Internet Mathematics, 11:352–384, 2015

ISSN: 1542-7951 print/1944-9488 online
DOI: 10.1080/15427951.2014.971203

Taylor & Francis
Taylor & Francis Group

SUBLINEAR COLUMN-WISE ACTIONS OF THE MATRIX EXPONENTIAL ON SOCIAL NETWORKS

David F. Gleich[1] and Kyle Kloster[2]
[1]*Computer Science Department, Purdue University, West Lafayette, Indiana, USA*
[2]*Mathematics Department, Purdue University, West Lafayette, Indiana, USA*

Abstract *We consider stochastic transition matrices from large social and information networks. For these matrices, we describe and evaluate three fast methods to estimate one column of the matrix exponential. The methods are designed to exploit the properties inherent in social networks, such as a power-law degree distribution. Using only this property, we prove that one of our three algorithms has a sublinear runtime. We present further experimental evidence showing that all three of them run quickly on social networks with billions of edges, and they accurately identify the largest elements of the column.*

1. INTRODUCTION

Matrix exponentials are used for node centrality [17, 18, 20], link prediction [28], graph kernels [27], and clustering [14]. In the majority of these problems, only a rough approximation of a column of the matrix exponential is needed. Here, we present methods for fast approximations of

$$\exp\{\boldsymbol{P}\}\,\mathbf{e}_c,$$

where $\boldsymbol{P}$ is a column-stochastic matrix and $\mathbf{e}_c$ is the cth column of the identity matrix. This suffices for many applications and also allows us to compute $\exp\left\{-\hat{\boldsymbol{L}}\right\}\mathbf{e}_c$, where $\hat{\boldsymbol{L}}$ is the normalized Laplacian.

To state the problem precisely and fix notation, let $\boldsymbol{G}$ be a graph adjacency matrix of a directed graph and let $\boldsymbol{D}$ be the diagonal matrix of out-degrees, where $\boldsymbol{D}_{ii} = d_i$, the degree of node i. For simplicity, we assume that all nodes have positive out-degrees, thus, $\boldsymbol{D}$ is invertible. The methods we present are designed to work for $\boldsymbol{P} = \boldsymbol{G}\boldsymbol{D}^{-1}$ and, by extension, the negative normalized Laplacian $-\hat{\boldsymbol{L}} = \boldsymbol{D}^{-1/2}\boldsymbol{G}\boldsymbol{D}^{-1/2} - \boldsymbol{I}$. This is because the relationship

$$\exp\left\{\boldsymbol{D}^{-1/2}\boldsymbol{G}\boldsymbol{D}^{-1/2} - \boldsymbol{I}\right\} = e^{-1}\boldsymbol{D}^{-1/2}\exp\left\{\boldsymbol{G}\boldsymbol{D}^{-1}\right\}\boldsymbol{D}^{1/2}$$

implies $\exp\{-\hat{\boldsymbol{L}}\}\,\mathbf{e}_c = \sqrt{d_c}e^{-1}\boldsymbol{D}^{-1/2}\exp\{\boldsymbol{P}\}\,\mathbf{e}_c$, which allows computation of either column given the other, at the cost of scaling the vector.

Address correspondence to Kyle Kloster, Mathematics Department, Purdue University, 150 N. University Street, West Lafayette, IN 47906, USA. E-mail: kkloste@purdue.edu

Color versions of one or more of the figures in the article can be found online at www.tandfonline.com/uinm.

1.1. Previous Work

Computing the matrix exponential for a general matrix $\boldsymbol{A}$ has a rich and "dubious" history [32]. For any matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and vector $\mathbf{b} \in \mathbb{R}^n$, one approach is to use a Taylor polynomial approximation:

$$\exp\{\boldsymbol{A}\}\,\mathbf{b} \approx \sum_{j=0}^{N} \frac{1}{j!}\boldsymbol{A}^j\mathbf{b}.$$

This sequence converges to the correct vector as $N \to \infty$ for any square matrix, however, it can be problematic numerically. A second approach is to first compute an $m \times m$ upper-Hessenberg form of $\boldsymbol{A}$, $\boldsymbol{H}_m$, via an m-step Krylov method, $\boldsymbol{A} \approx \boldsymbol{V}_m\boldsymbol{H}_m\boldsymbol{V}_m^T$. Using this form, we can approximate $\exp\{\boldsymbol{A}\}\,\mathbf{b} \approx \boldsymbol{V}_m \exp\{\boldsymbol{H}_m\}\,\mathbf{e}_1$ by performing $\exp\{\boldsymbol{H}_m\}\,\mathbf{e}_1$ on the much smaller, and better controlled, upper-Hessenberg matrix $\boldsymbol{H}_m$. These concepts underlie many standard methods for obtaining $\exp\{\boldsymbol{A}\}\,\mathbf{b}$.

Although the Taylor and Krylov approaches are fast and accurate — see references [23, 21, 3] for the numerical analysis — existing implementations depend on repeated matrix-vector products with the matrix $\boldsymbol{A}$. The Krylov-based algorithms also require orthogonalization steps between successive vectors. When these algorithms are used to compute exponentials of graphs with small diameter, such as the social networks we consider here, the repeated matrix-vector products cause the vectors involved to become dense after only a few steps. The subsequent matrix-vector products between the sparse matrix and dense vector require $O(|E|)$ work, where $|E|$ is the number of edges in the graph (and there are $O(|E|)$ nonzeros in the sparse matrix). This leads to a runtime bound of $O(T|E|)$ if there are T matrix vector products after the vectors become dense.

There are a few recent improvements to the Krylov methods that reduce the number of terms T that must be used [34, 33, 2, 3] or present additional special cases [7]. Both [33] and [3] present a careful bound on the maximum number of terms T. A new polynomial approximation for $\exp x$ improves on the Taylor polynomial approach and gives a tight bound on the necessary number of matrix-vector products in the case of a general symmetric positive semidefinite matrix $\boldsymbol{A}$ [33]. A bound on the number of Taylor terms for a matrix with bounded norm has also been presented [3].

Thus, the best runtimes provided by existing methods are $O(|E|)$ for the stochastic matrix of a graph. It should be noted, however, that the algorithms we present in this article operate in the specific context of matrices with 1-norm bounded by 1, and where the vector $\mathbf{b}$ is sparse with only 1 nonzero. In contrast, the existing methods we mention here apply more broadly.

In the case of exponentials of sparse graphs, a Monte Carlo procedure to estimate columns similar to our method was developed by [15]. They show that only a small number of random walks are needed to compute reasonably accurate solutions; however, the number of walks grows quickly with the desired accuracy. They also prove their algorithm [15] runs in time polylogarithmic in $1/\varepsilon$, although the ε accuracy is achieved in a degree-weighted infinity norm, making the computational goal distinct from our own. Our accuracy result is in the 1-norm, which provides uniform control over the error.

We note that, for a general sparse graph, it is impossible to get a work bound that is better than $O(n)$ for computing $\exp\{\boldsymbol{A}\}\,\mathbf{e}_c$ with accuracy ε in the 1-norm, even if $\boldsymbol{A}$ has only $O(n)$ nonzeros. For example, the star graph on n nodes requires $\Omega(n)$ work to compute certain columns of its exponential, as they have $O(n)$ nonzero entries of equal magnitude and, hence, cannot be approximated with less than $O(n)$ work. This shows there cannot

be a sublinear upper bound on work for accurately approximating general columns of the exponential of arbitrary sparse graphs.

We are able to obtain our sublinear work bound by assuming structure in the degree distribution of the underlying graph. Another case for which it is possible to show that sublinear algorithms are possible is when the matrices are banded, as considered by [8]. Banded matrices correspond to graphs that look like the line-graph with up to d connections among neighbors. If d is sufficiently small, or constant, then the exponential localizes and sublinear algorithms are possible. However, this case is unrealistic for social networks with highly skewed degree distributions.

1.2. Our Contributions

For networks with billions of edges, we want a procedure that avoids the dense vector operations involved in Taylor- and Krylov-based methods. In particular, we would like an algorithm to estimate a column of the matrix exponential that runs in time proportional to the number of *large* entries. Put another way, we want an algorithm that is *local* in the graph and produces a *local solution*.

Roughly speaking, a *local solution* is a vector that both accurately approximates the true solution, which can be dense, *and* has only a few nonzeros. A local algorithm is, then, a method for computing such a solution that requires work proportional to the size of the solution, rather than the size of the input. In the case of computing a column of the matrix exponential for a network with $|E|$ edges, the input size is $|E|$, but the desired solution $\exp\{\boldsymbol{P}\}\mathbf{e}_c$ has only a few significant entries. An illustration of this is given in Figure 1. From that figure, we see that the column of the matrix exponential has about 5 million nonzero entries. However, if we look at the approximation formed by the largest $3,000$ entries, it has a 1-norm error of roughly 10^{-4}. A local algorithm should be able to find these $3,000$ nonzeros without doing work proportional to $|E|$. For this reason, local methods

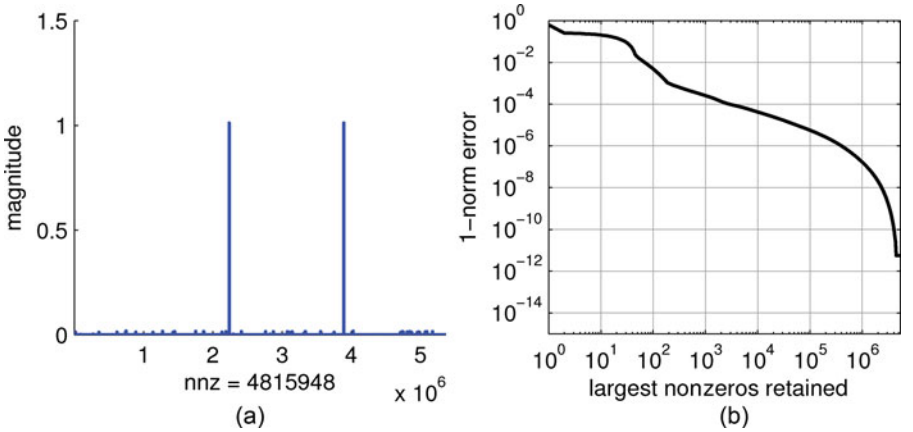

Figure 1 (a) A column of the matrix exponential from the LiveJournal graph with 5M vertices and 78M directed edges shows only two large entries and a total of 4.8M numerically nonzero entries. (b) The second figure shows the solution error in the 1-norm as only the largest entries are retained. This shows that there is a solution with 1-norm error of 10^{-4} with around 3,000 nonzero entries. These plots illustrate that the matrix exponential can be localized in a large network, and we seek local algorithms that will find these solutions without exploring the entire graph.

are a recognized and practical alternative to Krylov methods for solving massive linear systems from network problems; see, for instance [4, 12]. The essence of these methods is that they replace whole-graph matrix-vector products with targeted column-accesses; these correspond to accessing the out-links from a vertex in a graph structure.

In this article, we present three algorithms that approximate a specified column of $\exp\{\boldsymbol{P}\}$ where $\boldsymbol{P}$ is a sparse matrix satisfying $\|\boldsymbol{P}\|_1 \leq 1$ (Section 3). The main algorithm we discuss and analyze uses coordinate relaxation (Section 2.3) on a linear system to approximate a degree N Taylor polynomial (Section 2.1). This coordinate relaxation method yields approximations guaranteed to satisfy a prescribed error ε. For arbitrary graphs with maximum degree d, the error after l iterations of the algorithm we call `gexpm` is bounded by $O(l^{-1/(2d)})$ as shown in Theorem 4.2. Given an input error ε, the runtime to produce a solution vector with 1-norm error less than ε is, thus, sublinear in n for graphs with $d \leq O(\log\log n)$, as shown in our prior work [25].

This doubly logarithmic scaling of the maximum degree is unrealistic for social and information networks, where highly skewed degree distributions are typical. Therefore, in Section 5 we consider graphs with a power-law degree distribution, a property ubiquitous in social networks [19, 6]. By using this added assumption, we can show that for a graph with a particular power-law distribution, maximum degree d, and minimum degree δ, the `gexpm` algorithm produces a 1-norm error of ε in work that scales roughly as $d^2\log(d)^2$, and with total work bounded by $O(\log(1/\varepsilon)\,(1/\varepsilon)^{3\delta/2}\,d^2\log(d)\max\{\log(d), \log(1/\varepsilon)\})$ (Theorem 5.1). As a corollary, this theorem *proves* that columns of $\exp\{\boldsymbol{P}\}\,\mathbf{e}_c$ are localized.

Our second algorithm, `gexpmq`, is a faster heuristic approximation of the first algorithm. It retains the use of coordinate descent, but changes the choice of coordinate to relax to something that is less expensive to compute. It retains the rigorous convergence guarantee but loses the runtime guarantee.

The final method, `expmimv`, differs from the first two and does not use coordinate relaxation. Instead, it uses *sparse matrix-vector products* with only the z largest entries of the previous vector to avoid fill-in. This leads to a guaranteed runtime bound of $O(dz\log z)$, discussed in Theorem 3.3, but with no accuracy guarantee. Our experiments in Section 6.2 show that this method is orders of magnitude faster than the others.

Figure 2 compares the results of these algorithms on the same graph and vector from Figure 1 as we vary the desired solution tolerance ε for each algorithm. These results

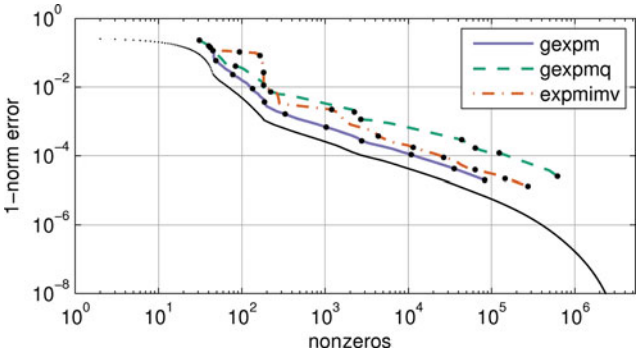

Figure 2 The result of running our three algorithms to approximate the vector studied in Figure 1. The small black dots show the optimal set of nonzeros chosen by *sorting* the true vector. The three curves show the results of running our algorithms as we vary the desired solution tolerance. Ideally, they would follow the tiny black dots exactly. Instead, they closely approximate this optimal curve with `gexpm` showing the best performance.

show that the algorithms all track the optimal curve, and sometimes closely! In the best case, they compute solutions with roughly three times the number of nonzeros as in the optimal solution; in the worst case, they need about 50 times the number of nonzeros. In the interest of full disclosure, we note that we altered the algorithms slightly for this figure. We removed a final step that significantly increases the number of nonzeros by making many tiny updates to the solution vector; these updates are so small that they do not alter the accuracy by more than a factor of 2. We also fixed an approximation parameter based on the Taylor degree to aid comparisons as we varied ε.

As we finish our introduction, let us note that the source code for all of our experiments and methods is available online.[1] The remainder of this article proceeds in a standard fashion by establishing the formal setting (Section 2), then introducing our algorithms (Section 3), analyzing them (Section 4, Section 5), and then showing our experimental evaluation (Section 6). This article extends our conference version [25] by adding the theoretical analysis with the power-law, presenting the `expmimv` method, and tightening the convergence criteria for `gexpmq`. Furthermore, we conduct an entirely new set of experiments on graphs with billions of edges.

2. BACKGROUND

The algorithm that we employ utilizes a Taylor polynomial approximation of $\exp\{\boldsymbol{P}\}\mathbf{e}_c$. Here, we provide the details of the Taylor approximation for the exponential of a general matrix. We also review the coordinate relaxation method we use in two of our algorithms.

Although the algorithms presented in subsequent sections are designed to work for $\boldsymbol{P}$, much of the theory in this section applies to any matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$. Thus, we present it in its full generality. For sections in which the theory is restricted to $\boldsymbol{P}$, we explicitly state so. Our rule of thumb is that we will use $\boldsymbol{A}$ as the matrix when the result is general and $\boldsymbol{P}$ when the result requires properties specific to our setting.

