Chapter 1

Introduction to Matrices

Contents (class version)

1.0 Introduction

This chapter reviews vectors and matrices and basic *properties* like shape, orthogonality, determinant, eigenvalues and trace. It also reviews some *operations* like multiplication and transpose.

Source material for this chapter includes [1, §1.1-1.4, 2.1, 3.1, 9.1].

1.1 Basics

Why vector? L§1.1

- Convenient **data** organization
- To group into matrices (data)
  or to be acted on by matrices (operations)

Example: Personal attributes: $\begin{bmatrix} \text{age} \\ \text{height} \\ \text{weight} \\ \text{eye color} \\ \vdots \end{bmatrix}$

Example: Digital grayscale image (!)

- A vector in the vector space of 2D arrays.
- I rarely think of a digital image as a "matrix" !

(Read the Appendix on p. 1.78 about general vector spaces.)

Why matrix?

Two reasons:
- Organizing **data** as an array
- Representing a **linear operation** aka **linear map** or **linear transformation**

Example. A $M \times N$ data matrix for a group of N people with M attributes each:

$$\begin{bmatrix} \text{age}_1 & \dots & \text{age}_N \\ \text{height}_1 & \dots & \text{height}_N \\ \text{weight}_1 & \dots & \text{weight}_N \\ \text{eye color}_1 & \dots & \text{eye color}_N \\ \vdots & \dots & \vdots \end{bmatrix}$$

Example: linear operation:
N-point **DFT** matrix in 1D (DFT is a prereq term)
DFT matrix in 2D
unified as matrix-vector operation: $\underbrace{\boldsymbol{X}}_{\text{spectrum in } \mathbb{C}^N} = \underbrace{\boldsymbol{W}}_{N \times N \text{ DFT}} \underbrace{\boldsymbol{x}}_{\hookrightarrow \text{ signal in } \mathbb{C}^N}$

where in 1D case: $W_{kn} = \mathrm{e}^{-\imath 2\pi (k-1)(n-1)/N}, \; k, n = 1, \dots, N.$

Example (classic): solve **system of linear equations** with N equations in N unknowns:

$$\begin{matrix} ax_1 + bx_2 + cx_3 = u \\ dx_1 + ex_2 + fx_3 = v \\ gx_1 + hx_2 + ix_3 = w \end{matrix} \implies \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \text{ with } \boldsymbol{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, \ \boldsymbol{b} = \begin{bmatrix} u \\ v \\ w \end{bmatrix}.$$

Certainly the matrix-vector notation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is more concise (and general).

A traditional linear algebra course would focus extensively on solving $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$.
This topic will not be a major focus here!

Example: linear operation:

convolution in DSP: (convolution is a prereq term)

(**LTI** is a prereq term)

$$\underbrace{(x[0],\ldots,x[N-1])}_{\substack{\text{input signal}\\ x}} \to \overset{\text{Filter}}{\boxed{\begin{matrix}\text{LTI}\\ \underbrace{h[0],\ldots,h[K-1]}_{\text{impulse response}}\end{matrix}}} \to \underbrace{(y[0],\ldots,y[M-1])}_{\underbrace{\text{output signal}}_{y}}$$

$$\underline{y[m] = \sum_k x[k]\, h[m-k]}$$

Matrix-vector representation of convolution as a linear map:

$$y = H x$$

where $\boldsymbol{x}$ is length-N vector, $\boldsymbol{y}$ is length-M vector and $\boldsymbol{H}$ is a $\underline{M \times N}$ matrix with elements

$$H_{mn} = h[m-n] \tag{1.1}$$

1. What is M in terms of N and K? (Choose best answer.) (Neighbor intro!) (DSP prerequisite!)

$N+K-1$

A: $\max(N,K)-1$ B: $\max(N,K)$ C: $N+K$ D: $N+K+1$ [✓ E: None of these.] ??

The convolution operation is the key component of any **convolutional neural network** (**CNN**) [2] [3, p. 514].
Other matrices represent other important linear operations: wavelet transform, DCT, 2D DFT, ...

Example: data matrix (Read)

A **term-document matrix** is used in **information retrieval**.

Document1: EECS551 meets on Tuesdays and Thursdays
Document2: Tuesday is the most exciting day of the week
Document3: Let us meet next Thursday.

Keywords (terms): EECS551 meet Tuesday Thursday exciting day week next
(Note: use stem, ignore generic words "the" "and")

Term-document (binary) matrix:

Term	Doc1	Doc2	Doc3
EECS551	1	0	0
meet	1	0	1
Tuesday	1	1	0
Thursday	1	0	1
exciting	0	1	0
day	0	1	0
week	0	1	0
next	0	0	1

The entries in a (mostly) numerical table like this are naturally represented by a 8×3 **matrix** $\boldsymbol{T}$ (because each column is a vector in $\mathbb{R}^8$ and there are three columns) as follows:

Why mostly? Column and row labels...

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Query as a matrix vector operation: find all documents relevant to the query “exciting days of the week.” See example query vector to the right:

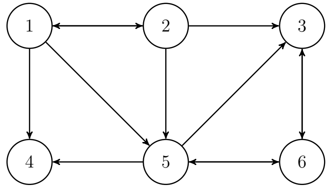
In matrix terms, find columns of $\boldsymbol{T}$ that are “close” (will be defined precisely later) to query vector $\boldsymbol{q}$.

$$\boldsymbol{q} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

This is an information retrieval application expressed using matrix/vector operations.

Example: Networks, Graphs, and Adjacency Matrices (Read)

Consider a set of six web pages that are related via web links (outlinks and inlinks) according to the following **directed graph** [4, Ex. 1.3, p. 7]:



The 6×6 **adjacency matrix** $\boldsymbol{A}$ for this graph has a nonzero value (unity) in element A_{ij} if there is a link from from node (web page) j to i:

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}, \qquad \boldsymbol{L} = \begin{bmatrix} 0 & 1/3 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 & 1/3 & 1/2 \\ 1/3 & 0 & 0 & 0 & 1/3 & 0 \\ 1/3 & 1/3 & 0 & 0 & 0 & 1/2 \\ 0 & 0 & 1 & 0 & 1/3 & 0 \end{bmatrix}.$$

The **link graph matrix** $\boldsymbol{L}$ is found from the adjacency matrix by normalizing (each column) with respect to the number of outlinks so each column sums to unity,

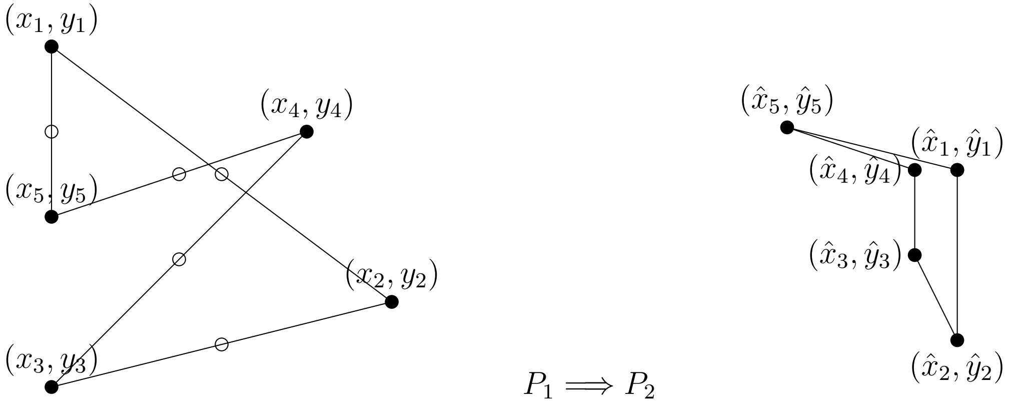
We will see later that Google's **PageRank** algorithm [5] for quantifying importance of web pages is related to an **eigenvector** of $\boldsymbol{L}$. (There is a **left eigenvector** with $\lambda = 1$ so there must also be a **right eigenvector** with that **eigenvalue** and that is the one of interest.) So evidently representing web page relationships using a **matrix** and computing properties of that matrix is useful in many domains, even some that might seem quite unexpected at first.

Example: a simple linear operation (Read)

In the preceding two examples (viewing a 2D function and computing volume under a 2D function), the matrix was a 2D array of *data*. Recall that there were two "whys" for matrices: data and *linear operations*. Now we turn to an example where the matrix is associated with an **linear operation** that we apply to vectors.

Consider a polygon P_1 defined by N vertex points $(x_1, y_1), \ldots, (x_N, y_N)$, connected in that order, and where the last point is also connected to the first point so there N edges.

Define a new polygon P_2 where each vertex is the *average* of the two points along an edge of polygon P_1.



Mathematically, the first new vertex point is linearly related to the old points as: $\hat{x}_1 = \dfrac{x_1 + x_2}{2}$, $\hat{y}_1 = \dfrac{y_1 + y_2}{2}$.

In general, the nth new vertex point is related to the old vertex points by the following formula:

$$\hat{x}_n = \frac{x_n + x_{(n+1) \bmod N}}{2}, \; \hat{y}_n = \frac{y_n + y_{(n+1) \bmod N}}{2}, \quad n = 1, \ldots, N.$$

It is convenient to collect all of these relationships into length-N vectors as follows:

$$\begin{bmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_{N-1} \\ \hat{x}_N \end{bmatrix} = \frac{1}{2} \begin{bmatrix} x_1 + x_2 \\ \vdots \\ x_{N-1} + x_N \\ x_N + x_1 \end{bmatrix}, \qquad \begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_{N-1} \\ \hat{y}_N \end{bmatrix} = \frac{1}{2} \begin{bmatrix} y_1 + y_2 \\ \vdots \\ y_{N-1} + y_N \\ y_N + y_1 \end{bmatrix}. \tag{1.2}$$

This is a **linear operation** so we can write it concisely using **matrix-vector multiplication**:

$$\hat{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}, \quad \hat{\boldsymbol{y}} = \boldsymbol{A}\boldsymbol{y}, \qquad \boldsymbol{A} \triangleq \tfrac{1}{2} \begin{bmatrix} 1 & 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 1 & 0 & \ldots & 0 \\ \vdots & & \ddots & \ddots & & \\ 0 & 0 & \ldots & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & \ldots & 1 \end{bmatrix}. \tag{1.3}$$

The matrix form might not seem more illuminating than the expressions in (1.2) at first. But now suppose we ask the following question: what happens to the polygon shape if we perform the process repeatedly (assuming we do some appropriate normalization)? The answer is not obvious from (1.2) but in matrix form the answer is related to the **power iteration** discussed in Ch. 7 for computing the **principle eigenvector** of $\boldsymbol{A}$. See [6] and `https://www.jasondavies.com/random-polygon-ellipse/` and Apr. 2018 SIAM News and Sep. 2018 SIAM News.

1.2 Matrix structures

Notation

L§1.1

- $\mathbb{R}$: real numbers
- $\mathbb{C}$: complex numbers
- $\mathbb{F}$: field (see Appendix on p. 1.79), which here will always be $\mathbb{R}$ or $\mathbb{C}$.
  We will use this symbol frequently to discuss properties that hold for both real and complex cases.
- $\mathbb{R}^N$: set of N-tuples of real numbers
- $\mathbb{C}^N$: set of N-tuples of complex numbers
- $\mathbb{F}^N$: either $\mathbb{R}^N$ or $\mathbb{C}^N$
- $\mathbb{R}^{M \times N}$: set of real $M \times N$ matrices
- $\mathbb{C}^{M \times N}$: set of complex $M \times N$ matrices
- $\mathbb{F}^{M \times N}$: either $\mathbb{R}^{M \times N}$ or $\mathbb{C}^{M \times N}$
  Because $\mathbb{R}^{M \times N} \subset \mathbb{C}^{M \times N}$, whenever you see the symbol $\mathbb{F}^{M \times N}$ you can just think of it as $\mathbb{C}^{M \times N}$, but it means the properties being discussed also hold for $\mathbb{R}^{M \times N}$.
- vectors are typically column vectors here (but see Appendix for more detail)
- row vectors are written $\boldsymbol{x}^\top$ or $\boldsymbol{x}'$, where $\boldsymbol{x} \in \mathbb{R}^n$ or $\boldsymbol{x} \in \mathbb{C}^n$.

(Read)

To explain the convenience of using the $\mathbb{F}$ notation, consider the following statements about **vector addition**, *i.e.*, $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$, all of which are true:

(i) $\boldsymbol{x} \in \mathbb{R}^N,\ \boldsymbol{y} \in \mathbb{R}^N \Longrightarrow \boldsymbol{z} \in \mathbb{R}^N$

(ii) $\boldsymbol{x} \in \mathbb{C}^N,\ \boldsymbol{y} \in \mathbb{C}^N \Longrightarrow \boldsymbol{z} \in \mathbb{C}^N$

(iii) $\boldsymbol{x} \in \mathbb{R}^N,\ \boldsymbol{y} \in \mathbb{C}^N \Longrightarrow \boldsymbol{z} \in \mathbb{C}^N$

(iv) $\boldsymbol{x} \in \mathbb{C}^N,\ \boldsymbol{y} \in \mathbb{R}^N \Longrightarrow \boldsymbol{z} \in \mathbb{C}^N$

(v) $\boldsymbol{x} \in \mathbb{R}^N,\ \boldsymbol{y} \in \mathbb{R}^N \Longrightarrow \boldsymbol{z} \in \mathbb{C}^N$

Because $\mathbb{R} \subset \mathbb{C}$, statement (ii) implies (iii), (iv), and (v). However, (ii) does *not* imply (i)! Of course (i) is true, but simply stating or proving (ii) by itself is insufficient to conclude that (i) is true.
So to thoroughly describe vector addition one would need to prove (or state as fact) both (i) and (ii) separately.

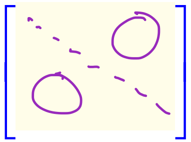
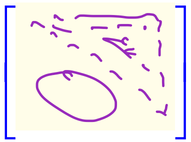
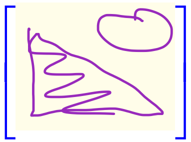
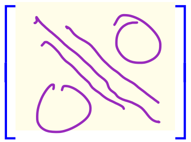
Instead these notes will write statements like this:

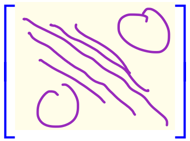
$$\boldsymbol{x} \in \mathbb{F}^N,\ \boldsymbol{y} \in \mathbb{F}^N \Longrightarrow \boldsymbol{z} \in \mathbb{F}^N,$$

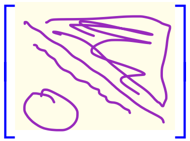
which is essentially shorthand for stating both (i) and (ii) above. Proving a statement like that often will require separately proving (i) and (ii).

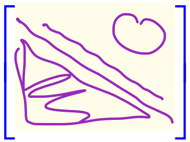
Common matrix shapes and types

For each matrix shape, this table gives one *of many* examples of why that shape is useful in practice.
These shapes are defined by where non-zero values may be. A matrix of all zeros is in all of these categories!

diagonal		Covariance matrix of a random vector with uncorrelated elements. Easiest to invert! Definition: $\underline{a_{ij} = 0 \text{ if } i \neq j}$
upper triangular		Arises in **Gaussian elimination** for solving systems of equations. Quite easy to invert. Definition: $a_{ij} = 0$ if $j < i$
lower triangular		Used in the **Cholesky decomposition**.
tridiagonal		**Finite difference** approximation of a 2nd derivative. Arises in numerical solutions to differential equations.

pentadiagonal  **Gram matrix** for discrete approximation to 2nd derivative.

upper Hessenberg  Arises in **eigenvalue algorithms**. Fairly easy to invert.

lower Hessenberg  Ditto.

