Efficient and Practical Tree Preconditioning for Solving Laplacian Systems

Luca Castelli Aleardi(✉), Alexandre Nolin, and Maks Ovsjanikov

LIX - École Polytechnique, Palaiseau, France
{amturing,maks}@lix.polytechnique.fr,
alexandre.nolin@polytechnique.edu

Abstract. We consider the problem of designing efficient iterative methods for solving linear systems. In its full generality, this is one of the oldest problems in numerical analysis with a tremendous number of practical applications. We focus on a particular type of linear systems, associated with Laplacian matrices of undirected graphs, and study a class of iterative methods for which it is possible to speed up the convergence through combinatorial preconditioning. We consider a class of preconditioners, known as tree preconditioners, introduced by Vaidya, that have been shown to lead to asymptotic speed-up in certain cases. Rather than trying to improve the structure of the trees used in preconditioning, we propose a very simple modification to the basic tree preconditioner, which can significantly improve the performance of the iterative linear solvers in practice. We show that our modification leads to better conditioning for some special graphs, and provide extensive experimental evidence for the decrease in the complexity of the preconditioned conjugate gradient method for several graphs, including 3D meshes and complex networks.

1 Introduction

Solving general linear systems of equations is one of the oldest and best studied areas of numerical analysis, with an abundance of both exact and approximate solutions of varying efficiency (see e.g., [12]). In this paper, we focus on iterative methods for solving a particular type of linear systems, associated with Laplacian matrices of undirected graphs. These linear systems arise in a variety of applications, which are related to solving the Poisson equation on discretized domains, including physical (e.g. fluid) simulation, complex system analysis, geometry processing and computer graphics [16,17], among many others. One class of techniques, which is especially useful in solving large systems of equations with Laplacians matrices of certain (sparse) graphs is the conjugate gradient method. This method can be classified as an iterative approach, since it provides progressively better estimates of the final solution, and only requires the ability to compute matrix-vector products. It is also known to terminate in a finite number of steps depending on the quality of the initial guess and the condition number of the matrix in question [25]. The convergence speed of the conjugate gradient method can further be improved significantly using preconditioning, which aims to approximate the given matrix A by another matrix B (a preconditioner),

© Springer International Publishing Switzerland 2015
E. Bampis (Ed.): SEA 2015, LNCS 9125, pp. 219–231, 2015.
DOI: 10.1007/978-3-319-20086-6_17

whose inverse can be readily computed. The quality of the improvement provided by the preconditioner is directly related to the difference between $B^{-1}A$ and identity. Recently a class of preconditioners has been proposed for solving the Poisson equation on undirected graphs, by using the so-called combinatorial (or geometric) preconditioning [5,24]. The main idea, proposed by Vaidya, is to approximate a given graph by its subgraph, on which the system of equations can be solved easily. The canonical example of this type of preconditioner is a spanning tree of the graph. Since the Poisson equation can be solved in linear time if the graph is a tree, the main idea in Vaidya's approach is to use the Laplacian of the spanning tree as a preconditioner to improve the convergence of iterative methods, such as the conjugate gradient. This basic framework has been extended significantly to both obtain near-optimal trees that can approximate arbitrary graphs, and to use a recursive approach in which a graph can be approximated by a progressively more accurate subgraphs, which can lead to very significant asymptotic speed-up in solving linear systems on general graphs [23]. While the theoretical framework for combinatorial preconditioners has been developed and in some ways settled, with a few notable exceptions, the practical implementations of these ideas are still largely lacking. This can be attributed, in part, to the highly complex nature of the algorithms for obtaining the optimal preconditioners, with potentially very large constants in the asymptotic analysis on the one hand [22,23], and the relatively little improvement provided by the tree preconditioner on the other hand [9]. As a result, despite the theoretical appeal and the near-optimality in the asymptotic sense of the resulting algorithms [10], the practitioners have not yet fully benefited from the potential practical improvements provided by the combinatorial preconditioners.

Contribution. In this paper, we concentrate on the basic setting of Vaidya's preconditioners where the Laplacian matrix of a single spanning tree is used as a preconditioner for the conjugate gradient method. Indeed, by extending the experiments of Chen et al. [9] to a variety of graphs and large networks, we show empirically that in most cases the improvement given by a single preconditioner is either minor or even non-existent compared to the baseline conjugate gradient approach. In this context, our main contribution is to propose a very simple modification to Vaidya's tree preconditioner, which provides significant practical improvements, with minimal implementation effort. Our modification can be seen as a combination of a basic Jacobi (diagonal) preconditioner with a combinatorial (tree) one. Despite its extreme simplicity, we show that on a set of important special cases, our approach can lead to a decrease in the condition number of the resulting system, compared to the baseline combinatorial preconditioner. Perhaps more importantly, however, we also show via extensive experimentation, that our modification can also lead to practical speedup in the convergence of the conjugate gradient method compared to both the Jacobi and tree preconditioners for a large number of classes of graphs and different target functions. Our approach is not meant to provide a preconditioner structurally very different from the existing ones, or to improve their asymptotic complexity. Rather, by showing that a simple modification can potentially lead to significant

practical improvements, we hope to demonstrate the usefulness of such preconditioners and to help eventually bridge theory and practice in this field.

