```julia
using PlutoUI
```

Fourier transformation

Given a function $f(x)$, the Fourier transformation is defined as

$$\hat{f}(u) = \int_{-\infty}^{\infty} e^{-2\pi i u x} f(x) dx.$$

Its inverse process, or the inverse Fourier transformation is defined as

$$f(x) = \int_{-\infty}^{\infty} e^{2\pi i u x} \hat{f}(u) dk$$

Similarly, given a two variable function $f(x, y)$, the two dimensional Fourier transformation is

$$\hat{f}(u, v) = \int_{-\infty}^{\infty} dy \int_{-\infty}^{\infty} e^{-2\pi i (ux+vy)} f(x, y) dx.$$

The two dimensional inverse Fourier transformation is

$$f(x, y) = \int_{-\infty}^{\infty} du \int_{-\infty}^{\infty} e^{2\pi i (ux+vy)} \hat{f}(u, v) dv.$$

Fourier transformation can be used in

1. Image and audio compression,
2. Solving solid state system with translational invariance,
3. Understanding quantum Fourier transformation,
4. Understanding the Fourier optics.

The definition of Descrete Fourier Transformation (DFT)

A n dimensional quantum Fourier transformation.

$$y_i = \sum_{n=0}^{n-1} x_j \cdot e^{-\frac{i2\pi}{n} ij}$$

This transformation is linear, which can be represented as the DFT matrix

$$F_n = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2n-2} & \dots & \omega^{(n-1)^2} \end{pmatrix}$$

where $\omega = e^{ik}$ This transformation is also reversible, which can be represented as $F_n^\dagger/n$.

```
dft_matrix (generic function with 1 method)
```

```
function dft_matrix(n::Int)
    ω = exp(-2π*im/n)
    return [ω^((i-1)*(j-1)) for i=1:n, j=1:n]
end
```

4

```
@bind fourier_n NumberField(1:20, default=4)
```

```
Fn = 4×4 Matrix{ComplexF64}:
      1.0+0.0im             1.0+0.0im          …             1.0+0.0im
      1.0+0.0im   6.12323e-17-1.0im               -1.83697e-16+1.0im
      1.0+0.0im            -1.0-1.22465e-16im             -1.0-3.67394e-16im
      1.0+0.0im  -1.83697e-16+1.0im                 5.51091e-16-1.0im
```

```
Fn = dft_matrix(fourier_n)
```

```
4×4 Matrix{ComplexF64}:
          1.0+0.0im          …   8.22616e-17+8.32667e-17im
 -4.59243e-17-5.55112e-17im              0.0+6.12323e-17im
          0.0-6.12323e-17im     -2.10293e-17+4.979e-17im
  8.22616e-17-8.32667e-17im              1.0+0.0im
```

```
# dft matrix is unitary upto constant.
dft_matrix(fourier_n) * dft_matrix(fourier_n)' ./ fourier_n
```

The Cooley–Tukey's Fast Fourier transformation (FFT)

We have a recursive algorithm to compute the DFT.

$$F_n x = \begin{pmatrix} I_{n/2} & D_{n/2} \\ I_{n/2} & -D_{n/2} \end{pmatrix} \begin{pmatrix} F_{n/2} & 0 \\ 0 & F_{n/2} \end{pmatrix} \begin{pmatrix} x_{\rm odd} \\ x_{\rm even} \end{pmatrix}$$

where $D_n = {\rm diag}(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

Quiz: What is the computing time of a $F_n x$?