The above are all **square matrix** shapes!
There is also a **rectangular diagonal matrix** shape that will be useful later for the **SVD**:

$$\text{tall: } \left[\begin{array}{ccc} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \\ \hline & \mathbf{0} & \end{array}\right] \quad \text{or wide: } \left[\begin{array}{ccc|c} \sigma_1 & & 0 & \\ & \ddots & & \mathbf{0} \\ 0 & & \sigma_r & \end{array}\right].$$

Often these are just called "diagonal matrices" too.

Important matrix classes

- A **full matrix** aka **dense matrix**: most entries are nonzero.
- A **sparse matrix**: many entries are zero or few entries are nonzero.
  Arises in many fields including medical imaging (tomography) [7]
  (Obviously terms like "most" "many" "few" are qualitative.)
  - To store a $M \times N$ dense matrix in memory we must store the MN values and also the size M, N.
  - To store a sparse matrix with K nonzero elements, we must store those values and also store the *index* of each value in some **sparse storage format**. Typically this is about $2K$ values.
- **Toeplitz**: elements are constant along each diagonal (not necessarily square).
  Arises in any application that has **time invariance** or **shift invariance**

  Fact. Matrix $\boldsymbol{A}$ is **Toeplitz** iff its elements a_{ij} have the form $\boxed{a_{ij} = f(i-j)}$ for some function $f : \mathbb{Z} \mapsto \mathbb{F}$.
- **Circulant**: each row is a shifted (circularly to the right by one column) version of the previous row.
  Always square in this class.
  Arises when considering periodic boundary conditions, *e.g.*, with the DFT.

  Fact. A $N \times N$ matrix $\boldsymbol{A}$ is **circulant** iff its elements a_{ij} have the form $\boxed{a_{ij} = f(i-j \mod N)}$ for some function $f : \{0, \dots, N-1\} \mapsto \mathbb{F}$.

Example. Toeplitz: $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 1 & 2 & 3 \\ 6 & 5 & 1 & 2 \end{bmatrix}$ Circulant: $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \\ 3 & 4 & 1 & 2 \\ 2 & 3 & 4 & 1 \end{bmatrix}$

3. Which statement is correct?

A: All Toeplitz matrices are circulant.
[✓ B: All circulant matrices are Toeplitz.]
C: Both are true.
D: Neither statement is true.

??

Which statement is correct?

A: All Toeplitz matrices are full.
B: All sparse matrices are Toeplitz.
C: There are no sparse Toeplitz matrices.
[✓ D: None of these statements is true.]

??

4. What kind of matrix is $\boldsymbol{A}$ in (1.3)? Choose most specific correct answer.

A: Diagonal B: Toeplitz [✓ C: Circulant] D: Upper Hessenberg E: None **??**

True

5. What kind of matrix is the convolution matrix $\boldsymbol{H}$ (1.1) on p. 1.6? Choose most specific correct answer.

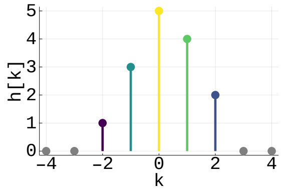
A: Diagonal [✓ B: Toeplitz] C: Circulant D: Upper Hessenberg E: None **??**

??

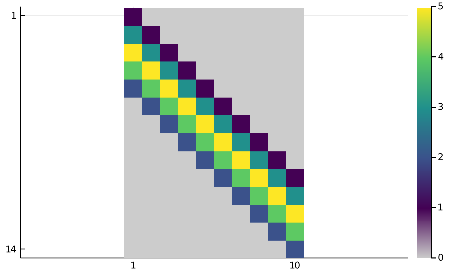
$\uparrow$ shift-invariance (time)

Convolution as a Toeplitz matrix (Read) ♦♦

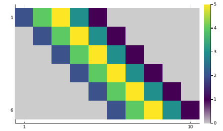
Example. Here is a PSF $h[k]$ with $K = 5$, and $\boldsymbol{H}$ for $N = 10$.
The `'full'` and `'same'` options implicitly assume zero boundary conditions.
In each case here, $\boldsymbol{H}$ is Toeplitz, and also circulant in one case.



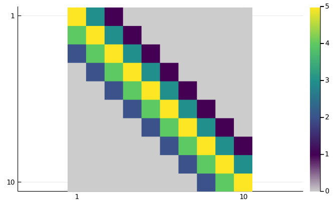
Matlab `'full'` $M = N+K-1$:



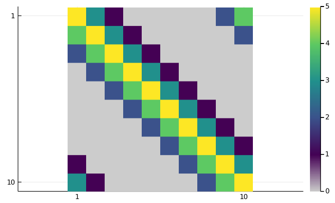
Matlab `'valid'` $M = N - K + 1$:



Matlab `'same'` $M = N$:



circulant (periodic end conditions) $M = N$:



Block matrix classes

The above definitions of shapes and classes were defined in terms of the scalar entries of a matrix.
We generalize these definitions by replacing scalars with matrices, leading to **block matrix** shapes.
We again use square brackets to denote the construction of a block matrix.

Example: **block diagonal** matrix

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{A}_2 \end{bmatrix}$$

bold

where $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ are arbitrary (possibly full) matrices and each $\boldsymbol{0}$ denotes an all-zero matrix of the appropriate size. (Wikipedia says that $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ should be square but I disagree with that restriction.)

6. If $\boldsymbol{A}_1$ is 6×8 and $\boldsymbol{A}_2$ is 7×9 what is the size of the $\boldsymbol{0}$ matrix in the upper right?
A: 6×7 [✓ B: 6×9] C: 7×8 D: 8×7 E: 8×9 $\boxed{??}$

Example: **block circulant** matrix

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} & \boldsymbol{C} \\ \boldsymbol{C} & \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{B} & \boldsymbol{C} & \boldsymbol{A} \end{bmatrix}$$

$$\begin{matrix} A & B \\ B & A \end{matrix}$$

$$\left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ \hline 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array}\right]$$

Combinations of block shapes and the shape of each block are also important.
Example: a matrix form of the 2D DFT is **block circulant with circulant blocks** (**BCCB**). (See EECS 556.)

Matrix transpose and symmetry

L§1.1

Define. The **transpose** of a $M \times N$ matrix $\boldsymbol{A}$ is denoted $\boldsymbol{A}^\top$ and is the $N \times M$ matrix whose (i, j)th entry is the (j, i)th entry of $\boldsymbol{A}$.
If $\boldsymbol{A} \in \mathbb{C}^{M \times N}$ then its **Hermitian transpose** (or **conjugate transpose** or **adjoint**) is the $N \times M$ matrix whose (i, j)th entry is the complex conjugate of the (j, i)th entry of $\boldsymbol{A}$.

Common notations for Hermitian transpose are: $\boldsymbol{A}'$, $\boldsymbol{A}^\mathsf{H}$ and $\boldsymbol{A}^\star$. (Here is some history.)
These notes will mostly use $\boldsymbol{A}'$ because that notation matches JULIA.
If $\boldsymbol{A}$ is real, then $\boldsymbol{A}' = \boldsymbol{A}^\top$ so these notes will mostly use $\boldsymbol{A}'$ regardless of whether $\boldsymbol{A}$ is real or complex for simplicity of notation.

Define. A (square) matrix $\boldsymbol{A}$ is called **symmetric** iff $\boldsymbol{A} = \boldsymbol{A}^\top$.

Define. A (square) matrix $\boldsymbol{A}$ is called **Hermitian symmetric** or just **Hermitian** iff $\boldsymbol{A} = \boldsymbol{A}'$.

See [1, Example 1.2].

Properties of transpose

- $(\boldsymbol{A}')' = \boldsymbol{A}$
- $(\boldsymbol{A} + \boldsymbol{B})' = \boldsymbol{A}' + \boldsymbol{B}'$ (if $\boldsymbol{A}$ and $\boldsymbol{B}$ have same size so that matrix addition is well defined)
- $(\boldsymbol{A}\boldsymbol{B})' = \boldsymbol{B}'\boldsymbol{A}'$ if $\boldsymbol{A} \in \mathbb{F}^{M \times N}$ and $\boldsymbol{B} \in \mathbb{F}^{N \times K}$, *i.e.*, if inner dimensions match (see matrix multiplication on p. 1.34)
- And more properties of **transpose** and **conjugate transpose** ...

Practical implementation

Caution: in JULIA: `x'` denotes the **Hermitian transpose** of a (possibly complex) vector or matrix `x`, whereas `x.'` in MATLAB or `transpose(x)` in JULIA denotes the **transpose**.
In JULIA, the **Hermitian transpose** syntax `A'` calls `adjoint(A)`. Try `@which ones(2)'` to see.

When performing mathematical operations with complex data, we usually need the Hermitian transpose. However, when simply rearranging how data is stored, sometimes we need only the transpose. Many hours of MATLAB software debugging time are spent on missing or extra periods (*i.e.*, `x'` vs `x.'`) for a transpose! To help avoid this problem, JULIA 0.7 and beyond has deprecated `x.'` so use `transpose(x)` in the rare cases where we have complex data but need a transpose rather than a Hermitian transpose.

- The transpose or Hermitian transpose of a **vector** takes negligible time in a modern language like JULIA because the array elements stay in the same order in memory; one just needs to modify the variable type from Array to Adjoint Array. See `02-vector` ←
- In many computing languages, the transpose of a (large) **matrix** requires considerable shuffling of values in memory and should be avoided when possible to save compute time. JULIA uses a clever approach that avoids memory shuffling. Conceptually, to compute `A'y`, JULIA essentially uses the equivalent form $\boldsymbol{A}'\boldsymbol{y} = (\boldsymbol{y}'\boldsymbol{A})'$ that does not require a matrix transpose. A'*Y
- Example. To compute $\boldsymbol{x}'\boldsymbol{A}'\boldsymbol{y}$ it may be faster (for large data, in some languages) to use `(A * x)' * y` instead of `x' * A' * y` because the latter may require a transpose operation (unless the compiler is smart enough to avoid it, like in JULIA).
  For complex data, `conj(y' * A * x)` is another alternative.

2021-08-31

1.3 Multiplication

The defining two operations in **linear algebra** are addition and multiplication of vectors and matrices. Addition is trivial so these notes focus on multiplication.

Vector-vector multiplication

If $\boldsymbol{x} \in \mathbb{C}^N$ and $\boldsymbol{y} \in \mathbb{C}^N$ are two vectors (of the same length) then their **dot product** or **inner product** is:

$$\langle \boldsymbol{x},\, \boldsymbol{y} \rangle = \boldsymbol{y}'\boldsymbol{x} = \underbrace{\begin{bmatrix} y_1^* & \dots & y_N^* \end{bmatrix}}_{1 \times N} \underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}}_{\text{“}N \times 1\text{”}} = \qquad \text{(scalar)}$$

Most books use $\langle \boldsymbol{x},\, \boldsymbol{y} \rangle = \boldsymbol{y}'\boldsymbol{x}$ whereas [1] uses $\langle \boldsymbol{x},\, \boldsymbol{y} \rangle = \boldsymbol{x}'\boldsymbol{y}$. These notes will use the common convention. With this convention, the inner product is **linear in the first argument**: $\langle \alpha\boldsymbol{x},\, \boldsymbol{y} \rangle = \alpha \langle \boldsymbol{x},\, \boldsymbol{y} \rangle,\ \forall \alpha \in \mathbb{F}$.

The dot product is central to **linear discriminant analysis** (**LDA**), a topic in pattern recognition and machine learning, for two-class classification. In SP terms, it is at the heart of the **matched filter** for signal detection. The dot product is central to convolution and to neural networks [8], *e.g.*, the **perceptron** [9].

In contrast, the **outer product** of vector $\boldsymbol{x} \in \mathbb{C}^M$ with vector of possibly different length $\boldsymbol{y} \in \mathbb{C}^N$ is the following $M \times N$ matrix:

$$\boldsymbol{x}\boldsymbol{y}' = \underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_M \end{bmatrix}}_{\text{“}M \times 1\text{”}} \underbrace{\begin{bmatrix} y_1^* & \dots & y_N^* \end{bmatrix}}_{1 \times N} = \underbrace{\begin{bmatrix} x_1 y_1^* & x_1 y_2^* & \dots & x_1 y_N^* \\ \vdots & & & \\ x_M y_1^* & x_M y_2^* & \dots & x_M y_N^* \end{bmatrix}}_{M \times N}.$$

Later we will see that this outer product is a **rank 1 matrix** (unless either $\boldsymbol{x}$ or $\boldsymbol{y}$ are $\boldsymbol{0}$).
Outer products are central to **matrix decompositions** like the **SVD** discussed soon.

Practical implementation

- Inner product:
  In JULIA: `y' * x` and `*(y',x)` and `y'x` and `dot(y,x)` and `⋅(y,x)` all perform $\boldsymbol{y}'\boldsymbol{x} = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$.
  The form `y'x` is remarkably similar to the math. Caution: `y' x` (extra space) does not work.
  For the form with `⋅`, first do `using LinearAlgebra`, then type `\cdot` and hit tab. See **dot**.
  Another (slower) way that matches the summation expression above is `sum(conj(y) .* x)`
- Outer product:
  In JULIA: `x * y'` is the most clear syntax, although `*(x, y')` also works.
  In fact `a * b` is translated by the compiler to `*(a,b)`.

Advanced JULIA coding for performance (Read) ♦♦

```julia
using BenchmarkTools: @btime
using LinearAlgebra: dot
N = 2^16; x = rand(ComplexF32, N); y = rand(ComplexF32, N)
f1(x,y) = y'x
f2(x,y) = dot(y,x)
f3(x,y) = sum(conj(y) .* x)
function f4(x,y) # basic loop
    accum = zero(promote_type(eltype(x), eltype(y)))
    for i in 1:length(x)
        accum += x[i] * conj(y[i])
    end
    return accum
end
function f5(x,y) # advanced loop
    accum = zero(promote_type(eltype(x), eltype(y)))
    @simd for i in 1:length(x)
        @inbounds accum += x[i] * conj(y[i])
    end
    return accum
end
```

```
@assert f1(x,y) == f2(x,y) ≈ f3(x,y) ≈ f4(x,y) ≈ f5(x,y) # check
# jf28: imac with 8 threads (1 thread run was the same)
# ir74,ir81: linux server with 2@20, 2@12 cpu cores
#                              jf28 ir74 ir81 (times in us)
@btime f1($x,$y); # y'x        13.2 32.4 24.0
@btime f2($x,$y); # dot(y,x)   13.2 32.4 24.0
@btime f3($x,$y); # sum()      160. 180. 178.
@btime f4($x,$y); # basic loop 60. 104. 82.4
@btime f5($x,$y); # fancy loop 73. 36.7 22.3 (@inbounds & @simd can help!)
```

The above code times several different ways of doing vector dot products.

- The `sum(conj(y) .* x)` needs memory allocations for intermediate variables, making it slow.
- The vector operation `y'x` calls the built-in `dot()` function from BLAS library that uses cpu-specific assembly code based on **single instruction, multiple data** (**SIMD**) to perform 4 multiplies in a single CPU instruction. Thus, the basic loop is 4-5$\times$ slower than `dot()`.
- However, we can instruct the JULIA compiler to use SIMD operations by adding the `@simd` macro.
- Sometimes we can also speed up code by promising the JULIA compiler that the array indexing operations like `x[i]` are valid, by adding the `@inbounds` macro.
- With these small code modifications, performance is comparable to `dot` on a newish linux server.
- The `promote_type` function ensures that the accumulator uses the better precision of the arguments.

Colon notation

Next we consider multiplication with a matrix.
First we need some notation because matrix multiplication uses rows and/or columns of a matrix.