Related works. A tremendous amount of progress has been done in solving linear systems associated to symmetric diagonally dominant matrices in the recent past. Classical iterations, as described in [12], were very sensitive to systems of poor condition number, and until quite recently efficient preconditioning was mostly a matter of heuristics. A major step was done by Spielman and Teng [23], presenting the first nearly linear algorithm (see [20–22]). The feat was made possible by the introduction and refinement of ideas such as spectral sparsification, ultra-sparsifiers and algorithms for constructing low-stretch spanning trees, which they cleverly combined to build a recursively preconditioned iterative solver. The idea of recursive preconditioning is the basis of today's best solvers [10], and its individual parts have been separately improved over the years. For instance, methods for obtaining low-stretch spanning trees, first introduced with no link to preconditioning [2], have been seen a lot of progress over the years. Their use as preconditioners was suggested in [24], in the continuity of the ideas of support theory and combinatorial preconditioning (see [5–7] for early work and formalizations of this theory). Interested readers can read the progression of the stretch in [1], where the currently best algorithm for computing trees of total stretch $O(m \log n \log\log n)$ in time $O(m \log n \log\log n)$ is give. If no better bound has been found since then, recent works introduced a generalization of stretch [11], creating new possibilities of optimization. Spectral sparsification has seen similar improvements, however its progression is less linear than that of spanning trees. Better sparsifiers are described in [3] and fast construction algorithms are given in recent works [14] (see [4] for more details).

2 Preliminaries and Background

Throughout the paper, we consider simple, undirected, unweighted graphs $G = (V, E)$ with $\#V = n$ and $\#E = m$, and $d(i) = \#\{j,\ (i,j) \in E\}$ the degrees of the vertices. The unweighted (and un-normalized) Laplacian matrix L_G is given via its relation to the diagonal degree matrix D_G and the adjacency matrix A_G:

$$D_G = \begin{cases} d(i) & \text{if i = j} \\ 0 & \text{o/w} \end{cases}, \quad A_G = \begin{cases} 1 & \text{if (i, j)} \in \text{E} \\ 0 & \text{o/w} \end{cases}, \quad L_G = D_G - A_G$$

The Laplacian matrix L_G is symmetric, *diagonally dominant*, and only has non-negative eigenvalues. Indeed, it is easy to see that the number of connected components of G equals the dimension of the null space of L_G. Throughout our paper we assume to be working with a connected and unweighted graph G (most of the material can be adapted to the case of positively-weighted edges). The eigenvalues of L_G are given as $0 = \lambda_1 < \lambda_2 \leq \ldots \leq \lambda_n$.

Solving Linear Systems. The canonical problem that we consider is to solve a linear system of equations of the form $Ax = b$, where, in our case $A = L_G$ for some known vector b. Depending on the domain, a problem of this form may

also be known as solving the discrete Poisson equation. In general, although the number n of vertices in the graph can be very large, the matrix L_G is typically sparse, which can make direct solvers inefficient or even not applicable, since a full n^2 set of variables can easily exceed the available memory. Instead, iterative solvers have been used to solve this problem, and most notably the *Conjugate Gradient* (CG) method, which is especially useful in cases with limited memory, since it requires only matrix-vector product computations. This method is applicable to symmetric positive (semi)-definite systems, and computes successive approximations of x by taking a step in a direction *conjugate* to previously taken steps, where conjugacy between two vectors x_1, x_2 is defined as $x_1^T A x_2 = 0$ (please see Chap. 11 in [12] for a full discussion of this method, and Figure 1 below for the pseudo-code). It is well-known that in the absence of rounding errors, the conjugate gradient method will converge in at most n iterations in the worst case. A more relevant bound, however, can be given by using the condition number $\kappa(A)$ of the matrix A, given by the ratio of its largest and smallest *non-zero* eigenvalues. After t iterations, the error of the algorithm is bounded by:

$$||x^{(t)} - x||_A \leq 2 \left(1 - \frac{2}{\sqrt{\lambda_n/\lambda_2} + 1}\right)^t ||x||_A \tag{1}$$

Note that while the Conjugate Gradient method is best suited for positive definite matrices, it can also be easily adapted to positive semi-definite systems, such as the ones including the graph Laplacian. One simply has to make sure that the right hand side of the equation lies in the span of the matrix. For us, this means that the vector b has to sum to zero. Let us also stress that λ_1 in the definition of the condition number is the first *non-zero* eigenvalue. This will become particularly important when we define and analyze the properties of the preconditioned conjugate gradient.

