Virkeligt Hurtige Fourier Transformationer

Aksel Mannstaedt

Kode med tekst og grafer - `https://nbviewer.org/github/unic0rn9k/fourier-notebook/blob/master/README.ipynb`
github repo - `https://github.com/unic0nr9k/fourier-notebook`

Jeg vil stærkt opfordre til at prøve at downloade notebooken og prøve at eksperimentere med at kombinere forskellige andre bølger for at se at min FFT implementering virkeligt virker.

Indhold

Teori

Rekursion og funktionel programmering

Rekursion er helt simpelt, det der sker når en funktion der kalder sig selv. Dette bliver brugt meget i funktionel programmering.
Funktionel programmeringssprog er et paradigme kendetegnet ved et fokus på funktioner i stedet for objekter og kontrol strukturer. [1]

Fordele ved funktionelle programmeringssprog er at de er designet til at formindske side effects, og derved uforudventet adfærd. Side effects er defineret som en operation der muterer data udenfor den funktion der ejer data'en. Dette vil altså siges, at hvis man har en funktion der tager en pointer til en værdi og så ændrer på den værdi via pointeren, vil det være en side effekt af funktionen.
Dette kan skabe uforudventet adfærd i et program og vil derfor typisk gerne undgås.
Fuldkommen funktionelle programmeringssprog er sprog, hvor man ikke kan mutere data overhovedet. Det betyder altså, at hvis man for eksempel vil lave et for loop, er man nødt til at bruge rekursion.
Fuldkommen funktionelle programmeringssprog bliver ikke rigtigt anvendt i praksis, men de er dog stadig relevante at skrive om, da de er tættere på matematisk notering end traditionelle programmeringssprog.
I min case har jeg anvendt fuldkommen funktionel psudo kode til at lave en model for hvordan jeg ville strukturere min implementering af Cooley-Tukey FFT algoritme.

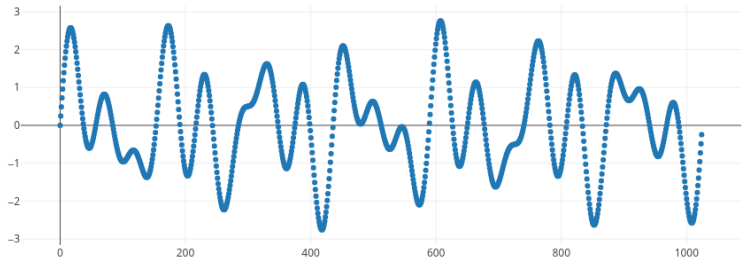
:)

[1] `https://en.wikipedia.org/wiki/Functional_programming`

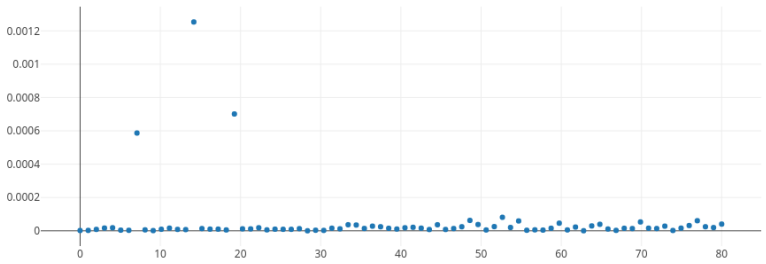
Hvad er en Fourier transformation?

En Fourier transformation er en vektor transformation (der i øvrigt også kan udvides til n-dimensionelle tensore) der blandt andet kan anvendes til at transformere et polynomium, repræsenteret som en koefficient-vektor, til en punkt-vektor repræsentation.
Dette er også ækvivalent til en time to frequency -domain transformation. Altså den kan også anvendes som en transformation af bølger, repræsenteret som intensitet over tid, til intensitet over frekvens.
Her vil jeg lige pointere at alle disse operationer er ækvivalente, altså det er bare forskellige måder at repræsentere, tænke på og visualizere den samme transformation. Først vil jeg primært fokusere på bølge repræsentationen af Fourier transformationen.

