Math 361S Lecture notes
LDL^T and Cholesky factorizations, Iterative methods, finding eigenvalues

Jeffrey Wong

February 14, 2018

Topics covered

- Special types of matrices
  - Existence of LU factorization without pivoting
  - Diagonal dominance
  - Positive definiteness
- Symmetric, positive definite systems
  - LDU factorization; variants of GE for LU
  - LDL^T and Cholesky Factorizations
- Iterative methods
  - Theory for basic iterative methods
  - Jacobi and Gauss-Seidel
  - SOR
- Finding eigenvalues
  - Power method
  - Inverse power method

1 Cholesky factorization (and related ideas)

In this section we build up to solving an important special class of systems. Along the way, we will develop an alternate method (different from Gaussian elimination) for obtaining a variant of the LU factorization and introduce a few useful things.

1.1 Existence of an LU factorization

It is useful to know when pivoting is not needed when factoring $PA = LU$ (so that the factorization is simply $A = LU$). A simple (in theory) sufficient ondition is the following. Define the k-th **principal minor** of A to be the k by k submatrix from the $(1,1)$ entry to the (k,k) entry, i.e.

$$A_k = \begin{bmatrix} a_{11} & \cdots & a_{kk} \\ \vdots & \cdots & \vdots \\ a_{k1} & \cdots & a_{kk}. \end{bmatrix}$$

The theorem is as follows:

Theorem. *If the principal minors A_k for $k = 1, \cdots n$ are non-singular, then A has a factorization $A = LU$.*

The factorization could be computed using Gaussian elimination with no pivoting. (Note: with pivoting on, some rows might be swapped around).

1.2 Positive definite matrices

A symmetric $n \times n$ square matrix A is called (symmetric) **positive definite** if

$$x^T Ax > 0 \quad \text{for all } x \in \mathbb{R}^n.$$

If the inequality is not strict then the matrix is called positive semi-definite.

Note: A positive definite matrix does not have to be symmetric, but we will only work with the symmetric kind here.

(Symmetric) positive definite matrices have some equivalent characterizations:

- All the eigenvalues are positive
- $\det(A_k) > 0$ for all the principal minors A_k (the k by k submatrix starting from the $(1,1)$ entry down to the (k,k) entry)

The second condition is often easiest to check for small matrices. Positive definite matrices play a key role in many important problems in mathematics and in applications, so good numerical methods that can exploit their structure are in high demand.

Note that by the previous theorem, every positive definite matrix has an LU factorization. If the matrix is also symmetric, we can derive an effiicent algorithm to do so, which is the subject of the next sections.

1.3 Diagonal dominance

Positive definiteness is difficult to check in practice because one must either compute determinants (e.g. using Gaussian elimination) or compute eigenvalues (not easy).

A square matrix A is called **diagonally dominant** if

$$|a_{ii}| > \sum_{j=1\, j\neq i}^{n} |a_{ij}|$$

i.e. the diagonal entry is larger than the (absolute) sum of all other entries in that row.

Every *symmetric* diagonally dominant matrix is symmetric positive definite[1]. As a consequence, Gaussian elimination on A does not require pivoting (and in fact, if pivoting is used then no row swaps will be done; the correct entry is always already on the diagonal).

Of course, checking if a matrix is diagonally dominant is trivial. In many applications (e.g. in solving differential equations), one needs to solve a system $Ax = b$ for a matrix A that is diagonally dominant or provably positive definite.

2 LDU factorization

Before considering symmetric matrices, let us first derive an alternate method for computing the LU factorization. Suppose A is a square matrix. We seek unit lower triangular matrices L and U and a diagonal matrix $D = \text{diag}(d_1, \cdots, d_n)$ such that

$$A = LDU.$$

Note that the LDU factorization exists whenever the the LU factorization $A = LU$ from Gaussian elimination exists (since we can just 'extract' D by taking $d_i = u_{ii}$).

Notation: A diagonal matrix D is a matrix with diagonal entries $d_{ii} \neq 0$ and all others zero. It is often compactly denoted by

$$D = \text{diag}(d_1, \cdots, d_n)$$

where d_i is short for the (i,i)-th entry. The `diag(d)` command in MATLAB constructs the matrix D from the diagonal 'vector' `d` $= (d_1, \cdots d_n)$, and `diag(D)` extracts the diagonal vector from the matrix.