Hint: $T(n) = 2T(n/2) + O(n)$.

```julia
using Test, SparseArrays, LinearAlgebra
```

```
DefaultTestSet("fft decomposition", [], 1, false, false, true, 1.679387944491704e9, 1.67
```

```julia
@testset "fft decomposition" begin
    n = 4
    Fn = dft_matrix(n)
    F2n = dft_matrix(2n)

    # the permutation matrix to permute elements at 1:2:n (odd) to 1:n÷2 (top half)
    pm = sparse([iseven(j) ? (j÷2+n) : (j+1)÷2 for j=1:2n], 1:2n, ones(2n), 2n, 2n)

    # construct the D matrix
    ω = exp(-π*im/n)
    d1 = Diagonal([ω^(i-1) for i=1:n])

    # construct F_{2n} from F_n
    F2n_ = [Fn d1 * Fn; Fn -d1 * Fn]
    @test F2n * pm' ≈ F2n_
end
```

```
Test Summary:      | Pass  Total  Time
fft decomposition  |    1      1  0.5s
```

The Julia implementation

We implement the $O(n \log(n))$ time Cooley-Tukey FFT algorithm.

```
fft! (generic function with 1 method)
```

```julia
function fft!(x::AbstractVector{T}) where T
    N = length(x)
    @inbounds if N <= 1
        return x
    end

    # divide
    odd  = x[1:2:N]
    even = x[2:2:N]

    # conquer
    fft!(odd)
    fft!(even)

    # combine
    @inbounds for i=1:N÷2
       t = exp(T(-2im*π*(i-1)/N)) * even[i]
       oi = odd[i]
       x[i]     = oi + t
       x[i+N÷2] = oi - t
    end
    return x
end
```

```
DefaultTestSet("fft", [], 1, false, false, true, 1.679387946373228e9, 1.679387946674061e
```

```
@testset "fft" begin
    x = randn(ComplexF64, 8)
    @test fft!(copy(x)) ≈ dft_matrix(8) * x
end
```

```
Test Summary: | Pass  Total  Time
fft           |    1      1  0.3s
```

The Julia package `FFTW.jl` contains a super fast FFT implementation.

```
using FFTW
```

```
DefaultTestSet("fft", [], 1, false, false, true, 1.679387946874157e9, 1.679387947025221e
```

```
@testset "fft" begin
    x = randn(ComplexF64, 8)
    @test fft(copy(x)) ≈ dft_matrix(8) * x
end
```

```
Test Summary: | Pass  Total  Time
fft           |    1      1  0.2s
```

Application 1: Fast polynomial multiplication

Given two polynomials $p(x)$ and $q(x)$

$$p(x) = \sum_{k=0}^{n-1} a_k x^k$$

$$q(x) = \sum_{k=0}^{n-1} b_k x^k$$

The multiplication of them is defined as

$$p(x)q(x) = \sum_{k=0}^{2n-2} c_k x^k$$

1. Evaluate $p(x)$ and $q(x)$ at $2n$ points $\omega^0, \ldots, \omega^{2n-1}$ using DFT. This step takes time $O(n \log n)$.
2. Obtain the values of $p(x)q(x)$ at these 2n points through pointwise multiplication

$$\begin{aligned}(p \circ q)(\omega^0) &= p(\omega^0)q(\omega^0),\\ (p \circ q)(\omega^1) &= p(\omega^1)q(\omega^1),\\ &\vdots\\ (p \circ q)(\omega^{2n-1}) &= p(\omega^{2n-1})q(\omega^{2n-1}).\end{aligned}$$

This step takes time $O(n)$.

3. Interpolate the polynomial $p \circ q$ at the product values using inverse DFT to obtain coefficients $c_0, c_1, \ldots, c_{2n-2}$. This last step requires time $O(n \log n)$.

We can also use FFT to compute the convolution of two vectors $a = (a_0, \ldots, a_{n-1})$ and $b = (b_0, \ldots, b_{n-1})$, which is defined as a vector $c = (c_0, \ldots, c_{n-1})$ where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}, \qquad j = 0, \ldots, n-1.$$

The running time is again $O(n \log n)$.

```
using Polynomials
```

p = $1 + 3 \cdot x + 2 \cdot x^2 + 5 \cdot x^3 + 6 \cdot x^4$

```
p = Polynomial([1, 3, 2, 5, 6])
```

q = $3 + x + 6 \cdot x^2 + 2 \cdot x^3 + 2 \cdot x^4$

```
q = Polynomial([3, 1, 6, 2, 2])
```

