IPCO Summer School June 25, 2017

Fast (Laplacian) Linear System Solving

Lecturer: Aleksander Mądry

1 Fast Linear System Solving

Today, let us focus on another fundamental computational problem: solving a linear system. We will discover that there is a lot of beautiful ideas and intimate connections to continuous optimization there.

1.1 Direct Methods

Consider a system of linear equations

$$Ax = b.$$

How could we go about solving it?

The first instinct is to simply compute A^{-1} and then multiply both sides of the linear system through it, obtaining an answer $x^* = A^{-1}b$.

Although this approach would certainly work, it has a couple of undesirable properties. First of all, computing an A^{-1} is quite costly computationally. Applying Gaussian elimination requires $O(n^3)$ time and if we resort to, fairly impractical, fast matrix multiplication algorithms we would obtain a running time of $O(n^\omega)$, where $2 \leq \omega \leq 2.373$. In either case, the running time would be super-quadratic and thus prohibitive from our point of view. Also, another problem with this approach relates to numerical issues. Computing A^{-1} might involve division by small numbers, which is quite problematic when working with real-world finite precision arithmetic.

A slightly better solution is to find a decomposition of A into $A = LU$ where L is lower triangular and U is upper triangular – so-called *LU-decomposition of A*. Once we have found such a decomposition, the problem reduces to finding an x such that $Ax = L(Ux) = b$. This can be easily done in two steps — first we would find z such that $Lz = b$, then we would find x, such that $Ux = z$. Each of those steps consist of solving a system of linear equation in a triangular matrix — this can be easily done in time $\mathcal{O}(n^2)$ via simple back-substitution. (Other decompositions also could be used, most notably one can use QR-decomposition: decompose $A = QR$, where Q is orthonormal and R is upper triangular. Again, solving a linear system both in R as well as in Q is quite easy.) Unfortunately, computing such decompositions also requires at least $\Omega(n^\omega)$ time.

Finally, one other problem with this kind of approaches is that they do not exploit sparsity of the matrix A. That is, many real-world matrices of interest are *sparse*, i.e., the number of their non-zero entries m is much smaller than the total number of entries $O(n^2)$. In such situations, one would hope to have methods whose complexity can take advantage of this fact. However, both A^{-1} and the LU-decomposition of A might be dense, i.e., have $\Omega(n^2)$ non-zero entries, even if A is very sparse, i.e., m is $O(n)$. This ability to exploit sparsity is particularly relevant in the context of Laplacian systems, as the sparsity of a Laplacian is within a constant of the sparsity of the underlying graph.

1.2 Iterative Approaches

Given all these drawbacks of the above approaches (which are usually called *direct methods*), we focus our attention on a different family of algorithms: *iterative approaches*.

These approaches, much in the spirit of this class, solve the linear system in a sequence of steps. They start from some initial guess x_1, and then iteratively refine it, which each consecutive answer x_s being increasingly better. The guiding principle here – and one of the key advantages of these method – is making each refinement step very simple and easy to compute. In fact, each step boils down to a small number of multiplications of A by some vector y. Note that this basic operation not only can be implemented fast, i.e., in $O(m)$ time, but it also exploits the sparsity of the problem.

On the other hand, one of the less desirable features of iterative methods is that they never really compute the exact solution. Instead, they provide a sequence of answer that only converges to the exact solution. Furthermore, the rate of this convergence will largely depend on numerical properties of the matrix A (usually, on its eigenvalue or singular values). Consequently, for some matrices the convergence will be very fast, while for others it might be extremely slow.

2 Iterative Approaches – the Setup

In the light of the above, our task is as follows. Given a system of linear equations

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

we want to design an iterative procedure that will provide us with increasingly better (approximate) solution. Our goal is to have each iteration to be simple and easy to execute. In fact, we want each iteration to boil down to a small number of basic operations being multiplication of the matrix A by a vector y. Note that each such multiplication can be implemented in $O(m)$ time, where m is the number of non-zero entries of A – called *sparsity*. This way each iteration of our approach is not only very fast to execute but also capable of benefiting from situations in which the matrix A is rather sparse, i.e., $m \ll n^2$, which is often the case in real-world applications (as well as in scenarios that we are interested in).

2.1 Making the Matrix A PSD

For the purpose of our analysis, we will assume that the matrix A is symmetric and positive-definite – we will call such matrices *PSD* matrices. Let us define these notions below.

Definition 1 *A square matrix A is symmetric iff $A^T = A$.*

One important implication of A being symmetric is that it has to have n *real* eigenvalues and we will denote these eigenvalues in non-decreasing order as $\lambda_1 \leq \ldots \leq \lambda_n$.

Definition 2 *A square matrix A is positive definite (resp., positive semi-definite), denoted as $A \succ 0$ (resp. $A \succeq 0$) iff $x^T Ax > 0$ (resp. $x^T Ax \geq 0$), for any $x \neq \vec{0}$.*

If A is symmetric, its positive definiteness (resp., positive semi-definiteness), implies – and, in fact, is equivalent to – having $\lambda_1 > 0$ (resp., $\lambda_1 \geq 0$). Also, these properties impose a partial order on the set of matrices, by defining $A \succ B$ (resp., $A \succeq B$) iff $A - B \succ 0$ (resp., $A - B \succeq 0$).

So, if our matrix A is PSD we know, in particular, that it is invertible and thus the linear system $Ax = b$ is non-degenerate.

At first glance, the above assumptions might seem pretty severe. However, due to the nature of our framework, this is not really the case, as long as A is invertible (which is quite basic assumption). To see this, note that if A is not symmetric then we can consider solving instead a linear system

$$\bar{A}x = \bar{b},$$

where $\bar{A} = A^T A$ and $\bar{b} = A^T b$. The matrix $\bar{A}$ is symmetric now and solving the linear system in it gives us the solution to our original problem. Also, it is important to note here, that we do not need to compute the matrix $\bar{A}$ explicitly here. All our methods require is the ability to quickly multiply a vector y times this matrix. And, we can easily do that just by multiplying y first by A and then by A^T, to get

$$A^T (Ay) = \bar{A}y,$$

and obviously this corresponds to just two matrix-vector multiplication in our original matrix A.

