

Computational Notebook

Tuesday, December 14, 2021 4:45 PM

Contents:

I. Matrix Factorizations

- 1) QR Decomposition
 - i) Classical Gram-Schmidt
 - ii) Modified Gram-Schmidt
 - iii) Iterated Gram-Schmidt
 - iv) Householder reflections
- 2) Singular Value Decomposition
 - i) Truncated SVD
 - ii) Image compression
 - iii) Voting tendencies.
- 3) LU decomposition
- 4) Cholesky Decomposition

II. Approximations and Solution Techniques for Linear Systems

- 1) Linear Least Squares
- 2) Forward/Backward Substitution
- 3) Matrix Inversion
- 4) Low Rank Approximation

III. Finite Precision and Matrix Conditioning

- 1) Floating Point Number System
- 2) Conditioning

- 1) Floating Point Number System
- 2) Conditioning

Part I. Matrix Factorizations

1. QR Decomposition.

Definition: Let A be a matrix in $\mathbb{C}^{n \times n}$

Then, A has a QR Decomposition

$$A = QR, \quad Q, R \in \mathbb{C}^{n \times n}$$

where Q is unitary and R is upper triangular. This definition can be slightly modified for $A \in \mathbb{C}^{n \times m}$

Reduced QR

$$A = \begin{bmatrix} Q \\ R \end{bmatrix}$$

$m \times n$ $n \times n$

$Q^*Q = I \neq QQ^*$

Full QR

$$A = \begin{bmatrix} Q \\ R \end{bmatrix}$$

$m \times m$ $n \times m$

$$Q^T Q = I \neq Q Q^T \quad \begin{matrix} \text{size } m \times m \\ \text{size } m \times n \\ \text{size } n \times n \end{matrix}$$

There are a few ways to derive $A = QR$.

i) Classical Gram-Schmidt

The Gram-Schmidt process is a deterministic algorithm that gives an orthonormal basis for the column space of A .

$$\tilde{Q}_1 = A_1 / \|A_1\|$$

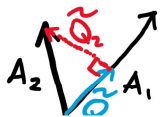
$$\tilde{Q}_2 = \tilde{Q}_2 / \|\tilde{Q}_2\| = \frac{A_2 - (Q_1 \cdot A_2) Q_1}{\|A_2 - (Q_1 \cdot A_2) Q_1\|}$$

where " $\cdot$ " is the inner product.

$$\tilde{Q}_j = \tilde{Q}_j / \|\tilde{Q}_j\| = \frac{A_j - \sum_{k=1}^{j-1} (\tilde{Q}_k \cdot A_j) \tilde{Q}_k}{\|A_j - \sum_{k=1}^{j-1} (\tilde{Q}_k \cdot A_j) \tilde{Q}_k\|}$$

Then,
$$r_{jk} = \begin{cases} \|\tilde{Q}_j\| & \text{if } j=k \\ \tilde{Q}_j \cdot A_k & \text{if } j < k \\ 0 & \text{if } j > k \end{cases}$$
 " $\cdot$ " is inner product

Visualization.



$$A_2 \vee A_1$$

Example.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad A_1 = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \quad A_2 = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

$$\vec{Q}_1 = \frac{A_1}{\|A_1\|} = \frac{\begin{bmatrix} 1 \\ 3 \end{bmatrix}}{\sqrt{1^2 + 3^2}} = \begin{bmatrix} 1/\sqrt{10} \\ 3/\sqrt{10} \end{bmatrix}$$

$$\tilde{Q}_2 = \begin{bmatrix} 2 \\ 4 \end{bmatrix} - \left(\begin{bmatrix} 1/\sqrt{10} \\ 3/\sqrt{10} \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 4 \end{bmatrix} \right) \begin{bmatrix} 1/\sqrt{10} \\ 3/\sqrt{10} \end{bmatrix}$$

$$= \begin{bmatrix} 2 - \frac{14}{10} \\ 4 - \frac{42}{10} \end{bmatrix} = \begin{bmatrix} 3/5 \\ -1/5 \end{bmatrix}$$

$$\vec{Q}_2 = \frac{\begin{bmatrix} 3/5 \\ -1/5 \end{bmatrix}}{\sqrt{(\frac{3}{5})^2 + (\frac{1}{5})^2}} = \frac{5}{\sqrt{10}} \begin{bmatrix} 3/5 \\ -1/5 \end{bmatrix}$$

$$= \begin{bmatrix} 3\sqrt{10}/25 \\ -\sqrt{10}/25 \end{bmatrix}$$

$$Q = \begin{bmatrix} 1/\sqrt{10} & 3/\sqrt{10} \\ 3/\sqrt{10} & -1/\sqrt{10} \end{bmatrix}$$

Finding R

$$r_{12} = \tilde{Q}_1^T A_2 = \begin{bmatrix} 1/\sqrt{10} \\ 3/\sqrt{10} \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \frac{14}{\sqrt{10}}$$

$$r_{22} = \|\tilde{Q}_2\| = \sqrt{(3/5)^2 + (1/5)^2} = \sqrt{10}/5$$

$$R = \begin{bmatrix} \sqrt{10} & 14/\sqrt{10} \\ 0 & \sqrt{10}/5 \end{bmatrix}$$

Matlab code here!

