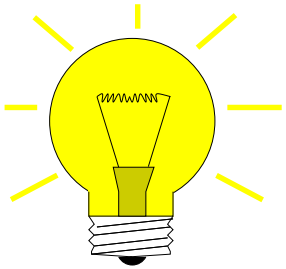
6 Trigonometric Interpolation

Is there something else than polynomials and Taylor approximation in the world?



Idea (J. Fourier 1822): Approximation of a function not by usual polynomials but by trigonometrical polynomials = partial sum of a Fourier series

We call trigonometrical polynomial of degree $\leq 2m$ the function

$$T_{2m}(t) := t \mapsto \sum_{j=-m}^{m} \gamma_j e^{2\pi i j t} , \qquad \gamma_j \in \mathbb{C}, t \in \mathbb{R} .$$

Remark 6.0.1. $T_{2m} : \mathbb{R} \to \mathbb{C}$ is periodic of period 1. Moreover,
if $\gamma_{-j} = \overline{\gamma_j}$ for all $j = 0, \dots, 2m$, then $T_{2m}(t)$ takes only real values and may be written as

$$T_{2m}(t) = \frac{a_0}{2} + \sum_{j=1}^{2m} \left(a_j \cos(2\pi jt) + b_j \sin(2\pi jt)\right)$$

with $a_0 = 2\gamma_0$ and $a_j = 2\,\mathrm{Re}\,\gamma_j$, $b_j = -2\,\mathrm{Im}\,\gamma_j$ for all $j = 1, \dots, 2m$.

Remark 6.0.2.
The functions $w_j(t) = e^{2\pi ijt}$ are orthogonal with respect to the $L^2(]0,1[)$ scalar product.

Let us take as granted (or known from lectures in Analysis, Mathematcial Methods of Physics, etc.):

Theorem 6.0.1 (L^2-convergence of the Fourier series)**.** *Every squared integrable function* $f \in L^2(]0,1[) := \{f :]0,1[\mapsto \mathbb{C} \colon \|f\|_{L^2(]0,1[)} < \infty\}$ *is the* $L^2(]0,1[)$*-limit of its Fourier series*

$$f(t) = \sum_{k=-\infty}^{\infty} \widehat{f}(k)\, e^{2\pi ikt} \quad \text{in } L^2(]0,1[)\ ,$$

with Fourier coefficients defined by

$$\widehat{f}(k) = \int_0^1 f(t)\, e^{-2\pi ikt}\, \mathrm{d}t\ , \quad k \in \mathbb{Z}\ .$$

Remark 6.0.3. In view of this theorem, we may think of *one* function in *two ways*: once in the *time (or space) domain* $t \to f(t)$ and once in the *frequency domain* $k \to \widehat{f}(k)$.

Isometry property (Parseval): $$\sum_{k=-\infty}^{\infty} |\widehat{f}(k)|^2 = \|f\|^2_{L^2(]0,1[)} \tag{6.0.1}$$

Remark 6.0.4. A real function f may be equally represented as

$$\frac{a_0}{2} + \sum_{j=1}^{\infty} \big(a_j \cos(2\pi jt) + b_j \sin(2\pi jt)\big) \quad \text{in } L^2(]0,1[) \ ,$$

with

$$a_j = 2\int_0^1 f(t)\ \cos(2\pi jt)\,\mathrm{d}t \ , \quad j \geq 0 \ ,$$
$$b_j = 2\int_0^1 f(t)\ \sin(2\pi jt)\,\mathrm{d}t \ , \quad j \geq 1 \ .$$

Lemma 6.0.2 (Derivative and Fourier coefficients)**.**
$f \in L^1(]0,1[)$ & $f' \in L^1(]0,1[) \quad \Rightarrow \quad \widehat{f'}(k) = 2\pi i k \widehat{f}(k), \quad k \in \mathbb{Z}.$

Remark 6.0.5.

$$\left\| f^{(n)} \right\|^2_{L^2([0,1])} = (2\pi)^{2n} \sum_{k=-\infty}^{\infty} k^{2n} |\widehat{f}(k)|^2 \ . \tag{6.0.2}$$

If $|\widehat{f^{(n)}}(k)| \leq \int_0^1 |f^{(n)}(t)|\,\mathrm{d}t < \infty \quad \Rightarrow \quad \widehat{f}(k) = O(k^{-n})$ for $|k| \to \infty$.

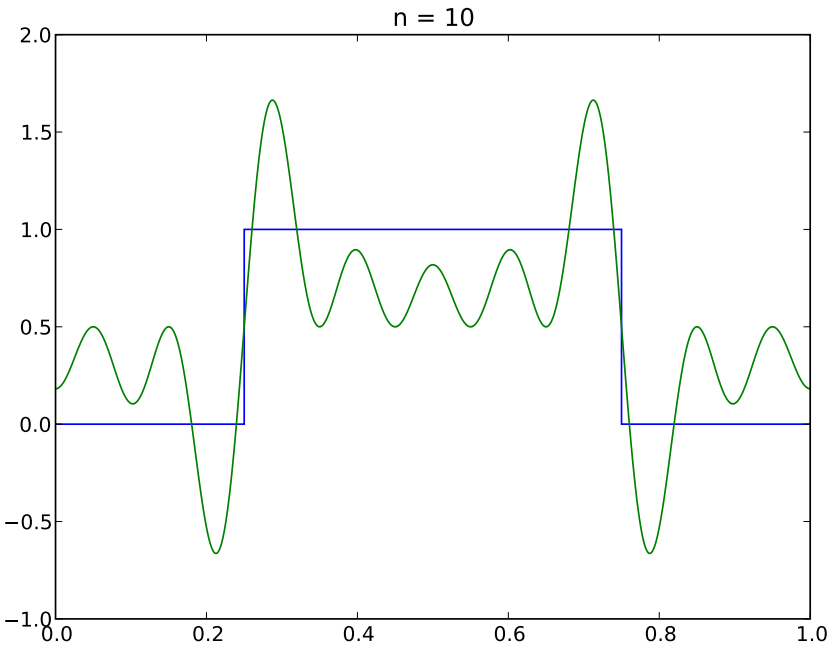
The smoothness of a function directly reflects in the quick decay of its Fourier coefficients.

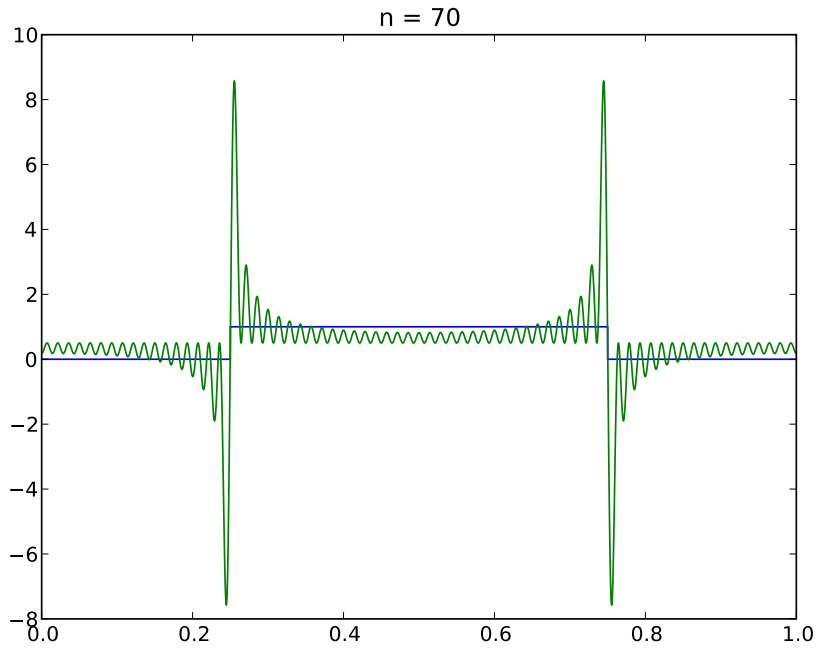
Example 6.0.6. The Fourier series associated with the characteristic function of an interval $[a,b] \subset]0.1[$ may be computed analytically as

$$b - a + \frac{1}{\pi} \sum_{|k|=1}^{\infty} e^{-ikc} \frac{\sin kd}{k} e^{i2\pi kt} , \quad t \in [0,1] ,$$

with $c = \pi(a+b)$ and $d = \pi(b-a)$.

Note the slow decay of the Fourier coefficients, and hence expect slow convergence of the series. Moreover, observe in the pictures below the Gibbs phenomenon: the ripples move closer to the discontinuities and increase with larger n. Explanation: we have L^2-convergence but no uniforme convergence of the series!





Remark 6.0.7. Usually one cannot compute analytically $\widehat{f}(k)$ or one has to rely only on discrete values at nodes $x_\ell = \frac{\ell}{N}$ for $\ell = 0, 1, \ldots, N$; the trapezoidal rule gives

$$\hat{f}(k) \approx \frac{1}{N} \sum_{\ell=0}^{N-1} f(x_\ell) e^{-2\pi i k x_\ell} =: \hat{f}_N(k) \tag{6.0.3}$$

6.1 Discrete Fourier Transform (DFT)

Let $n \in \mathbb{N}$ fixed and denote the nth root of unity $\omega_n := \exp(-2\pi i/n) = \cos(2\pi/n) - i\sin(2\pi/n)$

hence
$$\omega_n^k = \omega_n^{k+n}\ \forall k \in \mathbb{Z} \quad , \quad \omega_n^n = 1 \quad , \quad \omega_n^{n/2} = -1 \ , \tag{6.1.1}$$

$$\sum_{k=0}^{n-1} \omega_n^{kj} = \begin{cases} n & \text{, if } j = 0 \ , \\ 0 & \text{, if } j \neq 0 \ . \end{cases} \tag{6.1.2}$$