Similarly, if A is symmetric but not positive definite, the above transformation can be used again. Observe that the eigenvalues of $\bar{A} = A^T A = A^2$ are just $\lambda_1^2, \ldots, \lambda_n^2$. So, as long as none of the λ_is were zero (which would make A not invertible), the eigenvalues of $\bar{A}$ will be all strictly positive. That is, $\bar{A}$ will be PSD, as needed.

2.2 Measuring Our Error

As our iterative approaches will deliver to us increasingly better but still approximate solutions, we need to find a way to quantify our progress. There are two natural ways to define the error of our candidate solution x with respect to the actual solution x^*.

(a) $e(x) := x - x^*$, the so-called *left-hand side error*;

(b) $r(x) := b - Ax$, the so-called *right-hand side error*.

Clearly, once either of these errors becomes equal to an all-zero vector $\vec{0}$ we know $x = x^*$. However, these errors behave a bit differently when they are non-zero and we will alternate between them depending on the situation. We will also need a scalar error measure. One way to get that would be to look at the Euclidean norm of either $e(x)$ or $r(x)$. However, it turns out that it is more convenient to consider the norm of the error $e(x)$ induced by the matrix A. That is, to consider

$$\|e(x)\|_A, \tag{2}$$

where the *A-norm* $\|\cdot\|_A$ is defined as

$$\|y\|_A = \sqrt{y^T A y},$$

for any vector y. One can show that as long as A is PSD, which is the case in our context, the A-norm is indeed a norm. Also, note that when A is an identity matrix I then the A-norm becomes just the Euclidean norm.

3 Linear System Solving as an Optimization Problem

In the spirit of continuous optimization, we will develop our algorithm for solving linear system by casting this task as a convex optimization problem and then applying a gradient descent–based approach to it. Specifically, we will consider the following unconstrained minimization problem.

$$\min_x \frac{1}{2}\|e(x)\|_A^2. \tag{3}$$

Clearly, the optimum value of this program is 0 and it is achieved by taking $x = x^*$. So, solving the convex optimization problem (3) indeed captures solving the linear system $Ax = b$.

There is, however, an issue with the above formulation: to evaluate value of its objective $\frac{1}{2}\|e(x)\|_A^2$ at a point x, we need to know what x^* is! This makes it not too useful for computing what x^* actually should be.

Fortunately, there is an easy way to circumvent this problem. Observe that

$$\begin{aligned}\frac{1}{2}\|e(x)\|_A^2 &= \frac{1}{2}\left((x-x^*)^T A(x-x^*)\right) = \frac{1}{2}\left(x^T Ax - 2x^T(Ax^*) - (x^*)^T Ax^*\right)\\ &= \frac{1}{2}\left(x^T Ax - 2x^T b - (x^*)^T Ax^*\right) = g(x) + \frac{1}{2}(x^*)^T Ax^*,\end{aligned}$$

where

$$g(x) := \frac{1}{2}x^T Ax - b^T x, \tag{4}$$

and we used the fact that $A^T = A$ and that $Ax^* = b$.

Note that $g(x)$ does not depend on x^* at all and is only differing by an additive term of $\frac{1}{2}(x^*)^T Ax^*$ from $\frac{1}{2}\|e(x)\|_A$. As this additive term does not depend on x, we know that minimizing $g(x)$ (wrt x) is equivalent to minimizing $\frac{1}{2}\|e(x)\|_A$. Therefore, from now on, we can focus our attention on solving the following convex optimization problem

$$\min_x g(x). \tag{5}$$

Algorithm 1 Solving linear system $Ax = b$ with gradient descent method.

$x_1 \leftarrow \vec{0}$
for $s = 1 \dots T-1$ **do**
  $x_{s+1} \leftarrow x_s - \eta \cdot \nabla g(x_s)$
end for
return x_T

3.1 Applying the Gradient Descent Method

We now can apply the gradient descent method to the problem (5). The resulting algorithm is presented as Algorithm 1.

Observe that we have that

$$\nabla g(x) = \frac{1}{2}(2Ax) - b = Ax - b = -r(x). \tag{6}$$

So, the gradient descent step pushes us always in the direction of the residual error of the previous solution.

Now, from our analysis of gradient descent, we know that the performance of this algorithm boils down to analyzing the strong convexity α and smoothness β of our objective function g. Recall that these two quantities correspond to the value of the smallest (resp. largest) eigenvalue of the Hessian $\nabla^2 g(x)$ of g. Note though that

$$\nabla^2 g(x) = A. \tag{7}$$

So, we have that $\alpha = \lambda_1$ and $\beta = \lambda_n$.

At this point, we can use the performance guarantee we established for gradient descent to conclude that after

$$T = O\left(\kappa \log \frac{\|x^*\|}{\varepsilon}\right) \tag{8}$$

iterations, where $\kappa = \frac{\lambda_n}{\lambda_1}$ is the condition number of the matrix A, we have that

$$\frac{1}{2}\|e(x_T)\|_A^2 = |g(x_T) - g(x^*)| \leq \varepsilon,$$

where we used the fact that

$$\frac{1}{2}\|e(x_T)\|_A^2 = \frac{1}{2}\|e(x_T)\|_A^2 - \frac{1}{2}\|e(x^*)\|_A^2 = g(x_T) - g(x^*).$$

3.2 Geometric View on the Condition Number κ

As we can see above, the condition number κ has key impact on the convergence of our algorithm. It turns out that this quantity and its connection to the gradient descent method convergence has a particularly clean geometric interpretation in the context of our linear system solving scenario.

Recall that our objective function $g(x)$ (see (4)) is equal to $\|x - x^*\|_A^2$ up to a constant additive term of $(x^*)^T A x^*$. This implies that $g(x)$ is constant on every A-norm sphere that is centered on x^*, i.e., on every sphere

$$S_r := \{x \mid (x - x^*)^T A (x - x^*) = r\} = \{x \mid \|x - x^*\|_A^2 = r\},$$

for some r. That is, each S_r is a level set of g.

Observe that each such A-norm sphere corresponds to an ellipsoid in the Euclidean geometry. Each eigenvector of A determines one of this ellipsoid's principal axes and the corresponding eigenvalue gives us its scaling along that axis. Therefore, if the condition number $\kappa = \frac{\lambda_n}{\lambda_1}$ is large, this means that the ellipsoid is very "squeezed" along some of the axes while having κ close to 1 indicates that this ellipsoid is "round" and close to being an Euclidean sphere. See Figure 1 for illustration.