For $\boldsymbol{A} \in \mathbb{F}^{M \times N}$ (see (1.4) below):

- $\boldsymbol{A}_{:,1} = \boldsymbol{A}\boldsymbol{e}_1$ denotes the first **column** of $\boldsymbol{A}$ (a vector of length M), exactly like `A[:,1]` in JULIA, where $\boldsymbol{e}_1$ denotes the first **unit vector** in $\mathbb{R}^N$. Similarly let $\tilde{\boldsymbol{e}}_1$ denote the first **unit vector** in $\mathbb{R}^M$.
- $\boldsymbol{A}_{1,:} = \tilde{\boldsymbol{e}}_1' \boldsymbol{A}$ denotes the first **row** of $\boldsymbol{A}$ (a row vector of length N).
  Caution: `A[1,:]` in JULIA returns a *vector*, not a "$1 \times N$" array like MATLAB does.

Using this colon notation we have the following two ways to think about a matrix.

- Column partition of a $M \times N$ matrix: $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{:,1} & \dots & \boldsymbol{A}_{:,N} \end{bmatrix}$
- Row partition of a $M \times N$ matrix: $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{1,:} \\ \vdots \\ \boldsymbol{A}_{M,:} \end{bmatrix}$

Of course we also have the element-wise way of writing out a $M \times N$ matrix:

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & \dots & a_{1,N} \\ \vdots & \dots & \vdots \\ a_{M,1} & \dots & a_{M,N} \end{bmatrix},$$

but for multiplication operations we often think about the column or row partitions above instead.

Matrix-vector multiplication

If $\boldsymbol{A} \in \mathbb{F}^{M \times N}$ and $\boldsymbol{x} \in \mathbb{F}^N$, then the **matrix-vector product** $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ is a vector of length M defined by

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} \quad \Longleftrightarrow \quad \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} = \begin{bmatrix} a_{1,1}x_1 + \cdots + a_{1,N}x_N \\ a_{2,1}x_1 + \cdots + a_{2,N}x_N \\ \vdots \\ a_{M,1}x_1 + \cdots + a_{M,N}x_N \end{bmatrix}, \quad \textit{i.e.,} \; y_i = \sum_{j=1}^{N} a_{ij} x_j. \tag{1.4}$$

A matrix-vector product requires MN floating-point multiplies and $M(N-1)$ floating-point additions, or a total of $2MN - M$ **floating-point operations** (**FLOPs**).

Often when considering the computational costs of some method, our primary interest is how the computation increases as the problem size grows. So constant scaling factors like the "2" above are of secondary importance, and the "$-M$" term is small compared to the product MN as the problem size grows. Thus, as a concise summary using **big O** notation, we say that matrix-vector multiplication requires $O(MN)$ FLOPs.

Why is matrix-vector multiplication important?

- If $\boldsymbol{A}$ is a data matrix, then $\boldsymbol{A}\boldsymbol{x}$ is a linear combination of its columns, and we often use such linear combinations for modeling and prediction.
- If $\boldsymbol{A}$ corresponds to a (linear) operation, then we can think of $\boldsymbol{A}$ as representing a linear system and the operation $\boldsymbol{x} \mapsto \boldsymbol{A}\boldsymbol{x}$ describes the output of the system when the input is $\boldsymbol{x}$.

Practical implementation

In JULIA (or MATLAB) we simply write `y = A*x` to perform matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$.

The terrific similarity between this syntax and the mathematical expression is a key benefit of such high-level languages!

In contrast, for low-level languages (like C and Fortran), one must either call a function in a numerical linear algebra library or write a double loop as shown to the right, looking nothing like the math.

```
function mv_mn(A, x) # A * x
    (M,N) = size(A)
    y = similar(x, M) # preallocate!
    for m=1:M
        inprod = 0 # accumulator
        for n=1:N
            inprod += A[m,n] * x[n]
        end
        y[m] = inprod
    end
    return y
end
```

For efficiency reasons and for better readability, in JULIA, unlike in MATLAB, one must must preallocate space for the result vector so that memory does not grow with the iteration over `m`.

The **matrix-vector product** (1.4) has two mathematically (but not practically) equivalent vector forms:

$$\boldsymbol{A} \in \mathbb{F}^{M\times N},\ \boldsymbol{x} \in \mathbb{F}^N \Longrightarrow \boldsymbol{A}\boldsymbol{x} = \underbrace{\begin{bmatrix} \boldsymbol{A}_{1,:}\,\boldsymbol{x} \\ \vdots \\ \boldsymbol{A}_{M,:}\,\boldsymbol{x} \end{bmatrix}}_{M \text{ “inner products”}} = \underbrace{\sum_{n=1}^N \boldsymbol{A}_{:,n} x_n = \qquad\qquad}_{\text{linear combination of columns}} \tag{1.5}$$

In JULIA, instead of using `y = A * x` we could instead implement matrix-vector multiplication using either of the following two loops, corresponding to the two vector forms in (1.5):

```
function mv_row(A, x)
    M = size(A,1)
    y = similar(x, M) # preallocate
    for m=1:M
        y[m] = transpose(A[m,:]) * x
    end
    return y
end
```

```
function mv_col(A, x)
    (M,N) = size(A)
    y = zeros(eltype(x), M) # init 0
    for n=1:N
        y += A[:,n] * x[n]
    end
    return y
end
```

For a comparison of the compute efficiency of these implementations and optimized variants, see:
https://web.eecs.umich.edu/~fessler/course/551/julia/tutor/multiply-matrix-vector.html

Matrix-vector multiplication with transpose

If $\boldsymbol{A}$ is a $M \times N$ matrix and $\boldsymbol{y} \in \mathbb{F}^M$ is a vector, then the (Hermitian) transposed **matrix-vector product** is

$$\boldsymbol{A}'\boldsymbol{y} = \underbrace{\begin{bmatrix} (\boldsymbol{A}_{:,1})'\,\boldsymbol{y} \\ \vdots \\ (\boldsymbol{A}_{:,N})'\,\boldsymbol{y} \end{bmatrix}}_{N \text{ inner products}} = \underbrace{\sum_{m=1}^{M} (\boldsymbol{A}_{m,:})' y_m}_{\text{linear combination}}.$$

In JULIA (and MATLAB, but not `numpy`), a standard 2D `Array` , aka a `Matrix` is stored in memory with its first index varying fastest, called **column major** order.
(JULIA has other types like `StridedArray` and `AbstractArray` with other storage orders.) ♦♦

Example. If $\boldsymbol{A} = \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix}$ then `A[:]` or `vec(A)` both reveal the memory storage order: $\begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}.$

Computing $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ via inner products means $\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 5x_1 + 7x_2 \\ 6x_1 + 8x_2 \end{bmatrix}$, where $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$.
Here the memory access order is non-sequential: the top row uses 5 and 7, which are not adjacent in memory.
For large arrays, non-sequential access can lead to slower execution time because of poor memory cache use.

In contrast, computing $\boldsymbol{x} = \boldsymbol{A}'\boldsymbol{y}$ via inner products means $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5y_1 + 6y_2 \\ 7y_1 + 8y_2 \end{bmatrix}$.
Here the elements of $\boldsymbol{A}$ are accessed sequentially, improving cache use.

The following test compares matrix-vector multiplication versus transposed-matrix-vector multiplication. Both $\boldsymbol{A}\boldsymbol{x}$ and $\boldsymbol{A}'\boldsymbol{y}$ require $O(MN)$ FLOPs. The comments show time for experiments on a 2017 iMac Pro, 4.2GHz 4-core CPU, OS 10.14.6, for JULIA 1.5.

```
using BenchmarkTools: @btime
N = 2^11; M = N+1; A = rand(M,N); x = rand(N); y = rand(M)
f1(y,A,x) = A * x
f2(y,A,x) = A'y
f3(y,A,x) = A' * y
f4(y,A,x) = transpose(A) * y
@btime f1($y,$A,$x); # A*x          995 us
@btime f2($y,$A,$x); # A'y          930 us
@btime f3($y,$A,$x); # A'*y         930 us
@btime f4($y,$A,$x); # transpose(A)*y 930 us (was much slower in 0.6)
```

- `A'*y` is a bit faster than `A*x` for the reasons discussed above.
- Similarly, in MATLAB, the same test used 1001 us for `A*x` and 976 us for `A'*y`.
- For timing code, it is better to use `BenchmarkTools.@btime` than to use `@time`.
- For JULIA 0.7 and above, `transpose(A)` and `A'` and `adjoint(A)` do not reorder memory, but rather set an internal header flag so that it can perform `A'*y` and `transpose(A)*y` efficiently.
  Try: `A = ones(2,3); B = A'; B[4] = 5; display(A)`

7. In light of these considerations, which of the following weighted inner product calculations $\boldsymbol{y}'\boldsymbol{A}\boldsymbol{x}$ is likely to run fastest:

A: `y'*(A*x)` B: `(y'*A)*x` C: Roughly same time

$\boxed{??}$

Consider the following very similar computation of the weighted inner product $\boldsymbol{x}'\boldsymbol{A}'\boldsymbol{y}$.

```
using BenchmarkTools: @btime
N = 1000; M = N+1; A = rand(M,N); x = rand(N); y = rand(M);
f1(x,A,y) = x'*(A'*y)
f2(x,A,y) = (x'*A')*y
f3(x,A,y) = x'*A'*y
f4(x,A,y) = (x'*transpose(A))*y
@assert f1(x,A,y) ≈ f2(x,A,y) ≈ f3(x,A,y) ≈ f4(x,A,y) # check
@btime f1($x,$A,$y) # x'*(A'*y)                 # 61 us
@btime f2($x,$A,$y) # (x'*A')*y                 # 71 us
@btime f3($x,$A,$y) # x'*A'*y                   # 71 us
@btime f4($x,$A,$y) # (x'*transpose(A))*y # 71 us
```

- Here the compiler is *not* smart enough to determine the best execution order.
  (Apparently it parses code left to right.)
- Here, using appropriate parentheses can accelerate the code!
- A minute of thinking and a few seconds of typing (parentheses) can save compute time (and energy).

```
using BenchmarkTools: @btime
N = 1000; M = N+1; A = rand(M,N); x = rand(N); y = rand(M);
f1(y,A,x) = y'*(A*x)
f2(y,A,x) = (y'*A)*x
f3(y,A,x) = y'*A*x
f4(y,A,x) = transpose(y)*A*x
@assert f1(y,A,x) ≈ f2(y,A,x) ≈ f3(y,A,x) ≈ f4(y,A,x) # check
@btime f1($y,$A,$x) # y'*(A*x)              # 72 us
@btime f2($y,$A,$x) # (y'*A)*x              # 52 us
@btime f3($y,$A,$x) # y'*A*x                # 52 us
@btime f4($y,$A,$x) # transpose(y)*(A*x) # 52 us
```

- Here the left-to-right parsing happens to produce the faster result, so parentheses were not needed.
- Vector transpose takes no time because no memory shuffling is needed.
- Some past exam questions needed proper parentheses to earn full credit.

Matrix-matrix multiplication

L§1.2

Let $\boldsymbol{A} \in \mathbb{F}^{M\times K}$ and $\boldsymbol{B} \in \mathbb{F}^{K\times N}$ then the result of the **matrix-matrix product** is $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} \in \mathbb{F}^{M\times N}$.

Define. The standard definition for **matrix multiplication** uses the rows of $\boldsymbol{A}$ and the columns of $\boldsymbol{B}$:

$$C_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj}, \quad \text{for } i = 1, \ldots, M, \; j = 1, \ldots, N. \tag{1.6}$$

The "inner dimension" (K here) must match for valid matrix-matrix multiplication.
Matrices satisfying such suitable dimensions are called **conformable**.

Matrix-matrix multiplication is important for representing the **cascade** of two linear systems (when $\boldsymbol{A}$ and $\boldsymbol{B}$ both represent operations) and **matrix decomposition** relies on matrix products.

Eqn. (1.6) includes matrix-vector multiplication as a special case, *if* we abuse terminology and treat a vector in $\mathbb{F}^N$ as a $N \times 1$ matrix (like MATLAB does). These notes will avoid that abuse and treat vectors separately from matrices (like JULIA does).

Computation

- Using the formula (1.6), we must compute K scalar multiplies for each of MN elements of $\boldsymbol{C}$, for a total of MKN multiplies and roughly that number of adds.
- In particular, if $\boldsymbol{A}$ and $\boldsymbol{B}$ are both $N \times N$ matrices, then the natural implementation of (1.6) requires $O(N^3)$ operations.
- Recent work [10] provides a "laser method" that reduces it to $O(N^{2.37286})$ operations.

Matrix multiplication properties

- The **distributive property** of matrix multiplication is: $\boldsymbol{A}(\boldsymbol{B}+\boldsymbol{C}) = \boldsymbol{A}\boldsymbol{B} + \boldsymbol{A}\boldsymbol{C}$ if the sizes match.
- There is also **associative property**: $\boldsymbol{A}(\boldsymbol{B}\boldsymbol{C}) = (\boldsymbol{A}\boldsymbol{B})\boldsymbol{C}$ if the sizes match.
- There is no general **commutative property**! In general $\boldsymbol{A}\boldsymbol{B} \neq \boldsymbol{B}\boldsymbol{A}$ even if the sizes match.
- Multiplication by an (appropriately sized) identity matrix has no effect: $\boldsymbol{I}\boldsymbol{A} = \boldsymbol{A}\boldsymbol{I} = \boldsymbol{A}$.
  If $\boldsymbol{A}$ is $M \times N$, then we sometimes write: $\boldsymbol{I}_M \boldsymbol{A} = \boldsymbol{A}\boldsymbol{I}_N = \boldsymbol{A}$.
- Matrix **concatenation**: if $\boldsymbol{A} \in \mathbb{F}^{M\times N}$ and $\boldsymbol{B} \in \mathbb{F}^{N\times K}$ and $\boldsymbol{C} \in \mathbb{F}^{N\times L}$ then $\boldsymbol{A}\begin{bmatrix}\boldsymbol{B} & \boldsymbol{C}\end{bmatrix} = \begin{bmatrix}\boldsymbol{A}\boldsymbol{B} & \boldsymbol{A}\boldsymbol{C}\end{bmatrix}$.
  Exercise: find the corresponding property for vertical concatenation.
- See [wiki] for more matrix multiplication properties.

Practical consideration: Parentheses matter!

The **associative property** is a simple math equality, but parentheses choice can *greatly* affect code speed!

Example. Suppose $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$ are all vectors in $\mathbb{R}^N$.
How many FLOPs are needed for the following very simple code? `x * y' * z`

The answer depends on where the compiler puts the parentheses.

- Left to right evaluation `(x * y') * z` requires $O(N^2)$ FLOPs
- Right to left evaluation `x * (y' * z)` requires $O(N)$ FLOPs

Bottom line: put in parentheses yourself to ensure efficiency!
For long chains use: https://github.com/AustinPrivett/MatrixChainMultiply.jl ♦♦

Four views of matrix multiplication

Besides (1.6), there are at least four (!) ways of viewing (and implementing) **matrix-matrix products**.

Why would you need four different versions?

- For big data and distributed computing using **parallel processing** some versions are preferable [11].
- Different versions are useful for mathematical manipulations.