Preconditioning. Since the *condition number* gives a simple bound on the efficiency of iterative solvers, and of the conjugate gradient method in particular, it is natural to try to introduce linear systems equivalent to the original one, but with a lower condition number, and therefore better convergence properties. This process, called *preconditioning*, requires a non-singular matrix M, such that $M^{-1} \approx A^{-1}$. Then, instead of solving $Ax = b$ directly, we solve :

$$C^{-1} A C^{-1} \tilde{x} = C^{-1} b \tag{2}$$

where $C^2 = M$, and x is found by solving $Cx = \tilde{x}$. Ideally, the preconditioner M should be a positive (semi)-definite matrix, such that the condition number of $M^{-1}A$ is significantly smaller than that of A itself. The design of optimal preconditioners typically involves a trade-off: on the one hand, $M^{-1}A$ should be as close to identity as possible. On the other hand, it should be possible to solve a linear system of the form $Mx = b$ very quickly, since it has to be done at every CG iteration. An example of potentially useful preconditioning is the *Jacobi Preconditioner* for diagonally dominant systems. This consists in

$$\text{Initialization}\begin{cases} r^{(0)} = b - Ax^{(0)} \\ p^{(0)} = r^{(0)} \end{cases} \qquad \text{Initialization}\begin{cases} r^{(0)} = b - Ax^{(0)} \\ z^{(0)} = M^{-1}r^{(0)} \\ p^{(0)} = z^{(0)} \end{cases}$$

$$\text{Iteration}\begin{cases} \alpha^{(k)} = \frac{||r^{(k)}||^2}{||p^{(k)}||_A^2} \\ x^{(k+1)} = x^{(k)} + \alpha^{(k)}p^{(k)} \\ r^{(k+1)} = r^{(k)} - \alpha^{(k)}Ap^{(k)} \\ \beta^{(k)} = \frac{||r^{(k+1)}||^2}{||r^{(k)}||^2} \\ p^{(k+1)} = r^{(k+1)} + \beta^{(k)}p^{(k)} \end{cases} \qquad \text{Iteration}\begin{cases} \alpha^{(k)} = \frac{\langle r^{(k)}, z^{(k)}\rangle}{||p^{(k)}||_A^2} \\ x^{(k+1)} = x^{(k)} + \alpha^{(k)}p^{(k)} \\ r^{(k+1)} = r^{(k)} - \alpha^{(k)}Ap^{(k)} \\ z^{(k+1)} = M^{-1}r^{(k+1)} \\ \beta^{(k)} = \frac{\langle z^{(k+1)}, r^{(k+1)}\rangle}{\langle z^{(k)}, r^{(k)}\rangle} \\ p^{(k+1)} = z^{(k+1)} + \beta^{(k)}p^{(k)} \end{cases}$$

Fig. 1. Conjugate Gradient and Preconditioned Conjugate Gradient : this pseudocode shows the general idea of orthogonalization in both algorithms as well as how the preconditioning takes place in PCG

taking the matrix $D = (\delta_{ij}A_{(i,j)})_{(i,j)}$, i.e the diagonal of the original matrix, as preconditioner. This is both very easy to compute, and solving $Dx = b$ takes an optimal $O(n)$ operations. When both A and M are symmetric positive definite, then solving Eq. 2 can be done without explicitly computing the matrix C, by modifying the steps taken during the iterations of the Conjugate Gradient method. This results in the *Preconditioned Conjugate Gradient* for which we provide the pseudo-code in Figure 1. Note that for positive semi-definite systems, one has to use the pseudo-inverse in Eq. 2 above, and make sure that the kernel of M is contained in the kernel of A, for otherwise the system $Mx = b$, may not have a solution. In most cases, when the preconditioning is applied to positive semi-definite systems the kernels of M and A coincide, although the framework can also be applied in a more general case.

