Lecture 7

Lecturer: Aleksander Mądry *Scribe: Mathieu Dahan*

1 Overview

In this lecture, we continue our study of fast linear system solving, with a special focus on solving Laplacian systems. In particular, we introduce the notion of preconditioners and discuss how it can be used to speed up iterative linear solving approaches such as the conjugate gradient method. Finally, we will also construct simple but non-trivial preconditioners for Laplacian matrices.

2 Recap of the Last Lecture

Last lecture, we continued our interest in solving the following linear system

$$Ax = b, \tag{1}$$

where A is a PSD matrix, that is, it is symmetric ($A^T = A$) and positive-definite ($A \succ 0$). This implies, in particular, that the eigenvalues $\lambda_1 \leq \ldots \leq \lambda_n$ of the matrix A are all real and positive. So, in particular, the matrix A is invertible.

Our important achievements in this regard was developing and analyzing the performance of the *conjugate gradient method*. This method can compute a solution x_T to the linear system (1) with $\|x_T - x^*\|_A^2 \leq \varepsilon$ in time

$$O\left(\tau(A)\sqrt{\kappa(A)}\log\frac{\|x^*\|_A^2}{\varepsilon}\right), \tag{2}$$

where x^* is the exact solution to that system. Here, $\tau(A)$ is the time needed to multiply A by a vector y and $\kappa(A) := \frac{\lambda_n}{\lambda_1}$ is the condition number of A.

2.1 Spectrum of a Laplacian matrix

Our main motivation for development of the fast linear system solving machinery was to apply it to solving Laplacian systems, which have the form

$$L_G\varphi = \sigma, \tag{3}$$

where L_G is the Laplacian matrix of an underlying graph $G = (V, E)$. (In this lecture, we focus on the case of G being unweighted, but everything we say can be extended to the weighted case in a straightforward manner.) There is a number of ways to define a Laplacian, but the one that will be the most convenient today is to view it as a sum of elementary Laplacians L^e, for $e \in E$, where each L^e is a Laplacian of a graph with the vertex set V and containing only a single edge e. In other words, we have

$$L_G = \sum_{e\in E} L^e = \sum_{e\in E} \chi_e\chi_e^T \tag{4}$$

where

$$\chi_e(i) := \begin{cases} -1 & \text{if } i = u, \\ 1 & \text{if } i = v, \\ 0 & \text{otherwise,} \end{cases}$$

for u (resp., v) being the tail (resp., head) of e according to its orientation.

So, to apply the conjugate gradient method to the Laplacian system (3), we needed to understand the spectrum of the Laplacian. To this end, we stated last time the following theorem.

Theorem 1 *Let L_G be a Laplacian of a graph G and $\lambda_1 \leq \ldots \leq \lambda_n$ be its eigenvalues.*

(a) $\lambda_1 = 0$ and an all-ones vector $\vec{1}$ is an eigenvector corresponding to this eigenvalue;

(b) $\lambda_2 > 0$ iff G is connected;

(c) $\lambda_n \leq 2d_{\max}$, where $d_{\max}$ is the maximum (weighted) vertex degree, i.e., the largest entry of the degree matrix D.

The above theorem highlights an important issue with the Laplacian matrix: the fact that L_G is *not* and invertible matrix. Fortunately, as we explained last time, this is not a major problem. When G is connected, which will be always the case in our considerations, L_G is an invertible matrix provided we work with vectors that are orthogonal to its one-dimensional kernel spanned by the all-ones vector $\vec{1}$. In particular, the demand vector σ in our Laplacian system (3) has to be orthogonal to the vector $\vec{1}$ since it has to be balanced, i.e., the total amount of flow excesses at all the vertices has to be equal to the total amount of flow deficits.

To make all of this mathematically precise, in case of a Laplacian L_G, we do not think about its inverse L_G^{-1}, as it does not exist, but instead about its *Moore-Penrose pseudo-inverse* L_G^+ defined as

$$L_G^+ := \sum_{i>1}^{n} {\lambda_i}^{-1} v_i v_i^T$$

where $v_i v_i^T$ is the orthogonal projector onto the eigenspace associated to the eigenvalue λ_i.

Consequently, one can adjust the analysis of the conjugate gradient method to obtain that when we work in the space orthogonal to the eigenspace corresponding to $\lambda_1 = 0$ – or, more generally, in the space orthogonal to the (possibly many-dimensional) eigenspace corresponding to the eigenvalues $\lambda_1 = \ldots = \lambda_k = 0$ – the analog of the bound (2) is a bound of

$$O\left(\tau(A)\sqrt{\kappa^+(A)}\log\frac{\|x^*\|_A^2}{\varepsilon}\right), \tag{5}$$

where $\kappa^+(A) := \frac{\lambda_n}{\lambda_2}$ (or, more generally, $\kappa^+(A) := \frac{\lambda_n}{\lambda_{k+1}}$) is the *pseudo-condition number of A*.

