LU Factorization

Solving $Ax = b$

Familiar idea for solving $Ax = b$ is to use Gaussian elimination to transform $Ax = b$ to a triangular system

What is a triangular system?

- Upper triangular matrix $U \in \mathbb{R}^{n \times n}$: if $i > j$ then $u_{ij} = 0$
- Lower triangular matrix $L \in \mathbb{R}^{n \times n}$: if $i < j$ then $\ell_{ij} = 0$

Question: Why is triangular good?

Answer: Because triangular systems are easy to solve!

Solving $Ax = b$

Suppose we have $Ux = b$, then we can use "back-substitution"

$$\begin{aligned}
x_n &= b_n/u_{nn} \\
x_{n-1} &= (b_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \\
&\vdots \\
x_j &= \left(b_j - \sum_{k=j+1}^{n} u_{jk}x_k\right)/u_{jj} \\
&\vdots
\end{aligned}$$

Solving $Ax = b$

Similarly, we can use forward substitution for a lower triangular system $Lx = b$

$$\begin{aligned}
x_1 &= b_1/\ell_{11} \\
x_2 &= (b_2 - \ell_{21}x_1)/\ell_{22} \\
&\vdots \\
x_j &= \left(b_j - \sum_{k=1}^{j-1} \ell_{jk}x_k\right)/\ell_{jj} \\
&\vdots
\end{aligned}$$

Solving $Ax = b$

Back and forward substitution can be implemented with doubly nested for-loops

The computational work is dominated by evaluating the sum $\sum_{k=1}^{j-1} \ell_{jk} x_k$, $j = 1, \ldots, n$

We have $j - 1$ additions and multiplications in this loop for each $j = 1, \ldots, n$, *i.e.* $2(j - 1)$ operations for each j

Hence the total number of floating point operations in back or forward substitution is asymptotic to:

$$2 \sum_{j=1}^{n} j = 2n(n+1)/2 \sim n^2$$

Solving $Ax = b$

Here "$\sim$" refers to asymptotic behavior, *e.g.*

$$f(n) \sim n^2 \iff \lim_{n\to\infty} \frac{f(n)}{n^2} = 1$$

We often also use "big-O" notation, *e.g.* for remainder terms in Taylor expansion

$f(x) = O(g(x))$ if there exists $M \in \mathbb{R}_{>0}, x_0 \in \mathbb{R}$ such that $|f(x)| \leq M|g(x)|$ for all $x \geq x_0$

In the present context we prefer "$\sim$" since it indicates the correct scaling of the leading-order term

e.g. let $f(n) \equiv n^2/4 + n$, then $f(n) = O(n^2)$, whereas $f(n) \sim n^2/4$

Solving $Ax = b$

So transforming $Ax = b$ to a triangular system is a sensible goal, but how do we achieve it?

Observation: If we premultiply $Ax = b$ by a nonsingular matrix M then the new system $MAx = Mb$ has the same solution

Hence, want to devise a sequence of matrices $M_1, M_2, \cdots, M_{n-1}$ such that $MA \equiv M_{n-1} \cdots M_1 A \equiv U$ is upper triangular

This process is Gaussian Elimination, and gives the transformed system $Ux = Mb$

LU Factorization

We will show shortly that it turns out that if $MA = U$, then we have that $L \equiv M^{-1}$ is lower triangular

Therefore we obtain $A = LU$: product of lower and upper triangular matrices

This is the LU factorization of A

LU Factorization

LU factorization is the most common way of solving linear systems!

$Ax = b \Longleftrightarrow LUx = b$

Let $y \equiv Ux$, then $Ly = b$: solve for y via forward substitution[3]

Then solve for $Ux = y$ via back substitution

[3] $y = L^{-1}b$ is the transformed right-hand side vector (*i.e.* Mb from earlier) that we are familiar with from Gaussian elimination

LU Factorization

LU factorization is the most common way of solving linear systems!

$Ax = b \Longleftrightarrow LUx = b$

Let $y \equiv Ux$, then $Ly = b$: solve for y via forward substitution[1]

Then solve for $Ux = y$ via back substitution

[1] $y = L^{-1}b$ is the transformed right-hand side vector (*i.e.* Mb from earlier) that we are familiar with from Gaussian elimination

LU Factorization

Next question: How should we determine $M_1, M_2, \cdots, M_{n-1}$?

We need to be able to annihilate selected entries of A, below the diagonal in order to obtain an upper-triangular matrix

To do this, we use "elementary elimination matrices"

Let L_j denote j^{th} elimination matrix (we use "L_j" rather than "M_j" from now on as elimination matrices are lower triangular)

LU Factorization

Let $X(\equiv L_{j-1}L_{j-2}\cdots L_1A)$ denote matrix at the start of step j, and let $x_{(:,j)} \in \mathbb{R}^n$ denote column j of X

Then we define L_j such that

$$L_j x_{(:,j)} \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -x_{j+1,j}/x_{jj} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -x_{nj}/x_{jj} & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_{1j} \\ \vdots \\ x_{jj} \\ x_{j+1,j} \\ \vdots \\ x_{nj} \end{bmatrix} = \begin{bmatrix} x_{1j} \\ \vdots \\ x_{jj} \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

LU Factorization

To simplify notation, we let $\ell_{ij} \equiv \frac{x_{ij}}{x_{jj}}$ in order to obtain

$$L_j \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\ell_{j+1,j} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\ell_{nj} & 0 & \cdots & 1 \end{bmatrix}$$

LU Factorization

Using elementary elimination matrices we can reduce A to upper triangular form, one column at a time

Schematically, for a 4×4 matrix, we have

$$\underset{A}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}} \xrightarrow{L_1} \underset{L_1 A}{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix}} \xrightarrow{L_2} \underset{L_2 L_1 A}{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}}$$

Key point: L_k does not affect columns $1, 2, \ldots, k-1$ of $L_{k-1} L_{k-2} \ldots L_1 A$

LU Factorization

After $n-1$ steps, we obtain the upper triangular matrix $U = L_{n-1} \cdots L_2 L_1 A$

$$U = \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

LU Factorization

Finally, we wish to form the factorization $A = LU$, hence we need $L = (L_{n-1} \cdots L_2 L_1)^{-1} = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$

This turns out to be surprisingly simple due to two strokes of luck!

First stroke of luck: L_j^{-1} is obtained simply by negating the subdiagonal entries of L_j

$$L_j \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\ell_{j+1,j} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\ell_{nj} & 0 & \cdots & 1 \end{bmatrix}, \quad L_j^{-1} \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & \ell_{j+1,j} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \ell_{nj} & 0 & \cdots & 1 \end{bmatrix}$$

LU Factorization

Explanation: Let $\ell_j \equiv [0, \ldots, 0, \ell_{j+1,j}, \ldots, \ell_{nj}]^T$ so that $L_j = \mathrm{I} - \ell_j e_j^T$

Now consider $L_j(\mathrm{I} + \ell_j e_j^T)$:

$$L_j(\mathrm{I}+\ell_j e_j^T) = (\mathrm{I}-\ell_j e_j^T)(\mathrm{I}+\ell_j e_j^T) = \mathrm{I}-\ell_j e_j^T \ell_j e_j^T = \mathrm{I}-\ell_j(e_j^T \ell_j)e_j^T$$

Also, $(e_j^T \ell_j) = 0$ (why?) so that $L_j(\mathrm{I} + \ell_j e_j^T) = \mathrm{I}$

By the same argument $(\mathrm{I} + \ell_j e_j^T)L_j = \mathrm{I}$, and hence $L_j^{-1} = (\mathrm{I} + \ell_j e_j^T)$

LU Factorization

Next we want to form the matrix $L \equiv L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$

Note that we have

$$\begin{aligned} L_j^{-1} L_{j+1}^{-1} &= (\mathrm{I} + \ell_j e_j^T)(\mathrm{I} + \ell_{j+1} e_{j+1}^T) \\ &= \mathrm{I} + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T + \ell_j (e_j^T \ell_{j+1}) e_{j+1}^T \\ &= \mathrm{I} + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T \end{aligned}$$

Interestingly, this convenient result doesn't hold for $L_{j+1}^{-1} L_j^{-1}$, why?

LU Factorization

Similarly,

$$\begin{aligned} L_j^{-1}L_{j+1}^{-1}L_{j+2}^{-1} &= (\mathrm{I} + \ell_j e_j^T + \ell_{j+1}e_{j+1}^T)(\mathrm{I} + \ell_{j+2}e_{j+2}^T) \\ &= \mathrm{I} + \ell_j e_j^T + \ell_{j+1}e_{j+1}^T + \ell_{j+2}e_{j+2}^T \end{aligned}$$

That is, to compute the product $L_1^{-1}L_2^{-1}\cdots L_{n-1}^{-1}$ we simply collect the subdiagonals for $j = 1, 2, \ldots, n-1$

LU Factorization

Hence, second stroke of luck:

$$L \equiv L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1} = \begin{bmatrix} 1 & & & & \\ \ell_{21} & 1 & & & \\ \ell_{31} & \ell_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ \ell_{n1} & \ell_{n2} & \cdots & \ell_{n,n-1} & 1 \end{bmatrix}$$

LU Factorization

Therefore, basic LU factorization algorithm is

```
1: $U = A$, $L = \mathrm{I}$
2: for $j = 1 : n - 1$ do
3:     for $i = j + 1 : n$ do
4:         $\ell_{ij} = u_{ij} / u_{jj}$
5:         for $k = j : n$ do
6:             $u_{ik} = u_{ik} - \ell_{ij} u_{jk}$
7:         end for
8:     end for
9: end for
```

Note that the entries of U are updated each iteration so at the start of step j, $U = L_{j-1} L_{j-2} \cdots L_1 A$