Spanning Trees as Preconditioners for Graphs. While both the basic and the preconditioned Conjugate Gradient method can be applied for any positive (semi)-definite linear system, the design of preconditioners can benefit from the knowledge of the structure of the matrix in question. As mentioned above, in this paper, we concentrate on the linear systems arising from the Laplacian matrices of undirected graphs. In this case, a particularly promising idea, first proposed by Vaidya and then extended significantly in the recent years, is to use the Laplacian matrix of a subgraph as a preconditioner to the original system. Note that, if the subgraph is connected over the same set of nodes as the original graph, then the kernels of the Laplacian matrices both have dimension 1, and they contain the constant vector $\mathbf{1}_n$ and all the vectors parallel to it, making the use of preconditioning directly applicable. An appealing candidate for a subgraph to be used as a preconditioner is a spanning tree T of the graph G. This is because if L_T is the Laplacian matrix of the tree T, then the problem of type $L_T x = b$ can be solved very efficiently, in time $O(n)$, with two tree traversals. This makes spanning trees good candidates for preconditioning, because their use keeps the cost per PCG iteration in $O(m)$. It can be shown [25] that for a

spanning tree T of G, $\kappa(L_T^{\dagger} L_G) \leq stretch_T(G)$, where the stretch is defined as the sum of the distances in the tree between any two vertices connected by an edge in G. Together with Eq. 1 this can be used to establish the convergence of the preconditioned Conjugate Gradient for Laplacian matrices.

We note briefly that better bounds can be proved by also looking at the distribution of the eigenvalues. A proof using a lower bound for all eigenvalues and an upper bound on the number of eigenvalues above a certain threshold yields that PCG computes an $\epsilon-$approximation in $O\left((stretch_T(G))^{1/3} log(1/\epsilon)\right)$ iterations (see Lemma 17.2 in [25]). In the past several years, this basic framework for solving linear systems with Laplacian matrices has been extended significantly, with two major research directions: finding trees that can optimize the stretch with respect to arbitrary large graphs [1], and changing this basic framework to use a more sophisticated hierarchical graph approximation scheme in which preconditioners themselves can be solved via iterative (and possibly recursive) schemes [16]. Unfortunately, both of these directions lead to highly complex algorithms and their practical performance has been evaluated only very recently [13]. Rather than trying to improve either of these two directions, our goal is to show that a simple modification to the tree preconditioner can significantly improve the performance of the iterative solver both in theory (for some restricted cases) and in practice (over a large number of graph classes).

3 Contribution: Enhancing Tree-Based Preconditioners

As mentioned in the introduction, our main goal is to show that a simple modification of the combinatorial (tree) preconditioner can have a positive impact on the practical performance of the preconditioned conjugate gradient. Indeed, as has been noted by Chen et al. [9] and we confirm in Section 4, the basic version of Vaidya's approach rarely results in significant practical benefits for PCG. The critical remark behind our work is that it is possible to add positive terms to the diagonal of the preconditioning matrix L_T without changing its combinatorial structure that enables the fast resolution of associated linear systems. Thus, we introduce the matrix $H_T = L_T + D_G - D_T = D_G - A_T$. Note that the matrix H_T has the same diagonal as the Laplacian L_G, but the same sparsity structure as the Laplacian of the subgraph T. Therefore, solving a linear system of equations of the type $H_T x = b$ can still be done in exactly the same time as solving $L_T x = b$. Nevertheless, as we show below theoretically (on some restricted cases) and empirically on a large number of different graphs and linear systems, this simple modification can significantly boost the performance of the PCG method. Before proceeding to the analysis of our modification to Vaidya's preconditioner, we first note that unless $T = G$, the matrix H_T will be full-rank, unlike the L_G which has a kernel consisting of vectors parallel to the constant vector $\mathbf{1}_n$. While in practice, this does not change the method shown in Figure 1, we note that the analysis needs to be adapted slightly. Namely, since we are operating in the space orthogonal to the constant vector $\mathbf{1}_n$, we need to

make sure that the condition number of the preconditioned system is calculated correctly. For this, the following Lemma, which is readily verified, is useful:

Lemma 1. *The eigenvalues of the generalized eigenvalue system $L_G x = \lambda H_T x$ are the same as those of the system $L_G x = \lambda P H_T x$, where $P = (I_n - \frac{1}{n}\mathbf{1}_n \mathbf{1}_n^T)$ is the projection onto the space of vectors orthogonal to the constant vector.*

Therefore, computing the condition number $\kappa(L_G, H_T)$ of the preconditioned system can be done by considering the ratio of the largest to smallest non-zero eigenvalues of the matrix $H_T^{-1} L_G$. Equivalently, one can consider the smallest and largest value c such that $x^T(L_G - cH_T)x \geq 0$ for all x, such that $x^T H_T x_c = 0$.

To motivate the use of our preconditioner as well as to provide some intuition on its behavior we proceed in two stages. First, we show some bounds on the condition number for special graphs, and second, we demonstrate empirically that for a very wide range of large scale graphs and linear systems our approach can significantly outperform other baseline preconditioners (Section 4).

3.1 Some Bounds for Special Graphs

Here we provide bounds on the condition number of the preconditioned system for Laplacians and show that we can obtain significant improvement over Vaidya's preconditioners in some important special cases (proofs are given in [8]).

