Lecture 14: Eigenvalue Computations

This lecture discusses a few numerical methods for the computation of eigenvalues and eigenvectors of matrices. Most of this lecture will focus on the computation of a few eigenvalues of a large symmetric matrix, but some nonsymmetric matrices also will be considered, including the Google matrix. The QR-algorithm for both symmetric and nonsymmetric matrices of small to modest size will be discussed towards the end of these notes. We use the same notation as in Lecture 13. The norm $\|\cdot\|$ denotes the Euclidean vector norm.

The power method - symmetric matrices

Let the symmetric $n \times n$ matrix A have an eigenvalue, λ_1, of much larger magnitude than the remaining eigenvalues, and assume that we would like to determine this eigenvalue and an associated eigenvector. This can be done fairly efficiently and very simply with the power method. This method proceeds as follows: Let $v \in \mathbb{R}^n$ be an approximation of an eigenvector associated with the eigenvalue of largest magnitude. Carry out the following computations expressed in pseudo-MATLAB:

```
for k=1,2,3,...
  w=A*v;
  lambda=norm(w);
  v=w/lambda;
end
```

The scalars lambda converge to the magnitude of the desired eigenvalue and the vector v to an associated eigenvector as j increases. The reason for this is quite easy to see. Let $\{\lambda_j, v_j\}_{j=1}^n$ denote the eigenpairs of A. Since A is symmetric, we may assume that the eigenvectors v_j form an orthonormal basis of $\mathbb{R}^n$. The initial vector v for the power method can be expressed in terms of this basis, i.e.,

$$v = \sum_{j=1}^{n} \alpha_j v_j$$

for certain coefficients α_j, where we assume that $\alpha_1 \neq 0$. By assumption $|\lambda_1| \gg |\lambda_j|$ for $2 \leq j \leq n$. Multiplying v by the matrix A k times yields

$$A^k v = \sum_{j=1}^{n} \alpha_j A^k v_j = \sum_{j=1}^{n} \alpha_j \lambda_j^k v_j$$

and dividing by λ_1^k gives

$$\frac{A^k v}{\lambda_1^k} = \alpha_1 v_1 + \sum_{j=2}^{n} \alpha_j \left(\frac{\lambda_j}{\lambda_1}\right)^k v_j. \tag{1}$$

Since the quotients λ_j/λ_1, $2 \leq j \leq n$, are of magnitude smaller than unity, the vector $A^k v/\lambda_1^k$ converges to a multiple of v_1 as k increases.

The vector v determined by k steps of the pseudo-code differs from $A^k v/\lambda_1^k$ only by a scaling factor. It follows that the vectors v generated by the code converges to v_1 up to an arbitrary scaling factor. If $v = v_1$, then

$$||Av|| = |\lambda_1| \, \|v_1\| = |\lambda_1|.$$

This shows that the scalars lambda in the code converge to the magnitude of λ_1. We can determine the proper sign of λ_1 by comparing the signs of nonvanishing components of w and v. For instance, if v is an accurate approximation of v_1 whose first component is nonvanishing, then $\text{sign}(\lambda_1)$ is the sign of the quotient of the first components of w and v. The iterations with the power method are terminated when 2 consecutively determined values of lambda are sufficiently close.

Exercise 1. Generate the matrix $M = \mathsf{rand}(100)$, the vector $v = \mathsf{rand}(100, 1)$, and let $A = (M + M^T)/2$. This is a symmetric matrix; it is the symmetric part of the randomly generated matrix M. Apply the power method to A with initial vector v and print successive values of lambda. Do the scalars lambda converge quickly or slowly to the largest eigenvalue of A? Explain! Hint: Compute all eigenvalues of A with the MATLAB function eig. □

Exercise 2. Generate the matrix $M = \mathsf{randn}(100)$, the vector $v = \mathsf{randn}(100, 1)$, and let $A = (M+M^T)/2$. Apply the power method and print successive values of lambda. Do the scalars lambda converge quickly or slowly to the magnitude of the eigenvalue of largest magnitude of A? Explain! □

The power method - nonsymmetric matrices

The power method can be applied to the computation of the eigenvalue of largest magnitude of certain nonsymmetric matrices. For instance, the eigenvalue of largest magnitude of any matrix with only positive entries is real and positive, and the associated eigenvector has real positive entries only. This is the Perron-Frobenius theorem. Matrices with only real positive entries are said to be *positive*. The power method applied to a positive matrix with an initial vector with positive entries only often gives rapid convergence. We remark that the matrix in Exercise 1 above is symmmetric and positive, while the matrix in Exercise 2 is not.