Here line 4 comes straight from the definition $\ell_{ij} \equiv \frac{u_{ij}}{u_{jj}}$

LU Factorization

Line 6 accounts for the effect of L_j on columns $k = j, j+1, \ldots, n$ of U

For $k = j : n$ we have

$$L_j u_{(:,k)} \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\ell_{j+1,j} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\ell_{nj} & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{1k} \\ \vdots \\ u_{jk} \\ u_{j+1,k} \\ \vdots \\ u_{nk} \end{bmatrix} = \begin{bmatrix} u_{1k} \\ \vdots \\ u_{jk} \\ u_{j+1,k} - \ell_{j+1,j} u_{jk} \\ \vdots \\ u_{nk} - \ell_{nj} u_{jk} \end{bmatrix}$$

The vector on the right is the updated k^{th} column of U, which is computed in line 6

LU Factorization

LU Factorization involves a triply-nested for-loop, hence $O(n^3)$ calculations

Careful operation counting shows LU factorization requires $\sim \frac{1}{3}n^3$ additions and $\sim \frac{1}{3}n^3$ multiplications, $\sim \frac{2}{3}n^3$ operations in total

Solving a linear system using LU

Hence to solve $Ax = b$, we perform the following three steps:

Step 1: Factorize A into L and U: $\sim \frac{2}{3}n^3$

Step 2: Solve $Ly = b$ by forward substitution: $\sim n^2$

Step 3: Solve $Ux = y$ by back substitution: $\sim n^2$

Total work is dominated by Step 1, $\sim \frac{2}{3}n^3$

Solving a linear system using LU

An alternative approach would be to compute A^{-1} explicitly and evaluate $x = A^{-1}b$, but this is a bad idea!

Question: How would we compute A^{-1}?

Solving a linear system using LU

Answer: Let $a^{\text{inv}}_{(:,k)}$ denote the kth column of A^{-1}, then $a^{\text{inv}}_{(:,k)}$ must satisfy

$$Aa^{\text{inv}}_{(:,k)} = e_k$$

Therefore to compute A^{-1}, we first LU factorize A, then back/forward substitute for rhs vector e_k, $k = 1, 2, \ldots, n$

The n back/forward substitutions alone require $\sim 2n^3$ operations, inefficient!

A rule of thumb in Numerical Linear Algebra: It is almost always a bad idea to compute A^{-1} explicitly

Solving a linear system using LU

Another case where LU factorization is very helpful is if we want to solve $Ax = b_i$ for several different right-hand sides b_i, $i = 1, \ldots, k$

We incur the $\sim \frac{2}{3}n^3$ cost only once, and then each subequent forward/back subsitution costs only $\sim 2n^2$

Makes a huge difference if n is large!

Stability of Gaussian Elimination

There is a problem with the LU algorithm presented above

Consider the matrix

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

A is nonsingular, well-conditioned ($\kappa(A) \approx 2.62$) but LU factorization fails at first step (division by zero)

Stability of Gaussian Elimination

LU factorization doesn't fail for

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}$$

but we get

$$L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}$$

Stability of Gaussian Elimination

Let's suppose that $-10^{20} \in \mathbb{F}$ (a floating point number) and that round$(1 - 10^{20}) = -10^{20}$

Then in finite precision arithmetic we get

$$\widetilde{L} = \left[\begin{array}{cc} 1 & 0 \\ 10^{20} & 1 \end{array} \right], \qquad \widetilde{U} = \left[\begin{array}{cc} 10^{-20} & 1 \\ 0 & -10^{20} \end{array} \right]$$

Stability of Gaussian Elimination

Hence due to rounding error we obtain

$$\widetilde{L}\widetilde{U} = \left[\begin{array}{cc} 10^{-20} & 1 \\ 1 & 0 \end{array}\right]$$

which is not close to

$$A = \left[\begin{array}{cc} 10^{-20} & 1 \\ 1 & 1 \end{array}\right]$$

Then, for example, let $b = [3, 3]^T$

- Using $\widetilde{L}\widetilde{U}$, we get $\tilde{x} = [3, 3]^T$
- True answer is $x = [0, 3]^T$

Hence large relative error (rel. err. $= 1$) even though the problem is well-conditioned

Stability of Gaussian Elimination

In this example, standard Gaussian elimination yields a large residual

Or equivalently, it yields the exact solution to a problem corresponding to a large input perturbation: $\Delta b = [0, 3]^T$

Hence unstable algorithm! In this case the cause of the large error in x is numerical instability, not ill-conditioning

To stabilize Gaussian elimination, we need to permute rows, *i.e.* perform pivoting

Pivoting

Recall the Gaussian elimination process

$$\begin{bmatrix} \times & \times & \times & \times \\ & x_{jj} & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ & x_{jj} & \times & \times \\ & 0 & \times & \times \\ & 0 & \times & \times \end{bmatrix}$$

But we could just as easily do

$$\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & x_{ij} & \times & \times \\ & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ & 0 & \times & \times \\ & x_{ij} & \times & \times \\ & 0 & \times & \times \end{bmatrix}$$

Partial Pivoting

The entry x_{ij} is called the pivot, and flexibility in choosing the pivot is essential otherwise we can't deal with:

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

From a numerical stability point of view, it is crucial to choose the pivot to be the largest entry in column j: "partial pivoting"[2]

This ensures that each ℓ_{ij} entry — which acts as a multiplier in the LU factorization process — satisfies $|\ell_{ij}| \leq 1$

[2] Full pivoting refers to searching through columns $j:n$ for the largest entry; this is more expensive and only marginal benefit to stability in practice

Partial Pivoting

To maintain the triangular LU structure, we permute rows by premultiplying by permutation matrices

$$\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \\ & x_{ij} & \times & \times \end{bmatrix} \xrightarrow{P_1} \begin{bmatrix} \times & \times & \times & \times \\ & x_{ij} & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \end{bmatrix} \xrightarrow{L_1} \begin{bmatrix} \times & \times & \times & \times \\ & x_{ij} & \times & \times \\ & 0 & \times & \times \\ & 0 & \times & \times \end{bmatrix}$$

Pivot selection

Row interchange

In this case

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

and each P_j is obtained by swapping two rows of I

Partial Pivoting

Therefore, with partial pivoting we obtain

$$L_{n-1}P_{n-1}\cdots L_2P_2L_1P_1A = U$$

It can be shown (we omit the details here, see Trefethen & Bau) that this can be rewritten as

$$PA = LU$$

where[3] $P \equiv P_{n-1}\cdots P_2P_1$

Theorem: Gaussian elimination with partial pivoting produces nonsingular factors L and U if and only if A is nonsingular.

[3] The L matrix here is lower triangular, but not the same as L in the non-pivoting case: we have to account for the row swaps

Partial Pivoting

Pseudocode for LU factorization with partial pivoting (blue text is new):

```
$U = A$, $L = \mathrm{I}$, $P = \mathrm{I}$
for $j = 1 : n-1$ do
    Select $i(\geq j)$ that maximizes $|u_{ij}|$
    Interchange rows of $U$: $u_{(j,j:n)} \leftrightarrow u_{(i,j:n)}$
    Interchange rows of $L$: $\ell_{(j,1:j-1)} \leftrightarrow \ell_{(i,1:j-1)}$
    Interchange rows of $P$: $p_{(j,:)} \leftrightarrow p_{(i,:)}$
    for $i = j+1 : n$ do
        $\ell_{ij} = u_{ij}/u_{jj}$
        for $k = j : n$ do
            $u_{ik} = u_{ik} - \ell_{ij} u_{jk}$
        end for
    end for
end for
```

Again this requires $\sim \frac{2}{3}n^3$ floating point operations

Partial Pivoting: Solve $Ax = b$

To solve $Ax = b$ using the factorization $PA = LU$:

- Multiply through by P to obtain $PAx = LUx = Pb$
- Solve $Ly = Pb$ using forward substitution
- Then solve $Ux = y$ using back substitution

Partial Pivoting in Python

Python's `scipy.linalg.lu` function can do LU factorization with pivoting.

```
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> import scipy.linalg
>>> a=np.random.random((4,4))
>>> a
array([[ 0.30178809,  0.09895414,  0.75341645,  0.55745407],
       [ 0.08879282,  0.97137694,  0.04768167,  0.28140464],
       [ 0.87253281,  0.66021495,  0.4941091 ,  0.52966743],
       [ 0.7990001 ,  0.45251929,  0.55493106,  0.15781707]])
>>> (p,l,u)=scipy.linalg.lu(a)
>>> p
array([[ 0.,  0.,  1.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> l
array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [ 0.10176445,  1.        ,  0.        ,  0.        ],
       [ 0.34587592, -0.14310957,  1.        ,  0.        ],
       [ 0.91572499, -0.16816814,  0.17525841,  1.        ]])
>>> u
array([[ 0.87253281,  0.66021495,  0.4941091 ,  0.52966743],
       [ 0.        ,  0.90419053, -0.00260107,  0.22750332],
       [ 0.        ,  0.        ,  0.58214377,  0.40681276],
       [ 0.        ,  0.        ,  0.        , -0.36025118]])
```

Stability of Gaussian Elimination

Numerical stability of Gaussian Elimination has been an important research topic since the 1940s

Major figure in this field: James H. Wilkinson (English numerical analyst, 1919–1986)

Showed that for $Ax = b$ with $A \in \mathbb{R}^{n\times n}$:

- Gaussian elimination without partial pivoting is numerically unstable (as we've already seen)
- Gaussian elimination with partial pivoting satisfies

$$\frac{\|r\|}{\|A\|\|x\|} \leq 2^{n-1} n^2 \epsilon_{\text{mach}}$$

Stability of Gaussian Elimination

That is, pathological cases exist where the relative residual, $\|r\|/\|A\|\|x\|$, grows exponentially with n due to rounding error

Worst case behavior of Gaussian Elimination with partial pivoting is explosive instability but such pathological cases are extremely rare!