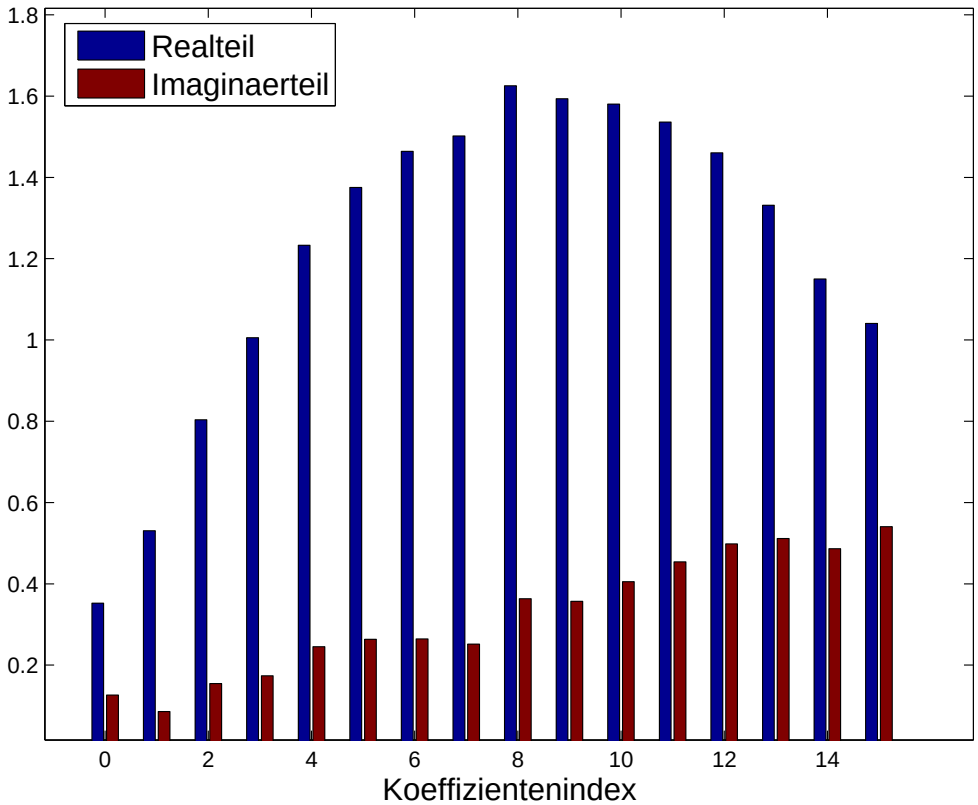
A change of basis in $\mathbb{C}^n$:

standard basis of $\mathbb{K}^n$ Trigonometrical Basis

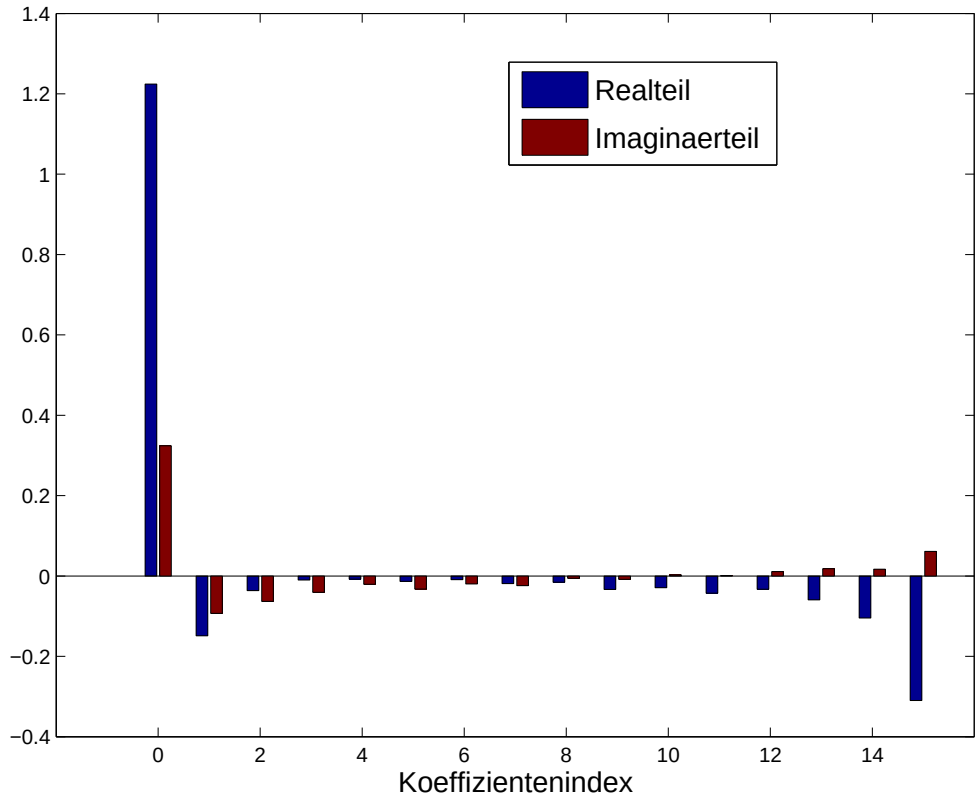
$$\left\{ \begin{pmatrix} 1 \\ 0 \\ \vdots \\ \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \end{pmatrix} \cdots \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ \vdots \\ \\ \vdots \\ 0 \\ 1 \end{pmatrix} \right\} \longleftarrow \left\{ \begin{pmatrix} \omega_n^0 \\ \vdots \\ \\ \\ \vdots \\ \omega_n^0 \end{pmatrix} \begin{pmatrix} \omega_n^0 \\ \omega_n^1 \\ \vdots \\ \\ \vdots \\ \omega_n^{n-1} \end{pmatrix} \cdots \begin{pmatrix} \omega_n^0 \\ \omega_n^{n-2} \\ \omega_n^{2(n-2)} \\ \vdots \\ \vdots \\ \omega_n^{(n-1)(n-2)} \end{pmatrix} \begin{pmatrix} \omega_n^0 \\ \omega_n^{n-1} \\ \omega_n^{2(n-1)} \\ \vdots \\ \vdots \\ \omega_n^{(n-1)^2} \end{pmatrix} \right\}$$

Matrix of change of basis trigonometrical basis $\longrightarrow$ standard basis: Fourier-matrix

$$\mathbf{F}_n = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{pmatrix} = \left(\omega_n^{ij}\right)_{i,j=0}^{n-1} \in \mathbb{C}^{n,n} . \qquad (6.1.3)$$



Vector in standardbasis



Vector in trigonometrical basis

Definition 6.1.1 (Diskrete Fourier transform)**.** *We call linear map* $\mathcal{F}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathcal{F}_n(\mathbf{y}) := \mathbf{F}_n\mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, *discrete Fourier transform (DFT), i.e. for* $\mathbf{c} := \mathcal{F}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j\, \omega_n^{kj} \quad , \quad k = 0, \dots, n-1 \; . \tag{6.1.4}$$

Convention: in the discussion of the DFT: vektor indexes run from 0 to $n-1$.

Lemma 6.1.2.
The scaled Fourier-matrix $\frac{1}{\sqrt{n}}\mathbf{F}_n$ *is unitary:* $\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^H = \frac{1}{n}\overline{\mathbf{F}}_n$

Remark 6.1.1. $\frac{1}{n}\mathbf{F}_n^2 = -I$ and $\frac{1}{n^2}\mathbf{F}_n^4 = I$, hence the eigenvalues of $\mathbf{F}_n$ are in the set $\{1, -1, i, -i\}$.

numpy-functions: `c=fft(y)` $\leftrightarrow$ `y=ifft(c);`

Example 6.1.2 (Frequency analysis with DFT).

Some vectors of the Fourier basis ($n = 16$):

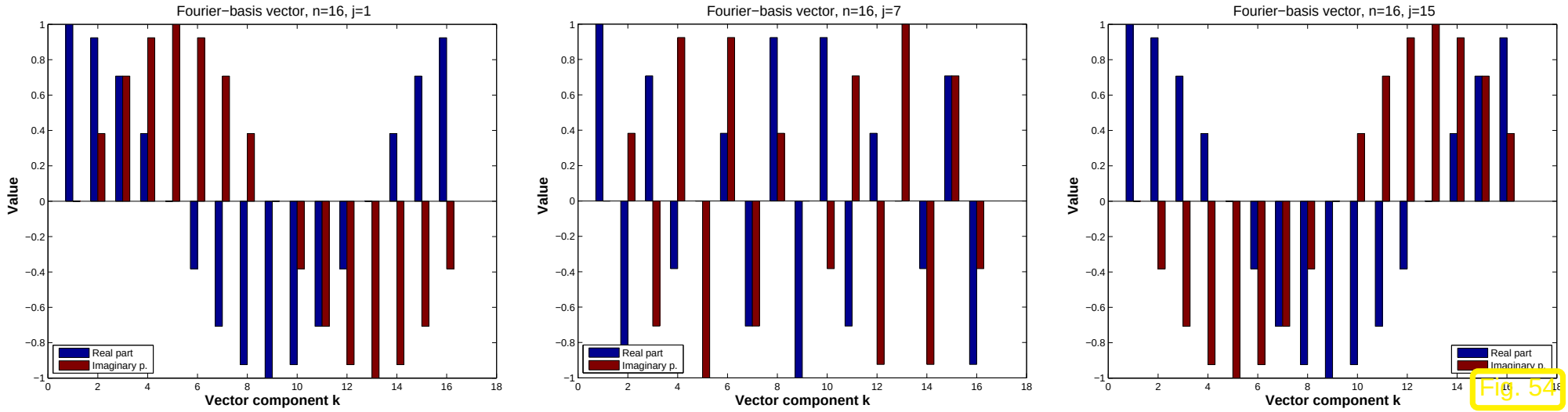


Fig. 54

"'low frequence" "'high frequence" "'low frequence"

Extraction of characteristical frequencies from a distorted discrete periodical signal:

```
from numpy import sin, pi, linspace, random, fft
from pylab import plot, bar, show
t = linspace(0,63,64); x = sin(2*pi*t/64)+sin(7*2*pi*t/64)
y = x + random.randn(len(t)) %distortion
c = fft.fft(y); p = abs(c)**2/64
plot(t,y,'-+'); show()
bar(t[:32],p[:32]); show()
```