The method is derived by simply writing out the equations for each component of $A = LDU$ and figuring out the right order to solve for all the elements of L, D and U in sequence.

[1]The result is a consequence of Gershgorin's Disk theorem, the proof of which is straightforward.

Noting that $(DU)_{ij} = d_i u_{ij}$, the formula for the (i,s) entry of A is

$$a_{is} = \sum_{s=1}^{n} \ell_{is} d_i u_{sj} = \sum_{s=1}^{\min\{i,j\}} \ell_{is} d_i u_{sj}. \tag{1}$$

Now assume, for $k \geq 1$, that we have computed the following:

- the first $k-1$ rows of U
- the first $k-1$ columns of L
- $d_1, \cdots d_{k-1}$.

From (1) we can then compute the k-th row/column of U and L and d_k.

Compute d_k: Look at the (k,k) entry of A. By (1),

$$a_{kk} = \sum_{s=1}^{k-1} \ell_{ks} d_s u_{sk} + d_k \ell_{kk} u_{kk}.$$

This equation can be used to find d_k since $\ell_{kk} = u_{kk} = 1$.

Compute k-th row of U: Now look at the k-th row of A (to the right of the diagonal). Again, by (1),

$$a_{kj} = \ell_{kk} d_k u_{kj} + \sum_{s=1}^{k-1} \ell_{ks} d_s u_{sj}, \qquad j = k+1, \cdots n$$

This equation gives the k-th row of U (the unknown is marked in red).

Compute k-th column of L: Now for the k-th column of A (below the diagonal):

$$a_{ik} = \ell_{ik} d_k u_{kk} + \sum_{s=1}^{k-1} \ell_{is} d_s u_{sk}, \qquad i = k+1, \cdots n.$$

This gives the k-th column of L.

Continuing the process (for $k = 1, \cdots n$), we eventually obtain the entirety of L and U, so long as $d_i \neq 0$ for $i = 1, \cdots n$.

Extra note: One could simplify a bit by instead finding $A = L(DU)$ or $A = (LD)U$ i.e. assuming only one of L or U is unit triangular. These methods are called **Doolittle's method** and **Crout's method**. The methods all have more or less the same numerical properties as Gaussian elimination, and are essentially re-orderings of the Gaussian elimination algorithm. The main difference is that the operations are arranged differently, which have various advantages for optimizing the code. See Section 4.2 of the textbook.

3 Cholesky factorization

Gaussian elimination works more or less as well as the algorithms above for a general A. However, when A is symmetric and positive definite, the LDU factorization has additional structure that we should exploit.

A symmetric positive definite A has a **Cholesky factorization**

$$A = LL^T$$

where L is lower triangular (not unit). The proof of existence is instructive, as it illustrates a few theoretical tricks:

Proof. We know that A has an LU factorization where L is unit lower triangular. Since A is symmetric,

$$LU = U^T L^T.$$

Since $\det(L) = 1$, we know L is invertible, so

$$U(L^T)^{-1} = L^{-1}U^T.$$

It is easy to show that the left hand side is upper triangular, and the right hand side is lower triangular. Thus both are equal to a matrix that is both upper and lower triangular. But the only such matrix is a diagonal matrix D, so

$$U = DL^T.$$

Thus $A = LDL^T$, which is the LDU factorization of A. It is also straightforward to show that D is positive definite (see homework) so all the entries of D are positive. This means that

$$D^{1/2} = \operatorname{diag}(\sqrt{d_{11}}, \cdots \sqrt{d_{nn}})$$

exists[2], so

$$A = LD^{1/2}(L^{1/2}D^{1/2})^T = \tilde{L}\tilde{L}^T$$

for the upper triangular matrix $\tilde{L} = LD^{1/2}$. □

3.1 LDL^T

The proof above also shows that if A is symmetric and its principal minors are non-singular then its LU factorization exists and has the form

$$A = LDL^T$$

where L is unit lower triangular and D is diagonal. When A is positive definite this form and the Cholesky factorization are related in the way shown in the proof, i.e. $A = \tilde{L}\tilde{L}^T$ where $\tilde{L} = LD^{1/2}$.

