Another decomposition: QR

- Decompose matrix into product of orthogonal and upper triangular matrices.

$$A = Q\,R$$

- Upper triangular R:

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1\mathrm{N}} \\ 0 & r_{22} & r_{23} & \cdots & r_{2\mathrm{N}} \\ 0 & 0 & r_{33} & \cdots & r_{3\mathrm{N}} \\ & & \vdots & & \\ 0 & 0 & 0 & \cdots & r_{NN} \end{pmatrix}$$

- Orthogonal matrix Q spanning space of A.

We will use the QR decomposition later....

Orthogonal matrix

- Orthogonal matrix: columns are vectors orthogonal to each other.

$$Q = \begin{pmatrix} \vdots & \vdots & \vdots & & \vdots \\ e_1 & e_2 & e_3 & \cdots & e_N \\ \vdots & \vdots & \vdots & & \vdots \end{pmatrix}$$

- Nice property of orthogonal matrices:

$$Q^T Q = I \quad \Leftrightarrow \quad Q^{-1} = Q^T$$

- But how to create an orthogonal matrix?

Gram-Schmidt procedure

- Details in handout on Canvas
- Goal: Given matrix A, create set of orthonormal basis vectors e_i spanning space of A.
- The idea is to take each column of input matrix A, and shave off non-orthogonal components of each column.
- Picture on white board.

Gram-Schmidt procedure

Start with: $A=(a_1|a_2|a_3|\cdots|a_N)$

First vector: $u_1=a_1$ $\qquad e_1=\frac{u_1}{\|u_1\|}$

$$u_2=a_2-(a_2\cdot e_1)e_1 \qquad e_2=\frac{u_2}{\|u_2\|}$$

$$u_3=a_3-(a_3\cdot e_2)e_2-(a_3\cdot e_1)e_1 \qquad e_3=\frac{u_3}{\|u_3\|}$$

Gram-Schmidt code

```
function e = gram_schmidt(A)
  % This fcn returns the orthogonalization of the
  % matrix A computed using Gram-Schmidt

  N = size(A, 2);
  e = zeros(size(A));

  % Initialize computation by computing u and e for first col.
  un = A(:, 1);
  e(:, 1) = un/norm(un);

  % Iterate over remaining columns of A
  for k = 2:N
    an = A(:, k);
    un = an;
    % Iterate over previous e values and subtract off
    % component parallel to each ei
    for i = k-1:-1:1
      un = un - dot(an,e(:, i))*e(:, i);
    end
    % Compute next col of e
    e(:, k) = un/norm(un);

  end
end
```

QR algorithm – Gram-Schmidt

Start by considering A as collection of column vectors.

$$A = (a_1 | a_2 | a_3 | \cdots | a_N)$$

$$= (e_1 | e_2 | e_3 | \cdots | e_N) \begin{pmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_N \cdot e_1 \\ 0 & a_2 \cdot e_2 & a_2 \cdot e_2 & \cdots & a_N \cdot e_2 \\ 0 & 0 & a_3 \cdot e_3 & \cdots & a_N \cdot e_3 \\ & & \vdots & & \\ 0 & 0 & 0 & \cdots & a_N \cdot e_N \end{pmatrix}$$

Use Gram-Schmidt to compute orthonormal basis set e_i from a_i vectors. This forms orthogonal matrix Q.

Use e_i and a_i to compute elements of R matrix.

$$= QR$$

Code implementing QR decomposition

```
function [Q, R] = my_qr(A)
  % This fcn implements the QR algorithm using Gram-Schmidt

  e = gram_schmidt(A);

  N = size(A, 1);
  Q = e;
  R = zeros(size(A));
  for r = 1:N
    for c = r:N
      R(r, c) = dot(A(:, c), e(:, r));
    end
  end

end
```

Two steps:

1. Compute the e vectors from A.
2. Compute the R elements from e and A.

Remarks

- You can do a QR decomposition on a non-square matrix.
  - Important in linear regression.
- Gram-Schmidt is numerically unstable. Better methods use:
  - Householder transformation
  - Givens Rotations

Next big topic

Eigen-decompositions: finding eigenvalues and eigenvectors

Recall eigendecomposition

Decomposition:

$$A = Q \Lambda Q^{-1}$$

A is square, non-singular

where:

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ 0 & 0 & \lambda_3 & \cdots & 0 \\ & & \vdots & & \\ 0 & 0 & 0 & \cdots & \lambda_N \end{pmatrix}$$

