Math 170A Lecture Notes (Professor: Lei Huang)

Isabelle Mills

January 27, 2024

Week 1 Notes (1/8 - 1/12/2024)

For this class, we shall define the number of flops an algorithm takes as the number of individual $+$, $-$, $\times$, $/$, and $\sqrt{\ }$ operations on real numbers used in the algorithm.

> For example: taking the inner product of two vectors $\vec{u} = (u_1, u_2, \ldots, u_n)$ and $\vec{v} = (v_1, v_2, \ldots, v_n)$ requires n multiplications and $(n-1)$ additions. So we'll say it has a flops count of $2n - 1$.

Technically, the word "flop" stands for floating (point) operation. Based on that knowledge, hopefully it is easier to guess what is and is not a flop. For instance, observe the code written below for taking an inner product of two n-vectors.

```
1   P = 0
2   for i = 1:n
3      P = P + v(i) * w(i)
4   end
```

Neither incrementing `i` nor initializing any other variables are counted towards the flop number. Because the code does `n` additions and multiplications between floating point numbers, we say this function has $2n$ flops.

Here is how we formally define Big-O Notation:

For a sequence a_n, we define $a_n = O(b_n)$ if there exists real constants $C, N \geq 0$ such that for $n \geq N$, $a_n \leq Cb_n$.

Example problem:

```
1       function x = lowertriangsolve(L, b)
2          N = size(L);
3          n=size(b,1);
4          x=b;
5
6          for i=1:N(1):
7             for j=1:i-1
8                x(i) = x(i) - L(i,j)*x(j);
9             end
10
11            if L(i, j) == 0
12               error('matrix is singular')
13            end
14            x(i) = x(i)/L(i,i);
15         end
16      end
```

Inside the main for loop (lines 6-15):
Line 8 has $2(\mathtt{i}-1)$ flops.
Line 14 has an additional flop.

Thus, the total number of flops is:

$$\sum_{\mathtt{i}=1}^{n}(2\mathtt{i}-1) = 2\left(\sum_{\mathtt{i}=1}^{n}\mathtt{i}\right) - n$$

This then simplifies to:

$$2\left(\frac{n(n+1)}{2}\right) - n = 1n^2$$

So `x` has $O(n^2)$ flops as `n` goes to infinity.

Lecture 4: 1/17/2024

Row operations used in Gaussian elemination can be represented via matrix multiplication as demonstrated below.

Action	Matrix representation of the operation
$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \xrightarrow{R_2 \leftrightarrow R_3} \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \xrightarrow{\frac{2}{3}R_2 \to R_2} \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ \frac{2}{3}a_{2,1} & \frac{2}{3}a_{2,2} & \frac{2}{3}a_{2,3} & \frac{2}{3}a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{2}{3} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \downarrow R_3 + \frac{2}{3}R_2 \to R_3 \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1}+\frac{2}{3}a_{2,1} & a_{3,2}+\frac{2}{3}a_{2,2} & a_{3,3}+\frac{2}{3}a_{2,3} & a_{3,4}+\frac{2}{3}a_{2,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{2}{3} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The importance of representing elementary row operations as matrices is that we can multiply these representations together to compose rows operations. Thus, these representations are central to many matrix decompositions.

For example, we can represent turning a matrix into row echelon form as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{2}{9} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & 5 \\ 6 & 3 & 4 \\ 4 & 6 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 5 \\ 0 & -9 & -11 \\ 0 & 0 & \frac{-50}{9} \end{bmatrix}$$

Here are some more observations about row operation matrices:

- Row scaling operations are represented by diagonal matrices and thus are also both lower and upper triangular.
- For $i < j$, adding a multiple of the ith row to the jth row is represented by a lower triangular matrix.
- For $i > j$, adding a multiple of the ith row to the jth row is represented by an upper triangular matrix.
- Row swaps are not represented as triangular matrices. However, they are permutation matrices.

Now note that in the normal algorithm for Gaussian elimination, assuming we never need to swap rows, all row operations will be such that their matrix representation is lower triangular.