2.1. Approximating with Taylor Polynomials

The Taylor series for the exponential of a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is given by

$$\exp\{\boldsymbol{A}\} = \boldsymbol{I} + \frac{1}{1!}\boldsymbol{A}^1 + \frac{1}{2!}\boldsymbol{A}^2 + \cdots + \frac{1}{k!}\boldsymbol{A}^k + \cdots$$

and it converges for any square matrix $\boldsymbol{A}$. By truncating this infinite series to N terms, we may define

$$T_N(\boldsymbol{A}) := \sum_{j=0}^{N} \frac{1}{j!}\boldsymbol{A}^j,$$

and then approximate $\exp\{\boldsymbol{A}\}\mathbf{b} \approx T_N(\boldsymbol{A})\mathbf{b}$. For general $\boldsymbol{A}$, this polynomial approximation can lead to inaccurate computations if $\|\boldsymbol{A}\|$ is large and $\boldsymbol{A}$ has oppositely signed entries, as the terms $\boldsymbol{A}^j$ can then contain large, oppositely signed entries that cancel only in exact arithmetic. However, our aim is to compute $\exp\{\boldsymbol{P}\}\mathbf{e}_c$ specifically for a matrix of bounded norm, that is, $\|\boldsymbol{P}\|_1 \leq 1$. In this setting, the Taylor polynomial approximation is a reliable

[1] `https://www.cs.purdue.edu/homes/dgleich/codes/nexpokit/`

ε Desired	N Predicted by Lemma 2.1	N Required
10^{-5}	24	8
10^{-10}	46	13
10^{-15}	70	17

Table I Choosing the degree, N, of a Taylor polynomial to ensure an accuracy of ε shows that the bound from Lemma 2.1 is not tight, and that both methods are slowly growing.

and accurate tool. What remains is to choose the degree N to ensure that the accuracy of the Taylor approximation makes $\| \exp\{\boldsymbol{P}\}\,\mathbf{e}_c - T_N(\boldsymbol{P})\,\mathbf{e}_c\|$ as small as desired.

Choosing the Taylor polynomial degree. Accuracy of the Taylor polynomial approximation requires a sufficiently large Taylor degree, N. On the other hand, using a large N requires the algorithms to perform more work. A sufficient value of N can be obtained algorithmically by exactly computing the number of terms of the Taylor polynomial required to compute exp(1) with accuracy ε. Formally:

$$N = \arg\min_k \left\{ k \text{ where } \left(e - \sum_{\ell=0}^{k} \frac{1}{\ell!} \right) \leq \varepsilon \right\}.$$

We provide the following simple upper bound on N:

Lemma 2.1. *Let $\boldsymbol{P}$ and $\mathbf{b}$ satisfy $\|\boldsymbol{P}\|_1, \|\mathbf{b}\|_1 \leq 1$. Then choosing the degree, N, of the Taylor approximation, $T_N(\boldsymbol{P})$, such that $N \geq 2\log(1/\varepsilon)$ and $N \geq 3$ will guarantee*

$$\| \exp\{\boldsymbol{P}\}\,\mathbf{b} - T_N(\boldsymbol{P})\mathbf{b}\|_1 \leq \varepsilon$$

We present the proof, which does not inform our current exposition, in the appendix. Because Lemma 2.1 provides only a loose bound, we display in Table I values of N determined via explicit computation of exp(1), which are tight in the case that $\|\boldsymbol{P}\|_1 = 1$.

2.2. Error from Approximating the Taylor Approximation

The methods we present in Section 3 produce an approximation of the Taylor polynomial expression $T_N(\boldsymbol{P})\mathbf{e}_c$, which itself approximates $\exp\{\boldsymbol{P}\}\,\mathbf{e}_c$. Thus, a secondary error is introduced. Let $\mathbf{x}$ be our approximation of $T_N(\boldsymbol{P})\mathbf{e}_c$. We find

$$\| \exp\{\boldsymbol{P}\}\,\mathbf{e}_c - \mathbf{x} \| \leq \| \exp\{\boldsymbol{P}\}\,\mathbf{e}_c - T_N(\boldsymbol{P})\,\mathbf{e}_c \| + \| T_N(\boldsymbol{P})\,\mathbf{e}_c - \mathbf{x} \|,$$

by the triangle inequality. Lemma 2.1 guarantees the accuracy of only the first term; so if the total error of our final approximation $\mathbf{x}$ is to satisfy $\| \exp\{\boldsymbol{P}\}\,\mathbf{e}_c - \mathbf{x} \|_1 \leq \varepsilon$, then we must guarantee that the right-hand summand is less than ε. More precisely, we want to ensure for some $\theta \in (0, 1)$ that the Taylor polynomial satisfies $\| \exp\{\boldsymbol{P}\}\,\mathbf{e}_c - T_N(\boldsymbol{P})\,\mathbf{e}_c \|_1 \leq \theta\varepsilon$ and, additionally, our computed approximation $\mathbf{x}$ satisfies $\| T_N(\boldsymbol{P})\,\mathbf{e}_c - \mathbf{x} \|_1 \leq (1-\theta)\varepsilon$. We pick $\theta = 1/2$, although we suspect there is an opportunity to optimize this term.

2.3. The Gauss–Southwell Coordinate Relaxation Method

One of the algorithmic procedures we employ is to solve a linear system via Gauss–Southwell. The Gauss–Southwell (GS) method is an iterative method related to the Gauss–Seidel and coordinate descent methods [31]. In solving a linear system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ with current solution $\boldsymbol{x}^{(k)}$ and residual $\boldsymbol{r}^{(k)} = \mathbf{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}$, the GS iteration acts via coordinate relaxation on the largest magnitude entry of the residual at each step, whereas the Gauss–Seidel method repeatedly cycles through all elements of the residual. Similarly to Gauss–Seidel, the GS method converges on diagonally dominant matrices, symmetric positive definite matrices, and M-matrices. It is strikingly effective when the underlying system is sparse and the solution vector can be approximated locally. Because of this, the algorithm has been reinvented in the context of local PageRank computations [4, 9, 24]. Next, we present the basic iteration of GS.

Given a linear system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ with initial solution $\boldsymbol{x}^{(0)} = 0$ and residual $\boldsymbol{r}^{(0)} = \mathbf{b}$, GS proceeds as follows. To update from step k to step $k+1$, set $m^{(k)}$ to be the maximum magnitude entry of $\boldsymbol{r}^{(k)}$, i.e., $m^{(k)} := (\boldsymbol{r}^{(k)})_{i_k}$; then, update the solution and residual:

$$\begin{aligned} \boldsymbol{x}^{(k+1)} &= \boldsymbol{x}^{(k)} + m^{(k)} \quad \cdot \quad \mathbf{e}_{i_k} \qquad \text{update the } i_k\text{th coordinate only} \\ \boldsymbol{r}^{(k+1)} &= \boldsymbol{r}^{(k)} - m^{(k)} \quad \cdot \quad \boldsymbol{A}\mathbf{e}_{i_k} \qquad \text{update the residual.} \end{aligned} \tag{2.1}$$

Observe that updating the residual $\boldsymbol{r}^{(k)}$ in (2.1) involves adding only a scalar multiple of a column of $\boldsymbol{A}$ to $\boldsymbol{r}^{(k)}$. If $\boldsymbol{A}$ is sparse, then the whole step involves updating a single entry of the solution $\boldsymbol{x}^{(k)}$, and only a small number of entries of $\boldsymbol{r}^{(k)}$. When $\boldsymbol{A} = \boldsymbol{P}$, the column-stochastic transition matrix, then updating the residual involves accessing the out-links of a single node.

The reason that GS is called a "coordinate relaxation" method is that it can be derived by *relaxing* or *freeing* the i_kth coordinate to satisfy the linear equations in that coordinate only. For instance, suppose, for the sake of simplicity, that $\boldsymbol{A}$ has 1s on its diagonal and let $\mathbf{a}_{i_k}^T$ be the i_kth row of $\boldsymbol{A}$. Then, at the kth step, we choose $\mathbf{x}^{(k+1)}$ such that $\mathbf{a}_{i_k}^T \mathbf{x}^{(k+1)} = b_{i_k}$, but we allow only x_{i_k} to vary—it was the coordinate that was relaxed. Because $\boldsymbol{A}$ has 1s on its diagonal, we can write this as:

$$x_{i_k}^{(k+1)} = b_{i_k} - \sum_{j \neq i_k} A_{i_k,j} x_j^{(k)} = (\boldsymbol{r}^{(k)})_{i_k} + x_{i_k}^{(k)}.$$

This is exactly the same update as in (2.1). It's also the same update as in the Gauss–Seidel method. The difference with Gauss–Seidel, as it is typically explained, is that it does not maintain an explicit residual, and it chooses coordinates cyclically.

3. ALGORITHMS

We now present three algorithms for approximating $\exp\{\boldsymbol{P}\}\mathbf{e}_c$ designed for matrices from sparse networks satisfying $\|\boldsymbol{P}\|_1 \leq 1$. Two of the methods consist of coordinate relaxation steps on a linear system, $\boldsymbol{M}$, that we construct from a Taylor polynomial approximating $\exp\{\boldsymbol{P}\}$, as explained in Section 3.1. The first algorithm, which we call `gexpm`, applies GS to $\boldsymbol{M}$ with sparse iteration vectors $\mathbf{x}$ and $\mathbf{r}$ and tracks elements of the residual in a heap to enable fast access to the largest entry of the residual. The second algorithm is a close relative of `gexpm`, but it stores *significant entries* of the residual in a queue rather than maintaining a heap. This makes it faster and also turns out to be closely related to a

truncated Gauss–Seidel method. Because of the queue, we call this second method `gexpmq`. The bulk of our analysis in Section 4 studies how these methods converge to an accurate solution.

The third algorithm approximates the product $T_N(\boldsymbol{P})\ \mathbf{e}_c$ using Horner's rule on the polynomial $T_N(\boldsymbol{P})$, in concert with a procedure we call an "incomplete" matrix-vector product (Section 3.5). This procedure deletes all but the largest entries in the vector before performing a matrix-vector product.

We construct `gexpm` and `gexpmq` such that the solutions they produce have guaranteed accuracy, as proved in Section 4. To the contrary, `expmimv` sacrifices predictable accuracy for a guaranteed fast runtime bound.

3.1. Forming a Linear System

We stated a coordinate relaxation method on a linear system. Thus, to use it, we require a linear system whose solution is an approximation of $\exp\{\boldsymbol{P}\}\,\mathbf{e}_c$. Here, we derive such a system using a Taylor polynomial for the matrix exponential. We present the construction for a general matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ because the Taylor polynomial, linear system, and iterative updates are all well defined for any real square matrix $\boldsymbol{A}$; it is only the convergence results that require the additional assumption that $\boldsymbol{A}$ is a graph-related matrix $\boldsymbol{P}$ satisfying $\|\boldsymbol{P}\|_1 \le 1$.

Consider the product of the degree N Taylor polynomial with $\mathbf{e}_c$:

$$T_N(\boldsymbol{A})\ \mathbf{e}_c = \sum_{j=0}^{N} \frac{1}{j!}\boldsymbol{A}^j\mathbf{e}_c \approx \exp\{\boldsymbol{A}\}\,\mathbf{e}_c,$$

and denote the jth term of the sum by $\mathbf{v}_j := \boldsymbol{A}^j\mathbf{e}_c/j!$. Then, $\mathbf{v}_0 = \mathbf{e}_c$, and the later terms satisfy the recursive relation $\mathbf{v}_{j+1} = \boldsymbol{A}\mathbf{v}_j/(j+1)$ for $j = 0, \ldots, N-1$. This recurrence implies that the vectors $\mathbf{v}_j$ satisfy the system

$$\begin{bmatrix} \boldsymbol{I} & & & & \\ -\boldsymbol{A}/1 & \boldsymbol{I} & & & \\ & -\boldsymbol{A}/2 & \ddots & & \\ & & \ddots & \boldsymbol{I} & \\ & & & -\boldsymbol{A}/N & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \vdots \\ \vdots \\ \mathbf{v}_N \end{bmatrix} = \begin{bmatrix} \mathbf{e}_c \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix}. \tag{3.1}$$

If $\hat{\mathbf{v}} = [\hat{\mathbf{v}}_0, \ldots, \hat{\mathbf{v}}_N]^T$ is an approximate solution to (3.1), then we have $\hat{\mathbf{v}}_j \approx \mathbf{v}_j$ for each term, and so $\sum_{j=0}^{N}\hat{\mathbf{v}}_j \approx \sum_{j=0}^{N}\mathbf{v}_j = T_N(\boldsymbol{A})\ \mathbf{e}_c$. Hence, an approximate solution of this linear system yields an approximation of $\exp\{\boldsymbol{A}\}\,\mathbf{e}_c$. Because the end goal is computing $\mathbf{x} := \sum_{j=0}^{N}\hat{\mathbf{v}}_j$, we need not form the blocks $\hat{\mathbf{v}}_j$; instead, all updates that would be made to a block of $\hat{\mathbf{v}}$ are instead made directly to $\mathbf{x}$.

We denote the block matrix by $\boldsymbol{M}$ for convenience; note that the explicit matrix can be expressed more compactly as $(\boldsymbol{I}_{N+1} \otimes \boldsymbol{I}_n - \boldsymbol{S} \otimes \boldsymbol{A})$, where $\boldsymbol{S}$ denotes the $(N+1)\times(N+1)$ matrix with first subdiagonal equal to $[1/1, 1/2, \ldots, 1/N]$, and $\boldsymbol{I}_k$ denotes the $k \times k$ identity matrix. Additionally, the right-hand side $[\mathbf{e}_c, 0, \ldots, 0]^T$ equals $\mathbf{e}_1 \otimes \mathbf{e}_c$. When we apply an iterative method to this system, we often consider sections of the matrix $\boldsymbol{M} = (\boldsymbol{I} \otimes \boldsymbol{I} - \boldsymbol{S} \otimes \boldsymbol{A})$, solution $\hat{\mathbf{v}} = [\hat{\mathbf{v}}_0, \ldots, \hat{\mathbf{v}}_N]^T$, and residual $\mathbf{r} = [\mathbf{r}_0, \ldots, \mathbf{r}_N]^T$

our approximation of $\exp\{\boldsymbol{P}\}\mathbf{e}_c$	$\mathbf{x}$
the degree N Taylor approximation to $\exp\{\boldsymbol{P}\}$	$T_N(\boldsymbol{P})$
term j in the sum $\sum_{k=0}^{N} \boldsymbol{P}^k\mathbf{e}_c/k!$	$\mathbf{v}_j$
the vector $[\mathbf{v}_0, \ldots, \mathbf{v}_N]^T$	$\mathbf{v}$
the $(N+1)n \times (N+1)n$ matrix $\boldsymbol{I} \otimes \boldsymbol{I} - \boldsymbol{S} \otimes \boldsymbol{P}$	$\boldsymbol{M}$
the $(N+1) \times (N+1)$ matrix with first subdiagonal $[1/1, \ldots, 1/N]$	$\boldsymbol{S}$
our GS approximate solution for $\boldsymbol{M}\mathbf{v} = \mathbf{e}_1 \otimes \mathbf{e}_c$ at step k	$\hat{\mathbf{v}}^{(\mathrm{k})}$
our GS residual for $\boldsymbol{M}\mathbf{v} = \mathbf{e}_1 \otimes \mathbf{e}_c$ at step k	$\mathbf{r}^{(\mathrm{k})}$
block j in $\hat{\boldsymbol{v}}^{(k)} = [\hat{\boldsymbol{v}}_0^{(k)}, \ldots, \hat{\boldsymbol{v}}_N^{(k)}]^T$	$\hat{\mathbf{v}}_{\mathrm{j}}^{(\mathrm{k})}$
entry q of the full vector $\hat{\boldsymbol{v}}^{(k)}$	$(\hat{\mathbf{v}}^{(\mathrm{k})})_{\mathrm{q}}$
an "incomplete"" matrix-vector product (Section 3.5)	IMV

Table II Notation for adapting GS.

partitioned into blocks. These vectors each consist of $N+1$ blocks of length n, whereas $\boldsymbol{M}$ is an $(N+1) \times (N+1)$ block matrix, with blocks of size $n \times n$.

In practice, this large linear system is never formed, and we work with it implicitly. That is, when the algorithms `gexpm` and `gexpmq` apply coordinate relaxation to the linear system (3.1), we will restate the iterative updates of each linear solver in terms of these blocks. We describe how this can be done efficiently for each algorithm in the following text.

We summarize the notation introduced thus far, which we will use throughout the rest of the discussion, in Table II.

3.2. Weighting the Residual Blocks

Before presenting the algorithms, it is necessary to develop some understanding of the error introduced using the linear system in (3.1) approximately. Our goal is to show that the error vector arising from using this system's solution to approximate $T_N(\boldsymbol{P})$ is a weighted sum of the residual blocks $\mathbf{r}_j$. This is important here because then we can use the coefficients of $\mathbf{r}_j$ to determine the terminating criterion in the algorithms. To begin our error analysis, we look at the inverse of the matrix $\boldsymbol{M}$.