3 Upperbounding the Pseudo-condition Number of a Laplacian

In the light of bound (5), we want to understand now what is the value of the pseudo-condition number $\kappa^+(L_G) = \frac{\lambda_n}{\lambda_2}$ of a Laplacian of a graph G. By Theorem 1, we know that $\lambda_n \leq 2d_{\max}$ and this bound turns out to be fairly tight for most graphs.

We thus need to focus on lower bounding the value of λ_2. How small can λ_2 be for a given Laplacian matrix L_G of an unweighted and connected graph G?

Observe that Theorem 1 indicates that there is a qualitative connection between the value of λ_2 and the fact that G is connected. As it turns out, this qualitative connection can be strengthened to yield a quantitative connection as well. This connection is known as *Cheeger's inequality*. We will not state this inequality now but very roughly speaking it tells us that the value of λ_2 is proportional to how well connected the underlying graph G is.

In particular, if G is an expander graph, i.e., a constant-degree graph in which the size of every cut is within a constant of the size of the smaller side of that cut, λ_2 is $\Omega(1)$. As in this case, $\lambda_n \leq 2d_{\max} = O(1)$, we have that $\kappa^+(L_G)$ is $O(1)$ too. This implies that by (2) the conjugate gradient method solves a linear system in L_G in nearly-linear time!

This is a quite remarkable fact and a testament to the power of conjugate gradient method. After all, expanders tend to have very non-trivial structure – in fact, they essentially look like random graphs – and there seems to be no obvious direct way of solving linear systems in them. Yet, they are very easy to crack for conjugate gradient method.

Unfortunately, despite shining when dealing with expander graphs, the conjugate gradient method performs quite poorly when confronted with a very simple graph: a path graph on n vertices. In contrast

to the expander graph that can be seen as the "most connected" graph among all (unweighted) connected sparse graphs, the path graph is the "least connected" among such graphs. Specifically, one can show that $\lambda_2 = \Theta(\frac{1}{n^2})$ in this case.

As $\lambda_n \leq 2d_{\max} = O(1)$ here again, the pseudo-condition number $\kappa^+(L_G)$ becomes as large as $\Theta(n^2)$. By (5), this means that the conjugate gradient method requires $\Omega(\sqrt{\kappa^+(L_G)}) = \Omega(n)$ iterations to deliver any non-trivial solution to a linear system in L_G.

It is worth pointing out here that this example also shows that the square-root dependence on the (pseudo-)condition number that the conjugate gradient method achieves is essentially best possible in our iterative framework. Roughly speaking, it is not hard to convince oneself that one needs to work with at least $(n-1)$-th powers of the Laplacian matrix to be able to propagate the information from one endpoint of the n-vertex path graph to its other endpoint. Recall, however, that in our iterative framework, working with k-th power of the Laplacian matrix requires the iterative method to run for at least k iterations. So, in the case of the Laplacian of a path graph, the number of iterations has to be at least $n-1$ and we cannot have significantly better than square-root dependence of the number of iterations on the (pseudo-)condition number.

4 Solving Laplacian Systems for Trees

The very poor performance of the conjugate gradient method on a path Laplacian that we described above should be somewhat surprising. After all, it is not hard to solve a linear system in path Laplacian directly!

Specifically, if we enumerate the vertices $v_1, \ldots, v_n$ of a path graph from left to right, the Laplacian of a path graph becomes this simple tri-diagonal matrix

$$\begin{pmatrix} 1 & -1 & 0 & \ldots & 0 \\ -1 & 2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2 & -1 \\ 0 & \ldots & 0 & -1 & 1 \end{pmatrix},$$

and we can solve a Laplacian system (3) in such a matrix via simple pivoting.