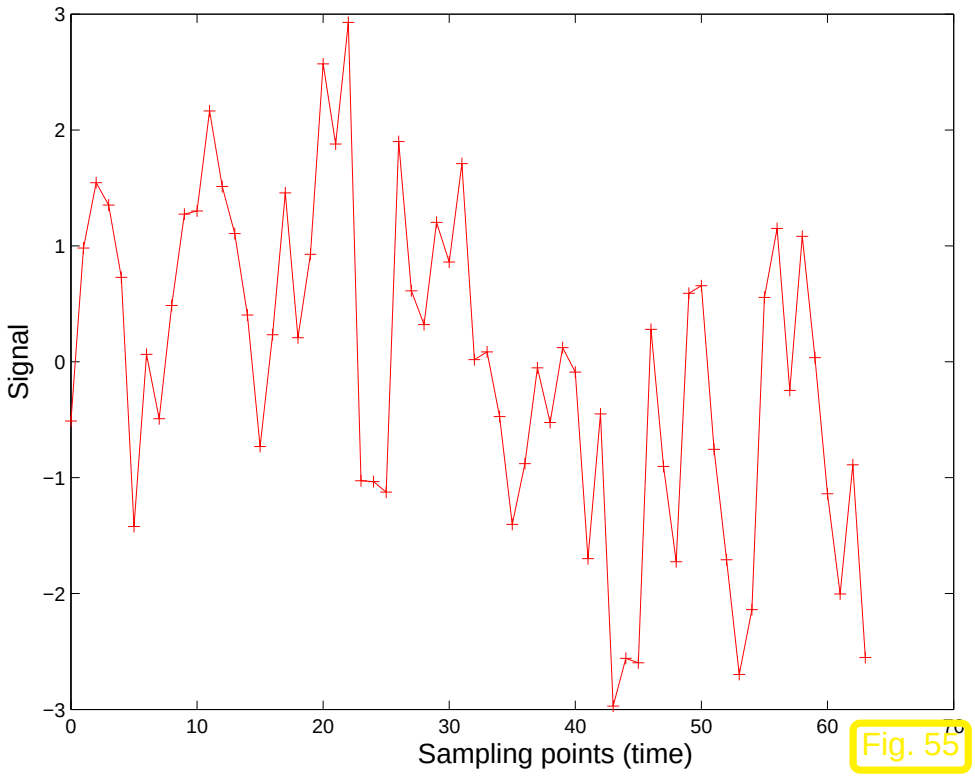


Fig. 55

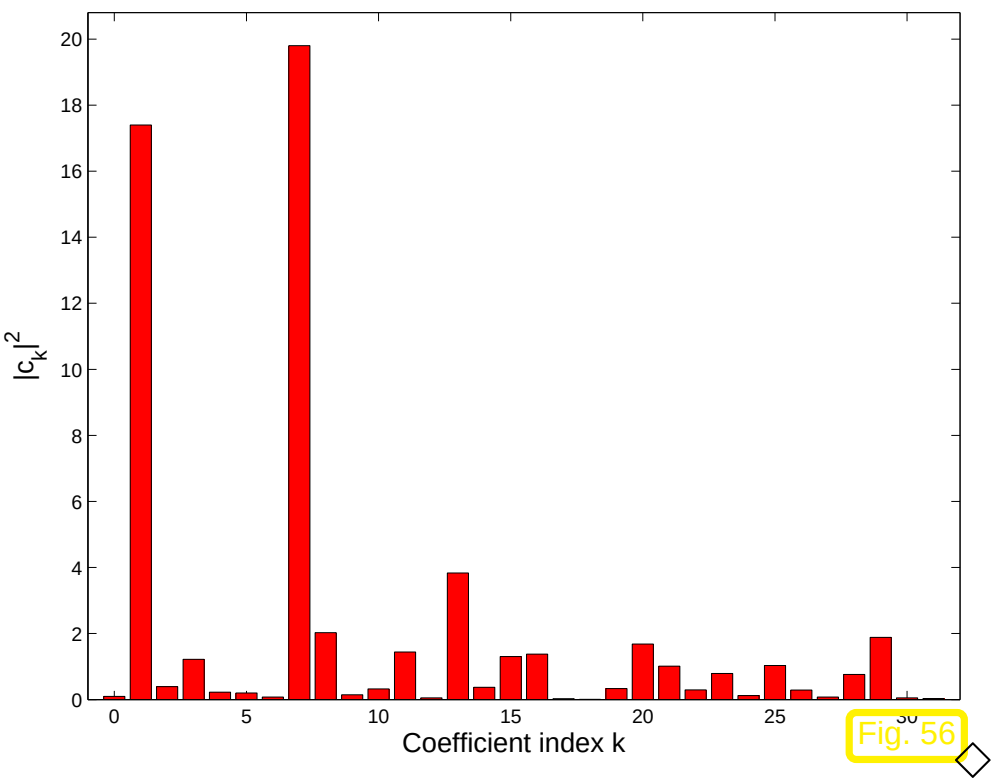


Fig. 56

6.2 Fast Fourier Transform (FFT)

At first glance (at (6.1.4)): DFT in $\mathbb{C}^n$ seems to require asymptotic computational effort of $O(n^2)$ (matrix$\times$vector multiplication with dense matrix).

Example 6.2.1 (Efficiency of `fft`).

`tic-toc`-timing: compare `fft`, loop based implementation, and direct matrix multiplication

Code 6.2.2: timing of different implementations of DFT

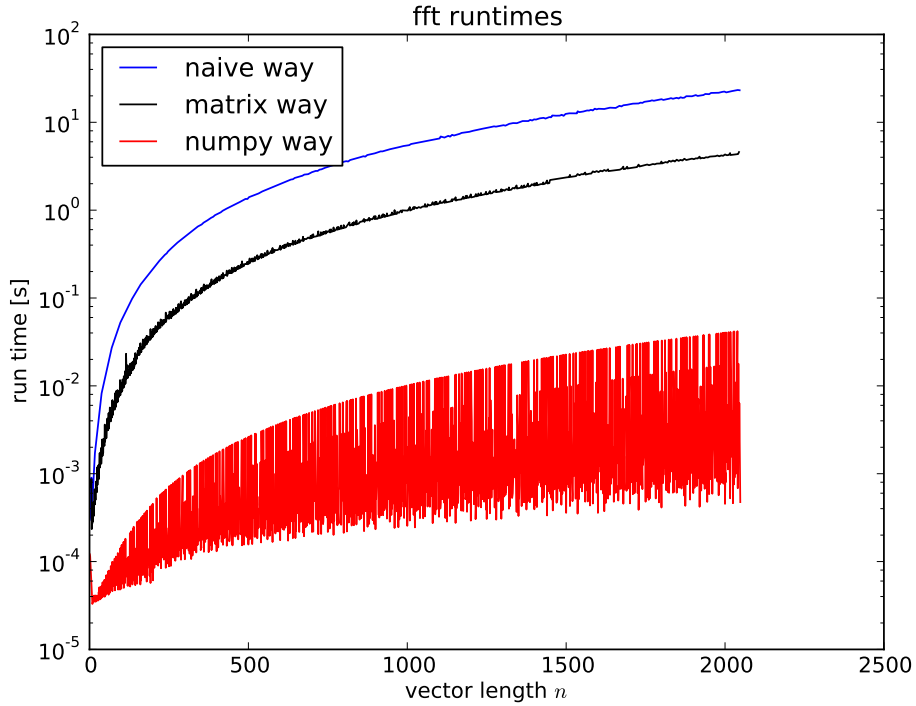
```
from numpy import zeros, random, exp, pi, meshgrid, r_, dot,  fft
import timeit

def naiveFT():
    global y, n

    c = 0.*1j*y
    # naive
    omega = exp(-2*pi*1j/n)
    c[0] = y.sum(); s = omega
    for jj in xrange(1,n):
        c[jj] = y[n-1]
        for kk in xrange(n-2,-1,-1): c[jj] = c[jj]*s+y[kk]
        s *= omega
    #return c

def matrixFT():
```

```
    global y, n

    # matrix based
    I, J = meshgrid(r_[:n], r_[:n])
    F = exp(-2*pi*1j*I*J/n)
    tm = 10**3
    c = dot(F,y)
    #return c

def fftFT():
    global y,n
    c = fft.fft(y)
    #return c

nrexp = 5
N = 2**11 # how large the vector will be
res = zeros((N,4))
for n in xrange(1,N+1):
    y = random.rand(n)

    t = timeit.Timer('naiveFT()', 'from __main__ import naiveFT')
    tn = t.timeit(number=nrexp)
    t = timeit.Timer('matrixFT()', 'from __main__ import matrixFT')
```

```
    tm = t.timeit(number=nrexp)
    t = timeit.Timer('fftFT()', 'from __main__ import fftFT')
    tf = t.timeit(number=nrexp)
    #print n, tn, tm, tf
    res[n-1] = [n, tn, tm, tf]

print res[:,3]
from pylab import semilogy, show, plot, savefig
semilogy(res[:,0], res[:,1], 'b-')
semilogy(res[:,0], res[:,2], 'k-')
semilogy(res[:,0], res[:,3], 'r-')
sevefig('fftime.eps')
show()
```

naive DFT-implementation

```
c = 0.*1j*y
omega = exp(-2*pi*1j/n)
c[0] = y.sum(); s = omega
for jj in xrange(1,n):
        c[jj] = y[n-1]
        for kk in xrange(n-2,-1,-
        s *= omega
```



Incredible! The `fft()`-function clearly beats the $O(n^2)$ asymptotic complexity of the other implementations. Note the logarithmic scale!

◇

The secret of `fft()`:

the Fast Fourier Transform algorithm [15]

(discovered by C.F. Gauss in 1805, rediscovered by Cooley & Tuckey in 1965, one of the "top ten algorithms of the century").

An elementary manipulation of (6.1.4) for $n = 2m$, $m \in \mathbb{N}$:

$$\begin{aligned} c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n} jk} \\ &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n}(2j+1)k} \\ &= \underbrace{\sum_{j=0}^{m-1} y_{2j} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\omega_m^{jk}}}_{=:\widetilde{c}_k^{\mathtt{even}}} + e^{-\frac{2\pi i}{n} k} \cdot \underbrace{\sum_{j=0}^{m-1} y_{2k+1} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\omega_m^{jk}}}_{=:\widetilde{c}_k^{\mathtt{odd}}} . \end{aligned} \tag{6.2.1}$$

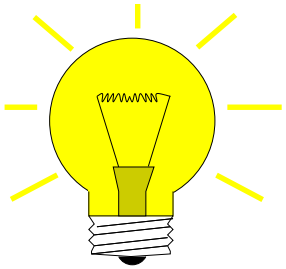
Note m-periodicity: $\widetilde{c}_k^{\mathtt{even}} = \widetilde{c}_{k+m}^{\mathtt{even}}$, $\widetilde{c}_k^{\mathtt{odd}} = \widetilde{c}_{k+m}^{\mathtt{odd}}$.

Note: $\widetilde{c}_k^{\texttt{even}}$, $\widetilde{c}_k^{\texttt{odd}}$ from DFTs of length m!

with $\mathbf{y}_{\texttt{even}} := (y_0, y_2, \dots, y_{n-2})^T \in \mathbb{C}^m$: $\left(\widetilde{c}_k^{\texttt{even}}\right)_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\texttt{even}}$,

with $\mathbf{y}_{\texttt{odd}} := (y_1, y_3, \dots, y_{n-1})^T \in \mathbb{C}^m$: $\left(\widetilde{c}_k^{\texttt{odd}}\right)_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\texttt{odd}}$.