Her er som eksempel en bølge og dens tilhørende Fourier transformation. Bølgen er en kombination af en 7, 14 og 19 frekvens bølge (Frekvensen er enhedsløs der den er bølge længde over vektorens længde).



Her kan man altså se der ligger lokale maksimal punkter lige omkring 7, 14 og 19 på x-aksen.



DFT

DFT er kort for diskret Fourier transformation, og var den originale formel opdaget af Joseph Fourier i 1830. Han fandt ud af, hvordan man kunne lave de her transformationer, før man kunne udregne dem på computere, eller havde nogen praktisk anvendelse til dem[2].
Her er f frekvens, t er tid, g er en sammensat bølge funktion og $\hat{g}$ er den Fourier transformerede funktion af g.

$$\hat{g}(f) = \int_{+}^{-} g(t) \cdot e^{-2\pi i f t} dt$$

Formlen ovenfor viser hvordan en diskret Fourier transformation kan udregnes, den kan se lidt skræmmende ud, men er nem nok at forstå når man lige har styr på et par koncepter først.

Komplekse tal, Euler's formel og Euler's identitet

Et komplekst tal er et tal bestående af en imaginær del og en reel del (summen af dem). Den imaginære del af det komplekse tal er defineret som produktet af et virkeligt tal og i.
i er defineret som kvadratroden af -1. (altså det findes ikke).
Det giver måske ikke lige umiddelbart så meget mening, men en nem måde at tænke på det, er som en 2d vektor der har nogen specielle regne regler.
For eksempel vil $(0 + 1i)^2$ være -1, da i^2 skal give -1.
Komplekse tal bliver typisk også visualiseret som en 2d-vektor, hvor den reelle del er på x-aksen, og den imaginære del er på y-aksen.
Her vil det svares til en 90° rotation, i det komplekse plan, at gange med i.

$$(1 + 0i) \cdot i = 0 + 1i$$
$$(0 + 1i) \cdot i = -1 + 0i$$
$$(-1 + 0i) \cdot i = 0 - 1i$$

osv...

[2] https://en.wikipedia.org/wiki/Joseph_Fourier

Euler's formel er helt central i Fourier transformationen, og kan forstås gennem Euler's identitet, som er en populær ligning i matematikkens verden, der er defineret som

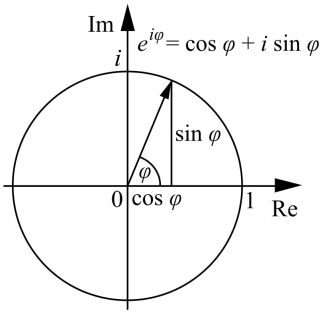
$$e^{\pi i} = -1$$

Dette udtryk kommer af at e^x er sit eget derivativ (e er Euler's konstant [3]), det vil altså siges at $\frac{d}{dx}e^{kx} = k \cdot e^{kx}$, hvor k er en konstant.
Altså har e^{kx} hældningen $k \cdot e^{kx}$. Det betyder at hvis vi bytter k ud med i, må e^{ix} bevæge sig mod en 90° rotation.

Det viser sig faktisk at Taylor serie udvidelsen af e^{ix} er den samme som den af $cos(x) + i \cdot sin(x)$, altså må de to funktioner være ens [4, 5].
Euler's formel, også kendt som Euler's notation, er derfor defineret som

$$e^{ix} = cos(x) + i \cdot sin(x)$$

Her vil en halv rotation derfor svare til $x = \pi$ og vi kan derved konkludere at $e^{\pi i} = -1$.



Figur 1: `https://en.wikipedia.org/wiki/Euler's_formula`

:)

[3] `https://en.wikipedia.org/wiki/Euler%27s_constant`
[4] `https://socratic.org/questions/how-do-you-use-a-taylor-series-to-prove-euler-s-formula`
[5] `https://en.wikipedia.org/wiki/Euler's_formula`

Uddybning af Fourier transformation

Fourier transformationen er nemmere at forstå hvis man deler den op i bider. Den inderste del af den diskrete Fourier transformering kan ses lidt som et prik produkt

$$f(t) = g(t) \cdot e^{-2\pi i f t}$$

Her vil det virkelige komponent af $f(t)$ være større når intensiteten af $g(t)$ matcher den der ville findes hvis frekvensen af g var f.
Dette kan intuitivt forstås, som at når t værdier ligger i bølgedale, vil $e^{-2\pi i f t}$ være negativ og derfor vil $f(t)$ være positiv hvis $g(t)$ også er negativ.
-2π sikre at en forøgelse af en tidsenhed svarende til en periode med frekvensen f også vil resultere i en fuld rotation af $e^{-2\pi i f t}$.