More precisely, consider changing our variables by making a substitution

$$\varphi'_{v_2} \leftarrow \varphi_{v_2} + \varphi_{v_2} \quad \text{and} \quad \varphi'_{v_i} \leftarrow \varphi_{v_i}.$$

for all $i \neq 2$. This corresponds to adding the first column of the Laplacian to the second one. Observe that after this substitution our linear system (3) becomes

$$\begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ -1 & 1 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2 & -1 \\ 0 & \ldots & 0 & -1 & 1 \end{pmatrix} \varphi' = \sigma.$$

Then, if we change our demand vector as

$$\sigma'_{v_2} \leftarrow \sigma_{v_1} + \sigma_{v_2} \quad \text{and} \quad \sigma'_{v_i} \leftarrow \sigma_{v_i},$$

for all $i \neq 2$, this will correspond to adding the first row of our constraint matrix to its second row. As

a result, our linear system becomes

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -1 & \ddots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2 & -1 \\ 0 & \cdots & 0 & -1 & 1 \end{pmatrix} \varphi' = \sigma'$$

Now, since the first row and column of our constraint matrix has only a single non-zero entry, solving this system immediately reduces to solving the subsystem corresponding to the constraint submatrix one obtains by dropping this first row and column. However, this problem is again a Laplacian system in a path graph. Only the number of vertices of this path is now $n-1$ instead of n.

Clearly, using $O(1)$ operations we reduced the size of our problem by 1. Therefore, a Laplacian system for a path graph can be solved directly (and exactly!) in $O(n)$ time.

In fact, it is not hard to generalize this approach to the case when the underlying graph is a tree. (One just has to apply the analogous pivoting sequentially to each degree-one vertex – as the graph is a tree, there always will be at least one such vertex.) We can conclude with the following lemma.

Lemma 2 *If G is a tree, then we can solve a Laplacian system (3) in the Laplacian L_G of G in $O(n)$ time.*

5 Preconditioning

The above discussion highlighted a significant issue with the conjugate gradient method. On one hand, this method can handle some highly non-trivial graphs, such as expanders, extremely well. On the other hand, its performance completely deteriorates when confronted with relatively simple graphs, due to its inability to take direct advantage of these graph simple structure. Is there some way of remedying this deficiency by providing a way of incorporating such direct structural insights into the whole iterative solving framework?

It turns out that the answer to this question is affirmative[1] and it corresponds to a fundamental notion of *preconditioning*.

As the notion of preconditioning applies much more broadly than just to Laplacian matrix, let us forget for a moment about Laplacians and consider the general question: what to do if we want to apply an iterative method to solving a linear system in a matrix A whose condition number $\kappa(A)$ is very large?

To illustrate our discussion, let us consider the following example of a matrix A

$$C := \begin{pmatrix} 1000000 & 2500 & 0.1 \\ 2500 & 10000 & 0.2 \\ 0.1 & 0.2 & 1 \end{pmatrix}.$$

This matrix is PSD but its condition number is very large $\kappa(C) \approx 1000000$, making running conjugate gradient method on a linear system in C very slow (given that this is only a 3×3 matrix).

At first glance, it seems that in this situation this bad performance of conjugate gradient method is unavoidable. However, a moment of thought might hint at a way to go around this. Namely, instead of solving the original linear system

$$Cx = b,$$

how about solving a linear system

$$D^{-1}Cx = D^{-1}b,$$

[1] In a sense, it has to be such – if there was no good way of incorporating these structural insights, the conjugate descent method wouldn't be as widespread and practical algorithm as it is. After all, in practice, the matrices one is dealing with tend to have a lot of structure and one must have a way of taking advantage of that.

where D is obtained from C by simply dropping all the off-diagonal entries? That is,

$$D := \begin{pmatrix} 1000000 & 0 & 0 \\ 0 & 10000 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Somewhat surprisingly, even though the condition number $\kappa(C)$ of the matrix C was very large (and so is the condition number $\kappa(D^{-1})$ of the matrix D^{-1}), the condition number $\kappa(D^{-1}C)$ of the matrix $D^{-1}C$ is very small – only around 1.4. This means that if we apply the conjugate gradient method to that second (essentially, equivalent) linear system then its number of iterations will be very small and its running time will be dominated by the time $\tau(D^{-1}C)$ needed to multiply the matrix $D^{-1}C$ by a vector. As D is a diagonal matrix, the latter can be easily done in $O(n) + \tau(C) = O(\tau(C))$ time.

Thus, we see that per-multiplying both sides of our linear system by the D^{-1} matrix dramatically improved the performance of the conjugate gradient method on it. How to apply this idea more broadly then?

The key notion here is the notion of a preconditioner. Given a linear system (1) in some matrix A, a *preconditioner* of that matrix is a matrix P that has two properties':

(1) The condition number $\kappa(P^{-1}A)$ of the matrix $P^{-1}A$ is small;

(2) The time $\tau(P^{-1})$ needed to multiply the matrix P^{-1} by a vector should be small as well.

Intuitively, the first condition above tells us that the matrix $P^{-1}A$ is close to being an identity matrix I. Or, in other words, that $P^{-1} \approx A^{-1}$. So, P should be thought of as a reasonably good approximation of the matrix A. (Note that in our example above, the diagonal D of the matrix C seems to indeed be a good sketch of C.)