The Google matrix, which determines the results of web searches with Google, is a huge (about $8.1 \cdot 10^9 \times 8.1 \cdot 10^9$ in 2006) nonsymmetric positive matrix. The power method is used to determine the left eigenvector associated with its largest eigenvalue, which is known to be 1. This eigenvector only has positive components, which are normalized to sum to one. The PageRank of web page j is the value of the jth component of the eigenvector. A large value indicates that the page is important. The following example provides some details on how the Google matrix is constructed. The book by Langville and Meyer [1] is a nice reference, which provides an in-depth, but not very mathematical, discussion.

Example 1. The PageRank of a web page P_i, denoted by $r(P_i)$, is the weighted sum of all the pages pointing to P_i,

$$r(P_i) = \sum \frac{r(P_j)}{|P_j|}.$$

Summation is over all pages P_j with pointers to P_i and $|P_j|$ denotes the number of outlinks from page P_j. A large PageRank signals that the page is important and therefore may be relevant for your search.

The PageRank numbers $r(P_j)$ are not explicitly known. They are determined by an iterative method as follows. Let $r_k(P_j)$ be the approximation of the PageRank of page P_j determined at the kth step of the iterative method. We determine PageRank of P_i at step $k + 1$ by the formula

$$r_{k+1}(P_i) = \sum \frac{r_k(P_j)}{|P_j|}. \tag{2}$$

This is the power method applied to a suitable matrix, known as the Google matrix. To see that (2) is one step of the power method, we introduce the hyperlink matrix $H = [h_{jk}]$ with $h_{jk} = 1/|P_j|$ if there is a link from page j to page k. The remaining entries of H are zero. Let $v^{(k)}$ be the PageRank vector at iteration k; the j entry of $v^{(k)}$ is given by $r_k(P_j)$. Then (2) can be written in the form

$$(v^{(k+1)})^T = (v^{(k)})^T H. \tag{3}$$

This is the power method transposed. Transposing (3) gives

$$v^{(k+1)} = H^T v^{(k)},$$

which shows that (2) is the power method applied to the transpose of H.

Note that the matrix H is huge; it is of size $n \times n$ with $n > 8 \cdot 10^9$. The matrix H also is very sparse. On average there are 10 outlinks from each web page. Therefore, on average, each row of H contains only 10 nonvanishing entries. Only the nonzero entries are stored. The computational effort required to evaluate the matrix-vector product in (3) is proportional to n.[1]

Unfortunately, the iterations (3) are not guaranteed to converge. However, the matrix H easily can be adjusted to secure convergence. First note that H may have zero rows caused by so-called dangling nodes, i.e.., web pages without outpointers. We replace these rows by the vector $1/n\, e^T$, where $e = [1, 1, \ldots, 1]^T$. This gives the $n \times n$ matrix S. Let the vector $a \in \mathbb{R}^n$ have the entry 1 in those rows that are zero rows of H; the remaining entries of A are zero. Then we can express S as

$$S = H + \frac{1}{n} a e^T.$$

Note that this matrix is not explicitly stored.

Convergence of the power method is not guaranteed when applied to S either, since S may have zero entries. We therefore modify S to obtain a positive matrix, the Google matrix. It is given by

$$G = \alpha S + (1 - \alpha) \frac{1}{n} e e^T.$$

The matrix G is not explicitly stored. Instead one expresses the matrix in the form

$$G = \alpha(H + \frac{1}{n} a e^T) + (1 - \alpha) \frac{1}{n} e e^T = \alpha H + (\alpha a + (1 - \alpha) e) \frac{1}{n} e^T. \tag{4}$$

Note that only the matrix H and vector a have to be stored. The entries of the vector e are known and therefore do not have to be explicitly stored. The matrix-vector products required for the iterations

$$(v^{(k+1)})^T = (v^{(k)})^T G. \tag{5}$$

are evaluated by using the decomposed representation (4) of G. Since G is positive the iterations are guaranteed to converge. The rate of convergence depends on the ratio of the second largest to largest eigenvalues of G. The largest eigenvalue of G is 1, because the largest eigenvalue of H is 1. The second largest eigenvalue of G generally is α. The value $\alpha = 0.85$ is said to be used by Google. The PangeRank is said to be updated once a week.

[1] If H were a dense matrix, then the evauation of a matrix-vector products would have been proportional to n^2 arithmetic floating point operations.