Integralet bliver taget af $g(t)e^{-2\pi i f t}$ for at sikre at frekvensen matcher over tid. Altså hvis man ikke brugte integralet, ville enkelte punkter, der matcher dem fra frekvensen f, resultere i en høj værdi på $\hat{g}(f)$ for frekvenser der egentligt ikke er til stede i $g(t)$.

DFT algoritmen har en algoritmisk kompleksitet på $O(n^2)$ der $\hat{g}$ er en funktion af både tid og frekvens.

FFT (Fast Fourier Transformation)

Koefficient to punkt repræsentation

Fourier transformeringen svares ikke kun til en tids til frekvens domæne transformering, men også til en koefficient til punkt repræsentation.

givet en bølge repræsenteret som en vektor af intensitet over tid

$$\vec{b} = [0, 1, 2, 3]$$

vil kunne repræsenteres som et polynomium

$$b(x) = x + 2 \cdot x^2 + 3 \cdot x^3$$

her vil det gælde at

$$\hat{b}(x) = b((-i)^x)$$

DFT eksempel med Julia:

```
julia> b(x) = 2*x^2 + 3*x^3 + x
b (generic function with 1 method)

julia> for n in 0:3
           println(b((-im)^n))
       end
 6 + 0im
-2 + 2im
-2 + 0im
-2 - 2im

julia> fft([0, 1, 2, 3])
4-element Vector{ComplexF64}:
  6.0 + 0.0im
 -2.0 + 2.0im
 -2.0 + 0.0im
 -2.0 - 2.0im
```

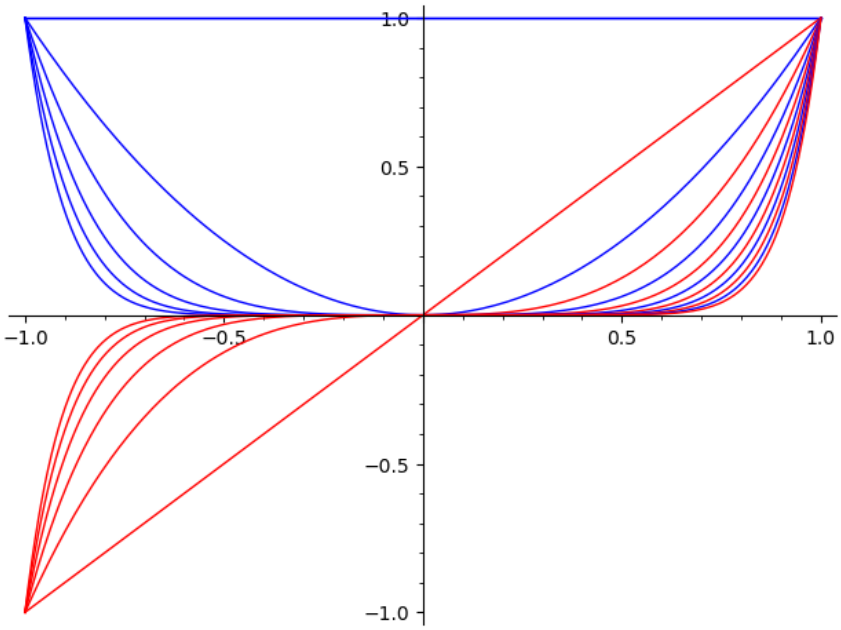
I eksemplet over kan man tydeligt se $O(n^2)$ kompleksiteten, der funktionen b, har n led der skal udregnes og b sig selv skal også computeres n gange.