Thus, given an invertible square matrix $\mathbf{A}$, we can represent doing Gaussian elimination on $\mathbf{A}$ by the equation: $\mathbf{L}_n\mathbf{L}_{n-1}\cdots\mathbf{L}_2\mathbf{L}_1\mathbf{A} = \mathbf{U}$ where $\mathbf{L}_i$ is a lower triangular matrix and $\mathbf{U}$ is an upper triangular matrix. Then, because the product of two lower triangular matrices is also lower triangular, we can multiply all the lower triangular matries together to get an equation of the form $\mathbf{LA} = \mathbf{U}$. Finally, we multiply both sides of the equation on the left by $\mathbf{L}^{-1}$ to get that $\mathbf{A} = \mathbf{L}^{-1}\mathbf{U}$. And as the inverse of a lower triangular matrix is also lower triangular, we know that we have decomposed $\mathbf{A}$ into the product of a lower triangular matrix and an upper triangular matrix. This algorithm is called LU decomposition.

As for why we would want to use LU decomposition, we can look at the number of flops different matrix operations require.

Assume we are given an invertible $n \times n$ matrix $\mathbf{A}$ and two vectors $\vec{x}, \vec{b} \in \mathbb{R}^n$.

Firstly note that row reduction takes approximately $\frac{2}{3}n^3$ flops while the back substitution algorithm takes approximately n^2 flops. Thus, for larger values of n, solving the matrix equation $\mathbf{A}\vec{x} = \vec{b}$ takes around $\frac{2}{3}n^3$ flops.

Now lets compare this to the number of flops the best algorithm we can make to do LU decomposition takes.

Assuming we don't need to do any row swaps, than the only elementary row operations we need to do are adding scaled rows to other rows. This is where our first optimization comes into play. The inverse of a row addition is merely a row subtraction. For example:

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

Thus, we can fairly straightforwardly get an expression of the form $\mathbf{A} = \mathbf{L}_1\mathbf{L}_2\cdots\mathbf{L}_k\mathbf{U}$ by multiplying the inverse of each elementary row operation matrix to both sides of the equation representing the actions we took to reduce $\mathbf{A}$ to $\mathbf{U}$.

Now, we apply another two shortcuts to speed up our calculations.

Firstly, observe that: $\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&a&1&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&b&0&1\end{bmatrix} = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&b&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&a&1&0\\0&0&0&1\end{bmatrix} = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&a&1&0\\0&b&0&1\end{bmatrix}$

There's nothing special about that column or that particular matrix size. In fact, this should make sense when you consider that those two elementary row operations don't effect each other.

Next, observe that: $\begin{bmatrix}1&0&0&0\\a&1&0&0\\b&0&1&0\\c&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&d&1&0\\0&e&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&f&1\end{bmatrix} = \begin{bmatrix}1&0&0&0\\a&1&0&0\\b&d&1&0\\c&e&f&1\end{bmatrix}$

To intuit why this works, think about how the matrix product has every row affect the rows underneath it before it itself is affected by any rows above it. So, it is specifically the initial state of every row that is effecting every other row in the matrix product.

The end result of these observations is that the (i,j)th element of $\mathbf{L}$ where $i > j$ is just going to be the negative of whatever coefficient one multiplied a copy of row i by before then adding that to row j as part of doing the row reduction algorithm. But now note that means that every element in $\mathbf{L}$ can be directly extracted from the calculations done to find $\mathbf{U}$. So finding $\mathbf{U}$ takes approximately $\frac{2}{3}n^3$ flops and finding $\mathbf{L}$ takes no additional flops.

Once, we've found $\mathbf{LU}$, we can then use back substitution twice to solve any matrix vector equation $\mathbf{A}\vec{x} = \mathbf{LU}\vec{x} = \vec{b}$. This will take $2n^2$ flops.

Technically, this means that if you are trying to solve a single matrix vector equation $\mathbf{A}\vec{x} = \vec{b}$ by first decomposing $\mathbf{A}$, then it will take longer than if you just solved it directly. However, if you have many matrix vector equations involving $\mathbf{A}$, then you can work way faster by decomposing $\mathbf{A}$. This is because once you decompose $\mathbf{A}$ once, you can just store $\mathbf{L}$ and $\mathbf{U}$ for later use. Thus, every subsequent matrix vector equation involving $\mathbf{A}$ can be done in approximately $2n^2$ flops instead of taking approximately $\frac{2}{3}n^3$ flops.

Lecture 5: 1/19/2024

A crucial assumption we made in the last lecture was that we never would have to do row swaps when doing row reduction. In this lecture, we'll now allow ourselves to do row swaps.

Firstly, here are two general reasons to want to do row swaps.