Lemma 3.1. *Let $\boldsymbol{M} = (\boldsymbol{I}_{N+1} \otimes \boldsymbol{I}_n - \boldsymbol{S} \otimes \boldsymbol{A})$, where $\boldsymbol{S}$ denotes the $(N+1) \times (N+1)$ matrix with first subdiagonal equal to $[1/1, 1/2, \ldots, 1/N]$, and $\boldsymbol{I}_k$ denotes the $k \times k$ identity matrix. Then $\boldsymbol{M}^{-1} = \sum_{k=0}^{N} \boldsymbol{S}^k \otimes \boldsymbol{A}^k$.*

For a proof, see the appendix. Next, we use the inverse of $\boldsymbol{M}$ to define our error vector in terms of the residual blocks from the linear system in Section 3.1. In order to do so, we need to define a family of polynomials associated with the degree N Taylor polynomial for e^x:

$$\psi_j(x) := \sum_{m=0}^{N-j} \frac{j!}{(j+m)!} x^m \tag{3.2}$$

for $j = 0, 1, \ldots, N$. Note that these are merely slightly altered truncations of the well-studied functions $\phi_j(x) = \sum_{m=0}^{\infty} \frac{x^m}{(m+j)!}$ that arise in exponential integrators, a class of

methods for solving initial value problems. These polynomials $\psi_j(x)$ enable us to derive a precise relationship between the error of the polynomial approximation and the residual blocks of the linear system $\boldsymbol{M}\mathbf{v} = \mathbf{e}_1 \otimes \mathbf{e}_c$ as expressed in the following lemma.

Lemma 3.2. *Consider an approximate solution* $\hat{\mathbf{v}} = [\hat{\mathbf{v}}_0; \hat{\mathbf{v}}_1; \cdots; \hat{\mathbf{v}}_N]$ *to the linear system*

$$(\boldsymbol{I}_{N+1} \otimes \boldsymbol{I}_n - \boldsymbol{S} \otimes \boldsymbol{A})[\mathbf{v}_0; \mathbf{v}_1; \cdots; \mathbf{v}_N] = \mathbf{e}_1 \otimes \mathbf{e}_c.$$

Let $\mathbf{x} = \sum_{j=0}^{N} \hat{\mathbf{v}}_j$, *let* $T_N(x)$ *be the degree* N *Taylor polynomial for* e^x, *and define* $\psi_j(x) = \sum_{m=0}^{N-j} \frac{j!}{(j+m)!} x^m$. *Define the residual vector* $\mathbf{r} = [\mathbf{r}_0; \mathbf{r}_1; \ldots; \mathbf{r}_N]$ *by* $\mathbf{r} := \mathbf{e}_1 \otimes \mathbf{e}_c - (\boldsymbol{I}_{N+1} \otimes \boldsymbol{I}_n - \boldsymbol{S} \otimes \boldsymbol{A})\hat{\mathbf{v}}$. *Then the error vector* $T_N(\boldsymbol{A})\,\mathbf{e}_c - \mathbf{x}$ *can be expressed*

$$T_N(\boldsymbol{A})\mathbf{e}_c - \mathbf{x} = \sum_{j=0}^{N} \psi_j(\boldsymbol{A})\mathbf{r}_j.$$

See the appendix for a proof. The essence of the proof is that, using Lemma 3.1, we can write a simple formulation for $\boldsymbol{M}^{-1}\mathbf{r}$, which is the expression for the error.

3.3. Approximating the Taylor Polynomial via Gauss–Southwell

The main idea of `gexpm`, our first algorithm, is to apply GS to the system (3.1) in a way that exploits the sparsity of both $\boldsymbol{M}$ and the input matrix $\boldsymbol{P}$. In particular, we need to adapt the coordinate and residual updates of GS in (2.1) for the system (3.1) by taking advantage of the block structure of the system.

We begin our iteration to solve $\boldsymbol{M}\mathbf{v} = \mathbf{e}_1 \otimes \mathbf{e}_c$ with $\hat{\boldsymbol{v}}^{(0)} = 0$ and $\boldsymbol{r}^{(0)} = \mathbf{e}_1 \otimes \mathbf{e}_c$. Consider an approximate solution after k steps of GS, $\hat{\boldsymbol{v}}^{(k)}$, and residual $\boldsymbol{r}^{(k)}$. The standard GS iteration consists of adding the largest entry of $\boldsymbol{r}^{(k)}$, call it $m^{(k)} := \boldsymbol{r}_q^{(k)}$, to $\hat{\boldsymbol{v}}_q^{(k)}$, and then updating $\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} - m^{(k)}\boldsymbol{M}\mathbf{e}_q$.

We want to rephrase the iteration using the block structure of our system. We will denote the jth block of $\mathbf{r}$ by $\mathbf{r}_{j-1}$, and entry q of $\mathbf{r}$ by $(\mathbf{r})_q$. Note that the entry q corresponds with node i in block $j-1$ of the residual, $\mathbf{r}_{j-1}$. Thus, if the largest entry is $(\boldsymbol{r}^{(k)})_q$, then we write $\mathbf{e}_q = \mathbf{e}_j \otimes \mathbf{e}_i$ and the largest entry in the residual is $m^{(k)} := (\mathbf{e}_j \otimes \mathbf{e}_i)^T \boldsymbol{r}^{(k)} = \mathbf{e}_i^T \mathbf{r}_{j-1}^{(k)}$. The standard GS update to the solution would then add $m^{(k)}(\mathbf{e}_j \otimes \mathbf{e}_i)$ to the iterative solution, $\hat{\boldsymbol{v}}^{(k)}$; but this simplifies to adding $m^{(k)}\mathbf{e}_i$ to block $j-1$ of $\hat{\boldsymbol{v}}^{(k)}$, i.e. $\hat{\boldsymbol{v}}_{j-1}^{(k)}$. In practice, we never form the blocks of $\hat{\mathbf{v}}$; we instead simply add $m^{(k)}\mathbf{e}_i$ to $\boldsymbol{x}^{(k)}$, our iterative approximation of $\exp\{\boldsymbol{P}\}\,\mathbf{e}_c$.

The standard update to the residual is $\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} - m^{(k)}\boldsymbol{M}\mathbf{e}_q$. Using the block notation and expanding $\boldsymbol{M} = \boldsymbol{I} \otimes \boldsymbol{I} - \boldsymbol{S} \otimes \boldsymbol{P}$, the residual update becomes $\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} - m^{(k)}\mathbf{e}_j \otimes \mathbf{e}_i + (\boldsymbol{S}\mathbf{e}_j) \otimes (\boldsymbol{P}\mathbf{e}_i)$. Furthermore, we can simplify the product $\boldsymbol{S}\mathbf{e}_j$ using the structure of $\boldsymbol{S}$: for $j = 1, \ldots, N$, we have $\boldsymbol{S}\mathbf{e}_j = \mathbf{e}_{j+1}/j$; if $j = N+1$, then $\boldsymbol{S}\mathbf{e}_j = 0$.

To implement this iteration, we needed q, the index of the largest entry of the residual vector. To ensure this operation is fast, we store the residual vector's nonzero entries in a heap. This allows $O(1)$ lookup time for the largest magnitude entry each step at the cost of reheaping the residual each time an entry of $\mathbf{r}$ is altered.

We want the algorithm to terminate once its 1-norm error is below a prescribed tolerance, ε. To ensure this, we maintain a weighted sum of the 1-norms of the residual

blocks, $t^{(k)} = \sum_{j=0}^{N} \psi_j(1) \| \boldsymbol{r}_j^{(k)} \|_1$. Now we can reduce the entire `gexpm` iteration to the following:

1. Set $m^{(k)} = (\mathbf{e}_j \otimes \mathbf{e}_i)^T \boldsymbol{r}^{(k)}$, the top entry of the heap, then delete the entry in $\boldsymbol{r}^{(k)}$ so that $(\mathbf{e}_j \otimes \mathbf{e}_i)^T \boldsymbol{r}^{(k+1)} = 0$.
2. Update $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + m^{(k)} \mathbf{e}_i$.
3. If $j < N + 1$, update $\boldsymbol{r}_j^{(k+1)} = \boldsymbol{r}_j^{(k)} + m^{(k)} \boldsymbol{P} \mathbf{e}_i / j$, reheaping $\mathbf{r}$ after each add.
4. Update $t^{(k+1)} = t^{(k)} - \psi_{j-1}(1) |m^{(k)}| + \psi_j(1) |m^{(k)}| / j$.

We show in Theorem 4.2 that iterating until $t^{(k)} \leq \varepsilon$ guarantees a 1-norm accuracy of ε. We discuss the complexity of the algorithms in Section 4.

3.4. Approximating the Taylor Polynomial via Gauss–Seidel

Next, we describe a similar algorithm that stores the residual in a queue to avoid the heap updates. Our original inspiration for this method was the relationship between the Bookmark Coloring Algorithm [9] and the Push method for Personalized PageRank [4]. The rationale for this change is that maintaining the heap in `gexpm` is slow. Remarkably, the final algorithm we create is actually a version of Gauss–Seidel that *skips* updates from insignificant residuals, whereas standard Gauss–Seidel cycles through coordinates of the matrix cyclically in index order. Our algorithm will use the queue to do *one* such pass and maintain *significant entries* of the residual that must be relaxed (and not skipped).

The basic iterative step is the same as in `gexpm`, except that the entry of the residual chosen, say $(\mathbf{e}_j \otimes \mathbf{e}_i)^T \mathbf{r}$, is not selected to be the largest in $\mathbf{r}$. Instead, it is the next entry in a queue storing significant entries of the residual. Then as entries in $\mathbf{r}$ are updated, we place them at the back of the queue, Q. Note that the block-wise nature of our update has the following property: an update from the jth block results in residuals changing in the $(j+1)$st block. Because new elements are added to the tail of the queue, all entries of $\mathbf{r}_{j-1}$ are relaxed before proceeding to $\mathbf{r}_j$.

If carried out exactly as described, this would be equivalent to performing each product $\mathbf{v}_j = \boldsymbol{P} \mathbf{v}_{j-1} / j$ in its entirety. But we want to avoid these full products; so we introduce a rounding threshold for determining whether or not to operate on the entries of the residual as we pop them off of Q.

The rounding threshold is determined as follows. After every entry in $\mathbf{r}_{j-1}$ is removed from the top of Q, then all entries remaining in Q are in block $\mathbf{r}_j$ (remember, this is because operating on entries in $\mathbf{r}_{j-1}$ adds to Q only entries that are from $\mathbf{r}_j$.) Once every entry in $\mathbf{r}_{j-1}$ is removed from Q, we set $Z_j = |Q|$, the number of entries in Q; this is equivalent to the total number of nonzero entries in $\mathbf{r}_j$ before we begin operating on entries of $\mathbf{r}_j$. Then, while operating on $\mathbf{r}_j$, the threshold used is

$$\text{threshold}(\varepsilon, j, N) = \frac{\varepsilon}{N \psi_j(1) Z_j}. \tag{3.3}$$

Then, at each step, an entry is popped off of Q, and if it is larger than this threshold, it is operated on; otherwise, it is simply discarded, and the next entry of Q is considered. Once again, we maintain a weighted sum of the 1-norms of the residual blocks, $t^{(k)} = \sum_{j=0}^{N} \psi_j(1) \| \boldsymbol{r}_j^{(k)} \|_1$, and terminate once $t^{(k)} \leq \varepsilon$, or if the queue is empty.

```
## Estimate column c of
## the matrix exponential vector
# G is the graph as a dictionary-of-sets,
# eps is set to stopping tolerance
def compute_psis(N):
  psis = {}
  psis[N] = 1.
  for i in xrange(N-1,-1,-1):
    psis[i] = psis[i+1]/(float(i+1.))+1.
  return psis
def compute_threshs(eps, N, psis):
  threshs = {}
  threshs[0] = (math.exp(1)*eps/float(N))/psis[0]
  for j in xrange(1, N+1):
    threshs[j] = threshs[j-1]*psis[j-1]/psis[j]
  return threshs
## Setup parameters and constants
N = 6
c = 1 # the column to compute
psis = compute_psis(N)
threshs = compute_threshs(eps,N,psis)
## Initialize variables
x = {} # Store x, r as dictionaries
r = {} # initialize residual
Q = collections.deque() # initialize queue
sumresid = 0.
r[(c,0)] = 1.
Q.append(c)
sumresid += psis[0]
```

```
## Main loop
for j in xrange(0, N):
  qsize = len(Q)
  relaxtol = threshs[j]/float(qsize)
  for qi in xrange(0, qsize):
    i = Q.popleft()
    rij = r[(i,j)]
    if rij < relaxtol:
      continue
    # perform the relax step
    if i not in x: x[i] = 0.
    x[i] += rij
    r[(i,j)] = 0.
    sumresid -= rij*psis[j]
    update = (rij/(float(j)+1.))/len(G[i])
    for u in G[i]:   # for neighbors of i
      next = (u, j+1)
      if j == N-1:
          if u not in x: x[u] = 0.
          x[u] += update
      else:
          if next not in r:
            r[next] = 0.
            Q.append(u)
          r[next] += update
          sumresid += update*psis[j+1]
    # after all neighbors u
    if sumresid < eps: break
  if len(Q) == 0: break
  if sumresid < eps: break
```

Figure 3 A working Python code to implement the `gexpmq` algorithm with a queue.[2] We implicitly normalize the graph structure into a stochastic matrix by dividing by the degree in the computation of `update`.

Step $k+1$ of `gexpmq` is as follows:

1. Pop the top entry of Q, call it $r = (\mathbf{e}_j \otimes \mathbf{e}_i)^T \boldsymbol{r}^{(k)}$, then delete the entry in $\boldsymbol{r}^{(k)}$, so that $(\mathbf{e}_j \otimes \mathbf{e}_i)^T \mathbf{r}^{(k+1)} = 0$.
2. If $r \geq \text{threshold}(\varepsilon, j, N)$ do the following:
   (a) Add $r\mathbf{e}_i$ to $\mathbf{x}_i$.
   (b) Add $r\boldsymbol{P}\mathbf{e}_i/j$ to residual block $\boldsymbol{r}_j^{(k+1)}$.
   (c) For each entry of $\boldsymbol{r}_j^{(k+1)}$ that was updated, add that entry to the back of Q.
   (d) Update $t^{(k+1)} = t^{(k)} - \psi_{j-1}(1)|r| + \psi_j(1)|r|/j$.

We also provide a working Python pseudocode for this method in Figure 3.

We show in the proof of Theorem 4.3 that iterating until $t^{(k)} \leq \varepsilon$, or until all entries in the queue satisfying the threshold condition have been removed, will guarantee that the resulting vector $\mathbf{x}$ will approximate $\exp\{\boldsymbol{P}\}\mathbf{e}_c$ with the desired accuracy.

[2]A full demo of the Python code is available from `https://gist.github.com/dgleich/10224374`.

3.5. A Sparse, Heuristic Approximation

The previously discussed algorithms guarantee that the final approximation attains the desired accuracy ε. Here, we present an algorithm designed to be faster. Because we have no error analysis for this algorithm currently, and because the steps of the method are well defined for any $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, we discuss this algorithm in a more general setting. This method also uses a Taylor polynomial for $\exp\{\boldsymbol{A}\}$, but does not use the linear system constructed for the previous two methods. Instead, the Taylor terms are computed via Horner's rule on the Taylor polynomial. But, rather than a full matrix-vector product, we apply what we call an "incomplete" matrix-vector product (IMV) to compute the successive terms. Thus, our name: `expmimv`. We describe the IMV procedure before describing the algorithm.

Incomplete matrix-vector products (IMV). Given any matrix $\boldsymbol{A}$ and a vector $\mathbf{v}$ of compatible dimension, the IMV procedure sorts the entries of $\mathbf{v}$, then removes all entries except for the largest z. Let $[\mathbf{v}]_z$ denote the vector $\mathbf{v}$ with all but its z largest-magnitude entries deleted. Then we define the *z-incomplete matrix-vector product* of $\boldsymbol{A}$ and $\mathbf{v}$ to be $\boldsymbol{A}[\mathbf{v}]_z$. We call this an *incomplete* product, rather than a rounded matrix-vector product, because, although the procedure is equivalent to rounding to 0 all entries in $\mathbf{v}$ below some threshold, that rounding threshold is not known a priori, and its value will vary from step to step in our algorithm.

