CMSC 27200 - Guerilla Section 3

Matrix rotations, complex numbers, FFT.

Topics

proofs and things if there's time.

1. Crash course on linear algebra.
   - Vectors and matrices.
   - matrix actions. ~~(rotation matrices)~~. rotation matrices.
     ~~(change of bases)~~.
2. Complex numbers.
   - Roots of polynomials $z^n = 1$. (deg d poly $\Leftrightarrow$ d roots).
     ↳ Completion of a number system
   - multiplying in polar form.

     $$e^{i\theta} = \cos\theta + i \cdot \sin\theta.$$
   - Roots of unity via polar form.
3. FFT. ← decoding ~~video~~ audio. → interpolating video.
   - polynomial evaluation → why it's useful.
   - what is FFT. → how to use it.

(46)

netflix recs. ← - Google pagerank.
- search results in ~ ms is due to hyperloglog.
- ~~Facetime and decoding~~
- OpenAI. | training

Crash course: Matrices.

Parts of this course, it will help to know a bit about linear algebra. This isn't for mathematical convenience but more so because

→ Tools from linear algebra belie the modern design of algorithms from systems / networking to machine learning and data science

⟹ Vectors. ← Vectors and matrices are fundamental objects of linear algebra.

A vector is an object w/ "magnitude" and "direction". We usually write it as a column of scalars.

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n$$

Vector Space $\mathbb{R}^n$ i.e. set of vectors. "closed under scalar mult and addition"

$$x, y \in \mathbb{R}^n \Leftrightarrow \forall a \in \mathbb{R},\ b \in \mathbb{R} \quad ax + by \in \mathbb{R}^n.$$

$\Longrightarrow$ matrices.

A <u>matrix</u> is a <u>2D array</u> of <u>scalars</u>. So for example a <u>real</u> matrix is a 2D array of <u>real #s</u>.

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

m-rows

n-columns

the set of all <u>real</u> <u>matrices</u>.

n-columns.

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{pmatrix}$$

m rows

The first time you encounter a matrix is probably in the setting where it "<u>stores data</u>".

$$A = \begin{pmatrix} \text{———} & A_1 & \text{———} \\ & \vdots & \\ \text{———} & A_m & \text{———} \end{pmatrix} \qquad A_j \in \mathbb{R}^n$$

m datapoints and you store it as a 2D array A[i][j]

(48)

And so these kinds of operations make sense as a way of "manipulating data".

- Scaling a matrix: $\forall c \in \mathbb{R}$, $cA \in \mathbb{R}^{m \times n}$.
- Adding a matrix: $\forall A, B \in \mathbb{R}^{m \times n}$. $A + B \in \mathbb{R}^{m \times n}$.

Add two data matrices together.

Scale every ~~row vector~~ data vector by c.

But now here's the main point of this crash course

→ Matrices also capture transformations b/w vectors.

And that's why these operations make sense

- Matrix vector multiplication: $\forall A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$.

$$y = Ax \in \mathbb{R}^m \quad \text{s.t.} \quad y_i = \sum_{j=1}^{n} A_{ij} y_j$$

Pictorially

$$i\ \underset{y}{\begin{pmatrix} y_i \end{pmatrix}} = \underset{A}{\begin{pmatrix} \boxed{A_{i1} \ \cdots \ A_{in}} \end{pmatrix}} \underset{x}{\begin{pmatrix} \boxed{\begin{matrix} x_1 \\ \vdots \\ x_n \end{matrix}} \end{pmatrix}} \longrightarrow \sum_{j=1}^{n} A_{ij} \cdot x_j .$$

Matrix-vector products $\underbrace{\text{map vectors}}_{\text{of dim \# cols}} \longrightarrow \underbrace{\text{vectors.}}_{\text{of dim \# rows.}}$

→ Think of this operation as a "transformation of a vector"

- Matrix-matrix product. $\forall A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$.

$A \cdot B \in \mathbb{R}^{m \times p}$ s.t.

$$\forall \begin{matrix} i = 1 \ldots m \\ j = 1 \ldots p \end{matrix} \qquad (AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} .$$

pictorially ... of $C = AB$.

\# cols for A = \# rows for B !

$$\underset{C}{\begin{pmatrix} & & \\ & c_{ij} & \\ & & \end{pmatrix}} = i \underset{\substack{m \text{ rows} \\ A}}{\begin{pmatrix} \boxed{A_{i1} \ \cdots \ A_{in}} \end{pmatrix}} \underset{B}{\begin{pmatrix} \overset{j}{\boxed{\begin{matrix} B_{1j} \\ \vdots \\ B_{nj} \end{matrix}}} \end{pmatrix}} \longrightarrow \sum_{k=1}^{n} A_{ik} B_{kj} .$$

p cols

Matrix - matrix products "compose vector transformations."

↳ follows by associativity. Take $x \in \mathbb{R}^p$.

$$(AB)x = \Big(A\big(\underbrace{Bx}_{\substack{\text{transform } x \mapsto y = Bx \\ \in \mathbb{R}^m}}\big)\Big) = Ay = z.$$

→ transform $y \mapsto z = Ay \in \mathbb{R}^m$.