The fact that the function g is constant on each such ellipsoid implies that the gradient descent step direction $-\nabla g(x)$ of g at point x is perpendicular to the surface of this sphere and pointing towards its

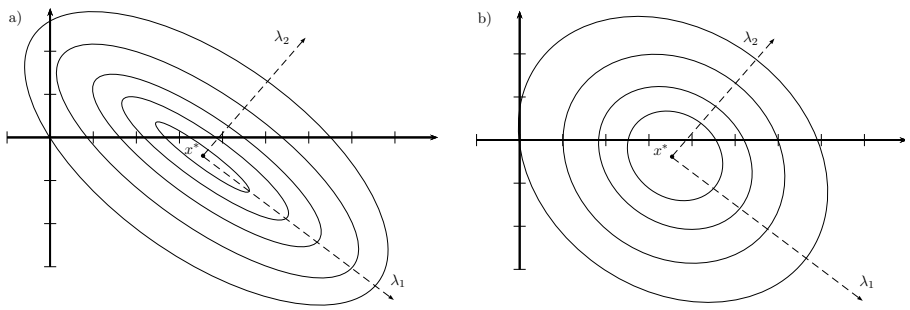

Figure 1: A-norm spheres in two dimensions: a) the case of $\kappa = \frac{\lambda_2}{\lambda_1}$ being large, i.e., $\lambda_2 \gg \lambda_1$; b) the case of $\kappa \approx 1$, i.e., $\lambda_1 \approx \lambda_2$. The dashed arrows denote the principal axes.

interior. In the ideal situation, i.e., if the condition number κ would be 1, this direction would point exactly towards the center of the sphere. As the center of each such sphere is the optimal solution x^*, taking gradient descent steps would result in very fast convergence. On the other hand, if κ is large the gradient descent step direction might be only loosely correlated with the direction towards the center/optimal solution x^*. This results in a "zig-zagging" behavior, as the algorithm gets sidetracked a lot in its journey towards optimum and thus a slower convergence of the algorithm. See Figure 2 for an illustration.

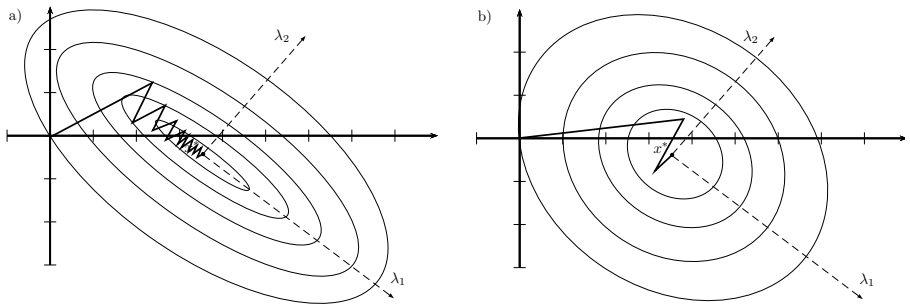

Figure 2: Trajectory of the gradient descent algorithm in two dimension: a) "zig-zagging" behavior corresponding to $\kappa = \frac{\lambda_2}{\lambda_1}$ being large; b) almost direct convergence when $\kappa \approx 1$, i.e., $\lambda_1 \approx \lambda_2$.

4 Building Polynomials with Gradient Descent

We see that e dependence on ε^{-1} is only logarithmic – which is what we want – and the key factor influencing the complexity of Algorithm 1 is the linear dependence on the condition number $\kappa = \frac{\lambda_n}{\lambda_1}$. It is natural to wonder then: can we improve this dependence?

In order to answer this question, we need to develop a different perspective on Algorithm 1 To this end, observe that

$$r_{s+1} = b - Ax_{s+1} = b - A(x_s + \eta r_s) = (I - \eta A)r_s = (I - \eta A)^s r_1 = (I - \eta A)^s b,$$

where r_s is a shorthand for $r(x_s)$. Consequently, we have that

$$x_T = x_{T-1} + \eta r_{T-1} = \sum_{i=1}^{T-1} \eta (I - \eta A)^i b = p_{T,\eta}(A)b,$$

where

$$p_{T,\eta}(x) := \sum_{i=1}^{T-1} \eta (1 - \eta x)^i \tag{9}$$

is a univariate polynomial of degree-$(T-1)$ that was applied to the matrix A.

In other words, one can view the output x_T of the Algorithm 1 as an application of degree-$(T-1)$ polynomial $p_{T,\eta}(x)$ to A and then multiplying it by the vector b.

Now, observe that the Algorithm 1 and its analysis is completely basis independent. So, we can choose a basis that is most convenient for us. The right choice here is to work in the eigenbasis of the matrix A, i.e., the orthonormal basis given by the eigenvectors of A. In this basis, all the objects we are analyzing become extremely simple, that is, diagonal. In particular, the matrix A becomes

$$A = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \lambda_n \end{bmatrix},$$

and the polynomial $p_{T,\eta}(x)$ applied to A is simply

$$p_{T,\eta}(A) = \begin{bmatrix} p_{T,\eta}(\lambda_1) & 0 & \dots & 0 \\ 0 & p_{T,\eta}(\lambda_2) & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & p_{T,\eta}(\lambda_n) \end{bmatrix}.$$

To understand what is the property of the polynomial $p_{T,\eta}(x)$ that we are looking for, recall that ideally we would like to compute $x^* = A^{-1}b$. So, our goal is to have the matrix $p_{T,\eta}(A)$ to become as close to possible to being A^{-1}, which in the eigenbasis of A is just

$$A^{-1} = \begin{bmatrix} \lambda_1^{-1} & 0 & \dots & 0 \\ 0 & \lambda_2^{-1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \lambda_n^{-1} \end{bmatrix},$$

Consequently, we see that there is an underlying optimization task at hand. Namely, we want to find a (univariate) polynomial $p(z)$ that, on one hand, approximates the (univariate) function $\frac{1}{z}$ well at each point corresponding to the eigenvalues of A, i.e., has

$$p(\lambda_i) \approx \lambda_i^{-1}, \tag{10}$$

for all $i = 1, \dots, n$, while, on the other hand, has its degree be at most $T-1$. The latter constraint arises since the degree of this polynomial corresponds directly to the number of iterations Algorithm 1 takes. (Observe that if there was no upper bound on the degree of that polynomial then we could get an arbitrarily good approximation in (10).)