There are likely to be a variety of ways to implement these IMVs. Ours computes $[\mathbf{v}]_z$ by filtering all the entries of $\mathbf{v}$ through a min-heap of size z. For each entry of $\mathbf{v}$, if that entry is larger than the minimum value in the heap, then replace the old minimum value with the new entry and reheap; otherwise, set that entry in $[\mathbf{v}]_z$ to be zero, then proceed to the next entry of $\mathbf{v}$. Many similar methods have been explored in the literature before [37].

Horner's rule with IMV. A Horner's rule approach to computing $T_N(\boldsymbol{A})$ considers the polynomial as follows:

$$\begin{aligned}\exp\{\boldsymbol{A}\} \approx T_N(\boldsymbol{A}) &= \boldsymbol{I} + \frac{1}{1!}\boldsymbol{A}^1 + \frac{1}{2!}\boldsymbol{A}^2 + \cdots + \frac{1}{N!}\boldsymbol{A}^N \\ &= \boldsymbol{I} + \frac{1}{1}\boldsymbol{A}\left(\boldsymbol{I} + \frac{1}{2}\boldsymbol{A}\left(\boldsymbol{I} + \cdots + \frac{1}{N-1}\boldsymbol{A}\left(\boldsymbol{I} + \frac{1}{N}\boldsymbol{A}\right)\cdots\right)\right).\end{aligned} \tag{3.4}$$

Using this representation, we can approximate $\exp\{\boldsymbol{A}\}\,\mathbf{e}_c$ by multiplying $\mathbf{e}_c$ by the innermost term, $\boldsymbol{A}/N$, and working from the inside out. More precisely, the `expmimv` procedure is as follows:

1. Fix $z \in \mathbb{N}$.
2. Set $\boldsymbol{x}^{(0)} = \mathbf{e}_c$.
3. For $k = 0, \ldots, N-1$ compute $\boldsymbol{x}^{(k+1)} = \boldsymbol{A}\left([\boldsymbol{x}^{(k)}]_z/(N-k)\right) + \mathbf{e}_c$.

Then, at the end of this process, we have $\boldsymbol{x}^{(N)} \approx T_N(\boldsymbol{A})\mathbf{e}_c$. The vector $[\boldsymbol{x}^{(k)}]_z$ used in each iteration of Step 3 is computed via the IMV procedure described above. For an experimental analysis of the speed and accuracy of `expmimv`, see Section 6.1.1.

Runtime analysis. Now assume that the matrix $\boldsymbol{A}$ in the above presentation corresponds to a graph, and let d be the maximum degree found in the graph related to $\boldsymbol{A}$. Each step of `expmimv` requires identifying the z largest entries of $\boldsymbol{v}^{(k)}$, multiplying $\boldsymbol{A}[\boldsymbol{v}^{(k)}]_z$, then adding $\mathbf{e}_c$. If $\mathbf{v}$ has $\mathrm{nnz}(\mathbf{v})$ nonzeros, and the largest z entries are desired, then computing $[\mathbf{v}]_z$ requires at most $O(\mathrm{nnz}(\mathbf{v})\log(z))$ work: each of the $\mathrm{nnz}(\mathbf{v})$ entries is put into the size-z heap, and each heap update takes at most $O(\log(z))$ operations.

Note that the number of nonzeros in $\boldsymbol{v}^{(k)}$, for any k, can be no more than dz. This is because the product $\boldsymbol{v}^{(k)} = \boldsymbol{A}[\boldsymbol{v}^{(k-1)}]_z$ combines exactly z columns of the matrix: the z columns corresponding to the z nonzeros in $[\boldsymbol{v}^{(k)}]_z$. Because no column of $\boldsymbol{A}$ has more than d nonzeros, the sum of these z columns can have no more than dz nonzeros. Hence, computing $[\boldsymbol{v}^{(k)}]_z$ from $\boldsymbol{v}^{(k)}$ requires at most $O(dz\log(z))$ work. Observe also that the work done in computing the product $\boldsymbol{A}[\boldsymbol{v}^{(k)}]_z$ cannot exceed dz. Since exactly N iterations suffice to evaluate the polynomial, we have proved Theorem 3.3:

Theorem 3.3. *Let $\boldsymbol{A}$ be any graph-related matrix having maximum degree d. Then the* `expmimv` *procedure, using a heap of size z, computes an approximation of* $\exp\{\boldsymbol{A}\}\,\mathbf{e}_c$ *via an N degree Taylor polynomial in work bounded by $O(Ndz\log z)$.*

If $\boldsymbol{A}$ satisfies $\|\boldsymbol{A}\|_1 \leq 1$, then by Lemma 2.1 we can choose N to be a small constant to achieve a coarse $O(10^{-3})$ approximation.

Although the `expmimv` method always has a sublinear runtime, we currently have no theoretical analysis of its accuracy. However, in our experiments we found that a heap size of $z = 10,000$ yields a 1-norm accuracy of $\approx 10^{-3}$ for social networks with millions of nodes (Section 6.1.1). Yet, even for a fixed value of z, the accuracy varied widely. For general-purpose computation of the matrix exponential, we do not recommend this procedure. If, instead, the purpose is to identify large entries of $\exp\{\boldsymbol{P}\}\,\mathbf{e}_c$, our experiments suggest that `expmimv` often accomplishes this task with high accuracy (Section 6.1.1).

4. ANALYSIS

We divide our theoretical analysis into two stages. In the first we establish the convergence of the coordinate relaxation methods, `gexpm` and `gexpmq`, for a class of matrices that includes column-stochastic matrices. Then, in Section 5, we give improved results when the underlying graph has a degree distribution that follows a power-law, which we define formally in Section 5.

4.1. Convergence of Coordinate Relaxation Methods

In this section, we show that both `gexpm` and `gexpmq` converge to an approximate solution with a prescribed 1-norm error ε for any matrix $\boldsymbol{P}$ satisfying $\|\boldsymbol{P}\|_1 \leq 1$.

Consider the large linear system (3.1) using a matrix $\boldsymbol{P}$ with 1-norm bounded by one. Then, by applying both Lemma 3.2 and the triangle inequality, we find that the error in approximately solving the system can be expressed in terms of the residuals in each block:

$$\| T_N(\boldsymbol{P})\,\mathbf{e}_c - \mathbf{x}\|_1 \leq \sum_{j=0}^{N} \|\psi_j(\boldsymbol{P})\|_1 \|\mathbf{r}_j\|_1.$$

Because the polynomials $\psi_j(t)$ have all nonnegative coefficients, and because $\psi_j(\boldsymbol{P})$ is a polynomial in $\boldsymbol{P}$ for each j, we have that $\| \psi_j(\boldsymbol{P}) \|_1 \leq \psi_j(\|\boldsymbol{P}\|_1)$. Finally, using the condition that $\|\boldsymbol{P}\|_1 \leq 1$, we have proved the following:

Lemma 4.1. *Consider the setting from Lemma 3.2 applied to a matrix $\|\boldsymbol{P}\|_1 \leq 1$. Then the norm of the error vector $T_N(\boldsymbol{A})\,\mathbf{e}_c - \mathbf{x}$ associated with an approximate solution is a*

weighted sum of the residual norms from each block:

$$\| T_N(\boldsymbol{P})\, \mathbf{e}_c - \mathbf{x} \|_1 \leq \sum_{j=0}^{N} \psi_j(1) \| \mathbf{r}_j \|_1.$$

Note that this does not require nonnegativity of either $\mathbf{r}$ or $\boldsymbol{P}$, only that $\|\boldsymbol{P}\|_1 \leq 1$; this improves on our original analysis [25].

We now show that our algorithms monotonically decrease the weighted sum of residual norms, and hence, converge to a solution. The analysis differs between the two algorithms (Theorem 4.2 for `gexpm` and Theorem 4.3 for `gexpmq`), but the intuition remains the same: each relaxation step reduces the residual in block j and increases the residual in block $j+1$, but by a smaller amount. Thus, the relaxation steps monotonically reduce the residuals.

Theorem 4.2. *Let $\boldsymbol{P} \in \mathbb{R}^{n\times n}$ satisfy $\|\boldsymbol{P}\|_1 \leq 1$. Then in the notation of Section 3.3, the residual vector after l steps of `gexpm` satisfies $\|\mathbf{r}^{(l)}\|_1 \leq l^{-1/(2d)}$ and the error vector satisfies*

$$\| T_N(\boldsymbol{P})\, \mathbf{e}_c - \mathbf{x} \|_1 \leq \exp(1) \cdot l^{\left(-\frac{1}{2d}\right)}, \tag{4.1}$$

so `gexpm` converges in at most $l = (\exp(1)/\varepsilon)^{2d}$ iterations.

Proof. The iterative update described in Section 3.3 involves a residual block, say $\mathbf{r}_{j-1}$, and a row index, say i, so that the largest entry in the residual at step l is $m^{(l)} = (\mathbf{e}_j \otimes \mathbf{e}_i)^T \boldsymbol{r}^{(l)}$. First, the residual is updated by deleting the value $m^{(l)}$ from entry i of the block $\boldsymbol{r}^{(l)}_{j-1}$, which results in the 1-norm of the residual decreasing by exactly $|m^{(l)}|$. Then, we add $m^{(l)}\boldsymbol{P}\mathbf{e}_i/j$ to $\boldsymbol{r}^{(l)}_j$, which results in the 1-norm of the residual increasing by at most $\|m^{(l)}\boldsymbol{P}\mathbf{e}_i/j\|_1 \leq |m^{(l)}/j|$, since $\|\boldsymbol{P}\mathbf{e}_i\|_1 \leq 1$. Thus, the net change in the 1-norm of the residual will satisfy

$$\|\boldsymbol{r}^{(l+1)}\|_1 \leq \|\boldsymbol{r}^{(l)}\| - |m^{(l)}| + \left|\frac{m^{(l)}}{j}\right|.$$

Note that the first residual block, $\mathbf{r}_0$, has only a single nonzero in it, since $\mathbf{r}_0 = \mathbf{e}_c$ in the initial residual. This means that every step after the first operates on residual $\mathbf{r}_{j-1}$ for $j \geq 2$. Thus, for every step after step 0, we have that $1/j \leq 1/2$. Hence, we have

$$\|\boldsymbol{r}^{(l+1)}\|_1 \leq \|\boldsymbol{r}^{(l)}\| - |m^{(l)}| + \left|\frac{m^{(l)}}{2}\right| = \|\boldsymbol{r}^{(l)}\| - \frac{|m^{(l)}|}{2}.$$

We can lower bound $|m^{(l)}|$, the largest-magnitude entry in the residual, with the average magnitude of the residual. The average value of $\mathbf{r}$ equals $\|\mathbf{r}\|_1$ divided by the number of non zeros in $\mathbf{r}$. After l steps, the residual can have no more than dl non zero elements, because, at most, d non zeros can be introduced in the residual each time $\boldsymbol{P}\mathbf{e}_i$ is added; hence, the average value at step l is lower bounded by $\|\boldsymbol{r}^{(l)}\|_1/dl$. Substituting this into the previous inequality, we have

$$\|\boldsymbol{r}^{(l+1)}\|_1 \leq \|\boldsymbol{r}^{(l)}\| - \frac{|m^{(l)}|}{2} \leq \|\boldsymbol{r}^{(l)}\| - \frac{\|\boldsymbol{r}^{(l)}\|_1}{2dl} = \|\boldsymbol{r}^{(l)}\| \left(1 - \frac{1}{2dl}\right).$$

Iterating this inequality yields the bound $\|\mathbf{r}^{(l)}\|_1 \leq \|\mathbf{r}^{(0)}\|_1 \prod_{k=1}^{l}(1 - 1/(2dk))$, and since $\mathbf{r}^{(0)} = \mathbf{e}_1 \otimes \mathbf{e}_c$, we have $\|\mathbf{r}^{(0)}\|_1 = 1$. Thus, $\|\mathbf{r}^{(l)}\|_1 \leq \prod_{k=1}^{l}(1 - 1/(2dk))$. The first inequality of (4.1) follows from using the facts $(1 + x) \leq e^x$ (for $x > -1$) and $\log(l) < \sum_{k=1}^{l} 1/k$ to write

$$\prod_{k=1}^{l}\left(1 - \frac{1}{2dk}\right) \leq \exp\left\{-\frac{1}{2d}\sum_{k=1}^{l}\frac{1}{k}\right\} \leq \exp\left\{-\frac{1}{2d}\log l\right\} = l^{(-\frac{1}{2d})}.$$

The inequality $(1 + x) \leq e^x$ follows from the Taylor series $e^x = 1 + x + o(x^2)$, and the lower bound for the partial harmonic sum $\sum_{k=1}^{l} 1/k$ follows from the left-hand rule integral approximation $\log(l) = \int_1^l (1/x)\,dx < \sum_{k=1}^{l} 1/k$.

Finally, to prove inequality (4.1), we use the fact from the proof of Lemma 3 in [25] that $\psi_j(1) \leq \psi_0(1) \leq \exp(1)$ for all $j = 0, \ldots, N$. For the readers' convenience, we include a proof of the inequalities $\psi_j(1) \leq \psi_0(1) \leq \exp(1)$ in the appendix. Thus, we have $\|T_N(\boldsymbol{P})\mathbf{e}_c - \mathbf{x}\|_1 \leq \psi_0(1)\sum_{j=0}^{N}\|\mathbf{r}_j\|_1$ by Lemma 4.1. Next, note that $\sum_{j=0}^{N}\|\mathbf{r}_j\|_1 = \|\mathbf{r}\|_1$, because $\mathbf{r} = [\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_N]^T$. Combining these facts, we have $\|T_N(\boldsymbol{P})\mathbf{e}_c - \mathbf{x}\|_1 \leq \exp(1)\|\mathbf{r}\|_1$, which proves the error bound. The bound on the number of iterations required for convergence follows from simplifying the inequality $\exp(1)l^{-1/(2d)} < \varepsilon$. □

Next, we state the convergence result for `gexpmq`.

Theorem 4.3. *Let $\boldsymbol{P} \in \mathbb{R}^{n\times n}$ satisfy $\|\boldsymbol{P}\|_1 \leq 1$. Then, in the notation of Section 3.4, using a threshold of*

$$\mathit{threshold}(\varepsilon, j, N) = \frac{\varepsilon}{N\psi_j(1)Z_j}$$

for each residual block $\mathbf{r}_j$ will guarantee that when `gexpmq` terminates, the error vector satisfies $\| T_N(\boldsymbol{P})\,\mathbf{e}_c - \mathbf{x} \|_1 \leq \varepsilon$.

Proof. From Lemma 4.1 we have $\| T_N(\boldsymbol{P})\mathbf{e}_c - \mathbf{x} \|_1 \leq \sum_{j=0}^{N}\psi_j(1) \| \mathbf{r}_j \|_1$. During the first iteration, we remove the only nonzero entry in $\mathbf{r}_0 = \mathbf{e}_c$ from the queue, then add $\boldsymbol{P}\mathbf{e}_c$ to $\mathbf{r}_1$. Thus, when the algorithm has terminated, we have $\|\mathbf{r}_0\|_1 = 0$, and so we can ignore the term $\psi_0(1)\|\mathbf{r}_0\|_1$ in the sum. In the other N blocks of the residual, $\mathbf{r}_j$ for $j = 1, \ldots, N$, the steps of `gexpmq` delete every entry with magnitude r satisfying $r \geq \varepsilon/(N\psi_j(1)Z_j)$. This implies that all entries remaining in block $\mathbf{r}_j$ are bounded above in magnitude by $\varepsilon/(N\psi_j(1)Z_j)$. Because there can be no more than Z_j nonzero entries in $\mathbf{r}_j$ (by definition of Z_j), we have that $\|\mathbf{r}_j\|_1$ is bounded above by $Z_j \cdot \varepsilon/(N\psi_j(1)Z_j)$. Thus, we have

$$\| T_N(\boldsymbol{P})\,\mathbf{e}_c - \mathbf{x} \|_1 \leq \sum_{j=1}^{N}\psi_j(1)\left(Z_j\frac{\varepsilon}{N\psi_j(1)Z_j}\right),$$

and simplifying completes the proof. □

Currently, we have no theoretical runtime analysis for `gexpmq`. However, because of the algorithm's similarity to `gexpm`, and because of our strong heuristic evidence (presented in Section 6), we believe a rigorous theoretical runtime bound exists.

