III.2 Discrete Fourier Transform

In the last lecture we explored using the trapezium rule for approximating Fourier coefficients. This is a linear map from function values to coefficients and thus can be reinterpreted as a matrix-vector product, called the the Discrete Fourier Transform. It turns out the matrix is unitary which leads to important properties including interpolation. Finally, we discuss how a clever way of decomposing the DFT leads to a fast way of applying and inverting it, which is one of the most influencial algorithms of the 20th century: the Fast Fourier Transform.

1. The Discrete Fourier Transform (DFT): We discuss the map from values to approximate Fourier coefficients, and back.
2. Interpolation: We show that the approximate Fourier expansion *exactly* interpolates the values at the sample grid.
3. The Fast Fourier Transform (FFT): We discuss how the DFT can be applied in $O(n \log n)$ operations.

1. The Discrete Fourier transform

Definition 1 (DFT) The *Discrete Fourier Transform (DFT)* is defined as:

$$Q_n := \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \mathrm{e}^{-\mathrm{i}\theta_1} & \mathrm{e}^{-\mathrm{i}\theta_2} & \cdots & \mathrm{e}^{-\mathrm{i}\theta_{n-1}} \\ 1 & \mathrm{e}^{-\mathrm{i}2\theta_1} & \mathrm{e}^{-\mathrm{i}2\theta_2} & \cdots & \mathrm{e}^{-\mathrm{i}2\theta_{n-1}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \mathrm{e}^{-\mathrm{i}(n-1)\theta_1} & \mathrm{e}^{-\mathrm{i}(n-1)\theta_2} & \cdots & \mathrm{e}^{-\mathrm{i}(n-1)\theta_{n-1}} \end{bmatrix}$$

$$= \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)^2} \end{bmatrix}$$

for the n-th root of unity $\omega = \mathrm{e}^{\mathrm{i}\pi/n}$. Note that

$$Q_n^\star = \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & e^{i\theta_1} & e^{i2\theta_1} & \cdots & e^{i(n-1)\theta_1} \\ 1 & e^{i\theta_2} & e^{i2\theta_2} & \cdots & e^{i(n-1)\theta_2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{i\theta_{n-1}} & e^{i2\theta_{n-1}} & \cdots & e^{i(n-1)\theta_{n-1}} \end{bmatrix}$$

$$= \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{(n-1)} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix}$$

Note that

$$\underbrace{\begin{bmatrix} f_0^n \\ \vdots \\ f_{n-1}^n \end{bmatrix}}_{\mathbf{f}^n} = \frac{1}{\sqrt{n}} Q_n \underbrace{\begin{bmatrix} f(\theta_0) \\ \vdots \\ f(\theta_n) \end{bmatrix}}_{\mathbf{f}^n}$$

The choice of normalisation constant is motivated by the following:

Proposition 1 (DFT is Unitary) $Q_n \in U(n)$, that is, $Q_n^\star Q_n = Q_n Q_n^\star = I$.

Proof

$$Q_n Q_n^\star = \begin{bmatrix} \Sigma_n[1] & \Sigma_n[e^{i\theta}] & \cdots & \Sigma_n[e^{i(n-1)\theta}] \\ \Sigma_n[e^{-i\theta}] & \Sigma_n[1] & \cdots & \Sigma_n[e^{i(n-2)\theta}] \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_n[e^{-i(n-1)\theta}] & \Sigma_n[e^{-i(n-2)\theta}] & \cdots & \Sigma_n[1] \end{bmatrix} = I$$

∎

In other words, Q_n is easily inverted and we also have a map from discrete Fourier coefficients back to values:

$$\sqrt{n} Q_n^\star \mathbf{f}^n = \mathbf{f}^n$$

2. Interpolation

We investigated briefly interpolation and least squares using polynomials at evenly spaced points, observing that there were issues with stability. We now show that the DFT actually gives coefficients that interpolate using Fourier expansions. As the DFT is a unitary matrix its (2-norm) condition number is 1, hence this is a stable process. Thus we arrive at the main result:

Corollary 1 (Interpolation)

$$f_n(\theta) := \sum_{k=0}^{n-1} f_k^n e^{ik\theta}$$

interpolates f at θ_j:

$$f_n(\theta_j) = f(\theta_j)$$

Proof We have

$$f_n(\theta_j) = \sum_{k=0}^{n-1} f_k^n e^{ik\theta_j} = \sqrt{n}\mathbf{e}_j^\top Q_n^\star \mathbf{f}^n = \mathbf{e}_j^\top Q_n^\star Q_n \mathbf{f}^n = f(\theta_j).$$

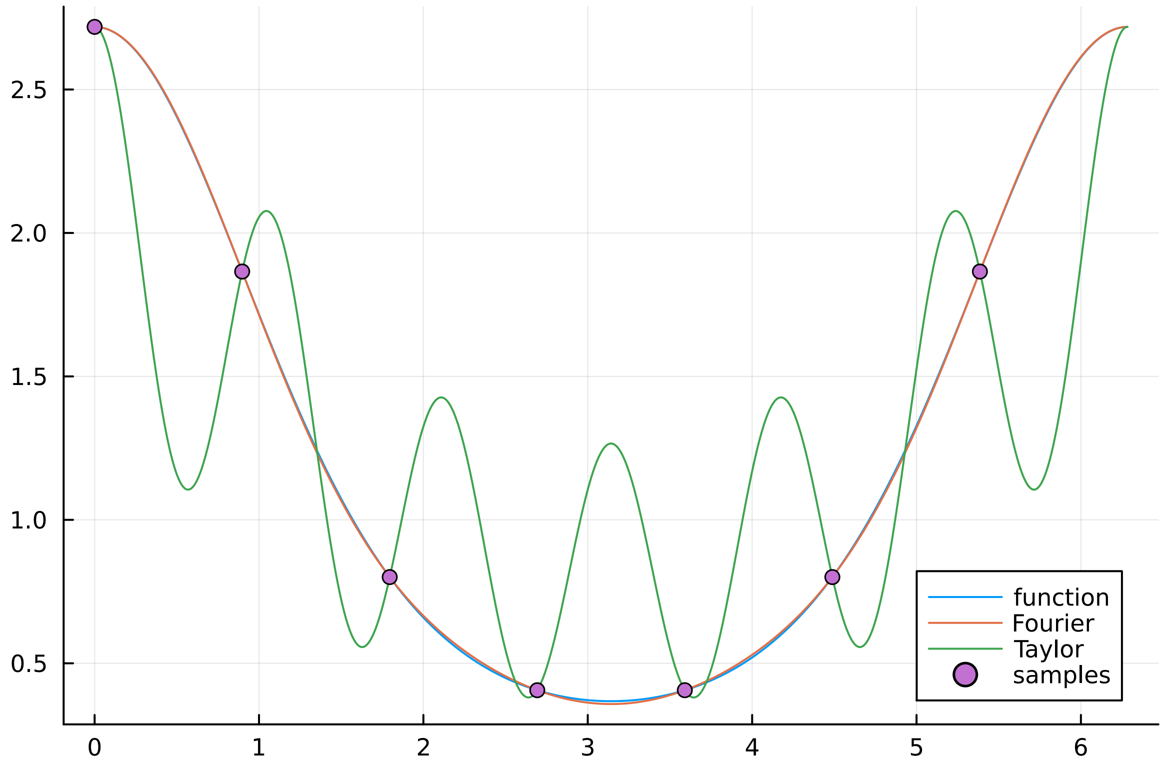
∎

We will leave extending this result to the general non-Taylor case to the problem sheet. Note that regardless of choice of coefficients we interpolate provided we have n consecutive coefficients, though some interpolations are better than others:

In [1]:
```julia
using Plots, LinearAlgebra


# evaluates f_n at a point
function finitefourier(𝐟ₙ, θ)
    m = n ÷ 2 # use coefficients between -m:m
    ret = 0.0 + 0.0im # coefficients are complex so we need complex arithmet
    for k = 0:m
        ret += 𝐟ₙ[k+1] * exp(im*k*θ)
    end
    for k = -m:-1
        ret += 𝐟ₙ[end+k+1] * exp(im*k*θ)
    end
    ret
end

function finitetaylor(𝐟ₙ, θ)
    ret = 0.0 + 0.0im # coefficients are complex so we need complex arithmet
    for k = 0:n-1
        ret += 𝐟ₙ[k+1] * exp(im*k*θ)
    end
    ret
end


f = θ -> exp(cos(θ))
n = 7
θ = range(0,2π; length=n+1)[1:end-1] # θ_0, …,θ_{n-1}, dropping θ_n == 2π
Qₙ = 1/sqrt(n) * [exp(-im*(k-1)*θ[j]) for k = 1:n, j=1:n]
𝐟ₙ = 1/sqrt(n) * Qₙ * f.(θ)
```

```
fₙ = θ -> finitefourier(𝔣ₙ, θ)
tₙ = θ -> finitetaylor(𝔣ₙ, θ)

g = range(0, 2π; length=1000) # plotting grid
plot(g, f.(g); label="function", legend=:bottomright)
plot!(g, real.(fₙ.(g)); label="Fourier")
plot!(g, real.(tₙ.(g)); label="Taylor")
scatter!(θ, f.(θ); label="samples")
```

Out[1]:



We now demonstrate the relationship of Taylor and Fourier coefficients and their discrete approximations for some examples:

Example 1 Consider the function

$$f(\theta) = \frac{2}{2 - \mathrm{e}^{\mathrm{i}\theta}}$$

Under the change of variables $z = \mathrm{e}^{\mathrm{i}\theta}$ we know for z on the unit circle this becomes (using the geometric series with $z/2$)

$$\frac{2}{2 - z} = \sum_{k=0}^{\infty} \frac{z^k}{2^k}$$

i.e., $\mathfrak{f}_k = 1/2^k$ which is absolutely summable:

$$\sum_{k=0}^{\infty} |\mathfrak{f}_k| = f(0) = 2.$$

If we use an n point discretisation we get (using the geoemtric series with 2^{-n})

$$f_k^n = f_k + f_{k+n} + f_{k+n} + \cdots = \sum_{p=0}^{\infty} \frac{1}{2^{k+pn}} = \frac{2^{n-k}}{2^n - 1}$$

We can verify this numerically:

```
In [2]: f = θ -> 2/(2 - exp(im*θ))
        n = 7
        θ = range(0,2π; length=n+1)[1:end-1] # θ_0, …,θ_{n-1}, dropping θ_n == 2π
        Qₙ = 1/sqrt(n) * [exp(-im*(k-1)*θ[j]) for k = 1:n, j=1:n]

        Qₙ/sqrt(n)*f.(θ) ≈ 2 .^ (n .- (0:n-1)) / (2^n-1)
```

```
Out[2]: true
```

Example 2 Define the following infinite sum (which has no name apparently, according to Mathematica):

$$S_n(k) := \sum_{p=0}^{\infty} \frac{1}{(k+pn)!}$$

We can use the DFT to compute $S_n(0), \ldots, S_n(n-1)$. Consider

$$f(\theta) = \exp(e^{i\theta}) = \sum_{k=0}^{\infty} \frac{e^{ik\theta}}{k!}$$

where we know the Fourier coefficients from the Taylor series of e^z. The discrete Fourier coefficients satisfy for $0 \leq k \leq n-1$:

$$f_k^n = f_k + f_{k+n} + f_{k+2n} + \cdots = S_n(k)$$

Thus we have

$$\begin{bmatrix} S_n(0) \\ \vdots \\ S_n(n-1) \end{bmatrix} = \frac{1}{\sqrt{n}} Q_n \begin{bmatrix} 1 \\ \exp(e^{2i\pi/n}) \\ \vdots \\ \exp(e^{2i(n-1)\pi/n}) \end{bmatrix}$$

3. Fast Fourier Transform (non-examinable)

Applying Q_n or its adjoint $Q_n^\star$ to a vector naively takes $O(n^2)$ operations. Both can be reduced to $O(n \log n)$ using the celebrated *Fast Fourier Transform* (FFT), which is one of the Top 10 Algorithms of the 20th Century (You won't believe number 7!).