In over 50 years of Scientific Computation, instability has only been encountered due to deliberate construction of pathological cases

In practice, Gaussian elimination is stable in the sense that it produces a small relative residual

Stability of Gaussian Elimination

In practice, we typically obtain

$$\frac{\|r\|}{\|A\|\|x\|} \lesssim n\epsilon_{\mathsf{mach}},$$

i.e. grows only linearly with n, and is scaled by ϵ_{mach}

Combining this result with our inequality:

$$\frac{\|\Delta x\|}{\|x\|} \leq \kappa(A)\frac{\|r\|}{\|A\|\|x\|}$$

implies that in practice Gaussian elimination gives small error for well-conditioned problems!

Cholesky Factorization

Cholesky factorization

Suppose that $A \in \mathbb{R}^{n \times n}$ is an "SPD" matrix, *i.e.*:

- Symmetric: $A^T = A$
- Positive Definite: for any $v \neq 0$, $v^T A v > 0$

Then the LU factorization of A can be arranged so that $U = L^T$, *i.e.* $A = LL^T$ (but in this case L may not have 1s on the diagonal)

Consider the 2×2 case:

$$\begin{bmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} \ell_{11} & 0 \\ \ell_{21} & \ell_{22} \end{bmatrix} \begin{bmatrix} \ell_{11} & \ell_{21} \\ 0 & \ell_{22} \end{bmatrix}$$

Equating entries gives

$$\ell_{11} = \sqrt{a_{11}}, \quad \ell_{21} = a_{21}/\ell_{11}, \quad \ell_{22} = \sqrt{a_{22} - \ell_{21}^2}$$

Cholesky factorization

This approach of equating entries can be used to derive the Cholesky factorization for the general $n \times n$ case

```
$L = A$
for $j = 1 : n$ do
    $\ell_{jj} = \sqrt{\ell_{jj}}$
    for $i = j + 1 : n$ do
        $\ell_{ij} = \ell_{ij} / \ell_{jj}$
    end for
    for $k = j + 1 : n$ do
        for $i = k : n$ do
            $\ell_{ik} = \ell_{ik} - \ell_{ij} \ell_{kj}$
        end for
    end for
end for
```

Cholesky factorization

Notes on Cholesky factorization:

- For an SPD matrix A, Cholesky factorization is numerically stable and does not require any pivoting
- Operation count: $\sim \frac{1}{3}n^3$ operations in total, *i.e.* about half as many as Gaussian elimination
- Only need to store L, hence uses less memory than LU

QR Factorization

QR Factorization

A square matrix $Q \in \mathbb{R}^{n \times n}$ is called orthogonal if its columns and rows are orthonormal vectors

Equivalently, $Q^T Q = QQ^T = \mathrm{I}$

Orthogonal matrices preserve the Euclidean norm of a vector, *i.e.*

$$\|Qv\|_2^2 = v^T Q^T Qv = v^T v = \|v\|_2^2$$

Hence, geometrically, we picture orthogonal matrices as reflection or rotation operators

Orthogonal matrices are very important in scientific computing, norm-preservation implies no amplification of numerical error!

QR Factorization

A matrix $A \in \mathbb{R}^{m \times n}$, $m \geq n$, can be factorized into

$$A = QR$$

where

- $Q \in \mathbb{R}^{m \times m}$ is orthogonal
- $R \equiv \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} \in \mathbb{R}^{m \times n}$
- $\hat{R} \in \mathbb{R}^{n \times n}$ is upper-triangular

QR is very good for solving overdetermined linear least-squares problems, $Ax \simeq b$ [4]

[4] QR can also be used to solve a square system $Ax = b$, but requires $\sim 2\times$ as many operations as Gaussian elimination hence not the standard choice

QR Factorization

To see why, consider the 2-norm of the least squares residual:

$$\begin{aligned}\|r(x)\|_2^2 &= \|b - Ax\|_2^2 = \|b - Q\left[\begin{array}{c}\hat{R}\\0\end{array}\right]x\|_2^2\\ &= \|Q^T\left(b - Q\left[\begin{array}{c}\hat{R}\\0\end{array}\right]x\right)\|_2^2\\ &= \|Q^Tb - \left[\begin{array}{c}\hat{R}\\0\end{array}\right]x\|_2^2\end{aligned}$$

(We used the fact that $\|Q^Tz\|_2 = \|z\|_2$ in the second line)

QR Factorization

Then, let $Q^T b = [c_1, c_2]^T$ where $c_1 \in \mathbb{R}^n, c_2 \in \mathbb{R}^{m-n}$, so that

$$\|r(x)\|_2^2 = \|c_1 - \hat{R}x\|_2^2 + \|c_2\|_2^2$$

Question: Based on this expression, how do we minimize $\|r(x)\|_2$?

QR Factorization

Answer: We can't influence the second term, $\|c_2\|_2^2$, since it doesn't contain an x

Hence we minimize $\|r(x)\|_2^2$ by making the first term zero

That is, we solve the $n \times n$ triangular system $\hat{R}x = c_1$ — this what Python does in its `lstsq` function for solving least squares

Also, this tells us that $\min_{x \in \mathbb{R}^n} \|r(x)\|_2 = \|c_2\|_2$

QR Factorization

Recall that solving linear least-squares via the normal equations requires solving a system with the matrix $A^T A$

But using the normal equations directly is problematic since $\text{cond}(A^T A) = \text{cond}(A)^2$ (this is a consequence of the SVD)

The QR approach avoids this condition-number-squaring effect and is much more numerically stable!

QR Factorization

How do we compute the QR Factorization?

There are three main methods

- Gram–Schmidt Orthogonalization
- Householder Triangularization
- Givens Rotations

We will cover Gram–Schmidt and Givens rotations in class

Gram–Schmidt Orthogonalization

Suppose $A \in \mathbb{R}^{m \times n}$, $m \geq n$

One way to picture the QR factorization is to construct a sequence of orthonormal vectors $q_1, q_2, \ldots$ such that

$$\text{span}\{q_1, q_2, \ldots, q_j\} = \text{span}\{a_{(:,1)}, a_{(:,2)}, \ldots, a_{(:,j)}\}, \quad j = 1, \ldots, n$$

We seek coefficients r_{ij} such that

$$\begin{aligned} a_{(:,1)} &= r_{11} q_1, \\ a_{(:,2)} &= r_{12} q_1 + r_{22} q_2, \\ &\vdots \\ a_{(:,n)} &= r_{1n} q_1 + r_{2n} q_2 + \cdots + r_{nn} q_n. \end{aligned}$$

This can be done via the Gram–Schmidt process, as we'll discuss shortly

Gram–Schmidt Orthogonalization

In matrix form we have:

$$\left[\begin{array}{c|c|c|c} & & & \\ a_{(:,1)} & a_{(:,2)} & \cdots & a_{(:,n)} \\ & & & \end{array}\right] = \left[\begin{array}{c|c|c|c} & & & \\ q_1 & q_2 & \cdots & q_n \\ & & & \end{array}\right] \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix}$$

This gives $A = \hat{Q}\hat{R}$ for $\hat{Q} \in \mathbb{R}^{m \times n}$, $\hat{R} \in \mathbb{R}^{n \times n}$

This is called the reduced QR factorization of A, which is slightly different from the definition we gave earlier

Note that for $m > n$, $\hat{Q}^T\hat{Q} = \mathrm{I}$, but $\hat{Q}\hat{Q}^T \neq \mathrm{I}$ (the latter is why the full QR is sometimes nice)

Full vs Reduced QR Factorization

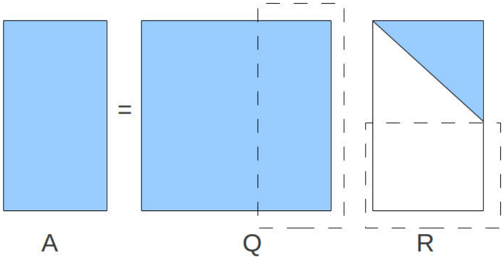
The full QR factorization (defined earlier)

$$A = QR$$

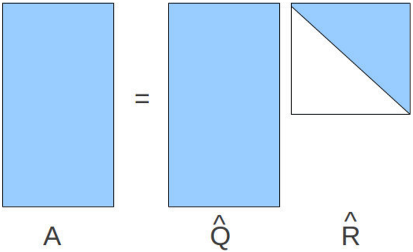
is obtained by appending $m - n$ arbitrary orthonormal columns to $\hat{Q}$ to make it an $m \times m$ orthogonal matrix

We also need to append rows of zeros to $\hat{R}$ to "silence" the last $m - n$ columns of Q, to obtain $R = \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix}$

Full vs Reduced QR Factorization



Full QR



Reduced QR

Full vs Reduced QR Factorization

Exercise: Show that the linear least-squares solution is given by $\hat{R}x = \hat{Q}^T b$ by plugging $A = \hat{Q}\hat{R}$ into the Normal Equations

This is equivalent to the least-squares result we showed earlier using the full QR factorization, since $c_1 = \hat{Q}^T b$

Full versus Reduced QR Factorization

In Python, `numpy.linalg.qr` gives the reduced QR factorization by default

```
Python 2.7.10 (default, Feb  7 2017, 00:08:15)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((5,3))
>>> (q,r)=np.linalg.qr(a)
>>> q
array([[-0.58479903,  0.18604305, -0.21857883],
       [-0.59514318, -0.34033765, -0.23588693],
       [-0.26381403,  0.14702842, -0.47775682],
       [-0.39324594, -0.43393772,  0.70189805],
       [-0.28208948,  0.79977432,  0.41912791]])
>>> r
array([[-1.50223926, -1.44239112, -1.0813288 ],
       [ 0.        ,  0.49087707,  0.4207912 ],
       [ 0.        , -0.        ,  0.65436304]])
```

Full versus Reduced QR Factorization

In Python, supplying the `mode=’complete’` option gives the complete QR factorization

```
Python 2.7.10 (default, Feb  7 2017, 00:08:15)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((5,3))
>>> (q,r)=np.linalg.qr(a,mode=’complete’)
>>> q
array([[-0.58479903,  0.18604305, -0.21857883, -0.18819304, -0.73498623],
       [-0.59514318, -0.34033765, -0.23588693, -0.40072023,  0.56013886],
       [-0.26381403,  0.14702842, -0.47775682,  0.80433265,  0.18325448],
       [-0.39324594, -0.43393772,  0.70189805,  0.39138159, -0.10590212],
       [-0.28208948,  0.79977432,  0.41912791, -0.06225843,  0.31818586]])
>>> r
array([[-1.50223926, -1.44239112, -1.0813288 ],
       [ 0.        ,  0.49087707,  0.4207912 ],
       [ 0.        , -0.        ,  0.65436304],
       [ 0.        ,  0.        , -0.        ],
       [ 0.        , -0.        , -0.        ]])
```

Gram–Schmidt Orthogonalization

Returning to the Gram–Schmidt process, how do we compute the q_i, $i = 1, \ldots, n$?