Clearly, once this condition holds, the conjugate gradient method applied to the linear system (1) after pre-multiplication of both sides by P^{-1} takes only a small number of iterations to converge.

On the other hand, the second condition ensures that each iteration of the conjugate gradient method applied to the pre-multiplied system is fast too. After all,

$$\tau(P^{-1}A) \leq \tau(P^{-1})\tau(A).$$

Note that we do not ever need here to perform the – potentially, computationally expensive – explicit multiplication of the matrices P^{-1} and A. Also, we do not really need to compute the inverse P^{-1} of the matrix P either. Multiplying a vector y by the matrix P^{-1} corresponds to simply solving a linear system

$$Pz = y,$$

where z will be the product $P^{-1}y$ that we want to compute.

This second condition is also the reason why simply taking $P = A$ –which would make the first condition satisfied in the best possible way ($\kappa(A^{-1}A) = 1$) – does not really help. This choice would require us to develop a fast way of solving a linear system in A anyway.

Finally, there are two more technical points that we should address. First one is the fact that the matrix $P^{-1}A$ might not be symmetric, even if both P and A are. Fortunately, this is not much of a problem. Even though $P^{-1}A$ might not be symmetric, it still has the key property we actually needed for the analysis of conjugate gradient method to go through: all its eigenvalues are real. To see that, observe that the matrix $P^{-\frac{1}{2}}AP^{-\frac{1}{2}}$ is symmetric and thus it has all its eigenvalues real. Furthermore, if some vector v is an eigenvector of the matrix $P^{-\frac{1}{2}}AP^{-\frac{1}{2}}$ corresponding to an eigenvalue λ then

$$P^{-1}A\left(P^{-\frac{1}{2}}v\right) = P^{-\frac{1}{2}}\left(P^{-\frac{1}{2}}AP^{-\frac{1}{2}}v\right) = \lambda\left(P^{-\frac{1}{2}}v\right).$$

That is, the vector $P^{-\frac{1}{2}}v$ is an eigenvector of the matrix $P^{-1}A$ corresponding to the same eigenvalue λ. In other words, the matrix $P^{-1}A$ not only has real eigenvalues but its eigenvalues are exactly the same as the eigenvalues of the matrix $P^{-\frac{1}{2}}AP^{-\frac{1}{2}}$.

The second more technical point relates to the error guarantee that the conjugate gradient method applied to the pre-multiplied system gives us. Note that direct adaptation of the canonical error guarantee of the conjugate gradient method – see (2) – would give us that a guarantee in terms of the $\|\cdot\|_{P^{-1}A}$ norm, instead of the original $\|\cdot\|_A$ norm. However, in the lecture notes from Lecture 6, we know that the conjugate gradient descent provides the same type of error guarantee for any matrix norm $\|\cdot\|_M$, so taking $M = A$, recovers the error bound in the "right" norm.

6 Constructing Preconditioners for Laplacian Matrices

We know now that finding an appropriate preconditioning matrix P for our linear system in a matrix A enables us to dramatically improve the performance of the conjugate gradient method. Recall that this matrix P, on one hand, should constitute a good approximation of A, i.e., $\kappa(P^{-1}A)$ should be small; and, on the other hand, should have enough structure to allow us to solve a linear system in it fast, i.e., to have $\tau(P^{-1})$ be small. The key question that was left unanswered so far is: how to find such a matrix P?

Unfortunately, there is no general answer to this question. Constructing preconditioners is more of a black art than science, with practitioners trying to come up with them on mostly ad-hoc basis, depending on the particular properties and structure of the matrices they are dealing with. In particular, there is plenty of heuristics in this domain, e.g., preconditioning with the diagonal of the input matrix, but no real rigorous and systematic approaches.

However, the above picture becomes very different once we focus our attention on Laplacian systems. It turns out that there *exists* a principled approach to constructing preconditioners for Laplacian matrices. The basic underlying principle is to precondition Laplacians with other Laplacians. That is, to make the preconditioner of a Laplacian L_G of a graph G be a Laplacian L_H of some *different*, "simpler" graph H.

This general design principle will be extremely fruitful and we will first consider its simplest incarnation: taking the graph H that defines our Laplacian preconditioner for L_G to be just some spanning tree T of the graph G.

Observe that by Lemma 2 we know that $\tau(L_T^+) = O(n)$. So, the second property we require of a preconditioner is already satisfied. How about the first one? That is, what is the pseudo-condition number $\kappa^+(L_T^+ L_G)$ of the matrix $L_T^+ L_G$?