Prosi:
Simple, lets us understand/prove
things about QR.

```
1 function [Q,R] = ClassialGS(A)
2 [m,n] = size(A);
3 Q = zeros(m,n);
4 R = zeros(n,n);
5
6 for j=1:n
7     R(j,j) = norm(A(:,j));
8     Q(:,j) = A(:,j) / R(j,j);
9
10    for i=1:j-1
11
12        R(i,j) = Q(:,i)'*A(:,j);
13        A(:,j) = A(:,j) - R(i,j)*Q(:,i); %This is the Ak - sum part of the algorithm
14
15    end
16 end
17 end
18
```

Cons:

Unstable, slow, loses orthogonality

The classical Gram-Schmidt process
is not practical except for learning
purposes. It is very inaccurate, and slow
for solving linear systems.

purposes. It is very inaccurate, and slow for solving linear systems.

ii) Modified Gram-Schmidt

This algorithm is an adjustment to the last one that makes it more useful.

Classical GS uses 1 projection of rank $m - j + 1$

Modified GS does $j - 1$ projections of rank $m - 1$.

Example:

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

$$r_{11} = \|A_1\| = \sqrt{1^2 + 3^2} = \sqrt{10}$$

$$Q_1 = A_1 / r_{11} = \frac{\begin{bmatrix} 1 \\ 3 \end{bmatrix}}{\sqrt{10}} = \begin{bmatrix} 1/\sqrt{10} \\ 3/\sqrt{10} \end{bmatrix}$$

$$r_{12} = Q_1 \cdot A_2 = \begin{bmatrix} 1/\sqrt{10} \\ 3/\sqrt{10} \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \frac{14}{\sqrt{10}}$$

$$r_{22} = \|A_2 - r_{12} \cdot Q_1\| =$$

..

$$r_2 = \|A_2 - r_2 \cdot Q_1\| =$$

$$\left\| \begin{bmatrix} 2 \\ 4 \end{bmatrix} - \frac{14}{\sqrt{10}} \begin{bmatrix} 1/\sqrt{10} \\ 3/\sqrt{10} \end{bmatrix} \right\| = \left\| \begin{bmatrix} 2 \\ 4 \end{bmatrix} - \begin{bmatrix} 14/10 \\ 42/10 \end{bmatrix} \right\|$$

$$= \left\| \begin{bmatrix} 3/5 \\ -1/5 \end{bmatrix} \right\| = \sqrt{10}/5$$

$$Q_2 = \frac{A_2 - r_2 Q_1}{r_2} = \frac{5}{\sqrt{10}} \begin{bmatrix} 3/5 \\ -1/5 \end{bmatrix} = \begin{bmatrix} 3/\sqrt{10} \\ -1/\sqrt{10} \end{bmatrix}$$

$$Q = \begin{bmatrix} 1/\sqrt{10} & 3/\sqrt{10} \\ 3/\sqrt{10} & -1/\sqrt{10} \end{bmatrix}$$

$$R = \begin{bmatrix} \sqrt{10} & 14/\sqrt{10} \\ 0 & \sqrt{10}/5 \end{bmatrix}$$

MATLAB here:

Pros:

Much more accurate than classical.
Also simple to calculate
Always exists.

Cons:

Just as slow as classical
Not optimal for linear systems.

```

1 function [Q,R] = ModifiedGS(A)
2     [m,n] = size(A);
3     Q = zeros(m,n);
4     R = zeros(n,n);
5     for j=1:n
6
7         vj = A(:,j);
8         R(j,j) = norm(A(:,j));
9         Q(:,j) = A(:,j) / R(j,j);
10        for i=1:j-1
11            R(i,j) = Q(:,i)'*vj;
12            vj = vj - R(i,j)*Q(:,i);
13        end
14    end
15 end
16

```

This algorithm can be applied practically and is much more stable than the classical GS. However, we can still do better.

iii) Iterated Gram-Schmidt

This algorithm uses the fact that the product of two upper triangular matrices is upper triangular and use the modified GS process repeatedly on Q .

$$A = Q_1 R_1 = Q_2 R_2 R_1 = \dots = Q_k R_k R_{k-1} \dots R_2 R_1 \\ = Q_k R$$

We can stop iterating by setting an upper bound on

$$\|Q_k^* Q_k - I\|_F.$$

Example:

Suppose a computer is trying to get a QR factorization of

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ with } \|Q^* Q - I\|_F \leq 0.5$$

First Iteration:

Say the computer performs the GS process and gets

$$Q_1 = \begin{bmatrix} 0.3 & 0.9 \\ 0.9 & -0.3 \end{bmatrix}$$

$$R_1 = \begin{bmatrix} 3.1 & 4.4 \\ 0 & 0.6 \end{bmatrix}$$

Then,

$$\|Q_1^* Q_1 - I\|_F = 0.86 > 0.5$$

so, the computer iterates again

$$Q_1 = Q_2 R_2 = \begin{bmatrix} 0.31 & 0.95 \\ 0.95 & -0.31 \end{bmatrix} \begin{bmatrix} 0.95 & -0.19 \\ 0 & -0.38 \end{bmatrix}$$

$$\|Q_2^* Q_2 - I\|_F = 2.9 \times 10^{-16} < 0.5$$

So the computer returns

$$Q = \begin{bmatrix} 0.31 & 0.95 \\ 0.95 & -0.31 \end{bmatrix}$$

$$R = R_2 R_1 = \begin{bmatrix} 0.95 & -0.19 \\ 0 & -0.38 \end{bmatrix} \begin{bmatrix} 3.1 & 4.4 \\ 0 & 0.6 \end{bmatrix}$$

$$= \begin{bmatrix} -2.94 & -4.06 \\ 0 & -0.22 \end{bmatrix}$$

MATLAB here

Pros:

We can get as accurate as we want.