Step 1: evaluate $p(x)$ at $2n - 1$ different points.

```
pvals =
 [17.0+0.0im, -4.49273-10.2802im, 1.73783+4.54839im, 0.5-6.06218im, -1.7451+1.83823im, -1
```

```
pvals = fft(vcat(p.coeffs, zeros(4)))
```

which is equivalent to computing:

```
 [17.0+0.0im, -4.49273-10.2802im, 1.73783+4.54839im, 0.5-6.06218im, -1.7451+1.83823im, -1
```

```
let
    n = 5
    ω = exp(-2π*im/(2n-1))
    map(k->p(ω^k), 0:(2n-1))
end
```

The same for $q(x)$.

```
qvals =
  [14.0+0.0im, 1.92855-8.96773im, -1.93242-0.0193026im, 0.5+2.59808im, 6.00387+3.75227im,
```

```
qvals = fft(vcat(q.coeffs, zeros(4)))
```

Step 2: Compute $p(x)q(x)$ at $2n-1$ points.

```
pqvals =
  [238.0+0.0im, -100.855+20.4636im, -3.27041-8.82294im, 16.0-1.73205im, -17.3749+4.48843ir
```

```
pqvals = pvals .* qvals
```

Step 3: Using the $2n-1$ point to fit the target polynomial.

```
  [3.0+0.0im, 10.0+0.0im, 15.0-5.95085e-16im, 37.0+0.0im, 43.0-6.8372e-16im, 46.0+6.8372e-
```

```
ifft(pqvals)
```

Summarize:

```
fast_polymul (generic function with 1 method)
```

```
function fast_polymul(p::AbstractVector, q::AbstractVector)
    pvals = fft(vcat(p, zeros(length(q)-1)))
    qvals = fft(vcat(q, zeros(length(p)-1)))
    pqvals = pvals .* qvals
    return real.(ifft(pqvals))
end
```

```
fast_polymul (generic function with 2 methods)
```

```
function fast_polymul(p::Polynomial, q::Polynomial)
    Polynomial(fast_polymul(p.coeffs, q.coeffs))
end
```

A similar algorithm has already been implemented in package `Polynomials`. One can easily verify the correctness.

$3 + 10 \cdot x + 15 \cdot x^2 + 37 \cdot x^3 + 43 \cdot x^4 + 46 \cdot x^5 + 50 \cdot x^6 + 22 \cdot x^7 + 12 \cdot x^8$

```
p * q
```

$3.0 + 10.000000000000002 \cdot x + 15.000000000000002 \cdot x^2 + 37.0 \cdot x^3 + 43.0 \cdot x^4 + 46.0 \cdot x^5 + 50.0 \cdot x^6 + 21.999999999999996 \cdot x^7 + 11.999999999999998 \cdot x^8$

```
fast_polymul(p, q)
```

Application 2: Image compression

If you google the logo of the Hong Kong University of Science and Technology, you will probably find the following png of size 2000×3000.

```
using Images
```

`img =`

```
img = Images.load("images/hkust-gz.png")
```

It is too large! We can compress it with the Fourier transformation algorithm. To simplify the discussion, let us using the gray scale image.

gray_image =


```julia
gray_image = Gray.(img)
```

```
Matrix{Gray{N0f8}} (alias for Array{Gray{Normed{UInt8, 8}}, 2})
```