It turns out that the polynomial $p_{T,\eta}(z)$ (see (9)) that the Algorithm 1 uses is very special. Namely, it is actually the $(T-1)$-th order Taylor series approximation of the function $\frac{1}{z}$ around the point $\frac{1}{\eta} = \frac{\lambda_1+\lambda_n}{2}$. In other words, the way our gradient descent method algorithm approaches the approximation task (10) is by computing the $(T-1)$-th order Taylor approximation of the intended function $\frac{1}{z}$ around the point $\frac{\lambda_1+\lambda_n}{2}$, which is the middle of the interval $[\lambda_1, \lambda_n]$ in which all the points of interest lie. This is really remarkable that such a principled and sophisticated choice came up out of our very general gradient descent design scheme!

As we implicitly proved, the choice of polynomial $p_{T,\eta}$ leads to the linear dependence of the number of iterations needed on the condition number κ. Is there maybe a better choice of the polynomial?

In short, the answer is: yes. The intuition here is that Taylor polynomials are not really best suited for solving the approximation tasks as (10). They are invented with the goal of providing the best possible *local* approximation around the given point. In (10), however, we are interested in getting good approximation on a number of points that are spread out over the whole interval $[\lambda_1, \lambda_n]$. As a

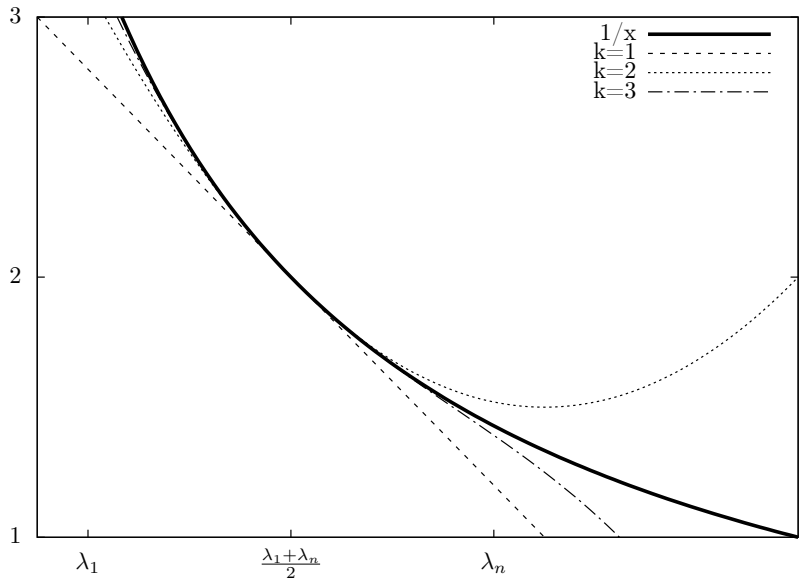

Figure 3: Approximation of the function $1/z$ around the point $z = \frac{\lambda_1+\lambda_n}{2}$ using Taylor approximation of orders $k = 1$, $k = 2$, and $k = 3$.

result, while the polynomial $p_{T,\eta}$ focuses on getting a very good fit around the middle point $\frac{\lambda_1+\lambda_n}{2}$, the corresponding much looser fit on the endpoints of that interval undermines the whole effort. See Figure 3 for an illustration.

Consequently, instead of the local approximation provided by Taylor series one should focus on Fourier-like series that were designed to provide such global type of approximation.

5 Conjugate Gradient Method: Beyond Taylor Approximation

Once we understood better the optimization question underlying our iterative linear system solving framework, let us try to abstract away the problem at hand and design a more direct algorithm for it.

To this end, let us note that in our framework each intermediate solution x_s, obtained after s iterations, belongs to a so-called *Krylov subspace (of order s)* $\mathcal{K}_s$, defined as

$$\mathcal{K}_s := \text{span}\left\{b, Ab, A^2b, \dots A^{s-1}b\right\}.$$

Furthermore, each vector $x \in \mathcal{K}_s$ corresponds to multiplying some polynomial in the matrix A of degree at most $s-1$ by the vector b. So, in principle, if we knew what this polynomial is, it could be computed in s iterations in our framework.

Now, from this perspective, we can view the Algorithm 1 as an approach to choosing a sequence of specific points $x_s \in \mathcal{K}_s$ for $s = 1, \dots, T$. As we discussed above, these choices correspond to taking successful higher-order Taylor series approximation of the function $1/x$ around the middle of the interval $[\lambda_1, \lambda_n]$. They, in turn, give us a solution to the approximation problem (10) that results in an iterations bound that has a linear dependence on the condition number κ.

However, once we know what our real goal is: to choose a point x_T in the Krylov subspace $\mathcal{K}_T$ of order T that aims to solve the problem (10), there might be a better approach to achieving it. In particular, instead of picking an explicitly constructed point x_T in hopes it will result in good performance, we can make our algorithm solve the underlying optimization directly.

This idea leads to an algorithm called *conjugate gradient method* that is presented as Algorithm 2. In oder words, this algorithm simply chooses x_T to be minimizer of our proxy objective function g for our target error measure $\|e(x)\|_A^2$.

Algorithm 2 Conjugate gradient method.

Compute

$$x_T := \operatorname*{argmin}_{x \in \mathcal{K}_T} g(x). \tag{11}$$

return x_T.

5.1 Efficient Implementation of the Conjugate Gradient Method

The first point we need to address is the implementation of the Algorithm 2. After all, our description of conjugate gradient method does not really explain how to find the point x_T efficiently. Also, the optimization problem (11) that defines the point x_T does not seem to be too different to our original optimization problem of minimizing the function g. Why should it be easier to solve?