Reuses an algorithm we are familiar with.

We can choose to iterate with whichever algorithm that gets $A = QR$.

Cons:

The more iterations, the more time is spent.

We are sacrificing cost for accuracy.

This algorithm is best for purposes that require high precision as, lowering the error of $\|Q^*Q - I\|_F$ makes the algorithm as precise as the user needs.

```

1 function [Q,R, iterations] = IteratedGS(A)
2     tol = 10^-15;
3     s = size(A);
4     [Q,R] = ClassialGS(A);
5     iterations = 1;
6     while norm(Q'*Q-eye(s), 'fro') < tol
7         [Q2,R2] = ClassialGS(Q);
8         Q = Q2;
9         R = R2 * R;
10        iterations = iterations + 1;
11    end
12 end

```

iV) Householder Reflections

This is another method that computes $A = QR$. It is more stable than Gram-Schmidt but is not iterable.

It uses a sequence of unitary matrices to form a triangular matrix. This is referred to as orthogonal triangularization.

Iterated GS

$$A = Q_n R_n \cdots Q_1 R_1$$

where all R_i are upper triangular and product of upper triangulars is upper triangular

Householder

$$Q_n \cdots Q_2 Q_1 A = R$$

where all the Q_i are unitary and the product of unitaries is unitary.

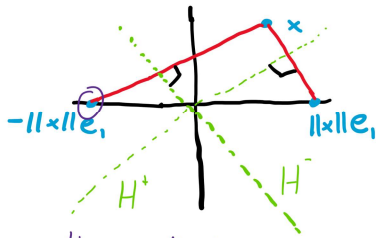
Visualization of the algebra:

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{Q_1} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{Q_2} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{Q_3} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \end{bmatrix}$$

$A \qquad Q_1 A \qquad Q_2 Q_1 A \qquad Q_3 \cdots$

$$\begin{array}{c} \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix} \\ A \end{array} \quad \begin{array}{c} \begin{bmatrix} 0 & x & x \\ & & \end{bmatrix} \\ Q_1 A \end{array} \quad \begin{array}{c} \begin{bmatrix} 0 & 0 & x \\ & & \end{bmatrix} \\ Q_2 Q_1 A \end{array} \quad \begin{array}{c} \begin{bmatrix} 0 & 0 & 0 \\ & & \end{bmatrix} \\ Q_3 Q_2 Q_1 A \\ \parallel \\ R \end{array}$$

Geometric visualization:



choose the one that moves x the most for stability.

Example:

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

$$v_1 = \begin{bmatrix} 1 \\ 3 \end{bmatrix} - \sqrt{1^2 + 3^2} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 - \sqrt{10} \\ 3 \end{bmatrix}$$

$$Q = I - 2 \frac{v_1 v_1^*}{v_1^* v_1} = I - \begin{bmatrix} 1 + 1/\sqrt{10} & -3/\sqrt{10} \\ -3/\sqrt{10} & 1 - 1/\sqrt{10} \end{bmatrix}$$

$$= \begin{bmatrix} 1/\sqrt{10} & 3/\sqrt{10} \\ 3/\sqrt{10} & -1/\sqrt{10} \end{bmatrix}$$

$$R = QA = \begin{bmatrix} 1/\sqrt{10} & 3/\sqrt{10} \\ 3/\sqrt{10} & -1/\sqrt{10} \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{10} & 14/\sqrt{10} \\ 0 & \sqrt{10}/5 \end{bmatrix}$$

MATLAB here

```

1 function [Q,R] = apply_HH(A)
2
3 [m,n] = size(A);
4 Q = eye(m,m);
5
6 for k = 1 : min(m-1,n) % Loop over all the cols
7
8     ak = A(k:end, k); % kth col and grab rows k to n
9     e1 = [1; zeros(m-k,1)];
10    vk = ak - norm(ak) * e1;
11
12    Hk = eye(m-k+1, m-k+1) - 2 * (vk*vk')/(vk'*vk);
13    Qk = [eye(k-1, k-1), zeros(k-1, m-k+1); zeros(m-k+1, k-1), Hk];
14    A = Qk * A; % Overwrites A for next iteration of loop
15    Q = Q * Qk;
16
17 end
18 R = A;
19 end

```

Pros:

More stable than one iteration of GS
without sacrificing cost.
Works with all matrices.

Cons:

Cannot be iterated as well as GS.

The householder reflections method is
an ingenious way to derive the QR
that is much more stable than GS.

2) The Singular Value Decomposition:

The SVD is a generalization of the
eigenvalue decomposition.