Polynomier af en lige grad er spejlet om y-aksen. Polynomier af en ulige grad er spejlet om y-aksen og x-aksen.

```
p = plot(1)

for n in range(0, 6):
    p = p + plot(x^(n * 2)) + plot(x^(n*2 +1), color='red')

p
```



Figur 2: Lavet med sagemath

Vi kan udnytte den egenskab af polynomier ved at splitte vores polynomium op i ulige og lige led og derved evaluer polynomiet på færre punkter.

Eksempel:

$$f(x) = (3x^2 + 4x^3 + 2x^4 + x^5) = (3x^2 + 2x^4) + x(4x^2 + x^4)$$

$$f_{lige}(x) = 3x + 2x^2$$

$$f_{ulige}(x) = 4x + x^2$$

Bemærk at graden af alle ledn'e er divideret med 2 og en er trukket fra graden af det ulige polynomium. Dette gør at vektor repræsentationen af f_{lige} og f_{ulige} vil svares til alle de lige/ulige værdier fra $\vec{f}$, samlet i nye vektorer. Her vil dimensionen af $\vec{f_{lige}}$ og $\vec{f_{lige}}$ altså være $n/2$, hvor n er dimensionen af $\vec{f}$.

$$f(x) = f_{lige}(x^2) + x \cdot f_{ulige}(x^2)$$

For negative x-værdier, er det altså kun f_{ulige} der skal sættes i minus, der de lige kun er spejlet om y-aksen.

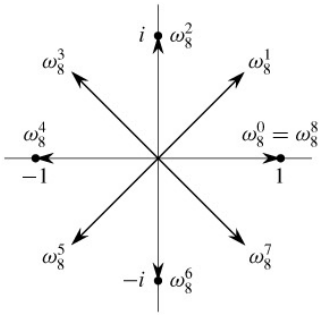
$$f(-x) = f_{lige}(x^2) - x \cdot f_{ulige}(x^2)$$

Her fra bliver f delt op i lige og ulige koefficienter rekursivt indtil den skal evalueres på et punkt.
For at kunne forsætte herfra uden at bryde rekursion, skal vi kunne generare n tal, der alle giver 1 når opløftet i n'de. Her skal n altid være 2^{noget}, for at polynomiet kan deles op i positive og negative par rekursivt ned til en koefficient.

n'th roots of unity

Her anvendes Euler's formel til at udregne n komplekse tal der er liggeligt fordelt på en enheds cirkel. De punkter bliver noteret som ω_n og kan udregnes med

$$\omega_n = e^{\frac{-2\pi i}{n}}$$



Figur 3: `http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/images/fig853_02.jpg`

Her vil ω^{j+1} svares til en n'de del rotation om enheds cirklen fra ω^j.

Alle ω punkter har tilsvarende negative punkter spejlet om 0,0

$$-\omega_n^j = \omega_n^{j+\frac{n}{2}}$$

$\vec{f}$ vil blive udregnet ved at evaluere f for vær værdi af j, hvor j antager værdierne fra 0 til $n/2$.

$$f(\omega_n^{j+\frac{n}{2}}) = f_{lige}(\omega_{n/2}^j) - \omega_n^j \cdot f_{ulige}(\omega_{n/2}^j)$$
$$f(\omega_n^j) = f_{lige}(\omega_{n/2}^j) + \omega_n^j \cdot f_{ulige}(\omega_{n/2}^j)$$

Her bliver $f(\omega_n^j)$ evalueret rekursivt indtil et 0 grads polynomium er tilbage, hvor FFT'en skal returnere koefficient af polynomiet. De negative evalueringer bliver placeret på højrehåndssiden af den resulterende vektor og de positive på højre side.