Version 1 (dot or inner product form)

Using the row partition of $\boldsymbol{A}$ and the column partition of $\boldsymbol{B}$, one can verify:

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} = \underbrace{\begin{bmatrix} \boldsymbol{A}_{1,:} \\ \vdots \\ \boldsymbol{A}_{M,:} \end{bmatrix}}_{M \times K} \underbrace{\begin{bmatrix} \boldsymbol{B}_{:,1} & \ldots & \boldsymbol{B}_{:,N} \end{bmatrix}}_{K \times N} = \underbrace{\begin{bmatrix} \boldsymbol{A}_{1,:}\boldsymbol{B}_{:,1} & \ldots & \boldsymbol{A}_{1,:}\boldsymbol{B}_{:,N} \\ \vdots & & \\ \boldsymbol{A}_{M,:}\boldsymbol{B}_{:,1} & \ldots & \boldsymbol{A}_{M,:}\boldsymbol{B}_{:,N} \end{bmatrix}}_{M \times N}.$$

Specifically, the elements of $\boldsymbol{C}$ are given by the following (conjugate-less) **inner products**:

$$C_{ij} =$$

In JULIA code:
Simple but opaque: `C = A * B`

Double loop of vector "inner products:"

```
C = similar(A,M,N)
for m=1:M
    for n=1:N
        C[m,n] = transpose(A[m,:]) * B[:,n]
    end
end
```

Triple loop in terms of scalars per (1.6):

```
C = similar(A,M,N)
for m=1:M
    for n=1:N
        inprod = 0 # accumulator
        for k=1:K
            inprod += A[m,k] * B[k,n]
        end
        C[m,n] = inprod
    end
end
```

The transpose `transpose(A)` (not Hermitian transpose `A'`) is crucial for complex matrices!

Instead of: `C[m,n] = transpose(A[m,:])  * B[:,n]`
you could use: `C[m,n] = dot(conj(A[m,:]), B[:,n])`
or `C[m,n] = sum(A[m,:]  .* B[:,n])`

Reminder: For a $K \times N$ matrix $\boldsymbol{B}$, in JULIA `B[k,:]` returns a 1D vector of length N, not a "$N \times 1$" array nor a "row vector" (a $1 \times N$ array, like in MATLAB). See `02-vector`.

Version 2 (column-wise accumulation)

Using the column partition of $\boldsymbol{A}$ and the elements of $\boldsymbol{B}$:

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} = \begin{bmatrix} \boldsymbol{A}_{:,1} & \dots & \boldsymbol{A}_{:,K} \end{bmatrix} \begin{bmatrix} B_{1,1} & \dots & B_{1,N} \\ \vdots & & \\ B_{K,1} & \dots & B_{K,N} \end{bmatrix}$$

$$\Longrightarrow \boldsymbol{C}_{:,n} = \begin{bmatrix} \boldsymbol{A}_{:,1} & \dots & \boldsymbol{A}_{:,K} \end{bmatrix} \begin{bmatrix} B_{1,n} \\ \vdots \\ B_{K,n} \end{bmatrix} = \qquad (1.7)$$

In JULIA code we would loop over columns of $\boldsymbol{A}$, performing vector-scalar multiplication:

```
C = zeros(eltype(A),M,N) # must preallocate in julia, unlike in matlab
for n=1:N
    for k=1:K
        C[:,n] += A[:,k] * B[k,n]
    end
end
```

Note the "`+=`" accumulation operation in this code akin to C.

Version 3 (matrix-vector products)

Using the column partition of $\boldsymbol{B}$, one can verify this "horizontal concatenation" property:

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} = \underbrace{\boldsymbol{A}}_{M \times K} \underbrace{\begin{bmatrix} \boldsymbol{B}_{:,1} & \dots & \boldsymbol{B}_{:,N} \end{bmatrix}}_{K \times N} = \underbrace{\begin{bmatrix} \boldsymbol{A}\boldsymbol{B}_{:,1} & \dots & \boldsymbol{A}\boldsymbol{B}_{:,N} \end{bmatrix}}_{M \times N} \implies \boldsymbol{C}_{:,j} = \qquad (1.8)$$

JULIA code using a single loop over columns of $\boldsymbol{B}$ of matrix-vector products:

```
C = similar(A,M,N)
for n=1:N
    C[:,n] = A * B[:,n]
end
```

Practical tip. By default, using `C = zeros(M,N)` would make a `Float64` array. That is fine for small problems and when very good numerical precision is needed. To save memory for large problems (at the price of reduced precision) use `zeros(Float32, M, N)`. If desperate for memory savings, consider `zeros(Float16, M, N)`. Here we use `C = similar(A,M,N)` so that `C` matches the precision of `A`. Professional code would consider the element types of both arrays:
`T = promote_type(eltype(A),eltype(B));` `C = similar(T, M, N)` ♦♦

This class will often use the **matrix-vector** operation view (1.8).

Version 4 (sum of outer products)

Using the column partition of $\boldsymbol{A}$ and the row partition of $\boldsymbol{B}$:

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} = \underbrace{\begin{bmatrix} \boldsymbol{A}_{:,1} & \dots & \boldsymbol{A}_{:,K} \end{bmatrix}}_{M \times K} \underbrace{\begin{bmatrix} \boldsymbol{B}_{1,:} \\ \vdots \\ \boldsymbol{B}_{K,:} \end{bmatrix}}_{K \times N} = \tag{1.9}$$

For proof, see [1, Theorem 1.3].

JULIA code using a single loop (over inner dimension) of **outer products**:

```
C = zeros(eltype(A),M,N)
for k=1:K
    C .+= A[:,k] * transpose(B[k,:]) # column times row!
end
```

This code may be slow due to irregular memory access order, but we will use the mathematical "sum of outer product" representation (1.9) extensively.

Block matrix multiplication (Read)

The inner operation in (1.9) is an example of **block matrix multiplication** and the definition for such operations is basically the same as (1.6) as long as all the matrix dimensions involved match properly. (HW)

Kronecker product and Hadamard product and the vec operator (Read)

The formula (1.6) is the standard form of **matrix multiplication**. There are at least two other types of matrix multiplication that are important and have their own names.

- The **Kronecker product** is usually denoted $\boldsymbol{A} \otimes \boldsymbol{B}$

  In JULIA it is `kron(A,B)`
- The **Hadamard product** is the element-wise multiplication of two *same-sized* matrices. It is usually denoted $\boldsymbol{A} \odot \boldsymbol{B}$.

  In JULIA it is `A .* B`
- There are other kinds of matrix products like the **Tracy-Singh and Khatri-Rao products** ♦♦

The vec operator and matrix multiplication (Read)

The **vec operator** converts a matrix to a vector. It is often denoted

$$\mathsf{vec}(\boldsymbol{A}) = \underbrace{\mathsf{vec}([\boldsymbol{A}_{:,1} \; \ldots \; \boldsymbol{A}_{:,N}])}_{M \times N} = \underbrace{\begin{bmatrix} \boldsymbol{A}_{:,1} \\ \vdots \\ \boldsymbol{A}_{:,N} \end{bmatrix}}_{\text{vector in } \mathbb{F}^{MN}}.$$

In JULIA there are at least three ways to perform this operation, all of which stack the columns:

- `vec(A)` (this is the most memory efficient because it uses a JULIA `view`)
- `A[:]`
- `reshape(A, length(A))`

  Note that `reshape(A, length(A), 1)` is *not* the same thing because the result in this case is a 2D array of size $MN \times 1$, which is a different variable type than a 1D vector of length MN.

The following **vec trick** property [12, 13] is particularly important (HW):

$$\mathsf{vec}(\boldsymbol{A}\boldsymbol{X}\boldsymbol{B}) = (\boldsymbol{B}^{\mathsf{T}} \otimes \boldsymbol{A}) \, \mathsf{vec}(\boldsymbol{X}) \, .$$

In this formula $\boldsymbol{B}^{\mathsf{T}}$ must be a regular transpose, not a Hermitian transpose, even if $\boldsymbol{B}$ is complex!

Challenge: generalize to $(\boldsymbol{C} \otimes \boldsymbol{B} \otimes \boldsymbol{A}) \, \mathsf{vec}(\boldsymbol{X})$.

Using matrix-vector operations in high-level computing languages (Read)

Particularly in high-level, array-oriented languages like JULIA, matrix and vector operations abound. One should be on the lookout for opportunities to "parallelize" (vectorize) using such operations.

Example. Suppose we want to visualize the 2D Gaussian bump function $f(x, y) = \mathrm{e}^{-(x^2+30y^2)}$.

Elementary implementation with double loop:

```julia
using Plots
x = LinRange(-2, 2, 101)
y = LinRange(-1.1, 1.1, 103)
M = length(x)
N = length(y)
F = zeros(M,N)
for m=1:M
    for n=1:N
        F[m,n] = exp(-(x[m]^2 + 30 * y[n]^2))
    end
end
heatmap(x, y, F, transpose=true, color=:grays, aspect_ratio=:equal)
```

How do we leverage matrix-vector concepts to write this more concisely?
(Concise code is often easier to read and maintain, and often looks more like the mathematical expressions.)

Idea: Define a matrix $\boldsymbol{A}$ such that $A_{ij} = x_i^2 + 30y_j^2$.
Then use element-wise exponential operation: `F = exp.(-A)`

In JULIA, `exp.(A)` applies exponentiation **element-wise** to an array. This is called a **broadcast** operation.
In JULIA, `exp(A)` is meaningless for a matrix; use `expm(A)` for a **matrix exponential**.

How do we define $\boldsymbol{A}$ where $A_{ij} = x_i^2 + 30y_j^2$ without writing a double loop?

One way is to use outer products. If $\boldsymbol{u} \in \mathbb{R}^M$ has elements $u_i = x_i^2$ and $\boldsymbol{v} \in \mathbb{R}^N$ has elements $v_j = y_j^2$, then the following **sum of outer products** yields a $M \times N$ matrix:

$$\boldsymbol{A} = \boldsymbol{u}\boldsymbol{1}_N' + 30\boldsymbol{1}_M\boldsymbol{v}' = \begin{bmatrix} x_1^2 \\ \vdots \\ x_M^2 \end{bmatrix} \underbrace{\begin{bmatrix} 1 & \dots & 1 \end{bmatrix}}_{1 \times N} + 30 \underbrace{\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}}_{\text{length } M} \begin{bmatrix} y_1^2 & \dots & y_N^2 \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} x_1^2 & \dots & x_1^2 \\ \vdots & \vdots & \vdots \\ x_M^2 & \dots & x_M^2 \end{bmatrix}}_{\text{repeated column}} + 30 \underbrace{\begin{bmatrix} y_1^2 & \dots & y_N^2 \\ \vdots & \vdots & \vdots \\ y_1^2 & \dots & y_N^2 \end{bmatrix}}_{\text{repeated row}}, \qquad (1.10)$$

where $\boldsymbol{1}_N$ denotes the vector of all ones, *i.e.*, `ones(N)` in JULIA or `ones(N,1)` in MATLAB.

The key line of the following JULIA code looks reasonably close to the above outer-product form.

```julia
using Plots
x = LinRange(-2, 2, 101)
y = LinRange(-1.1, 1.1, 103)
M = length(x)
N = length(y)
A = (x.^2) * ones(N)' + 30 * ones(M) * (y.^2)'
F = exp.(-A)
heatmap(x, y, F, transpose=true, color=:grays, aspect_ratio=:equal)
```

This version avoids you, the user, from writing a double loop. Of course JULIA itself must do double loops internally when evaluating `exp.(-A)` and similar expressions.

The outer product approach has the benefit of avoiding writing double loops. However, this type of operation arises so frequently that JULIA provides software functions that avoid the unnecessary operations of multiplying by the `ones` vector.

In JULIA there is automatic **broadcast** [details here] of singleton dimensions by the scalar addition operator:

```
A = x.^2 .+ 30 * (y.^2)'
```

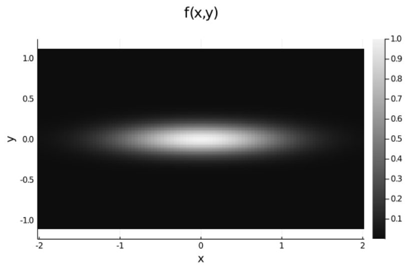
Two somewhat distinct uses of " `.` " here:
- `.^2` element-wise power
- `.+` normally element-wise addition of arrays but here with automatic broadcast like in (1.10).

This broadcast feature leads to the most concise code that looks the most like the math:

```
using Plots
x = LinRange(-2, 2, 101)
y = LinRange(-1.1, 1.1, 103)
A = x.^2 .+ 30 * (y.^2)' # a lot is happening here!
F = exp.(-A)
heatmap(x, y, F, transpose=true, color=:grays, aspect_ratio=:equal)
```

Finally, if your goal is merely to make a picture of the function $f(x,y)$, without illustrating any matrix properties, the following is a "JULIA way" to do it using its **multiple dispatch** feature. This version is the shortest of all so I added a few labeling commands.

```
using Plots
x = LinRange(-2, 2, 101)
y = LinRange(-1.1, 1.1, 103)
f = (x,y) -> exp(-(x^2 + 30y^2)) # look, no "*" !
heatmap(x, y, f, color=:grays, aspect_ratio=:equal,
    xlabel="x", ylabel="y", title="f(x,y)")
```



https://web.eecs.umich.edu/~fessler/course/551/julia/demo/01_gauss2d.html

https://web.eecs.umich.edu/~fessler/course/551/julia/demo/01_gauss2d.ipynb

Example. One can perform 1D numerical integration using a vector operation (a dot product). (Read)
To compute the area under a curve:

$$\text{Area} = \int_a^b f(x)\,\mathrm{d}x \approx \sum_{m=1}^M \underbrace{(x_m - x_{m-1})}_{\text{base}} \underbrace{f(x_m)}_{\text{height}} = \boldsymbol{w}'\boldsymbol{f}, \qquad \boldsymbol{w} = \begin{bmatrix} x_1 - x_0 \\ \vdots \\ x_M - x_{M-1} \end{bmatrix}, \quad \boldsymbol{f} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_M) \end{bmatrix}, \tag{1.11}$$

where (possibly nonuniformly spaced) sampled points satisfy: $a = x_0 < x_1 < \ldots < x_M = b$.
This summation is a **dot product** between two vectors in $\mathbb{R}^M$ and is easy in JULIA.

```
f(x) = x^2 # parabola
x = LinRange(0,3,2000) # sample points
w = diff(x) # "widths" of rectangles
Area = w' * f.(x[2:end])
```

What should the exact value be for the area in this example? $\int_0^3 x^2\,\mathrm{d}x = x^3/3|_{x=3} = 9$

Why `x[2:end]` instead of `x[1:end]` or simply `x`?
Because `w = diff(x)` returns a vector without the first element "$x_1 - x_0$" in (1.11).

https://web.eecs.umich.edu/~fessler/course/551/julia/demo/01_area.html

https://web.eecs.umich.edu/~fessler/course/551/julia/demo/01_area.ipynb

(Read)

Example. Consider the problem of computing numerically the volume under a 2D function $g(x, y)$:

$$V = \int_a^b \int_c^d g(x, y) \, \mathrm{d}y \, \mathrm{d}x \, .$$

We can also use matrix-vector products for this operation:

$$V = \int_a^b \int_c^d g(x, y) \, \mathrm{d}y \, \mathrm{d}x \approx S \triangleq \sum_{m=1}^{M} \sum_{n=1}^{N} (x_m - x_{m-1})(y_n - y_{n-1}) f(x_m, y_n),$$

where here $c = y_0 < y_1 < \ldots < y_N = d$.