The Complete Graph. Let us first consider $G = K_n$, the complete graph on n vertices and let T be a *star* spanning tree, consisting of one root vertex of degree $n-1$ which is adjacent to all remaining $n-1$ vertices.

Lemma 2. *Given the complete graph G and the tree T described above, then for any $n > 2$ we have $\kappa(L_G, H_T) = \frac{n}{n-1} < \kappa(L_G, L_T) = n$.*

Note, in particular that $\kappa(L_G, H_T) \to 1$ whereas $\kappa(L_G, L_T)$ grows with n.

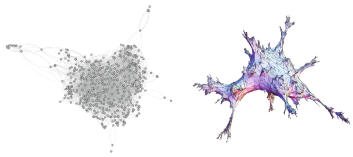
The Ring Graph. Another important example is the cycle (ring) graph with n vertices. Here, the tree T differs from G by a single edge. In this case:

Lemma 3. *If G is a cycle and T is a spanning tree of G, then $\kappa(L_G, H_T) < 2$, while $\kappa(L_G, L_T) = n$ for any n.*

Note that again, the system preconditioned with H_T remains well-conditioned for all n, unlike the system preconditioned by the tree itself, which has an unbounded condition number. Indeed, a strictly more general result holds:

Lemma 4. *Let G be any graph and T be a tree on G, such that the edge-complement T^c of T in G is a star. Then: $\kappa(L_G, H_T) \leq 2$.*

Note that this lemma generalizes the previous one since the complement of the tree in the ring graph is a single edge.



Network/Graph	n	m	$\kappa(L_G)$	$\kappa(L_G, L_T)$ (max tree)	$\kappa(L_G, H_T)$ (max tree)	$\kappa(L_G, L_T)$ (min tree)	$\kappa(L_G, H_T)$ (min tree)
C. Elegans	453	2025	922.53	373.03	20.42	11857	19.67
Email URV	1133	5451	217.44	16476	9.71	24540	10.23
Power grid network	4941	6594	26487	452.89	2366	6445	2331
Random triang.	102	300	115.42	102.65	42.95	286.51	41.59
Random triang.	1002	3000	2009	1206	461	2359	475.11

Fig. 2. Condition numbers for unweighted graphs: we consider a few example of complex networks and random triangulations. Left pictures show the metabolic system of the *C. elegans* worm and a random planar triangulation (picture by Nicolas Curien).

The Wheel Graph. Our final example is the wheel graph, consisting of a cycle with $n-1$ vertices that are all connected to a central vertex s, which does not belong to the cycle. In this case, let T be the star graph centered around s.

Lemma 5. *Given the graph G and the spanning tree T described above then, for any n odd, $\kappa(L_G, H_T) < \kappa(L_G, L_T) = 5$.*

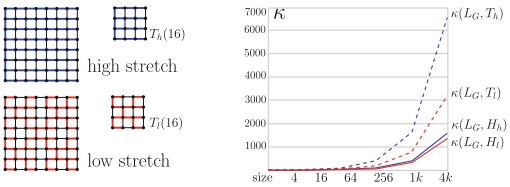
This example is instructive since the wheel graph can be considered to be a simple case of a triangle mesh, a class of graphs for which we show empirically a significant improvement over Vaidya's preconditioners in the next section.

A Counterexample. We also note that there exist graphs for which the condition number of our system is worse than that of the unmodified approach. The simplest example of such a graph is a path-graph with 6 nodes, with additional edges between nodes $(1,3)$ and $(4,6)$, and where the tree is the path. In this case, it can be shown that $\kappa(L_G, L_T) = 3 < \kappa(L_G, H_T)$. Nevertheless our experiments suggest that such cases are rare, and seem to occur when G is very close to the tree T. We leave the characterization of such cases as interesting future work.

4 Experimental Results

We provide experimental evaluations[1] of the performance of our preconditioner against *CG* (conjugate gradient with no preconditioning), the diagonal preconditioner *JPCG* (Jacobi preconditioned conjugate gradient) and *TPCG* (tree-based Vaidya's preconditioned conjugate gradient). As our preconditioner is a combination of the tree-based and diagonal approaches, we denote it by *JTPCG*.We run our experiments on a wide collection of graphs including triangle meshes (obtained from the AIM@SHAPE Shape repository), 2D regular grids, and complex networks (from the Stanford Large Network Dataset Collection). We also consider random planar triangulations, generated by the uniform random sampler by Poulalhon and Schaeffer [19], as well as graphs randomly generated according to the small-world and preferential attachment models.