It is easy to derive the algorithm for LDL^T factorization in the same way that is done for Cholesky factorization below (it is the same as for LDU, but we replace U with L^T).

[2]In general, $B = A^{1/2}$ means that B is a matrix such that $B^2 = A$. The matrix $A^{1/2}$ is, as you might expect, called a 'square root' of the matrix.

3.2 Algorithm

We derive an algorithm in the same way as for the LDU factorization (it is, more or less, the same algorithm but exploiting the symmetry). Since A is symmetric we need only consider elements on or above the diagonal:

$$a_{kj} = \sum_{i=1}^{n} \ell_{ki}\ell_{ji} = \sum_{i=1}^{k} \ell_{ki}\ell_{ji}, \qquad k \geq j.$$

We solve for the entries of L, starting with the first column, then the second and so on. To get the k-th column, first solve for ℓ_{kk} using

$$\ell_{kk}^2 = a_{kk} - \sum_{i=1}^{k-1} \ell_{ki}^2$$

and then for $r = k+1, \cdots n$:

$$\ell_{rk} = \frac{1}{\ell_{kk}} \left(a_{rk} - \sum_{i=1}^{k-1} \ell_{ri}\ell_{ki} \right).$$

So long as the computed ℓ_{ii}'s are positive, the iteration will go to completion, yielding the Cholesky factorization. No pivoting is necessary, and it can be shown that the Cholesky factorization algorithm is numerically stable.

Note on storage: As with Gaussian elimination, we can save some space by updating A directly instead of having a separate L. In the updates, we can replace ℓ with a and the algorithm still works.

Algorithm 1 Cholesky factorization (overwriting)

```
Input: sym. p.d. $A$
Output: $L$ such that $A = LL^T$ (stored in the lower. tri. part of $A$)
  for $k = 1 : n$ do
      $a_{kk} = \sqrt{a_{kk} - \sum_{s=1}^{k-1} a_{ks}^2}$
      for $i = k+1 : n$ do
          $a_{ik} = \frac{1}{a_{kk}} \left( a_{ik} - \sum_{s=1}^{k-1} a_{is}a_{ks} \right)$
      end for
  end for
```

3.3 Example

Let

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 5 & 6 \\ 2 & 6 & 17 \end{bmatrix}.$$

Note that, for the principal minors, $\det(A_1) = 1$ and $\det(A_2) = 5 - 1 = 4$ and $\det(A_3) = 36$, all of which are positive, so A is positive definite. We apply the algorithm to obtain the Cholesky factorization:

First column: $\ell_{11} = \sqrt{a_{11}} = 1$, and

$$\ell_{21} = \frac{1}{\ell_{11}}(a_{21}) = 1/1 = 1, \quad \ell_{31} = \frac{1}{\ell_{11}} a_{31} = 2.$$

We now have (x denoting unknown entries)

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & x & 0 \\ 2 & x & x \end{bmatrix}.$$

Second column: $\ell_{22} = \sqrt{a_{22} - \ell_{21}^2} = 2$ and

$$\ell_{32} = \frac{1}{\ell_{22}}(a_{32} - \ell_{31}\ell_{21}) = 2$$

which gives

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 2 & 2 & x \end{bmatrix}.$$

Finally, $\ell_{33} = \sqrt{a_{33} - \ell_{31}^2 - \ell_{32}^2} = 3$. Thus $A = LL^T$ where

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 2 & 2 & 3 \end{bmatrix}.$$

3.3.1 Obtaining LDL^T from Cholesky

Note that the $A = \tilde{L}D\tilde{L}^T$ where $\tilde{L}$ (unit lower triangular) and D (diagonal) can be obtained by taking D to have the square root of the diagonal entries of L and then $\tilde{L} = LD^{-1/2}$. The result is

$$\tilde{L} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 9 \end{bmatrix}.$$

Extra note: It is possible to do pivoting to move the largest entry in the diagonal to the right place. To preserve symmetry, we must consider only **symmetric pivoting**:

$$A \to PAP^T$$

If P swaps rows i and j then PAP^T swaps the (j,j) entry with the (i,i) entry of A. Pivoting is useful when obtaining the Cholesky factorization when some of the d_i's encountered are nearly zero.