Diagonal matrix of eigenvalues

$$Q = \begin{pmatrix} \vdots & \vdots & \vdots & & \vdots \\ e_1 & e_2 & e_3 & \cdots & e_N \\ \vdots & \vdots & \vdots & & \vdots \end{pmatrix}$$

Matrix of orthogonal column vectors e_i (eigenvectors)

Some things to remember about eigendecomposition

$$A = Q \Lambda Q^{-1}$$

SVD handles both square and rectangular matrices

- Square matrix only.
- In general, eigenvalues are complex.
- For real matrix, eigenvalues are either real, or come in complex conjugate pairs.
- For real symmetric matrix, eigenvalues are real.
- For real SPD matrix, eigenvalues are positive.

The Matrix Zoo

2.15.2016 -- SDB

Real, square matrices

Matrix type	Common symbols	Properties	Comment
Arbitrary matrix with real elements	A	• Eigenvalues are complex	Visualize as stretching & rotating unit circle into ellipse
Symmetric	A	• $A = A^T$ • Eigenvalues are real • Eigenvectors are orthogonal	
Antisymmetric	A	• $A = -A^T$ • Eigenvalues are imaginary • Eigenvectors are orthogonal	
Symmetric, positive definite	A	• $A = A^T$ • Eigenvalues are real, positive • Eigenvectors are orthogonal • $x^T A x \geq 0$ for all vectors x	Visualize using upward parabolas
Symmetric, negative definite	A	• $A = A^T$ • Eigenvalues are real, negative • Eigenvectors are orthogonal • $x^T A x \leq 0$ for all vectors x	Visualize using upward parabolas
Orthogonal	Q	• Eigenvectors are orthonormal • Eigenvectors $\lvert\lambda_i\rvert = 1$ (lie on unit circle) • Q3 = Q1*Q2 (Orthogonality persists) • $Q^T = Q^{-1}$	Visualize as producing generalized rotations and reflections of vectors in N-dimensional space.

What about the SVD?

- Recall the SVD is like a generalization of the eigenvalue decomposition (EVD).
- I will give you some algorithms this evening which calculate the EVD.
- I will not give any SVD algorithms in this class.
  - They are similar to the EVD algos, but more complicated.
  - The important thing to know about the SVD is how to use it, not how to compute it.
  - Matlab gives you the SVD as a built-in.

How to compute eigendecomposition?

- What you learned as an undergrad: Eigenvalues are roots of characteristic polynomial:

$$det(A-\lambda I)=0$$

This gives polynomial in λ for which you find the roots.

- Computing determinant using method you learned in school is O(N!) -- don't do it!
  - Also involves solving Nth degree polynomial.
- In numerical analysis: Compute eigenvalues using iterative methods.

First method: Power iteration

- Simplest method to compute one eigenvalue and associated eigenvector.
- Computes “dominant” eigenvalue – i.e. largest eigenvalue.
- Method:
  1. Compute eigenvector using iteration:

$$b_{n+1} = \frac{A\,b_n}{\|A\,b_n\|}$$

Question: What is time-complexity of this method?

  2. After iteration, b_n converges to eigenvector.
  3. Once eigenvector is found, compute eigenvalue:

$$\lambda = \frac{b_n^T A\,b_n}{b_n^T b_n}$$

Rayleigh quotient

Power iteration code

```
function [lam, b] = my_eig_power(A)

  % Conversion tolerance
  tol = 1e-6;

  % Initial guess vector
  bn = randn(size(A, 1), 1);
  for cnt = 1:100
    t = A*bn;
    bnp1 = t/norm(t);
    diff = norm(bnp1-bn);
    if (diff < tol)
      lam = (bnp1'*A*bnp1)/(bnp1'*bnp1);
      b = bnp1;
      return
    else
      bn = bnp1;
    end
  end

  % If we get here it is because we didn't converge.
  error('Did not converge in my_eig_power!')
end
```

$$b_{n+1} = \frac{A\,b_n}{\|A\,b_n\|}$$

$$\lambda = \frac{b_n^T A\,b_n}{b_n^T b_n}$$

Test run

```
*********   Testing [3, 3] matrix   ************
eigenvalue after iteration 1 = 7.507233e+02
eigenvalue after iteration 2 = 7.805656e+02
eigenvalue after iteration 3 = 7.832009e+02
eigenvalue after iteration 4 = 7.834320e+02
eigenvalue after iteration 5 = 7.834522e+02
eigenvalue after iteration 6 = 7.834540e+02
eigenvalue after iteration 7 = 7.834542e+02
eigenvalue after iteration 8 = 7.834542e+02
eigenvalue after iteration 9 = 7.834542e+02
Matlab eigenvalue = 783.454194, my eigenvalue = 783.454194
       diff = 1.148195e-07
Pass!
Checking eigenvector, norm(diff) = 1.442874e-05
Pass!
```