Define vector $\boldsymbol{w} \in \mathbb{R}^M$ as in 1D example above and $\boldsymbol{u} \in \mathbb{R}^N$ and $\boldsymbol{F} \in \mathbb{R}^{M \times N}$ by

$$\boldsymbol{w} = \begin{bmatrix} x_1 - x_0 \\ \vdots \\ x_M - x_{M-1} \end{bmatrix}, \qquad \boldsymbol{u} = \begin{bmatrix} y_1 - y_0 \\ \vdots \\ y_N - y_{N-1} \end{bmatrix}, \qquad \boldsymbol{F} = \begin{bmatrix} f(x_1, y_1) & \ldots & f(x_1, y_N) \\ \vdots & & \vdots \\ f(x_M, y_1) & \ldots & f(x_M, y_N) \end{bmatrix}.$$

Then the above double sum is the following product in matrix-vector form: (*cf.* HW)

$$V \approx S = \boldsymbol{w}' \boldsymbol{F} \boldsymbol{u}.$$

Proof.

$$\boldsymbol{w}'\boldsymbol{F}\boldsymbol{u} = \begin{bmatrix} w_1 & \dots & w_M \end{bmatrix} \begin{bmatrix} f(x_1,y_1) & \dots & f(x_1,y_N) \\ \vdots & & \vdots \\ f(x_M,y_1) & \dots & f(x_M,y_N) \end{bmatrix} \begin{bmatrix} u_1 \\ \vdots \\ u_N \end{bmatrix}$$

$$= \begin{bmatrix} w_1 & \dots & w_M \end{bmatrix} \begin{bmatrix} \sum_{n=1}^N u_n f(x_1,y_n) \\ \vdots \\ \sum_{n=1}^N u_n f(x_M,y_n) \end{bmatrix} = \sum_{m=1}^M w_m \sum_{n=1}^N u_n f(x_m,y_n) = S.$$

Again this computation is easy to code in a high-level language like JULIA.

```
f(x,y) = exp(-(x^2 + 3*y^2)) # gaussian bump function
x = LinRange(0,3,2000) # sample points
y = LinRange(0,2,1000) # sample points
w = diff(x) # "widths" of rectangles in x
u = diff(y) # "widths" of rectangles in y
F = f.(x[2:end], y[2:end]') # automatic broadcasting again!
S = w' * F * u
```

(It would also be fine to write it as a double loop, albeit with more typing and possibly slower in some languages.)

In this case it is possible to determine the analytical value (*cf.* EECS 501) but that would probably take more time than writing and running this code.

Invertibility

(Read)

For a nonzero scalar x, we know that $x\frac{1}{x} = 1$. Matrix inversion is the generalization of this identity.

If $\boldsymbol{A} \in \mathbb{F}^{M\times N}$ and $\boldsymbol{B} \in \mathbb{F}^{N\times M}$ and $\boldsymbol{B}\boldsymbol{A} = \boldsymbol{I}_{N\times N}$ then we call $\boldsymbol{B}$ a **left inverse** of $\boldsymbol{A}$.

If and $\boldsymbol{C} \in \mathbb{F}^{N\times M}$ and $\boldsymbol{A}\boldsymbol{C} = \boldsymbol{I}_{M\times M}$ then we call $\boldsymbol{C}$ a **right inverse** of $\boldsymbol{A}$.

For non-square matrices, at most only one of a left or right inverse can exist, and is not unique.

A square matrix $\boldsymbol{A}$ is called **invertible** iff there exists a matrix $\boldsymbol{X}$ of the same size such that $\boldsymbol{A}\boldsymbol{X} = \boldsymbol{X}\boldsymbol{A} = \boldsymbol{I}$. Another name for invertibility is **nonsingular**.

Fact. For square matrices, a left inverse exists iff only a right inverse exists and the two are identical [wiki]. Proving this fact seems to be a bit subtle. See:
https://math.stackexchange.com/questions/216569/assuming-ab-i-prove-ba-i

Basic matrix inversion properties

If $\boldsymbol{A}$ is an **invertible matrix**, then:

- $(\boldsymbol{A}^{-1})^{-1} = \boldsymbol{A}$
- $c \neq 0 \implies (c\boldsymbol{A})^{-1} = \frac{1}{c}\boldsymbol{A}^{-1}$
- $\boldsymbol{A}'$ is also invertible and $(\boldsymbol{A}')^{-1} = (\boldsymbol{A}^{-1})'$
- $(\boldsymbol{A}\boldsymbol{B})^{-1} = \boldsymbol{B}^{-1}\boldsymbol{A}^{-1}$ if $\boldsymbol{B}$ is invertible and has same size as $\boldsymbol{A}$
- The equation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}$ has only the trivial solution $\boldsymbol{x} = \boldsymbol{0}$ iff $\boldsymbol{A}$ is invertible.

Trying to "avoid" matrix inversion (Read)

Typical methods for inverting a $N \times N$ matrix use $O(N^3)$ operations, which is very expensive for large matrices, so we often seek matrix identities to reduce computation when possible.

- The following inverse of 2×2 block matrices holds if $\boldsymbol{A}$ and $\boldsymbol{B}$ are invertible (and if $\boldsymbol{D}$,$\boldsymbol{C}$ sizes are appropriate):

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{D} \\ \boldsymbol{C} & \boldsymbol{B} \end{bmatrix}^{-1} = \begin{bmatrix} [\boldsymbol{A} - \boldsymbol{D}\boldsymbol{B}^{-1}\boldsymbol{C}]^{-1} & -\boldsymbol{A}^{-1}\boldsymbol{D}\boldsymbol{\Delta}^{-1} \\ -\boldsymbol{\Delta}^{-1}\boldsymbol{C}\boldsymbol{A}^{-1} & \boldsymbol{\Delta}^{-1} \end{bmatrix}, \tag{1.12}$$

  where $\boldsymbol{\Delta} = \boldsymbol{B} - \boldsymbol{C}\boldsymbol{A}^{-1}\boldsymbol{D}$ denotes the **Schur complement** of $\boldsymbol{A}$ for this matrix.
- Using the associative property and some rearranging yields the **Sherman-Morrison-Woodbury identity** also known as the **matrix inversion lemma**:

$$(\boldsymbol{A} + \boldsymbol{B}\boldsymbol{C}\boldsymbol{D})^{-1} = \boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{B}\left(\boldsymbol{C}^{-1} + \boldsymbol{D}\boldsymbol{A}^{-1}\boldsymbol{B}\right)^{-1}\boldsymbol{D}\boldsymbol{A}^{-1}, \tag{1.13}$$

  provided the matrices have compatible sizes and $\boldsymbol{A}$ and $\boldsymbol{C}$ are invertible.
- A special case that is useful if we have previously computed the inverse of $\boldsymbol{A}$ and now need the inverse of the "rank-one update" $\boldsymbol{A} + \boldsymbol{x}\boldsymbol{y}'$, if it is invertible, is the **Sherman–Morrison formula**:

$$(\boldsymbol{A} + \boldsymbol{x}\boldsymbol{y}')^{-1} = \boldsymbol{A}^{-1} - \frac{1}{1 + \boldsymbol{y}'\boldsymbol{A}^{-1}\boldsymbol{x}} \boldsymbol{A}^{-1}\boldsymbol{x}\boldsymbol{y}'\boldsymbol{A}^{-1}.$$

- A more general special case of (1.13) is when $\boldsymbol{U}$ and $\boldsymbol{V}$ are $N \times K$ matrices, leading to the following "rank-K" update formula:

$$(\boldsymbol{A} + \boldsymbol{U}\boldsymbol{V}')^{-1} = \boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{U}\left(\boldsymbol{I}_K + \boldsymbol{V}'\boldsymbol{A}^{-1}\boldsymbol{U}\right)^{-1}\boldsymbol{V}'\boldsymbol{A}^{-1},$$

provided $\boldsymbol{I}_K + \boldsymbol{V}' \boldsymbol{A}^{-1} \boldsymbol{U}$ is non-singular.

- Multiplying (1.13) on the right by $\boldsymbol{B}$ and simplifying yields the following useful related equality, sometimes called the **push-through identity**:

$$[\boldsymbol{A} + \boldsymbol{B}\boldsymbol{C}\boldsymbol{D}]^{-1} \boldsymbol{B} = \boldsymbol{A}^{-1} \boldsymbol{B} \left[\boldsymbol{D}\boldsymbol{A}^{-1}\boldsymbol{B} + \boldsymbol{C}^{-1}\right]^{-1} \boldsymbol{C}^{-1}. \tag{1.14}$$

In particular, if $\boldsymbol{C} = \boldsymbol{I}_N$ and $\boldsymbol{A} = a\boldsymbol{I}_M$ then

$$[a\boldsymbol{I}_M + \boldsymbol{B}\boldsymbol{D}]^{-1} \boldsymbol{B} = \boldsymbol{B} [a\boldsymbol{I}_N + \boldsymbol{D}\boldsymbol{B}]^{-1}, \tag{1.15}$$

which is especially useful if one of N or M is much smaller than the other.

Challenge: prove (or disprove) that if $\boldsymbol{A}$ and $\boldsymbol{C}$ are invertible,
then $(\boldsymbol{A} + \boldsymbol{B}\boldsymbol{C}\boldsymbol{D})$ is invertible iff $(\boldsymbol{C}^{-1} + \boldsymbol{D}\boldsymbol{A}^{-1}\boldsymbol{B})$ is invertible, so that (1.13) is self consistent.

More invertible matrix properties (Read)

The following properties of any **invertible** matrix $\boldsymbol{A}$ relate to topics *discussed later in the chapter*:

- $\boldsymbol{A}$ has linearly independent columns.
- $\boldsymbol{A}$ has full rank, *i.e.*, all of its eigenvalues are nonzero.
- If $\boldsymbol{A}$ is unitary then $\boldsymbol{A}^{-1} = \boldsymbol{A}'$.
- The determinant of $\boldsymbol{A}$ is nonzero and $\det\{\boldsymbol{A}^{-1}\} = 1/\det\{\boldsymbol{A}\}$.

8. If $\boldsymbol{B}$ is a $N \times N$ matrix and $\boldsymbol{X}\boldsymbol{B} = \boldsymbol{I}_N$ and $\boldsymbol{B}\boldsymbol{Y} = \boldsymbol{I}_N$, then $\boldsymbol{X} = \boldsymbol{Y}$.

A: True B: False $\boxed{??}$

1.4 Orthogonality

For ordinary scalars: $0 \cdot x = 0$.
For vectors and matrices, there are more interesting ways to multiply and end up with zero!

Orthogonal vectors

L§1.3

Define. We say two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ (of same length) are **orthogonal** (or **perpendicular**) if their inner product is zero: $\boldsymbol{x}'\boldsymbol{y} = 0$. In such cases we write $\boldsymbol{x} \perp \boldsymbol{y}$.
If, in addition, $\boldsymbol{x}'\boldsymbol{x} = \boldsymbol{y}'\boldsymbol{y} = 1$ (*i.e.*, both **unit norm**), then we call them **orthonormal** vectors.

Define. A collection of vectors that are pairwise orthogonal is called an **orthogonal set**.
A collection of **unit-norm** vectors that are pairwise orthogonal is called an **orthonormal set**.

Example. In $\mathbb{R}^3$, the vectors $\boldsymbol{v}_1 = (4, 2, 0)$ and $\boldsymbol{v}_2 = (-1, 2, 0)$ are orthogonal (but not orthonormal).

Euclidean norm

Define. The (Euclidean) **norm** of a vector $\boldsymbol{x} \in \mathbb{F}^N$ is defined by

$$\|\boldsymbol{x}\| = \sqrt{\langle \boldsymbol{x}, \boldsymbol{x} \rangle} = \sqrt{\boldsymbol{x}'\boldsymbol{x}} = \sqrt{\sum_{n=1}^N |x_n|^2}. \tag{1.16}$$

Later we will write $\|\boldsymbol{x}\|_2$ when needed, but for now we focus on the Euclidean norm, aka **2 norm**.

Properties of the Euclidean norm (Read)

- Homogeneity: $\|\alpha \boldsymbol{x}\| = |\alpha| \, \|\boldsymbol{x}\|$ for $\alpha \in \mathbb{F}$
- **Triangle inequality**: $\|\boldsymbol{u} + \boldsymbol{v}\| \leq \|\boldsymbol{u}\| + \|\boldsymbol{v}\|$
- Quadratic expansion:
$$\|\boldsymbol{u} + \boldsymbol{v}\|_2^2 = \|\boldsymbol{u}\|_2^2 + 2\,\text{real}\{\boldsymbol{u}'\boldsymbol{v}\} + \|\boldsymbol{v}\|_2^2 \tag{1.17}$$
Proof: $\|\boldsymbol{u} + \boldsymbol{v}\|_2^2 = \sum_{i=1}^n |u_i + v_i|^2 = \sum_{i=1}^n (u_i + v_i)(u_i + v_i)^* = \sum_{i=1}^n |u_i|^2 + 2\,\text{real}\{u_i^* v_i\} + |v_i|^2$
- **Pythagorean theorem**: $\boldsymbol{u} \perp \boldsymbol{v} \iff \|\boldsymbol{u} + \boldsymbol{v}\|_2^2 = \|\boldsymbol{u}\|_2^2 + \|\boldsymbol{v}\|_2^2$
- Stacking property: $\left\| \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{bmatrix} \right\|_2^2 = \|\boldsymbol{u}\|_2^2 + \|\boldsymbol{v}\|_2^2 = \sum_i |u_i|^2 + \sum_j |v_j|^2$

Cauchy-Schwarz inequality

The **Cauchy-Schwarz inequality** (or **Schwarz** or **Cauchy-Bunyakovsky-Schwarz** inequality) relates inner products and norms as follows:

(1.18)

where $\|\cdot\|_2$ denotes the Euclidean norm on $\mathbb{F}^N$.

For a proof, see Ch. 5.

Angle between vectors

Define. The **angle** θ between two nonzero vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{V}$ is defined by

The Cauchy-Schwarz inequality is equivalent to the statement $|\cos\theta| \leq 1$.

9. What is the angle between two orthogonal vectors?

A: 0 B: 1 C: $\pi/2$ D: π E: None of these $\boxed{??}$

Orthogonal matrices

Define. We say a (square) matrix $\boldsymbol{Q} \in \mathbb{R}^{N \times N}$ is an **orthogonal matrix** iff $\boldsymbol{Q}^\top \boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{Q}^\top = \boldsymbol{I}$.

(Personally I think the term "orthonormal matrix" would be more appropriate, but alas.)

Define. We say a (usually complex) square matrix $\boldsymbol{Q} \in \mathbb{C}^{N \times N}$ is a **unitary matrix** iff $\boldsymbol{Q}'\boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{Q}' = \boldsymbol{I}$.

The set of columns of a **unitary matrix** is an **orthonormal set**. (So is the set of rows.)
The set of columns of an **orthogonal matrix** is **orthonormal set**. (So is the set of rows.)

A interesting generalization is a **tight frame** where only one of the two conditions holds [14, 15]. ♦♦

Unitary / orthogonal matrices are a key building block in this course.

10. If a (possibly complex) matrix $\boldsymbol{A}$ has orthonormal columns, then it is unitary. (?)
A: True B: False $\boxed{??}$

11. If a square, real matrix $\boldsymbol{A}$ has orthogonal columns, then it is an orthogonal matrix. (?)
A: True B: False $\boxed{??}$

Every orthogonal matrix can be decomposed into a product of **Givens rotation** matrices and **Householder reflection** matrices. So intuitively, an $N \times N$ orthogonal matrix generalizes rotation and reflection operations. ♦♦

Invertibility of unitary matrices (Read)

A key property of orthogonal and unitary matrices is that their inverse is simply their transpose: $\boldsymbol{Q}^{-1} = \boldsymbol{Q}'$.

In other words, the easiest (non-diagonal) matrices to invert are unitary matrices.