I min kode har jeg divideret n med 2 i de rekursive kald i stedet for at anvende ω^2, der det vil give det same resultat.

$$\begin{aligned} \omega_n^2 &= e^{\frac{-2\pi i \cdot 2}{n}} \\ &= e^{\frac{-2\pi i}{n/2}} \\ &= \omega_{\frac{n}{2}} \end{aligned} \tag{1}$$

Tooling (programmering)

Jeg valgte at skrive koden til denne case i rust, da jeg er komfortabel med sproget, og gerne ville eksperimentere med at lave en hurtig implementering af Cooley-Tukey algoritmen.
Rust er et rigtigt hurtigt sprog, dette skyldes blandt andet at det bruger llvm som backend, men også rust's brug af zero-cost-abstractions.
Jeg valgte at skrive koden i en jupyter notebook, da jeg ikke havde nogen egentlig applikation af min kode i tankerne under forløbet. Det viste sig også at være super praktisk til at lave TDD (test driven development), da det betød jeg kunne smide nogen grafer ind, og have dem opdateret i næsten realtime, mens jeg arbejdede på implementeringen af FFT algoritmen.

Rust sprog paradimer

Rust er et memory-safe programmeringssprog, hvilket betyder at det by-default ikke lader en skrive koder, der kan resultere i undefined-behavior[6]. Dette betyder at rust har en borrow-checker der ikke lader ens kode compile' hvis det bryder nogen regler defineret i rust sprog specifikationerne. Man kan for eksempel ikke bruge en reference i flere funktioner på en gang, og alle værdier skal makkeres med en mutability specificer, der bestemmer om man kan ændre på den. Derudover introducere rust også et koncept der hedder lifetimes, som kort sagt betyder at kompileren sikre at man ikke kan bruge references til værdier der er blevet deallokeret.

Disse regler er ikke absolutte. Man kan makkere kode som 'unsafe' for at slippe uden om reglerne introduceret af compileren.
I min kode har jeg for eksempel valgt at lave en meget unsafe implementering af FFT algoritmen, men har så lavet en safe wrapper til den, der sikre at man ikke kan introducere undefined behavior i sin kode ved brug af min algoritme.

Min implementering af FFT algoritmen

I min implementering af Cooley-Tukey algoritmen har jeg ikke valgt at tage en fuldkommen funktionel tilgang. Jeg har i stedet valgt at bruge pointere der bliver shiftet til højre for hvert rekursivt kaldt. Dette gør at jeg har kunne lave en safe wrapper funktion der tjekker om længden af input vektoren er 2^{noget} og derefter allokere alt hukommelse der skal anvendes på en gang (Dette er hurtigere en mange små).

```
fn fft<const LEN: usize>(x: StaticVecRef<Complex<f32>, LEN>)
    -> [Complex<f32>; LEN]
{
    // Tjek om dimensionaliteten af vektoren er valid.
    assert_eq!(LEN & (LEN - 1), 0);
    // Allokere en kopi af x (input vektoren).
    let mut ret = **x;
    // Kald til rekursiv funktion
    unsafe{ unsafe_fft::<LEN>(x.as_ptr(), ret.as_mut_ptr(), LEN, 1) };
    // Implicit return.
    ret
}
```

[6] https://doc.rust-lang.org/reference/behavior-considered-undefined.html

fft funktionen allokere altså et stykke hukommelse på samme størrelse som input vektoren, som det eneste.

Kan den blive endnu hurtigere?

I min fft bliver ω_n udregnet for hvert rekursivt kald hvor n er støre end 1. Dette er ret unødvendigt, der man altid kan udregne $\omega_{n/2}$ kan udregnes ud fra ω_n.
Her kan man altså udregne n ω-værdier for ω_n, hvor n er et virkeligt stort tal, på compiletime.
Dette bliver kald for twiddle factors.
Her ville man altså kunne for et signifikant performance increase, ved at indicere ind i et array med præ-udregnede twiddle factors, når Fourier transformation til en vektor med færre dimensioner end n skal udregnes.

Bibliografi

1: Undefined-behavior - `https://doc.rust-lang.org/reference/behavior-considered-undefine`
`html`
2: Funktionel programmering - `https://en.wikipedia.org/wiki/Functional_`
`programming`
3: Fourier - `https://en.wikipedia.org/wiki/Joseph_Fourier`
4: Euler's konstant - `https://en.wikipedia.org/wiki/Euler%27s_constant`
5: Euler's formel bevis med Taylor serier - `https://socratic.org/questions/`
`how-do-you-use-a-taylor-series-to-prove-euler-s-formula`
6: Euler's formel - `https://en.wikipedia.org/wiki/Euler's_formula`

Bilag

Kode som pdf vedhæftet på næste side... Grafer kan ikke vises i pdf'en, derfor anbefaler jeg at kigge på notebook'en linket til i toppen af dokumentet.