It is used to obtain the best low-rank

eigenvalue decomposition.

It can be used to obtain the best low-rank approximation of any matrix A . The SVD has many applications like image compression and analyzing voting groups.

Definition: The singular value decomposition (SVD) of $A \in \mathbb{C}^{n \times m}$ with rank r is

$$A := U \Sigma V^H$$

where $U \in \mathbb{C}^{n \times r}$, $V \in \mathbb{C}^{r \times m}$ orthonormal and $\Sigma \in \mathbb{C}^{r \times r}$ diagonal with positive entries

Definition:

The columns of U are called left singular vectors of A

The columns of V are called right singular vectors of A

The diagonal entries of Σ are called singular values of A .

Singular values are in decreasing order.

How to calculate SVD:

Reminder: The eigenvalue decomposition of A is

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

First, find the eigenvectors $\vec{u}$ of AA^* and $\vec{v}$ of A^*A .

$$AA^* = (U \Sigma V^*)(V \Sigma^* U^*) = U \Sigma \Sigma^* U^*$$

and $A^*A = V \Sigma^* \Sigma V$

since $UU^* = I = VV^*$ since U, V are orthonormal.

So, U is the eigenvector matrix for AA^* and V for A^*A .

Then, normalize all of the vectors and write.

$$A = \sigma_1 \vec{u}_1 \vec{v}_1^* + \sigma_2 \vec{u}_2 \vec{v}_2^* + \dots + \sigma_n \vec{u}_n \vec{v}_n^*$$

where σ_i is the square root of the equal eigenvalues of AA^T and $A^T A$.

Finally,

→

Finally,

$$A = (\vec{u}_1, \vec{u}_2, \dots, \vec{u}_m) \begin{pmatrix} \sigma_1 & & \\ & \sigma_2 & \\ & & \ddots \\ & & & \sigma_r \end{pmatrix} \begin{pmatrix} \vec{v}_1^* \\ \vec{v}_2^* \\ \vdots \\ \vec{v}_n^* \end{pmatrix}$$

Example:

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

$$AA^* = \begin{bmatrix} 5 & 11 & 17 \\ 11 & 25 & 39 \\ 17 & 39 & 61 \end{bmatrix} \quad A^*A = \begin{bmatrix} 35 & 44 \\ 44 & 56 \end{bmatrix}$$

Compute eigenvectors using

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

$$\tilde{U} = \begin{bmatrix} -0.40 & 0.88 & 0.23 \\ 0.81 & 0.24 & 0.52 \\ -0.41 & 0.40 & 0.82 \end{bmatrix} \begin{matrix} \text{eigenvalues} \\ \text{are} \\ 0, 0.26, 90.73 \end{matrix}$$

$$\tilde{V} = \begin{bmatrix} -0.78 & 0.62 \\ 0.62 & 0.78 \end{bmatrix} \text{ eigenvalues are } 0.26, 90.73$$

Notice how the nonzero eigenvalues are the same!

Then, by ordering

$$\sigma_1 = \sqrt{90.73} = 9.53 \quad \sigma_2 = \sqrt{0.26} = 0.51$$

Now for U, V , ^{change} their order to match the singular values

$$U = \begin{bmatrix} -0.23 & 0.88 & 0.41 \\ -0.52 & 0.24 & -0.82 \\ -0.81 & -0.40 & 0.41 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 9.53 & 0 \\ 0 & 0.51 \\ 0 & 0 \end{bmatrix}$$

$$V^* = \begin{bmatrix} -0.62 & -0.78 \\ -0.78 & 0.62 \end{bmatrix}$$

MATLAB here

Pros:

Generalizes eigenvalue decomposition for all matrices

Extremely applicable

Cons:

Slow for solving linear systems

The SVD is really good for saving storage as we will now see.

Truncated SVD and Approximation:

Definition: the truncated SVD of A
or k-rank SVD denoted A_k is

$$A_k = \sum_{i=1}^k \tilde{u}_i \sigma_i v_i^*$$

The idea is that if σ_i is small, we may
discard the term without much change.
This is a good estimate of A with less

This is a good estimate of A with less data. In fact, it is the best rank k approximation of A .

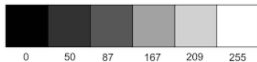
Theorems (Eckart-Young Theorem)

$$\|A - A_k\|_2 = \min_{B: \text{rank}(B)=k} \|A - B\|_2$$

So, the truncated SVD is the best rank k approximation of A !

Image compression:

Computers store images as matrices where each pixel is represented by a number.



The truncated SVD can approximate this image and create a new image with less storage:

MATLAB:

```

3  % read image
4  A = imread('SVD Picture.png'); %% Use any image here !!
5
6  % convert image to double precision format
7  A = im2double(A);
8  R = A(:, :, 1);
9  B = A(:, :, 2);
10 G = A(:, :, 3);
11
12 % compute Ak, the truncated SVD of A, for all k
13
14 red = compress(R);
15 blue = compress(B);
16 green = compress(G);
17
18 new_image = cat(3, red, blue, green);
19 imshow(new_image)
20
21
22 function B = compress(A)
23     % compute svd of A
24     [U,S,V] = svd(A);
25
26     % initialize Ak
27     Ak = zeros(size(A));
28
29     % compute truncated SVD here
30     for i=1:250
31         Ak = Ak + S(i, i) * U(:,i) * V(i,i)';
32         % add the i-th column
33     end
34     B = Ak

```

MATLAB:

```

30= for i=1:250
31=     Ak = Ak + S(i, i) * U(:,i) * V(:,i)';
32=     %add the ith column
33= end
34= B = Ak
35= end

```

Picture:

Voting Tendencies:

We can use the SVD to compare voting tendencies by turning voting data into a matrix.