It turns out that despite this seeming similarity, solving the optimization problem (11) can indeed be done efficiently. The key reason for that is the fact that this optimization problem exhibits a very convenient structure when one works in an appropriate basis. Specifically, a basis $v_1, \ldots, v_T$ that is A-orthogonal, i.e., a one in which each v_i and v_j with $i \neq j$ are orthogonal with respect to the *A-inner product* $\cdot_A$ defined as

$$x \cdot_A y := x^T A y.$$

In such A-orthonormal basis, our objective function g becomes separable, breaking down in T independent and simple to solve problems. Also, by applying Gram-Schmidt orthogonalization procedure in a careful manner, one can compute such an A-orthogonal basis $v_1, \ldots, v_T$ in only $O(T)$ (as opposed to $O(T^2)$) matrix-vertex multiplications of A. Working out the details of this implementation are a part of the Problem set 1. We will show there that indeed one can compute x_T in the conjugate gradient method can be implemented using only $O(T)$ matrix-vector multiplications of A overall.

5.2 Analyzing the Performance of the Conjugate Gradient Method

Once we discussed the implementation of the conjugate gradient descent, we can turn our attention to analyzing its performance. Specifically, once we know that computing the point x_T requires only $O(T)$ iterations of our framework, we would like to know how large T has to be to obtain the desired quality of the solution.

To this end, let us reformulate the underlying optimization question (10) into a slightly more convenient to work with form. More precisely, let us note that if our solution x_T is represented as $p_T(A)b$, for some degree-$(T-1)$ univariate polynomial $p(z)$, then we have that

$$e(x_T) = x_T - x^* = p_T(A)b - x^* = p_T(A)Ax^* - x^* = q_T(A)(-x^*), \tag{12}$$

where $q_T(z)$ is also a degree-T polynomial defined as

$$q_T(z) := 1 - z p_T(z). \tag{13}$$

So, there is a one-to-one correspondence between polynomials $p_T(z)$ and polynomials $q_T(z)$. Namely, any degree-$(T-1)$ polynomial $p_T(z)$ defines a degree-T polynomial $q_T(z)$ via (13). On the other hand, any degree-T polynomial $q_T(z)$ with $q_T(0) = 1$ defines a degree-$(T-1)$ polynomial $p_T(z) := \frac{q_T(z)-1}{z}$. (Note that the requirement that $q_T(0) = 1$ implies that the polynomial $q_T(z) - 1$ is divisible by z.)

Therefore, instead of thinking of the polynomials $p_T(z)$ and the corresponding problem (10) of approximating the $1/z$ function on the eigenvalue points, one can think of trying to design polynomials $q_T(z)$ that make the initial error $-x^*$ in (12) vanish as quickly as possibly. More formally, the latter task boils down to finding for a given desired error parameter ε a polynomial $q(z)$ that has as small degree as possible while satisfying

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

for all $i = 1, \ldots, n$. Once we find such a polynomial, we will know that by (12)

$$\|e(x)\|_A^2 = \|q_T(A)(-x^*)\|_A^2 \leq \|q_T(A)\|_2^2 \|x^*\|_A^2 = \max_i |q_T(\lambda_i)|^2 \|x^*\|_A^2 \leq \varepsilon \|x^*\|_A^2,$$

where we also used the fact that the eigenvectors of the matrix $q_T(A)$ are $q_T(\lambda_1), \ldots, q_T(\lambda_n)$. This is exactly the error guarantee that we would like to get in our algorithm. It is also worth noting that the above error guarantee bound works for *any* M-norm $\|\cdot\|_M^2$, i.e., we have

$$\|e(x)\|_M^2 \leq \varepsilon \|x^*\|_M^2,$$

for any PSD matrix M, even if it is unrelated to A.

In the light of the above, from now on, we just need to focus exclusively on the question underlying the optimization problem (28). That is, to bound the minimum degree T_ε that a polynomial $q(z)$ has to have in order to satisfy the conditions (28).

Observe that this question is *no longer* algorithmic! It is now just a question from approximation theory, a well-studied branch of mathematics. In particular, we would be completely satisfied by a purely existential statements, i.e., the polynomial $q(z)$ of the minimal degree just has to exist, we do not need to be able to efficiently compute it. (Although, one can show that the constructions we will use below can be performed efficiently.)

Now, turning to that question, if we were able to have the polynomial $q(z)$ have degree at most n then we can just resort to interpolation to get a perfect fit at all the eigenvalues. That is, we can consider a degree-n polynomial $q_{int}(z)$ defined as

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

Clearly, $q_{int}(0) = 1$ and $q_{int}(\lambda_i) = 0$, for all i. So, $T_\varepsilon \leq n$ or, in other words, once we allow the conjugate gradient method run for n iterations, the computed point x_T will be exactly x^*.

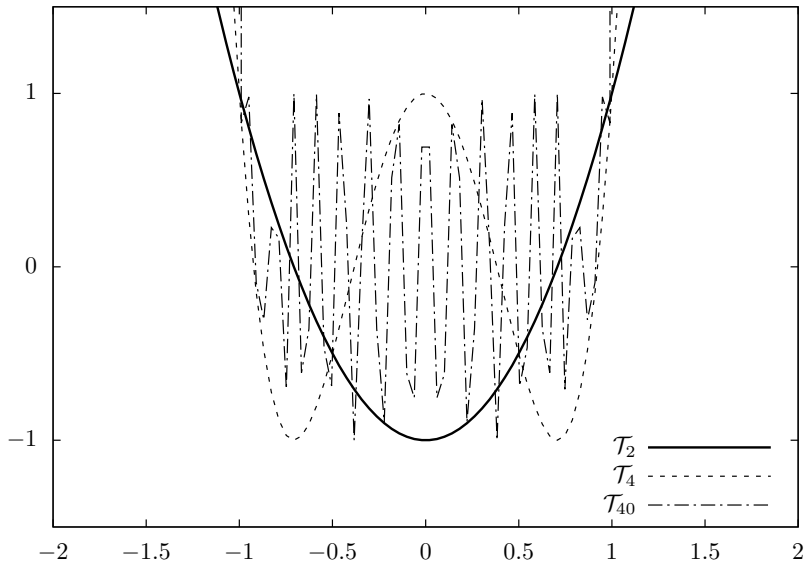

Figure 4: Chebyshev polynomials $\mathcal{T}_2$, $\mathcal{T}_4$, and $\mathcal{T}_{40}$.

