AM 205: lecture 8

- Last time: LU factorization and timing
- Today: Cholesky factorization, QR decomposition
- Reminder: assignment 1 due at 5 PM on Friday September 26

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

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

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

```
1:  L = A
2:  for j = 1 : n do
3:      ℓ_jj = √ℓ_jj
4:      for i = j + 1 : n do
5:          ℓ_ij = ℓ_ij / ℓ_jj
6:      end for
7:      for k = j + 1 : n do
8:          for i = k : n do
9:              ℓ_ik = ℓ_ik − ℓ_ij ℓ_kj
10:         end for
11:     end for
12: 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

Sparse Matrices

Sparse Matrices

In applications, we often encounter sparse matrices

A prime example is in discretization of partial differential equations (covered in the next section)

"Sparse matrix" is not precisely defined, roughly speaking it is a matrix that is "mostly zeros"

From a computational point of view it is advantageous to store only the non-zero entries

The set of non-zero entries of a sparse matrix is referred to as its sparsity pattern

Sparse Matrices

```
A =
     2     2     2     2     2
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1

A_sparse =
   (1,1)        2
   (1,2)        2
   (2,2)        1
   (1,3)        2
   (3,3)        1
   (1,4)        2
   (4,4)        1
   (1,5)        2
   (5,5)        1
```

Sparse Matrices

From a mathematical point of view, sparse matrices are no different from dense matrices

But from Sci. Comp. perspective, sparse matrices require different data structures and algorithms for computational efficiency

e.g., can apply LU or Cholesky to sparse A, but "new" non-zeros (i.e outside sparsity pattern of A) are introduced in the factors

These new non-zero entries are called "fill-in" — many methods exist for reducing fill-in by permuting rows and columns of A

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

As we indicated earlier, QR is very good for solving overdetermined linear least-squares problems, $Ax \simeq b$ [1]

[1] 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:

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

(We used the fact that $\|Q^T z\|_2 = \|z\|_2$ 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 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

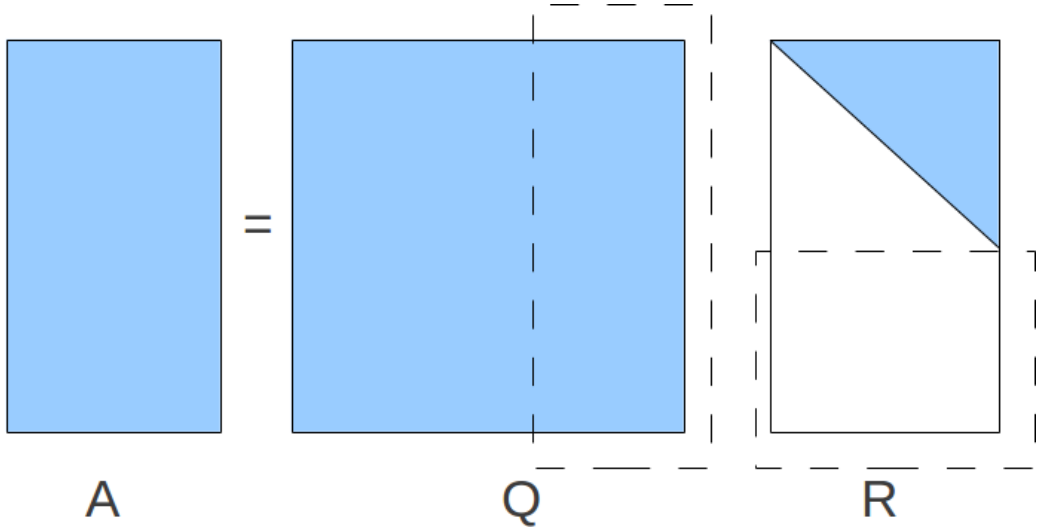
The full QR factorization (defined earlier)

$$A = QR$$

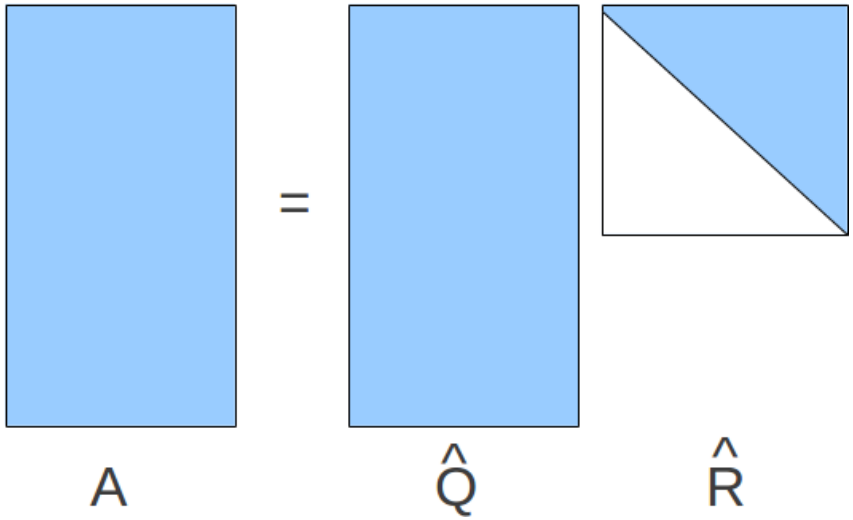
is obtained by appending $m - n$ arbitrary orthonormal columns to $\hat{Q}$ to make it an $m \times m$ orthogonal matrix

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 2.7.8 (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((5,3))
>>> (q,r)=np.linalg.qr(a)
>>> q
array([[-0.07519839, -0.74962369,  0.041629  ],
       [-0.25383796, -0.42804369, -0.24484042],
       [-0.78686491,  0.40126393, -0.21414012],
       [-0.06631853, -0.17513071, -0.79738432],
       [-0.55349522, -0.25131537,  0.5065989 ]])
>>> r
array([[-1.17660526, -0.58996516, -1.49606297],
       [ 0.        , -0.34262421, -0.72248544],
       [ 0.        ,  0.        , -0.35192857]])
```