- Firstly, sometimes we have no choice.

$$\begin{bmatrix} 0 & 4 & 1 \\ 1 & 3 & 4 \\ 2 & 2 & 5 \end{bmatrix}$$

For this matrix, if we were to divide the second and third row by $a_{1,1}$, we'd be dividing them by 0. Thus, we clearly can't do that.

- Secondly, dividing by small numbers causes more roundoff erros. So, we can slow the accumulation of roundoff erros by swapping rows in order to prioritize dividing by larger elements.

So here's an algorithm called parial pivoting for taking a PLU decomposition of an invertible $n \times n$ matrix $\mathbf{A}$.

When doing Guassian elimination, for every new pivot of $\mathbf{A}$, first perform a row swap with the top non-reduced row and the non-reduced row with the largest would-be pivot element. Only, after that do you then add a scaled version of the pivot row to the other rows like you were doing before.

Now note that performing the same row swap twice in a row is equivalent to doing nothing. Thus, every matrix representing a row swap is its own inverse.
Additionally, as we already covered, the inverse of a row addition operation is just a row subtraction operation. Because of this, we can easily get an equation for A as the product of many elementary triangular matrices and permutation matrices.

Let's denote $\mathbf{L}_i$ to be the lower triangular matrix representing all row additions of row i to the rest of the matrix. In other words, every element below the main diagonal in $\mathbf{L}_i$ will be nonzero except for the elements in column i. Additionally, let's denote $\mathbf{P}_{i,j}$ to be the permutation matrix swapping row i and row j. Then, we can say that:

$$\mathbf{A} = \mathbf{P}_{1,k_1}\mathbf{L}_1\mathbf{P}_{2,k_2}\mathbf{L}_2 \cdots \mathbf{P}_{n-1,k_{n-1}}\mathbf{L}_{n-1}\mathbf{U}$$

(Note that $i < k_i \leq n$ since it doesn't make sense to swap a row that has already been dealt with.)

We can rewrite this as $\mathbf{P}_{1,k_1}\mathbf{A} = \mathbf{L}_1\mathbf{P}_{2,k_2}\mathbf{L}_2 \cdots \mathbf{P}_{n-1,k_{n-1}}\mathbf{L}_{n-1}\mathbf{U}$. And now here is where the magic starts. If we multiply both sides of that equation on the left by $\mathbf{P}_{2,k_2}$, then we can say that $(\mathbf{P}_{2,k_2}\mathbf{L}_1\mathbf{P}_{2,k_2}) = \mathbf{L}_1'$ where $\mathbf{L}_1'$ is the matrix which would have resulted if we had only applied the permutation $\mathbf{P}_{2,k_2}$ to the elements off the main diagonal.

Now we can do the same process again to $(\mathbf{L}_1'\mathbf{L}_2)\mathbf{P}_{3,k_3}$, multiplying both sides of our equation by $\mathbf{P}_{3,k_3}$ and then saying that $\mathbf{L}_2' = (\mathbf{P}_{3,k_3}\mathbf{L}_1'\mathbf{L}_2\mathbf{P}_{3,k_3})$ is a lower triangular matrix whose only nonzero elements below the diagonal are in columns 1 and 2. Doing this for all remaining permutation matrices on the right side of the equation, we will eventually get an equation of the form:

$$\mathbf{P}_{n-1,k_{n-1}} \cdots \mathbf{P}_{2,k_2}\mathbf{P}_{1,k_1}\mathbf{A} = \mathbf{LU}$$

Finally, by multiplying those permutation matrices together, we get an equation:

$$\mathbf{PA} = \mathbf{LU}$$

Here is why the product $\mathbf{P}_{i+1,k_{i+1}}\mathbf{L}_{i-1}'\mathbf{L}_i\mathbf{P}_{i+1,k_{i+1}}$ was equivalent to only applying the row permutation underneath the diagonal of $\mathbf{L}_{i-1}'\mathbf{L}_i$.

Observe the following diagram of where the permutation matrices send the elements in row $i+1$, row k_{i+1}, column $i+1$, and column k_{i+1} of the matrix $\mathbf{L}_{i-1}'\mathbf{L}_i$:

$$\begin{bmatrix} & & 0 & & 0 & & \\ & & 0 & & 0 & & \\ a & b & 1 & 0 & 0 & 0 & 0 \\ & & 0 & & 0 & & \\ & & 0 & & 0 & & \\ c & d & 0 & 0 & 1 & 0 & 0 \\ & & 0 & & 0 & & \\ & & 0 & & 0 & & \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} & & 0 & & 0 & & \\ & & 0 & & 0 & & \\ c & d & 1 & 0 & 0 & 0 & 0 \\ & & 0 & & 0 & & \\ & & 0 & & 0 & & \\ a & b & 0 & 0 & 1 & 0 & 0 \\ & & 0 & & 0 & & \\ & & 0 & & 0 & & \end{bmatrix}$$

Theorem: Every invertible matrix has a PLU-decomposition.