→ questions?

⟹ How do you interpret what a vector transform does?

↳ ⊛ The "qualitative" description of how a matrix transforms a vector is called its "action"

This is a procedure.

1) Take a matrix $A \in \mathbb{R}^{m \times n}$,

2) Let $e_i = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$ i be the i^{th} indicator in $\mathbb{R}^n$.

3) Write down an <u>analytic expression</u> for Ae_i.

4) Repeat for all i.

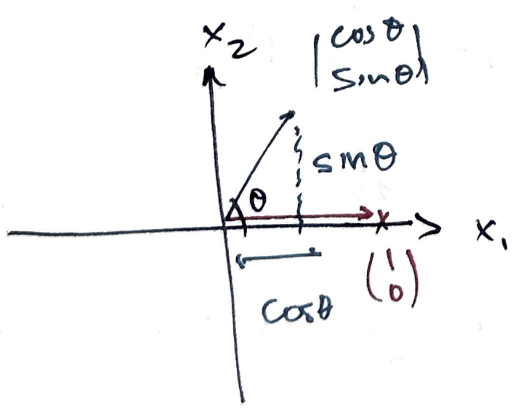
Example:

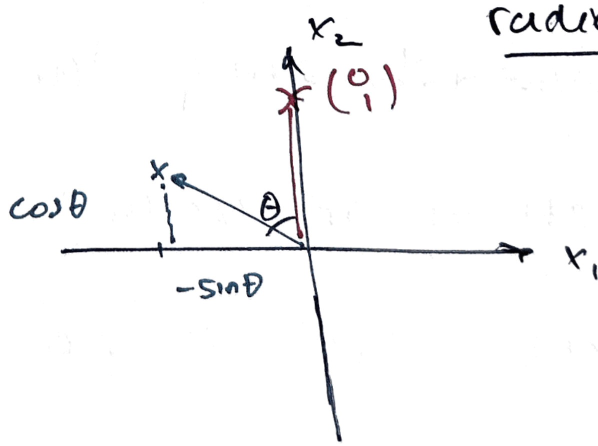
1) What does $A = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$ for given $\theta \in \mathbb{R}$ do.?

$$Ae_1 = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$$

$$Ae_2 = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix}$$

Now plot it. $\longrightarrow$ A rotates a vector by θ <u>radians CCW</u>





→ wait but isn't Ae_i $\forall i \in [n]$ going to just read out the i^{th} column? (skippable).

↳ yes and in fact that's exactly the point!

- ~~It is so Suppose you only had an imperfect description of what A does on each i.~~

- ~~Then you immediately get a matrix representation.~~

$$A = \begin{pmatrix} | & | & & | \\ Ae_1 & Ae_2 & \cdots & Ae_n \\ | & | & & | \end{pmatrix}$$

- ~~Understanding what A does on each e_i.~~ is sufficient

- ~~Understanding the action of A, it is "necessary"~~

- Getting an understanding of Ae_i for each $i \in [n]$ is "necessary" in the sense that as soon as I have A in matrix form then I can read out its action by looking at its columns...

$$A = \begin{pmatrix} | & | & & | \\ Ae_1 & Ae_2 & \cdots & Ae_n \\ | & | & & | \end{pmatrix}$$

- But it's also sufficient! If I have explicit description of Ae_i $\forall i \in [n]$ then I can immediately understand Ax for any $x \in \mathbb{R}^n$

↳ why? Because $\forall x \in \mathbb{R}^n$, I can write

$$x = \sum_{i=1}^{n} x_i \cdot e_i .$$

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = x_1 \cdot \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + x_2 \cdot \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} + \dots + x_n \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$$

then by linearity...

$$Ax = A\left(\sum_{i=1}^{n} x_i \cdot e_i\right) = \sum_{i=1}^{n} A(\overset{\text{scalar}}{x_i} \cdot \overset{\text{vector.}}{e_i})$$

$$= \sum_{i=1}^{n} x_i \cdot (Ae_i).$$

this we have an explicit description for.

→ So to summarize...

1. Matrices can represent data, but more so, they also represent vector transformation

2. You can understand* a vector transformation given matrix A by reading out Ae_i.

You might ask... given any vector mapping $f: \mathbb{R}^n \to \mathbb{R}^m$, do you get a unique matrix A s.t.

$$f(x) = Ax \quad ?$$

↪ Only if f is linear!

Fast-Fourier Transform.

When discussing an algorithm, it's important to clearly state what problem it's solving

Polynomial evaluation* ⟷ FFT

Polynomial interpolation** ⟷ Inverse FFT.

both together

↕

polynomial multiplication

* at conveniently chosen pts.

** from conveniently chosen pts.

This is the diagram to remember.

⟹ Polynomials.

In grade school / university one might learn that a degree-d polynomial is given by

$$p(x) = a_d \cdot x^d + a_{d-1} x^{d-1} + \dots + a_1 x + a_0.$$

where $a_0, a_2 \dots a_d \in \mathbb{R}$ are given. For example a quadratic fn is a deg = 2 polynomial

$$p(x) = x^2 - 2x + 1.$$

but now, we also learn how factor polynomials

$$p(x) = x^2 - 1 = (x-1)(x+1)$$

to determine its roots ~~$p(x) = 0 \Leftrightarrow x$~~ x s.t.