In the jth step, find a unit vector $q_j \in \text{span}\{a_{(:,1)}, a_{(:,2)}, \ldots, a_{(:,j)}\}$ that is orthogonal to $\text{span}\{q_1, q_n, \ldots, q_{j-1}\}$

We set

$$v_j \equiv a_{(:,j)} - (q_1^T a_{(:,j)})q_1 - \cdots - (q_{j-1}^T a_{(:,j)})q_{j-1},$$

and then $q_j \equiv v_j / \|v_j\|_2$ satisfies our requirements

We can now determine the required values of r_{ij}

Gram–Schmidt Orthogonalization

We then write our set of equations for the q_i as

$$\begin{aligned} q_1 &= \frac{a_{(:,1)}}{r_{11}}, \\ q_2 &= \frac{a_{(:,2)} - r_{12} q_1}{r_{22}}, \\ &\vdots \\ q_n &= \frac{a_{(:,n)} - \sum_{i=1}^{n-1} r_{in} q_i}{r_{nn}}. \end{aligned}$$

Then from the definition of q_j, we see that

$$\begin{aligned} r_{ij} &= q_i^T a_{(:,j)}, \qquad\qquad i \neq j \\ |r_{jj}| &= \|a_{(:,j)} - \sum_{i=1}^{j-1} r_{ij} q_i\|_2 \end{aligned}$$

The sign of r_{jj} is not determined uniquely, *e.g.* we could choose $r_{jj} > 0$ for each j

Classical Gram–Schmidt Process

The Gram–Schmidt algorithm we have described is provided in the pseudocode below

```
for $j = 1 : n$ do
    $v_j = a_{(:,j)}$
    for $i = 1 : j - 1$ do
        $r_{ij} = q_i^T a_{(:,j)}$
        $v_j = v_j - r_{ij} q_i$
    end for
    $r_{jj} = \|v_j\|_2$
    $q_j = v_j / r_{jj}$
end for
```

This is referred to the classical Gram–Schmidt (CGS) method

Gram–Schmidt Orthogonalization

The only way the Gram–Schmidt process can "fail" is if $|r_{jj}| = \|v_j\|_2 = 0$ for some j

This can only happen if $a_{(:,j)} = \sum_{i=1}^{j-1} r_{ij} q_i$ for some j, *i.e.* if $a_{(:,j)} \in \text{span}\{q_1, q_n, \ldots, q_{j-1}\} = \text{span}\{a_{(:,1)}, a_{(:,2)}, \ldots, a_{(:,j-1)}\}$

This means that columns of A are linearly dependent

Therefore, Gram–Schmidt fails $\implies$ cols. of A linearly dependent

Gram–Schmidt Orthogonalization

Equivalently, by contrapositive: cols. of A linearly independent $\implies$ Gram–Schmidt succeeds

Theorem: Every $A \in \mathbb{R}^{m\times n}(m \geq n)$ of full rank has a unique reduced QR factorization $A = \hat{Q}\hat{R}$ with $r_{ii} > 0$

The only non-uniqueness in the Gram–Schmidt process was in the sign of r_{ii}, hence $\hat{Q}\hat{R}$ is unique if $r_{ii} > 0$

Gram–Schmidt Orthogonalization

Theorem: Every $A \in \mathbb{R}^{m \times n} (m \geq n)$ has a full QR factorization.

Case 1: A has full rank

- We compute the reduced QR factorization from above
- To make Q square we pad $\hat{Q}$ with $m - n$ arbitrary orthonormal columns
- We also pad $\hat{R}$ with $m - n$ rows of zeros to get R

Case 2: A doesn't have full rank

- At some point in computing the reduced QR factorization, we encounter $\|v_j\|_2 = 0$
- At this point we pick an arbitrary q_j orthogonal to span$\{q_1, q_2, \ldots, q_{j-1}\}$ and then proceed as in Case 1

Modified Gram–Schmidt Process

The classical Gram–Schmidt process is numerically unstable! (sensitive to rounding error, orthogonality of the q_j degrades)

The algorithm can be reformulated to give the modified Gram–Schmidt process, which is numerically more robust

Key idea: when each new q_j is computed, orthogonalize each remaining column of A against it

Modified Gram–Schmidt Process

Modified Gram–Schmidt (MGS):

```
for $i = 1 : n$ do
    $v_i = a_{(:,i)}$
end for
for $i = 1 : n$ do
    $r_{ii} = \|v_i\|_2$
    $q_i = v_i / r_{ii}$
    for $j = i + 1 : n$ do
        $r_{ij} = q_i^T v_j$
        $v_j = v_j - r_{ij} q_i$
    end for
end for
```

Modified Gram–Schmidt Process

Key difference between MGS and CGS:

- In CGS we compute orthogonalization coefficients r_{ij} wrt the "raw" vector $a_{(:,j)}$
- In MGS we remove components of $a_{(:,j)}$ in $\mathrm{span}\{q_1, q_2, \ldots, q_{i-1}\}$ before computing r_{ij}

This makes no difference mathematically: In exact arithmetic components in $\mathrm{span}\{q_1, q_2, \ldots, q_{i-1}\}$ are annihilated by q_i^T

But in practice it reduces degradation of orthogonality of the q_j $\implies$ superior numerical stability of MGS over CGS

Operation Count

Work in MGS is dominated by lines 8 and 9, the innermost loop:

$$\begin{aligned} r_{ij} &= q_i^T v_j \\ v_j &= v_j - r_{ij} q_i \end{aligned}$$

First line requires m multiplications, $m - 1$ additions; second line requires m multiplications, m subtractions

Hence $\sim 4m$ operations per single inner iteration

Hence total number of operations is asymptotic to

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} 4m \sim 4m \sum_{i=1}^{n} i \sim 2mn^2$$

Alternative QR computation methods

The QR factorization can also be computed using Householder triangularization and Givens rotations.

Both methods take the approach of applying a sequence of orthogonal matrices $Q_1, Q_2, Q_3, \ldots$ to the matrix that successively remove terms below the diagonal (similar to the method employed by the LU factorization).

We will discuss Givens rotations.

A Givens rotation

For $i < j$ and an angle θ, the elements of the $m \times m$ Givens rotation matrix $G(i, j, \theta)$ are

$$\begin{gathered}
g_{ii} = c, \qquad g_{jj} = c, \qquad g_{ij} = s, \qquad g_{ji} = -s, \\
g_{kk} = 1 \quad \text{for } k \neq i, j, \\
g_{kl} = 0 \quad \text{otherwise,}
\end{gathered} \tag{1}$$

where $c = \cos\theta$ and $s = \sin\theta$.

A Givens rotation

Hence the matrix has the form

$$G(i,j,\theta) = \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{pmatrix}$$

It applies a rotation within the space spanned by the ith and jth coordinates

Effection of a Givens rotation

Consider a $m \times n$ rectangular matrix A where $m \geq n$

Suppose that a_1 and a_2 are in the ith and jth positions in a particular column of A

Restricting to just ith and jth dimensions, a Givens rotation $G(i, j, \theta)$ for a particular angle θ can be applied so that

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} \alpha \\ 0 \end{pmatrix},$$

where α is non-zero, and the jth component is eliminated

Stable computation

α is given by $\sqrt{a_1^2 + a_2^2}$. We could compute

$$c = a_1/\sqrt{a_1^2 + a_2^2}, \qquad s = a_2/\sqrt{a_1^2 + a_2^2}$$

but this is susceptible to underflow/overflow if α is very small.

A better procedure is as follows:

- if $|a_1| > |a_2|$, set $t = \tan\theta = a_2/a_1$, and hence $c = \frac{1}{\sqrt{1+t^2}}, s = ct$.
- if $|a_2| \geq |a_1|$, set $\tau = \cot\theta = a_1/a_2$, and hence $s = \frac{1}{\sqrt{1+\tau^2}}, c = s\tau$.