To answer this question, let us first lower bound the value of $\lambda_2(L_T^+ L_G)$, i.e., the value of the second smallest eigenvalue of the matrix $L_T^+ L_G$. (It is not hard to show that $\lambda_2(L_T^+ L_G)$ is positive.) Note that, for any vector y, we have that

$$y^T L_T y = y^T \left(\sum_{e \in T} \chi_e \chi_e^T \right) y = \sum_{(u,v) \in T} (y_u - y_v)^2 \leq \sum_{(u,v) \in G} (y_u - y_v)^2 = y^T \left(\sum_{e \in G} \chi_e \chi_e^T \right) y = y^T L_G y,$$

where the inequality follows since T is a subgraph of G and we also used the definition (4) of the Laplacian matrix. In other words, we have that

$$L_G \succeq L_T,$$

which implies that

$$L_T^+ L_G \succeq I^+,$$

where I^+ is an identity matrix on the space orthogonal to $\ker(L_T^+ L_G) = \ker(L_T) = \ker(L_G) = \text{span}(\vec{1})$. Consequently, we have that

$$\lambda_2(L_T^+ L_G) \geq 1. \tag{6}$$

6.1 Effective Resistance and Upper Bounding $\lambda_n(L_T^+ L_G)$

So, we know that the smallest non-zero eigenvalue of the matrix $L_T^+ L_G$ is at least one. We thus need to obtain now an upper bound on the value of the largest eigenvalue $\lambda_n(L_T^+ L_G)$ of that matrix.

To this end, we will use the fact that all the eigenvalues of the matrix $L_T^+ L_G$ are non-negative and therefore we have that

$$\lambda_n(L_T^+ L_G) \leq \sum_{i=1}^{n} \lambda_i(L_T^+ L_G) = \mathrm{Tr}(L_T^+ L_G),$$

where $\mathrm{Tr}(\cdot)$ denotes the matrix trace operator that corresponds to taking the sum of all the eigenvalues.

We can thus focus on bounding the trace $\mathrm{Tr}(L_T^+ L_G)$ of the matrix $L_T^+ L_G$. By linearity of the $\mathrm{Tr}(\cdot)$ operator and the fact that $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$, we obtain that

$$\mathrm{Tr}(L_T^+ L_G) \quad = \quad \mathrm{Tr}\left(L_T^+ \left(\sum_{e\in G} \chi_e \chi_e^T\right)\right) = \sum_{e\in G} \mathrm{Tr}\left(L_T^+ \chi_e \chi_e^T\right) = \sum_{e\in G} \mathrm{Tr}\left(\chi_e^T L_T^+ \chi_e\right) = \sum_{e} R_{\mathrm{eff}}^T(e), \quad (7)$$

where $R_{\mathrm{eff}}^H(e)$ denotes the *effective resistance* of the edge e in graph H defined as

$$R_{\mathrm{eff}}^H(e) := \chi_e^T L_H^+ \chi_e. \quad (8)$$

(Note that $\chi_e^T L_H^+ \chi_e$ is a scalar, so $\mathrm{Tr}(\chi_e^T L_H^+ \chi_e)$ is simply $\chi_e^T L_H^+ \chi_e$.)

It turns out that the effective resistance $R_{\mathrm{eff}}^H(e)$ is a very important quantity (and we will see it a number of times in the future). Observe that $\varphi = L_H^+ \chi_e$ can be interpreted as a vector of vertex potentials that induces an electrical u-v-flow of value 1 in H between the endpoints of the edge $e = (u, v)$. Consequently, the effective resistance of e in H $R_{\mathrm{eff}}^H(e) = \varphi_v - \varphi_u$ is the vertex potential difference between the endpoints of e induced in this electrical flow. (Note that $\varphi_v - \varphi_u$ is always non-negative.)

6.2 Bounding the Total Stretch of a Tree

Now, let us get a better grasp of the sum we obtained in (7) by understanding what the effective resistance $R_{\mathrm{eff}}^H(e)$ of an edge e corresponds to if the graph H is a tree T. Note that in this case, there is only one way of routing a flow from one endpoint u to the other endpoint v of the edge e: sending it along the unique u-v-path in the tree T. As a result, the effective resistance is equal to the length of this path. In other words,

$$R_{\mathrm{eff}}^T(e) = \mathrm{dist}_T(u, v) = \mathrm{stretch}_T(e), \quad (9)$$

where the *stretch* $\mathrm{stretch}_T(e)$ *of an edge* $e = (u, v)$ *in* T is defined as

$$\mathrm{stretch}_T(u, v) := \frac{\mathrm{dist}_T(u, v)}{l(e)},$$

i.e., the ratio of the distance between the endpoints of e in the tree T (this is the "stretched" length of the edge e in T) and the "original" length $l(e)$ of the edge e – see Figure 1. (Here, we are working in the unweighted case, so all $l(e) = 1$. But if our Laplacian L_G corresponded to a weighted graph G then the lengths would become $l(e) = \frac{1}{w_e}$, for each edge e.)