$p(x) = 0$ i.e.

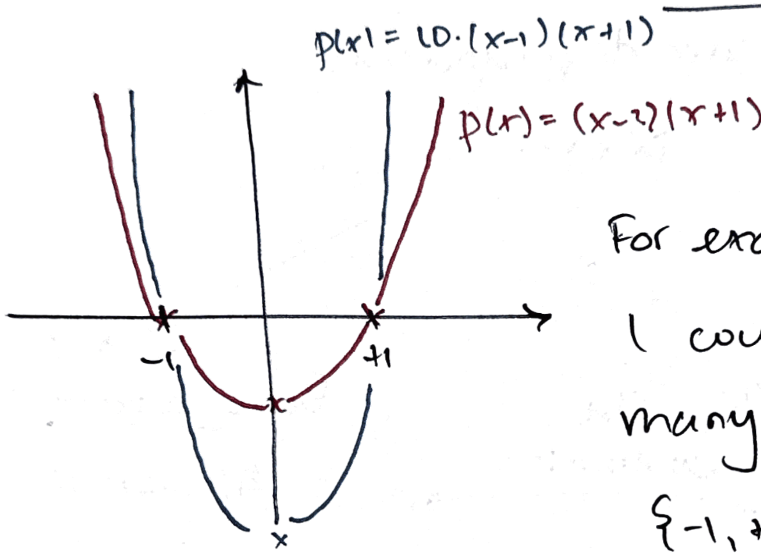
$$f(x) = 0 \Leftrightarrow x \in \{+1, -1\}.$$

and in the process of ~~learning~~ finding roots we encounter the following ~~fact~~. inverse problem

~~Fact: Every degree d polynomial has d potentially complex~~

→ If I have the roots of a polynomial, can I find a polynomial which fits those roots?

→ And the answer is not uniquely.



For example in deg = 2 I could have infinitely many $p(x)$ who have $\{-1, +1\}$ as roots.

↳ but as soon as I have a third point I have a unique polynomial

→ Finally it doesn't matter if I'm ~~[illegible]~~ fitting to roots or other points, I always have this fact

Fact: A degree-d polynomial is characterized uniquely by its evaluation at any distinct $d+1$ points.

(58)

This means I have two ways of representing polynomials.

degree d polynomial

$p(x)$

By $d+1$ coefficients. $a_0 \ldots a_d$

$$p(x) = \sum_{k=0}^{d} a_k \cdot x^k$$

$\xrightarrow{\text{evaluation}}$

$\xleftarrow[\text{interpolation.}]{}$

By evaluations at $d+1$ distinct points $x_0 \ldots x_d$

its value

$p(x_0) \ \ldots \ p(x_d)$

This gives us two computational tasks.

$\Longrightarrow$ polynomial evaluation.

Input: $d+1$ coeffs. $a_0 \ldots a_d$, ~~$x \in \mathbb{R}$~~

Output: The value $p(x) = \sum_{k=0}^{d} a_k \cdot x^k$

evaluated at $d+1$ "conveniently chosen points"

→ polynomial interpolation.

<u>INPUT</u>: d+1 distinct ~~evaluations~~ $\cancel{p(x_0) \dots p(x_d)}$

~~points~~ $\cancel{x_0 \dots x_d \in \mathbb{R}}$ ~~and~~ evaluations $p(x_0) \dots p(x_d)$

at conveniently chosen $x_0 \dots x_d$.

<u>OUTPUT</u>: The coeffs. $a_0 \dots a_d \in \mathbb{R}$ s.t.

$$p(x_i) = \sum_{k=0}^{d} a_k \cdot x_i^k \qquad \forall i = 0 \dots d.$$

→ questions?

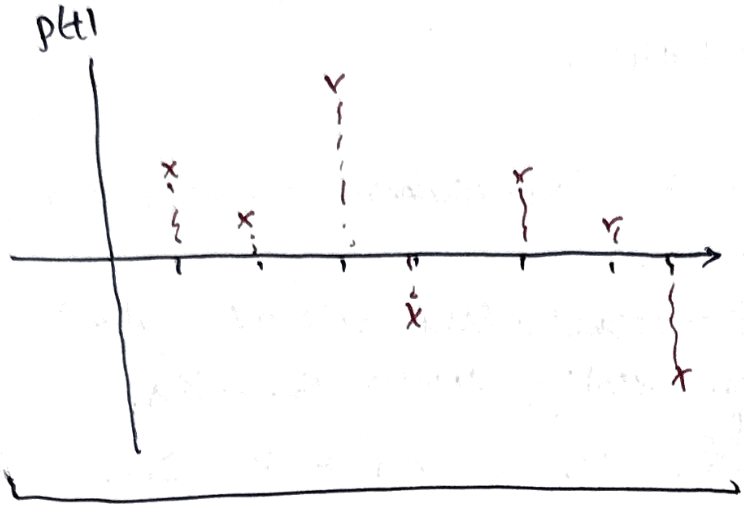
⟹ Okay why do we care? (skippable).