Power iteration -- magic!

$$b_{n+1} = \frac{A b_n}{\|A b_n\|}$$

Converges to eigenvector after some number of iterations

- Why does this work?
- Ignore scalar denominator for now. The iteration is:

$$b_1 = A b_0$$

$$b_2 = A(A b_0)$$

$$b_3 = A(A(A b_0))$$

$$\vdots$$

$$b_n = A^n b_0$$

Consider eigenvalue decomposition of A

- Eigenvalue decomposition:

$$A = Q \Lambda Q^{-1}$$

- Therefore for any integer power of A,

$$\begin{aligned} A^n &= (Q \Lambda Q^{-1})^n \\ &= (Q \Lambda Q^{-1})(Q \Lambda Q^{-1})(Q \Lambda Q^{-1}) \cdots \\ &= (Q \Lambda \Lambda \Lambda \cdots \Lambda Q^{-1}) \\ &= Q \Lambda^n Q^{-1} \end{aligned}$$

This matrix is diagonal, so simply compute scalar power.

Why does power iteration work?

- Now consider writing vector b_0 as sum of eigenvectors of A (assume rank k):

$$\vec{b}_0 = \beta_1 \vec{q}_1 + \beta_2 \vec{q}_2 + \beta_3 \vec{q}_3 + \cdots + \beta_k \vec{q}_k$$

q forms linearly independent basis set, so this is legal.

- Multiply by A:

$$A\vec{b}_0 = \beta_1 A\vec{q}_1 + \beta_2 A\vec{q}_2 + \beta_3 A\vec{q}_3 + \cdots + \beta_k A\vec{q}_k$$

$$= \beta_1 \lambda_1 \vec{q}_1 + \beta_2 \lambda_2 \vec{q}_2 + \beta_3 \lambda_3 \vec{q}_3 + \cdots + \beta_k \lambda_k \vec{q}_k$$

- Do it again:

$$A(A\vec{b}_0) = \beta_1 \lambda_1 A\vec{q}_1 + \beta_2 \lambda_2 A\vec{q}_2 + \beta_3 \lambda_3 A\vec{q}_3 + \cdots + \beta_k \lambda_4 A\vec{q}_k$$

$$= \beta_1 \lambda_1^2 \vec{q}_1 + \beta_2 \lambda_2^2 \vec{q}_2 + \beta_3 \lambda_3^2 \vec{q}_3 + \cdots + \beta_k \lambda_4^2 \vec{q}_k$$

Why does power iteration work?

- Now consider writing vector b_0 as sum of eigenvectors of A (assume rank k):

$$\vec{b}_0 = \beta_1 \vec{q}_1 + \beta_2 \vec{q}_2 + \beta_3 \vec{q}_3 + \cdots + \beta_k \vec{q}_k$$

q forms linearly independent basis set, so this is legal.

- In general,

Assume eigenvalues are sorted in reverse (largest first) order

$$A^n \vec{b}_0 = \beta_1 \lambda_1^n \vec{q}_1 + \beta_2 \lambda_2^n \vec{q}_2 + \beta_3 \lambda_3^n \vec{q}_3 + \cdots + \beta_k \lambda_k^n \vec{q}_k$$

- Assume eigenvalues are sorted, we can write:

$$A^n \vec{b}_0 = \lambda_1^n \left(\beta_1 \vec{q}_1 + \beta_2 \left(\frac{\lambda_2}{\lambda_1} \right)^n \vec{q}_2 + \beta_3 \left(\frac{\lambda_3}{\lambda_1} \right)^n \vec{q}_3 + \cdots + \beta_k \left(\frac{\lambda_k}{\lambda_1} \right)^n \vec{q}_k \right)$$

Why does power iteration work?