(6.2.1): DFT of length $2m$ = 2$\times$ DFT of length m + 2m additions & multiplications



Idea:

divide & conquer recursion

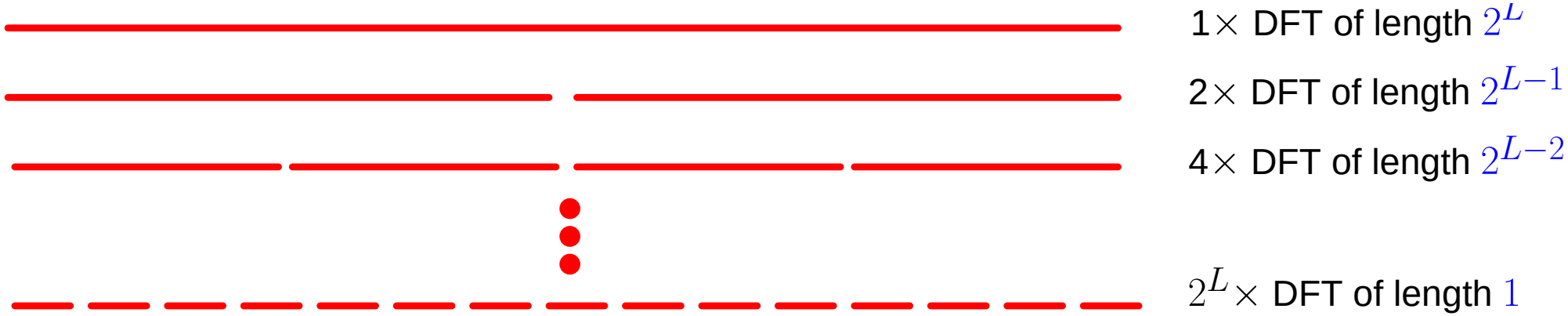
(for DFT of length $n = 2^L$)

FFT-algorithm

Code 6.2.3: Recursive FFT

```
from numpy import linspace, random,
  hstack, pi, exp
def fftrec(y):
    n = len(y)
    if n == 1:
        c = y
        return c
    else:
        c0 = fftrec(y[::2])
        c1 = fftrec(y[1::2])
        r = (-2.j*pi/n) *
          linspace(0,n-1,n)
        c = hstack((c0,c0)) +
          exp(r) *   hstack((c1,c1))
    return c
```

Computational cost of `fftrec`:



Code 6.2.2: each level of the recursion requires $O(2^L)$ elementary operations.

Asymptotic complexity of FFT algorithm, $n = 2^L$: $O(L2^L) = O(n \log_2 n)$

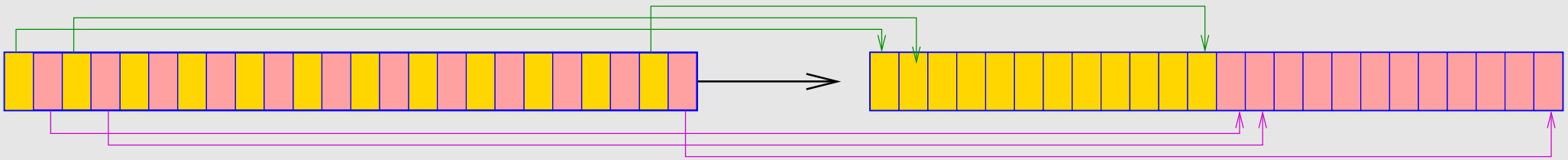
(`fft`-function: cost $\approx 5n \log_2 n$).

➢

Remark 6.2.4 (FFT algorithm by matrix factorization).

For $n = 2m$, $m \in \mathbb{N}$,

permutation $P_m^{\mathrm{OE}}(1,\dots,n) = (1,3,\dots,n-1,2,4,\dots,n)$.



As $\omega_n^{2j} = \omega_m^j$:

permutation of rows $$P_m^{\mathrm{OE}}\mathbf{F}_n = \left(\begin{array}{c|c} \mathbf{F}_m & \mathbf{F}_m \\ \hline \mathbf{F}_m \begin{pmatrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n/2-1} \end{pmatrix} & \mathbf{F}_m \begin{pmatrix} \omega_n^{n/2} & & & \\ & \omega_n^{n/2+1} & & \\ & & \ddots & \\ & & & \omega_n^{n-1} \end{pmatrix} \end{array}\right)$$

$$\left(\begin{array}{c|c} \mathbf{F}_m & \\ \hline & \mathbf{F}_m \end{array}\right) \left(\begin{array}{c|c} \mathbf{I} & \mathbf{I} \\ \hline \begin{matrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n/2-1} \end{matrix} & \begin{matrix} -\omega_n^0 & & & \\ & -\omega_n^1 & & \\ & & \ddots & \\ & & & -\omega_n^{n/2-1} \end{matrix} \end{array}\right)$$

Example: factorization of Fourier matrix for $n = 10$

$$P_5^{\mathrm{OE}}\mathbf{F}_{10} = \left(\begin{array}{ccccc|ccccc}
\omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\
\omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 \\
\omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 & \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 \\
\omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 & \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 \\
\omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 & \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 \\
\hline
\omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 & \omega^8 & \omega^9 \\
\omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^2 & \omega^5 & \omega^8 & \omega^1 & \omega^4 & \omega^7 \\
\omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 \\
\omega^0 & \omega^7 & \omega^4 & \omega^1 & \omega^8 & \omega^5 & \omega^2 & \omega^9 & \omega^6 & \omega^3 \\
\omega^0 & \omega^9 & \omega^8 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1
\end{array}\right) , \quad \omega := \omega_{10} .$$

△

What if $n \neq 2^L$**?** Quoted from MATLAB manual:

To compute an n-point DFT when n is composite (that is, when $n = pq$), the FFTW library decomposes the problem using the Cooley-Tukey algorithm, which first computes p transforms of size q, and then computes q transforms of size p. The decomposition is applied recursively to both the p- and q-point DFTs until the problem can be solved using one of several machine-generated fixed-size

"codelets." The codelets in turn use several algorithms in combination, including a variation of Cooley-Tukey, a prime factor algorithm, and a split-radix algorithm. The particular factorization of n is chosen heuristically.

The execution time for fft depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors $\longrightarrow$ Ex. 6.2.1.

Remark 6.2.5 (FFT based on general factorization).

Fast Fourier transform algorithm for DFT of length $n = pq$, $p, q \in \mathbb{N}$ (Cooley-Tuckey-Algorithm)

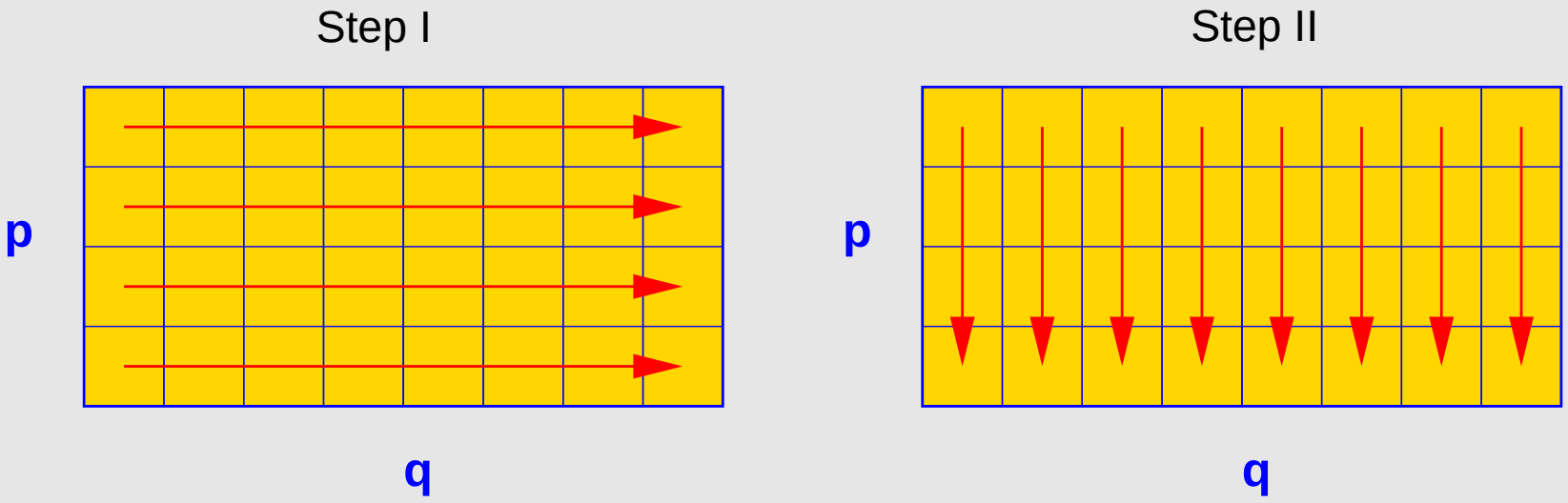
$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{jk} \overset{[j=:lp+m]}{=} \sum_{m=0}^{p-1} \sum_{l=0}^{q-1} y_{lp+m} e^{-\frac{2\pi i}{pq}(lp+m)k} = \sum_{m=0}^{p-1} \omega_n^{mk} \sum_{l=0}^{q-1} y_{lp+m}\, \omega_q^{l(k \mod q)}. \tag{6.2.2}$$

Step I: perform p DFTs of length q $\quad z_{m,k} := \sum_{l=0}^{q-1} y_{lp+m}\, \omega_q^{lk}, \quad 0 \le m < p, 0 \le k < q.$

Step II: for $k =: rq + s, \quad 0 \le r < p, 0 \le s < q$

$$c_{rq+s} = \sum_{m=0}^{p-1} e^{-\frac{2\pi i}{pq}(rq+s)m} z_{m,s} = \sum_{m=0}^{p-1} \left(\omega_n^{ms} z_{m,s}\right) \omega_p^{mr}$$

and hence q DFTs of length p give all c_k.



Remark 6.2.6 (FFT for prime n).

When $n \neq 2^L$, even the Cooley-Tuckey algorithm of Rem. 6.2.5 will eventually lead to a DFT for a vector with prime length.

Quoted from the MATLAB manual:

When n is a prime number, the FFTW library first decomposes an n-point problem into three $(n-1)$-point problems using Rader's algorithm [43]. It then uses the Cooley-Tukey decomposition described above to compute the $(n-1)$-point DFTs.

Details of Rader's algorithm: a theorem from number theory:

$$\forall p \in \mathbb{N} \text{ prime} \quad \exists g \in \{1,\dots,p-1\}: \quad \{g^k \mod p: k = 1,\dots,p-1\} = \{1,\dots,p-1\} \ ,$$

- permutation $P_{p,g} : \{1,\dots,p-1\} \mapsto \{1,\dots,p-1\}$, $P_{p,g}(k) = g^k \mod p$,
  reversing permutation $P_k : \{1,\dots,k\} \mapsto \{1,\dots,k\}$, $P_k(i) = k - i + 1$.

For Fourier matrix $\mathbf{F} = (f_{ij})_{i,j=1}^{p}$: $P_{p-1} P_{p,g} (f_{ij})_{i,j=2}^{p} P_{p,g}^T$ is circulant.

Example for $p = 13$:

$g = 2$, permutation: $(2\ 4\ 8\ 3\ 6\ 12\ 11\ 9\ 5\ 10\ 7\ 1)$.

$$\mathbf{F}_{13} \longrightarrow \begin{array}{c|ccccccccccccc}
\omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\
\hline
\omega^0 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 \\
\omega^0 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 \\
\omega^0 & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} \\
\omega^0 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 \\
\omega^0 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 \\
\omega^0 & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} \\
\omega^0 & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} \\
\omega^0 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 \\
\omega^0 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 \\
\omega^0 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 \\
\omega^0 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 \\
\omega^0 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2
\end{array}$$

Then apply fast (FFT based!) algorithms for multiplication with circulant matrices to right lower $(n-1) \times (n-1)$ block of permuted Fourier matrix .