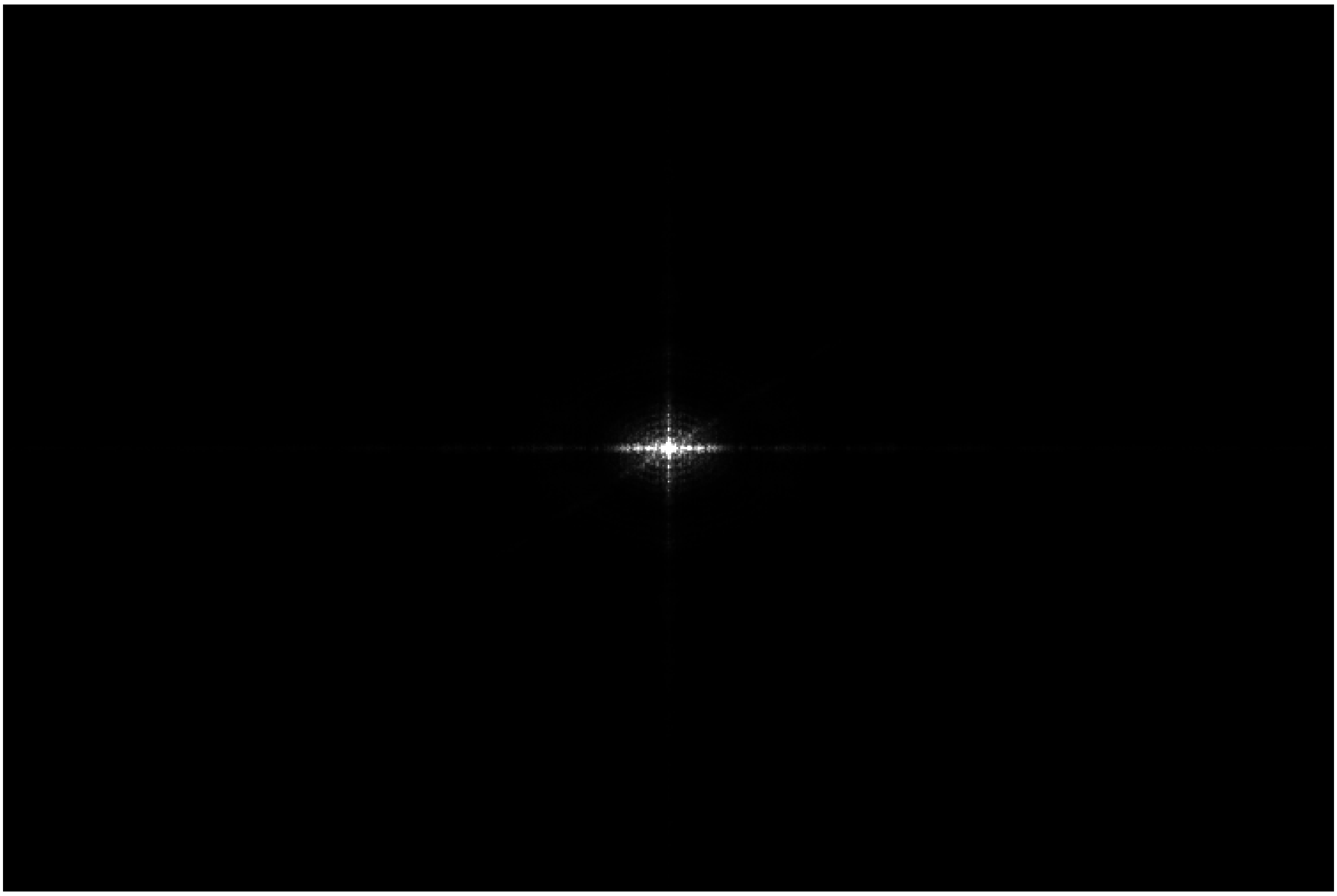
```julia
# The gray scale image uses 8-bit fixed point numbers as the pixel storage type.
typeof(gray_image)
```

```
img_data =
2000×3000 Matrix{Float32}:
 0.376471  0.376471  0.376471  0.376471  …  0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471  …  0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
 ⋮                                       ⋱
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471  …  0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
 0.376471  0.376471  0.376471  0.376471     0.376471  0.376471  0.376471  0.376471
```

```julia
img_data = Float32.(gray_image)
```

```
img_data_k =
2000×3000 Matrix{ComplexF32}:
  14.1177-9.69074f-7im  -16.5889+1.66456im  …   9.90118-5.09644im  -16.5889-1.66456im
 -12.1824-10.0836im      19.6015+5.63716im     -5.73792-3.81363im   10.0119+14.3428im
  7.55086+13.5317im     -17.3148-3.91325im     -1.96508+12.5134im  -2.42017-22.694im
 -3.00379-9.20718im       11.248-1.95294im      10.6497-14.1396im  -4.79038+19.1743im
  1.09106+1.94913im     -5.93928+3.75958im     -17.5779+7.40923im   11.0074-5.10588im
 -2.26381+2.12877im      5.33122+2.14914im  …   19.7864+2.65691im  -14.9784-10.0922im
  4.27849-0.864077im    -8.88375-11.6353im     -15.1348-9.3566im      14.901+16.4903im
         ⋮                                  ⋱
  4.27848+0.864073im      14.901-16.4903im      15.2803-5.02406im  -8.88375+11.6353im
 -2.26382-2.12877im     -14.9784+10.0922im  …  -9.13847-2.98708im   5.33123-2.14914im
  1.09106-1.94913im      11.0074+5.10589im      2.30744+7.36655im  -5.93928-3.75958im
  -3.0038+9.20717im     -4.79038-19.1743im     -0.67651-4.75071im    11.248+1.95294im
  7.55085-13.5317im     -2.42017+22.694im       4.40713-2.43937im  -17.3148+3.91326im
 -12.1824+10.0836im      10.0119-14.3429im     -8.99988+7.49633im   19.6015-5.63716im
```

```julia
img_data_k = fftshift(fft(img_data))
```