[1] A pure Java implementation of our algorithms is available at www.lix.polytechnique.fr/~amturing/software.html.



grid size	$\kappa(L_G)$	$\kappa(L_G, T)$ (low stretch)	$\kappa(\mathbf{L_G}, \mathbf{H})$ (low stretch)	$\kappa(L_G, T)$ (high stretch)	$\kappa(\mathbf{L_G}, \mathbf{H})$ (high stretch)
4	2	4	1.6	4	1.6
16	11.6	9.88	4.52	21.02	4.84
64	50.54	42.64	17.37	95.14	19.63
256	206.17	187.02	77.07	399.89	87.82
1024	828.69	788.56	332.88	1631.64	379.68
4096	3318	3242	1390	6585	1585

Fig. 3. We compute condition numbers for regular (unweighted) grids endowed with different spanning trees: blue and red edges correspond to trees with high and low stretch factors respectively. Our precondition matrix H allows to drastically decrease the condition number in both cases, when compared to standard tree preconditioning (dashed lines).

3D Mesh	n	m	$\kappa(L_G)$	$\kappa(L_G, L_T)$ (max tree)	$\kappa(\mathbf{L_G}, \mathbf{H_T})$ (max tree)	$\kappa(L_G, L_T)$ (min tree)	$\kappa(\mathbf{L_G}, \mathbf{H_T})$ (min tree)
Sphere	162	480	33.4	723.	25.6	1384	26.06
Helmet	496	1482	245.8	2885	142.4	5341	143.8
Venus	711	2106	411.8	2591	229.6	3950	251.46
Genus 3 mesh	1660	4992	304.9	5862	226.5	13578	227.2
Triceratops	2832	8490	2079	12342	1454	13332	1530
Cow	2904	8706	2964	15184	1853	8868	1982

Fig. 4. Condition numbers for 3D surface meshes: we compare tree preconditioning and Jacobi-tree preconditioning. Meshes are endowed with both minimum (blue) and maximum (red) spanning trees (weights correspond to Euclidean edge length).

4.1 Evaluating the Condition Number

Regular grids. Our first experiments concern the evaluation of the condition numbers for regular grids, for which we know how to construct spanning trees of high and low stretch factors. It is not difficult to see that the total stretch of the blue tree T_h in Fig. 3 is $\Theta(n\sqrt{n})$ (observe that a vertical edge $(u_i, v_i) \in G \setminus T$ belonging to the i-th column contributes with a stretch of $\Theta(i)$, where i ranges from 1 to $\sqrt{n}$). The red edges in Fig. 3 define a spanning tree T_l having a low stretch factor, which can be evaluated to be $O(n \log n)$ using an inductive argument (we refer to [16] for more details). These bounds reflect the numerical evaluation of the condition numbers for both trees T_h and T_l (plotted as dashed lines in Fig. 3). Experimental evaluations show that our Jacobi-tree preconditioner allows to drastically decrease the condition numbers for both trees T_h and T_l. More interestingly, using H_T instead of L_T we obtain new bounds which are extremely close, despite the very different performances of the corresponding spanning trees. Not surprisingly, this behavior does not only concern regular grids, but it is common to a wide class of graph Laplacians, as suggested by experimental evidence provided in next sections.

Mesh graphs and complex networks. We also compute the condition numbers for Laplacians corresponding to several 3D meshes, of different sizes and topology (see Fig. 4). We test and compare our preconditioner against the CG method (without preconditioning) and tree preconditioning, using as test trees both minimum and maximum spanning trees. We consider min spanning trees because

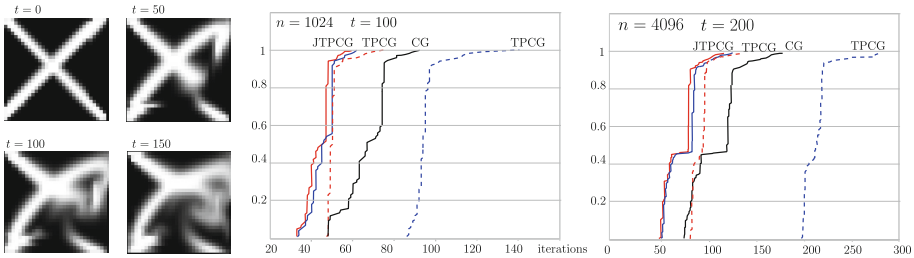


Fig. 5. Fluid simulation: we compare JTPCG against CG and TPCG (the performance of JPCG is very similar to CG). We plot the proportion of the number of iterations required to solve 100 (resp. 200) linear systems with a precision of $1e-5$. For instance, JTPCG (colored curves) takes between 51 and 127 iterations per system on a grid of size 4096, while *CG* (black curve) requires between 75 and 184 iterations.

their performances are in general worse than those of maximum spanning trees: weights are computed according to the Euclidean edge length of the 3D embedding (in the case of unweighted graphs, in order to compute min and max spanning tree, we reweight edges according to a vertex degree driven strategy). We note that our experiments confirm the intuition of Vaidya's seminal work: maximum spanning trees perform in general better as preconditioners than other trees. Once again, our preconditioner is able to get condition numbers which are significantly lower than the ones obtained with the simple preconditioner L_T. We note that this difference in performance is much less prominent when using H_T. These phenomena occur for all tested graphs and result in a significant improvement of the performance of iterative solvers.