Lecture 6: 1/22/2024

A matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is said to be positive definite if:

1. $\mathbf{A}^T = \mathbf{A}$ ($\mathbf{A}$ is symmetric)
2. $\vec{x}^T \mathbf{A} \vec{x} = \langle \vec{x}, \mathbf{A}\vec{x} \rangle > 0$ for all vectors $\vec{x} \in \mathbb{R}^n$ with $\vec{x} \neq 0$.

$$\langle \vec{x}, \mathbf{A}\vec{x} \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i a_{i,j} x_j$$

Lemma: If $\mathbf{A}$ is positive definite, then $\mathbf{A}$ is invertible.

Proof: (we proceed towards a contradiction)

Assume there exists a vector $\vec{y} \in \mathbb{R}^n$ not equal to 0 such that $\mathbf{A}\vec{y} = 0$. Then $\langle \vec{y}, \mathbf{A}\vec{x} \rangle = \langle \vec{y}, 0 \rangle = 0$. So, $\mathbf{A}$ cannot be positive definite.

Theorem: Let $\mathbf{M} \in \mathbb{R}^{n \times n}$ be invertible. Then $\mathbf{A} = \mathbf{M}^T\mathbf{M}$ is positive definite.

Proof:

1. $\mathbf{A}^T = \left(\mathbf{M}^T\mathbf{M}\right)^T = \mathbf{M}^T \left(\mathbf{M}^T\right)^T = \mathbf{M}^T\mathbf{M} = \mathbf{A}$. So $\mathbf{A}$ is symmetric.
2. Note that $\vec{x}^T\mathbf{A}\vec{x} = \vec{x}^T\mathbf{M}^T\mathbf{M}\vec{x} = (\mathbf{M}\vec{x})^T\mathbf{M}\vec{x} = \langle \mathbf{M}\vec{x}, \mathbf{M}\vec{x} \rangle = |\mathbf{M}\vec{x}|$. Now, $|\mathbf{M}\vec{x}| > 0$ for all $\mathbf{M}\vec{x} \neq 0$. And because $\mathbf{M}$ is invertible, we know the only $\vec{x}$ such that $\mathbf{M}\vec{x} = 0$ is $\vec{x} = 0$. So, $\vec{x}^T\mathbf{A}\vec{x} > 0$ for all $\vec{x} \neq 0$.

   If $\mathbf{M}$ is not invertible, then $\mathbf{A}$ is positive semidefnite as $|\mathbf{M}\vec{x}|$ cannot be less than 0 but we can find a nonzero $\vec{x}$ such that $|\mathbf{M}\vec{x}| = 0$.

Theorem (Cholesky decomposition):

Let $\mathbf{A}$ be positive definite. Then there exists an upper triangular matrix $\mathbf{R}$ such that $\mathbf{A} = \mathbf{R}^T\mathbf{R}$. We call $\mathbf{R}$ the Cholesky factor and can calculate it as follows:

If such an $\mathbf{R}$ exists, then we can arrive at the following formula for each element of $\mathbf{A} = \mathbf{R}^T\mathbf{R}$:

$$a_{i,j} = \sum_{k=1}^{\min(i,j)} r_{k,i} r_{k,j}$$

Now as $\mathbf{A}$ must be symmetric based on our formula, we can safely ignore the elements of $\mathbf{A}$ below the main diagonal. So, assuming that $j \geq i$, we can rearrange terms to get the following equation including the element $a_{i,j}$:

$$r_{i,i} r_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} r_{k,i} r_{k,j}$$

And with that, we now have a way of expressing any element $r_{i,j}$ in terms of elements of $\mathbf{R}$ which are in previous rows or previous columns of $\mathbf{R}$. So, we can inductively solve for the coefficients of $\mathbf{R}$ as follows:

for $i = 1, \ldots, n$:

$$r_{i,i} = \sqrt{a_{i,i} - \sum_{k=1}^{i-1} r_{k,i}^2}$$

for $j = (i+1), \ldots, n$:

$$r_{i,j} = \frac{1}{r_{i,i}} \left(a_{i,j} - \sum_{k=1}^{i-1} r_{k,i} r_{k,j} \right)$$

end

end

Now it's worth noting what can go wrong in the above algorithm.