The key observation is that hidden in Q_{2n} are 2 copies of Q_n. We will work with multiple n we denote the n-th root as $\omega_n = \exp(2\pi/n)$. Note that we can relate a vector of powers of ω_{2n} to two copies of vectors of powers of ω_n:

$$\underbrace{\begin{bmatrix} 1 \\ \omega_{2n} \\ \vdots \\ \omega_{2n}^{2n-1} \end{bmatrix}}_{\vec{\omega}_{2n}} = P_\sigma^\top \begin{bmatrix} I_n \\ \omega_{2n} I_n \end{bmatrix} \underbrace{\begin{bmatrix} 1 \\ \omega_n \\ \vdots \\ \omega_n^{n-1} \end{bmatrix}}_{\vec{\omega}_n}$$

where σ has the Cauchy notation

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n & n+1 & \cdots & 2n \\ 1 & 3 & 5 & \cdots & 2n-1 & 2 & \cdots & 2n \end{pmatrix}$$

That is, P_σ is the following matrix which takes the even entries and places them in the first n entries and the odd entries in the last n entries:

In [3]:
```
n = 4
σ = [1:2:2n-1; 2:2:2n]
P_σ = I(2n)[σ,:]
```

Out[3]:
```
8×8 Matrix{Bool}:
 1  0  0  0  0  0  0  0
 0  0  1  0  0  0  0  0
 0  0  0  0  1  0  0  0
 0  0  0  0  0  0  1  0
 0  1  0  0  0  0  0  0
 0  0  0  1  0  0  0  0
 0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  1
```

and so $P_\sigma^\top$ reverses the process. Thus we have

$$\begin{aligned} Q_{2n}^\star &= \frac{1}{\sqrt{2n}} \left[\mathbf{1}_{2n} | \vec{\omega}_{2n} | \vec{\omega}_{2n}^2 | \cdots | \vec{\omega}_{2n}^{2n-1} \right] \\ &= \frac{1}{\sqrt{2n}} P_\sigma^\top \begin{bmatrix} \mathbf{1}_n & \vec{\omega}_n & \vec{\omega}_n^2 & \cdots & \vec{\omega}_n^{n-1} & \vec{\omega}_n^n & \cdots & \vec{\omega}_n^{2n-1} \\ \mathbf{1}_n & \omega_{2n}\vec{\omega}_n & \omega_{2n}^2\vec{\omega}_n^2 & \cdots & \omega_{2n}^{n-1}\vec{\omega}_n^{n-1} & \omega_{2n}^n\vec{\omega}_n^n & \cdots & \omega_{2n}^{2n-1}\vec{\omega}_n^{2n-1} \end{bmatrix} \\ &= \frac{1}{\sqrt{2}} P_\sigma^\top \begin{bmatrix} Q_n^\star & Q_n^\star \\ Q_n^\star D_n & -Q_n^\star D_n \end{bmatrix} = \frac{1}{\sqrt{2}} P_\sigma^\top \begin{bmatrix} Q_n^\star & \\ & Q_n^\star \end{bmatrix} \begin{bmatrix} I_n & I_n \\ D_n & -D_n \end{bmatrix} \end{aligned}$$

In other words, we reduced the DFT to two DFTs applied to vectors of half the dimension.

We can see this formula in code:

In [4]:
```
function fftmatrix(n)
    θ = range(0,2π; length=n+1)[1:end-1] # θ_0, …,θ_{n-1}, dropping θ_n == 2
    [exp(-im*(k-1)*θ[j]) for k = 1:n, j=1:n]/sqrt(n)
end

Q₂ₙ = fftmatrix(2n)
Qₙ = fftmatrix(n)
```

```
Dₙ = Diagonal([exp(im*k*π/n) for k=0:n-1])
(P_σ'*[Qₙ' Qₙ'; Qₙ'*Dₙ -Qₙ'*Dₙ])[1:n,1:n] ≈ sqrt(2)Q₂ₙ'[1:n,1:n]
```