Why would it matter that we solve these two <u>problems</u>?

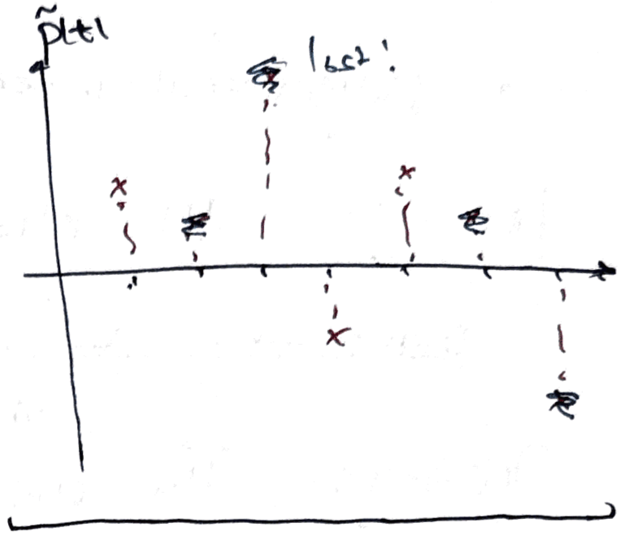
↳ It's literally how ~~cell phones work~~. any form of modern <u>long-range</u> <u>communication</u> works. -

- Let's just think of a cell-phone.

- When you talk, it can be modeled as an <u>intensity</u> per <u>discrete time step</u>.



your voice.



your voice through air.

If you send this intensity per time data through air, some of the data could be lost!

→ Suppose I have T discrete samples and I want to be robust against any k erasures

↳ how can I encode p(t) ~~s.t. I can~~ to $\hat{p}(t)$ ~~recover p(t)~~ s.t. no matter what k positions get erased, I can recover p(t) exactly?

IDEA: use polynomials.

1. Pick any deg-~~k~~ = T polynomial~~s~~. $\hat{p}$ s.t.
$$\hat{p}(1) = p(1) \quad \ldots \quad \hat{p}(T) = p(T).$$

2. pad your signal w/ k+1 additional evaluations

FFT.

$$\hat{p}(T+1) \quad \dots \quad \hat{p}(T+k+1)$$

3. Send the padded signal.

$\hat{p}(1)$	$\hat{p}(2)$	$\dots$	$\hat{p}(T+k+1)$

Now its possible that an arbitrary k points get ~~to~~ erased but that means you still have T+1 remaining points.

↳ ~~you can fit a unique~~ the polynomial that fits those points has to be unique

↳ It has to be the original polynomial that you used to encode the signal.

Inverse FFT

→ To decode, interpolate the polynomial from what remains of your signal

↳ evaluate at $t = 1 \ldots T$. to get back original data.

↳ ~~This is the basic principle behind error correction~~ → ~~This encoded~~

~~Let me now try to impress how deep this idea~~.

This encoding | decoding scheme is called a Reed - Solomon code.

~~Now let me try to impress upon you how deep this idea is~~.

The ~~fundamental principle is this~~.

~~you have data that you want to represent "robustly". So you encode it as a "simple~~

↳ This idea of <u>encoding</u> data <u>robustly</u> using a "simple" <u>mathematical object</u> that has.

$$\text{1) A "pointwise" representation} \underset{\text{inverse "FFT"}}{\overset{\text{"FFT"}}{\rightleftarrows}} \text{2) A "generative" representation}$$

w/ a definable "FFT" and inverse "FFT" is a <u>very deep idea</u>.

↳ It's not just that we found a <u>fast</u> "FFT" for polynomials. ⟶ It's really that we <u>needed</u> a fast FFT so that <u>polynomial</u> ~~ops~~ evaluation | interpolation | multiplication could be scaled to handle the <u>massive</u> quantities of data required in modern day application.

- from ECC on your RAM ⟶ training image models at OpenAI.

⟶ questions?

$\Longrightarrow$ Primer on complex numbers.

Before talking about FFT we'll need to talk about complex numbers.

$\longrightarrow$ What are complex numbers?

You might recall that when I write a degree d polynomial, I could have anywhere from 0 to d real roots.

$p(x) = x^2 + x + 1$ no roots

$p(x) = x^2 + 2x + 1 = (x+1)^2$ 1 root

$p(x) = x^2 - 1 = (x-1)(x+1)$ 2 roots.

↳ The complex numbers is the smallest set of numbers (field) s.t. every real polynomial of degree d has d unique roots.

$\mathbb{C}$, "algebraic completion".