By (7), we can summarize our considerations so far with the following bound

$$\lambda_n(L_T^+ L_G) \leq \sum_{i=1}^{n} \lambda_i(L_T^+ L_G) = \mathrm{Tr}(L_T^+ L_G) = \sum_{e\in G} \mathrm{stretch}_T(e), \quad (10)$$

where the last sum is sometimes called the *total stretch* $\mathrm{stretch}_T(G)$ of the tree T with respect to the graph G.

The obvious question now is: how large can $\mathrm{stretch}_T(G) = \sum_{e\in G} \mathrm{stretch}_T(e)$ be?

Clearly, stretch of an edge can be at most n (at least in the unweighted case that we are considering here) and, in principle, this worst-case bound is tight. The latter follows by considering G to be a cycle. Every spanning tree in such G has to remove a single edge and this edge will have a stretch $n - 1$.

Consequently, the total stretch $\mathrm{stretch}_T(G)$ can be bounded by mn. Putting this together with (6) and (10) gives us that the condition number $\kappa(L_T^+ L_G)$ is at most

$$\kappa(L_T^+ L_G) = \frac{\lambda_n(L_T^+ L_G)}{\lambda_2(L_T^+ L_G)} \leq \mathrm{stretch}_T(G) \leq mn.$$

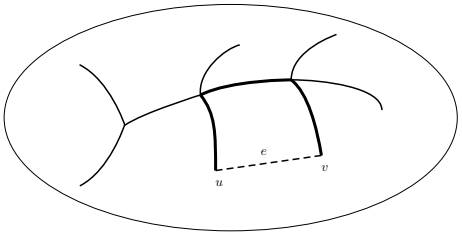

Figure 1: A stretch of an edge $e = (u, v)$ in a graph. The plain edges represent a spanning tree T. The bold edges represent the path in T connecting u to v. The stretch $\mathrm{stretch}_T(e)$ of this edge is 3.

This bound, however, is not too satisfying. After all, it does not even provide any significant improvement over the "worst-case" $O(n^2)$ condition number bound that we claimed in Section 4 in the context of the path graph. Given that we are utilizing tree preconditioners here and thus taking the Laplacian of this path graph as such preconditioner is completely legitimate, this should be an indication that our analysis is probably far from being tight.

Indeed, even though the worst-case stretch can be indeed close to n, bounding $\mathrm{stretch}_T(G)$ corresponds to bound the *average* stretch $\overline{\mathrm{stretch}}_T(G)$. That is, we have that

$$\mathrm{stretch}_T(G) = m \cdot \overline{\mathrm{stretch}}_T(G),$$

where $\overline{\mathrm{stretch}}_T(G) := \frac{1}{m}\sum_{e \in G} \mathrm{stretch}_T(e)$.

Note that in our example of G being a cycle graph the worst-case stretch is $n-1$, but the average stretch $\overline{\mathrm{stretch}}_T(G)$ is only $2(1-\frac{1}{n})$.

So, for an *arbitrary* G, how small average stretch can the *best* choice of a spanning tree T achieve? Somewhat astonishingly, one can prove the following theorem.

Theorem 3 (Low-stretch Spanning Trees) *For any graph G, one can construct in $\widetilde{O}(m)$ time a spanning tree T of G such that*

$$\overline{\mathrm{stretch}}_T(G) = \widetilde{O}(\log n).$$

(The $\widetilde{O}(\log n)$ above hides $\log\log n$ factors.)

In other word, no matter what the graph G is, there always exists a tree T with average stretch being essentially logarithmic. (Even though the worst-case stretch can be still close to n.) Furthermore, such tree can be found very fast – in nearly-linear time.

As a result, by Theorem 3, (6) and (10), we can conclude that

$$\kappa(L_T^+ L_G) = \frac{\lambda_n(L_T^+ L_G)}{\lambda_2(L_T^+ L_G)} \leq \mathrm{Tr}(L_T^+ L_G) \leq m \cdot \overline{\mathrm{stretch}}_T(G) = \widetilde{O}(m), \tag{11}$$

which by (5) enables us to solve Laplacian systems in time

$$O\left(\tau(L_T^+ L_G)\sqrt{\kappa(L_T^+ L_G)}\log\frac{\|x^*\|_{L_G}^2}{\varepsilon}\right) = \widetilde{O}\left((m+n)\sqrt{m}\log\frac{\|x^*\|_{L_G}^2}{\varepsilon}\right). \tag{12}$$