Recall $\boldsymbol{Q}$ is unitary iff

$$\boldsymbol{Q}'\boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{Q}' = \boldsymbol{I}. \tag{1.19}$$

Thus by definition of matrix inverse we have

$$\boldsymbol{Q}^{-1} = \boldsymbol{Q}'. \tag{1.20}$$

Because of the equivalence of the left and right inverses for invertible square matrices, we really only need one of the two conditions in (1.19) to conclude the other and (1.20). So often people just write $\boldsymbol{Q}'\boldsymbol{Q} = \boldsymbol{I}$ when mentioning that a (square) matrix $\boldsymbol{Q}$ is unitary.

Norm invariance to rotations

Orthogonal/unitary matrices act somewhat like rotation operations. An important property of $N \times N$ **orthogonal** or **unitary** matrices is that they do not change the **Euclidean norm** of a vector:

$$\boldsymbol{Q} \text{ orthogonal or unitary } \Longrightarrow \qquad \forall \boldsymbol{x} \in \mathbb{C}^N. \tag{1.21}$$

Proof: $\|\boldsymbol{Q}\boldsymbol{x}\| = \sqrt{(\boldsymbol{Q}\boldsymbol{x})'(\boldsymbol{Q}\boldsymbol{x})} = \sqrt{\boldsymbol{x}'\boldsymbol{Q}'\boldsymbol{Q}\boldsymbol{x}} = \sqrt{\boldsymbol{x}'\boldsymbol{I}\boldsymbol{x}} = \sqrt{\boldsymbol{x}'\boldsymbol{x}} = \|\boldsymbol{x}\|$, using the orthogonality of $\boldsymbol{Q}$.

This fact is related to **Parseval's theorem**.

1.5 Determinant of a matrix

L§1.4

Now we begin discussing important matrix properties. We start with the matrix **determinant** [1, Sect. 1.4], a property that is defined only for square matrices. Data matrices are very rarely square, so we essentially never examine the determinant of a data matrix directly! But if $\boldsymbol{X}$ is a $M \times N$ data matrix, often we will work with the $N \times N$ **Gram matrix** $\boldsymbol{X}'\boldsymbol{X}$ (*e.g.*, when solving least-squares problems) and a Gram matrix is always square. Many operator matrices (like DFT) are also square.

There a many ways to introduce the determinant of a matrix because it has many properties[1]. Here we consider the following four "axioms" expressed in terms of matrices.

Define. If $\boldsymbol{A} \in \mathbb{F}^{N\times N}$ then the determinant of $\boldsymbol{A}$ is defined so that
- D1: If $\boldsymbol{A}$ is upper triangular and $N \times N$ then $\det\{\boldsymbol{A}\} = a_{11} \cdot a_{22} \cdot \cdots \cdot a_{NN}$
- D2: If $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{F}^{N\times N}$ then $\det\{\boldsymbol{A}\boldsymbol{B}\} = \det\{\boldsymbol{A}\} \det\{\boldsymbol{B}\}$
- D3: $\det\{\boldsymbol{A}'\} = \det\{\boldsymbol{A}\}^*$ where z^* denotes the complex conjugate of z.
- D4: If $\boldsymbol{P}_{i,j} \in \mathbb{R}^{N\times N}$ denotes the matrix that swaps the ith and jth rows, $i \neq j$, then $\det\{\boldsymbol{P}_{i,j}\} = -1$.

Geometrically, the determinant of a matrix is the (signed) **volume of the parallelpiped** defined by its column vectors. The determinant of the **Jacobian matrix** arises in multivariate calculus for **integration by substitution**.

[1] [wiki] lists about 13 properties and claims that a certain set of 3 of them completely characterize the determinant. For our purposes there is no need to make more work for ourselves by using a minimal set so we start with 4 axioms instead.

Fact. A matrix $\boldsymbol{A}$ is **invertible** iff its determinant is nonzero.

So a linear system of N equations with N unknowns, *e.g.*, $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, has a unique solution iff the **determinant** corresponding to the coefficients is nonzero. This is the historical reason for the term **determinant** because it "determines" uniqueness of the solution to such a set of equations.

From D2, the following **determinant commutative property** follows immediately:

$$\boldsymbol{A}, \boldsymbol{B} \in \mathbb{F}^{N \times N} \Longrightarrow \qquad (1.22)$$

A concise formula for the row-swapping matrix $\boldsymbol{P}_{i,j}$ (a special **permutation matrix**) used in D4 is

$$\boldsymbol{P}_{i,j} = \boldsymbol{I} - \boldsymbol{e}_j \boldsymbol{e}_j' - \boldsymbol{e}_i \boldsymbol{e}_i' + \boldsymbol{e}_j \boldsymbol{e}_i' + \boldsymbol{e}_i \boldsymbol{e}_j',$$

for $i, j \in \{1, \ldots, N\}$, where $\boldsymbol{e}_i$ denotes the ith unit vector.

Example. For $N = 5$: $\boldsymbol{P}_{1,4} = \boldsymbol{P}_{4,1} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$, so $\boldsymbol{P}_{1,4} \begin{bmatrix} \boldsymbol{A}_{1,:} \\ \boldsymbol{A}_{2,:} \\ \boldsymbol{A}_{3,:} \\ \boldsymbol{A}_{4,:} \\ \boldsymbol{A}_{5,:} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{4,:} \\ \boldsymbol{A}_{2,:} \\ \boldsymbol{A}_{3,:} \\ \boldsymbol{A}_{1,:} \\ \boldsymbol{A}_{5,:} \end{bmatrix}$.

In words, left multiplying a matrix by $\boldsymbol{P}_{i,j}$ swaps the ith and jth rows of the matrix.

Row (or column) swap property

A consequence of D2 and D4 is that swapping any two rows of a matrix negates the sign of the determinant:

$$i \neq j \Longrightarrow$$

By D3, the same property holds for swapping two columns.

Column (or row) scaling property

Use the above four axioms to prove that multiplying a column of $\boldsymbol{A}$ by a scalar gives a new matrix whose determinant scales by that scalar factor:

$$\boldsymbol{B} = [\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{n-1}, b\boldsymbol{a}_n, \boldsymbol{a}_{n+1}, \ldots, \boldsymbol{a}_N] \Longrightarrow \mathsf{det}\{\boldsymbol{B}\} = b \, \mathsf{det}\{\boldsymbol{A}\}.$$

Proof. Simply write $\boldsymbol{B} = \boldsymbol{A}\boldsymbol{D}$ where $\boldsymbol{D} = \mathsf{Diag}\{1, \ldots, 1, b, 1, \ldots, 1\}$,
so $\mathsf{det}\{\boldsymbol{B}\} = \mathsf{det}\{\boldsymbol{A}\} \, \mathsf{det}\{\boldsymbol{D}\} = b \, \mathsf{det}\{\boldsymbol{A}\}$.

Note the strategy of writing (linear) operations in terms of matrix products so we can apply D2.

Matrix inversion property

Exercise. Use two of the above "axioms" to prove $\mathsf{det}\{\boldsymbol{A}^{-1}\} = 1/\,\mathsf{det}\{\boldsymbol{A}\}$.

Transpose property

$\mathsf{det}\{\boldsymbol{A}^T\} = \mathsf{det}\{\boldsymbol{A}\}$

Scaling property

$\det\{c\boldsymbol{A}\} = c^N \det\{\boldsymbol{A}\} \, \forall \boldsymbol{A} \in \mathbb{F}^{N\times N}, \ c \in \mathbb{F}$

Row (or column) combination property

Multiplying a row of a matrix by a scalar and adding it to a different row does not change the determinant [1, property 8, p. 5], *i.e.*:

$$\text{if } \boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{1,:} \\ \vdots \\ \boldsymbol{A}_{M,:} \end{bmatrix} \text{ and } \boldsymbol{B} = \begin{bmatrix} \boldsymbol{A}_{1,:} \\ \vdots \\ \boldsymbol{A}_{m-1,:} \\ b\boldsymbol{A}_{k,:} + \boldsymbol{A}_{m,:} \\ \boldsymbol{A}_{m+1,:} \\ \vdots \\ \boldsymbol{A}_{M,:} \end{bmatrix}, \text{ with } k \neq m, \text{ then } \det\{\boldsymbol{B}\} = \det\{\boldsymbol{A}\}.$$

Proof: $\boldsymbol{B} = \boldsymbol{A} + b\boldsymbol{e}_m\boldsymbol{e}_k'\boldsymbol{A} = (\boldsymbol{I} + b\boldsymbol{e}_m\boldsymbol{e}_k')\boldsymbol{A} \Longrightarrow \det\{\boldsymbol{B}\} = \det\{\boldsymbol{I} + b\boldsymbol{e}_m\boldsymbol{e}_k'\} \det\{\boldsymbol{A}\} = \det\{\boldsymbol{A}\}$.

Note that $\boldsymbol{e}_k'\boldsymbol{A} = \boldsymbol{A}_{k,:}$ and think about the **outer product** $\boldsymbol{e}_m(\boldsymbol{e}_k'\boldsymbol{A}) = \boldsymbol{e}_m\boldsymbol{A}_{k,:}$.

Exercise. Where did this proof use the assumption that $k \neq m$?

Exercise. What is $\det\{\boldsymbol{B}\}$ if $k = m$? $\boxed{\textbf{??}}$

12. A $M \times N$ matrix $\boldsymbol{X}$ has orthornormal columns; the determinant of the Gram matrix $\boldsymbol{X}'\boldsymbol{X}$ is:
A: 0 B: 1 C: N D: M E: None of these **??**

Determinant formula (Read)

A general rule, called **Laplace's formula**, for a $N \times N$ matrix $\boldsymbol{A}$ is:

$$\det\{\boldsymbol{A}\} = \sum_{n=1}^{N} (-1)^{m+n} a_{m,n} \det\{\boldsymbol{A}_{m,n}\},$$

for any $m \in \{1, \ldots, N\}$, where here $\boldsymbol{A}_{m,n}$ denotes the submatrix of $\boldsymbol{A}$ formed by deleting the mth row and nth column and $\det\{\boldsymbol{A}_{m,n}\}$ is called a **minor** of $\boldsymbol{A}$. (I have rarely needed to use this.)

See above Wikipedia link for an example.

Avoiding computation

Naive methods for computing the determinant of a $N \times N$ matrix would require $O(N!)$ operations, but more efficient **decomposition methods** require $O(N^3)$ operations (or less). This is still expensive for large N, so we often seek properties to use to reduce computation when possible.

- Generalizing property D2 on p. 1.60, if $\boldsymbol{A}$ is **block upper triangular** (or **block lower triangular**), with square blocks, then its determinant is the product of the determinants of its diagonal blocks:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & \boldsymbol{A}_{13} & \dots \\ 0 & \boldsymbol{A}_{22} & \boldsymbol{A}_{23} & \dots \\ \vdots & & \ddots & \\ 0 & \dots & 0 & \boldsymbol{A}_{KK} \end{bmatrix} \Longrightarrow \det\{\boldsymbol{A}\} = \qquad (1.23)$$

- Using **block matrix triangularization** and (1.23), if $\boldsymbol{A}$ is invertible then [16]:

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} & 0 \\ \boldsymbol{C} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & \boldsymbol{A}^{-1}\boldsymbol{B} \\ 0 & \boldsymbol{D} - \boldsymbol{C}\boldsymbol{A}^{-1}\boldsymbol{B} \end{bmatrix} \Longrightarrow \det\left\{\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix}\right\} = \qquad (1.24)$$

Similarly if $\boldsymbol{D}$ is invertible then we can find the determinant of a large block matrix in terms of determinants of combinations of its blocks:

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{B} \\ 0 & \boldsymbol{D} \end{bmatrix} \begin{bmatrix} \boldsymbol{A} - \boldsymbol{B}\boldsymbol{D}^{-1}\boldsymbol{C} & 0 \\ \boldsymbol{D}^{-1}\boldsymbol{C} & \boldsymbol{I} \end{bmatrix} \Longrightarrow \det\left\{\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix}\right\} = \qquad (1.25)$$

- The **matrix determinant lemma** says if $\boldsymbol{A}$ is square and invertible and $\boldsymbol{x}$ and $\boldsymbol{y}$ have the appropriate size then the determinant of the "rank-one update" is: (proof)

$$\det\{\boldsymbol{A} + \boldsymbol{x}\boldsymbol{y}'\} = (1 + \boldsymbol{y}'\boldsymbol{A}^{-1}\boldsymbol{x}) \det\{\boldsymbol{A}\} . \tag{1.26}$$

This property can be useful if the inverse and determinant of $\boldsymbol{A}$ are already known.

Example. Find the determinant of $\boldsymbol{B} = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} =$.

Check: `det([2 1 1; 1 2 1; 1 1 2])`

- **Sylvester's determinant identity** can be useful for rectangular matrices $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{F}^{M \times N}$ with $N \ll M$:

$$\det\{\boldsymbol{I}_M + \boldsymbol{A}\boldsymbol{B}'\} = \det\{\boldsymbol{I}_N + \boldsymbol{B}'\boldsymbol{A}\} . \tag{1.27}$$

- More generally:

$$\boldsymbol{A}, \boldsymbol{B} \in \mathbb{F}^{M \times N}, \boldsymbol{X} \in \mathbb{F}^{M \times M} \text{ invertible } \Longrightarrow \det\{\boldsymbol{X} + \boldsymbol{A}\boldsymbol{B}'\} = \det\left\{\boldsymbol{I}_N + \boldsymbol{B}'\boldsymbol{X}^{-1}\boldsymbol{A}\right\} \det\{\boldsymbol{X}\} .$$

The proof follows from (1.24) and (1.25).

Practical use in JULIA:

`using LinearAlgebra` or `using LinearAlgebra: det`

then `det(A)`

2 by 2 matrix (Read)

Use the above properties to show that the determinant of a 2×2 matrix $\boldsymbol{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ is $ad - bc$.

Proof. If a is nonzero, then multiplying the first row by $-c/a$ and adding to the second row yields:
$\boldsymbol{B} = \begin{bmatrix} a & b \\ 0 & d - bc/a \end{bmatrix}$, which is upper triangular so has determinant $a(d - bc/a) = ad - bc$.

If a is zero, then swapping the first and second rows yields $\boldsymbol{C} = \begin{bmatrix} c & d \\ 0 & b \end{bmatrix}$ which again is upper triangular and the determinant is cb, so the determinant of $\boldsymbol{A}$ in this case is $-cb = 0d - cb = ad - cb$. □

Example. $\det\left\{ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} = 0 \cdot 0 - 1 \cdot 1 = -1$, consistent with D4.

Determinant of 3×3 matrix (Read)

If $\boldsymbol{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$ then $\det\{\boldsymbol{A}\} = a \det\left\{ \begin{bmatrix} e & f \\ h & i \end{bmatrix} \right\} - b \det\left\{ \begin{bmatrix} d & f \\ g & i \end{bmatrix} \right\} + c \det\left\{ \begin{bmatrix} d & e \\ g & h \end{bmatrix} \right\}$.
You should verify this yourself from the properties.

1.6 Eigenvalues

L§9.1

Define. An important property of any square matrix $\boldsymbol{A} \in \mathbb{F}^{N \times N}$ is its **eigenvalues**, often denoted $\lambda_1, \ldots, \lambda_N$, defined as the set of solutions of the **characteristic equation**

$$\det\{\boldsymbol{A} - z\boldsymbol{I}\} = 0, \tag{1.28}$$

where $\boldsymbol{I}$ denotes the identity matrix of the same size as $\boldsymbol{A}$.

Viewed as a function of z, we call $\det\{\boldsymbol{A} - z\boldsymbol{I}\}$ the **characteristic polynomial**, and the eigenvalues of $\boldsymbol{A}$ are its roots.