Firstly, if $a_{i,i} - \sum_{k=1}^{i-1} r_{k,i}^2 < 0$, then $r_{i,i} \notin \mathbb{R}$. So, $\mathbf{R}$ does not exist

Secondly, if $r_{i,i} = 0$, then you can't divide by it to isolate $r_{i,j}$. Thus, one of two possibilities will arise:

- If $\left(a_{i,j} - \sum_{k=1}^{i-1} r_{k,i} r_{k,j} \right) = 0$, then you can set $r_{i,j}$ to anything and maintain equality. Thus, if a Cholesky factorization exists, it is not unique.

- If $\left(a_{i,j} - \sum_{k=1}^{i-1} r_{k,i} r_{k,j} \right) \neq 0$, then there is nothing you can set $r_{i,j}$ to. So, no Cholesky factor of $\mathbf{A}$ exists.

Now unfortunately, this class is not interested in proving when and when not each of the above problems will arise. So, for now just know that:

1. $\mathbf{A}$ is positive definite if and only if $\mathbf{R}$ exists and is unique.
2. $\mathbf{A}$ is positive semidefinite if and only if $\mathbf{R}$ exists but is not invertible and thus also not unique.
3. $\mathbf{A}$ is not positive definite or semidefinite if and only if $\mathbf{R}$ does not exist.

The second theorem covered in lecture today let's us easily prove one direction in each of the above implications. So, the challenging task would be to prove the other direction, and to do that you would need to show that for any positive definite or semidefinite matrix $\mathbf{A}$, you can always use the above algorithm to find a matrix $\mathbf{R}$.

It takes approximately $\frac{1}{3}n^3$ flops to do Cholesky decomposition as n gets larger. This is notably half of what LU decomposition takes.

Lecture 7: 1/24/2024

A norm of a vector $\vec{x} \in \mathbb{R}^n$ is a real number $\|\vec{x}\|$ that is assigned to $\vec{x}$ satisfying:

- $\vec{x} \neq 0 \implies \|\vec{x}\| > 0$ whereas $\|\vec{0}\| = 0$.
- $\|c\vec{x}\| = |c|\|\vec{x}\|$ for all $c \in \mathbb{R}$.
- $\|\vec{x} + \vec{y}\| \leq \|\vec{x}\| + \|\vec{y}\|$ for all $\vec{x}, \vec{y} \in \mathbb{R}^n$

Some important vector norms:

1. Vector p-norm: For an integer $p \geq 1$, we define $\|\vec{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{\frac{1}{p}}$
2. Infinity Norm: $\|\vec{x}\|_\infty = \max\{|x_i| \mid 1 \leq i \leq n\}$

A matrix norm assigns a real value $\|\mathbf{A}\|$ to a matrix $\mathbf{A}$ satisfying:

- $\mathbf{A} \neq \mathbf{0} \implies \|\mathbf{A}\| > 0$ whereas $\|\mathbf{0}\| = 0$.
- $\|c\mathbf{A}\| = |c|\|\mathbf{A}\|$ for all $c \in \mathbb{R}$.
- $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$ for all $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$

Some important matrix norms:

1. Frobenius norm: $\|\mathbf{A}\| = \left(\sum_{i,j=1}^{n} a_{i,j}^2\right)^{\frac{1}{2}}$

   Assuming $\mathbf{A}$ is an $m \times n$ matrix, this norm is equivalent to stringing out $\mathbf{A}$'s rows or columns to form an mn element vector and then taking the 2-norm of the resulting vector.

   $$\left\| \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \right\| = \|(a, b, c, d, e, f, g, h, i)\|$$

2. Matrix p-norm: For an integer $p \geq 1$, we define:

   $$\|\mathbf{A}\|_p = \max\left\{ \frac{\|\mathbf{A}\vec{x}\|_p}{\|\vec{x}\|_p} \;\middle|\; \vec{x} \in \mathbb{R}^n \text{ and } \vec{x} \neq 0 \right\}$$

Basically, a matrix p-norm measures the maximum stretch which a linear function $L(\vec{x}) = \mathbf{A}\vec{x}$ applies to a vector relative to its starting length.