When searching something with Google the PageRank determines, in part, the result of the search. Pages with a large PageRank are listed before pages with a small PageRank. □

Exercise 3. Let the largest and second largest eigenvalue of G be 1 and 0.85, respectively. How many iterations are required to determine the eigenvector with about 3 significant digits? □

Exercise 4. Generate the matrix $A = \mathsf{rand}(100)$, the vector $v = \mathsf{rand}(100, 1)$. This is (generally) a nonsymmetric nonnegative matrix. Apply the power method to A with initial vector v and print successive values of lambda. Do the scalars lambda converge quickly or slowly to the largest eigenvalue of A? Hint: Compute all eigenvalues of A with the MATLAB function eig. □

Exercise 5. The MATLAB command $\mathsf{magic}(n)$ determines an $n \times n$ matrix, whose entries form a magic square. Determine the eigenvalues of a few magic squares using the power method. What is the largest eigenvalue? Explain! □

Subspace iteration

The purpose of this method is to determine several, say ℓ, of the largest eigenvalues and associated eigenvectors of a large symmetric matrix $A \in \mathbb{R}^{n\times n}$. Let the matrix $V \in \mathbb{R}^{n\times \ell}$ have linearly independent columns. The function orth in the pseudo-code for subspace iteration below orthogonalizes these columns.

```
V=[v_1,v_2,...,v_l]=orth(V);
for k=1,2,3,...
  W=[w_1,w_2,...,w_l]^T=A*V;
  Lambda=diag[norm(w_1),norm(w_2),...,norm(w_l)];
  for j=1,2,...,l
    v_j=w_j/Lambda_j;
  end
  V=[v_1,v_2,...,v_l]=orth(V);
end
```

The orthogonalization of the columns in each loop secures convergence to eigenvalues and eigenvectors associated with the ℓ eigenvalues of A of largest magnitude. Without orthogonalization all columns would converge to eigenvectors associated with the eigenvalue(s) of largest magnitude.

Let the eigenvalues λ_j of A be ordered so that their magnitude decreases with increasing index,

$$|\lambda_1| \geq |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_\ell| \gg |\lambda_{\ell+1}| \geq \ldots \ .$$

The eigenvalues λ_ℓ and $\lambda_{\ell+1}$ are required to be of different magnitudes. Then subspace iteration yields convergence towards the jth eigenvector for $j \leq \ell$ with rate $|\lambda_j/\lambda_{\ell+1}|$.

Exercise 6. Determine the two largest eigenvalues of the symmetric tridiagonal 2000×2000 matrix with diagonal entries 2 and sub- and super-diagonal entries -1 by subspace iteration. Use sparse storage format for the matrix, i.e., first generate it as a dense matrix and then store it as a sparse one using the MATLAB command sparse. The sparse format only stores the nonvanishing elements of the matrix and of its LU-factorization. Hint: Type help sparse in MATLAB. The MATLAB command spy lets you see which entries are stored. The dense matrix can be generated by the command toeplitz. □

Inverse iteration

Assume that we would like to determine an eigenvalue of the symmetric matrix A in the vicinity of μ, say $\mu = 1$. We would like to use a method that is simple to code, such as the power method. However, when μ is inside the convex hull of the spectrum of A, the power method will not converge to a desired eigenpair. Inverse iteration is a simple modification of the power method, which makes it possible to determine the desired eigenpair. The following pseudo-code describes the inverse iteration method:

```
Compute the LU-factorization of A-mu I;
for k=1,2,3,...
  Solve (A-mu*I)w=v for w using the LU-factorization;
  lambda=norm(w);
  v=w/lambda;
end
```

Inverse iteration can be seen to be the power method applied to the matrix $(A - \mu I)^{-1}$. This matrix has the same eigenvectors as A. Let λ_j be an eigenvalue of A. Then $(\lambda_j - \mu)^{-1}$ is an eigenvalue of $(A - \mu I)^{-1}$; see Exercise 7. Inverse iterations gives convergence to the eigenvalue $(\lambda_j - \mu)^{-1}$ of largest magnitude of the matrix $(A - \mu I)^{-1}$. The expression $|\lambda_j - \mu|^{-1}$ is the largest when λ_j is the closest to μ. In order for inverse iteration to converge, there must be a single closest eigenvalue of A to μ. Similarly as the power method, inverse iteration is terminated when two consecutively computed values of lambda are sufficiently close. The desired eigenvalue approximation has to be determined from lambda and μ. The parameter μ in inverse iteration is referred to as the *shift*.