- Eigenvalues sorted from largest to smallest in sum:

$$A^n\vec{b}=\lambda_1^n\left(\beta_1\vec{q}_1+\beta_2\cancelto{0}{\left(\frac{\lambda_2}{\lambda_1}\right)^n}\vec{q}_2+\beta_3\cancelto{0}{\left(\frac{\lambda_3}{\lambda_1}\right)^n}\vec{q}_3+\cdots+\beta_k\cancelto{0}{\left(\frac{\lambda_k}{\lambda_1}\right)^n}\vec{q}_k\right)$$

- Observe that all terms tend to zero as n tends to infinity, except for first term. Therefore,

$$\lim_{n\to\infty} A^n\vec{b}\approx\lambda_1^n\beta_1\vec{q}_1$$

- Ignore the scalar multipliers. This vector points in the same direction as $\vec{q}_1$. Therefore, power iteration converges to this eigenvector.

Convergence rate

- If eigenvalues are sorted from largest to smallest in sum:

$$A^n \vec{b} = \lambda_1^n \left(\beta_1 \vec{q}_1 + \beta_2 \left(\frac{\lambda_2}{\lambda_1} \right)^n \vec{q}_2 + \beta_3 \left(\frac{\lambda_3}{\lambda_1} \right)^n \vec{q}_3 + \cdots + \beta_k \left(\frac{\lambda_k}{\lambda_1} \right)^n \vec{q}_k \right)$$

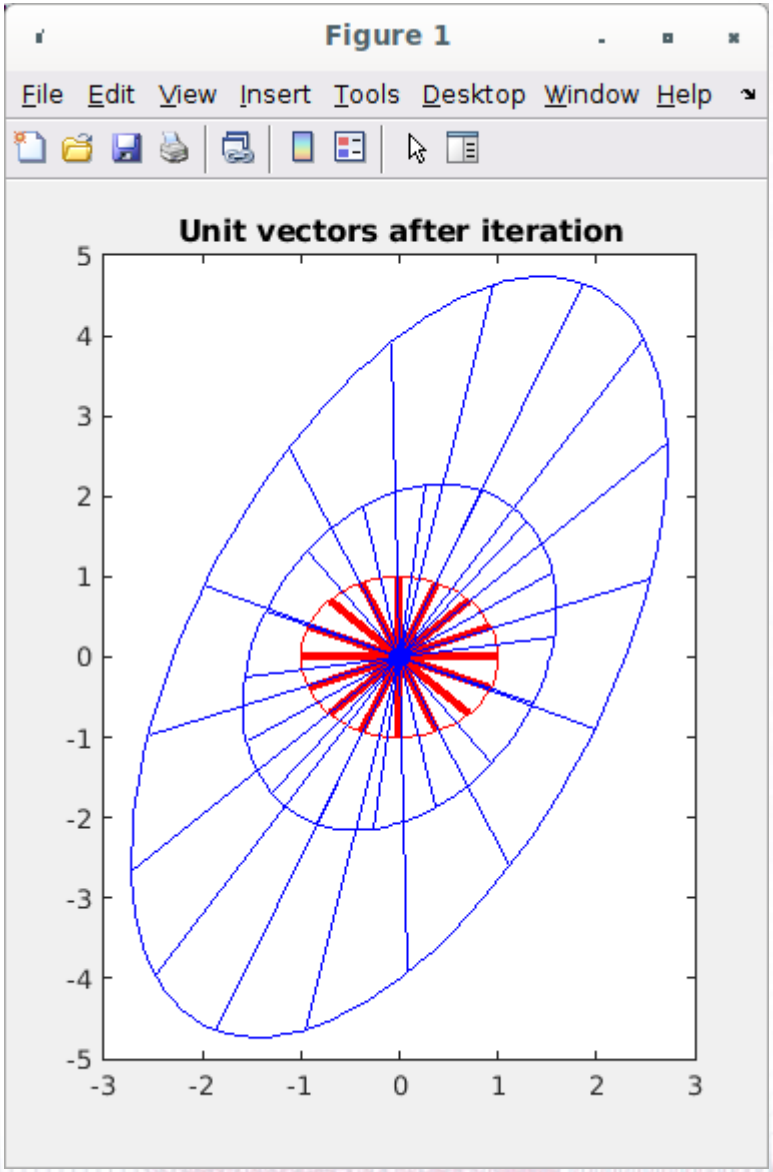
Slowest decreasing is second term.

Fastest decreasing is last term.