Givens rotation algorithm

To perform the Givens procedure on a dense $m \times n$ rectangular matrix A where $m \geq n$, the following algorithm can be used:

```
$R = A, Q = I$
for $k = 1 : n$ do
    for $j = m : k + 1$ do
        Construct $G = G(j-1, j, \theta)$ to eliminate $a_{jk}$
        $A = GA$
        $Q = QG^T$
    end for
end for
```

Givens rotation advantages

In general, for dense matrices, Givens rotations are not as efficient as the other two approaches (Gram–Schmidt and Householder)

However, they are advantageous for sparse matrices, since non-zero entries can be eliminated one-by-one. They are also amenable to parallelization. Consider the 6×6 matrix:

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ 5 & \times & \times & \times & \times & \times \\ 4 & 6 & \times & \times & \times & \times \\ 3 & 5 & 7 & \times & \times & \times \\ 2 & 4 & 6 & 8 & \times & \times \\ 1 & 3 & 5 & 7 & 9 & \times \end{pmatrix}$$

The numbers represent the steps at which a particular matrix entry can be eliminated. *e.g.* on step 3, elements $(4, 1)$ and $(6, 2)$ can be eliminated concurrently using $G(3, 4, \theta_a)$ and $G(5, 6, \theta_b)$, respectively, since these two matrices operate on different rows.

Singular Value Decomposition

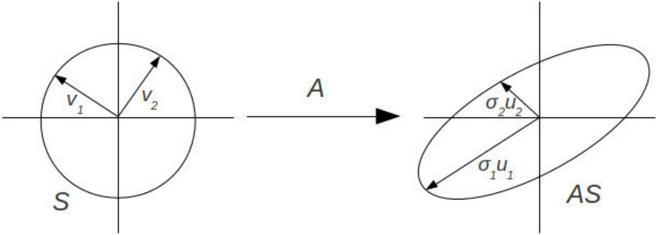
The Singular Value Decomposition (SVD) is a very useful matrix factorization

Motivation for SVD: image of the unit sphere, S, from any $m \times n$ matrix is a hyperellipse

A hyperellipse is obtained by stretching the unit sphere in $\mathbb{R}^m$ by factors $\sigma_1, \ldots, \sigma_m$ in orthogonal directions $u_1, \ldots, u_m$

Singular Value Decomposition

For $A \in \mathbb{R}^{2\times 2}$, we have



Singular Value Decomposition

Based on this picture, we make some definitions:

- Singular values: $\sigma_1, \sigma_2, \ldots, \sigma_n \geq 0$ (we typically assume $\sigma_1 \geq \sigma_2 \geq \ldots$)
- Left singular vectors: $\{u_1, u_2, \ldots, u_n\}$, unit vectors in directions of principal semiaxes of AS
- Right singular vectors: $\{v_1, v_2, \ldots, v_n\}$, preimages of the u_i so that $Av_i = \sigma_i u_i$, $i = 1, \ldots, n$

(The names "left" and "right" come from the formula for the SVD below)

Singular Value Decomposition

The key equation above is that

$$Av_i = \sigma_i u_i, \quad i = 1, \ldots, n$$

Writing this out in matrix form we get

$$\left[\begin{array}{c} \\ \\ A \\ \\ \\ \end{array} \right] \left[\begin{array}{c|c|c|c} v_1 & v_2 & \cdots & v_n \end{array} \right] = \left[\begin{array}{c|c|c|c} u_1 & u_2 & \cdots & u_n \end{array} \right] \left[\begin{array}{cccc} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{array} \right]$$

Or more compactly:

$$AV = \widehat{U}\widehat{\Sigma}$$

Singular Value Decomposition

Here

- $\widehat{\Sigma} \in \mathbb{R}^{n \times n}$ is diagonal with non-negative, real entries
- $\widehat{U} \in \mathbb{R}^{m \times n}$ with orthonormal columns
- $V \in \mathbb{R}^{n \times n}$ with orthonormal columns

Therefore V is an orthogonal matrix ($V^T V = VV^T = \mathrm{I}$), so that we have the reduced SVD for $A \in \mathbb{R}^{m \times n}$:

$$A = \widehat{U}\widehat{\Sigma}V^T$$

Singular Value Decomposition

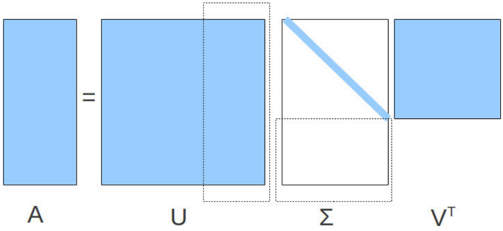
Just as with QR, we can pad the columns of $\widehat{U}$ with $m - n$ arbitrary orthogonal vectors to obtain $U \in \mathbb{R}^{m \times m}$

We then need to "silence" these arbitrary columns by adding rows of zeros to $\widehat{\Sigma}$ to obtain Σ

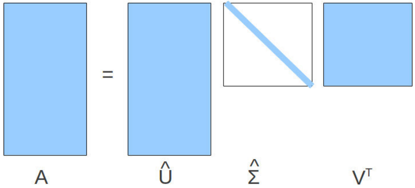
This gives the full SVD for $A \in \mathbb{R}^{m \times n}$:

$$A = U \Sigma V^T$$

Full vs Reduced SVD



Full SVD



Reduced SVD

Singular Value Decomposition

Theorem: Every matrix $A \in \mathbb{R}^{m \times n}$ has a full singular value decomposition. Furthermore:

- The σ_j are uniquely determined
- If A is square and the σ_j are distinct, the $\{u_j\}$ and $\{v_j\}$ are uniquely determined up to sign

Singular Value Decomposition

This theorem justifies the statement that the image of the unit sphere under any $m \times n$ matrix is a hyperellipse

Consider $A = U\Sigma V^T$ (full SVD) applied to the unit sphere, S, in $\mathbb{R}^n$:

1. The orthogonal map V^T preserves S
2. Σ stretches S into a hyperellipse aligned with the canonical axes e_j
3. U rotates or reflects the hyperellipse without changing its shape

SVD in Python

Python's `numpy.linalg.svd` function computes the full SVD of a matrix

```
Python 2.7.8 (default, Jul 13 2014, 17:11:32)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((4,2))
>>> (u,s,v)=np.linalg.svd(a)
>>> u
array([[-0.38627868,  0.3967265 , -0.44444737, -0.70417569],
       [-0.4748846 , -0.845594  , -0.23412286, -0.06813139],
       [-0.47511682,  0.05263149,  0.84419597, -0.24254299],
       [-0.63208972,  0.35328288, -0.18704595,  0.663828  ]])
>>> s
array([ 1.56149162,  0.24419604])
>>> v
array([[-0.67766849, -0.73536754],
       [-0.73536754,  0.67766849]])
```

SVD in Python

The `full_matrices=0` option computes the reduced SVD

```
Python 2.7.8 (default, Jul 13 2014, 17:11:32)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((4,2))
>>> (u,s,v)=np.linalg.svd(a,full_matrices=0)
>>> u
array([[-0.38627868,  0.3967265 ],
       [-0.4748846 , -0.845594  ],
       [-0.47511682,  0.05263149],
       [-0.63208972,  0.35328288]])
>>> s
array([ 1.56149162,  0.24419604])
>>> v
array([[-0.67766849, -0.73536754],
       [-0.73536754,  0.67766849]])
```

Matrix Properties via the SVD

- The rank of A is r, the number of nonzero singular values[2]

Proof: In the full SVD $A = U\Sigma V^T$, U and V^T have full rank, hence it follows from linear algebra that $\text{rank}(A) = \text{rank}(\Sigma)$

- $\text{image}(A) = \text{span}\{u_1, \ldots, u_r\}$ and $\text{null}(A) = \text{span}\{v_{r+1}, \ldots, v_n\}$

Proof: This follows from $A = U\Sigma V^T$ and

$$\begin{aligned} \mathrm{image}(\Sigma) &= \text{span}\{e_1, \ldots, e_r\} \in \mathbb{R}^m \\ \mathrm{null}(\Sigma) &= \text{span}\{e_{r+1}, \ldots, e_n\} \in \mathbb{R}^n \end{aligned}$$

[2] This also gives us a good way to define rank in finite precision: the number of singular values larger than some (small) tolerance

Matrix Properties via the SVD

- $\|A\|_2 = \sigma_1$

Proof: Recall that $\|A\|_2 \equiv \max_{\|v\|_2=1} \|Av\|_2$. Geometrically, we see that $\|Av\|_2$ is maximized if $v = v_1$ and $Av = \sigma_1 u_1$.

- The singular values of A are the square roots of the eigenvalues of $A^T A$ or AA^T

Proof: (Analogous for AA^T)

$$A^T A = (U\Sigma V^T)^T (U\Sigma V^T) = V\Sigma U^T U\Sigma V^T = V(\Sigma^T \Sigma)V^T,$$

hence $(A^T A)V = V(\Sigma^T \Sigma)$, or $(A^T A)v_{(:,j)} = \sigma_j^2 v_{(:,j)}$

Matrix Properties via the SVD

The pseudoinverse, A^+, can be defined more generally in terms of the SVD

Define pseudoinverse of a scalar σ to be $1/\sigma$ if $\sigma \neq 0$ and zero otherwise

Define pseudoinverse of a (possibly rectangular) diagonal matrix as transpose of the matrix and taking pseudoinverse of each entry

Pseudoinverse of $A \in \mathbb{R}^{m \times n}$ is defined as

$$A^+ = V\Sigma^+ U^T$$

A^+ exists for any matrix A, and it captures our definitions of pseudoinverse from previously

Matrix Properties via the SVD

We generalize the condition number to rectangular matrices via the definition $\kappa(A) = \|A\| \|A^+\|$

We can use the SVD to compute the 2-norm condition number:

- $\|A\|_2 = \sigma_{\mathsf{max}}$
- Largest singular value of A^+ is $1/\sigma_{\mathsf{min}}$ so that $\|A^+\|_2 = 1/\sigma_{\mathsf{min}}$

Hence $\kappa(A) = \sigma_{\mathsf{max}}/\sigma_{\mathsf{min}}$

Matrix Properties via the SVD

These results indicate the importance of the SVD, both theoretically and as a computational tool