5. NETWORKS WITH A POWER-LAW DEGREE DISTRIBUTION

In our convergence analysis for `gexpm` in Section 4, the inequalities rely on our estimation of the largest entry in the residual vector at step l, $m^{(l)}$. In this section we achieve a tighter bound on $m^{(l)}$ by using the distribution of the degrees of the underlying graph instead of just d, the maximum degree. In the case that the degrees follow a power-law distribution, we show that the improvement on the bound on $m^{(l)}$ leads to a sublinear runtime for the algorithm.

The degree distribution of a graph is said to follow a power law if the kth largest degree of the graph, $d(k)$, satisfies $d(k) = Q \cdot d \cdot k^{-p}$ for $d = d(1)$ the largest degree in the graph, and positive constants Q and p. A degree distribution of this kind applies to a variety of real-world networks [19]. A more commonly used definition states that the number of nodes having degree k is equal to k^{-a}, but the two definitions can be shown to be equivalent for a certain range of values of their respective exponents, p and a [1]. In this definition, the values of the exponent a for real-world networks range from 2 to 3, frequently closer to 2. These values correspond to $p = 1$ (for $a = 2$) and $p = 1/2$ (for $a = 3$) in the definition that we use. Finally, we note that, though the definition that we use contains an equality, our results hold for any graph with a degree distribution satisfying a "sub" power law, meaning $d(k) \leq Q \cdot k^{-p}$. We now state our main result, then establish some preliminary technical lemmas before finally proving it.

Theorem 5.1. *For a graph with degree distribution following a power law with $p \in (0, 1]$, max degree d, and minimum degree δ, `gexpm` converges to a 1-norm error of ε in work bounded by*

$$work(\varepsilon) = \begin{cases} O\left(\log\left(\frac{1}{\varepsilon}\right)\left(\frac{1}{\varepsilon}\right)^{\frac{3\delta}{2}} d^2 \log(d) \max\left\{\log(d), \log\left(\frac{1}{\varepsilon}\right)\right\}\right) & \text{if } p = 1 \\ O\left(\log\left(\frac{1}{\varepsilon}\right)\left(\frac{1}{\varepsilon}\right)^{\frac{3\delta}{2}} d^{1+\frac{1}{p}} \max\left\{\log(d), \log\left(\frac{1}{\varepsilon}\right)\right\}\right) & \text{if p} \neq 1 \end{cases} \tag{5.1}$$

Note that when the maximum degree satisfies $d < n^r$ for any $r < 1/(1 + 1/p)$, and the minimum degree is a constant independent of n, Theorem 5.1 implies that the runtime scales sublinearly with the graph size, for a fixed 1-norm error of ε.

In practice, having a minimum degree that is a small constant independent of n is extremely common, and values of p are typically near or slightly less than 1. The condition on the maximum degree (that $d < n^r$ for $r < 1/(1 + 1/p)$) is slightly less common, with five of our seven datasets (listed in Table III) satisfying $d < 2.5 \cdot n^{1/2}$.

5.1. Bounding the Number of Nonzeros in the Residual

In the proof of Theorem 4.2, we showed that the residual update satisfies $\|\boldsymbol{r}^{(l+1)}\|_1 \leq \|\boldsymbol{r}^{(l)}\|_1 - m^{(l)}(1 - 1/j)$, where $m^{(l)}$ is the largest entry in $\boldsymbol{r}^{(l)}$, and $\mathbf{r}_{j-1}$ is the section of the residual vector where the entry $m^{(l)}$ is located. We used the bound $m^{(l)} \geq \|\boldsymbol{r}^{(l)}\|_1/(dl)$, which is a lower bound on the average value of all entries in $\boldsymbol{r}^{(l)}$. This follows from the loose upper bound dl on the number of nonzeros in $\boldsymbol{r}^{(l)}$. We also used the naïve upper bound $1/2$ on $(1 - 1/j)$. Here, we prove new bounds on these quantities. For the sake of simpler expressions in the proofs, we express the number of iterations as a multiple of N, i.e., Nl.

Lemma 5.2. *Let $d(k) :=$ the kth largest degree in the graph (with repetition), let $f(m) := \sum_{k=1}^{m} d(k)$, and let $nnz(l) :=$ the number of nonzero entries in $\boldsymbol{r}^{(l)}$. Then after Nl iterations of* `gexpm` *we have*

$$nnz(Nl) \leq Nf(l). \tag{5.2}$$

Proof. At any given step, the number of new nonzeros we can create in the residual vector is bounded above by the largest degree of all the nodes that have not already had their neighborhoods added to $\boldsymbol{r}^{(Nl)}$. If we have already explored the node with degree = $d(1)$, then the next node we introduce to the residual cannot add more than $d(2)$ new nonzeros to the residual, because the locations in $\mathbf{r}$ in which the node $d(1)$ would create nonzeros already have nonzero value.

We cannot conclude $\text{nnz}(l) \leq \sum_{k=1}^{l} d(k) = f(l)$ because this ignores the fact that the same set of $d(1)$ nodes can be introduced into each different time step of the residual, $j = 2, \cdots, N$. Recall that entries of the residual are of the form $\mathbf{e}_j \otimes \mathbf{e}_i$, where i is the index of the node, $i = 1, \cdots, n$; and j is the section of the residual, or time step: $j = 2, \cdots, N$ (note that j skips 1 because the first iteration of GS deletes the only entry in section $j = 1$ of the residual). Recall that the entry of $\mathbf{r}$ corresponding to the 1 in the vector $\mathbf{e}_j \otimes \mathbf{e}_i$ is located in block $\mathbf{r}_{j-1}$. Then, for each degree, $d(1), d(2), \cdots, d(l)$, we have to add nonzeros to that set of $d(k)$ nodes in each of the $N - 1$ different blocks $\mathbf{r}_{j-1}$ before we move on to the next degree, $d(l + 1)$:

$$\begin{aligned}\text{nnz}(Nl) &\leq d(1) + \cdots + d(1) + d(2) + \cdots + d(2) + \cdots + d(l) \\ &\leq Nd(1) + Nd(2) + \cdots + Nd(l),\end{aligned}$$

which equals $N \cdot \sum_{k=1}^{l} d(k) = Nf(l)$. □

Lemma 5.2 enables us to rewrite the inequality $-m_{Nl} \leq -\|\boldsymbol{r}^{(Nl)}\|/(dNl)$ from the proof of Theorem 4.2 as $-m_{Nl} \leq -\|\boldsymbol{r}^{(Nl)}\|/(Nf(l))$. Letting σ_{Nl} represent the value of $(1 - 1/j)$ in step Nl, we can write our upper bound on $\|\boldsymbol{r}^{(Nl+1)}\|_1$ as follows:

$$\|\boldsymbol{r}^{(Nl+1)}\|_1 \leq \|\boldsymbol{r}^{(Nl)}\|_1 \left(1 - \frac{\sigma_{Nl}}{Nf(l)}\right). \tag{5.3}$$

We want to recur this by substituting a similar inequality in for $\|\boldsymbol{r}^{(Nl)}\|_1$, but the indexing does not work out because inequality (5.2) holds only when the number of iterations is of the form Nl. We can overcome this by combining N iterations into one expression:

Lemma 5.3. *In the notation of Lemma 5.2, let $s_{l+1} := \min\{\sigma_{Nl+N}, \sigma_{Nl+N-1}, \cdots, \sigma_{Nl+1}\}$. Then the residual from Nl iterations of* `gexpm` *satisfies*

$$\|\boldsymbol{r}^{(N(l+1))}\|_1 \leq \|\boldsymbol{r}^{(Nl)}\|_1 \left(1 - \frac{s_{l+1}}{Nf(l+1)}\right)^N. \tag{5.4}$$

Proof. By Lemma 5.2 we know that $Nf(l) \geq \text{nnz}(Nl)$ for all l. In the proof of Lemma 5.2 we showed that, during step Nl, no more than $d(l)$ new nonzeros can be created in the residual vector. By the same argument, no more than $d(l + 1)$ nonzeros can be created in

the residual vector during steps $Nl+k$, for $k=1,\ldots,N$. Thus, we have $\text{nnz}(Nl+k) \leq Nf(l)+k\cdot d(l+1) \leq Nf(l)+Nd(l+1) = Nf(l+1)$ for $k=0,1,\ldots,N$. With this we can bound $-m_{Nl+k} \leq -\|\boldsymbol{r}^{(Nl+k)}\|_1/(Nf(l+1))$ for $k=0,\ldots,N$. Recall we defined σ_{Nl} to be the value of $(1-1/j)$ in step Nl. With this in mind, we establish a new bound on the residual decrease at each step:

$$\begin{aligned}
\|\boldsymbol{r}^{(N(l+1))}\|_1 &\leq \|\boldsymbol{r}^{(Nl+N-1)}\|_1 - m_{Nl+N-1}\sigma_{Nl+N-1} \\
&\leq \|\boldsymbol{r}^{(Nl+N-1)}\|_1 - \frac{\|\boldsymbol{r}^{(Nl+N-1)}\|_1\sigma_{Nl+N-1}}{Nf(l+1)} \\
&= \|\boldsymbol{r}^{(Nl+N-1)}\|_1\left(1-\frac{\sigma_{Nl+N-1}}{Nf(l+1)}\right) \\
&\leq \left(\|\boldsymbol{r}^{(Nl+N-2)}\|_1 - m_{Nl+N-2}\sigma_{Nl+N-2}\right)\left(1-\frac{\sigma_{Nl+N-1}}{Nf(l+1)}\right) \\
&\leq \|\boldsymbol{r}^{(Nl+N-2)}\|_1\left(1-\frac{\sigma_{Nl+N-2}}{Nf(l+1)}\right)\left(1-\frac{\sigma_{Nl+N-1}}{Nf(l+1)}\right),
\end{aligned}$$

and recurring, this yields

$$\|\boldsymbol{r}^{(N(l+1))}\|_1 \leq \|\boldsymbol{r}^{(Nl)}\|_1 \prod_{t=1}^{N}\left(1-\frac{\sigma_{Nl+N-t}}{Nf(l+1)}\right).$$

From the definition of s_{l+1} in the statement of the lemma, we can upper bound $-\sigma_{Nl+N-t}$ in the last inequality with $-s_{l+1}$. This enables us to replace the product in the last inequality with $(1-s_{l+1}/(Nf(l+1)))^N$, which proves the lemma. □

Recurring the inequality in (5.4) bounds the residual norm in terms of $f(m)$:

Corollary 5.4. In the notation of Lemma 5.3, after Nl iterations of `gexpm` the residual satisfies

$$\|\boldsymbol{r}^{(Nl)}\|_1 \leq \exp\left\{-\sum_{k=1}^{l}\frac{s_k}{f(k)}\right\} \tag{5.5}$$

Proof. By recurring the inequality of Lemma 5.3, we establish the new bound $\|\boldsymbol{r}^{(Nl)}\|_1 \leq \|\boldsymbol{r}^{(0)}\|_1\prod_{k=1}^{l}(1-s_k/(Nf(k)))^N$. The factor $(1-s_k/(Nf(k)))^N$ can be upper bounded by $\exp\{\sum_{k=1}^{l}(-s_k/(Nf(k)))\cdot N\}$, using the inequality $1-x\leq\exp(-x)$. Cancelling the factors of N and noting that $\|\boldsymbol{r}^{(0)}\|_1 = 1$ completes the proof. □

We want an upper bound on $-\sum_{k=1}^{m}1/f(k)$, so we need a lower bound on $\sum_{k=1}^{m}1/f(k)$. This requires a lower bound on $1/f(k)$, which in turn requires an upper bound on $f(k)$. So next we upper bound $f(k)$ using the degree distribution, which ultimately will allow us to upper bound $\|\boldsymbol{r}^{(Nl)}\|_1$ by an expression of d, the max degree.

5.2. Power-Law Degree Distribution

If we assume that the graph has a power-law degree distribution, then we can bound $d(k)\leq Q\cdot d\cdot k^{-p}$ for constants p and $Q>0$. Let δ denote the minimum degree of the graph and note that, for sparse networks in which $|E| = O(n)$, δ is a small constant (which,

realistically, is $\delta = 1$ in real-world networks). We can assume that $Q = 1$ because it will be absorbed into the constant in the big-O expression for work in Theorem 5.1. With these bounds in place, we will bound $f(k)$ in terms of d, δ, and p.

Lemma 5.5. *Define*

$$C_p = \begin{cases} d(1+\log d) & \text{if } p = 1 \\ \frac{1}{(1-p)} d^{\frac{1}{p}} & \text{if } p \in (0,1)\,. \end{cases} \tag{5.6}$$

Then, in the notation described above we have $f(k) \leq C_p + \delta k$.

Proof. The power-law bound on the degrees states that $d(k) \leq d \cdot k^{-p}$. Note that for $k > (d/\delta)^{\frac{1}{p}}$ the power-law definition of $d(k)$ above implies $d(k) < \delta$, the minimum degree, which is impossible. This leads to two cases: $k > (d/\delta)^{\frac{1}{p}}$ and $k \leq (d/\delta)^{\frac{1}{p}}$.

The sum of the first $\lfloor (d/\delta)^{\frac{1}{p}} \rfloor$ terms is $\sum_{t=1}^{\lfloor (d/\delta)^{\frac{1}{p}} \rfloor} d(t) = f(\lfloor (d/\delta)^{\frac{1}{p}} \rfloor) \leq \sum_{t=1}^{\lfloor (d/\delta)^{\frac{1}{p}} \rfloor} d \cdot t^{-p}$. If we add terms to this sum, then each term that would be added after $d \cdot (\lfloor (d/\delta)^{\frac{1}{p}} \rfloor)^{-p}$ will simply be $d(t) = \delta$, the minimum degree. We can upper bound the sum of any terms beyond $k > (d/\delta)^{\frac{1}{p}}$ by δk. Thus, we have the bound $f(k) \leq f(\lfloor (d/\delta)^{\frac{1}{p}} \rfloor) + \delta k$. In the case that $p = 1$, the bound on the partial harmonic sum used in the proof of Theorem 4.2 yields $f(\lfloor (d/\delta)^{\frac{1}{p}} \rfloor) = f(\lfloor d/\delta \rfloor) \leq d \cdot \sum_{t=1}^{\lfloor d/\delta \rfloor} t^{-1} \leq d(1 + \log(d/\delta)) \leq d(1+\log d)$, proving the $p = 1$ case. If $p \neq 1$, we instead upper bound

$$\begin{aligned} \sum_{t=1}^{\lfloor (d/\delta)^{\frac{1}{p}} \rfloor} t^{-p} \leq d \left(1 + \int_1^{(d/\delta)^{\frac{1}{p}}} x^{-p} \mathrm{d}x \right) &= d \left(1 + \frac{1}{1-p} \left((d/\delta)^{\frac{1}{p}-1} - 1 \right) \right) \\ &\leq \frac{1}{1-p} \left(d \cdot d^{\frac{1}{p}-1} \right), \end{aligned}$$

where the last inequality holds because $1 - \frac{1}{1-p} < 0$ for $p \in (0,1)$. Simplifying yields $\frac{1}{1-p} d^{\frac{1}{p}}$ as the final bound. □

We want to use this tighter bound on $f(k)$ to establish a tighter bound on $\|\boldsymbol{r}^{(Nl)}\|_1$. We can accomplish this using inequality (5.5) if we first bound the sum $\sum_{k=b}^{m} 1/f(k)$ for constants b, m.