4 Iterative methods

The methods we have looked at so far are **direct methods**, in which the system is reduced to something trivial in a finite number of steps. With exact arithmetic, an exact solution is produced after a finite amount of work.

Another approach to solving $Ax = b$ is to devise an **iterative method**, where we successively improve an approximation that will converge to the true solution.

Key motivation: Typically, a direct method has to be run to completion to have a usable answer, but this answer is likely to be as accurate as we can get. On the other hand, iterative methods may get satisfactorily 'close' to the exact solution within only a few (simple) steps - even though it may take an infinite number of steps to reach the exact solution.

Here we consider the most basic iterative methods, which have the form

$$Mx^{(k+1)} = Nx^{(k)} + b \tag{2}$$

where $A = M - N$ is a **splitting** of A into the sum of two matrices. To determine what to choose, we need some theory to say when the iteration will converge. Write

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b = Gx^{(k)} + c$$

where $G = M^{-1}N$. This is the matrix that will determine the convergence properties. Let $e^{(k)} = x^{(k)} - x$ be the error vector. Then

$$e^{(k+1)} = Ge^{(k)}.$$

Taking norms (any one), we find that

$$\|e^{(k+1)}\| \leq \|G\|\|e^{(k)}\| \implies \|e^{(k)}\| \leq \|G\|^k\|e^{(0)}\|.$$

It follows that if $\|G\| < 1$ in any subordinate matrix norm, then the iteration will converge to x, i.e. $\|x^{(k)} - x\| \to 0$ as $k \to \infty$. Note that once we have established convergence in one norm, it follows that the iteration converges in all norms.

A more precise statement makes use of the following theorem (see K&C, Section. 4.6):

Theorem 1. *The spectral radius* $\rho(A) = \max|\lambda|$ *is given by*

$$\rho(A) = \inf_{\|\cdot\|} \|A\|$$

where the infimum is taken over all subordinate matrix norms.

In particular, if $\rho(A) < 1$ then there is a matrix norm for which $\|A\| < 1$. Thus, it follows from the theorem that if $\rho(G) < 1$ then the iteration (2) converges.

The iterative method (2) will be useful if the splitting $A = M - N$ is chosen so that

i) $\rho(M^{-1}N) < 1$

ii) Computing the solution to $Mx = y$ is simple.

4.1 Jacobi method

4.1.1 Theory

Let L_A, U_A and D be the lower/upper triangular and diagonal parts of A (not the same as in the LU factorization!):

$$L_A = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ a_{21} & 0 & \ddots & \ddots & \vdots \\ a_{31} & a_{32} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & 0 \end{bmatrix}, \qquad U_A = \begin{bmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & 0 & a_{23} & \ddots & a_{2n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{n-1,n} \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

and $.D = \operatorname{diag}(a_{11}, \cdots, a_{nn})$.

For Jacobi's method we take $M = D$ and $N = -(L_A + U_A) = D - A$ (i.e. $-A$ but without the diagonal entries), leading to

$$x^{(k+1)} = -D^{-1}(L_A + U_A)x^{(k)} + D^{-1}b.$$

It is straightforward to show that

Theorem 2. *If A is diagonally dominant then $\rho(D^{-1}(L_A + U_A)) < 1$ and so the Jacobi iteration converges to the solution to $Ax = b$ for any starting vector $x^{(0)}$.*

Proof. Since A is diagonally dominant we have

$$\|G\|_\infty = \|D^{-1}(L_A + U_A)\|_\infty = \max_{1 \le i \le n} \frac{1}{|a_{ii}|} \left(\sum_{j=1, j \ne i}^{n} |a_{ij}| \right) < 1.$$

It follows from this inequality that Jacobi's method converges; the theorem on the spectral radius shows that $\rho(G) < \|G\|_\infty < 1$. □

4.1.2 Implementation

In practice, there is a simple way to compute the iteration. Write out $Ax = b$:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}.$$

Now we 'solve' for x_i in the i-th equation:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \ne i}^{n} a_{ij} x_j \right).$$

Jacobi's method uses the values of $x^{(k)}$ on the right side to get $x^{(k+1)}$:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k)}\right), \qquad i = 1, \cdots n.$$

The entries 'evaluated' at $k+1$ are labeled in red. Note that we do not overwrite $x_i^{(k)}$ with its updated value, because the old value is still needed to update all the other components. Further note that the order in which we calculate updated entries of x is not important ($i = 1, \cdots, n$ can be done in any order), which makes Jacobi's method amenable to parallel computation.