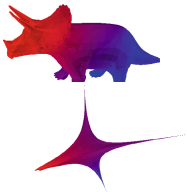
4.2 Counting Iterations: Comparison of Iterative Linear Solvers

In this section we provide experimental evidence for the improvement achieved by our JTPCG preconditioner. We test it against other linear solvers (CG, JPCG, and TPCG) on a large set of surface meshes and random graphs. In order to obtain a fair comparison, we measure the convergence rates of linear solvers for $Lx = b$ counting the total number of iterations required to achieve a given error: as metrics we use the standard relative residual error. We use iterative solvers as core linear solvers for simulating fluid diffusion on regular 2D grids of different sizes, while counting the number of iterations required by different solvers at each time step (we use fixed precision $1e-5$). As shown by the plots in Fig. 5, JTPCG is able to drastically decrease the number of iterations, using both the high and low stretch factor spanning trees (red and blue curves). Observe that tree-based preconditioner perform pretty well (even without diagonal modification) when combined with low stretch factors (red curves in Fig. 5).

Solving mesh laplacians. The results reported in Table 1 concern the resolution of linear systems of the form $A\mathbf{x} = b$, where the vector b is a random vector orthogonal to the constant vector. We use the same starting vector as an initial

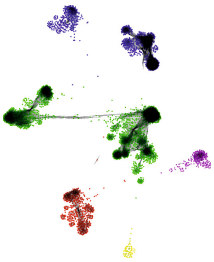
Table 1. Solving linear systems: we compare the JTPCG against the classical CG method, the JPCG and TPCG preconditioners. We count total number of iterations required to achieve fixed precision $1e-7$.

Graph	n	m	CG (no prec.)	JPCG	TPCG (max tree)	**JTPCG** (max tree)	TPCG (min tree)	**JTPCG** (min tree)
Triceratops	2832	$8K$	225	196	341	188	426	181
Cow	2904	$8K$	214	192	347	170	366	182
Egea	8268	$24K$	305	249	701	219	974	221
Bunny	26002	$78K$	536	432	1632	416	1892	419
Feline	49864	$149K$	962	745	1946	663	2362	682
Eros	476596	$1.4M$	2185	1560	16122	1474	13257	1488
Random triang.	100002	$300K$	2382	1215	1776	1082	1247	1006



Graph	n	m	CG (no prec.)	JPCG	TPCG (max tree)	**JTPCG** (max tree)	TPCG (min tree)	**JTPCG** (min tree)
Triceratops	2832	$8K$	5139	5057	6811	4842	7505	4997
Cow	2904	$8K$	5158	5145	6854	4907	6989	4980
Egea	8268	$24K$	7980	7314	12525	6988	15206	7031
Bunny	26002	$78K$	32187	30634	49048	30231	51405	30312
Aphrodite	46096	$138K$	13669	12228	37547	11803	41991	11303
Feline	49864	$149K$	46404	42217	62595	40371	71095	40727
Iphigenia	49922	$149K$	19490	18111	54008	16984	60973	17306

Fig. 6. The picture above shows a mesh together with its 3D spectral embedding. The table reports the total number of iterations performed by the iterative linear solvers during the inverse power iteration (our tests are run with fixed precision $1e-5$).



Network	n	m	CG	JPCG	TPCG (max tree)	**JTPCG** (max tree)
C. Elegans	453	2025	8123	7129	7795	7051
Email URV	1133	5451	24395	23540	25684	23435
Facebook social circles	4039	88234	11832	7702	8044	7677
Power grid network	4941	6594	15623	13430	8812	10481
PGP network	10680	24316	64068	54806	55356	53852
Pref. attachment	100000	$500K$	61125	59399	80451	59455
Small world	100000	$500K$	4972	5010	125446	4963
Gowalla	196591	$950K$	202247	146883	176644	147322

Fig. 7. Spectral clustering and complex networks: the picture above shows a partition into five sets of a social network (facebook, 4k nodes) obtained by applying the K-means algorithm to the spectral embedding of the graph.

guess for all linear solvers (tests are repeated several times, in order to take into account the dependency of the convergence speed on the initial guess). As confirmed by the results reported in Table 1, our preconditioner always performs better than other solvers (this has been confirmed for all tested meshes).