Lemma 5.6. *In the notation of Lemma 5.5 we have*

$$\sum_{k=b}^{m} \frac{1}{f(k)} \geq \frac{1}{\delta} \log \left(\frac{\delta m + \delta + C_p}{\delta b + C_p} \right) \tag{5.7}$$

Proof. From Lemma 5.5 we can write $f(k) \leq C_p + \delta k$. Then $1/f(k) \geq 1/(C_p + \delta k)$, and so we have $\sum_{k=1}^{m} 1/f(k) \geq \sum_{k=1}^{m} 1/(C_p + \delta k)$. Using a left-hand rule integral approximation, we get

$$\sum_{k=b}^{m} \frac{1}{C_p + \delta k} \geq \int_b^{m+1} \frac{1}{C_p + \delta x} \mathrm{d}x = \frac{1}{\delta} \log \left(\frac{\delta m + \delta + C_p}{\delta b + C_p} \right). \tag{5.8}$$

□

Plugging (5.7) into (5.5) yields, after some manipulation, our sublinearity result:

Theorem 5.7. *In the notation of Lemma 5.5, for a graph with power-law degree distribution with exponent $p \in (0, 1]$, `gexpm` attains $\|\boldsymbol{r}^{(Nl)}\|_1 < \varepsilon$ in Nl iterations if $l > (3/\delta)(1/\varepsilon)^{3\delta/2}C_p$.*

Proof. Before we can substitute (5.7) into (5.5), we have to control the coefficients s_i. Note that the only entries in $\mathbf{r}$ for which $s_k = (1 - 1/j)$ is equal to $1/2$ are the entries that correspond to the earliest time step, $j = 2$ (in the notation of Section 3.1). There are at most d iterations that have a time step value of $j = 2$, because only the neighbors of the starting node, node c, have nonzero entries in the $j = 2$ time step. Hence, every iteration other than those d iterations must have $s_k \geq (1 - 1/j)$ with $j \geq 3$, which implies $s_k \geq \frac{2}{3}$. We cannot say *which* d iterations of the Nl total iterations occur in time step $j = 2$. However, the first d values of $1/f(k)$ in $\sum_{k=1}^{m} s_k/f(k)$ are the largest in the sum, so by assuming that those d terms have the smaller coefficient (1/2 instead of 2/3), we can guarantee that

$$-\sum_{k=1}^{l} \frac{s_k}{f(k)} < -\sum_{k=1}^{d} \frac{1/2}{f(k)} - \sum_{k=d+1}^{l} \frac{2/3}{f(k)}. \tag{5.9}$$

To make the proof simpler, we omit the sum $\sum_{k=1}^{d}(1/2)/f(k)$ outright. From Corollary 5.4 we have $\|\boldsymbol{r}^{(Nl)}\|_1 \leq \exp\{-\sum_{k=1}^{l} s_k/f(k)\}$, which we can bound above with $\exp\{-(2/3)\sum_{k=d+1}^{l} 1/f(k)\}$, using inequality (5.9). Lemma 5.6 allows us to upper bound the sum $-\sum_{k=d+1}^{l} 1/f(k)$, and simplifying yields

$$\|\boldsymbol{r}^{(Nl)}\|_1 \leq \left(\frac{\delta l + \delta + C_p}{\delta(d+1) + C_p}\right)^{-\frac{2}{3\delta}}.$$

To guarantee $\|\boldsymbol{r}^{(Nl)}\|_1 < \varepsilon$, then, it suffices to show that $\delta l + \delta + C_p > (1/\varepsilon)^{3\delta/2}(\delta d + \delta + C_p)$. This inequality holds if l is greater than $(1/\delta)(1/\varepsilon)^{3\delta/2}(\delta d + \delta + C_p)$. Hence, it is enough for l to satisfy

$$l \geq \frac{3}{\delta}\left(\frac{1}{\varepsilon}\right)^{\frac{3\delta}{2}} C_p.$$

This last line requires the assumption $(\delta d + \delta + C_p) < 3C_p$, which holds only if $\log d$ is larger than δ (in the case $p = 1$), or if $d^{\frac{1}{p}-1}$ is larger than δ (in the case $p \neq 1$). Since we have been assuming that d is a function of n and δ is a constant independent of n, it is safe to assume this. □

With these technical lemmas in place, we are prepared to prove Theorem 5.1 that gives the runtime bound for the `gexpm` algorithm on graphs with a power-law degree distribution.

Proof of Theorem 5.1. Theorem 5.7 states that $l \geq (3/\delta)(1/\varepsilon)^{3\delta/2}C_p$ will guarantee $\|\boldsymbol{r}^{(Nl)}\|_1 < \varepsilon$. It remains to count the number of floating point operations performed in Nl iterations.

Each iteration involves adding a vector that consists of at most d operations, and adding a column of $\boldsymbol{P}$ to the residual, which consists of at most d adds. Then, each entry that is added to the residual requires a heap update. The heap updates at iteration k involve at most $O(\log \mathrm{nnz}(k))$ work, since the residual heap contains at most $\mathrm{nnz}(k)$ nonzeros at that

iteration. The heap is largest at the last iteration, so we can upper bound $\text{nnz}(k) \leq \text{nnz}(Nl)$ for all $k \leq Nl$. Thus, each iteration consists of no more than d heap updates, and so $d \log(\text{nnz}(Nl))$ total operations involved in updating the heap. Hence, after Nl iterations, the total amount of work performed is upper bounded by $O(Nld \log \text{nnz}(Nl))$.

After applying Lemmas 5.2 and 5.5, we know the number of nonzeros in the residual (after Nl iterations) will satisfy $\text{nnz}(Nl) \leq Nf(l) < N(C_p + \delta l)$. Substituting in the expression for l from Theorem 5.7 yields $\text{nnz}(Nl) < N(C_p + \delta(3/\delta)(1/\varepsilon)^{3\delta/2} C_p)$. Upper bounding $C_p < (1/\varepsilon)^{3\delta/2} C_p$ allows us to write

$$\text{nnz}(Nl) < 4N \left(\frac{1}{\varepsilon}\right)^{\frac{3\delta}{2}} C_p. \tag{5.10}$$

We can upper bound the work, $\text{work}(\varepsilon)$, required to produce a solution with error $< \varepsilon$, by using the inequalities in (5.10) and in Theorem 5.7. We expand the bound $\text{work}(\varepsilon) < Nld \log \text{nnz}(Nl)$ to

$$\begin{aligned} &< Nd \left(\left(\frac{3}{\delta}\right) \left(\frac{1}{\varepsilon}\right)^{\frac{3\delta}{2}} C_p \right) \cdot \log \left(4N \left(\frac{1}{\varepsilon}\right)^{\frac{3\delta}{2}} C_p \right) \\ &< N \left(\left(\frac{3}{\delta}\right) \left(\frac{1}{\varepsilon}\right)^{\frac{3\delta}{2}} dC_p \right) \cdot \left(\log(4N) + \frac{3\delta}{2} \log\left(\frac{1}{\varepsilon}\right) + \log(C_p) \right), \end{aligned}$$

which we can upper bound with $O(3N(\frac{1}{\varepsilon})^{\frac{3\delta}{2}} dC_p \cdot 4 \cdot \max\{\log(d), \log(1/\varepsilon)\})$. This proves $\text{work}(\varepsilon) = O(N(1/\varepsilon)^{3\delta/2} dC_p \cdot \max\{\log(C_p), \log(1/\varepsilon)\})$. Replacing N with the expression from Lemma 2.1 yields the bound on total work given in Theorem 5.1.

6. EXPERIMENTAL RESULTS

Here we evaluate the accuracy and speed of each of our algorithms for large real-world and synthetic networks.

Overview. To evaluate accuracy, we examine how well the `gexpmq` function identifies the largest entries of the true solution vector. This is designed to study how well our approximation would work in applications that use large-magnitude entries to find important nodes (Section 6.1). We find that a tolerance of 10^{-4} is sufficient to accurately find the largest entries at a variety of scales. We also provide more insight into the convergence properties of `expmimv` by measuring the accuracy of the algorithm as the size z of its heap varies (Section 6.1.1). Based on these experiments, we recommend setting the subset size for that algorithm to be near $(\text{nnz}(\boldsymbol{P})/n)$ times the number of large entries desired.

We then study how the algorithms scale with graph size. We first compare their runtimes on real-world graphs with varying sizes (Section 6.2). The edge density and maximum degree of the graph will play an important role in the runtime. This study illustrates a few interesting properties of the runtime, which we examine further in an experiment with synthetic forest-fire graphs of up to a billion edges. Here, we find that the runtime scaling grows roughly as d^2, as predicted by our theoretical results.

Real-world networks. The datasets used are summarized in Table III. They include a version of the flickr graph from [12] containing just the largest strongly connected component of the original graph; dblp-2010 from [10], itdk0304 in [35], ljournal-2008

Graph	$\|V\|$	nnz($\boldsymbol{P}$)	nnz($\boldsymbol{P}$)/$\|V\|$	d	$\sqrt{\|V\|}$
itdk0304	190,914	1,215,220	6.37	1,071	437
dblp-2010	226,413	1,432,920	6.33	238	476
flickr-scc	527,476	9,357,071	17.74	9,967	727
ljournal-2008	5,363,260	77,991,514	14.54	2,469	2,316
webbase-2001	118,142,155	1,019,903,190	8.63	3,841	10,870
twitter-2010	33,479,734	1,394,440,635	41.65	768,552	5,786
friendster	65,608,366	3,612,134,270	55.06	5,214	8,100

Table III The real-world datasets we use in our experiments span three orders of magnitude in size.

from [10, 13], twitter-2010 [29] webbase-2001 from [22, 11], and the friendster graph in [36].

Implementation details. All experiments were performed on either a dual processor Xeon e5-2670 system with 16 cores (total) and 256GB of RAM or a single processor Intel i7-990X, 3.47 GHz CPU and 24 GB of RAM. Our algorithms were implemented in C++ using the MATLAB MEX interface. All data structures used are memory efficient: the solution and residual are stored as hash tables using Google's `sparsehash` package.[3]

Comparison. We compare our implementation with a state-of-the-art MATLAB function for computing the exponential of a matrix times a vector, `expmv`, which uses a Taylor polynomial approach [3]. We customized this method with the knowledge that $\|\boldsymbol{P}\|_1 = 1$. This single change results in a great improvement to the runtime of their code. In each experiment, we use as the "true solution" the result of a call to `expmv` using the "single" option, which guarantees a relative backward error bounded by 2^{-24}, or, for smaller problems, we use a Taylor approximation with the number of terms predicted by Lemma 2.1.

6.1. Accuracy on Large Entries

When both `gexpm` and `gexpmq` terminate, they satisfy a 1-norm error of ε. Many applications do not require precise solution *values* but instead would like the correct *set* of large-magnitude entries. To measure the accuracy of our algorithms in identifying these large-magnitude entries, we examine the set precision of the approximations. Recall that the precision of a set T that approximates a desired set S is the size of their intersection divided by the total size: $|S \cap T|/|S|$. Precision values near 1 indicate accurate sets and values near 0 indicate inaccurate sets. We show the precision as we vary the solution tolerance ε for the `gexpmq` method in Figure 4. The experiment we conduct is to take a graph, estimate the matrix exponential for 100 vertices (trials) for our method with various tolerances ε, and compare the sets of the top 100 vertices *that are not neighbors of the seed node* between the true solution and the solution from our algorithm. We remove the starting node and its neighbors because these entries are *always* large, so accurately identifying them is a near-guarantee. The results show that, in median performance, we get the top-100 set completely correct with $\varepsilon = 10^{-4}$ for the small graphs.

[3]The precise code for the algorithms and the experiments are available at `https://www.cs.purdue.edu/homes/dgleich/codes/nexpokit/`.

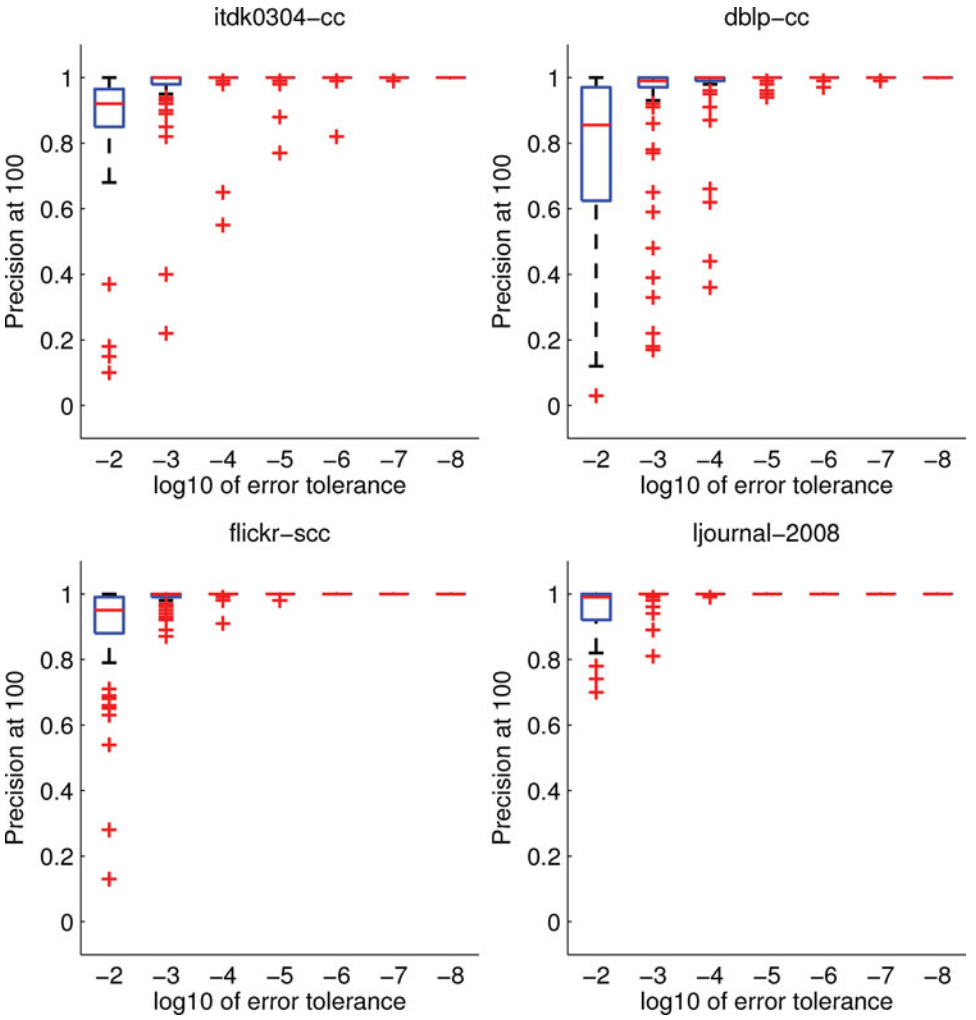

Figure 4 We ran our method `gexpmq` for 100 different seed nodes as we varied the tolerance ε. These four figures show box plots over the 100 trials of the set precision scores of the top-100 results compared to the true top-100 set. These illustrate that a tolerance of 10^{-4} is sufficient for high accuracy.

Next, we study how the work performed by the algorithm `gexpmq` scales with the accuracy. For this study, we pick a vertex at random and vary the maximum number of iterations performed by the `gexpmq` algorithm. Then, we look at the set precision for the top-k sets. The horizontal axis in Figure 5 measures the number of effective matrix-vector products based on the number of edges explored divided by the total number of nonzeros of the matrix. Thus, one matrix-vector product of work corresponds with looking at each nonzero in the matrix once.

The results show that we get good accuracy for the top-k sets up to $k = 1000$ with a tolerance of 10^{-4}, *and* convergence in less than one matrix-vector product, with the sole exception of the flickr network. This network has been problematic for previous studies as

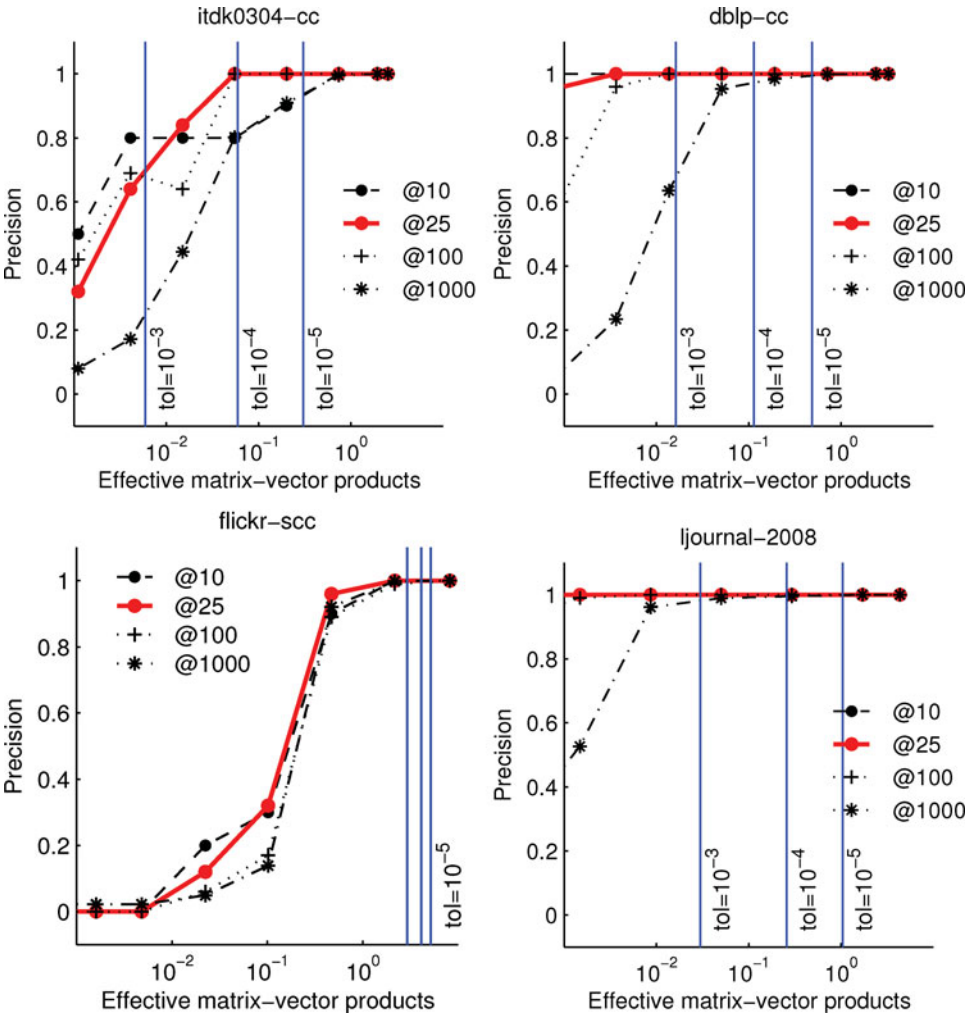

Figure 5 For a single vertex seed, we plot how the precision in the top-k entries varies with the amount of work for the `gexpmq` algorithm. The four pictures show the results of the four small graphs. We see that for every graph, except flickr, the results are good with a tolerance of 10^{-4}, which requires less than one matrix-vector worth of work.

well [12]. Here, we note that we get good results in less than one matrix-vector product, but we do not detect convergence until after a few matrix-vector products worth of work.