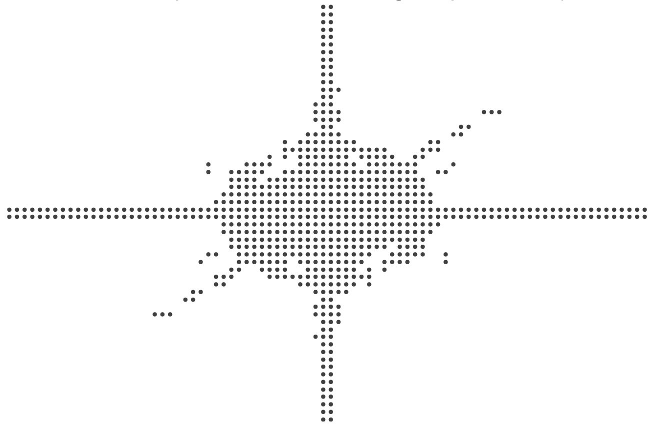
```julia
# it is sparse!
Gray.(abs2.(img_data_k) ./ length(img_data_k))
```

We can store it in the sparse matrix format.

100

```julia
@bind tolerence Slider(1:1000; default=100, show_value=true)
```

sparse_img = 2000×3000 SparseMatrixCSC{ComplexF32, Int64} with 183662 stored entries:



```
sparse_img = let
    # let us discard all variables smaller than 1e-5
    img_data_k[abs.(img_data_k) .< tolerence] .= 0
    sparse(img_data_k)
end
```

compression_ratio = 0.030610333333333333

```
compression_ratio = nnz(sparse_img) / (2000 * 3000)
```

```
recovered_img =
2000×3000 Matrix{ComplexF32}:
 0.375853+1.55815f-9im   0.374952-3.13009f-10im  …   0.37396+1.25688f-10im
 0.375397+3.36788f-9im   0.374721+1.78173f-9im      0.373507+2.74718f-9im
 0.379161+3.16208f-9im   0.378495+1.77227f-9im      0.377427+1.67699f-9im
 0.379559+2.96936f-9im   0.378829+1.69112f-9im      0.377953+1.81772f-9im
 0.377977+2.37799f-9im   0.377086+5.19308f-10im     0.376568+1.72905f-9im
 0.377675+1.52573f-9im   0.376744-1.54441f-10im  …  0.376586-2.61279f-11im
 0.375919+2.27278f-9im   0.374757-9.21654f-11im     0.375595+9.73378f-10im
          ⋮                                      ⋱
 0.378013+1.96365f-9im   0.377307+1.12899f-9im      0.376072+2.04565f-10im
 0.376327+3.28278f-9im   0.375426+3.341f-9im     …  0.374943+1.53107f-9im
 0.373405+1.73329f-9im   0.371949+2.01569f-9im      0.372425-2.35389f-11im
 0.374815+3.24951f-9im   0.372966+2.16595f-9im      0.373752+1.83784f-9im
  0.37447-3.27971f-10im  0.372698-8.31304f-10im     0.373063-2.00152f-9im
 0.375094+2.73001f-9im   0.373747+1.05759f-9im      0.373381+1.76576f-9im
```

```
recovered_img = ifft(fftshift(Matrix(sparse_img)))
```


```
Gray.(abs.(recovered_img))
```

