For unsymmetric matrices

For the unsymmetric case, we took a set of real, ill-conditioned, non-singular matrices from University of Florida's sparse matrix collection [4], ranging from sizes 1157 to 8192 with a maximum of 83842 non-zeros. In addition, we generated some smaller dense matrices in MATLAB (**unsym-rand01** to **unsym-rand5**) using the built-in **rand** command (**rand** returns matrices with entries picked uniformly from the open interval (0,1)). To make these random matrices ill conditioned, half of its rows (randomly selected) were scaled by 10. The same was done for the columns. The condition number in the 1-norm before and after equilibration with MC64 and ALG3 are shown in In Figure 1. These are listed as $\text{cond}_{orig}(A)$, $\text{cond}_{MC64}(A)$, and $\text{cond}_{ALG3}(A)$) respectively. The epsilon tolerance for ALG3 was set to 10^{-8}.

Table 1: Condition numbers before and after equilibration for ALG3 and MC64

matrix	n	nnz of A	$\text{cond}_{orig}(A)$	$\text{cond}_{MC64}(A)$	$\text{cond}_{ALG3}(A)$
dw4096	8192	41746	$1.50 \cdot 10^7$	$9.63 \cdot 10^5$	$5.90 \cdot 10^5$
rajat13	7598	48762	$1.46 \cdot 10^{11}$	$1.62 \cdot 10^1$	$6.07 \cdot 10^2$
utm5940	5940	83842	$1.91 \cdot 10^9$	$2.75 \cdot 10^9$	$3.90 \cdot 10^9$
tols2000	2000	5184	$6.92 \cdot 10^6$	$1.08 \cdot 10^2$	$1.11 \cdot 10^2$
rajat19	1157	3699	$9.17 \cdot 10^{10}$	$5.87 \cdot 10^{11}$	$7.33 \cdot 10^8$
unsym-rand05	1024	1048576	$2.73 \cdot 10^{13}$	$1.77 \cdot 10^5$	$2.27 \cdot 10^5$
unsym-rand04	512	262144	$2.07 \cdot 10^{11}$	$1.85 \cdot 10^5$	$5.66 \cdot 10^5$
unsym-rand03	256	65536	$1.26 \cdot 10^{11}$	$1.80 \cdot 10^4$	$2.78 \cdot 10^4$
unsym-rand02	128	16384	$1.03 \cdot 10^9$	$1.18 \cdot 10^4$	$1.26 \cdot 10^4$
unsym-rand01	64	4096	$1.59 \cdot 10^7$	$1.40 \cdot 10^3$	$2.10 \cdot 10^3$

As we can see in Table 1, ALG3 and MC64 are comparable, though ALG3 is asymptotically faster. For most matrices however, MC64 manages to obtain a lower condition number than ALG3.

For the **utm5940** and **rajat19** matrix, both ALG3 and MC64 were unable to decrease the condition number by much. These cases are to be expected, as MC64 and ALG3 are algorithms based on heuristics. An interesting extension of this project would be to work out specific worst cases for both algorithms, and see how to alter them as to make these worst cases less damaging.

3.2 For symmetric matrices

For the symmetric case, we again took a set of real, symmetric, non-singular matrices from University of Florida's sparse matrix collection [4], ranging from sizes 1806 to 11948 with a maximum of 149090 non-zeros. In addition, we generated some smaller dense symmetric matrices in MATLAB. As in the unsymmetric case, the matrices were generated with **rand** and then scaled to be ill-conditioned. They were then made symmetric by adding each matrix to their transpose. For each matrix, the condition numbers in the 1-norm before equilibration, after equilibration with Bunch's Algorithm, and after equilibration with ALG3 were computed. These are listed as $\text{cond}_{orig}(A)$, $\text{cond}_{Bunch}(A)$, and $\text{cond}_{ALG3}(A)$) respectively. The results are shown in in Figure 2. As in the unsymmetric case, the epsilon tolerance for ALG3 was set to 10^{-8}.