4.2 Gauss-Seidel

We can, however, use the updated values of x_i immediately as they become available, which leads to the **Gauss-Seidel method**.

4.2.1 Implementation

Proceed as in the Jacobi iteration, but use the updated components of x immediately as they become available (again, entries 'evaluated' at $k+1$ are labeled in red; these use the updated values):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}.$$

The formulas for the update are

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j<i}^{n} a_{ij}x_j^{(k+1)} - \sum_{j=1, j>i}^{n} a_{ij}x_j^{(k)}\right). \tag{3}$$

Note that when computing the i-th component of the updated $x^{(k+1)}$, the only components of $x^{(k+1)}$ needed are the ones we have already computed (from 1 to $i-1$). **Note that** the update must be computed in the order $i = 1, \cdots n$ (or backwards, with i decreasing from n to 1).

In practice, the update (4) can be simply coded as

$$x_i \leftarrow \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j\right), \qquad i = 1, \cdots n.$$

The simplicity makes Gauss-Seidel an appealing choice for an iterative method. Note that using the udpated values immediately is convenient when computing each x_i one at a time; if it is faster to do calculations in parallel, the Jacobi iteration may be preferable.

4.2.2 Theory

Gauss-Seidel can also be written in matrix form. Let $A = L_A + D + L_U$ as before, and take $M = L_A + D$:

$$x^{(k+1)} = -(L_A + D)^{-1}U_A x^{(k)} + (L_A + D)^{-1}b.$$

The matrix for the iteration is then $G = -(L_A + D)^{-1}U_A$. Like Jacobi, if A is diagonally dominant then Gauss-Seidel converges (see K&C for the proof). Another related result is that

Theorem 3. *If A is symmetric positive definite then Gauss-Seidel converges independent of the starting vector $x^{(0)}$.*

The proof involves showing that $\rho(G) < 1$, which takes a bit of work (proof omitted). The theorem is notable because SPD matrices appear so often in applications.

Key point: In practice, some trial and error is involved in getting iterative methods to work well on a given linear system, because there is no single 'best' method. For an iterative method $x^{(k+1)} = Gx^{(k)} + c$, the error decreases by at worst a factor of $\rho(G)$ at each step. If $\rho(G)$ is close to 1, then the convergence may be prohibitively slow. The right iterative method to use depends on the matrix A, the splitting $A = M + N$ and the spectral radius of $\rho(M^{-1}N)$, which can be difficult to identify. Preconditioning can help to reduce the size of $\rho(G)$, but doing so is itself a non-trivial problem.

4.3 SOR (optional)

A disadvantage of Gauss-Seidel is that if $\rho(G)$ Is close to 1, the method will converge rather slowly (too slow to be o any use). One way to correct this is to adjust the splitting. For a *relaxation parameter* $\omega, \in (0, 1)$, use the update

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1,j<i}^{n} a_{ij}x_j^{(k+1)} - \sum_{j=1,j>i}^{n} a_{ij}x_j^{(k)}\right) + (1-\omega)x_i^{(k)}. \qquad (4)$$

If $\omega = 1$ this is Gauss-Seidel; otherwise the method 'relaxes' the update by leaving a part of the old value (the $(1-\omega)x_i^{(k)}$ part). In matrix terms,

$$M_\omega x^{(k)} = N_\omega x^{(k)} + \omega b$$

where

$$M_\omega = D + \omega L_A, \qquad N_\omega = M_\omega - \omega A.$$

It can be shown that if $0 < \omega < 2$ and A is sym. positive definite then the method converges. Ideally, one could find a value $\omega \in (0, 2)$ so that

$$\rho(I - \omega M_\omega^{-1}A) < 1$$

is minimized, which can greatly accelerate convergence. For some systems, such optimization is possible, but in general it is difficult to find the right value of ω.