Assignment

Watch this YouTube video: https://youtu.be/jnxqHcObNK4

Use what you have learnt to solve the analyse the following sequential data.

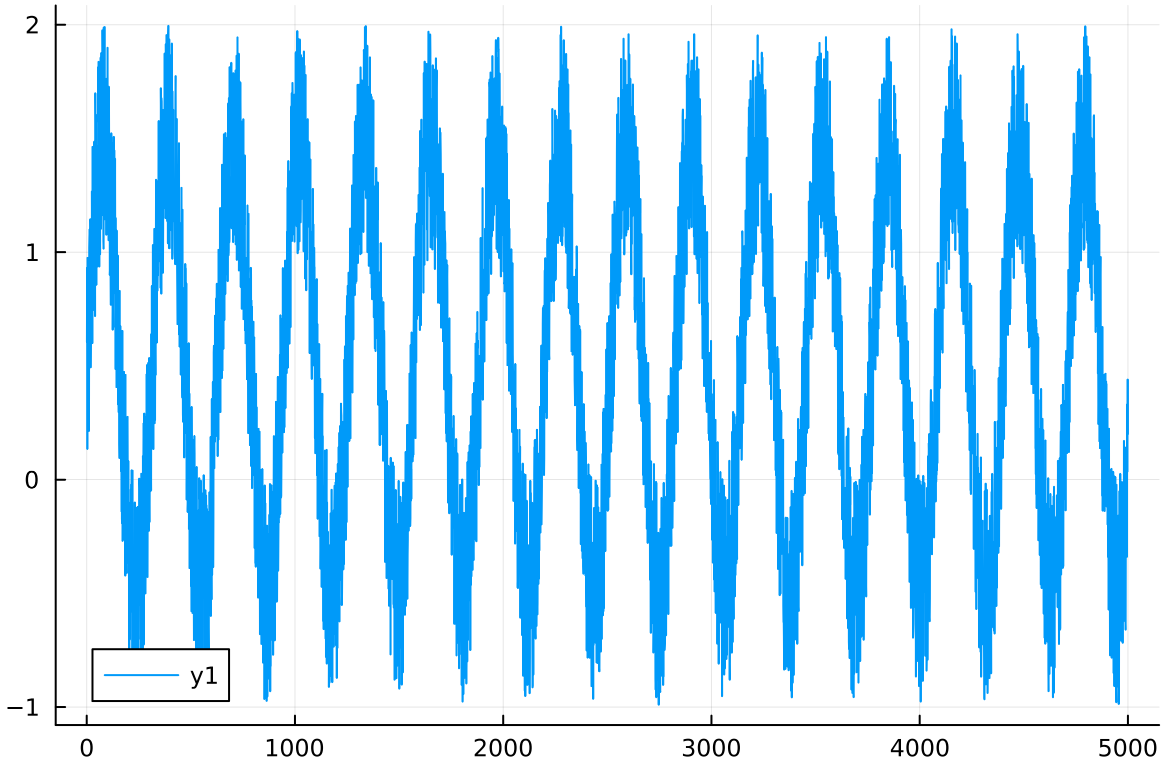
N = 5000

```
N = 5000
```

brain_signal =

[0.605677, 0.932417, 0.135376, 0.975365, 0.522075, 0.629284, 0.569592, 0.216832, 0.34012

```
brain_signal = sin.(LinRange(0, 1000, N) ./ 10) .+ rand(N)
```




```
plot(brain_signal)
```

Here, we use the Ricker wavelet to analyse the above wave function.