The above statement should not be too surprising, as one could expect that the span of the Krylov subspace $\mathcal{K}_n$ is the whole of $\mathbb{R}^n$. Our key interest, however, is in obtaining bounds on T_ε that are much smaller than n. To get a hold on this regime, we need to resort to certain fundamental results in

approximation theory. Specifically, the object of our interest will be a family of extremal polynomials $\{\mathcal{T}_k(z)\}_k$ called *Chebyshev polynomials (of the first kind)*, defined as

$$\mathcal{T}_k(z) := \frac{1}{2}\left(\left(z+\sqrt{z^2-1}\right)^k + \left(z-\sqrt{z^2-1}\right)^k\right).$$

Note that, despite appearances, each $\mathcal{T}_k(z)$ is indeed a polynomial. (Try working this out for k being 1 and 2.) This polynomials are extremal in a number of ways. The way we will care about here most is that

$$|\mathcal{T}_k(z)| \leq 1, \tag{15}$$

when $z \in [-1,1]$ (they actually oscillate between -1 and 1 in that interval) and the value of $|\mathcal{T}_k(z)|$ has the sharpest increases outside of that interval among all the polynomials of degree k.

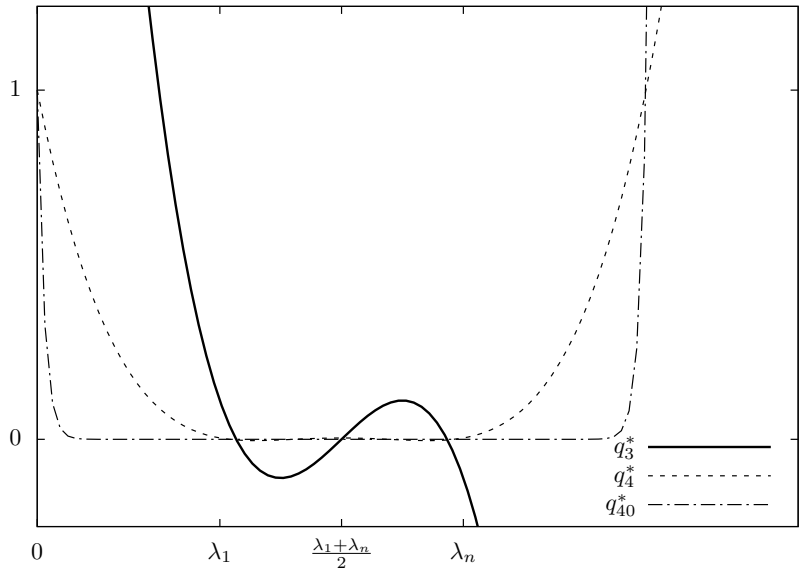

Figure 5: Polynomials q_3^*, q_4^*, and q_{40}^* over the interval $[\lambda_1, \lambda_n]$.

Let us consider now, for a given $k \geq 1$, a degree-k polynomial $q_k^*(z)$ defined as

$$q_k^*(z) := \frac{\mathcal{T}_k\left(\frac{\lambda_n+\lambda_1-2z}{\lambda_n-\lambda_1}\right)}{\mathcal{T}_k\left(\frac{\lambda_n+\lambda_1}{\lambda_n-\lambda_1}\right)}. \tag{16}$$

Observe that due to our normalization by $\mathcal{T}_k\left(\frac{\lambda_n+\lambda_1}{\lambda_n-\lambda_1}\right)$, we have that

$$q_k^*(0) = 1,$$

as desired. Furthermore, for any $z \in [\lambda_1, \lambda_n]$ we have by (15) that

$$\begin{aligned} |q_k^*(z)|^2 &\leq \mathcal{T}_k\left(\frac{\lambda_n+\lambda_1}{\lambda_n-\lambda_1}\right)^{-2} \\ &= \mathcal{T}_k\left(\frac{\kappa+1}{\kappa-1}\right)^{-2} \\ &= 4\left(\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k + \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k\right)^{-2} \\ &\leq 4\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^{2k} \leq 4\cdot\exp\left(-\frac{4k}{\sqrt{\kappa}+1}\right), \end{aligned} \tag{17}$$

where $\kappa = \frac{\lambda_n}{\lambda_1}$ is the condition number and we used the fact that, by (15),

$$\left| \mathcal{T}_k \left(\frac{\lambda_n + \lambda_1 - 2z}{\lambda_n - \lambda_1} \right) \right| \leq 1,$$

for any $z \in [\lambda_1, \lambda_n]$.

As a result, by using this polynomial $q_k^*(z)$ to solve our optimization problem, we obtain the following bound on the performance of the conjugate gradient method.

Theorem 3 *After T iterations of the conjugate gradient method (Algorithm 2), we have that*

$$\|e(x_T)\|_A^2 \leq 4 \cdot \exp\left(-\frac{4k}{\sqrt{\kappa} + 1} \right) \|x^*\|_A^2.$$

In other words, we obtain that number of iterations T_ε needed to get $\|e(x_T)\|_A^2 \leq \varepsilon$ is at most

$$T_\varepsilon \leq O\left(\sqrt{\kappa} \ln \frac{\|x^*\|_A}{\varepsilon} \right), \tag{18}$$

which improves our dependence on κ from linear to square root one.

Finally, it is important to note that the upper bound (17) on the value of our polynomial $q_k^*(z)$ holds not only for all the eigenvalue points λ_i, but actually it is valid for all z in the whole interval $[\lambda_1, \lambda_n]$. So, $q_k^*(z)$ provides a more robust solution than what is required.

6 Solving Laplacian Linear Systems

Let us now focus on solving a particular type of linear systems: *Laplacian linear system*, i.e., solving a linear system in graph Laplacian, of the form:

$$Lx = b, \tag{19}$$

where L is the Laplacian matrix of the underlying graph.

As we now have in hand algorithms for solving linear system, we need to examine how they can be applied in the context of Laplacian and what would be the resulting running times. This, in turn, requires us to analyze certain numerical properties of the Laplacian, such as its eigenvalue spectrum.