Exercise 7. Let $\{\lambda, v\}$ be an eigenpair of A. Show that v is an eigenvector of $(A - \mu I)^{-1}$. What is the corresponding eigenvalue? □

Exercise 8. Give an expression for the rate of convergence of the eigenvector associated with the eigenvalue closest to μ for inverse iteration similar to (1). □

Exercise 9. Determine the smallest eigenvalue of the symmetric tridiagonal matrix of Exercise 6. Use sparse storage format for the matrix. Then also the factors in the LU-factorization are stored as sparse matrices. Verify this with the MATLAB command spy. □

Rayleigh quotient iteration

The parameter μ in inverse iteration is important for being able to determine an eigenvalue close to μ. The closer μ is to an eigenvalue, the faster the convergence; see Exercise 8. This suggests that we may obtain even faster convergence by moving μ closer to the desired eigenvalue every iteration. This is done in Rayleigh quotient iteration.

Let v be an approximation of the eigenvector v_1. We would like to use v to determine an improved approximation of λ_1. This is achieved by solving the least-squares problem

$$\min_{\alpha} \|Av - \alpha v\|. \tag{6}$$

Here α is the unknown, v the "matrix", and Av the "right-hand side" of a standard least-squares problem. This least-square problem seeks to determine the best "approximate eigenvalue" associated with the "approximate eigenvector" v.

The normal equations associated with (6) are

$$v^T v\,\alpha = v^T A v,$$

and have the solution

$$\alpha = \frac{v^T A V}{v^T v}. \tag{7}$$

This quotient is referred to as the *Rayleigh quotient*. It is our best available approximation of the desired eigenvalue. We therefore use it as shift in inverse iteration. This gives the Rayleigh quotient iteration method:

```
for k=1,2,3,...
  Solve (A-alpha*I)w=v for w using LU-factorization;
  lambda=norm(w);
  v=w/lambda;
  alpha=v'*A*v;
end
```

The desired output are alpha and the vector v. The Rayleigh quotient iteration method requires LU-factorization of the matrix A-alpha*I in every iteration. This generally is the most expensive part of the method, and one step of Rayleigh quotient iteration is more expensive than one step of inverse iteration. Generally Rayleigh quotient iteration requires fewer iterations than inverse iteration and can be competitive, in particular, when high accuracy is desired.

Exercise 10. Apply Rayleigh quotient iteration to the problem in Exercise 9. Compare the rate of convergence with inverse iteration. □

The QR-algorithm

This algorithm is designed to compute all eigenvalues and associated eigenvector of a small to moderately sized matrix $A \in \mathbb{R}^{n\times n}$. We will focus on the computation of eigenvalues and first describe the algorithm for nonsymmetric matrices A. At the end of this subsection, we discuss simplifications that arise when A is symmetric.

Let μ be a shift close to a desired eigenvalue. The basic step of the QR-algorithm is the QR-factorization of the matrix $A - \mu I$ and the multiplication of the factors in reverse order:

```
for k=1,2,3,...
  1. Compute QR-factorization: A-mu*I => Q and R;
  2. Multiply factors: A:=R*Q + mu*I;
end
```

Here Q is an $n \times n$ orthogonal matrix and R an $n \times n$ upper triangular matrix. The matrix in line 2 of the above algorithm is similar to the one in line 1. This can be seen by expressing the matrix R in line 1 as

$$R = Q^T(A - \mu I).$$

Substituting this expression into line 2 yields

$$A = RQ + \mu I = Q^T(A - \mu I)Q + \mu I = Q^T AQ - \mu Q^T Q + \mu I = Q^T AQ,$$

where the last equality follows from the fact that Q is orthogonal. Since $Q^T = Q^{-1}$ the matrices A and $Q^T AQ$ are similar and therefore have the same eigenvalues.

The QR-algorithm is generally applied to upper Hessenberg matrices. These are matrices of the form

$$\tilde{A} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}.$$

Thus, all entries below the subdiagonal of an upper Hessenberg matrix vanish. We can bring A into upper Hessenberg form by orthogonal similarity transformation. In other words, there is an orthogonal matrix U, such that

$$\tilde{A} = UAU^T.$$

is an upper Hessenberg matrix. The orthogonal matrix U can be determined as a product of Householder-type matrices H_k. These are $n \times n$ matrices of the form

$$H_k = \begin{bmatrix} I & 0 \\ 0 & H \end{bmatrix},$$

with leading $k \times k$ principal submatrix the identity matrix, the trailing principal $(n-k) \times (n-k)$ matrix is a Householder matrix, and whose remaining elements are zero; see Lecture 12 from the fall semester for details on Householder and Householder-type matrices. We will use that Householder-type matrices are symmetric and orthogonal.