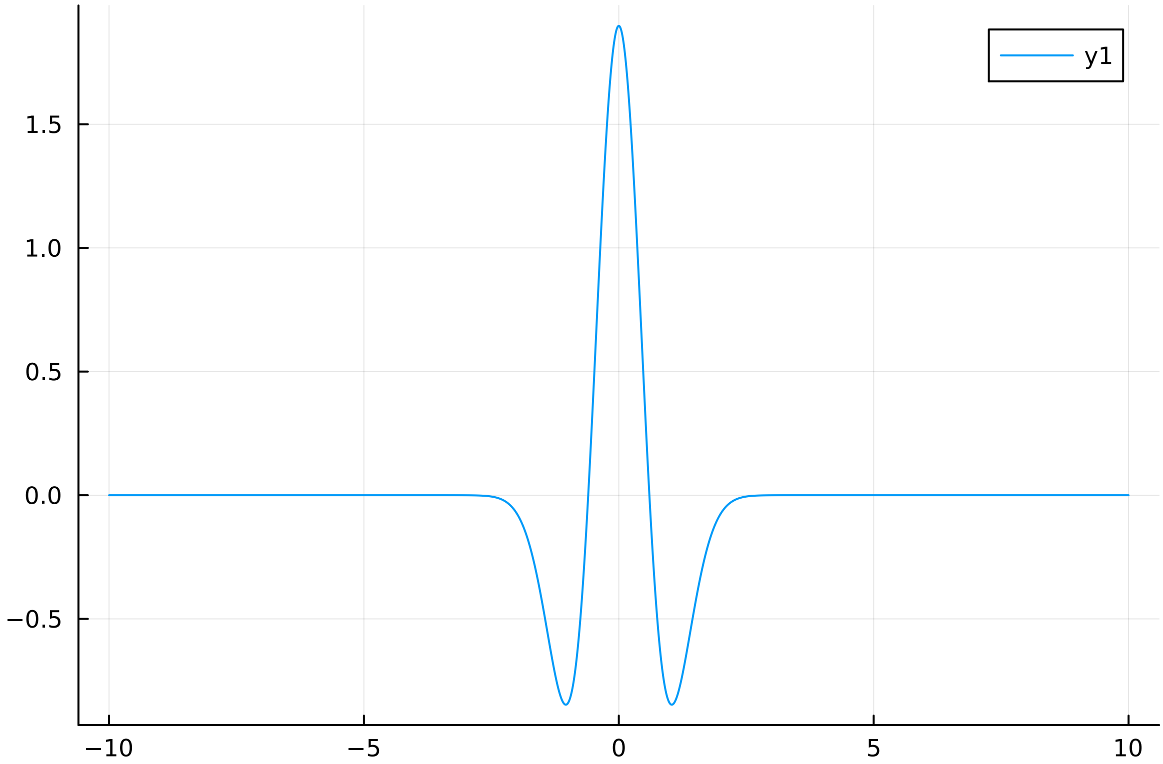
$$A\left(1-\left(\frac{x}{a}\right)^2\right)e^{-\frac{x^2}{2a^2}},$$

where $A = \frac{8}{\sqrt{3a\pi}}$.

```
ricker (generic function with 1 method)
```

```
function ricker(x, a)
    A = 8/π/sqrt(3a)
    return A * (1 - (x/a)^2) * exp(-x^2/a^2/2)
end
```

```
using Plots
```




```julia
let
    x = -10:0.01:10
    y = ricker.(x, 0.6)
    plot(x, y)
end
```