To this end, let us consider an undirected graph $G = (V, E, r)$ with $n = |V|$ vertices, $m = |E|$ edges, and edge resistances r_e assigned to each edge $e \in E$. Also, let us impose an arbitrary orientation on the edges of G. One way to define the Laplacian of G is as

$$L = B^T R^{-1} B,$$

where B is an $n \times m$ *edge-vertex incidence matrix* given by

$$B_{v,e} := \begin{cases} -1 & v \text{ is the head of } e \\ 1 & v \text{ is the tail of } e \\ 0 & \text{otherwise} \end{cases},$$

and R is an $m \times m$ diagonal *resistances matrix* defined as

$$R_{e,e'} := \begin{cases} r_e & \text{if } e = e' \\ 0 & \text{otherwise} \end{cases}.$$

However, there is a different, equivalent definition of the Laplacian that we mentioned too. Namely, we have that

$$L = D - A,$$

where D is a diagonal $n \times n$ *(weighted) degree matrix* D defined as

$$D_{uv} := \begin{cases} \sum_{e \in N(u)} r_e^{-1} & \text{if u=v} \\ 0 & \text{otherwise,} \end{cases}$$

and A is an $n \times n$ *adjacency matrix* given by

$$A_{uv} := \begin{cases} r_e^{-1} & \text{if } (u,v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Consequently, one can express the entries of the Laplacian explicitly as

$$L_{uv} := \begin{cases} \sum_{e \in N(u)} r_e^{-1} & \text{if } u = v \\ -r_e^{-1} & \text{if } e = (u,v) \in E \\ 0 & \text{otherwise,} \end{cases}$$

where $N(u)$ is the set of edges incident to the vertex u.

Interestingly, one can also decompose the Laplacian of G into a sum of elementary Laplacians

$$L = \sum_{e \in E} r_e^{-1} L^e = \sum_{e \in E} r_e^{-1} \chi_e \chi_e^T,$$

where each L^e is just a Laplacian of a simple graph on the vertex set V that contains only a single (unweighted) edge $e = (u,v)$ or, alternatively, an outer product of characteristic vectors χ_e with a -1 at u-th coordinate, 1 at v-th coordinate, and zeros at all the other coordinates. (So, each L^e is an $n \times n$ matrix with exactly four non-zero entries.)

Now, observe that L is a symmetric matrix. So, it has n real eigenvalues $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$. Furthermore, last time we discussed Laplacians we proved the following theorem about its eigenvalues.

Theorem 4 *Let L 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.

One of important implication of the above theorem is that L is *not* positive definite. That is, L is *positive semi-definite*, i.e., $L \succeq 0$, but $L \not\succ 0$. Even more importantly, as $\lambda_1 = 0$, the matrix L is also *not* invertible. This is very undesirable for us, as our algorithms for linear system solving assumed the underlying matrix is positive definite and, in particular, that it is invertible.

Fortunately, this turns out to be not too much of a problem. Observe that, as long as G is connected (which is always the case for us), the eigenspace of L corresponding to the eigenvalue 0 is just the one dimensional subspace of all the constant vectors, i.e., span($\vec{1}$). So, the linear system (19) can be solved whenever the demand vector b is orthogonal to that subspace. Formally, instead of considering the inverse L^{-1} of the Laplacian, which does not exist, one can consider the next best thing: the *Moore-Penrose pseudoinverse* L^+ defined as

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

where v_i is the eigenvector corresponding to eigenvalue λ_i in the orthonormal eigenbasis of L. In other words, L^+ behaves as an inverse of the Laplacian for every vector that is orthogonal to the eigenspaces corresponding to eigenvalue 0.

Consequently, the linear system (19) has a solution, given by $L^+ b$ as long as $b \perp \vec{1}$. Observe, however, that the latter condition boils down to insisting that the entries of the demand vector b sum up to zero. (As we will see, this will be always the case in our applications.)

Furthermore, one can check that once this $b \perp \vec{1}$ condition is satisfied, our algorithms end up computing a correct solution with the condition number κ becoming the *pseudo-condition number*

$$\kappa^+ := \frac{\lambda_n}{\lambda_2}.$$

Finally, it is worth pointing out that in this setting the solution that we end up computing will not be unique. That is, for any solution x^* to the Laplacian system 19, the solution $x^* + \alpha\vec{1}$, for any $\alpha \in \mathbb{R}$, will also be a solution. As we will see later, this is not a bug but a "feature", i.e., a manifestation of an important principle.

7 Upperbounding the Pseudo-condition Number of a Laplacian

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 4, 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 4 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* and we will talk about it soon. 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 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 $\Theta(n^2)$. Thishis 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.

8 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 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 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 5 *If G is a tree, then we can solve a Laplacian system $L_G x = b$ in the Laplacian L_G of G in $O(n)$ time.*

9 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

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

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 $Ax = b$ 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.)

[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.

Clearly, once this condition holds, the conjugate gradient method applied to the linear system $Ax = b$ 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 would give us that a guarantee in terms of the $\|\cdot\|_{P^{-1}A}$ norm, instead of the original $\|\cdot\|_A$ norm. However, 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.

10 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 5 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 (**??**) 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

$$\lambda_2(L_T^+ L_G) \geq 1. \tag{20}$$

10.1 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

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

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

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

$$\text{Tr}(L_T^+ L_G) \quad = \quad \text{Tr}\left(L_T^+ \left(\sum_{e \in G} \chi_e \chi_e^T \right) \right) = \sum_{e \in G} \text{Tr}\left(L_T^+ \chi_e \chi_e^T \right) = \sum_{e \in G} \text{Tr}\left(\chi_e^T L_T^+ \chi_e \right) = \sum_{e} R_{\text{eff}}^T(e), \tag{21}$$

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

$$R_{\text{eff}}^H(e) := \chi_e^T L_H^+ \chi_e. \tag{22}$$

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

It turns out that the effective resistance $R_{\text{eff}}^H(e)$ 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 $R_{\text{eff}}^H(e) = \varphi_v - \varphi_u$ 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.)

10.2 Bounding the Total Stretch of a Tree

Now, let us get a better grasp of the sum we obtained in (21) by understanding what the effective resistance $R^H_{\text{eff}}(e)$ 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,

$$R^T_{\text{eff}}(e) = \text{dist}_T(u, v) = \text{stretch}_T(e), \tag{23}$$

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

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

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 6. (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 (21), we can summarize our considerations so far with the following bound

$$\lambda_n(L_T^+ L_G) \leq \sum_{i=1}^{n} \lambda_i(L_T^+ L_G) = \text{Tr}(L_T^+ L_G) = \sum_{e \in G} \text{stretch}_T(e), \tag{24}$$

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

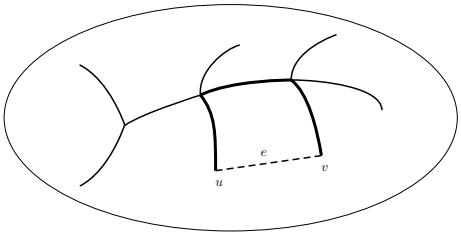

Figure 6: A stretch of an edge $e = (u, v)$ in a graph. The plain edges represent a spanning tree T. The bold edges represent the path in T connecting u to v. The stretch $\text{stretch}_T(e)$ of this edge is 3.