Full versus Reduced QR Factorization

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

```
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((5,3))
>>> (q,r)=np.linalg.qr(a,mode='complete')
>>> q
array([[-0.07519839, -0.74962369,  0.041629  ,  0.12584638, -0.64408015],
       [-0.25383796, -0.42804369, -0.24484042, -0.74729789,  0.3659835 ],
       [-0.78686491,  0.40126393, -0.21414012, -0.09167938, -0.40690266],
       [-0.06631853, -0.17513071, -0.79738432,  0.51140185,  0.25995667],
       [-0.55349522, -0.25131537,  0.5065989 ,  0.3946791 ,  0.46697922]])
>>> r
array([[-1.17660526, -0.58996516, -1.49606297],
       [ 0.        , -0.34262421, -0.72248544],
       [ 0.        ,  0.        , -0.35192857],
       [ 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

1: **for** $j = 1 : n$ **do**
2: $\quad v_j = a_{(:,j)}$
3: $\quad$ **for** $i = 1 : j - 1$ **do**
4: $\quad\quad r_{ij} = q_i^T a_{(:,j)}$
5: $\quad\quad v_j = v_j - r_{ij} q_i$
6: $\quad$ **end for**
7: $\quad r_{jj} = \|v_j\|_2$
8: $\quad q_j = v_j / r_{jj}$
9: **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 $\text{span}\{q_1, q_2, \ldots, q_{j-1}\}$ 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):

1: **for** $i = 1 : n$ **do**
2: $\quad v_i = a_{(:,i)}$
3: **end for**
4: **for** $i = 1 : n$ **do**
5: $\quad r_{ii} = \|v_i\|_2$
6: $\quad q_i = v_i / r_{ii}$
7: $\quad$ **for** $j = i + 1 : n$ **do**
8: $\quad\quad r_{ij} = q_i^T v_j$
9: $\quad\quad v_j = v_j - r_{ij} q_i$
10: $\quad$ **end for**
11: **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 $\text{span}\{q_1, q_2, \ldots, q_{i-1}\}$ before computing r_{ij}

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

But in practice it reduces degradation of orthogonality of the q_j
$\Longrightarrow$ 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

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