6.3 Going Beyond the $\widetilde{O}(m^{\frac{3}{2}})$ Pseudo-Condition Number Bound

The bound (12) constitutes some progress. However, it turns out that we can tighten our analysis even further. To this end, let us simplify our notation and denote the eigenvalues of the matrix $L_T^+ L_G$ simply as $0 = \lambda_1 < \lambda_2 \leq \ldots \leq \lambda_n$. Observe that taking T to be a low-stretch spanning tree as per Theorem 3 allows us to not only bound the largest eigenvalue $\lambda_n = \lambda_n(L_T^+ L_G)$ of the matrix $L_T^+ L_G$ but also the sum of *all* its eigenvalues. Specifically, by (11) and the definition of the trace, we have that

$$\sum_i \lambda_i = \mathrm{Tr}\left(L_T^+ L_G\right) \leq m \cdot \overline{\mathrm{stretch}}_T(G) = \widetilde{O}(m). \tag{13}$$

This statement turns out to give us a much stronger grasp of the complexity of the conjugate gradient method than what we can get from merely looking at the pseudo-condition number $\kappa^+(L_T^+ L_G)$.

Recall from the previous lecture that the number of iterations of the conjugate gradient method is tied to existence of certain univariate polynomial $q(z)$ that is equal to 1 for $z = 0$ and vanishes as quickly as possible at the points $z = \lambda_i$. Formally, in our case, the number of iterations that the conjugate gradient descent method takes to get a solution x to the linear system in the matrix $L_T^+ L_G$ with $\|x\|_{L_G}^2 \leq \varepsilon \|x^*\|_{L_G}^2$ is the minimum degree of a polynomial $q(z)$ such that

$$\begin{aligned} q(0) &= 1 \\ |q(\lambda_i)|^2 &\leq \varepsilon, \end{aligned} \tag{14}$$

for $i > 1$.

Also, we showed then that, for any $0 < \varepsilon$ and $0 < \lambda \leq \lambda'$, one can construct a polynomial $q^*_{\varepsilon,\lambda,\lambda'}(z)$ such that

$$\begin{aligned} q^*_{\varepsilon,\lambda,\lambda'}(0) &= 1 \\ |q^*_{\varepsilon,\lambda,\lambda'}(z)|^2 &\leq \varepsilon, \end{aligned} \tag{15}$$

for all $z \in [\lambda, \lambda']$, and the degree of $q^*_{\varepsilon,\lambda,\lambda'}(z)$ is only

$$O\left(\sqrt{\frac{\lambda'}{\lambda}} \log \varepsilon^{-1}\right). \tag{16}$$

Thus, taking $\lambda = \lambda_2$ and $\lambda' = \lambda_n$ recovers the standard pseudo-condition number based running time bound (5).

Now, it turns out that one can use (13) to construct a polynomial that has even better performance than the polynomial $q^*_{\varepsilon,\lambda_2,\lambda_n}(z)$ in our setting. To this end, for a given $\gamma > 1$, let $i(\gamma)$ be the largest index i such that $\lambda_{i(\gamma)} \leq \gamma$. Observe that by (6), $\lambda_i \geq 1$, for all $i \geq 2$, and thus, by (13), we can bound the number $n - i(\gamma)$ of eigenvalues larger than γ as

$$n - i(\gamma) \leq \frac{\sum_{i=i(\gamma)+1}^{n} \lambda_i}{\gamma} \leq \frac{\operatorname{Tr}\left(L_T^+ L_G\right)}{\gamma} \leq \widetilde{O}\left(\frac{m}{\gamma}\right). \tag{17}$$

Let us consider now a polynomial $Q_\varepsilon(z)$ defined as

$$Q_\varepsilon(z) := q^*_{\varepsilon,\lambda_2,\lambda_{i(\gamma)}}(z) \cdot q_{>\gamma}(z),$$

where

$$q_{>\gamma}(z) := \prod_{i=i(\gamma)}^{n} \left(1 - \frac{z}{\lambda_i}\right).$$

Intuitively, $Q_\varepsilon(z)$ is a product of two polynomials. One $q^*_{\varepsilon,\lambda_2,\lambda_{i(\gamma)}}(z)$ is "taking care" of all the eigenvalues λ_i being at most γ, while the other one $q_{>\gamma}(z)$ handles all the eigenvalues λ_i that are greater than γ via interpolation. See Figure 2.