The set is given by.

$$\mathbb{C} = \{ z = a + bi : a, b \in \mathbb{R} \}$$

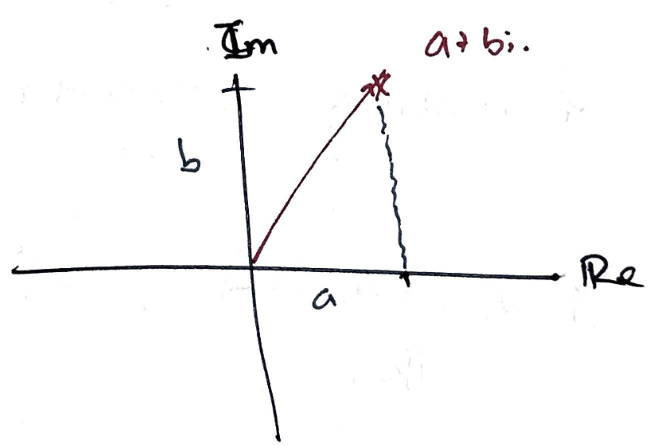
where i is called the "imaginary unit" and is defined to satisfy

$$i^2 = -1$$

What do you need to know?

1) Every complex number can be plotted on the complex plane like so.

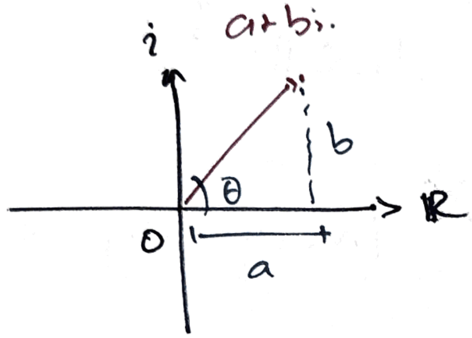
$z = a + bi$.



its "magnitude" is $|z| := \sqrt{a^2 + b^2}$. i.e. Euclidean length.

2) It can also be equivalently expressed using "polar" coordinates.

Given $z = a + bi$ ...



↳ let $r = |z| = \sqrt{a^2+b^2}$

$\theta = \tan^{-1}\left(\frac{b}{a}\right)$

then ...

$$z = r \cdot (\cos\theta + i \cdot \sin\theta).$$

~~alternat~~ conversely if you're given (r, θ) then you can get $z = a+bi$ by setting

$a = r \cdot \cos\theta$ $\qquad$ $b = r \cdot \sin\theta$.

3) As it would turn out, there's a very convenient way of writing $\cos\theta + i \cdot \sin\theta$ using e.

FACT: $\forall\, \theta \in \cancel{[0, 2\pi]}\ \mathbb{R}$, $e^{i\theta} = \cos\theta + i \cdot \sin\theta$.

Pf: Taylor series.

Consequently complex #s in polar form can be written as...

$$z = r \cdot (\cos\theta + i \cdot \sin\theta) = r \cdot e^{i\theta}.$$

4) Why use polar coordinates? because multiplication is convenient. Suppose I have z_1, z_2 s.t.

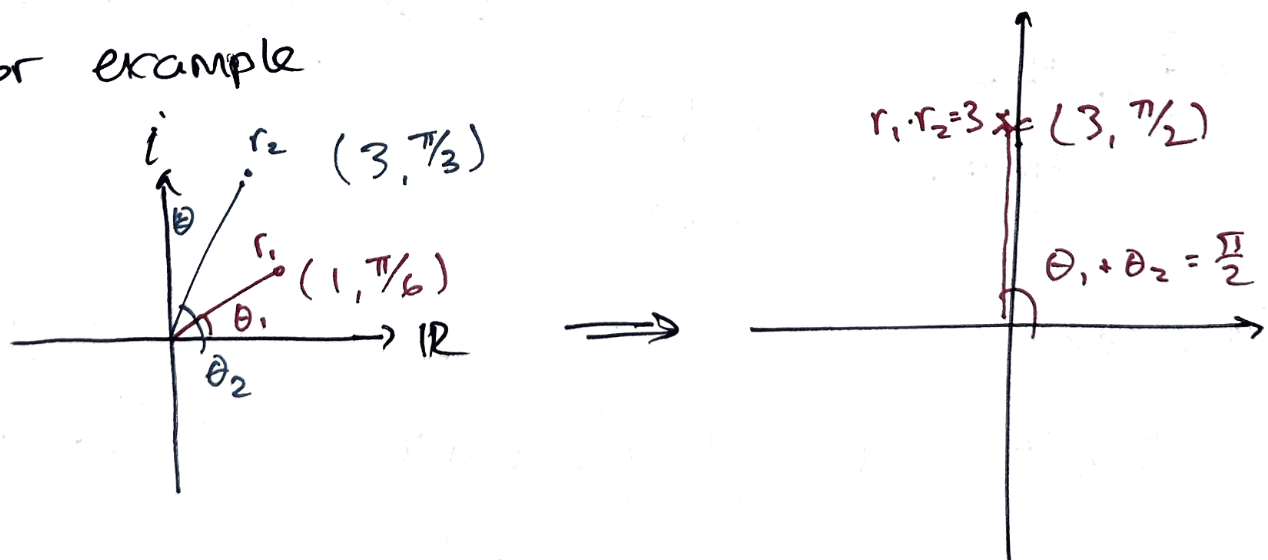
$$z_1 = r_1 \cdot e^{i\theta_1} \qquad z_2 = r_2 \cdot e^{i\theta_2}$$

Then

$$z_1 \cdot z_2 = r_1 \cdot e^{i\theta_1} \cdot r_2 \cdot e^{i\theta_2} = \underline{(r_1 r_2) \cdot e^{i \cdot (\theta_1 + \theta_2)}}$$

1) multiply the magnitude
2) add the angles.