4.4 A practical concern: when to stop?

According to the theory, we have error bounds of the form

$$\|e^{(k)}\| \le C\|G\|^k$$

where C depends on the initial error. Typically, it is true that the best bound uses the spectral radius instead of $\|G\|$ and that this is the 'true' decay rate of the error:

$$\|e^{(k)}\| \approx C\rho(G)^k.$$

Thus every step k is expeced to reduce the error by about a factor of $\rho(G)$. However, we do not know the constant, as we cannot calculate the error vector without the exact solution.

Knowing when to stop involves using some heuristics and educated guesses. Suppose we want a solution $\tilde{x}$ that is within 10^{-N} of the true solution (i.e. about $N-1$ significant digits plus a little more accuracy). Two choices for stopping criteria are

$$\|x^{(k+1)} - x^{(k)}\| \le \epsilon_{\text{rel}}\|x^{(k)}\|$$

and/or

$$\|x^{(k+1)} - x^{(k)}\| \le \epsilon_{\text{abs}}$$

where ϵ_{rel} and ϵ_{abs} are **relative** and **absolute** tolerances chosen in advance. Typically one would choose a relative tolerance to be on the order of the desired accuracy (e.g. 10^{-N}, perhaps somewhat smaller to be safe. Note that neither condition guarantees that the error is less than ϵ. If $\rho(G)$ is close to one, we may have to pick tolerances much less than the desired accuracy.

One can be a bit more precise about all this (e.g. monitor how fast the solution is converging, estimate the error from quantities we can compute like $\|x^{(k+1)} - x^k\|$ and so on), which will be addressed in later topics.

A third condition is to stop when the residual is small:

$$\|Ax^{(k)} - b\| < \epsilon$$

(or a relative version). The usefulness of this condition depends on whether you care more about the residual or the error, the condition of A and whether $Ax^{(k)}$ is costly to compute (if the algorithm needs $Ax^{(k)}$ anyway, then we have the residual for free).

5 A brief discussion of computing eigenvalues

In deriving error bounds for iterative methods, we implicitly used the fact that

$$\lim_{k\to\infty} A^k x = \vec{0} \quad \text{if and only if } \rho(A) < 1.$$

This idea of applying a matrix repeatedly to a vector can be used to estimate eigenvalues. For simplicity, we will consider only the case where A is an $n \times n$ real-valued matrix with eigenvalues $\lambda_1, \cdots, \lambda_n$ and a basis of eigenvectors

$$\{v_1, v_2 \cdots v_n\} \text{ spanning } \mathbb{R}^n.$$

Note that each v_i is a vector (the subscript does not refer to the i-th component).

5.1 The power method

Assume now that there is a single eigenvalue of largest magnitude. The **power method** is a simple way to obtain this value. Label the eigenvalues as follows:

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

Note that if A has real-valued entries, it must be that λ_1 is real (why?).

Intuition: The idea here is that the v_1 component of a vector $x \in \mathbb{R}^n$ is the one that grows the most when x is multiplied by A. Thus if we apply A repeatedly:

$$x, Ax, A^2x, A^3x, \cdots$$

the v_1 term will dominate, and the other components will vanish:

$$A^k x \approx c\lambda_1^k v_1 + \text{smaller terms}.$$

Analysis: Since the eigenvectors form a basis, any $x \in \mathbb{R}^n$ can be written in the form

$$x = c_1 v_1 + \cdots + c_n v_n$$

for scalars c_i. Since $A^k v_j = \lambda_j^k v_j$ it follows that

$$A^k x = \sum_{j=1}^n c_j A^k v_j = \sum_{j=1}^n c_j \lambda_j^k v_j = \lambda_1^k \left(c_1 v_1 + \sum_{j=2}^n c_j \left(\frac{\lambda_j}{\lambda_1}\right)^k v_j \right).$$

All the terms except the first go to zero in magnitude as $k \to \infty$, so

$$A^k x = \lambda_1^k c_1 v_1 + e_k \quad \text{with} \quad \|e_k\| = O((\lambda_2/\lambda_1)^k).$$

Now we need to extract the value of λ_1 by taking a 'ratio' of $A^k x$ and $A^{k-1}x$. To divide, we need scalars to work with. To this end, let $w \in \mathbb{R}^n$ be such that $w^T v_1 \neq 0$. Then

$$\frac{w^T A^k x}{w^T A^{k-1} x} = \lambda_1 + O((\lambda_2/\lambda_1)^k) \text{ as } k \to \infty \tag{5}$$

which is straightforward to check. Thus the power method computes the **largest eigenvalue** (in magnitude) of the matrix, and the speed of convergence depends on λ_2/λ_1.