Tasks

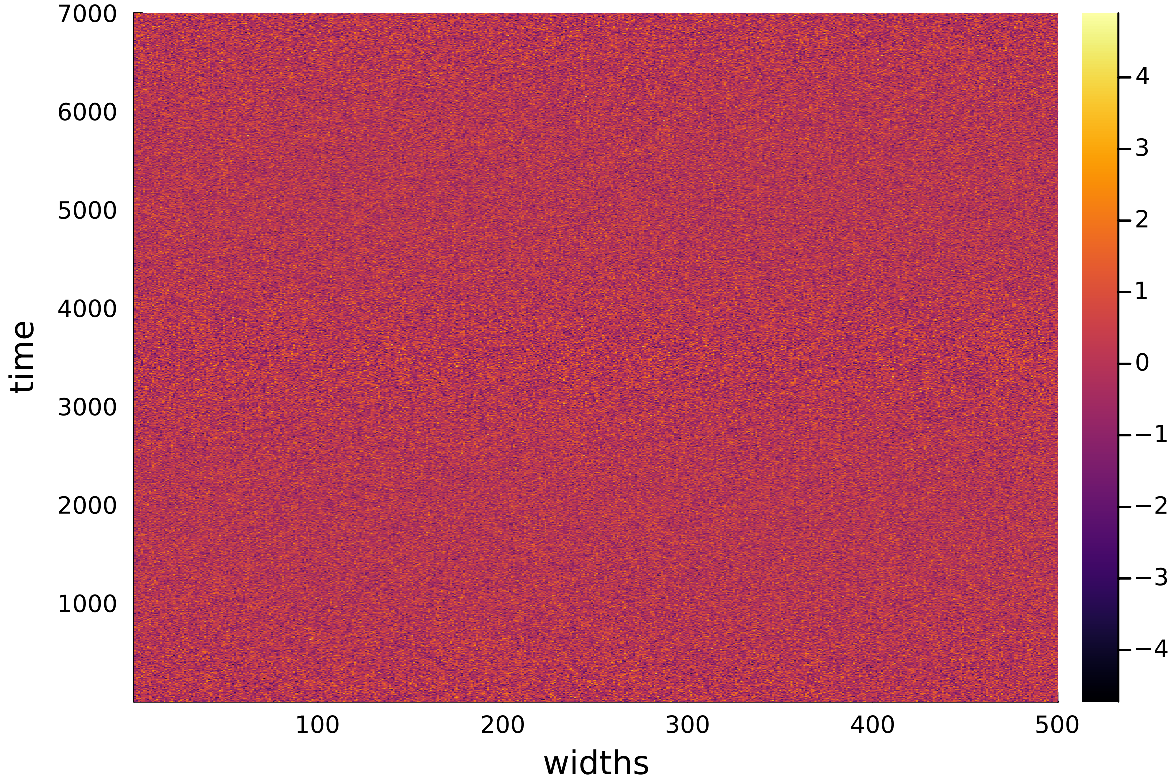
Please help me fix the following code to let the output be what we want. You need to implement the wavelet transformation `z = wavelet_transformation(x, y)`, such that

$$z_i = \sum_j x_{j-i} y_j$$

You are supposed to implement the fast wavelet transformation that having time complexity $O(n\log(n))$ where n is the size of x and y.

```
wavelet_transformation (generic function with 1 method)
```

```julia
function wavelet_transformation(signal::AbstractVector{T}, fw) where T
    # TODO: please remove the following line and add your own implementation!
    resulting_vector = randn(length(signal) + length(fw)-1)
    return resulting_vector
end
```




```julia
# this is the test program
let
    # the width parameter 'a' in the Ricker wavelet is 1..500
    widths = 1:N÷10
    res = []
    for (j, a) in enumerate(widths)
        fw = ricker.(-1000:1000, a)    # the descretized wavelet of width 'a'
        res_a = wavelet_transformation(brain_signal, fw)
        push!(res, res_a)
    end
    heatmap(hcat(res...); ylabel="time", xlabel="widths")
end
```

In the submission (pull request), the following contents should be included

1. The correct implementation of the `wavelet_transformation` function.
2. The output image created by the above code block,
3. An interpretation of the output image.