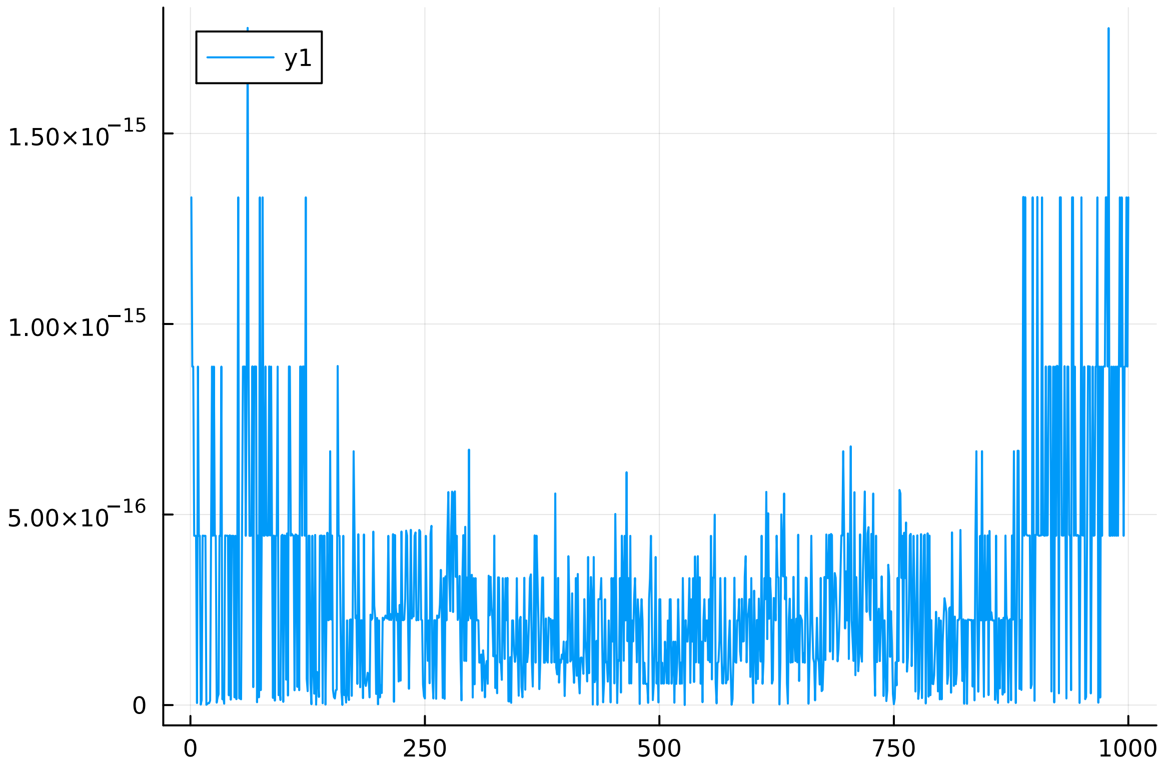
Out[4]:
```
true
```

Now assume $n = 2^q$ so that $\log_2 n = q$. To see that we get $O(n \log n) = O(nq)$ operations we need to count the operations. Assume that applying F_n takes $\leq 3nq$ additions and multiplications. The first n rows takes n additions. The last n has n multiplications and n additions. Thus we have $6nq + 3n \leq 6n(q+1) = 3(2n)\log_2(2n)$ additions/multiplications, showing by induction that we have $O(n \log n)$ operations.

Remark The FFTW.jl package wraps the FFTW (Fastest Fourier Transform in the West) library, which is a highly optimised implementation of the FFT that also works well even when n is not a power of 2. (As an aside, the creator of FFTW Steven Johnson is now a Julia contributor and user.) Here we approximate $\exp(\cos(\theta - 0.1))$ using 31 nodes:

In [5]:
```
using FFTW
f = θ -> exp(cos(θ-0.1))
n = 31
m = n÷2
# evenly spaced points from 0:2π, dropping last node
θ = range(0, 2π; length=n+1)[1:end-1]

# fft returns discrete Fourier coefficients n*[f̂ⁿ_0, …, f̂ⁿ_(n-1)]
fc = fft(f.(θ))/n

# We reorder using [f̂ⁿ_(-m), …, f̂ⁿ_(-1)] == [f̂ⁿ_(n-m), …, f̂ⁿ_(n-1)]
#  == [f̂ⁿ_(m+1), …, f̂ⁿ_(n-1)]
f̂ = [fc[m+2:end]; fc[1:m+1]]

# equivalent to f̂ⁿ_(-m)*exp(-im*m*θ) + … + f̂ⁿ_(m)*exp(im*m*θ)
fₙ = θ -> transpose([exp(im*k*θ) for k=-m:m]) * f̂

# plotting grid
g = range(0, 2π; length=1000)
plot(abs.(fₙ.(g) - f.(g)))
```