$\triangle$

Asymptotic complexity of `c=fft(y)` for $\mathbf{y} \in \mathbb{C}^n$ = $O(n \log n)$.

6.3 Trigonometric Interpolation: Computational Aspects

Theorem 6.3.1 (Trigonometric interpolant)**.** *Let N even and $z = \mathcal{F}_N y \in \mathbb{C}^N$. The trigonometric interpolant*

$$p_N(x) := \sum_{k=-N/2}^{N/2}{}' z_k e^{2\pi i k x} := \frac{1}{2}\left(z_{-N/2} e^{-2\pi i x N/2} + z_{N/2} e^{2\pi i x N/2}\right) + \sum_{|k|\leq N/2} z_k e^{2\pi i k x}$$

has the property that $p_N(x_\ell) = y_\ell$, where $x_\ell = \ell/N$.

- Fast intepolation by means of FFT: $O(n \log n)$ asymptotic complexity, see Sect. 6.2, provide uniformly distributed nodes $t_k = \dfrac{k}{2n+1}$, $k = 0, \ldots, 2n$

- $(2n+1) \times (2n+1)$ linear system of equations:

$$\sum_{j=0}^{2n} \gamma_j \exp\left(2\pi i \frac{jk}{2n+1}\right) = z_k := \exp\left(2\pi i \frac{nk}{2n+1}\right) y_k \;, \quad k = 0, \dots, 2n \;.$$

$$\Updownarrow$$

$$\overline{\mathbf{F}}_{2n+1}\mathbf{c} = \mathbf{z} \;, \quad \mathbf{c} = (\gamma_0, \dots, \gamma_{2n})^T \quad \overset{\text{Lemma 6.1.2}}{\Rightarrow} \quad \mathbf{c} = \frac{1}{2n+1}\mathbf{F}_{2n}\mathbf{z} \;. \tag{6.3.1}$$

$(2n+1) \times (2n+1)$ (conjugate) Fourier matrix, see (6.1.3)

Code 6.3.2: Efficient computation of coefficient of trigonometric interpolation polynomial (*equidistant nodes*)

```
from numpy import transpose, hstack, arange
from scipy import pi, exp, fft

def trigipequid(y):
    """Efficient computation of coefficients in expansion
      \eqref{eq:trigpreal} for a trigonometric
    interpolation polynomial in equidistant points \Blue{(j/(2n+1), y_j)},
  \Blue{j = 0,...,2n}
    \texttt{y} has to be a row vector of odd length, return values are
      column vectors
    """

    N = y.shape[0]
```

```
    if N%2 != 1:
        raise ValueError("Odd number of points required!")

    n = (N-1.0)/2.0
    z = arange(N)

    # See (6.3.1)
    c = fft(exp(2.0j*pi*(n/N)*z)*y) / (1.0*N)

    # From (??): α_j = 1/2(γ_{n-j} + γ_{n+j}) and
    # β_j = 1/(2i)(γ_{n-j} - γ_{n+j}), j = 1,...,n, α_0 = γ_n

    a = hstack([ c[n], c[n-1::-1]+c[n+1:N] ])
    b = hstack([ -1.0j*(c[n-1::-1]-c[n+1:N]) ])

    return (a,b)

if __name__ == "__main__":
    from numpy import linspace

    y = linspace(0,1,9)

    # Expected Result:
```

```
# a =
#
# 0.50000 - 0.00000i
# -0.12500 + 0.00000i
# -0.12500 + 0.00000i
# -0.12500 - 0.00000i
# -0.12500 + 0.00000i
#
# b =
#
# -0.343435 + 0.000000i
# -0.148969 + 0.000000i
# -0.072169 - 0.000000i
# -0.022041 + 0.000000i

w = trigipequid(y)

print(w)
```

Code 6.3.3: Computation of coefficients of trigonometric interpolation polynomial, general nodes

```
from numpy import hstack, arange, sin, cos, pi, outer
from numpy.linalg import solve

def trigpolycoeff(t, y):
```

```
    """Computes␣expansion␣coefficients␣of␣trigonometric␣polyonomials␣
      \eqref{eq:trigpreal}
␣␣␣␣\texttt{t}:␣row␣vector␣of␣nodes␣\Blue{t_0,...,t_n ∈ [0,1[}
    \texttt{y}: row vector of data \Blue{y_0,...,y_n}
    return values are column vectors of expansion coefficients \Blue{α_j},
       \Blue{β_j}
    """

    N = y.shape[0]

    if N%2 != 1:
        raise ValueError("Odd␣number␣of␣points␣required!")

    n = (N-1.0)/2.0

    M = hstack([ cos(2*pi*outer(t,arange(0,n+1))),
      sin(2*pi*outer(t,arange(1,n+1))) ])
    c = solve(M, y)

    a = c[0:n+1]
    b = c[n+1:]

    return (a, b)
```

```python
if __name__ == "__main__":
    from numpy import linspace

    t = linspace(0, 1, 5)
    y = linspace(0, 5, 5)

    # Expected values:
    # a =
    #
    # 2.5000e+00
    # -1.1150e-16
    # -2.5000e+00
    #
    # b =
    #
    # -2.5000e+00
    # -1.0207e+16

    z = trigpolycoeff(t, y)

    print(z)
```

Example 6.3.4 (Runtime comparison for computation of coefficient of trigonometric interpolation polynomials).

`tic-toc`-timings ▷

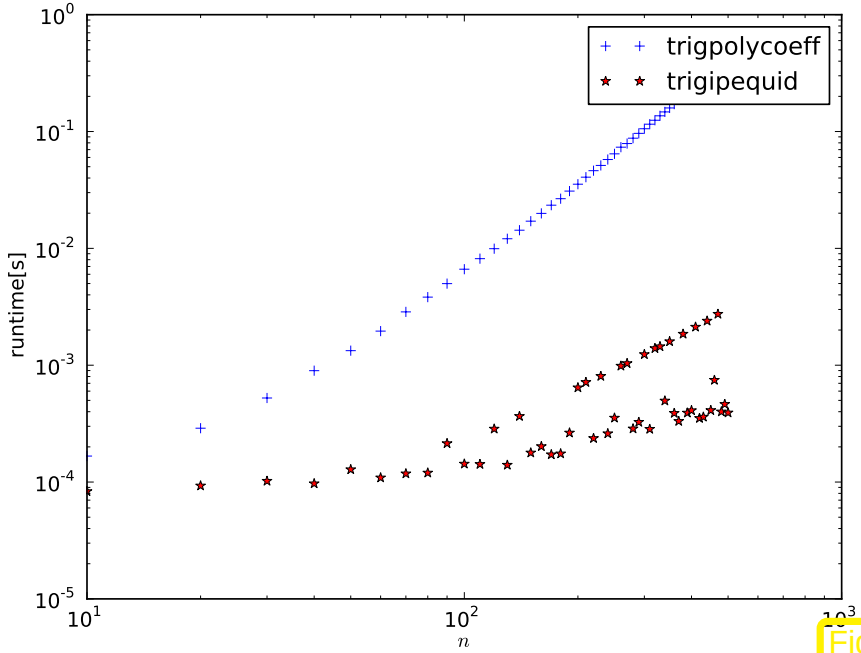


Fig. 57

Code 6.3.5: Runtime comparison

```
from numpy import linspace, pi, exp, cos, array
from matplotlib.pyplot import *
import time

from trigpolycoeff import trigpolycoeff
from trigipequid import trigipequid

def trigipequidtiming():
    """Runtime comparison between efficient (→ Code~\ref{trigipequid}) and direct
      computation
    (→ Code~\ref{trigpolycoeff} of coefficients of trigonoetric interpolation
      polynomial in
    equidistant points.
```

```
"""
Nruns = 3
times = []

for n in xrange(10, 501, 10):
    print(n)

    N = 2*n+1

    t = linspace(0,1-1.0/N,N)
    y = exp(cos(2*pi*t))

    # Infinity for all practical purposes
    t1 = 10.0**10
    t2 = 10.0**10

    for k in xrange(Nruns):
        tic = time.time()
        a, b = trigpolycoeff(t, y)
        toc = time.time() - tic
        t1 = min(t1, toc)

        tic = time.time()
        a, b = trigipequid(y)
        toc = time.time() - tic

        t2 = min(t2, toc)

    times.append( (n, t1, t2) )
```

```
    times = array(times)

    fig = figure()
    ax = fig.gca()

    ax.loglog(times[:,0], times[:,1], "b+", label="trigpolycoeff")
    ax.loglog(times[:,0], times[:,2], "r*", label="trigipequid")

    ax.set_xlabel(r"n")
    ax.set_ylabel(r"runtime[s]")
    ax.legend()

    fig.savefig("../PICTURES/trigipequidtiming.eps")

if __name__ == "__main__":
    trigipequidtiming()
    show()
```


Same observation as in Ex. 6.2.1: massive gain in efficiency through relying on FFT.

◇

Remark 6.3.6 (Efficient evaluation of trigonometric interpolation polynomials).