Picture

Picture

The columns are bill numbers, the rows are senators, and entries are 1 for "yes", -1 for "no", and 0 otherwise

Perform the SVD on this matrix.

Plot the first two singular vectors.

Picture

This is all my own work and more can be found on



	0013	0014	0002	0001	...	2036	1969	1666	1661
Berdiel Rivera, Luis A.	1.0	1.0	1.0	1.0	...	0.0	0.0	0.0	0.0
Bhatia Gautier, Eduardo	1.0	1.0	1.0	-1.0	...	1.0	1.0	1.0	1.0
Correa Rivera, Eric	1.0	1.0	0.0	1.0	...	0.0	0.0	0.0	0.0
Cruz Santiago, Nelson V.	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Dalmau Ram��rez, Juan	-1.0	1.0	1.0	-1.0	...	0.0	0.0	0.0	0.0
Dalmau Santiago, Jos�� L.	1.0	1.0	1.0	-1.0	...	1.0	1.0	1.0	1.0
Laboy Alvarado, Zo��	1.0	1.0	1.0	1.0	...	0.0	0.0	0.0	0.0
Laureano Correa, Miguel A.	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
L��pez Le��n, Rossana	-1.0	1.0	1.0	-1.0	...	0.0	0.0	0.0	0.0
Mart��nez Santiago, Angel	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Mu��iz Cort��s, Luis Daniel	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Nadal Power, Jos�� R.	1.0	1.0	1.0	-1.0	...	0.0	0.0	0.0	0.0
Nazario Qui��z��nes, Abel	1.0	1.0	0.0	1.0	...	0.0	0.0	0.0	0.0
Neumann Zayas, Henry	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Nolasco Santiago, Margarita	1.0	1.0	1.0	1.0	...	0.0	0.0	0.0	0.0
Padilla Alvelo, Migdalia	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Pe��za Ram��rez, Itzamar	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
Pereira Castillo, Miguel A.	-1.0	-1.0	1.0	-1.0	...	0.0	0.0	0.0	0.0
PA��rez Rosa, Jose	0.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0
RA��os Santiago, Carmelo J.	1.0	1.0	-1.0	1.0	...	1.0	1.0	1.0	1.0



This is all my own work and more
can be found on
Links here

<https://www.dam.brown.edu/drp/proposals/ArturoOrtizSanMiguel.pdf>
<https://github.com/arturoortiz/SenatorsPR>

Each point represents a senator. The
closer two points are, the more similarly
those senators vote.

This works because the first two
singular values are much larger than
the rest

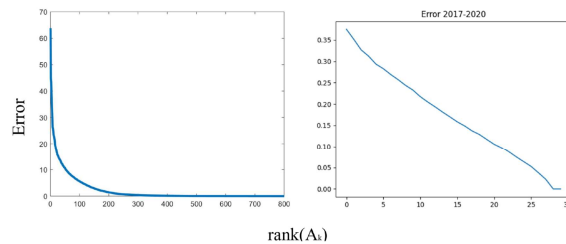
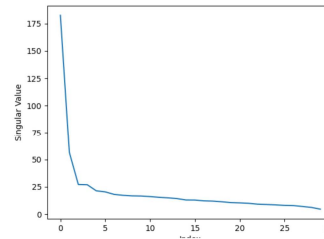
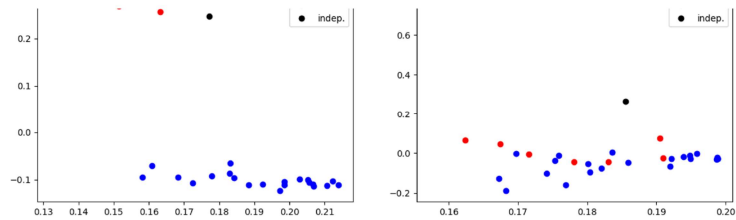
Picture

Approximation Errors:

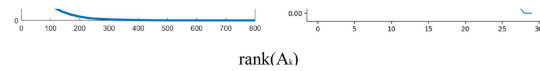
An interesting thing to note is the
error graphs of image compression
(left) and voting (right)

Picture here.

This happens because the pixels surrounding
any pixel in an image are very similar in color
while a vote in the senate is not.



any pixel in an image are very similar in color while a vote in the voting matrix has no correlation so, one cannot give a good approximation like one would for an image.



3) LU Decomposition:

This is used to solve linear systems quickly and accurately.

Definition: The LU decomposition of A is

$$A = LU$$

where L is lower triangular and U is upper triangular.

The process is simply doing Gaussian Elimination and thinking of row operations as matrices.