Table 2: Condition numbers before and after equilibration for ALG3 and Bunch's algorithm

matrix	n	nnz of A	$\text{cond}_{orig}(A)$	$\text{cond}_{Bunch}(A)$	$\text{cond}_{ALG3}(A)$
bcsstk18	11948	149090	$6.48 \cdot 10^{11}$	$1.21 \cdot 10^5$	$1.21 \cdot 10^5$
c-44	10728	85000	$1.14 \cdot 10^8$	$1.47 \cdot 10^5$	$2.15 \cdot 10^5$
meg4	5860	25258	$4.19 \cdot 10^{10}$	$1.73 \cdot 10^1$	$1.73 \cdot 10^1$
sts4098	4098	72356	$4.44 \cdot 10^8$	$3.18 \cdot 10^4$	$3.18 \cdot 10^4$
bcsstk14	1806	63454	$1.31 \cdot 10^{10}$	$9.91 \cdot 10^3$	$9.91 \cdot 10^3$
sym-rand05	1024	1048576	$1.79 \cdot 10^{11}$	$2.81 \cdot 10^5$	$2.24 \cdot 10^5$
sym-rand04	512	262144	$3.07 \cdot 10^9$	$9.63 \cdot 10^4$	$7.42 \cdot 10^4$
sym-rand03	256	65536	$1.93 \cdot 10^9$	$3.32 \cdot 10^4$	$4.20 \cdot 10^4$
sym-rand02	128	16384	$9.12 \cdot 10^{11}$	$2.19 \cdot 10^4$	$2.12 \cdot 10^4$
sym-rand01	64	4096	$2.04 \cdot 10^8$	$1.90 \cdot 10^3$	$9.36 \cdot 10^3$

As we can see in Table 2, the resulting condition numbers between ALG3 and Bunch's algorithm are quite comparable, with ALG3 slightly better in most test cases. In all tests cases, the condition number was successfully decreased.

4 Conclusion

We presented three algorithms that scale the max-norm of every row and column in a matrix to 1. In the unsymmetric case, MC64 seemed to be slightly better, though both ALG3 and MC64 were unable to decrease the condition number for some matrices. In the symmetric case, ALG3 seemed to be slightly better, though Bunch's algorithm was much simpler and faster.

All three algorithms have unique strengths:

- MC64 handles unsymmetric matrices well, and is open to extension by reweighing the edges of the bipartite graph described in Section 2.1.
- Bunch's algorithm is simple to code, and achieves an asymptotically optimal runtime of $\mathcal{O}(n + nnz)$.
- ALG3 is general purpose, and can be used for both symmetric and unsymmetric matrices. Furthermore, experimental evidence suggests that it can be extended to equilibrate the p-norm, not just the max-norm.

Ultimately, the choice of an equilibration method will depend on the specific problem being solved. What we demonstrated here however, is that there are many ways to equilibrate a matrix: using techniques from graph theory (MC64), iterative methods (ALG3), and greedy algorithm design (Bunch). The author hopes that the reader too will be inspired to create their own equilibration technique, and looks forward to future techniques to come.

5 Acknowledgment

We thank Prof. Chen Greif, for his support and generous extension on the due date of this project.

References

[1] F. L. Bauer. Optimally scaled matrices. *Numerische Mathematik*, 5:73–87, 1963.

[2] James Bunch. Equilibration of symmetric matrices in the max-norm. *Journal of the ACM*, 18:566–572, 1971.

[3] A.R. Curtis and J.K. Reid. On the automatic scaling of matrices for gaussian elimination. *J. Inst. Maths. Applics.*, 10:118–124, 1972.

[4] T. A. Davis and Y. Hu. The university of florida sparse matrix collection.

[5] Iain Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22:973–996, 2001.

[6] L. R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[7] D. R. Fulkerson and P. Wolfe. An algorithm for scaling matrices. *Siam Review*, 4:142–146, 1962.

[8] Chen Greif, Shiwen He, and Paul Liu. SYM-ILDL, a package for the incomplete LDL^T factorization of indefinite symmetric matrices.

[9] Daniel Ruiz. A symmetry preserving algorithm for matrix scaling. Technical Report 7552, INRIA, 2001.