The obvious question now is: how large can $\text{stretch}_T(G) = \sum_{e \in G} \text{stretch}_T(e)$ 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 $\text{stretch}_T(G)$ can be bounded by mn. Putting this together with (20) and (24) gives us that the condition number $\kappa(L_T^+ L_G)$ is at most

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

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 8 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 $\text{stretch}_T(G)$ corresponds to bound the *average* stretch $\overline{\text{stretch}}_T(G)$. That is, we have that

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

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

Note that in our example of G being a cycle graph the worst-case stretch is $n-1$, but the average stretch $\overline{\text{stretch}}_T(G)$ 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 6 (Low-stretch Spanning Trees) *For any graph G, one can construct in $\widetilde{O}(m)$ time a spanning tree T of G such that*

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

(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 6, (20) and (24), we can conclude that

$$\kappa(L_T^+L_G) = \frac{\lambda_n(L_T^+L_G)}{\lambda_2(L_T^+L_G)} \le \text{Tr}(L_T^+L_G) \le m\cdot\overline{\text{stretch}}_T(G) = \widetilde{O}(m), \tag{25}$$

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

$$O\left(\tau(L_T^+L_G)\sqrt{\kappa(L_T^+L_G)}\log\frac{\|x^*\|_{L_G}^2}{\varepsilon}\right) = \widetilde{O}\left((m+n)\sqrt{m}\log\frac{\|x^*\|_{L_G}^2}{\varepsilon}\right). \tag{26}$$

10.3 Going Beyond the $\widetilde{O}(m^{\frac{1}{2}})$ Pseudo-Condition Number Bound

The bound (26) 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 $0=\lambda_1<\lambda_2\le\ldots\le\lambda_n$. Observe that taking T to be a low-stretch spanning tree as per Theorem 6 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 (25) and the definition of the trace, we have that

$$\sum_i \lambda_i = \text{Tr}\left(L_T^+L_G\right) \le m\cdot\overline{\text{stretch}}_T(G) = \widetilde{O}(m). \tag{27}$$

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 $\|x\|_{L_G}^2 \le \varepsilon\|x^*\|_{L_G}^2$ is the minimum degree of a polynomial $q(z)$ such that

$$\begin{aligned} q(0) &= 1 \\ |q(\lambda_i)|^2 &\le \varepsilon, \end{aligned} \tag{28}$$

for $i>1$.

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

$$\begin{aligned} q^*_{\varepsilon,\lambda,\lambda'}(0) &= 1 \\ |q^*_{\varepsilon,\lambda,\lambda'}(z)|^2 &\le \varepsilon, \end{aligned} \tag{29}$$

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

$$O\left(\sqrt{\frac{\lambda'}{\lambda}}\log\varepsilon^{-1}\right). \tag{30}$$

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

Now, it turns out that one can use (27) 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 (20), $\lambda_i \geq 1$, for all $i \geq 2$, and thus, by (27), we can bound the number $n - i(\gamma)$ of eigenvalues larger than γ as

$$n - i(\gamma) \leq \frac{\sum_{i=i(\gamma)+1}^{n} \lambda_i}{\gamma} \leq \frac{\operatorname{Tr}\left(L_T^+ L_G\right)}{\gamma} \leq \widetilde{O}\left(\frac{m}{\gamma}\right). \tag{31}$$

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 7.

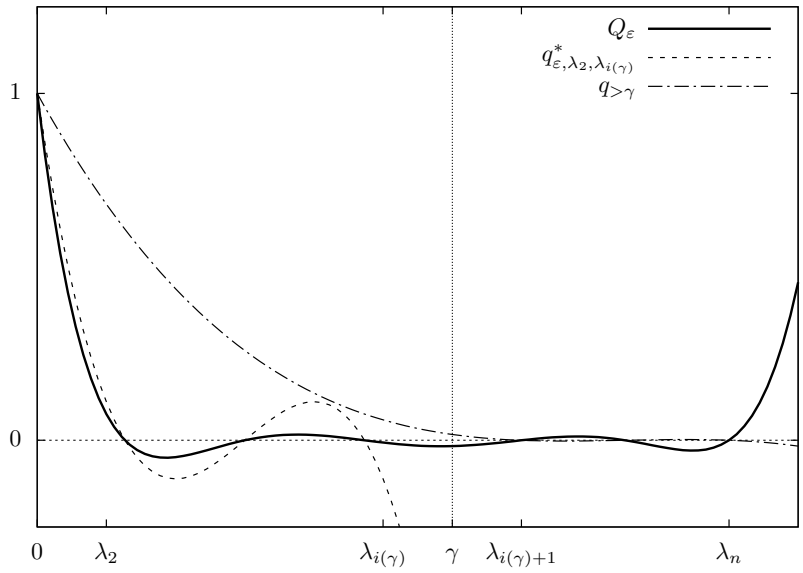

Figure 7: Examples of polynomials Q_ε, $q^*_{\varepsilon,\lambda_2,\lambda_{i(\gamma)}}$, and $q_{>\gamma}$.

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

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

where we used (29) and the fact that $|q_{>\gamma}(z)|^2 \leq 1$ if $z \in [-\gamma, \gamma]$.

So, the polynomial $Q_\varepsilon(z)$ satisfies the properties (28) 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 (20), (30), (31) as well as Lemma 5, 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 7 *For any graph G and $\varepsilon > 0$, the conjugate gradient method with low-stretch spanning tree preconditioning computes a solution x to the Laplacian system in the Laplacian L_G of G such that $\|x\|^2_{L_G} \leq \varepsilon$ in time*

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

It turns out that one can go much further and actually use more sophisticated *recursive* preconditioning to get a *nearly-linear time* Laplacian solver. That is, a solver that runs in time $\widetilde{O}\left(m \log 1/\varepsilon\right)$. This development is at the heart of many modern graph algorithms. In particular, all the modern maximum flow algorithm.