For example



$\longrightarrow$ I don't have to do distribution.

5) And then finally let's just check that we actually get d roots for a simple degree $= d$ polynomial.

Let $p(z) = z^n - 1$. Do we get <u>n roots</u>?

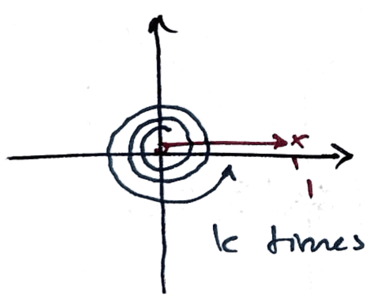
↳ yes! Let $\underset{\omega}{\cancel{z}}^{(k)} := \exp\left(\frac{2k \cdot \pi}{n} \cdot i\right)$ for $k = 0 \dots n-1$. Then indeed $\forall k = 0 \dots n-1$

$$\left(\underset{\omega}{\cancel{z}}^{(k)}\right)^n - 1 = \exp\left(i \cdot \frac{2k\pi}{n}\right)^n - 1$$

$$= \exp(i \cdot 2k\pi) - 1$$

$$= \underbrace{\exp(i \cdot k \cdot 2\pi)}_{=1 \text{ because } \theta = k \cdot 2\pi.} - 1$$

plot it !



$$= 1 - 1$$

$$= 0.$$

The set

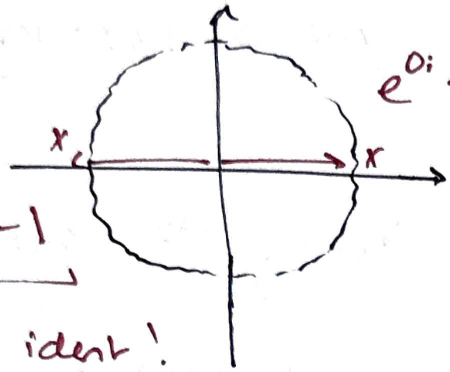
$$\{\omega^{(k)} \in \mathbb{C} : k = 0 \dots n-1\}$$

is called the <u>n^{th} roots of unity</u>

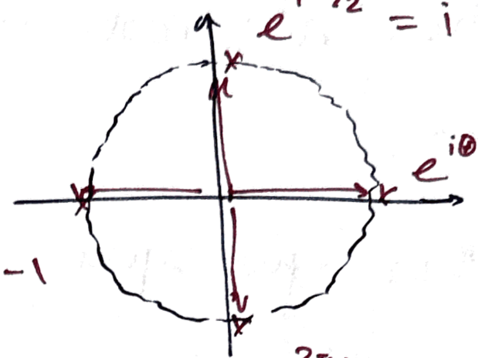
Since they denote $z = \sqrt[n]{1}$.

6) What are these n^{th} roots of unity?

$n = 2$

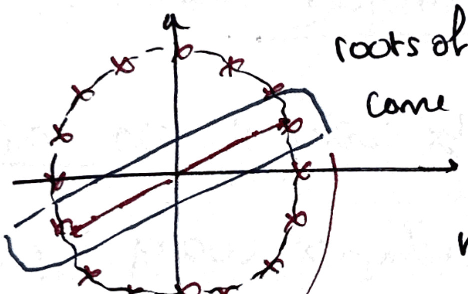


$n = 4$



And now here's the key observation for FFT.

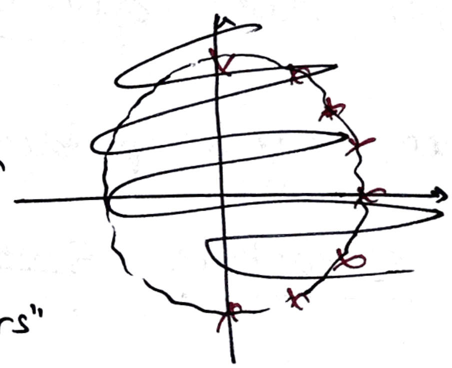
$n = \cancel{8}\ 16$



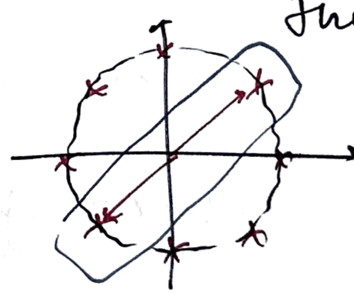
roots of unity come in pos-neg pairs when n is even

"conjugate pairs"

$n = \cancel{16}\ 8$.



→ when you square them they're still in +/- pairs



$$\omega^1 = \exp\left(i \cdot \frac{2\pi}{16}\right)$$

$$\omega^9 = \exp\left(i\ \frac{2\pi \cdot 9}{16}\right)$$

$$= \exp\left(i \cdot \frac{2\pi\,(8+1)}{16}\right)$$

$$= \exp\left(i \cdot \frac{2\pi \cdot 8}{16}\right) \cdot \exp\left(i \cdot \frac{2\pi}{16}\right)$$

$$= -1 \cdot \exp\left(i \cdot \frac{2\pi}{16}\right) = -\omega^1$$

Just to reiterate for n even...