- Slowest decreasing term is λ_2/λ_1 because it is closest to 1.
- Therefore, rate convergence is dominated by this term.

Recall ellipse visualization

- Start with all vectors in unit ball, x. (Red)
- Compute Ax to get first ellipse. (Blue)
- Compute A(Ax) to get second ellipse (Blue).
- Note vectors are starting to cluster around the long ends of the ellipses.



Class6/SimultaneousIterationArrows

Aside: Compute eigenvalue from eigenvector

- We just showed: Power iteration converges to eigenvector q_1:

$$b_n = \frac{A\,b_{n-1}}{\|A\,b_{n-1}\|} \;\rightarrow\; q_1 \text{ for } n \rightarrow \infty$$

- Once we have eigenvector $b_n = q_1$, we can compute eigenvalue using Rayleigh quotient:

$$\lambda = \frac{b_n^T A\,b_n}{b_n^T b_n}$$

Recall $A\,b_n = \lambda\,b_n$

$$\lambda = \frac{q_1^T \lambda_1 q_1}{q_1^T q_1} = \lambda_1$$

Remarks on power iteration

- Only computes dominant eigenvector/value pair (i.e. only one eigenvalue – the largest).
- Not typically used in real world since better algorithms exist to compute all eigenvalues.
- However, underlying concepts are very important, and appear again and again.....
- Also, Google page rank algorithm is very similar.

Next: Inverse iteration

- Power iteration returns dominant eigenvector/value pair.
- Can we get other eigenvectors/values?
- Yes – simplest method is inverse iteration
- Suppose we have a guess for an eigenvalue of matrix A, λ_i ← Actual, unknown eigenvalue

  Close guess
- Consider λ_{guess} close to λ_i. Then the matrix $(A-\lambda_{guess} I)^{-1}$ has very large eigenvalue (dominant).
- Therefore, we can find λ_i via power iteration using that matrix.

Inverse iteration algorithm

1. Start with random vector b_0 and initial guess for eigenvalue $\mu = \lambda_{guess}$

2. Compute iteration

$$b_{n+1} = \frac{(A - \mu I)^{-1} b_n}{\|(A - \mu I)^{-1} b_n\|}$$

Converges to eigenvector corresponding to largest eigenvalue of numerator's matrix.

3. Check for convergence:

$$\|b_{n+1} - b_n\| < tol$$

4. If converged, compute eigenvalue and return eigenvector b_n and computed eigenvalue λ

$$\lambda = \frac{b^T A b}{b^T b}$$

```
function [lnp1, qnp1] = myeig_inverse(l0, A)

  % Max number of iterations
  M = 2500;

  % Convergence tolerance.
  tol = 1e-6;

  % Convenience matrix
  I = eye(size(A));

  % Choose random start vector
  qn = randn(size(A,1),1);

  % Iterate
  for idx = 1:M
    zn = (A - l0*I)\qn;
    qnp1 = zn/norm(zn);

    % Check for convergence.
    diff = norm(qnp1 - qn);
    if (diff < tol)
      fprintf('---  We converged after %d iterations! \n', idx)
      lnp1 = (qnp1'*A*qnp1)/(qnp1'*qnp1);
      return
    end
    % Did not converge yet.  Do next iteration.
    qn = qnp1;
  end
  error('Error in myeig_inverse -- Did not converge in M iterations!')
end
```

Use linear solve instead of matrix inverse.

Compute eigenvalue using Rayleigh quotient

```
>> A

A =

   -0.3482   -1.7585    0.4918    0.9279   -0.9205
   -1.7585    2.1446   -0.6291   -0.2147    1.1180
    0.4918   -0.6291   -1.4778   -0.5470    1.4575
    0.9279   -0.2147   -0.5470   -0.0994   -1.7822
   -0.9205    1.1180    1.4575   -1.7822   -2.3920

>> eigs(A)

ans =

   -4.1202
    3.7274
   -1.9773
    1.2180
   -1.0207

>> myeig_inverse(3.5, A)
---  We converged after 8 iterations!  Returning eigenvalue & eigenvector.....

ans =

    3.7274

>> myeig_inverse(-2, A)
---  We converged after 6 iterations!  Returning eigenvalue & eigenvector.....

ans =

   -1.9773
```

Next: Rayleigh quotient iteration

- Computes one eigenvector and eigenvalue, like power iteration and inverse iteration.
- Basic idea: Combine inverse iteration with updated approximation to eigenvalue.

$$z_n = (A - \mu I)^{-1} q_n$$

Inverse iteration: Keep initial guess constant for all iterations.

$$z_n = (A - \lambda_n I)^{-1} q_n$$

Rayleigh quotient iteration: Update estimate of eigenvalue at each iteration.