Iterative eigensolvers. Spectral methods proved their relevance in various application domains, ranging from graph drawing and data visualization to complex networks analysis (for more details we refer to [15,25]). We also have integrated our preconditioner as the core of an iterative eigensolver (we have implemented

a hybrid version of the inverse power iteration). We evaluate its performance by computing the smallest non-trivial eigenvalues of the Laplacian: a fundamental step in problems such as spectral drawing and spectral clustering. Tables in Fig. 6 and 7 report the number of iterations performed by the linear solvers required to compute the first three eigenvalues for 3D meshes and complex networks.

Acknowledgments. This work is supported by the ANR EGOS 12 JS02 002 01, a Google Faculty Research Award, the Marie Curie grant CIG-334283-HRGP, and a CNRS chaire dexcellence, Jean Marjoulet professorial chair.

References

1. Abraham, I., Neiman, O.: Using petal-decompositions to build a low stretch spanning tree. In: Proc. STOC, pp. 395–406. ACM (2012)
2. Alon, N., Karp, R.M., Peleg, D., West, D.B.: A graph-theoretic game and its application to the k-server problem. SIAM J. Comput. **24**(1), 78–100 (1995)
3. Batson, J.D., Spielman, D.A., Srivastava, N.: Twice-ramanujan sparsifiers. SIAM Review **56**(2), 315–334 (2014)
4. Batson, J.D., Spielman, D.A., Srivastava, N., Teng, S.: Spectral sparsification of graphs: theory and algorithms. Commun. ACM **56**(8), 87–94 (2013)
5. Beauwens, R.: Lower eigenvalue bounds for pencils of matrices. Linear Algebra and its Applications **85**, 101–119 (1987)
6. Bern, M.W., Gilbert, J.R., Hendrickson, B., Nguyen, N., Toledo, S.: Support-graph preconditioners. SIAM J. Matrix Analysis Applications **27**(4), 930–951 (2006)
7. Boman, E., Hendrickson, B.: Support theory for preconditioning. SIAM Journal on Matrix Analysis and Applications **25**(3), 694–717 (2003)
8. Castelli Aleardi, L., Nolin, A., Ovsjanikov, M.: Efficient and practical tree preconditioning for solving Laplacian systems (2015). Preprint https://hal.inria.fr/hal-01138603
9. Chen, D., Toledo, S.: Vaidya's preconditioners: implementation and experimental study. Elect. Trans. on Numerical Analysis **16**, 30–49 (2003)
10. Cohen, M.B., Kyng, R., Miller, G.L., Pachocki, J., Peng, R., Rao, A., Xu, S.C.: Solving SDD linear systems in nearly $m \log^{1/2} n$ time. STOC, pp. 343–352 (2014)
11. Cohen, M.B., Miller, G.L., Pachocki, J.W., Peng, R., Xu, S.C.: Stretching stretch (2014). CoRR, abs/1401.2454
12. Golub, G.H., Van Loan, C.F.: Matrix Computations. 4th edn (2013)
13. Hoske, D., Lukarski, D., Meyerhenke, H., Wegner, M.: Is nearly-linear the same in theory and practice? a case study with a combinatorial laplacian solver. In: Proc, SEA (2015)
14. Kolla, A., Makarychev, Y., Saberi, A., Teng, S.-H.: Subgraph sparsification and nearly optimal ultrasparsifiers. In: Proc. STOC, pp. 57–66. ACM (2010)
15. Koren, Y.: Drawing graphs by eigenvectors: Theory and practice. Computers and Mathematics with Applications **49**, 2005 (2005)
16. Koutis, I., Miller, G.L., Peng, R.: A fast solver for a class of linear systems. Commun. ACM **55**(10), 99–107 (2012)
17. Krishnan, D., Fattal, R., Szeliski, R.: Efficient preconditioning of Laplacian matrices for Computer Graphics. ACM Trans. Graph. **32**(4), 142 (2013)
18. Poulalhon, D., Schaeffer, G.: Optimal coding and sampling of triangulations. Algorithmica **46**(3–4), 505–527 (2006)

19. Spielman, D.A., Teng, S.: Spectral sparsification of graphs. SIAM J. Comput. **40**(4), 981–1025 (2011)
20. Spielman, D.A., Teng, S.: A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning. SIAM J. Comput. **42**(1), 1–26 (2013)
21. Spielman, D.A., Teng, S.: Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. SIAM J. Matrix Analysis Applications **35**(3), 835–885 (2014)
22. Spielman, D.A., Teng, S.-H.: Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. STOC, pp. 81–90 (2004)
23. Vaidya, P.M.: Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. Unpublished manuscript (1991)
24. Vishnoi, N.K.: Lx = b. Foundations and Trends in Theoretical Computer Science **8**(1–2), 1–141 (2013)
25. von Luxburg, U.: A tutorial on spectral clustering. Statistics and Computing **17**(4), 395–416 (2007)