1) The n^{th} roots of unity come in conjugate pairs.

2) When you square them... they ~~st~~ become the $n/2$ roots of unity and still come in conjugate pairs.

→ when does divide and conquer work?

- When your problem can be solved on input w/ structure ~~such~~ in such a way that.

  1) There's a natural way to ~~make~~ break your input up.

  2) When you break it up, the input enjoys the same structure in such a way that you can now focus on solving the problem on the smaller input.

→ you have n roots of unity that come in conjugate pairs.

↳ you square them and now you have $n/2$ roots of unity that still come in conjugate pairs.

→ Try to design a divide and conquer algo. which evaluates $p(x)$ on roots of unity.

→ questions?

⟹ The FFT algorithm.

So just to reiterate ...

Polynomial evaluation ⟺ FFT

Polynomial Interpolation ⟺ Inverse FFT.

Together...

↓

polynomial multiplication

So FFT takes the coefficient representation to pointwise representation.

⟹ Main Idea. (skippable)

Given a degree = $d-1$ polynomial $P(x)$, FFT will evaluate $p(x)$ at $\cancel{d}$ points ... $x_0 \dots x_{n}$.
$2^k = n \geq d$ distinct.

↳ The main observation is this. Notice that.

for any polynomial, we can write

$$p(x) = \underbrace{P_{even}(x^2)}_{\text{monomials of even degree}} + x \cdot \underbrace{P_{odd}(x^2)}_{\text{monomials of odd degree w/ } x \text{ factored out}}.$$

for example

$$x^4 + 3x^3 - 2x^2 + 10x - 1 \leftarrow p(x)$$

$$= (x^4 - 2x^2 - 1) + (3x^3 + 10x)$$

$$= \underbrace{(x^4 - 2x^2 - 1)}_{} + x \underbrace{(3x^2 + 10)}_{}.$$

$P_{even}(x) = x^2 - 2x - 1$

$P_{odd}(x) = 3x + 10.$

Now, this means we have a lot of shared work if we evaluate $p(\cdot)$ on x and $-x$ because.

$$p(x) = \underbrace{p_{even}(x^2)} + x \cdot \underbrace{p_{odd}(x^2)}$$

$$p(-x) = p_{even}((-x)^2) - x \cdot p_{odd}((-x)^2)$$

$$= \underbrace{p_{even}(x^2)} - x \cdot \underbrace{p_{odd}(x^2)}$$

shared work.

↳ If we want p evaluated on n pts. then it would be great if $x_1 \ldots x_n$ come in ± 1 pairs

⟶ That way we could use ~~recursion~~ divide-and-conquer to evaluate $p(\cdot)$ by recursively evaluating $p_{even}(\cdot)$ and $p_{odd}(\cdot)$.

↳ But why complex numbers?

Now here's the issue... suppose our $x_1 \ldots x_n$ actually did come in ± 1 pairs.

(74)

$$\begin{tikzcd}[column sep=small]
+x_1 \arrow[dr] & & -x_1 \arrow[dl] & +x_2 \arrow[dr] & & -x_2 \arrow[dl] & \cdots & +x_{n/2} \arrow[dr] & & -x_{n/2} \arrow[dl] \\
& x_1^2 & & & x_2^2 & & & & x_{n/2}^2 &
\end{tikzcd}$$

when we square $\pm x_i$ they both become x_i^2. Thus in the next level of recursion we need

$$x_1^2 \quad x_2^2 \quad \cdots \quad x_{n/2}^2$$

to also come in $+/-$ pairs.

↳ When can a square of a number be negative?

→ When it's imaginary!

The n points FFT evaluates $p(x)$ on are the n roots of unity.

$$\omega_k = \exp\left(\frac{2\pi \cdot k}{n} \cdot i\right).$$

→ Given d, let $\cancel{n = 2^k}$ n = smallest power of 2 larger than d to ensure x^2 is also a root of unity

$\Rightarrow$ Pseudocode

So let's write down the pseudocode...

<u>PROC</u>: FFT

<u>INPUT</u>: $p(x)$ in coeff form, $n = 2^k \geq d+1$, ω an n^{th} root of unity

<u>Do</u>: the following.

1. If $\omega = 1 \longrightarrow$ return $p(1)$
2. Let P_{even}, P_{odd} be such that.

$$p(x) = P_{even}(x^2) + x \cdot P_{odd}(x^2).$$

3. Call <u>FFT</u>(P_{even}, $n/2$, ω^2) to evaluate $P_{even}(\omega^{2j})$ for $j = 0 \ldots n/2 - 1$.
4. Call <u>FFT</u> (P_{odd}, $n/2$, ω^2) to evaluate $P_{odd}(\omega^{2j})$ for $j = 0 \ldots n/2 - 1$
5. For $j = 0 \ldots n/2 - 1$: do.
   - Compute $p(\omega^j) = P_{even}(\omega^{2j}) + \omega^j \cdot P_{odd}(\omega^{2j})$
   - Compute $p(-\omega^j) = P_{even}(\omega^{2j}) - \omega^j \cdot P_{odd}(\omega^{2j})$
6. <u>Output</u> $p(\omega^j)$ for $j = 0 \ldots n-1$.