- Unlike power iteration, this algorithm is very fast – cubic convergence.

Concept: Rayleigh quotient and approximate eigenvalue

- Rayleigh quotient is function of input vector *u*

$$r(u)=\frac{u^T A u}{u^T u}$$

- Note that when *u* is exactly an eigenvector of A,

$$r(u)=\frac{u^T A u}{u^T u}=\frac{u^T \lambda u}{u^T u}=\lambda$$

- When u is close to an eigenvector, the Rayleigh quotient returns an “approximate eigenvalue”.

Rayleigh quotient algorithm

1. Start with random start vector q_n
2. Compute approximate eigenvalue

$$\lambda_n = \frac{q_n^T A q_n}{q_n^T q_n}$$

3. Start loop:
4. Compute: $z_n = (A - \lambda_n I) \backslash q_n$
5. Compute new q_n vector: $q_n = z_n / \|z_n\|$

Inverse iteration step

Normalization step

6. Compute new eigenvalue:

$$\lambda_n = \frac{q_n^T A q_n}{q_n^T q_n}$$

Note that since q is already normalized I actually don't need this denominator.

7. Check for convergence. If diff(old eigenvalue – new eigenvalue) is small, then you have converged. In this case, return new eigenvector and eigenvalue.

8. Otherwise, loop back to 3.

```
**********   Testing [17, 17] matrix   ************
eigenvalue after iteration 1 = -2.323507e+00
eigenvalue after iteration 2 = -2.715766e+00
eigenvalue after iteration 3 = -2.934998e+00
eigenvalue after iteration 4 = -2.947358e+00
eigenvalue after iteration 5 = -2.947359e+00
eigenvalue after iteration 6 = -2.947359e+00
---  We converged!  Returning eigenvalue & eigenvector.....
my eigenvalue = -2.947359e+00, Matlab eigenvalue = -2.947359e+00 ...
Pass!
Checking eigenvector, norm(diff) = 6.736208e-15 ...Pass!
**********   Testing [24, 24] matrix   ************
eigenvalue after iteration 1 = 5.724041e+01
eigenvalue after iteration 2 = 5.648475e+01
eigenvalue after iteration 3 = 5.667111e+01
eigenvalue after iteration 4 = 5.667123e+01
eigenvalue after iteration 5 = 5.667123e+01
---  We converged!  Returning eigenvalue & eigenvector.....
my eigenvalue = 5.667123e+01, Matlab eigenvalue = 5.667123e+01 ...
Pass!
Checking eigenvector, norm(diff) = 1.062175e-15 ...Pass!
```

Note fast convergence for larger matrices

Remarks

- Power method
  - Simple, easy to understand.
  - Convergence is slow.
- Inverse iteration
  - Use to choose which eigenvalue to converge to.
  - Assumes a priori knowledge of your eigenvalues (or just get min eigenvalue).
- Rayleigh iteration
  - Fast convergence.
  - Should use with a priori knowledge of your desired eigenvalue (or just get min one).

All methods return only one eigenvalue/vector

Next: How to compute *all* eigenvalues of matrix A?

- Power iteration gives us “dominant” (i.e. largest) eigenvalue and associated eigenvector.
- Inverse iteration and Rayleigh quotient give us eigenvalue “close” to initial guess.
- Can we find an iterative method to give us all eigenvalues at once?
- Yes:
  - Simultaneous iteration
  - QR algorithm
  - Lanczos iteration – not covered
  - Arnoldi iteration – not covered

Simultaneous iteration

- Idea: Use power method on matrix made of random column vectors.

$$V = \begin{pmatrix} \vdots & \vdots & \vdots & \\ v_1 & v_2 & v_3 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix}$$

Place random column vectors into matrix V

$$A^n V = \begin{pmatrix} \vdots & \vdots & \vdots & \\ A^n v_1 & A^n v_2 & A^n v_3 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix}$$

Without orthogonalization, each column would converge to dominant eigenvector

- Idea: Orthogonalize the columns after every multiplication so each column converges to a different eigenvector.