So,

$$\underline{L_{m-1} L_{m-2} \dots L_2 L_1} A = U$$

$$\underbrace{L_{m-1} L_{m-2} \dots L_2 L_1}_{L^{-1}} A = u$$

similar to Householder reflectors:

$$\begin{aligned} \left[\begin{array}{ccc|c} x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{array} \right] &\xrightarrow{L_1} \left[\begin{array}{ccc|c} x & x & x & x \\ o & x & x & x \\ o & x & x & x \end{array} \right] &\xrightarrow{L_3} \left[\begin{array}{ccc|c} x & x & x & x \\ o & o & x & x \\ o & o & x & x \end{array} \right] &\xrightarrow{L_2} \left[\begin{array}{ccc|c} x & x & x & x \\ o & o & o & x \\ o & o & o & x \end{array} \right] \\ A & L_1 A & L_2 L_1 A & L_3 L_2 L_1 A \\ & & & \parallel \\ & & & U \end{aligned}$$

This can be described as triangular triangulation

Example:

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

$$L_{11}u_{11} + \underbrace{L_{12}u_{21}}_{=0} = 1$$

$$L_{11}u_{12} + L_{12}u_{22} = 2$$

$$L_{21}u_{11} + L_{22}u_{12} = 3$$

$$L_2, u_{12} + L_{22} u_{22} = 4$$

$$L_{21}U_{12} + \underbrace{L_{22}U_{22}}_0 = 4$$

$$L = \begin{bmatrix} 1 & 3 & 0 \\ 1 & -2 & \end{bmatrix} \quad U = \begin{bmatrix} 3 & 6 \\ 0 & 1 \end{bmatrix}$$

is an LU decomposition of A.

MATLAB Here

Pros:

Much faster than QR and SVD.

Simple and deterministic

Very stable with pivoting.

Useful for solving $Ax = b$

Useful for computing A^{-1}

Cons:

Slower with pivoting

unstable without pivoting.

Not as usable as QR or SVD.

LU decomposition is best used to solve linear systems and Gaussian elimination is very simple.

4) Cholesky Decomposition:

This can be used in special cases where we know A is Hermetian positive definite.

...which makes this algorithm much faster

```

1 function [L,U] = LUDecomp(A)
2 [m,n] = size(A);
3 U = A;
4 L = eye(size(A));
5
6 for k = 1:m-1
7     for j=k+1:m
8
9         L(j,k) = U(j,k)/U(k,k);
10
11
12     end
13 end
14
15 end

```

we know A is Hermitian positive definite, which makes this algorithm much faster than LU.

Definition: The Cholesky factorization of A Hermitian, positive definite is

$$A = R^* R$$

where R is upper triangular.

This algorithm uses symmetric Gaussian elimination repeatedly

$$A = \begin{bmatrix} a_{11} & w^* \\ w & k \end{bmatrix} \text{ and let } \alpha = \sqrt{a_{11}}$$

$$= \begin{bmatrix} \alpha & 0 \\ w/\alpha & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & k - ww^*/a_{11} \end{bmatrix} \begin{bmatrix} \alpha & w^*/\alpha \\ 0 & I \end{bmatrix}$$

$$= R_1^* A_1 R_1$$

and then repeat for submatrix k to get

$$A = R_1^* A_1 R_1 = R_1^* R_2^* A_2 R_2 R_1 = \dots$$

$$= R_1^* R_2^* \dots R_n^* R_n \dots R_2 R_1 = R^* R$$

Example

$$A = \begin{bmatrix} 3 & i \\ -i & 3 \end{bmatrix}$$

$$\tilde{R}_{2,2} = A_{2,2} - A_{1,2} \bar{A}_{1,2} / A_{2,2}$$

$$= 3 - (i)(-i)/(3) = 3 - \frac{1}{3} = \frac{8}{3}$$

$$R_1^T = A_1^T / \sqrt{A_{11}} = [3 \ i] / \sqrt{3}$$

$$R_{2,2} = \tilde{R}_{2,2} / \sqrt{\tilde{R}_{2,2}} = (8/3) / \sqrt{8/3} = \sqrt{8}/\sqrt{3}$$

$$R = \begin{bmatrix} \sqrt{3} & i/\sqrt{3} \\ 0 & \sqrt{8}/\sqrt{3} \end{bmatrix}$$

MATLAB here.

Pro:

Hyperbolic...

```

1 function R = Cholesky(A)
2
3 [m,n] = size(A);
4 R = zeros(n,n);
5
6 for i = 1:n
7
8     R(i,i) = sqrt(A(i,i) - R(1:i-1,i)' * A(1:i-1,i));
9
10
11     for j = i+1:n
12
13         R(i,j) = (A(i,j) - R(1:i-1,i)' * R(1:i-1,j)) / R(i,i);

```

Pros:

Hyperefficient since everything is symmetric.

Very stable.

Good for solving $A\vec{x} = \vec{b}$

Cons:

Must have A hermitian positive definite which is a big specification.

The Cholesky factorization is the fastest of the four discussed but is the least applicable.