Note that

$$Q_\varepsilon(0) = q^*_{\varepsilon,\lambda_2,\lambda_{i(\gamma)}}(0) \cdot q_{>\gamma}(0) = 1,$$

as desired. Further, we have that, for any $i(\gamma) < i \leq n$,

$$|Q_\varepsilon(\lambda_i)|^2 = \left|q^*_{\varepsilon,\lambda_2,\lambda_{i(\gamma)}}(\lambda_i)\right|^2 \cdot |q_{>\gamma}(\lambda_i)|^2 = 0 \leq \varepsilon,$$

since $q_{>\gamma}(\lambda_i) = 0$. Also, for any of the remaining eigenvalues λ_i with $2 \leq i \leq i(\gamma)$, we obtain that

$$|Q_\varepsilon(\lambda_i)|^2 = \left|q^*_{\varepsilon,\lambda_2,\lambda_{i(\gamma)}}(\lambda_i)\right|^2 \cdot |q_{>\gamma}(\lambda_i)|^2 \leq \varepsilon,$$

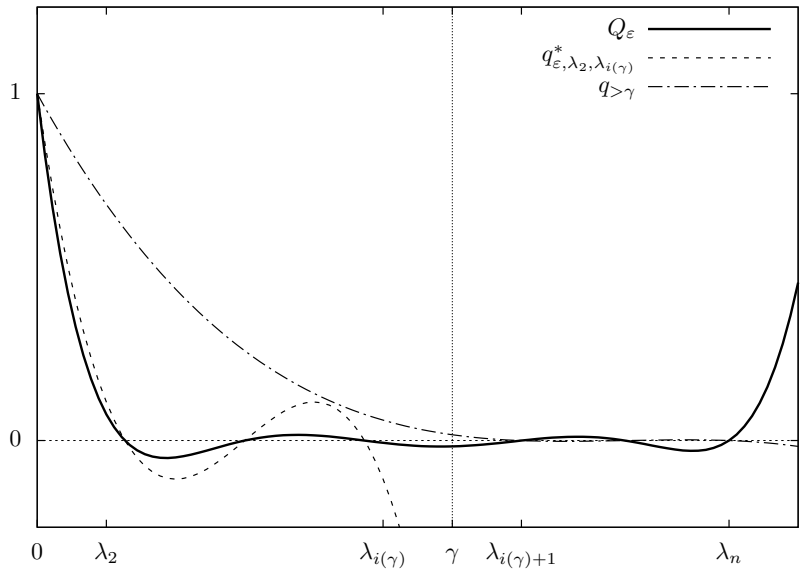

Figure 2: Examples of polynomials Q_ε, $q^*_{\varepsilon,\lambda_2,\lambda_{i(\gamma)}}$, and $q_{>\gamma}$.

where we used (15) and the fact that $|q_{>\gamma}(z)|^2 \leq 1$ if $z \in [-\gamma, \gamma]$.

So, the polynomial $Q_\varepsilon(z)$ satisfies the properties (14) and thus its degree $\deg(Q_\varepsilon)$ provides an upper bound on the number of iterations of gradient descent method. The running time bound that $Q_\varepsilon(z)$ implies is thus

$$\begin{aligned} O\left(\tau(L_T^+ L_G) \cdot \deg(Q_{\varepsilon'})\right) &= O\left(\left(\tau(L_G) + \tau(L_T^+)\right)\left(\deg\left(q^*_{\varepsilon',\lambda_2,\lambda_{i(\gamma)}}\right) + \deg\left(q_{>\gamma}\right)\right)\right) \\ &= \widetilde{O}\left((m+n)\left(\sqrt{\frac{\lambda_{i(\gamma)}}{\lambda_2}}\log\frac{1}{\varepsilon'} + \frac{m}{\gamma}\right)\right) = \widetilde{O}\left(m\left(\sqrt{\gamma}\log\frac{\|x^*\|^2_{L_G}}{\varepsilon} + \frac{m}{\gamma}\right)\right), \end{aligned}$$

where we used (6), (16), (17) as well as Lemma 2, and

$$\varepsilon' := \frac{\varepsilon}{\|x^*\|^2_{L_G}}$$

is the accuracy needed to ensure that the solution x we compute has $\|x\|^2_{L_G} \leq \varepsilon$.

Taking $\gamma = m^{\frac{2}{3}}$ in the expression above allows us to conclude the following theorem.

Theorem 4 *For any graph G and $\varepsilon > 0$, the conjugate gradient method with low-stretch spanning tree preconditioning computes a solution ϕ to the Laplacian system in the Laplacian L_G of G such that $\|\phi\|^2_{L_G} \leq \varepsilon$ in time*

$$\widetilde{O}\left(m^{\frac{4}{3}}\log\frac{\|\phi^*\|^2_{L_G}}{\varepsilon}\right).$$