76

Now... what happens when ~~do~~ you want to run FFT by hand...

$\Longrightarrow$ Running FFT by hand

Do you actually simulate the code? No. You compute a matrix-vector product.

1. Write down the Vandermonde matrix of ω's. Call it $M(\omega)$.
2. Write down the vector of coeffs. $\underline{v}$.
3. Compute $M(\omega) \cdot \underline{v}$.

Example: Suppose I want to run FFT on

$$p(x) = 3x^3 - 2x^2 + 4x - 1$$

First, let

$$M(\omega) = \begin{array}{c} \begin{matrix} x^0 & x & x^2 & x^3 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{pmatrix} \end{array}$$

Then set

$$V = \begin{array}{cl} \begin{pmatrix} -1 \\ 4 \\ -2 \\ 3 \end{pmatrix} & \begin{array}{l} a_0 \\ a_1 \\ a_2 \\ a_3 \end{array} \end{array}$$

finally compute...

$$M(\omega) \cdot V = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{pmatrix} \begin{pmatrix} -1 \\ 4 \\ -2 \\ 3 \end{pmatrix}$$

$$= \begin{pmatrix} 4 \\ -1 + 4i - 2i^2 + 3i^3 \\ -1 + 4i^2 - 2i^4 + 3i^6 \\ -1 + 4i^3 - 2i^6 + 3i^9 \end{pmatrix}$$

$$= \begin{pmatrix} 4 \\ 1 + i \\ -30 \\ 1 - i \end{pmatrix}$$

$\longrightarrow$ questions?

$\Rightarrow$ The inverse FFT.

We also said that the inverse FFT brings the ~~coefficient representation~~ ~~to~~ value representation to the coefficient representation. How so?

The matrix multiply that we did is literally going from coeff $\Rightarrow$ value

$$\begin{pmatrix} & M(\omega) & \end{pmatrix} \underbrace{\begin{pmatrix} \\ \\ \end{pmatrix}}_{\text{coeff}} = \underbrace{\begin{pmatrix} \\ \\ \end{pmatrix}}_{\text{value}}$$

That means... if we <u>invert</u> $M(\omega)$ then we can go from value $\Rightarrow$ coeff

$$\underbrace{\begin{pmatrix} \\ \\ \end{pmatrix}}_{\text{coeff}} = \begin{pmatrix} & M(\omega)^{-1} & \end{pmatrix} \underbrace{\begin{pmatrix} \\ \\ \end{pmatrix}}_{\text{value}}$$

$\longrightarrow$ Do I literally need to compute $M(\omega)^{-1}$?

$\longrightarrow$ No.!.

<u>FACT:</u> $M(\omega)^{-1} = \frac{1}{n} \cdot M(\omega^{-1})$.

Thus to run the inverse FFT. by hand, do the following.

1. Let y = evaluation vector

2. Compute $\frac{1}{n} \cdot M(\omega^{-1}) \cdot y$.

<u>Example</u> : Suppose our evaluation vector is.

$$y = \begin{pmatrix} 3 \\ -i \\ 1 \\ 2-i \end{pmatrix}$$

Let's compute the <u>inverse-FFT</u>. Note

$$M(\omega^{-1}) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} \\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9} \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}$$

So then with $n=4$...

$$\frac{1}{n} \cdot M(\omega^{-1}) \cdot y = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} 3 \\ -i \\ 1 \\ i \end{pmatrix}$$

$$= \frac{1}{4} \cdot \begin{pmatrix} 3-i+1+i \\ 3+i^2-1+i^2 \\ 3+i+1-i \\ 3-i^2-1-i^2 \end{pmatrix} = \frac{1}{4} \cdot \begin{pmatrix} 4 \\ 0 \\ 4 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

and that's the coeff vector for $p(x) = x^3 + x^2 + 1$.

⟹ What's the inverse FFT "algorithm"?

$$\frac{1}{n} \cdot FFT(\langle values \rangle, n, \omega^{-1}).$$

⟹ How do you do polynomial multiplication in $O(n \cdot \log n)$ time?

→ Suppose $p(x)$ is <u>degree d_1</u>, $q(x)$ is <u>degree d_2</u>.

1) Choose $n = 2^k \geq (d_1 + 1) + (d_2 + 1)$

2) Evaluate $p(x)$ at n points by calling

$$FFT(\langle coeff\ p \rangle, n, \omega)$$

to get $p(x_1) \dots p(x_n)$

3) Evaluate $q(x)$ at n points by calling

$$FFT(\langle coeff\ q \rangle, n, \omega)$$

to get $q(x_1) \dots q(x_n)$

4. For each $j = 1 \dots n$ compute

$$a_j = p(x_j) \cdot q(x_j)$$

5. Call inverse FFT to get the coeffs of $p \cdot q$.

$$\frac{1}{n} \cdot FFT(\langle a_1 \dots a_n \rangle, n, \omega^{-1}).$$

→ questions?