6.1.1. Accuracy and nonzeros with incomplete matrix-vector products. The previous studies explored the accuracy of the `gexpmq` method. Our cursory experiments showed that `gexpm` behaves similarly because it also achieves an ε error in the 1-norm. In contrast, the `expmimv` method is rather different in its accuracy because it prescribes only a total size of intermediate heap; we are interested in determining accuracy as we let the heap size increase.

The precise experiment is as follows. For each graph, repeat the following: first, compute 50 node indices uniformly at random. For each node index, use `expmimv` to compute $\exp\{\boldsymbol{P}\}\mathbf{e}_c$, using different values for the heap size parameter: $z = 100, 200,$

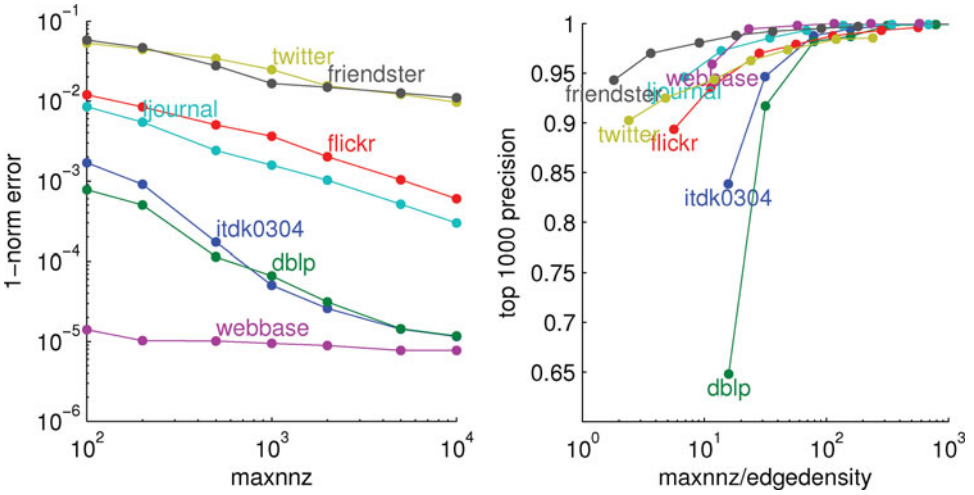

Figure 6 Here, we display the performance of the `expmimv` method. The left figure shows the 1-norm error compared with the number of nonzeros retained in the matrix-vector products on our set of graphs. The groups of curves with similar convergence have comparable edge densities. The right figure shows how the set precision converges as we increase the number of nonzeros, relative to the edge density of the graph ($\text{nnz}(\boldsymbol{P})/n$).

500, 1,000, 2,000, 5,000, 10,000. Figure 6 displays the median of these 50 trials for each parameter setting for both the 1-norm error and the top-1,000 set precision of the `expmimv` approximations. (The results for the top-100 precision, as in the previous study, were effectively the same.) The plot of the 1-norm error in Figure 6 (left) displays clear differences for the various graphs, yet there are pairs of graphs that have nearby errors (such as itdk0304 and dblp). The common characteristic for each pair appears to be the edge density of the graph. We see this effect more strongly in the right plot, where we look at the precision in the top-1,000 set.

Again, set precision improves for all datasets as more nonzeros are used. If we normalize by edge density (by dividing the number of nonzeros used by the edge density of each graph) then the curves cluster. Once the ratio (nonzeros used/edge density) reaches 100, `expmimv` attains a set precision of over 0.95 for *all* datasets on the 1,000 largest-magnitude nodes, regardless of the graph size. We view this as strong evidence that this method should be useful in many applications for which precise numeric values are not required.

6.2. Runtime and Input-Size

Because the algorithms presented here are intended to be fast on large, sparse networks, we continue our study by investigating how their speed scales with data size. Figure 7 displays the median runtime for each graph, where the median is taken over 100 trials for the smaller graphs, and 50 trials for the twitter and friendster datasets. Each trial consists of computing a column $\exp\{\boldsymbol{P}\}\,\mathbf{e}_c$ for a randomly chosen c. All algorithms use a fixed 1-norm error tolerance of 10^{-4}; the `expmimv` method uses 10,000 nonzeros, which may not achieve our desired tolerance but identifies the right set with high probability, as evidenced in the experiment of Section 6.1.1.

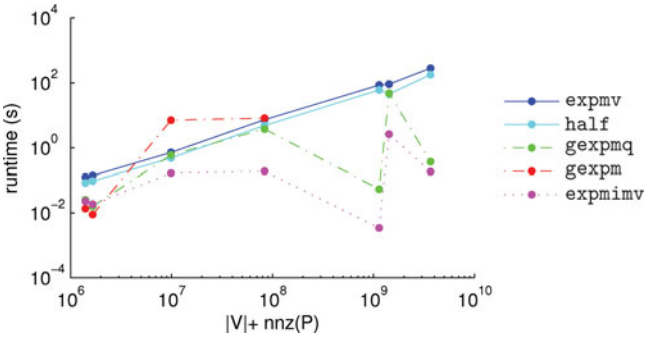

Figure 7 The median runtime of our methods for the seven graphs over 100 trials (only 50 trials for the largest two datasets), compared with the method `expmv` of [3] using the `single` and `half` accuracy settings (which we label as `expmv` and `half`, respectively). The coordinate relaxation methods have highly variable runtimes, but can be very fast on graphs such as webbase (the point nearest 10^9 on the x-axis). We did not run the `gexpm` function for matrices larger than the livejournal graph.

6.3. Runtime Scaling

The final experimental study we conduct attempts to better understand the runtime scaling of the `gexpmq` method. This method yields a prescribed accuracy ε more rapidly than `gexpm`, but the study with real-world networks did not show a clear relationship between error and runtime. We conjecture that this is because the edge density varies too much between our graphs. Consequently, we study the runtime scaling on forest-fire synthetic graphs [30]. We use a symmetric variation on the forest-fire model with a single "burning" probability. We vary the number of vertices generated by the model to get graphs ranging from around 10,000 vertices to around 100,000,000 vertices.

The runtime distributions for burning probabilities, p_f, of 0.4 and 0.48 are shown in the left two plots of Figure 8. With $p_f = 0.48$, the graph is fairly dense—more like

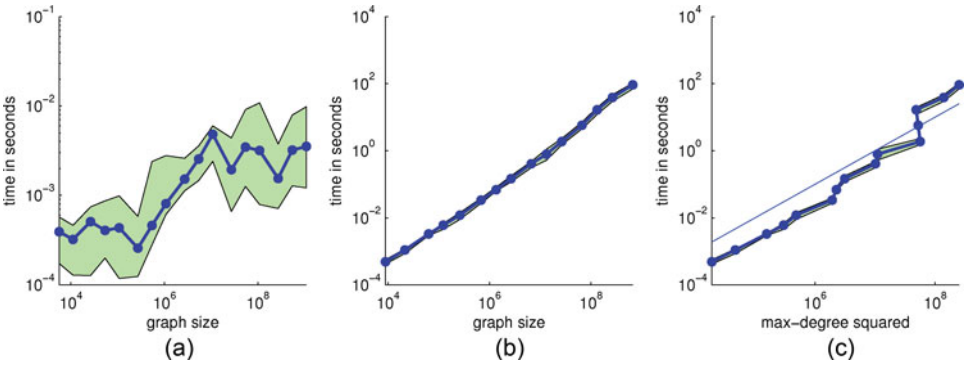

Figure 8 The distribution of runtimes for the `gexpm` method on two forest-fire graphs: (a) $p_f = 0.4$, (b) $p_f = 0.48$, of various graph sizes, where graph size is computed as the sum of the number of vertices and the number of nonzeros in the adjacency matrix. The thick line is the median runtime over 50 trials, and the shaded region shows the 25% to 75% quartiles (the shaded region is very tight for the second two figures). The final plot (c) shows the relationship between the max-degree squared and the runtime in seconds. This figure shows that the runtime scales in a nearly linear relationship with the max-degree squared, as predicted by our theory. The large deviations from the line of best fit might be explained by the fact that only a single forest-fire graph was generated for each graph size.

the friendster network—whereas the graph with $p_f = 0.4$ is highly sparse and is a good approximation for the webbase graph. Even with billions of edges, it takes less than 0.01 seconds for `gexpmq` to produce a solution with 1-norm error $\varepsilon = 10^{-4}$ on this sparse graph. For $p_f = 0.48$, the runtime grows with the graph size.

We find that the scaling of d^2 seems to match the empirical scaling of the runtime (right plot), which is a plausible prediction based on Theorem 5.1. (Recall that one of the log factors in the bound of Theorem 5.1 arose from the heap updates in the `gexpm` method.) These results show that our method is extremely fast when the graph is sufficiently sparse, but it does slow down when running on networks with higher edge density.

7. CONCLUSIONS AND FUTURE WORK

The algorithms presented in this article compute a column of the matrix exponential of sparse matrices $\boldsymbol{P}$ satisfying $\|\boldsymbol{P}\|_1 \leq 1$. They range from `gexpm`, with its strong theoretical guarantees on accuracy and runtime, to `gexpmq`, which drops the theoretical runtime bound but is empirically faster and provably accurate, to `expmimv`, which does not guarantee accuracy but with which there is an even smaller runtime bound. We also showed that they outperform a state-of-the-art Taylor method, `expmv`, to compute a column of the matrix exponential experimentally. This suggests that these methods have the potential to become methods of choice for computing columns of the matrix exponential on social networks.

We anticipate that our method will be useful in scenarios where the goal is to compare the matrix exponential with other network measures, such as the personalized PageRank vector. We have used a variant of the ideas presented here, along with a new runtime bound for a degree-weighted error, to perform such a comparison for the task of community detection in networks [26].

Functions beyond the exponential. We believe we can generalize the results here to apply to a larger class of inputs: namely, sparse matrices $\boldsymbol{A}$ satisfying $\|\boldsymbol{A}\|_1 \leq c$ for small c. Furthermore, all three algorithms described in this article can be generalized to work for functions other than e^x. In our future work, we also plan to explore better polynomial approximations of the exponential [33].

Improved implementations. Because of the slowdown due to the heap updates in `gexpm`, we hope to improve on our current heap-based algorithm by implementing a new data structure that can provide fast access to large entries as well as fast update to and deletion of entries. One of the possibilities we wish to explore is a Fibonacci heap. We also plan to explore parallelizing the algorithms using asynchronous methods. Recent analysis suggests that strong, rigorous runtime guarantees are possible [5].

New analysis. Finally, we hope to improve on the analysis for `expmimv`. Namely, we believe there is a rigorous relationship between the input graph size, the nonzeros retained, the Taylor degree selected, and the error of the solution vector produced by `expmimv`. Recently, one such method was rigorously analyzed [16], which helped to establish new bounds on the planted clique problem.

FUNDING

This research was supported by NSF CAREER award CCF-1149756.

APPENDIX: PROOFS

Lemma 2.1. Let $\boldsymbol{P}$ and $\mathbf{b}$ satisfy $\|\boldsymbol{P}\|_1, \|\mathbf{b}\|_1 \leq 1$. Then, choosing the degree, N, of the Taylor approximation, $T_N(\boldsymbol{P})$, such that $N \geq 2\log(1/\varepsilon)$ and $N \geq 3$ will guarantee

$$\|\exp\{\boldsymbol{P}\}\,\mathbf{b} - T_N(\boldsymbol{P})\,\mathbf{b}\,\|_1 \leq \varepsilon.$$

Proof. We first show that the degree N Taylor approximation satisfies

$$\|\exp\{\boldsymbol{P}\}\,\mathbf{b} - T_N(\boldsymbol{P})\,\mathbf{b}\,\|_1 \leq \frac{1}{N!N}. \tag{A.1}$$

To prove this, observe that our approximation's remainder, $\|\exp\{\boldsymbol{P}\}\,\mathbf{b} - T_N(\boldsymbol{P})\mathbf{b}\|_1$, equals $\left\|\sum_{k=N+1}^{\infty} \boldsymbol{P}^k\mathbf{e}_c/k!\right\|_1$. Using the triangle inequality, we can upper bound this by $\sum_{k=N+1}^{\infty} \|\boldsymbol{P}^k\|_1\|\mathbf{e}_c\|_1/k!$. We then have

$$\|\exp\{\boldsymbol{P}\}\,\mathbf{b} - T_N(\boldsymbol{P})\,\mathbf{b}\,\|_1 \leq \sum_{k=N+1}^{\infty} \frac{1}{k!}$$

because $\|\boldsymbol{P}\|_1 \leq 1$ and $\|\mathbf{e}_c\|_1 = 1$. By factoring out $1/(N+1)!$ and majorizing $(N+1)!/(N+1+k)! \leq 1/(N+1)^k$ for $k \geq 0$, we finish:

$$\|\exp\{\boldsymbol{P}\}\,\mathbf{b} - T_N(\boldsymbol{P})\,\mathbf{b}\,\|_1 \leq \left(\frac{1}{(N+1)!}\right)\sum_{k=0}^{\infty}\left(\frac{1}{N+1}\right)^k = \frac{1}{(N+1)!}\frac{N+1}{N}, \tag{A.2}$$

where the last step substitutes the limit for the convergent geometric series.

Next, we prove the lemma. We will show that $2\log(N!) > N\log N$, then use this to relate $\log(N!N)$ to $\log(\varepsilon)$. First we write $2\log(N!) = 2\cdot\sum_{k=0}^{N-1}\log(1+k) = \sum_{k=0}^{N-1}\log(1+k)+\sum_{k=0}^{N-1}\log(1+k)$. By noting that $\sum_{k=0}^{N-1}\log(1+k) = \sum_{k=0}^{N-1}\log(N-k)$, we can express $2\log(N!) = \sum_{k=0}^{N-1}\log(k+1) + \sum_{k=0}^{N-1}\log(N-k)$, which is equal to $\sum_{k=0}^{N-1}\log((k+1)(N-k))$. Finally, $(k+1)(N-k) = N+Nk-k^2-k = N+k(N-k-1) \geq N$ because $N \geq k+1$, and so

$$2\log(N!) \geq \sum_{k=0}^{N-1}\log(N) = N\log(N). \tag{A.3}$$

By the first claim, we know that $1/N!N < \varepsilon$ guarantees the error we want, but for this inequality to hold, it is sufficient to have $\log(N!N) > \log(1/\varepsilon)$. Certainly, if $\log(N!) > \log(1/\varepsilon)$ then $\log(N!N) > \log(1/\varepsilon)$ holds, so by (A.3) it suffices to choose N satisfying $N\log(N) > 2\log(1/\varepsilon)$. Finally, for $N \geq 3$ we have $\log(N) > 1$, and so Lemma 2.1 holds for $N \geq 3$. □

Lemma 3.1. Let $\boldsymbol{M} = (\boldsymbol{I}_{N+1}\otimes\boldsymbol{I}_n - \boldsymbol{S}\otimes\boldsymbol{A})$, where $\boldsymbol{S}$ denotes the $(N+1)\times(N+1)$ matrix with first sub-diagonal equal to $[1/1, 1/2, \ldots, 1/N]$, and $\boldsymbol{I}_k$ denotes the $k\times k$ identity matrix. Then $\boldsymbol{M}^{-1} = \sum_{k=0}^{N}\boldsymbol{S}^k\otimes\boldsymbol{A}^k$.