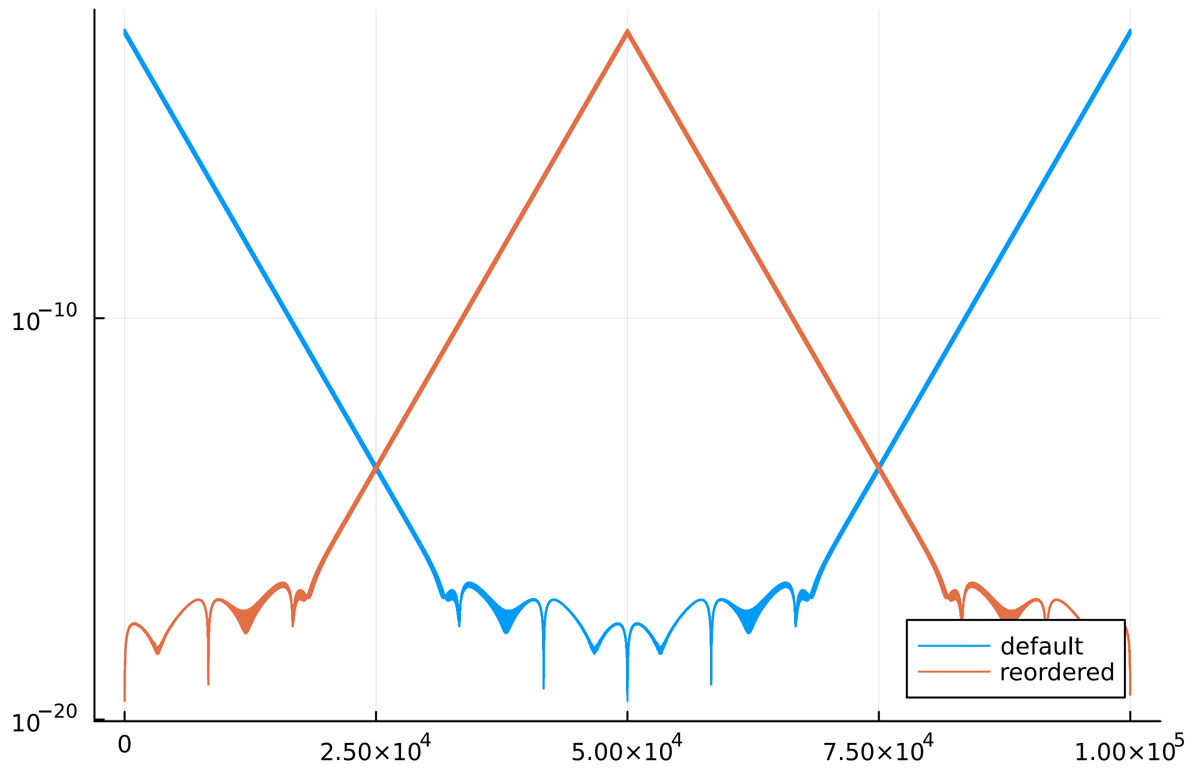
Out[5]:



Thus we have successfully approximate the function to roughly machine precision. The magic of the FFT is because it's $O(n \log n)$ we can scale it to very high orders. Here we plot the Fourier coefficients for a function that requires around 100k coefficients to resolve:

In [6]:
```
f = θ -> exp(sin(θ))/(1+1e6cos(θ)^2)
n = 100_001
m = n÷2
# evenly spaced points from 0:2π, dropping last node
θ = range(0, 2π; length=n+1)[1:end-1]

# fft returns discrete Fourier coefficients n*[f̂ⁿ_0, …, f̂ⁿ_(n-1)]
fc = fft(f.(θ))/n


# We reorder using [f̂ⁿ_(-m), …, f̂ⁿ_(-1)] == [f̂ⁿ_(n-m), …, f̂ⁿ_(n-1)]
#  == [f̂ⁿ_(m+1), …, f̂ⁿ_(n-1)]
f̂ = [fc[m+2:end]; fc[1:m+1]]

plot(abs.(fc); yscale=:log10, legend=:bottomright, label="default")
plot!(abs.(f̂); yscale=:log10, label="reordered")
```

```
Out[6]:
```



10^{-10}
10^{-20}
0
2.50×10^4
5.00×10^4
7.50×10^4
1.00×10^5
default
reordered