Task: evaluation of trigonometric polynomial (**??**) at *equidistant* points $\frac{k}{N}$, $N > 2n$. $k = 0, \dots, N-1$.

$$(??) \quad \blacktriangleright \quad q(k/N) = e^{-2\pi i k/N} \sum_{j=0}^{2n} \gamma_j \exp(2\pi i \frac{kj}{N}) \,, \quad k = 0, \dots, N-1 \,.$$

$$\blacktriangleright \quad q(k/N) = e^{-2\pi i k n/N} v_j \quad \text{with} \quad \mathbf{v} = \overline{\mathbf{F}}_N \widetilde{\mathbf{c}} \,, \tag{6.3.2}$$

Fourier matrix, see (6.1.3).

where $\widetilde{\mathbf{c}} \in \mathbb{C}^N$ is obtained by zero padding of $\mathbf{c} := (\gamma_0, \dots, \gamma_{2n})^T$:

$$(\widetilde{\mathbf{c}})_k = \begin{cases} \gamma_j & \text{, for } k = 0, \dots, 2n \,, \\ 0 & \text{, for } k = 2n+1, \dots, N-1 \,. \end{cases}$$

Code 6.3.7: Fast evaluation of trigonometric polynomial at *equidistant* points

```
from numpy import transpose, vstack, hstack, zeros, conj, exp, pi
from scipy import fft

def trigipequidcomp(a, b, N):
    """Efficient␣evaluation␣of␣trigonometric␣polynomial␣at␣equidistant␣
      points
␣␣␣␣column␣vectors␣\texttt{a}␣and␣\texttt{b}␣pass␣coefficients␣\Blue{$\alpha_j$},
```

```
\Blue{β_j} in
 representation \eqref{eq:trigpreal}
 """

 n = a.shape[0] - 1

 if N < 2*n-1:
     raise ValueError("N too small")

 gamma = 0.5*hstack([ a[-1:0:-1] + 1.0j*b[-1::-1], 2*a[0], a[1:] -
   1.0j*b[0:] ])

 # Zero padding
 ch = hstack([ gamma, zeros((N-(2*n+1),)) ])

 # Multiplication with conjugate Fourier matrix
 v = conj(fft(conj(ch)))

 # Undo rescaling
 q = exp(-2.0j*pi*n*arange(N)/(1.0*N)) * v

 return q
```

```python
if __name__ == "__main__":
    from numpy import arange

    a = arange(1,6)
    b = arange(6,10)
    N = 10

    # Expected values:
    #
    # 15.00000 - 0.00000i 21.34656 + 0.00000i -5.94095 - 0.00000i
    # 8.18514 + 0.00000i -3.31230 - 0.00000i 3.00000 + 0.00000i
    # -1.68770 - 0.00000i -2.71300 - 0.00000i 0.94095 + 0.00000i
    # -24.81869 - 0.00000i

    print(a)
    print(b)
    print(N)

    w = trigipequidcomp(a, b, N)

    print(w)
```

Code 6.3.8: *Equidistant* points: fast on the fly evaluation of trigonometric interpolation polynomial

```
from numpy import zeros, fft, sqrt, pi, real, linalg, linspace, sin

def evaliptrig(y,N):
    n = len(y)
    if (n%2) == 0:
        c = fft.ifft(y)
        a = zeros(N, dtype=complex)
        a[:n/2] = c[:n/2]
        a[N-n/2:] = c[n/2:]
        v = fft.fft(a);
        return v
    else: raise TypeError, 'odd length'

f = lambda t: 1./sqrt(1.+0.5*sin(2*pi*t))

vlinf = []; vl2 = []; vn = []
N = 4096; n = 2
while n < 1+2**7:
    t = linspace(0,1,n+1)
    y = f(t[:-1])
    v = real(evaliptrig(y,N))
    t = linspace(0,1,N+1)
    fv = f(t)
    d = abs(v-fv[:-1]); linf = d.max()
```

```
    l2 = linalg.norm(d)/sqrt(N)
    vlinf += [linf]; vl2 += [l2]; vn += [n]
    n *= 2

from pylab import semilogy, show
semilogy(vn,vl2,'-+'); show()
```

compare with

Code 6.3.9: *Equidistant* points: fast on the fly evaluation of trigonometric interpolation polynomial

```
from numpy import zeros, fft, sqrt, pi, real, linalg, linspace, sin

def evaliptrig(y,N):
    n = len(y)
    if (n%2) == 0:
        c = fft.fft(y)*1./n
        a = zeros(N, dtype=complex)
        a[:n/2] = c[:n/2]
        a[N-n/2:] = c[n/2:]
        v = fft.ifft(a)*N;
        return v
    else: raise TypeError, 'odd length'

f = lambda t: 1./sqrt(1.+0.5*sin(2*pi*t))
```

```
vlinf = []; vl2 = []; vn = []
N = 4096; n = 2
while n < 1+2**7:
    t = linspace(0,1,n+1)
    y = f(t[:-1])
    v = real(evaliptrig(y,N))
    t = linspace(0,1,N+1)
    fv = f(t)
    d = abs(v-fv[:-1]); linf = d.max()
    l2 = linalg.norm(d)/sqrt(N)
    vlinf += [linf]; vl2 += [l2]; vn += [n]
    n *= 2

from pylab import semilogy, show
semilogy(vn,vl2,'-+'); show()
```

△

6.4 Trigonometric Interpolation: Error Estimates

Linear trigonometric interpolation operator: for 1-periodic $f : \mathbb{R} \mapsto \mathbb{C}$

$$\mathsf{T}_n(f) := p \in \mathcal{P}_{n-1}^T \quad \text{with} \quad p(\tfrac{j}{n}) = y_j \,, \quad j = 0, \dots, n-1 \,.$$

Example 6.4.1 (Trigonometric interpolation).

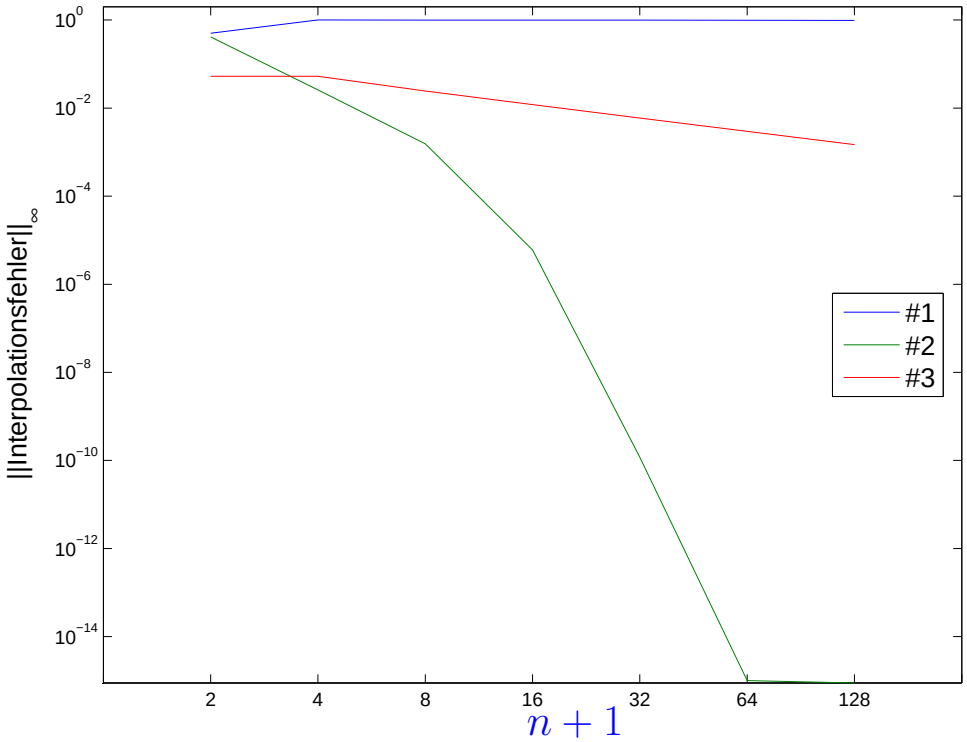
#1 step function: $f(t) = 0$ for $|t - \frac{1}{2}| > \frac{1}{4}$, $f(t) = 1$ for $|t - \frac{1}{2}| \leq \frac{1}{4}$

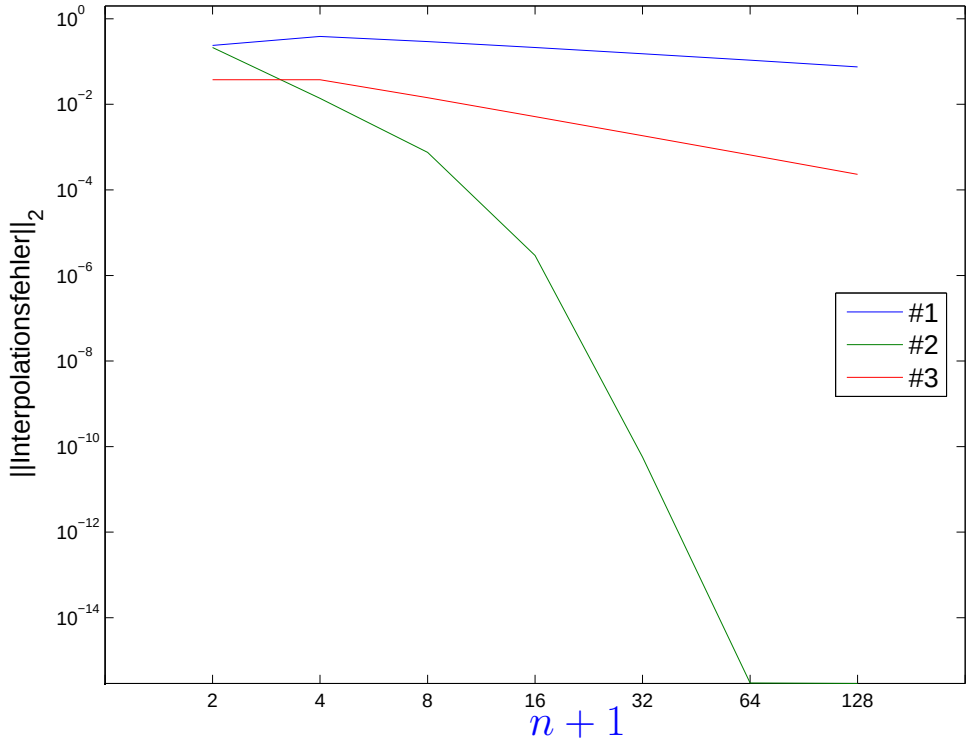
#2 smooth periodic function: $f(t) = \dfrac{1}{\sqrt{1 + \frac{1}{2}\sin(2\pi t)}}$.

#3 hat function: $f(t) = |t - \frac{1}{2}|$

Note: computation of the norms of the interpolation error:

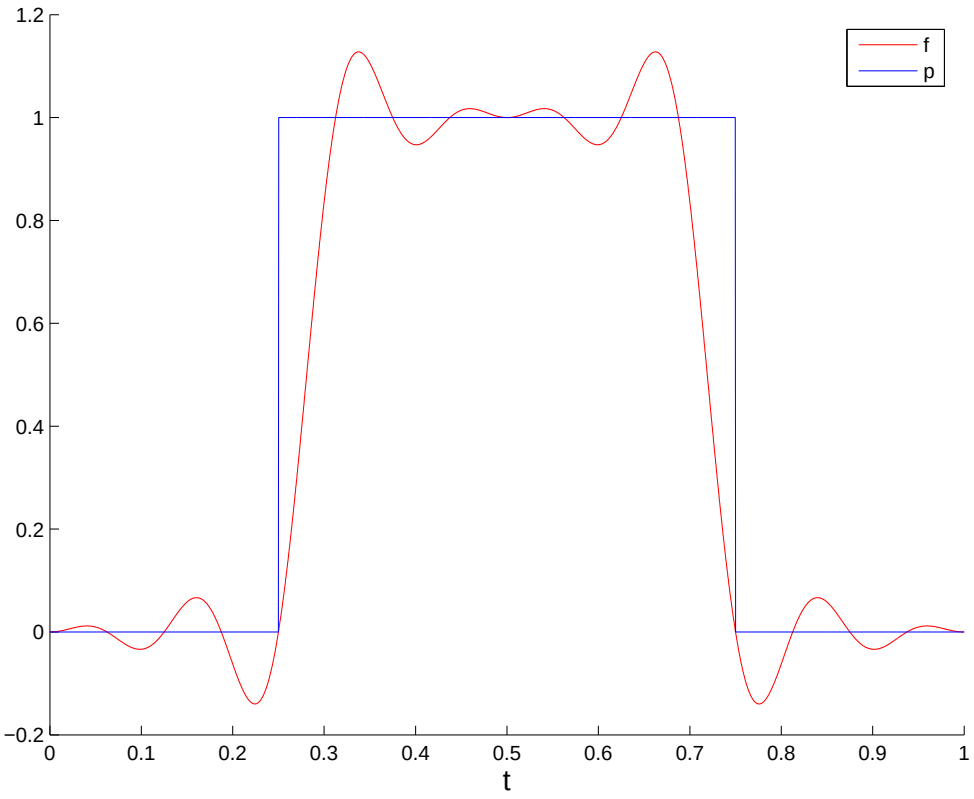
```
t = linspace(0,1,n+1);  y = f(t[:-1])
v = real(evaliptrig(y,N))
t = linspace(0,1,N+1);  fv = f(t)
d = abs(v-fv[:-1]); linf = d.max()
l2 = linalg.norm(d)/sqrt(N)
```