README

February 26, 2022

1 Configurations

Nightly toolchain is needed for `slas` to work, which is a linear algebra system written by me.

Plotly and itertools is also needed for plotting data.

[2]:
```
:toolchain nightly

:dep slas = { git = "https://github.com/unic0rn9k/slas", features =␣
 ↪["fast-floats"] }
:dep plotly
:dep itertools-num

#![allow(incomplete_features)]
#![feature(generic_const_exprs, test)]

use slas::prelude::*;
use slas_backend::*;

extern crate plotly;
extern crate rand_distr;
extern crate itertools_num;
extern crate itertools;

use itertools_num::linspace;
use plotly::common::{
    ColorScale, ColorScalePalette, DashType, Fill, Font, Line, LineShape,␣
 ↪Marker, Mode, Title,
};
use plotly::layout::{Axis, BarMode, Layout, Legend, TicksDirection};
use plotly::{Bar, NamedColor, Plot, Rgb, Rgba, Scatter};
use rand_distr::{Distribution, Normal, Uniform};
```

[2]:
```
Toolchain: nightly
```

2 Some example code

For some reasone slas vectors needs type annotations when used in evcxr (Jupyter), even though this is not required normally...

[3]:
```
let mut a: StaticCowVec<f32, 3> = moo![f32: 1, 2, 3];
let b: StaticCowVec<f32, 3> = moo![f32: 0..3];

println!("{}", b.static_backend::<Blas>().dot(&a.static_backend::<Blas>()));
```

```
8
```

[4]:
```
println!("{a:?}");
```

```
[1.0, 2.0, 3.0]
```

3 Function for plotting

first we define a function for plotting a vector with index on the x-axis and values on the y-axis

[5]:
```
extern crate serde;

fn plot_vector<const LEN: usize>(v: StaticVecRef<f32, LEN>){
    let t: Vec<f64> = linspace(0., LEN as f64, LEN).collect();
    let trace = Scatter::new(t, **v).mode(Mode::Markers);
    let mut plot = Plot::new();
    plot.add_trace(trace);
    let layout = Layout::new().height(v.iter().map(|n|*n as usize).max().
 ↪unwrap());
    plot.set_layout(layout);
    plot.notebook_display();
}
```

3.1 Generating and plotting an example wave

[68]:
```
use std::f32::consts::PI;

const LEN: usize = 2_usize.pow(10);

// Fast floats are more inacurrate, but faster to do operations on then regular␣
 ↪f32 and f64 floats.
// When you see something like fast(PI), it means a fast float is being used␣
 ↪instead of a regular one.
use fast_floats::*;

let example: StaticCowVec<f32, LEN> = moo![|t|{
    let t = fast(t as f32) * fast(PI) / fast(512.);
```

```
    *((t * 22.).sin_() + (t * 25.).sin_())
}; LEN];

plot_vector(example.moo_ref());
```

4 Euler's identity

This is just an example that shows complex math using slas works.

[7]:
```
im(std::f32::consts::PI).exp_()
```

[7]:
```
(-1 - 0.00000008742278im)
```

5 Cooley-Tukey in pseudo-code

```
FFT(x){
    if x.len < 2 then return x

    even = FFT(x[%2==0])
    odd  = FFT(x[%2==1])

    k = range(0, len / 2)

      = [e^exp(-2im * np.pi * k / N) * odd[k]]

    return [even[k] +  [k]].append([even[k] -  [k]])
}
```

6 Cooley-Tukey in Rust

This implementation of the cooley-tukey FFT algorithm is entirely statically allocated. Theoretically it should be able to just do one allocation (besides pointers), as it uses a return vector, allocated when calling the outer function, to store all temporary values, besides pointers.