Part II: Approximations and Solutions for Linear Systems:

1) Linear Least Squares:

The solution to the least squares problem is

$$\min_{\vec{x}} \|A\vec{x} - \vec{b}\|$$

This is $A\vec{x} = \vec{b}$ but when there is no solution, we pick the closest possible $\vec{x}$.

```
11  
12-  
13-  
14-  
15-  
16-  
R(i,j) = (A(i,j) - R(1:i-1,i)' * R(1:i-1,j)) / R(i,i);  
end  
end  
end
```

This is $A\vec{x} = \vec{b}$ but when there is no solution, we pick the closest possible $\vec{x}$.

This can be solved using the normal equations and using the pseudo inverse. However the QR factorization and SVD

is also a powerful tool here.

Example via QR

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \vec{b} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

$$Q = \begin{bmatrix} 1/\sqrt{10} & 3/\sqrt{10} \\ 3/\sqrt{10} & -1/\sqrt{10} \end{bmatrix}$$

$$R = \begin{bmatrix} \sqrt{10} & 14/\sqrt{10} \\ 0 & \sqrt{10}/5 \end{bmatrix}$$

$$\vec{Qb} = \begin{bmatrix} 1/\sqrt{10} & 3/\sqrt{10} \\ 3/\sqrt{10} & -1/\sqrt{10} \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 7.27 \\ 2.85 \end{bmatrix}$$

Solve $R\vec{x} = \vec{Qb}$

$$\begin{bmatrix} \sqrt{10} & 14/\sqrt{10} \\ 0 & \sqrt{10}/5 \end{bmatrix} \vec{x} = \begin{bmatrix} 7.27 \\ 2.85 \end{bmatrix}$$

$$\begin{bmatrix} \cdot & \cdot \\ 0 & \sqrt{10}/5 \end{bmatrix} \vec{x} = \begin{bmatrix} \cdot \\ 2.85 \end{bmatrix}$$

$$\vec{x} = \begin{bmatrix} -4 \\ 9/2 \end{bmatrix}$$

MATLAB here

Pros:
Different ways of solving lead to new insights

Cons:
Slow with QR and SVD.

2) Forward/Backward Substitution:

A way to solve $A\vec{x} = \vec{b}$ using the LU factorization.

$$LU\vec{x} = \vec{b}$$

$$L\vec{y} = \vec{b} \quad \text{forward substitution}$$

$$U\vec{x} = \vec{y} \quad \text{Backward substitution}$$

```
1 function x = leastsquares(A, b)
2
3     [Q,R] = qr(A);
4     v = Q*b;
5     x = v\R;
6
7 end
```

Example

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

$$LU = \begin{bmatrix} 1/3 & 0 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} 3 & 6 \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1/3 & 0 \\ 1 & -2 \end{bmatrix} \tilde{y} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

$$y_1 = b_1 / L_{11} = 5 / (1/3) = 15$$

$$y_2 = (b_2 - L_{21}x_1) / L_{22} = \frac{6 - (1)15}{-2} = \frac{11}{2}$$

$$y = \begin{bmatrix} 15 \\ 11/2 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 6 \\ 0 & 1 \end{bmatrix} \tilde{x} = \begin{bmatrix} 15 \\ 11/2 \end{bmatrix}$$

$$x_1 = b_1 / u_{11} = 5/3$$

$$x_2 = b_2 - u_{21}x_1 / u_{22} = \frac{6 - 6 \cdot 15}{1} = -84$$

$$x = \begin{bmatrix} 5/3 \\ -84 \end{bmatrix}$$

MATLAB here

Pros:

used over QR because it's 2x faster and very accurate.

Been used for a long time since it was one of the first ways to solve $Ax = b$.

Cons:

Unstable without pivoting.

Using forward/backward substitution has been around for a really long time and is still very practical.

3) Matrix Inversion

The LU decomposition gives us a nice

```
function x = forwardbackward(A, b)

[L, U, P] = lu(A);
[m, n] = size(A);
b = P*b;
% Forward substitution Ly = b

y = zeros(n,1);

y(1) = b(1) / L(1,1);

for i = 1:n

    y(i) = (b(i) - L(i,1:i-1)*y(1:i-1)) / L(i,i);
end

% Backward substitution Ux = y

x = zeros(n,1);
x(n) = y(n) / U(n,n);

for i = n-1:-1:1 % from (n-1) to 1 decreasing

    x(i) = (y(i) - (U(i,i+1:end)*x(i+1:end))) / U(i,i);
end
end
```

...

The LU decomposition gives us a nice way to find A^{-1} , which can be very useful.

First get $A = LU$

then compute

$$A^{-1} = U^{-1}L^{-1}$$

U^{-1} , L^{-1} are easy to compute since they are triangular.

Example:

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

$$LU = \begin{bmatrix} 1/3 & 0 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} 3 & 6 \\ 0 & 1 \end{bmatrix}$$

$$L^{-1} = \begin{bmatrix} 3 & 0 \\ 3/2 & -1/2 \end{bmatrix} \begin{array}{l} L_{11}^{-1} = 1/L_{11} \\ L_{22}^{-1} = 1/L_{22} \\ L_{12}^{-1} = \frac{1}{L_{11} - L_{22}} \end{array}$$

$$U^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$U^{-1} = \begin{bmatrix} 1/3 & -2 \\ 0 & 1 \end{bmatrix} \quad L_{11} - L_{22}$$

$$\begin{bmatrix} 1/3 & -2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 5/2 & -1/2 \end{bmatrix} = \begin{bmatrix} -2 & 1 \\ 3/2 & -1/2 \end{bmatrix}$$

MATLAB here

Pros:

This algorithm is very precise and efficient

Cons:

However, it is unstable without pivoting which would add more cost.