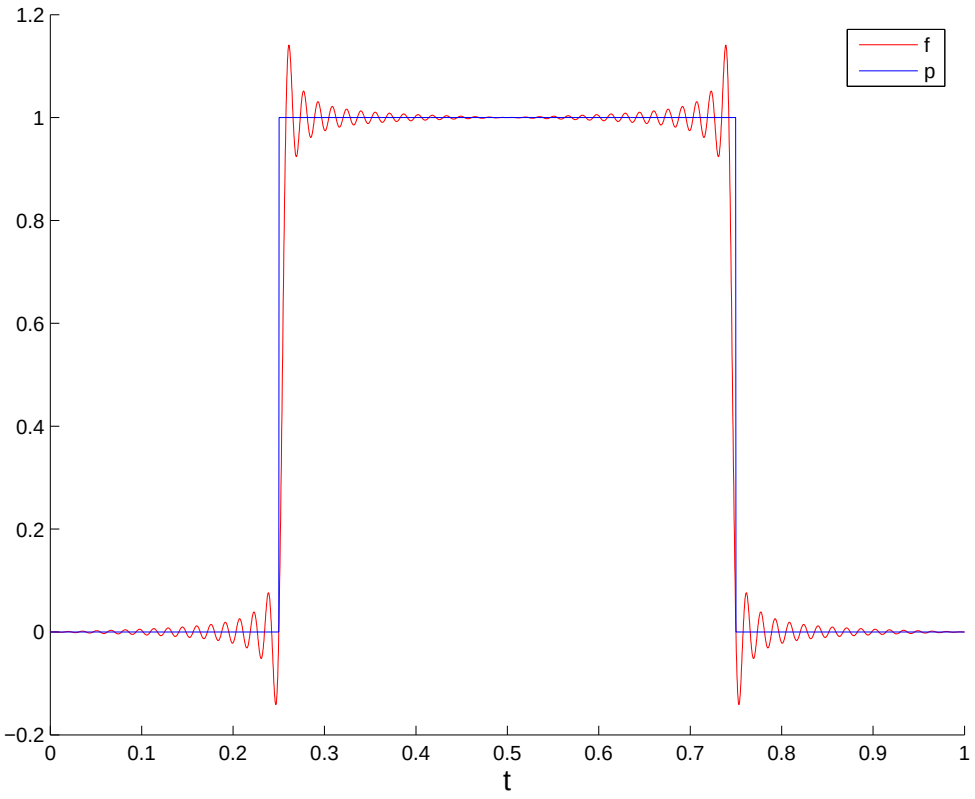




Note: Cases #1, #3: algebraic convergence

Case #2: exponential convergence





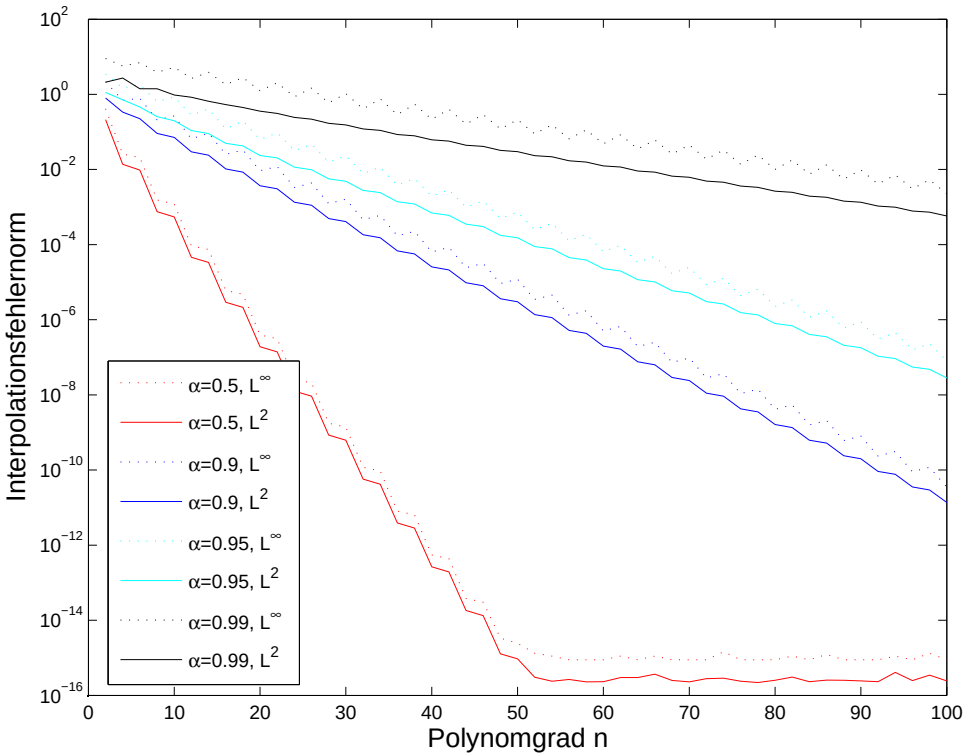
$n = 16$ $n = 128$

➢ Gibbs'phenomenon: ripples near discontinuities ◇

Example 6.4.2 (Trigonometric interpolation of analytic functions).

$$f(t) = \frac{1}{\sqrt{1 - \alpha \sin(2\pi t)}} \quad \text{auf } I = [0,1] \ .$$

approximation of the norms: "oversampling" in 4096 points



➢ Note: exponential convergence in degree n, quicker for smaller α ◇

Analysis uses: trigonometric polynomials = partial sums of Fourier series

Theorem 6.4.1 (Error of DFT; aliasing formula)**.** *If* $\sum_{k\in\mathbb{Z}} \hat{f}(k)$ *absolut converges, then*

$$\hat{f}_N(k) - \hat{f}(k) = \sum_{\substack{j\in\mathbb{Z}\\ j\neq 0}} \hat{f}(k + jN)$$

Corollary 6.4.2. *Let $f \in C^p$ with $p \geq 2$ and 1-periodic. Then it holds:*

$$\hat{f}_N(k) - \hat{f}(k) = O(N^{-p}) \quad \text{for} \quad |k| \leq \frac{N}{2}$$

$$\text{for:} \quad h = \frac{1}{N}, \quad h \sum_{j=0}^{N-1} f(x_j) - \int_0^1 f(x)dx = O(h^p).$$

Remark 6.4.3. $\hat{f}_N(k)$ is N-periodic, while $\hat{f}(k) \to 0$ quickly

$\hat{f}_N(k)$ is a bad approximation of $\hat{f}(k)$ for k large ($k \approx N$), but for $|k| \leq \frac{N}{2}$ it is good enough.

Theorem 6.4.3 (Error of trigonometric interpolantion)**.** *If f is 1-periodic and $\sum_{k\in\mathbb{Z}} \hat{f}(k)$ absolut converges, then*

$$|p_N(x) - f(x)| \leq 2 \sum_{|k|\geq N/2}' |\hat{f}(k)| \quad \forall x \in \mathbb{R}$$

Corollary 6.4.4 (Sampling-Theorem)**.** *Let f 1-periodic with maximum frequency M: $\hat{f}(k) = 0$ for all $|k| > M$. Then $p_N(x) = f(x)$ for all x, if $N > 2M$*

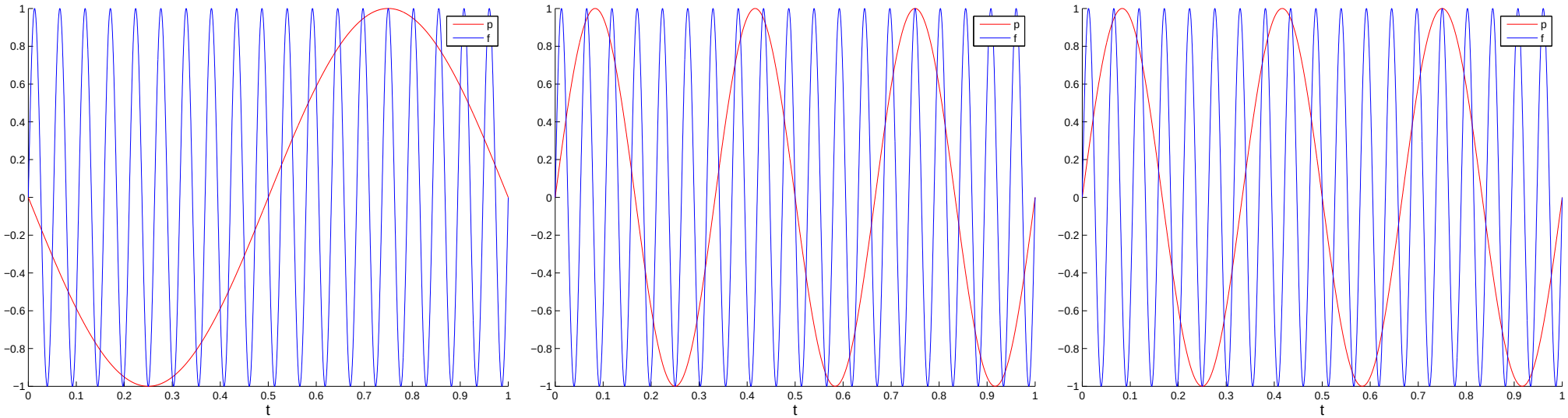
data compression

Remark 6.4.4 (Aliasing).