[8]:
```
// x    = pointer to input vector.
//
// o    = pointer to output vector.
//
// len  = length of ipnut and output vectors.
//        (This function will not check if the length is valid,
//         Which is why `fft` should always be used instead.)
//
// ofset = 2 ^ recursion depth (starts at 0).

unsafe fn unsafe_fft<const LEN: usize>(x: *const Complex<f32>, o: *mut␣
 ↪Complex<f32>, len: usize, ofset: usize)
```

```
    -> *const Complex<f32>
{
    if len < 2{
        return x
    }

    // Recursively compute even and odd fourier coefficient's.
    // The even fourier coefficient will be stored at the right hand side of␣
 ↪the vector,
    // hence why o.add(len/2) is used as the output pointer for the function␣
 ↪call.
    let even = unsafe_fft::<LEN>(x, o.add(len/2), len/2, ofset * 2);

    // ofset is added to the input pointer, so the memory read from is shiftet␣
 ↪to the right,
    // This is done because 0 is even.
    let odd = unsafe_fft::<LEN>(x.add(ofset), o, len/2, ofset * 2);

    for k in 0..len/2{
        // Memory adress where the k'th even fourier coefficient is stored
        // (This is calculated by shifting the `even` pointer k*size_of(f32) to␣
 ↪the right)
        let even = *even.add(k);

        // Memory adress where the k'th odd fourier coefficient is stored
        let odd = *odd.add(k);

        // This will compute the  value,
        // which will temporarily be stored in the location of the positive␣
 ↪fourier coefficient.
        *o.add(k) = im(-2. * PI * k as f32 / len as f32).exp_();

        // After the negative fourier coefficient is calculated, as to not␣
 ↪overwrite  ,
        // which is also needed to compute the positive fourier coefficient.
        *o.add(k+len/2) = even - *o.add(k) * odd;

        // Positive fourier coefficient is computed.
        *o.add(k) = even + *o.add(k) * odd;
    }

    o
}

fn fft<const LEN: usize>(x: StaticVecRef<Complex<f32>, LEN>) -> [Complex<f32>;␣
 ↪LEN]{
```

```
    assert_eq!(LEN & (LEN - 1), 0);
    let mut ret = **x;
    unsafe{ unsafe_fft::<LEN>(x.as_ptr(), ret.as_mut_ptr(), LEN, 1) };
    ret
}
```

[9]:
```
let a = [re(1.), re(2.), re(3.), re(4.)];

let b = fft(a.moo_ref());

println!("{:#?}", b);
// 10.0 + 0.0im
// -2.0 + 2.0im
// -2.0 + 0.0im
// -2.0 - 2.0im
```

```
[
    (10 + 0im),
    (-1.9999999 + 2im),
    (-2 + 0im),
    (-2 - 2im),
]
```

[132]:
```
// Create some empty vectors to hold our wave sample,
// and fourier transformed result.
let mut y: [f32; LEN] = [0f32; LEN];
let mut y_hat: [Complex<f32>; LEN] = [re(0f32); LEN];

// Add two sinus waves together and write them to y and y_hat.
for i in 0..y_hat.len(){
    // t (time) is equal to i (index) times pi divided by the sample rate.
    // The sample rate could be pretty much any number, results may vary though.
 ↪..
    let t = fast(i as f32) * fast(PI) / fast(512.);

    // y(t) = sin(t * f_1) + sin(t * f_2) ... + sin(t * f_n)
    // In this example Hertz is just a made up unit,
    // there really is no units on any of these numbers.
    y[i] = *((t * 14.).sin_() + (t * 19.).sin_() + (t * 7.).sin_());
    y_hat[i] = re(y[i]);
}

plot_vector(y.moo_ref());
```

[133]:
```
// Fourier Magic...
y_hat = fft(y_hat.moo_ref());
```

```
// Copy the real part of all the fourier transformed points to a new vector.
for i in 0..y.len(){
    y[i] = y_hat[i].re.abs()
}
```

[133]:
```
()
```

[134]:
```
plot_vector(y[0..80].moo_ref::<80>());
// Here only the points from 0 to 80 are plotted.
// This is just to make the relevant peaks more visible.
//
// The frquencies used above for generating the y wave,
// should be the same as the labeled x component shown on the graph bellow.
// You can hover you mouse over the points on the graph to show theri␣
 ↪cordinates :)
```

[ ]:
```

```