Algorithms for calculating the SVD are an important topic in Numerical Linear Algebra, but outside scope of this course

Requires $\sim 4mn^2 - \frac{4}{3}n^3$ operations

For more details on algorithms, see Trefethen & Bau, or Golub & van Loan

Eigenvalue Problems

Eigenvalues and Eigenvectors

- Standard *eigenvalue problem*: Given $n \times n$ matrix $\boldsymbol{A}$, find scalar λ and nonzero vector $\boldsymbol{x}$ such that

$$\boldsymbol{A}\,\boldsymbol{x} = \lambda\,\boldsymbol{x}$$

- λ is *eigenvalue*, and $\boldsymbol{x}$ is corresponding *eigenvector*

- λ may be complex even if $\boldsymbol{A}$ is real

- *Spectrum* $= \lambda(\boldsymbol{A}) =$ set of all eigenvalues of $\boldsymbol{A}$

- *Spectral radius* $= \rho(\boldsymbol{A}) = \max\{|\lambda| : \lambda \in \lambda(\boldsymbol{A})\}$

Geometric Interpretation

- Matrix expands or shrinks any vector lying in direction of eigenvector by scalar factor
- Scalar expansion or contraction factor is given by corresponding eigenvalue λ
- Eigenvalues and eigenvectors decompose complicated behavior of general linear transformation into simpler actions

Eigenvalue Problems

- Eigenvalue problems occur in many areas of science and engineering, such as structural analysis
- Eigenvalues are also important in analyzing numerical methods
- Theory and algorithms apply to complex matrices as well as real matrices
- With complex matrices, we use conjugate transpose, $\boldsymbol{A}^H$, instead of usual transpose, $\boldsymbol{A}^T$

Examples: Eigenvalues and Eigenvectors

- $\boldsymbol{A} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$: $\lambda_1 = 1,\ \boldsymbol{x}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \lambda_2 = 2,\ \boldsymbol{x}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
- $\boldsymbol{A} = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$: $\lambda_1 = 1,\ \boldsymbol{x}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \lambda_2 = 2,\ \boldsymbol{x}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- $\boldsymbol{A} = \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix}$: $\lambda_1 = 2,\ \boldsymbol{x}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \lambda_2 = 4,\ \boldsymbol{x}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$
- $\boldsymbol{A} = \begin{bmatrix} 1.5 & 0.5 \\ 0.5 & 1.5 \end{bmatrix}$: $\lambda_1 = 2,\ \boldsymbol{x}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \lambda_2 = 1,\ \boldsymbol{x}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$
- $\boldsymbol{A} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$: $\lambda_1 = i,\ \boldsymbol{x}_1 = \begin{bmatrix} 1 \\ i \end{bmatrix}, \quad \lambda_2 = -i,\ \boldsymbol{x}_2 = \begin{bmatrix} i \\ 1 \end{bmatrix}$

  where $i = \sqrt{-1}$

Characteristic Polynomial and Multiplicity

Characteristic Polynomial

- Equation $\boldsymbol{Ax} = \lambda \boldsymbol{x}$ is equivalent to

  $$(\boldsymbol{A} - \lambda \boldsymbol{I})\boldsymbol{x} = \boldsymbol{0}$$

  which has nonzero solution $\boldsymbol{x}$ if, and only if, its matrix is singular

- Eigenvalues of $\boldsymbol{A}$ are roots λ_i of *characteristic polynomial*

  $$\det(\boldsymbol{A} - \lambda \boldsymbol{I}) = 0$$

  in λ of degree n

- *Fundamental Theorem of Algebra* implies that $n \times n$ matrix $\boldsymbol{A}$ always has n eigenvalues, but they may not be real nor distinct

- Complex eigenvalues of real matrix occur in complex conjugate pairs: if $\alpha + i\beta$ is eigenvalue of real matrix, then so is $\alpha - i\beta$, where $i = \sqrt{-1}$

Example: Characteristic Polynomial

- Characteristic polynomial of previous example matrix is

$$\det\left(\begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) =$$

$$\det\left(\begin{bmatrix} 3-\lambda & -1 \\ -1 & 3-\lambda \end{bmatrix}\right) =$$

$$(3-\lambda)(3-\lambda) - (-1)(-1) = \lambda^2 - 6\lambda + 8 = 0$$

so eigenvalues are given by

$$\lambda = \frac{6 \pm \sqrt{36-32}}{2}, \quad \text{or} \quad \lambda_1 = 2, \quad \lambda_2 = 4$$

Companion Matrix

- Monic polynomial

$$p(\lambda) = c_0 + c_1\lambda + \cdots + c_{n-1}\lambda^{n-1} + \lambda^n$$

is characteristic polynomial of *companion matrix*

$$\boldsymbol{C}_n = \begin{bmatrix} 0 & 0 & \cdots & 0 & -c_0 \\ 1 & 0 & \cdots & 0 & -c_1 \\ 0 & 1 & \cdots & 0 & -c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -c_{n-1} \end{bmatrix}$$

- Roots of polynomial of degree > 4 cannot always computed in finite number of steps

- So in general, computation of eigenvalues of matrices of order > 4 requires (theoretically infinite) iterative process

Characteristic Polynomial, continued

- Computing eigenvalues using characteristic polynomial is *not* recommended because of
  - work in computing coefficients of characteristic polynomial
  - sensitivity of coefficients of characteristic polynomial
  - work in solving for roots of characteristic polynomial
- Characteristic polynomial is powerful theoretical tool but usually not useful computationally

Example: Characteristic Polynomial

- Consider

  $$\boldsymbol{A} = \begin{bmatrix} 1 & \epsilon \\ \epsilon & 1 \end{bmatrix}$$

  where ϵ is positive number slightly smaller than $\sqrt{\epsilon_{\mathrm{mach}}}$

- Exact eigenvalues of $\boldsymbol{A}$ are $1 + \epsilon$ and $1 - \epsilon$

- Computing characteristic polynomial in floating-point arithmetic, we obtain

  $$\det(\boldsymbol{A} - \lambda \boldsymbol{I}) = \lambda^2 - 2\lambda + (1 - \epsilon^2) = \lambda^2 - 2\lambda + 1$$

  which has 1 as double root

- Thus, eigenvalues cannot be resolved by this method even though they are distinct in working precision

Multiplicity and Diagonalizability

- *Multiplicity* is number of times root appears when polynomial is written as product of linear factors

- *Simple* eigenvalue has multiplicity 1

- *Defective* matrix has eigenvalue of multiplicity $k > 1$ with fewer than k linearly independent corresponding eigenvectors

- Nondefective matrix $\boldsymbol{A}$ has n linearly independent eigenvectors, so it is *diagonalizable*

  $$\boldsymbol{X}^{-1}\boldsymbol{A}\boldsymbol{X} = \boldsymbol{D}$$

  where $\boldsymbol{X}$ is nonsingular matrix of eigenvectors

Eigenspaces and Invariant Subspaces

- Eigenvectors can be scaled arbitrarily: if $\boldsymbol{Ax} = \lambda \boldsymbol{x}$, then $\boldsymbol{A}(\gamma \boldsymbol{x}) = \lambda(\gamma \boldsymbol{x})$ for any scalar γ, so $\gamma \boldsymbol{x}$ is also eigenvector corresponding to λ

- Eigenvectors are usually *normalized* by requiring some norm of eigenvector to be 1

- *Eigenspace* $= \mathcal{S}_\lambda = \{\boldsymbol{x} : \boldsymbol{Ax} = \lambda \boldsymbol{x}\}$

- Subspace $\mathcal{S}$ of $\mathbb{R}^n$ (or $\mathbb{C}^n$) is *invariant* if $\boldsymbol{A}\mathcal{S} \subseteq \mathcal{S}$

- For eigenvectors $\boldsymbol{x}_1 \cdots \boldsymbol{x}_p$, span($[\boldsymbol{x}_1 \cdots \boldsymbol{x}_p]$) is invariant subspace

Properties of Eigenvalue Problems

Properties of eigenvalue problem affecting choice of algorithm and software

- Are all eigenvalues needed, or only a few?
- Are only eigenvalues needed, or are corresponding eigenvectors also needed?
- Is matrix real or complex?
- Is matrix relatively small and dense, or large and sparse?
- Does matrix have any special properties, such as symmetry, or is it general matrix?