Simultaneous iteration idea

- Idea: Orthogonalize the columns after every multiplication.

  Use Gram-Schmidt

- Each column converges to a different eigenvector.

$$V_{new} = A\,Q = \begin{pmatrix} \vdots & \vdots & \vdots & \\ A\,q_1 & A\,q_2 & A\,q_3 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix}$$

Converges to $\hat{e}_{1,}\lambda_1$

Converges to $\hat{e}_{2,}\lambda_2$

Converges to $\hat{e}_{3,}\lambda_3$

Algorithm

1. Create random start matrix

$$V = \begin{pmatrix} \vdots & \vdots & \vdots & \\ v_1 & v_2 & v_3 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix}$$

2. Orthogonalize

$$Q = \text{gram\_schmidt}(V)$$

3. Multiply Q through by A, giving new V

$$V_{new} = AQ = \begin{pmatrix} \vdots & \vdots & \vdots & \\ Aq_1 & Aq_2 & Aq_3 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix}$$

4. Check for convergence: Is the difference between old and new V small? If yes, V contains eigenvectors on columns. Exit loop. If no, loop back to 2.

At end, to get eigenvalues, use Rayleigh quotient on q vectors:

$$\lambda_i = \frac{q_i^T A q_i}{q_i^T q_i}$$

```
for idx = 1:M
  % First multiply by A
  U = A*Vn;

  % Now re-orthogonalize columns
  Vnp1 = gram_schmidt(U);
  fprintf('V matrix after iteration %d = \n', idx)
  disp(Vnp1)
  pause

  % Check for convergence.
  diff = norm(Vnp1 - Vn);
  if (diff < tol)
    fprintf('---  We converged after %d iterations!  ---\n', idx)
    % We have converged.  Each column holds an eigenvector.
    % Compute eigenvalues for each eigenvector using Rayleigh
    % quotient.
    N = size(Vnp1, 1);
    l = zeros(N, 1);
    for col = 1:N
      u = Vnp1(:,col);
      l(col) = (u'*A*u)/(u'*u);
    end
    return
  end
  % Did not converge yet.  Do next iteration.
  Vn = Vnp1;
end
```

Multiply

Orthogonalize

Demo

```
>> run_my_eig_si

N =

     5

*********   Testing [5, 5] matrix   ************
Input matrix =
    3.0076   -0.9029    0.1893   -1.8502    0.3559
   -0.9029    2.4467    2.8748   -1.3666    0.1417
    0.1893    2.8748    1.7990   -0.9225    1.3617
   -1.8502   -1.3666   -0.9225   -0.9389    1.2099
    0.3559    0.1417    1.3617    1.2099    0.7376

V matrix after iteration 1 =
    0.9001    0.2974    0.2868    0.1140    0.0783
    0.0817   -0.7748    0.4273   -0.0137    0.4586
    0.2178   -0.4385    0.0720   -0.1711   -0.8519
    0.3561   -0.3313   -0.8527    0.0784    0.1738
   -0.0945   -0.0959    0.0537    0.9754   -0.1661
```

Convergence of simultaneous iteration

- Basic idea: power iteration in each column.

$$V_{new} = A\,Q = \begin{pmatrix} \vdots & \vdots & \vdots & \\ A\,q_1 & A\,q_2 & A\,q_3 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix}$$

Slowest decreasing term

- Power iteration converged like λ_2/λ_1
- However, in simultaneous iteration, each col is forced orthogonal to previous columns...

$$Q = \begin{pmatrix} \vdots & \vdots & \vdots & \\ q_1 & q_2 & q_3 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix}$$

Ortho. to q1, q2

Ortho. to q1

Convergence...

- Due to orthogonalization, each column converges to next highest eigenvalue.

Contains no components in q1 direction.

$$V_{new} = AQ = \begin{pmatrix} \vdots & \vdots & \vdots & \\ Aq_1 & Aq_2 & Aq_3 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix}$$

Contains no components in q1 or q2 directions.

Converges to λ_3

Converges to λ_1

Converges to λ_2

- Convergence rate of column j dominated by λ_{j+1}/λ_j
- Therefore, global convergence is governed by $max\left(\lambda_{j+1}/\lambda_j\right)$

Remember this for later....

Topics covered in this session

- Google page rank algorithm
- Power method
- Inverse iteration
- Rayleigh iteration
- Simultaneous iteration

Computing eigendecompositions using iterative methods!