Implementation: In practice, we normalize x at each step to have $\|x\|_2 = 1$, which keeps elements from growing exponentially in size. Because λ is computed from a ratio, the normalization does not affect the answer. The resulting algorithm is:

- Pick $q^{(0)}$ such that $\|q^{(0)}\|_2 = 1$
- For $k = 1, 2, \cdots$:
    - $x^{(k)} = Aq^{(k-1)}$
    - $q^{(k)} = x^{(k)}/\|x^{(k)}\|_2$
    - $\lambda^{(k)} = (q^{(k)})^T Aq^{(k)}$.

The result is that $\lambda^{(k)}$ converges to the largest eigenvalue λ_1 and $q^{(k)}$ converges to the eigenvector. Note that the vector w has been replaced by $q^{(k)}$; it is not too hard to show that this choice[3] also works instead of a 'fixed' w chosen in advance.

One should, of course, implement the above while storing only two vectors and a scalar, and evaluating Aq only once per iteration:

- Pick q such that $\|q\|_2 = 1$ and set $x = Aq$
- For $k = 1, 2, \cdots$:
    - $q = x/\|x\|_2$
    - $\lambda = x^T q$
    - $x = Aq$

Note: Various stopping criteria exist; one can, for example, check whether $Aq - \lambda q$ is small (a residual condition for the equation $Aq = \lambda q$, which should hold if q is an eigenvector).

[3]The expression $(q^T aq)/(q^T q)$ is called the *Rayleigh quotient*, useful in some variants of the power method)

5.2 Inverse power method

A simple change allows us to compute the **smallest** eigenvalue (in magnitude). Let us assume now that A has eigenvalues

$$|\lambda_1| \geq |\lambda_2| \cdots > |\lambda_n|.$$

Then A^{-1} has eigenvalues λ_j^{-1} satisfying

$$|\lambda_n^{-1}| > |\lambda_2^{-1}| \geq \cdots \geq |\lambda_n^{-1}|.$$

Thus if we apply the power method to A^{-1}, the algorithm will give $1/\lambda_n$, yielding the smallest eigenvalue of A (after taking the reciprocal at the end).

Note that in practice, instead of computing A^{-1}, we first compute an LU factorization of A, and then solve

$$Ax^{(k+1)} = x^{(k)}$$

at each step, which only takes $O(n^2)$ operations after the initial work.

Now suppose instead we want to find the eigenvalue closest to a number μ. Notice that the matrix $(A - \mu I)^{-1}$ has eigenvalues

$$\frac{1}{\lambda_j - \mu}, \qquad j = 1, \cdots n.$$

The eigenvalue of largest magnitude will be $1/(\lambda_{j_0} - \mu)$ where λ_{j_0} is the closest eigenvalue to μ (assuming there is only one). Thus we need only apply the power method to $(A - \mu I)^{-1}$, computing the iterates

$$(A - \mu I)x^{(k+1)} = x^{(k)}.$$

This iteration is also referred to as the **inverse power method** or **inverse iteration**. The method is a cheap, often effective way of computing the eigenvalue closest to μ. In applications, often only the extreme eigenvalue (largest, smallest, closest to a number) is the one that matters, in which case inverse iteration can be a good choice.

5.3 More on eigenvalues

The methods above give an extreme eigenvalue, but all the other eigenvalues disappear as the iteration proceeds. Finding the other eigenvalues is more difficult, and the power/inverse power methods are not particularly well suited to doing so. There are some ways to reduce the problem after finding one eigenvalue (to find the next largest using the power method, and so on), but they can be problematic.

For computing all the eigenvalues of A, there is a powerful class of iterative methods that can be used, such as the QR algorithm, which will not be covered here (it uses the QR factorization, the numerical implementation of which is a bit involved).