In the first step, we apply the Householder-type matrix H_1 from the left to generate zeros in the first column of A below the subdiagonal. Thus,

$$H_1 A = \begin{bmatrix} * & * & * & * & * \\ + & + & + & + & + \\ 0 & + & + & + & + \\ 0 & + & + & + & + \\ 0 & + & + & + & + \end{bmatrix}.$$

Elements marked by $+$ may be nonvanishing and are generally not the same as in A; elements marked by $*$ are the same as in A. Thus, multiplication by the Householder-type matrix H_1 does not change the entries of the first row of A. Similarly, multiplying $H_1 A$ by $H_1 = H_1^T$ from the right does not change the entries in the first column. In particular, the zeros generated in $H_1 A$ are preserved and we obtain

$$H_1 A H_1 = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{bmatrix}.$$

Note that this matrix is similar to A.

We now proceed by applying a Householder-type matrix H_2 from the left to H_1AH_1 to generate zeros below the subdiagonal of the second column. This gives the matrix

$$H_2H_1AH_1 = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & + & + & + & + \\ 0 & 0 & + & + & + \\ 0 & 0 & + & + & + \end{bmatrix}.$$

Entries that may be nonvanishing and different from those in H_1AH_1 are marked by $+$. The similarity transformation is completed by applying H_2 from the right. This does not affect the entries in the first 2 columns of $H_2H_1AH_1$. We obtain

$$H_2H_1AH_1H_2 = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix}.$$

We apply similarly the Householder-type matrix H_3 from the right and left, to obtain the upper Hessenberg matrix

$$\tilde{A} = H_3H_2H_1AH_1H_2H_3 = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix},$$

which is similar to the given matrix A.

We apply the iterations of the QR-algorithm to the upper Hessenberg matrix $\tilde{A}$. The shift μ is often chosen to be the eigenvalue of the trailing principal 2×2 matrix of $\tilde{A}$ closest to the last diagonal entry. The upper Hessenberg form of the matrix is preserved by the iterations. The QR-algorithm therefore generates a sequence of upper Hessenberg matrices. The last subdiagonal element of the Hessenberg matrices in this sequence typically converges to zero. Only rarely another subdiagonal entry becomes zero. When the last subdiagonal entry vanishes, the last diagonal entry is an eigenvalue. We can continue the computations with the leading $(n-1) \times (n-1)$ principal submatrix of the Hessenberg matrix. This submatrix also is of upper Hessenberg form. The reduction of size is referred to as *deflation*. In actual implementations in finite-precision floating-point arithmetic, deflation is carried out when the last subdiagonal element is of sufficiently small magnitude. The computations proceed until a 2×2 upper Hessenberg matrix remains. Its eigenvalues can be determined in closed form. The eigenvalue approximations determined by the QR-algorithm converge at least quadratically with the iteration number.

The computation of the QR-factorization of an upper Hessenberg matrix is much cheaper than determining the QR-factorization of a general matrix of the same size. This depends on that the Householder-type matrices are sparse; see Exercise 12 below.

We turn to symmetric matrices A. Then the upper Hessenberg matrix $\tilde{A}$ is symmetric. It therefore is symmetric and tridiagonal. This matrix form is preserved by the QR-algorithm.

Exercise 11. Show that $\lambda = 3$ is an eigenvalue of the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 0 & 4 & 5 & 6 \\ 0 & 0 & 0 & 3 \end{bmatrix}.$$

Exercise 12. Compute the QR-factorization of the upper Hessenberg matrix

$$\tilde{A} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 0 & 4 & 5 & 6 & 7 \\ 0 & 0 & 6 & 7 & 8 \\ 0 & 0 & 0 & 9 & 10 \end{bmatrix}.$$

Do the Householder matrices in the Householder-type matrices have a special form? If so, explain. □

Exercise 13. The number of iterations required by the QR-algorithm typically is independent of the size of the matrix. Under this assumption, how fast does the computational work grow with n when the given matrix A is i) nonsymmetric and of upper Hessenberg form, and ii) symmetric and tridiagonal. How does this computational effort compare with the initial reduction of the matrix to upper Hessenberg or symmetric tridiagonal form? □

Exercise 14. The MATLAB command hilb(n) determines the $n \times n$ Hilbert matrix. Compute the eigenvalues of Hilbert matrices of orders $3, 4, 5, \ldots$. Determine experimentally the growth of the condition number of Hilbert matrices with n. Does the condition number grow linearly, quadratically, exponentially? Faster than exponentially? How can you find out by computing and plotting? We will come across Hilbert matrices in the next lecture. □

[1] A. N. Langville and C. D. Meyer, Google's PageRank and Beyond, Princeton University Press, Princeton, NJ, 2006.