When $\boldsymbol{A} \in \mathbb{F}^{N \times N}$, the characteristic polynomial has **degree** N and thus N (possibly complex) roots by the **fundamental theorem of algebra**. Thus by the definition (1.28), $\boldsymbol{A}$ has N eigenvalues (but they are not necessarily distinct).

The literature can be inconsistent about how many eigenvalues a $N \times N$ matrix has. For example, the characteristic polynomial of the 2×2 matrix $\boldsymbol{A} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ is $\det\{\boldsymbol{A} - z\boldsymbol{I}\} = z^2 = (z - 0)(z - 0)$ which has a repeated root at zero and sometimes is said to have "only one eigenvalue" [wiki]. We will not use that terminology; we will say that matrix has two eigenvalues, both of which are zero.

Example. Determine the eigenvalues of $\boldsymbol{A} = \boldsymbol{I}_N + \boldsymbol{y}\boldsymbol{y}'$ for $\boldsymbol{y} \in \mathbb{F}^N$. Using (1.26):

$$\begin{aligned}\det\{\boldsymbol{A} - z\boldsymbol{I}_N\} &= \det\{\boldsymbol{I}_N + \boldsymbol{y}\boldsymbol{y} - z\boldsymbol{I}_N\} = \det\{(1-z)\boldsymbol{I}_N + \boldsymbol{y}\boldsymbol{y}'\} \\ &= \left(1 + \boldsymbol{y}' \left((1-z)\boldsymbol{I}_N\right)^{-1} \boldsymbol{y}\right) \det\{(1-z)\boldsymbol{I}_N\} \\ &= \left(1 + \|\boldsymbol{y}\|_2^2 / (1-z)\right) (1-z)^N = (1-z)^{N-1} \left(1 - z + \|\boldsymbol{y}\|^2\right),\end{aligned}$$

which has $N-1$ roots (eigenvalues) at 1 and one root (eigenvalue) at $1 + \|\boldsymbol{y}\|^2$.

Square vs rectangular matrices

As mentioned earlier, data matrices are nearly never square so basically we will never need to examine the eigenvalues of a data matrix $\boldsymbol{X} \in \mathbb{F}^{M \times N}$ directly. But often we will consider the eigenvalues of the square $M \times M$ **Gram matrix** $\boldsymbol{X}'\boldsymbol{X}$ or of the square $N \times N$ **outer-product matrix** $\boldsymbol{X}\boldsymbol{X}'$ that arises when estimating a **covariance matrix** or a **scatter matrix**.

Eigenvectors

If z is an eigenvalue of $\boldsymbol{A}$, then (1.28) implies $\boldsymbol{A} - z\boldsymbol{I}$ is a **singular matrix** (not invertible), so there exists a nonzero vector $\boldsymbol{v}$ such that $(\boldsymbol{A} - z\boldsymbol{I})\boldsymbol{v} = \boldsymbol{0}$ or equivalently

$$\boldsymbol{A}\boldsymbol{v} = z\boldsymbol{v}. \tag{1.29}$$

Conversely, if (1.29) holds for a nonzero vector $\boldsymbol{v}$, then z is an eigenvalue of $\boldsymbol{A}$.

Define. Any nonzero vector $\boldsymbol{v}$ that satisfies (1.29) is called an **eigenvector** of $\boldsymbol{A}$.

Example. For the matrix $\boldsymbol{A} = \boldsymbol{I}_N + \boldsymbol{y}\boldsymbol{y}'$ for $\boldsymbol{y} \in \mathbb{F}^N$, one eigenvector is $\boldsymbol{y}$ because $\boldsymbol{A}\boldsymbol{y} = (\boldsymbol{I}_N + \boldsymbol{y}\boldsymbol{y}')\boldsymbol{y} = (\boldsymbol{y}+\boldsymbol{y}\boldsymbol{y}'\boldsymbol{y}) = (1+\|\boldsymbol{y}\|^2)\boldsymbol{y}$. Any vector of the form $\alpha\boldsymbol{y}$ for $\alpha \in \mathbb{F}$ is also an eigenvector. All other eigenvectors are orthogonal to $\boldsymbol{y}$ because if $\boldsymbol{x} \perp \boldsymbol{y}$ then $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{x}$.

For a $N \times N$ matrix $\boldsymbol{A}$, let $\lambda_1, \ldots, \lambda_N$ denote its N eigenvalues. For each eigenvalue λ_i, there is a corresponding eigenvector $\boldsymbol{v}_i$. So we have N equations of the form (1.29):

$$\boldsymbol{A}\boldsymbol{v}_1 = \lambda_1\boldsymbol{v}_1, \quad \ldots, \quad \boldsymbol{A}\boldsymbol{v}_N = \lambda_N\boldsymbol{v}_N.$$

It is convenient to collect these N matrix-vector equations into a single important matrix-matrix equation:

$$\boldsymbol{A}\boldsymbol{V} = \boldsymbol{V}\boldsymbol{\Lambda}, \qquad \boldsymbol{V} \triangleq \begin{bmatrix} \boldsymbol{v}_1 & \ldots & \boldsymbol{v}_N \end{bmatrix}, \qquad \boldsymbol{\Lambda} \triangleq \mathsf{Diag}\{\lambda_1, \ldots, \lambda_N\}. \tag{1.30}$$

The $N \times N$ matrix $\boldsymbol{V}$ has the N eigenvectors as its columns, and the $N \times N$ matrix $\boldsymbol{\Lambda}$ has the N eigenvalues along its diagonal, and their product has the scaled eigenvectors as its columns:

$$\boldsymbol{V}\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_1 \boldsymbol{v}_1 & \dots & \lambda_N \boldsymbol{v}_N \end{bmatrix}.$$

Practical implementation

To find a set of eigenvalues and eigenvectors of a square matrix $\boldsymbol{A}$ in JULIA:

```
using LinearAlgebra
z,V = eigen(A)
```

To obtain just the eigenvalues use any of

```
eigvals(A)
eigen(A).values
z,_ = eigen(A)
```

The underscore `_` output is discarded.

To obtain just a set of eigenvectors use any of

```
eigvecs(A)
eigen(A).vectors
_,V = eigen(A)
```

The usual notation for an eigenvalue is λ and *when the eigenvalues are all real*, by convention we often choose to order them such that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_N$. But the `eigvals` and `eigen` commands return the eigenvalues in essentially arbitrary order (depends on the matrix type), usually not decreasing; see **eigen** documentation.

Example. Find the eigenvalues of $\boldsymbol{A} = \begin{bmatrix} 2 & 1 \\ -1 & 2 \end{bmatrix}$. (Read)

The characteristic polynomial is $\det\{\boldsymbol{A} - z\boldsymbol{I}\} = \det\left\{\begin{bmatrix} 2-z & 1 \\ -1 & 2-z \end{bmatrix}\right\} = (2-z)(2-z)+1 = z^2 - 4z + 5$,
which has roots $z = 2 \pm \imath$, so the two eigenvalues of $\boldsymbol{A}$ are $\lambda_1 = 2 + \imath,\ \lambda_2 = 2 - \imath$, where $\imath = \sqrt{-1}$.
To find the eigenvectors, note that (by inspection):

$$(\boldsymbol{A} - \lambda_1 \boldsymbol{I})\ \boldsymbol{v}_1 = \begin{bmatrix} -\imath & 1 \\ -1 & -\imath \end{bmatrix} \boldsymbol{v}_1 = \boldsymbol{0} \text{ when } \boldsymbol{v}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ \imath \end{bmatrix}$$
$$(\boldsymbol{A} - \lambda_2 \boldsymbol{I})\ \boldsymbol{v}_2 = \begin{bmatrix} \imath & 1 \\ -1 & \imath \end{bmatrix} \boldsymbol{v}_2 = \boldsymbol{0} \text{ when } \boldsymbol{v}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -\imath \end{bmatrix}.$$

JULIA check: `eigen([2 1; -1 2])`

Properties of eigenvalues

L§9.1

The defining properties of determinant on p. 1.60 lead to useful eigenvalue properties.

- D1 $\Longrightarrow$ If $\boldsymbol{A}$ is upper triangular, then the eigenvalues of $\boldsymbol{A}$ are its diagonal elements.
- D3 $\Longrightarrow$ The eigenvalues of $\boldsymbol{A}'$ are the complex conjugates of the eigenvalues of $\boldsymbol{A}$.
- D2 $\Longrightarrow$ The eigenvalues of $\alpha\boldsymbol{A}$ are α times the eigenvalues of $\boldsymbol{A}$ for $\alpha \in \mathbb{F}$.
- D2 $\Longrightarrow$ The eigenvalues of $\boldsymbol{A}$ are invariant to similarity transforms [1, Theorem 9.19].

  If $\boldsymbol{A}$ is $N \times N$ and $\boldsymbol{T}$ is an $N \times N$ invertible matrix then $\boldsymbol{T}\boldsymbol{A}\boldsymbol{T}^{-1}$ has the same eigenvalues as $\boldsymbol{A}$.

  As a special case, if $\boldsymbol{Q}$ is $N \times N$ is **orthogonal** (or **unitary** in complex case) matrix, then $\boldsymbol{Q}\boldsymbol{A}\boldsymbol{Q}'$ has the same eigenvalues as $\boldsymbol{A}$.

  Proof. $\det\{\boldsymbol{T}\boldsymbol{A}\boldsymbol{T}^{-1} - z\boldsymbol{I}\} = \det\{\boldsymbol{T}\boldsymbol{A}\boldsymbol{T}^{-1} - z\boldsymbol{T}\boldsymbol{T}^{-1}\} = \det\{\boldsymbol{T}(\boldsymbol{A} - z\boldsymbol{I})\boldsymbol{T}^{-1}\} = \det\{(\boldsymbol{A} - z\boldsymbol{I})\boldsymbol{T}^{-1}\boldsymbol{T}\}$
  $= \det\{(\boldsymbol{A} - z\boldsymbol{I})\boldsymbol{I}\} = \det\{\boldsymbol{A} - z\boldsymbol{I}\}$. So $\boldsymbol{T}\boldsymbol{A}\boldsymbol{T}^{-1}$ and $\boldsymbol{A}$ have the same characteristic equation.
  Here we used the **distributive property** of matrix multiplication.
- The determinant of any square matrix is the product of its eigenvalues [1, Theorem 9.25]:

  $$\boldsymbol{A} \in \mathbb{F}^{N \times N} \Longrightarrow \det\{\boldsymbol{A}\} = \prod_{i=1}^{N} \lambda_i(\boldsymbol{A}).$$
- $\boldsymbol{A}$ is **invertible** iff all of its eigenvalues are nonzero.

More product properties...

If $\boldsymbol{A}$ has **eigenvalues** $\{\lambda_1, \ldots, \lambda_N\}$, then $\boldsymbol{A}^2$ has eigenvalues $\{\lambda_1^2, \ldots, \lambda_N^2\}$ and $\boldsymbol{A}^k$ has eigenvalues $\left\{\lambda_1^k, \ldots, \lambda_N^k\right\}$ for $k \in \mathbb{N}$.

Proof: $\boldsymbol{A}\boldsymbol{V} = \boldsymbol{V}\boldsymbol{\Lambda} \Longrightarrow \boldsymbol{A}^2\boldsymbol{V} = \boldsymbol{A}(\boldsymbol{A}\boldsymbol{V}) = \boldsymbol{A}(\boldsymbol{V}\boldsymbol{\Lambda}) = (\boldsymbol{A}\boldsymbol{V})\boldsymbol{\Lambda} = (\boldsymbol{V}\boldsymbol{\Lambda})\boldsymbol{\Lambda} = \boldsymbol{V}\boldsymbol{\Lambda}^2$.

Define. Let $\mathcal{S} - \{\boldsymbol{x}\}$ denote the set $\mathcal{S}$ with the vector $\boldsymbol{x}$ removed.

Suppose $\boldsymbol{A} \in \mathbb{F}^{M \times N}$ and $\boldsymbol{B} \in \mathbb{F}^{N \times M}$.
If z is a *nonzero* eigenvalue of $\boldsymbol{A}\boldsymbol{B}$, then z is also a *nonzero* eigenvalue of $\boldsymbol{B}\boldsymbol{A}$.
We summarize this concisely as the following **commutative property** for eigenvalues:

(1.31)

i.e., the *nonzero* elements of each set of eigenvalues are the same.

Proof. $\boldsymbol{A}\boldsymbol{B}\boldsymbol{v} = z\boldsymbol{v}$ for some nonzero $\boldsymbol{v}$, then clearly $\boldsymbol{u} \triangleq \boldsymbol{B}\boldsymbol{v}$ is also nonzero. Multiplying by $\boldsymbol{B}$ yields $\boldsymbol{B}\boldsymbol{A}\boldsymbol{B}\boldsymbol{v} = z\boldsymbol{B}\boldsymbol{v} \Longrightarrow \boldsymbol{B}\boldsymbol{A}\boldsymbol{u} = z\boldsymbol{u}$ where $\boldsymbol{u}$ is nonzero, so z is a nonzero eigenvalue of $\boldsymbol{B}\boldsymbol{A}$. $\square$

13.

What is the set of nonzero eigenvalues of the outer product matrix $\boldsymbol{A} = \begin{bmatrix}1\\2\\3\end{bmatrix}\begin{bmatrix}1 & 1 & 1\end{bmatrix}$?

A: $\{1, 2, 3\}$ B: $\{1, 2\}$ C: $\{1\}$ D: $\{2\}$ E: $\{6\}$ $\boxed{\textbf{??}}$

Here is another simple proof of (1.31) that goes father by showing that the nonzero eigenvalues of $\boldsymbol{CB}$ and $\boldsymbol{BC}$ have the same **multiplicity**, when $\boldsymbol{C} \in \mathbb{F}^{N \times M}$ and $\boldsymbol{B} \in \mathbb{F}^{M \times N}$. (By Yifan Shen in F19.)

Let $\boldsymbol{A} = \sqrt{\lambda}\boldsymbol{I}_M$ and $\boldsymbol{D} = \sqrt{\lambda}\boldsymbol{I}_N$ in (1.24) and (1.25), so that

$$\begin{aligned}
\det\{\boldsymbol{A}\}\det\{\boldsymbol{D} - \boldsymbol{C}\boldsymbol{A}^{-1}\boldsymbol{B}\} &= \det\{\boldsymbol{D}\}\det\{\boldsymbol{A} - \boldsymbol{B}\boldsymbol{D}^{-1}\boldsymbol{C}\} \\
\Longrightarrow \lambda^{M/2}\det\left\{\sqrt{\lambda}\boldsymbol{I}_N - \lambda^{-1/2}\boldsymbol{CB}\right\} &= \lambda^{N/2}\det\left\{\sqrt{\lambda}\boldsymbol{I}_M - \lambda^{-1/2}\boldsymbol{BC}\right\} \\
\Longrightarrow \lambda^{(M-N)/2}\det\{\lambda\boldsymbol{I}_N - \boldsymbol{CB}\} &= \lambda^{(N-M)/2}\det\{\lambda\boldsymbol{I}_M - \boldsymbol{BC}\}.
\end{aligned}$$

So all the nonzero eigenvalues $\boldsymbol{CB}$ and $\boldsymbol{BC}$ are the same and have the same multiplicities. □

I usually use (1.31) to look for the smallest and or largest eigenvalue, for which multiplicity is unimportant.

(Read)

Exercise. Find the eigenvalues of $\boldsymbol{AB} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -2 & 2 & -2 & 2 \end{bmatrix}$. Solution: $\boldsymbol{BA} = \begin{bmatrix} 4 & 2 \\ 0 & 4 \end{bmatrix}$. So the only nonzero eigenvalue of $\boldsymbol{BA}$ is 4 (repeated) and thus the only nonzero eigenvalue of $\boldsymbol{AB}$ is also 4. Using the proof on the preceding page, we know that $\text{eig}\{\boldsymbol{AB}\} = (4, 4, 0, 0)$

1.7 Trace

Another important matrix property is its **trace**.