Computing Eigenvalues and Eigenvectors

Problem Transformations

- *Shift*: If $\boldsymbol{Ax} = \lambda \boldsymbol{x}$ and σ is any scalar, then $(\boldsymbol{A} - \sigma \boldsymbol{I})\boldsymbol{x} = (\lambda - \sigma)\boldsymbol{x}$, so eigenvalues of shifted matrix are shifted eigenvalues of original matrix

- *Inversion*: If $\boldsymbol{A}$ is nonsingular and $\boldsymbol{Ax} = \lambda \boldsymbol{x}$ with $\boldsymbol{x} \neq \boldsymbol{0}$, then $\lambda \neq 0$ and $\boldsymbol{A}^{-1}\boldsymbol{x} = (1/\lambda)\boldsymbol{x}$, so eigenvalues of inverse are reciprocals of eigenvalues of original matrix

- *Powers*: If $\boldsymbol{Ax} = \lambda \boldsymbol{x}$, then $\boldsymbol{A}^k \boldsymbol{x} = \lambda^k \boldsymbol{x}$, so eigenvalues of power of matrix are same power of eigenvalues of original matrix

- *Polynomial*: If $\boldsymbol{Ax} = \lambda \boldsymbol{x}$ and $p(t)$ is polynomial, then $p(\boldsymbol{A})\boldsymbol{x} = p(\lambda)\boldsymbol{x}$, so eigenvalues of polynomial in matrix are values of polynomial evaluated at eigenvalues of original matrix

Similarity Transformation

- $\boldsymbol{B}$ is *similar* to $\boldsymbol{A}$ if there is nonsingular matrix $\boldsymbol{T}$ such that

$$\boldsymbol{B} = \boldsymbol{T}^{-1}\boldsymbol{A}\ \boldsymbol{T}$$

- Then

$$\boldsymbol{B}\boldsymbol{y} = \lambda\boldsymbol{y} \ \Rightarrow\ \boldsymbol{T}^{-1}\boldsymbol{A}\boldsymbol{T}\boldsymbol{y} = \lambda\boldsymbol{y} \ \Rightarrow\ \boldsymbol{A}(\boldsymbol{T}\boldsymbol{y}) = \lambda(\boldsymbol{T}\boldsymbol{y})$$

  so $\boldsymbol{A}$ and $\boldsymbol{B}$ have same eigenvalues, and if $\boldsymbol{y}$ is eigenvector of $\boldsymbol{B}$, then $\boldsymbol{x} = \boldsymbol{T}\boldsymbol{y}$ is eigenvector of $\boldsymbol{A}$

- Similarity transformations preserve eigenvalues, and eigenvectors are easily recovered

Example: Similarity Transformation

- From eigenvalues and eigenvectors for previous example,

$$\begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}$$

  and hence

$$\begin{bmatrix} 0.5 & 0.5 \\ 0.5 & -0.5 \end{bmatrix} \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}$$

- So original matrix is similar to diagonal matrix, and eigenvectors form columns of similarity transformation matrix

Diagonal Form

- Eigenvalues of diagonal matrix are diagonal entries, and eigenvectors are columns of identity matrix
- Diagonal form is desirable in simplifying eigenvalue problems for general matrices by similarity transformations
- But not all matrices are diagonalizable by similarity transformation
- Closest one can get, in general, is *Jordan form*, which is nearly diagonal but may have some nonzero entries on first superdiagonal, corresponding to one or more multiple eigenvalues

Triangular Form

- Any matrix can be transformed into triangular (*Schur*) form by similarity, and eigenvalues of triangular matrix are diagonal entries

- Eigenvectors of triangular matrix less obvious, but still straightforward to compute

- If

  $$\boldsymbol{A} - \lambda \boldsymbol{I} = \begin{bmatrix} \boldsymbol{U}_{11} & \boldsymbol{u} & \boldsymbol{U}_{13} \\ \boldsymbol{0} & 0 & \boldsymbol{v}^T \\ \boldsymbol{O} & \boldsymbol{0} & \boldsymbol{U}_{33} \end{bmatrix}$$

  is triangular, then $\boldsymbol{U}_{11}\boldsymbol{y} = \boldsymbol{u}$ can be solved for $\boldsymbol{y}$, so that

  $$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{y} \\ -1 \\ \boldsymbol{0} \end{bmatrix}$$

  is corresponding eigenvector

Block Triangular Form

- If

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & \cdots & \boldsymbol{A}_{1p} \\ & \boldsymbol{A}_{22} & \cdots & \boldsymbol{A}_{2p} \\ & & \ddots & \vdots \\ & & & \boldsymbol{A}_{pp} \end{bmatrix}$$

  with *square* diagonal blocks, then

$$\lambda(\boldsymbol{A}) = \bigcup_{j=1}^{p} \lambda(\boldsymbol{A}_{jj})$$

  so eigenvalue problem breaks into p smaller eigenvalue problems

- *Real* Schur form has 1×1 diagonal blocks corresponding to real eigenvalues and 2×2 diagonal blocks corresponding to pairs of complex conjugate eigenvalues

Forms Attainable by Similarity

$\boldsymbol{A}$	$\boldsymbol{T}$	$\boldsymbol{B}$
distinct eigenvalues	nonsingular	diagonal
real symmetric	orthogonal	real diagonal
complex Hermitian	unitary	real diagonal
normal	unitary	diagonal
arbitrary real	orthogonal	real block triangular (real Schur)
arbitrary	unitary	upper triangular (Schur)
arbitrary	nonsingular	almost diagonal (Jordan)

- Given matrix $\boldsymbol{A}$ with indicated property, matrices $\boldsymbol{B}$ and $\boldsymbol{T}$ exist with indicated properties such that $\boldsymbol{B} = \boldsymbol{T}^{-1}\boldsymbol{A}\boldsymbol{T}$

- If $\boldsymbol{B}$ is diagonal or triangular, eigenvalues are its diagonal entries

- If $\boldsymbol{B}$ is diagonal, eigenvectors are columns of $\boldsymbol{T}$

Power Iteration

Power Iteration

- Simplest method for computing one eigenvalue-eigenvector pair is *power iteration*, which repeatedly multiplies matrix times initial starting vector

- Assume $\boldsymbol{A}$ has unique eigenvalue of maximum modulus, say λ_1, with corresponding eigenvector $\boldsymbol{v}_1$

- Then, starting from nonzero vector $\boldsymbol{x}_0$, iteration scheme

$$\boldsymbol{x}_k = \boldsymbol{A}\boldsymbol{x}_{k-1}$$

  converges to multiple of eigenvector $\boldsymbol{v}_1$ corresponding to *dominant* eigenvalue λ_1

Convergence of Power Iteration

- To see why power iteration converges to dominant eigenvector, express starting vector $\boldsymbol{x}_0$ as linear combination

  $$\boldsymbol{x}_0 = \sum_{i=1}^{n} \alpha_i \boldsymbol{v}_i$$

  where $\boldsymbol{v}_i$ are eigenvectors of $\boldsymbol{A}$

- Then

  $$\boldsymbol{x}_k = \boldsymbol{A}\boldsymbol{x}_{k-1} = \boldsymbol{A}^2\boldsymbol{x}_{k-2} = \cdots = \boldsymbol{A}^k\boldsymbol{x}_0 =$$

  $$\sum_{i=1}^{n} \lambda_i^k \alpha_i \boldsymbol{v}_i = \lambda_1^k \left(\alpha_1 \boldsymbol{v}_1 + \sum_{i=2}^{n} (\lambda_i/\lambda_1)^k \alpha_i \boldsymbol{v}_i \right)$$

- Since $|\lambda_i/\lambda_1| < 1$ for $i > 1$, successively higher powers go to zero, leaving only component corresponding to $\boldsymbol{v}_1$

Example: Power Iteration

- Ratio of values of given component of $\boldsymbol{x}_k$ from one iteration to next converges to dominant eigenvalue λ_1

- For example, if $\boldsymbol{A} = \begin{bmatrix} 1.5 & 0.5 \\ 0.5 & 1.5 \end{bmatrix}$ and $\boldsymbol{x}_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, we obtain

k	$\boldsymbol{x}_k^T$		ratio
0	0.0	1.0	
1	0.5	1.5	1.500
2	1.5	2.5	1.667
3	3.5	4.5	1.800
4	7.5	8.5	1.889
5	15.5	16.5	1.941
6	31.5	32.5	1.970
7	63.5	64.5	1.985
8	127.5	128.5	1.992

- Ratio is converging to dominant eigenvalue, which is 2

Limitations of Power Iteration

Power iteration can fail for various reasons

- Starting vector may have *no* component in dominant eigenvector $\boldsymbol{v}_1$ (i.e., $\alpha_1 = 0$) — not problem in practice because rounding error usually introduces such component in any case
- There may be more than one eigenvalue having same (maximum) modulus, in which case iteration may converge to linear combination of corresponding eigenvectors
- For real matrix and starting vector, iteration can never converge to complex vector

Normalized Power Iteration

- Geometric growth of components at each iteration risks eventual overflow (or underflow if $\lambda_1 < 1$)

- Approximate eigenvector should be normalized at each iteration, say, by requiring its largest component to be 1 in modulus, giving iteration scheme

$$\begin{aligned} \boldsymbol{y}_k &= \boldsymbol{A}\boldsymbol{x}_{k-1} \\ \boldsymbol{x}_k &= \boldsymbol{y}_k / \|\boldsymbol{y}_k\|_\infty \end{aligned}$$

- With normalization, $\|\boldsymbol{y}_k\|_\infty \to |\lambda_1|$, and $\boldsymbol{x}_k \to \boldsymbol{v}_1 / \|\boldsymbol{v}_1\|_\infty$

Example: Normalized Power Iteration

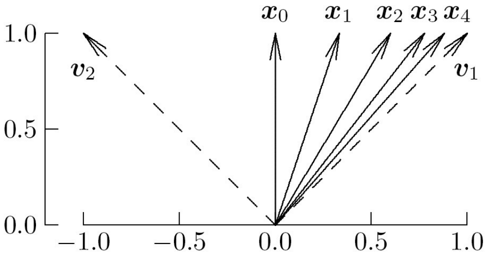
- Repeating previous example with normalized scheme,

k	$\boldsymbol{x}_k^T$		$\|\boldsymbol{y}_k\|_\infty$
0	0.000	1.0	
1	0.333	1.0	1.500
2	0.600	1.0	1.667
3	0.778	1.0	1.800
4	0.882	1.0	1.889
5	0.939	1.0	1.941
6	0.969	1.0	1.970
7	0.984	1.0	1.985
8	0.992	1.0	1.992

$\langle$ interactive example $\rangle$

Geometric Interpretation

- Behavior of power iteration depicted geometrically



- Initial vector $\boldsymbol{x}_0 = \boldsymbol{v}_1 + \boldsymbol{v}_2$ contains equal components in eigenvectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ (dashed arrows)

- Repeated multiplication by $\boldsymbol{A}$ causes component in $\boldsymbol{v}_1$ (corresponding to larger eigenvalue, 2) to dominate, so sequence of vectors $\boldsymbol{x}_k$ converges to $\boldsymbol{v}_1$