This is a brilliant shortcut to find A^{-1} .

4) Low Rank Approximation:

This was discussed in Part I (2)(i)

Part III: Finite Precision and

```

1 function Ainv = inverse(A)
2
3 [m,n] = size(A)
4 [L,U] = LUDecomp(A);
5
6 Linv = zeros(n,n);
7 Uinv = zeros(n,n);
8
9 Linv(1,1) = 1/L(1,1);
10 Linv(2,2) = 1/L(2,2);
11 Linv(1,2) = 1/(L(1,1)-L(2,2));
12
13 Uinv(1,1) = 1/U(1,1);
14 Uinv(2,2) = 1/U(2,2);
15 Uinv(2,1) = 1/(U(1,1)-U(2,2));
16
17
18 Ainv = Uinv*Linv;
19 end

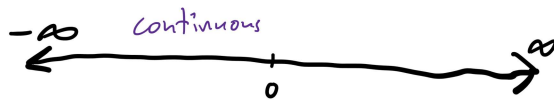
```

Part III: Finite Precision and Matrix conditioning.

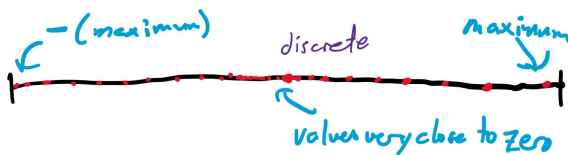
1) Finite Precision Number System.

Computers have a limited amount of storage, so, they cannot have an infinite amount of numbers. The finite precision system is what is used. It is what causes errors in computer computation.

Real number line



Finite precision number line



Definition:

Definition:

A floating point system is one where

β : the base

t : the precision

$e_{\min}$: minimum exponent

$e_{\max}$: maximum exponent

$$F = \{ \pm m \beta^e : \text{either}$$

$$\beta^{t-1} \leq m < \beta^t \text{ and } e_{\min} \leq e \leq e_{\max}$$

$$\text{or } 0 \leq m < \beta^{t-1} \text{ and } e = e_{\min} \}$$

Most modern computers use base $\beta = 2$
 t affects how dense the red dots are in the line above.

$e_{\min}, e_{\max}$ control smallest and largest values in the system

m allows us to specify significant digits

```
>> 0.1 + 0.1 + 0.1 - 0.3
```

```
ans =
```

m allows us to specify significant digits.

MATLAB

In Matlab, $\beta=2$ $t=53$

$$e_{\min} = -1021$$

$$e_{\max} = 1024$$

$$\text{range is } 10^{\pm 308}$$

$$\text{and } \epsilon_{\text{mach}} = 2^{-53} = 2.22 \times 10^{-16}$$

2) Conditioning

Since computers use a different number system, we must study how accurate they can be in our number systems

Definition: The condition number

$K(A)$ of A is

$$K(A) = \lim_{\Delta \rightarrow 0} \max_{\| \delta y \| \leq \Delta} \frac{\| \delta f \| / \| f \|}{\| \delta y \| / \| y \|}$$
$$= \| A^{-1} \| \| A \|$$

```
>> 0.1 + 0.1 + 0.1 - 0.3
```

```
ans =
```

```
5.551115123125783e-17
```

```
>> factorial(200)
```

```
ans =
```

```
Inf
```


$$= \|A^{-1}\| \|A\|$$

A lower condition number means that a small change in y will make Ay have a small change.

Example of an ill-conditioned matrix is the Hilbert matrix

$$H \text{ such that } H_{ij} = \frac{1}{i+j-1}$$

$$\text{hilb}(3) = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

$$\kappa(\text{hilb}(3)) = 524.06$$

Example of a well conditioned matrix

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \kappa(I) = 1$$

MATLAB ex. of bad system.

```
>> norm(forwardbackward(hilb(200),ones(200))-hilb(200)\ones(200))  
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 9.018146e-22.
```

```
ans =
```

```
7.892734119635136e+06
```

```
1 % Tests for computational notebook. All MATLAB examples for the code above is here  
2 A = [1,2,3;4,5,6;7,8,9];  
3  
4 ClassialGS(A)  
5 ClassialGS(hilb(3))  
6  
7 ModifiedGS(A)  
8 ModifiedGS(hilb(3))  
9  
10 IteratedGS(A)  
11 IteratedGS(hilb(3))  
12  
13 apply_HH(A)  
14 apply_HH(hilb(3))  
15  
16 LUDecomp(A)  
17 LUDecomp(hilb(3))  
18  
19  
20 H = [5, 1; -1, 5];  
21 Cholesky(H)  
22 Cholesky(H)'  
23  
24 b = [5;6;7];  
25 leastsquares(A,b)  
26 forwardbackward(A,b)  
27  
28 C = [1,2;3,4];  
29 inverse(C)  
30
```