$$\mathcal{T} = \{\tfrac{j}{n}\}_{j=0}^{n-1} \quad \blacktriangleright \quad e^{2\pi i N t}_{|\mathcal{T}} = e^{2\pi i (N-n) t}_{|\mathcal{T}} \quad \blacktriangleright \quad \mathsf{T}_n(e^{2\pi i N \cdot}) = \mathsf{T}_n(e^{2\pi i (N \bmod n) \cdot}) \ .$$

hence: The trigonometric interpolation of $t \longrightarrow e^{2\pi i N t}$ and $t \longrightarrow e^{2\pi i (N \bmod n) t}$ of degree n give the same trigonometric interpolation polynomial ! $\longrightarrow$ Aliasing

Example for $f(t) = \sin(2\pi \cdot 19 t)$ $\quad\succ\quad$ $N = 38$:



$n = 4$ $\qquad$ $n = 8$ $\qquad$ $n = 16$

△

[Linearity of trigonometric interpolation]

For $f \in C([0,1])$ 1-periodic, $n = 2m$:

$$f(t) = \sum_{j=-\infty}^{\infty} \widehat{f}(j) e^{2\pi i j t} \quad \Rightarrow \quad \mathsf{T}_n(f)(t) = \sum_{j=-m+1}^{m} \gamma_j e^{2\pi i j t} \,, \quad \gamma_j = \sum_{l=-\infty}^{\infty} \widehat{f}(j+ln) \,.$$

► Fourier coefficients of the error function $e = f - \mathsf{T}_n(f)$:

$$\widehat{e}(j) = \begin{cases} -\sum\limits_{|l|=1}^{\infty} \widehat{f}(j+ln) & \text{, if } -m+1 \le j \le m \,, \\ \widehat{f}(j) & \text{, if } j \le -m \vee j > m \,. \end{cases}$$

$$\overset{(6.0.1)}{\Rightarrow} \quad \|f - \mathsf{T}_n(f)\|^2_{L^2(]0,1[)} \le \Big| \sum_{|l|=1}^{\infty} \widehat{f}(j+ln) \Big|^2 + \sum_{|j| \ge m} |\widehat{f}(j)|^2 \,. \tag{6.4.1}$$

► may be estimated, if we know the decay of the Fourier coefficients $\widehat{f}(j)$ $\Leftrightarrow$ smoothness of f, see (6.0.2)

Theorem 6.4.5 (Error estimate for trigonometric interpolation). *For* $k \in \mathbb{N}$*:*

$$f^{(k)} \in L^2(]0,1[) \quad \Rightarrow \quad \|f - \mathsf{T}_n f\|_{L^2(]0,1[)} \leq \sqrt{1+c_k}\, n^{-k} \left\| f^{(k)} \right\|_{L^2(]0,1[)} ,$$

with $c_k = 2\sum_{l=1}^{\infty}(2l-1)^{-2k}$.

6.5 DFT and Chebychev Interpolation

Let $p \in \mathcal{P}_n$ be the Chebychev interpolation polynom on $[-1,1]$ of $f : [-1,1] \mapsto \mathbb{C}$ ($\longrightarrow$ section 5.4)

$$p(t_k) = f(t_k) \quad \text{for Chebychev nodes, see (5.4.5),} \quad t_k := \cos\left(\frac{2k+1}{2(n+1)}\pi\right), \; k = 0, \dots, n .$$

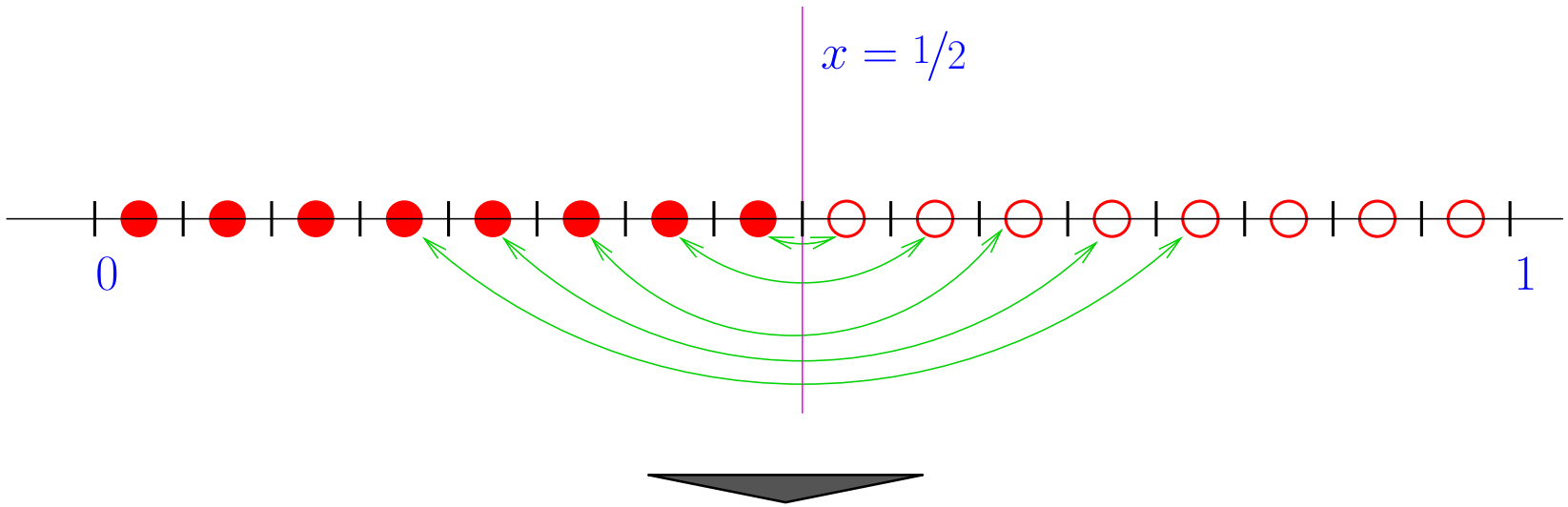
Define now the corresponding functions g, q:

$$\left.\begin{array}{lcl} f : [-1,1] \mapsto \mathbb{C} & \leftrightarrow & g(s) := f(\cos 2\pi s) , \\ p : [-1,1] \mapsto \mathbb{C} & \leftrightarrow & q(s) := p(\cos 2\pi s) , \end{array}\right\} \quad \text{1-periodic, symmetric wrt. } 0 \text{ and } \frac{1}{2}$$

Hence:

$$p(t_k) = f(t_k) \quad \Leftrightarrow \quad q\left(\frac{2k+1}{4(n+1)}\right) = g\left(\frac{2k+1}{4(n+1)}\right) . \tag{6.5.1}$$

and with a translation $\tilde{s} = s + \frac{1}{4(n+1)}$ we have:



$$\widetilde{g}(s) := g(s + \frac{1}{4(n+1)}) \ , \qquad \widetilde{q}(s) := q(s + \frac{1}{4(n+1)}) \quad \Rightarrow \quad \widetilde{q}\left(\frac{k}{2(n+1)}\right) = \widetilde{g}\left(\frac{k}{2n+2}\right) \ , \quad k = 0, \ldots, 2n \ .$$

! We show: $q(s)$ is trigonometric polynomial: $\boxed{q \in \mathcal{P}_{2n}^T}$, namely the trigonometric interpolation polynomial of $\widetilde{g}$. Indeed, since p is the Chebychev interpolation polynomial:

$$q(s) = p(\cos 2\pi s) = \sum_{j=0}^{n} \gamma_j T_j(\cos 2\pi s) = \sum_{j=0}^{n} \gamma_j \cos 2\pi s = \gamma_0 + \sum_{j=-n, j\neq 0}^{n} \tfrac{1}{2}\gamma_{|j|} e^{-2\pi i j s} =$$

$$\gamma_0 + \sum_{j=-n, j\neq 0}^{n} \tfrac{1}{2}\gamma_{|j|} e^{2\pi i \frac{j}{4(n+1)}} \cdot e^{-2\pi i j \left(s + \frac{1}{4(n+1)}\right)} = \sum_{j=-n}^{n} \tilde{\gamma}_j e^{-2\pi i j \tilde{s}}$$

with $\quad \widetilde{\gamma}_j := \begin{cases} \frac{1}{2}\exp\left(2\pi i \frac{j}{4(n+1)}\right)\gamma_{|j|} & \text{, if } j \neq 0 \text{ ,} \\ \gamma_0 & \text{, for } j = 0 \text{ .} \end{cases}$

Hence

$$\tilde{q}(s) = \sum_{j=-n}^{n} \tilde{\gamma}_j e^{-2\pi i j s}$$

and $\tilde{q}\left(\dfrac{k}{2(n+1)}\right) = \tilde{g}\left(\dfrac{k}{2(n+1)}\right)$, for all $k = 0, \ldots, 2n$.

that means that $\tilde{q}$ is the trigonometric interpolation polynomial of $\tilde{g}$.

Hence $\|f - p\|_{L^\infty([-1,1])} = \|\widetilde{g} - \widetilde{q}\|_{L^\infty([0,1])}$

and the error estimates from the trigonometric interpolation directly transfers here.

Code 6.5.1: Efficient Chebyshev interpolation

```
from numpy import exp, pi, real, hstack, arange
from scipy import ifft

def chebexp(y):
    """Efficiently compute coefficients \Blue{α_j} in the Chebychev expansion
    \Blue{p = Σ_{j=0}^{n} α_j T_j} of \Blue{p ∈ P_n} based on values \Blue{y_k},
    \Blue{k = 0,...,n}, in Chebychev nodes \Blue{t_k}, \Blue{k = 0,...,n}. These values are
    passed in the row vector \texttt{y}.
    """

    # degree of polynomial
    n = y.shape[0] -1

    # create vector z by wrapping and componentwise scaling

    # r.h.s. vector
    t = arange(0, 2*n+2)
    z = exp(-pi*1.0j*n/(n+1.0)*t) * hstack([y,y[::-1]])

    # Solve linear system (??) with effort O(n log n)
```

```
    c = ifft(z)

    # recover β_j, see (??)
    t = arange(-n, n+2)
    b = real(exp(0.5j*pi/(n+1.0)*t) * c)

    # recover α_j, see (??)
    a = hstack([ b[n], 2*b[n+1:2*n+1] ])

    return a

if __name__ == "__main__":
    from numpy import array

    # Test with arbitrary values
    y = array([1, 2, 3, 3.5, 4, 6.5, 6.7, 8, 9])

    # Expected values:
    # 4.85556 -3.66200 0.23380 -0.25019 -0.15958 -0.36335 0.18889 0.16546 -0.27329

    w = chebexp(y)

    print(w)
```

6.6 Essential Skills Learned in Chapter 6

You should know:

- the idea behind the trigonometric approximation;
- what is the Gibbs phenomenon;
- the dicrete Fourier transform and its use for the trigonometric interpolation;
- the idea and the importance of the fast Fourier transform;
- how to use the fast Fourier transform for the efficient trigonometrical interpolation;
- the error behavior for the trionomteric interpolation;
- the aliasing formula and the Sampling-Theorem;
- how to use the fast Fourier transform for the efficient Chebychev interpolation.