Power Iteration with Shift

- Convergence rate of power iteration depends on ratio $|\lambda_2/\lambda_1|$, where λ_2 is eigenvalue having second largest modulus

- May be possible to choose shift, $\boldsymbol{A} - \sigma\boldsymbol{I}$, such that

  $$\left|\frac{\lambda_2 - \sigma}{\lambda_1 - \sigma}\right| < \left|\frac{\lambda_2}{\lambda_1}\right|$$

  so convergence is accelerated

- Shift must then be added to result to obtain eigenvalue of original matrix

- In earlier example, for instance, if we pick shift of $\sigma = 1$, (which is equal to other eigenvalue) then ratio becomes zero and method converges in one iteration

- In general, we would not be able to make such fortuitous choice, but shifts can still be extremely useful in some contexts, as we will see later

Inverse and Rayleigh Quotient Iterations

Inverse Iteration

- To compute smallest eigenvalue of matrix rather than largest, can make use of fact that eigenvalues of $\boldsymbol{A}^{-1}$ are reciprocals of those of $\boldsymbol{A}$, so smallest eigenvalue of $\boldsymbol{A}$ is reciprocal of largest eigenvalue of $\boldsymbol{A}^{-1}$
- This leads to *inverse iteration* scheme

  $$\begin{aligned} \boldsymbol{A}\boldsymbol{y}_k &= \boldsymbol{x}_{k-1} \\ \boldsymbol{x}_k &= \boldsymbol{y}_k / \|\boldsymbol{y}_k\|_\infty \end{aligned}$$

  which is equivalent to power iteration applied to $\boldsymbol{A}^{-1}$
- Inverse of $\boldsymbol{A}$ not computed explicitly, but factorization of $\boldsymbol{A}$ used to solve system of linear equations at each iteration
- Inverse iteration converges to eigenvector corresponding to *smallest* eigenvalue of $\boldsymbol{A}$
- Eigenvalue obtained is dominant eigenvalue of $\boldsymbol{A}^{-1}$, and hence its reciprocal is smallest eigenvalue of $\boldsymbol{A}$ in modulus

Example: Inverse Iteration

- Applying inverse iteration to previous example to compute smallest eigenvalue yields sequence

k	$\boldsymbol{x}_k^T$		$\|\boldsymbol{y}_k\|_\infty$
0	0.000	1.0	
1	-0.333	1.0	0.750
2	-0.600	1.0	0.833
3	-0.778	1.0	0.900
4	-0.882	1.0	0.944
5	-0.939	1.0	0.971
6	-0.969	1.0	0.985

which is indeed converging to 1 (which is its own reciprocal in this case)

⟨ interactive example ⟩

Inverse Iteration with Shift

- As before, shifting strategy, working with $\boldsymbol{A} - \sigma \boldsymbol{I}$ for some scalar σ, can greatly improve convergence

- Inverse iteration is particularly useful for computing eigenvector corresponding to approximate eigenvalue, since it converges rapidly when applied to shifted matrix $\boldsymbol{A} - \lambda \boldsymbol{I}$, where λ is approximate eigenvalue

- Inverse iteration is also useful for computing eigenvalue closest to given value β, since if β is used as shift, then desired eigenvalue corresponds to smallest eigenvalue of shifted matrix

Rayleigh Quotient

- Given approximate eigenvector $\boldsymbol{x}$ for real matrix $\boldsymbol{A}$, determining best estimate for corresponding eigenvalue λ can be considered as $n \times 1$ linear least squares approximation problem

$$\boldsymbol{x}\lambda \cong \boldsymbol{A}\boldsymbol{x}$$

- From normal equation $\boldsymbol{x}^T\boldsymbol{x}\lambda = \boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}$, least squares solution is given by

$$\lambda = \frac{\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}}{\boldsymbol{x}^T\boldsymbol{x}}$$

- This quantity, known as *Rayleigh quotient*, has many useful properties

Example: Rayleigh Quotient

- Rayleigh quotient can accelerate convergence of iterative methods such as power iteration, since Rayleigh quotient $\boldsymbol{x}_k^T \boldsymbol{A} \boldsymbol{x}_k / \boldsymbol{x}_k^T \boldsymbol{x}_k$ gives better approximation to eigenvalue at iteration k than does basic method alone

- For previous example using power iteration, value of Rayleigh quotient at each iteration is shown below

k	$\boldsymbol{x}_k^T$		$\|\boldsymbol{y}_k\|_\infty$	$\boldsymbol{x}_k^T \boldsymbol{A} \boldsymbol{x}_k / \boldsymbol{x}_k^T \boldsymbol{x}_k$
0	0.000	1.0		
1	0.333	1.0	1.500	1.500
2	0.600	1.0	1.667	1.800
3	0.778	1.0	1.800	1.941
4	0.882	1.0	1.889	1.985
5	0.939	1.0	1.941	1.996
6	0.969	1.0	1.970	1.999

Rayleigh Quotient Iteration

- Given approximate eigenvector, Rayleigh quotient yields good estimate for corresponding eigenvalue

- Conversely, inverse iteration converges rapidly to eigenvector if approximate eigenvalue is used as shift, with one iteration often sufficing

- These two ideas combined in *Rayleigh quotient iteration*

$$\sigma_k = \boldsymbol{x}_k^T \boldsymbol{A} \boldsymbol{x}_k / \boldsymbol{x}_k^T \boldsymbol{x}_k$$

$$(\boldsymbol{A} - \sigma_k \boldsymbol{I}) \boldsymbol{y}_{k+1} = \boldsymbol{x}_k$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{y}_{k+1} / \|\boldsymbol{y}_{k+1}\|_\infty$$

  starting from given nonzero vector $\boldsymbol{x}_0$

Rayleigh Quotient Iteration, continued

- Rayleigh quotient iteration is especially effective for symmetric matrices and usually converges very rapidly

- Using different shift at each iteration means matrix must be refactored each time to solve linear system, so cost per iteration is high unless matrix has special form that makes factorization easy

- Same idea also works for complex matrices, for which transpose is replaced by conjugate transpose, so Rayleigh quotient becomes $\boldsymbol{x}^H \boldsymbol{A}\boldsymbol{x} / \boldsymbol{x}^H \boldsymbol{x}$

Example: Rayleigh Quotient Iteration

- Using same matrix as previous examples and randomly chosen starting vector $\boldsymbol{x}_0$, Rayleigh quotient iteration converges in two iterations

k	$\boldsymbol{x}_k^T$		σ_k
0	0.807	0.397	1.896
1	0.924	1.000	1.998
2	1.000	1.000	2.000

Deflation

Deflation

- After eigenvalue λ_1 and corresponding eigenvector $\boldsymbol{x}_1$ have been computed, then additional eigenvalues $\lambda_2, \dots, \lambda_n$ of $\boldsymbol{A}$ can be computed by *deflation*, which effectively removes known eigenvalue

- Let $\boldsymbol{H}$ be any nonsingular matrix such that $\boldsymbol{H}\boldsymbol{x}_1 = \alpha \boldsymbol{e}_1$, scalar multiple of first column of identity matrix (Householder transformation is good choice for $\boldsymbol{H}$)

- Then similarity transformation determined by $\boldsymbol{H}$ transforms $\boldsymbol{A}$ into form

$$\boldsymbol{H}\boldsymbol{A}\boldsymbol{H}^{-1} = \begin{bmatrix} \lambda_1 & \boldsymbol{b}^T \\ \boldsymbol{0} & \boldsymbol{B} \end{bmatrix}$$

  where $\boldsymbol{B}$ is matrix of order $n-1$ having eigenvalues $\lambda_2, \dots, \lambda_n$

Deflation, continued

- Thus, we can work with $\boldsymbol{B}$ to compute next eigenvalue λ_2
- Moreover, if $\boldsymbol{y}_2$ is eigenvector of $\boldsymbol{B}$ corresponding to λ_2, then

  $$\boldsymbol{x}_2 = \boldsymbol{H}^{-1} \begin{bmatrix} \alpha \\ \boldsymbol{y}_2 \end{bmatrix}, \quad \text{where} \quad \alpha = \frac{\boldsymbol{b}^T \boldsymbol{y}_2}{\lambda_2 - \lambda_1}$$

  is eigenvector corresponding to λ_2 for original matrix $\boldsymbol{A}$, provided $\lambda_1 \neq \lambda_2$
- Process can be repeated to find additional eigenvalues and eigenvectors

Deflation, continued

- Alternative approach lets $\boldsymbol{u}_1$ be any vector such that $\boldsymbol{u}_1^T \boldsymbol{x}_1 = \lambda_1$
- Then $\boldsymbol{A} - \boldsymbol{x}_1 \boldsymbol{u}_1^T$ has eigenvalues $0, \lambda_2, \dots, \lambda_n$
- Possible choices for $\boldsymbol{u}_1$ include
  - $\boldsymbol{u}_1 = \lambda_1 \boldsymbol{x}_1$, if $\boldsymbol{A}$ is symmetric and $\boldsymbol{x}_1$ is normalized so that $\|\boldsymbol{x}_1\|_2 = 1$
  - $\boldsymbol{u}_1 = \lambda_1 \boldsymbol{y}_1$, where $\boldsymbol{y}_1$ is corresponding left eigenvector (i.e., $\boldsymbol{A}^T \boldsymbol{y}_1 = \lambda_1 \boldsymbol{y}_1$) normalized so that $\boldsymbol{y}_1^T \boldsymbol{x}_1 = 1$
  - $\boldsymbol{u}_1 = \boldsymbol{A}^T \boldsymbol{e}_k$, if $\boldsymbol{x}_1$ is normalized so that $\|\boldsymbol{x}_1\|_\infty = 1$ and kth component of $\boldsymbol{x}_1$ is 1