Define. The **trace** of a square matrix, denoted $\operatorname{trace}\{\boldsymbol{A}\}$, is the sum of its diagonal elements [1, p. 6].

Properties of matrix trace (proved in HW)

- $\operatorname{trace}\{\cdot\}$ is a linear function: $\operatorname{trace}\{\alpha \boldsymbol{A} + \beta \boldsymbol{B}\} = \alpha \operatorname{trace}\{\boldsymbol{A}\} + \beta \operatorname{trace}\{\boldsymbol{B}\}, \ \forall \alpha, \beta \in \mathbb{F}$
- A **cyclic commutative property**:

$$\boldsymbol{A} \in \mathbb{F}^{M \times N}, \ \boldsymbol{B} \in \mathbb{F}^{N \times M} \Longrightarrow \qquad (1.32)$$

- Why cyclic? Because (let $\boldsymbol{A} = \boldsymbol{X}$ and $\boldsymbol{B} = \boldsymbol{Y}\boldsymbol{Z}$):

$$\operatorname{trace}\{\boldsymbol{X}\boldsymbol{Y}\boldsymbol{Z}\} =$$

- $\operatorname{trace}\{\boldsymbol{A}\}$ is the sum of the eigenvalues of $\boldsymbol{A}$ [1, Theorem 9.25].
(The proof involves the **Jordan normal form**, a linear algebra topic of less importance in SP/ML.)

Practical use in JULIA:
`using LinearAlgebra` or `using LinearAlgebra: tr`
then `tr(A)`

1.8 Summary

This chapter has reviewed basic properties of vectors and matrices. It is notable that **multiplication** operations are prevalent throughout. For example, **orthogonality** is a property involving multiplication, and even the key properties defining a **determinant** (and hence **eigenvalues**) involve multiplication. The only property that is defined in terms of a sum is the matrix **trace**, but after defining it we immediately considered matrix products (and sums). Matrix and vector products will continue to be prevalent hereafter.

1.9 Appendix: Fields, Vector Spaces, Linear Transformations

There are multiple different meanings of the term **vector** in the literature.

- In MATLAB, a **vector** is simply a column of a 2D matrix.
  In other words, a MATLAB (column) vector is a "$N \times 1$" array of numbers, *i.e.*, a 2D array where the second dimension is 1.
- In JULIA, a `Vector` is a 1D array of N numbers. This convention is more consistent with numerical linear algebra. More precisely, a variable of type `Vector{<:Number}` is a 1D array of numbers. The JULIA `Vector` type is general enough to hold any elements.
- In general mathematics, *e.g.*, linear algebra and functional analysis, a vector belongs to a **vector space**.
  For a nice overview of how this distinction affected the design of the JULIA language, see this video.

Although this course will mostly use the numerical methods perspective, students who want a thorough understanding should also be familiar with the more general notion of a vector space.

This Appendix reviews vector spaces and linear operators defined on vector spaces. Although the definitions in this section are quite general and thus might appear somewhat abstract, the ideas are important even for the topics of a sophomore-level signals and systems course! For example, analog systems like passive RLC networks are linear systems that are represented mathematically by linear transformations from one (infinite dimensional) vector space to another.

The definition of a vector space uses the concept of a field of scalars, so we first review that.

Field of scalars L§2.1

A **field** or **field of scalars** $\mathbb{F}$ is a collection of elements $\alpha, \beta, \gamma, \ldots$ along with an "addition" and a "multiplication" operator [17]. For every pair of scalars α, β in $\mathbb{F}$, there must correspond a scalar $\alpha + \beta$ in $\mathbb{F}$, called the **sum** of α and β, such that

- Addition is commutative: $\alpha + \beta = \beta + \alpha$
- Addition is associative: $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$
- There exists a unique element $0 \in \mathbb{F}$, called **zero**, for which $\alpha + 0 = \alpha$, $\forall \alpha \in \mathbb{F}$
- For every $\alpha \in \mathbb{F}$, there corresponds a unique scalar $(-\alpha) \in \mathbb{F}$ for which $\alpha + (-\alpha) = 0$.

For every pair of scalars α, β in $\mathbb{F}$, there must correspond a scalar $\alpha\beta$ in $\mathbb{F}$, called the **product** of α and β, such that

- Multiplication is commutative: $\alpha\beta = \beta\alpha$
- Multiplication is associative: $\alpha(\beta\gamma) = (\alpha\beta)\gamma$
- Multiplication distributes over addition: $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$
- There exists a unique element $1 \in \mathbb{F}$, called **one**, or **unity**, or the **identity** element, for which $1\alpha = \alpha$, $\forall \alpha \in \mathbb{F}$
- For every nonzero $\alpha \in \mathbb{F}$, there corresponds a unique scalar $\alpha^{-1} \in \mathbb{F}$, called the **inverse** of α for which $\alpha\alpha^{-1} = 1$.

Example. The set $\mathbb{Q}$ of rational numbers is a field (with the usual definitions of addition and multiplication).

The only fields that we will need are the field of real numbers $\mathbb{R}$ and the field of complex numbers $\mathbb{C}$.

Vector spaces

A **vector space** or **linear space** consists of

- A field $\mathbb{F}$ of scalars.
- A set $\mathcal{V}$ of entities called **vectors**
- An operation called **vector addition** that associates a **sum** $\boldsymbol{x}+\boldsymbol{y} \in \mathcal{V}$ with each pair of vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{V}$ such that
  - Addition is commutative: $\boldsymbol{x}+\boldsymbol{y}=\boldsymbol{y}+\boldsymbol{x}$
  - Addition is associative: $\boldsymbol{x}+(\boldsymbol{y}+\boldsymbol{z})=(\boldsymbol{x}+\boldsymbol{y})+\boldsymbol{z}$
  - There exists a unique element $\mathbf{0} \in \mathcal{V}$, called the **zero vector**, for which $\boldsymbol{x}+\mathbf{0}=\boldsymbol{x}, \forall \boldsymbol{x} \in \mathcal{V}$
  - For every $\boldsymbol{x} \in \mathcal{V}$, there corresponds a unique vector $(-\boldsymbol{x}) \in \mathcal{V}$ for which $\boldsymbol{x}+(-\boldsymbol{x})=\mathbf{0}$.
- An operation called **multiplication by a scalar** that associates with each scalar $\alpha \in \mathbb{F}$ and vector $\boldsymbol{x} \in \mathcal{V}$ a vector $\alpha \boldsymbol{x} \in \mathcal{V}$, called the **product** of α and $\boldsymbol{x}$, such that:
  - Associative: $\alpha(\beta \boldsymbol{x})=(\alpha \beta) \boldsymbol{x}$
  - Distributive $\alpha(\boldsymbol{x}+\boldsymbol{y})=\alpha \boldsymbol{x}+\alpha \boldsymbol{y}$
  - Distributive $(\alpha+\beta) \boldsymbol{x}=\alpha \boldsymbol{x}+\beta \boldsymbol{x}$
  - If 1 is the identity element of $\mathbb{F}$, then $1 \boldsymbol{x}=\boldsymbol{x}$. $\forall \boldsymbol{x} \in \mathcal{V}$.
- No operations are presumed to be defined for multiplying two vectors or adding a vector and a scalar.

Examples of important vector spaces

- **Euclidean n-dimensional space** or **n-tuple space**: $\mathcal{V} = \mathbb{R}^n$.
  If $\boldsymbol{x} \in \mathcal{V}$, then $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \in \mathbb{R}$.
  The field of scalars is $\mathbb{F} = \mathbb{R}$. Of course $\boldsymbol{x}+\boldsymbol{y} = (x_1+y_1, x_2+y_2, \ldots, x_n+y_n)$, and $\alpha\boldsymbol{x} = (\alpha x_1, \ldots, \alpha x_n)$.
  This space is very closely related to the definition of a vector as a column of a matrix. They are so close that most of the literature does not distinguish them (nor does MATLAB). However, to be rigorous, $\mathbb{R}^n$ is not the same as a column of a matrix because strictly speaking there is no inherent definition of the **transpose** of a vector in $\mathbb{R}^n$, whereas transpose is well defined for any matrix including a $n \times 1$ matrix.

- **Complex Euclidean n-dimensional space**: $\mathcal{V} = \mathbb{C}^n$. If $\boldsymbol{x} \in \mathcal{V}$, then $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \in \mathbb{C}$.
  The field of scalars is $\mathbb{F} = \mathbb{C}$ and $\boldsymbol{x} + \boldsymbol{y} = (x_1 + y_1, \ldots, x_n + y_n)$ and $\alpha\boldsymbol{x} = (\alpha x_1, \ldots, \alpha x_n)$.

- $\mathcal{V} = \mathcal{L}_2(\mathbb{R}^3)$. The set of functions $f : \mathbb{R}^3 \to \mathbb{C}$ that are **square integrable**:
  $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} |f(x,y,z)|^2 \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z < \infty$. The field is $\mathbb{F} = \mathbb{C}$.
  Addition and scalar multiplication are defined in the natural way.
  To show that $f, g \in \mathcal{L}_2(\mathbb{R}^3)$ implies $f + g \in \mathcal{L}_2(\mathbb{R}^3)$, one can apply the triangle inequality:
  $\|f + g\| \leq \|f\| + \|g\|$ where $\|f\| = \langle f,\, f\rangle$, and $\langle f,\, g\rangle = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y,z) g^*(x,y,z) \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z\,.$

- The set of functions on the plane $\mathbb{R}^2$ that are zero outside of the unit square.

- The set of solutions to a homogeneous linear system of equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}$.

Linear transformations and linear operators L§3.1

Define. Let $\mathcal{U}$ and $\mathcal{V}$ be two vector spaces over a common field $\mathbb{F}$.
A function $\mathcal{A} : \mathcal{U} \to \mathcal{V}$ is called a **linear transformation** or **linear mapping** from $\mathcal{U}$ into $\mathcal{V}$ iff $\forall \boldsymbol{u}_1, \boldsymbol{u}_2 \in \mathcal{U}$ and all scalars $\alpha, \beta \in \mathbb{F}$:

$$\mathcal{A}(\alpha \boldsymbol{u}_1 + \beta \boldsymbol{u}_2) = \alpha \mathcal{A}(\boldsymbol{u}_1) + \beta \mathcal{A}(\boldsymbol{u}_2).$$

Example:

- Let $\mathbb{F} = \mathbb{R}$ and let $\mathcal{V}$ be the space of continuous functions on $\mathbb{R}$. Define the linear transformation $\mathcal{A}$ by: if $F = \mathcal{A}(f)$ then $F(x) = \int_0^x f(t) \, \mathrm{d}t$. Thus integration (with suitable limits) is linear.

If $\mathcal{A}$ is a linear transformation from $\mathcal{V}$ into $\mathcal{V}$, then we say $\mathcal{A}$ is a **linear operator**. However, the terminology distinguishing linear transformations from linear operators is not universal, and the two terms are often used interchangeably.

Simple fact for linear transformations:

- $\mathcal{A}[\mathbf{0}] = \mathbf{0}$. Proof: $\mathcal{A}[\mathbf{0}] = \mathcal{A}[0\mathbf{0}] = 0\mathcal{A}[\mathbf{0}] = \mathbf{0}$. This is called the "zero in, zero out" property.

Caution! (From [18]) By induction it follows that $\mathcal{A}\left(\sum_{i=1}^n \alpha_i \boldsymbol{u}_i\right) = \sum_{i=1}^n \alpha_i \mathcal{A}(\boldsymbol{u}_i)$ for any finite n, but the above *does not* imply in general that linearity holds for infinite summations or integrals. Further assumptions about "smoothness" or "regularity" or "continuity" of $\mathcal{A}$ are needed for that.

Bibliography

[1] A. J. Laub. *Matrix analysis for scientists and engineers*. Soc. Indust. Appl. Math., 2005 (cit. on pp. 1.2, 1.20, 1.22, 1.40, 1.60, 1.63, 1.73, 1.76).

[2] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. In: *Proc. IEEE* 86.11 (Nov. 1998), 2278–2324 (cit. on p. 1.6).

[3] F. Rosenblatt. *Principles of neurodynamics; perceptrons and the theory of brain mechanisms*. Washington: Spartan, 1962 (cit. on p. 1.6).

[4] L. Eldén. *Matrix methods in data mining and pattern recognition*. Errata: http://users.mai.liu.se/larel04/matrix-methods/index.html http://users.mai.liu.se/larel04/matrix-methods/. Soc. Indust. Appl. Math., 2007 (cit. on p. 1.9).

[5] K. Bryan and T. Leise. “The $25,000,000,000 eigenvector: The linear algebra behind Google”. In: *SIAM Review* 48.3 (Sept. 2006), 569–81 (cit. on p. 1.9).

[6] A. N. Elmachtoub and C. F. Van Loan. “From random polygon to ellipse: An eigenanalysis”. In: *SIAM Review* 52.1 (2010), 151–70 (cit. on p. 1.11).

[7] J. A. Fessler. *ASPIRE 3.0 user’s guide: A sparse iterative reconstruction library*. Tech. rep. 293. Available from `http://web.eecs.umich.edu/~fessler`. Univ. of Michigan, Ann Arbor, MI, 48109-2122: Comm. and Sign. Proc. Lab., Dept. of EECS, July 1995 (cit. on p. 1.16).

[8] C. Champlin, D. Bell, and C. Schocken. “AI medicine comes to Africa’s rural clinics”. In: *IEEE Spectrum* 54.5 (May 2017), 42–8 (cit. on p. 1.22).

[9] R. P. Lippmann. “An introduction to computing with neural nets”. In: *IEEE ASSP Mag.* 4.2 (Apr. 1987), 4–22 (cit. on p. 1.22).

[10] J. Alman and V. V. Williams. “A refined laser method and faster matrix multiplication”. In: *Proc. ACM-SIAM Symp. on Discrete Alg. (SODA)*. 2021, 522–39 (cit. on p. 1.34).

[11] T. Davis. *Block matrix methods: Taking advantage of high-performance computers*. Univ. Florida TR-98-204. 1998 (cit. on p. 1.36).

[12] W. E. Roth. “On direct product matrices”. In: *Bull. Amer. Math. Soc.* 40.6 (1934), 461–8 (cit. on p. 1.42).

[13] A. Airola and T. Pahikkala. “Fast Kronecker product kernel methods via generalized vec trick”. In: *IEEE Trans. Neural Net. Learn. Sys.* 29.8 (Aug. 2018), 3374–87 (cit. on p. 1.42).

[14] J. Kovacevic and A. Chebira. “Life beyond bases: the advent of frames (Part I)”. In: *IEEE Sig. Proc. Mag.* 24.4 (July 2007), 86–104 (cit. on p. 1.58).

[15] J. Kovacevic and A. Chebira. “Life beyond bases: the advent of frames (Part II)”. In: *IEEE Sig. Proc. Mag.* 24.5 (Sept. 2007), 115–25 (cit. on p. 1.58).

[16] A. G. Akritas, E. K. Akritas, and G. I. Malaschonok. “Various proofs of Sylvester’s (determinant) identity”. In: *Mathematics and Computers in Simulation* 42.4 (Nov. 1996), 585–93 (cit. on p. 1.65).

[17] B. Noble and J. W. Daniel. *Applied linear algebra*. 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1977 (cit. on p. 1.79).

[18] T. Kailath. *Linear systems*. New Jersey: Prentice-Hall, 1980 (cit. on p. 1.82).