Proof. Because S is a subdiagonal matrix, it is nilpotent, with $S^{N+1} = 0$. This implies that $S \otimes A$ is also nilpotent, since $(S \otimes A)^{N+1} = S^{N+1} \otimes A^{N+1} = 0 \otimes A = 0$. Thus, we have

$$\begin{aligned} M\left(\sum_{k=0}^{N} S^k \otimes A^k\right) &= (I - S \otimes A)\left(\sum_{k=0}^{N} S^k \otimes A^k\right) \\ &= I - (S \otimes A)^{N+1} \qquad \text{the sum telescopes,} \end{aligned}$$

which is I. This proves $(\sum_{k=0}^{N} S^k \otimes A^k)$ is the inverse of M. □

Lemma 3.2. Consider an approximate solution $\hat{\mathbf{v}} = [\hat{\mathbf{v}}_0; \hat{\mathbf{v}}_1; \cdots; \hat{\mathbf{v}}_N]$ to the linear system

$$(I_{N+1} \otimes I_n - S \otimes A)[\mathbf{v}_0; \mathbf{v}_1; \cdots; \mathbf{v}_N] = \mathbf{e}_1 \otimes \mathbf{e}_c.$$

Let $\mathbf{x} = \sum_{j=0}^{N} \hat{\mathbf{v}}_j$, let $T_N(x)$ be the degree N Taylor polynomial for e^x, and define $\psi_j(x) = \sum_{m=0}^{N-j} \frac{j!}{(j+m)!} x^m$. Define the residual vector $\mathbf{r} = [\mathbf{r}_0; \mathbf{r}_1; \ldots; \mathbf{r}_N]$ by $\mathbf{r} := \mathbf{e}_1 \otimes \mathbf{e}_c - (I_{N+1} \otimes I_n - S \otimes A)\hat{\mathbf{v}}$. Then, the error vector $T_N(A)\mathbf{e}_c - \mathbf{x}$ can be expressed

$$T_N(A)\mathbf{e}_c - \mathbf{x} = \sum_{j=0}^{N} \psi_j(A)\mathbf{r}_j.$$

Proof. Recall that $\mathbf{v} = [\mathbf{v}_0; \mathbf{v}_1; \cdots; \mathbf{v}_N]$ is the solution to (3.1), and our approximation is $\hat{\mathbf{v}} = [\hat{\mathbf{v}}_0; \hat{\mathbf{v}}_1; \cdots; \hat{\mathbf{v}}_N]$. We showed in Section 3.1 that the error $T_N(A)\mathbf{e}_c - \mathbf{x}$ is in fact the sum of the error blocks $\mathbf{v}_j - \hat{\mathbf{v}}_j$. Now, we will express the error blocks $\mathbf{v}_j - \hat{\mathbf{v}}_j$ in terms of the residual blocks of the system (3.1), i.e. $\mathbf{r}_j$.

The following relationship between the residual vector and solution vector always holds: $\mathbf{r} = \mathbf{e}_1 \otimes \mathbf{e}_c - M\hat{\mathbf{v}}$, so premultiplying by M^{-1} yields $M^{-1}\mathbf{r} = \mathbf{v} - \hat{\mathbf{v}}$, because $\mathbf{v} = M^{-1}\mathbf{e}_1 \otimes \mathbf{e}_c$ exactly, by definition of $\mathbf{v}$. Note that $M^{-1}\mathbf{r} = \mathbf{v} - \hat{\mathbf{v}}$ is the error vector for the linear system (3.1). Substituting the expression for M^{-1} in Lemma 3.1 yields

$$\begin{bmatrix} \mathbf{v}_0 - \hat{\mathbf{v}}_0 \\ \mathbf{v}_1 - \hat{\mathbf{v}}_1 \\ \vdots \\ \mathbf{v}_N - \hat{\mathbf{v}}_N \end{bmatrix} = \left(\sum_{k=0}^{N} S^k \otimes A^k\right) \begin{bmatrix} \mathbf{r}_0 \\ \mathbf{r}_1 \\ \vdots \\ \mathbf{r}_N \end{bmatrix}. \tag{A.4}$$

Let $\mathbf{e}$ be the vector of all 1s of appropriate dimension. Then observe that premultiplying (A.4) by $(\mathbf{e}^T \otimes I)$ yields, on the left-hand side, $\sum_{j=0}^{N}(\mathbf{v}_j - \hat{\mathbf{v}}_j)$. Now, we can accomplish our goal of expressing $\sum_{j=0}^{N}(\mathbf{v}_j - \hat{\mathbf{v}}_j)$ in terms of the residual blocks $\mathbf{r}_j$ by expressing the right-hand side $(\mathbf{e}^T \otimes I)(\sum_{k=0}^{N} S^k \otimes A^k)\mathbf{r}$ in terms of the blocks $\mathbf{r}_j$. So next we consider the product of a fixed block $\mathbf{r}_{j-1}$ with a particular term $(S^k \otimes A^k)$. Note that, because $\mathbf{r}_{j-1}$ is in block-row j of $\mathbf{r}$, it multiplies with only the block-column j of $(S^k \otimes A^k)$, so we now examine the blocks in block-column j of $(S^k \otimes A^k)$.

Because S is a subdiagonal matrix, there is only one nonzero in each column of S^k, for each $k = 0, \ldots, N$. As mentioned in Section 3.3, $S\mathbf{e}_j = \mathbf{e}_{j+1}/j$ when $j < N + 1$, and

0 otherwise. This implies that

$$\boldsymbol{S}^k \mathbf{e}_j = \begin{cases} \dfrac{(j-1)!}{(j-1+k)!}\mathbf{e}_{j+k}, & \text{if } 0 \le k \le N+1-j \\ 0, & \text{otherwise.} \end{cases}$$

Thus, block-column j of $(\boldsymbol{S}^k \otimes \boldsymbol{A}^k)$ contains only a single nonzero block, $(j-1)!\boldsymbol{A}^k/(j-1+k)!$, for each $k = 0, \ldots, N+1-j$. Hence, summing the $n \times n$ blocks in block-column j of all powers $(\boldsymbol{S}^k \otimes \boldsymbol{A}^k)$ for $k = 0, \ldots, N$ yields

$$\sum_{k=0}^{N+1-j} \frac{(j-1)!}{(j-1+k)!}\boldsymbol{A}^k. \tag{A.5}$$

as the matrix coefficient of the term $\mathbf{r}_{j-1}$ in the expression $(\mathbf{e}^T \otimes \boldsymbol{I})(\sum_{k=0}^{N} \boldsymbol{S}^k \otimes \boldsymbol{A}^k)\mathbf{r}$. Thus, we have

$$(\mathbf{e}^T \otimes \boldsymbol{I}) \left(\sum_{k=0}^{N} \boldsymbol{S}^k \otimes \boldsymbol{A}^k \right) \mathbf{r} = \sum_{j=1}^{N+1} \left(\sum_{k=0}^{N+1-j} \frac{(j-1)!}{(j-1+k)!)} \boldsymbol{A}^k \right) \mathbf{r}_{j-1}$$

Finally, reindexing so that the outer summation on the right-hand side goes from $j = 0$ to N, then substituting our definition for $\psi_j(\boldsymbol{A}) = \sum_{k=0}^{N-m} \frac{j!}{(j+k)!}\boldsymbol{A}^k$, we have that $\sum_{j=0}^{N}(\mathbf{v}_j - \hat{\mathbf{v}}_j) = \sum_{j=0}^{N} \psi_j(\boldsymbol{A})\mathbf{r}_j$, as desired. □

Lemma 3.2. *from [25] Let* $\psi_j(x) = \sum_{m=0}^{N-j} \frac{j!}{(j+m)!}x^m$. *Then* $\psi_j(1) \le \psi_0(1) \le \exp(1)$.

Proof. By definition, $\psi_j(1) = \sum_{m=0}^{N-j} \frac{j!}{(j+m)!}$ and $\psi_{j+1}(1) = \sum_{m=0}^{N-j-1} \frac{(j+1)!}{(j+1+m)!}$. Note that the sum for $\psi_j(1)$ has more terms, and the general terms of the two summations satisfy $\frac{j!}{(j+m)!} \ge \frac{(j+1)!}{(j+1+m)!}$ because multiplying both sides by $\frac{(j+m)!}{j!}$ yields $1 \ge \frac{j+1}{j+1+m}$. Hence, $\psi_j(1) \ge \psi_{j+1}(1)$ for $j = 0, \ldots, N-1$, and so the statement follows.

To see that $\psi_0(1) \le \exp(1)$, note that $\psi_0(1)$ is the degree N Taylor polynomial expression for $\exp(1)$, which is a finite approximation of the Taylor series, an infinite sum of positive terms; hence, $\psi_0(1) \le \exp(1)$. □

REFERENCES

[1] L. A. Adamic. "Zipf, Power-Laws, and Pareto – A Ranking Tutorial." Available online (`http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html`), 2002.

[2] M. Afanasjew, M. Eiermann, O. G. Ernst, and S. Güttel. "Implementation of a Restarted Krylov Subspace Method for the Evaluation of Matrix Functions." *Linear Algebra Appl.*. 429:10 (2008), 2293–2314.

[3] A. H. Al-Mohy and N. J. Higham. "Computing the Action of the Matrix Exponential, with an Application to Exponential Integrators." *SIAM J. Sci. Comput.* 33:2 (2011), 488–511.

[4] R. Andersen, F. Chung, and K. Lang. "Local Graph Partitioning Using PageRank Vectors." In *Proceedings of FOCS2006*, 475-486. IEEE Computer Society, 2006.

[5] H. Avron, A. Druinsky, and A. Gupta. Revisiting Asynchronous Linear Solvers: Provable Convergence Rate Through Randomization. In *Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2014.

[6] A. L. Barabási and R. Albert. "Emergence of Scaling in Random Networks." *Science* 286:5439 (1999), 509–512.

[7] M. Benzi and P. Boito. "Quadrature Rule-Based Bounds for Functions of Adjacency Matrices." *Linear Algebra and its Applications* 433:3 (2010), 637–652.

[8] M. Benzi and N. Razouk. "Decay Bounds and O(n) Algorithms for Approximating Functions of Sparse Matrices." *Electronic Transactions on Numerical Analysis* 28 (2007), 16–39 Available online (`http://www.emis.ams.org/journals/ETNA/vol.28.2007/pp16-39.dir/pp16-39.pdf`).

[9] P. Berkhin. "Bookmark-Coloring Algorithm for Personalized PageRank Computing." *Internet Mathematics* 3:1 (2007), 41–62.

[10] P. Boldi, M. Rosa, M. Santini, and S. Vigna. "Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks." In *Proceedings of the 20th WWW2011*, pp. 587–596. New York, NY: ACM, 2011.

[11] P. Boldi and S. Vigna. "Codes for the World Wide Web." *Internet Mathematics* 2:4 (2005), 407–429.

[12] F. Bonchi, P. Esfandiar, D. F. Gleich, C. Greif, and L. V. S. Lakshmanan. "Fast Matrix Computations for Pairwise and Columnwise Commute Times and Katz Scores." *Internet Mathematics* 8:1-2 (2012), 73–112.

[13] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. "On Compressing Social Networks." In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pp. 219–228, New York, NY, USA:ACM, 2009.

[14] F. Chung. "The Heat Kernel as the PageRank of a Graph." *Proceedings of the National Academy of Sciences* 104:50 (2007), 19735–19740.

[15] F. Chung and O. Simpson. "Solving Linear Systems with Boundary Conditions Using Heat Kernel PageRank." In *Algorithms and Models for the Web Graph*, pp. 203–219. Springer, 2013.

[16] Y. Deshpande and A. Montanari. "Finding Hidden Cliques of Size $\sqrt{N/e}$ in Nearly Linear Time." *arXiv*, math.PR:1304.7047Available online (`http://arxiv.org/abs/1304.7047`), 2013.

[17] E. Estrada. "Characterization of 3D Molecular Structure." *Chemical Physics Letters* 319:5-6 (2000), 713–718.

[18] E. Estrada and D. J. Higham. "Network Properties Revealed Through Matrix Functions." *SIAM Review* 52:4 (2010), 696–714.

[19] M. Faloutsos, P. Faloutsos, and C. Faloutsos. "On Power-Law Relationships of the Internet Topology." *SIGCOMM Comput. Commun. Rev.* 29: (1999), 251–262.

[20] A. Farahat, T. LoFaro, J. C. Miller, G. Rae, and L. A. Ward. "Authority Rankings from HITS, PageRank, and SALSA: Existence, Uniqueness, and Effect of Initialization." *SIAM Journal on Scientific Computing* 27:4 (2006), 1181–1201.

[21] E. Gallopoulos and Y. Saad. "Efficient Solution of Parabolic Equations by Krylov Approximation Methods." *SIAM J. Sci. Stat. Comput.* 13:5 (1992), 1236–1264.

[22] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. "Webbase: A Repository of Web Pages." *Computer Networks* 33:1-6 (2000), 277–293.

[23] M. Hochbruck and C. Lubich. "On Krylov Subspace Approximations to the Matrix Exponential Operator." *SIAM J. Numer. Anal.* 34:5 (1997), 1911–1925.

[24] G. Jeh and J. Widom. "Scaling Personalized Web Search." In *Proceedings of the 12th International Conference on the World Wide Web*, pp. 271–279. ACM, 2003.

[25] K. Kloster and D. F. Gleich. "A Nearly-Sublinear Method for Approximating a Column of the Matrix Exponential for Matrices from Large, Sparse Networks." In *Algorithms and Models for the Web Graph*, edited by A. Bonato, M. Mitzenmacher, and P. Praat, pp. 68–79, Lecture Notes in Computer Science 8305. New York, NY: Springer International, 2013.

[26] K. Kloster and D. F. Gleich. "Heat Kernel Based Community Detection." In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pp. 1386–1395, New York, NY, USA:ACM 2014.

[27] R. I. Kondor and J. D. Lafferty. "Diffusion Kernels on Graphs and Other Discrete Input Spaces." In *Proceedings of the 19th Annual International Conference on Machine Learning, ICML '02*, pp. 315–322. Morgan Kaufman, 2002.

[28] J. Kunegis and A. Lommatzsch. "Learning Spectral Graph Transformations for Link Prediction." In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09*, pp. 561–568. New York, NY, USA: ACM, 2009.

[29] H. Kwak, C. Lee, H. Park, and S. Moon. "What is Twitter, a Social Network or a News Media?" In *WWW '10: Proceedings of the 19th International Conference on World Wide Web*, pp. 591–600. New York, NY, USA: ACM, 2010.

[30] J. Leskovec, J. Kleinberg, and C. Faloutsos. "Graph Evolution: Densification and Shrinking Diameters." *ACM Trans. Knowl. Discov. Data*. 1:(2007), 1–41.

[31] Z. Q. Luo and P. Tseng. "On the Convergence of the Coordinate Descent Method for Convex Differentiable Minimization." *J. Optim. Theory Appl.* 72:1 (1992), 7–35.

[32] C. Moler and C. Van Loan. "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-five Years Later." *SIAM Review*, 45:1 (2003), 3–49.

[33] L. Orecchia, S. Sachdeva, and N. K. Vishnoi. "Approximating the Exponential, the Lanczos Method and an $\tilde{O}(m)$-Time Spectral Algorithm for Balanced Separator." In *STOC '12*, pp. 1141–1160. New York, NY, USA: AMC, 2012.

[34] R. B. Sidje. "ExpoKit: A Software Package for Computing Matrix Exponentials." *ACM Trans. Math. Softw.* 24 (1998), 130–156.

[35] Center for Applied Internet DataAnalysis (CAIDA). "Network Datasets." *Center for Applied Internet Dara Analysis. Available online* (`http://www.caida.org/tools/measurement/skitter/router_topology/`), 2005.

[36] J. Yang and J. Leskovec. "Defining and Evaluating Network Communities Based on Ground-Truth." In *The IEEE 12th International Conference on Data Mining (ICDM)*, 2012, pp. 745–754. IEEE, 2012.

[37] X. T. Yuan and T. Zhang. "Truncated Power Method for Sparse Eigenvalue Problems." *CoRR*, abs/1112.2679, Available online (http://arxiv.org/abs/1112.2679), 2011.