

$\kappa(\mathbf{B}^{-1}\mathbf{A})$  "small"  $\rightarrow$   $\mathbf{B}^{-1}$ -energy norm of residual  $\approx$   $\mathbf{A}$ -norm of error!  
 $(\mathbf{r}_l \cdot \mathbf{B}^{-1}\mathbf{r}_l = \mathbf{q}^T \mathbf{r}$  in Algorithm (5.4.2))

MATLAB-function: `[x,flag,relr,it,rv] = pcg(A,b,tol,maxit,B,[],x0);`  
 (A, B may be handles to functions providing  $\mathbf{Ax}$  and  $\mathbf{B}^{-1}\mathbf{x}$ , resp.)

Remark 5.4.8 (Termination criterion in MATLAB-pcg).

Implementation (skeleton) of MATLAB built-in pcg:

MATLAB PCG algorithm

```
function x = pcg(Afun,b,tol,maxit,Binvfun,x0)
x = x0; r = b - feval(Afun,x); rho = 1;
for i = 1 : maxit
    y = feval(Binvfun,r);
    rho1 = rho; rho = r' * y;
    if (i == 1)
        p = y;
    else
        beta = rho / rho1;
        p = y + beta * p;
    end
    q = feval(Afun,p);
    alpha = rho / (p' * q);
    x = x + alpha * p;
    if (norm(b - evalf(Afun,x)) <= tol*b*norm(b)), return; end
    r = r - alpha * q;
end
```

Dubious termination criterion !

Summary:

Numerical Methods 401-0654

Advantages of Krylov methods vs. direct elimination (, IF they converge at all/sufficiently fast).

- They require system matrix  $\mathbf{A}$  in procedural form  $y = \text{evalA}(x) \leftrightarrow y = \mathbf{Ax}$  only.
- They can perfectly exploit sparsity of system matrix.
- They can cash in on low accuracy requirements (, IF viable termination criterion available).
- They can benefit from a good initial guess.

Numerical Methods 401-0654

## 5.5 Essential Skills Learned in Chapter 5

You should know:

- the relation between a linear system of equations with a s.p.d. matrix and a quadratic minimization problem
- how the steepest descent method works and its convergence properties
- the idea behind the conjugate gradient method and its convergence properties
- how to use the Matlab-function `pcg`
- the importance of the preconditioner

V. Gradinaru D-ITET, D-MATL

V. Gradinaru D-ITET, D-MATL

5.4 p. 281

5.5 p. 28

Numerical Methods 401-0654

Numerical Methods 401-0654

# 6

## Filtering Algorithms

Perspective of **signal processing**:

vector  $\mathbf{x} \in \mathbb{R}^n \leftrightarrow$  finite discrete (= sampled) signal.

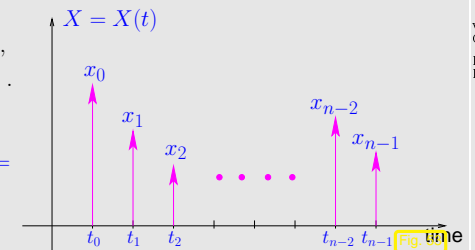
$X = X(t) \hat{=}$  time-continuous signal,  $0 \leq t \leq T$ ,

"sampling":  $x_j = X(j\Delta t)$ ,  $j = 0, \dots, n-1$ ,  
 $n \in \mathbb{N}$ ,  $n\Delta t \leq T$ .

$\Delta t > 0 \hat{=}$  time between samples.

Sampled values arranged in a vector  $\mathbf{x} = (x_0, \dots, x_{n-1})^T \in \mathbb{R}^n$ .

Note: vector indices  $0, \dots, n-1$ !  
 ("C-style indexing").



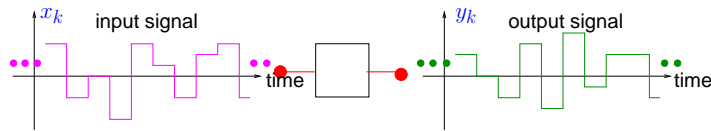
△

5.4 p. 282

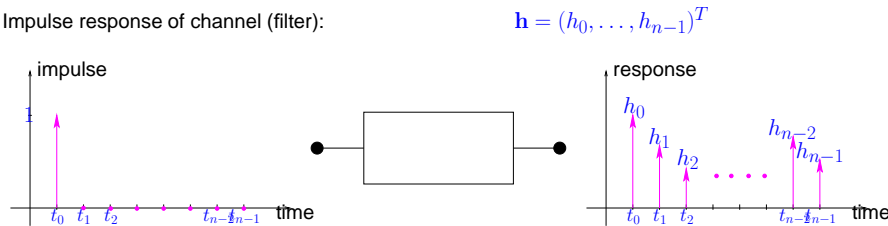
6.1 p. 28

# 6.1 Discrete convolutions

Example 6.1.1 (Discrete finite linear time-invariant causal channel (filter)).



Impulse response of channel (filter):



**Impulse response** = output when filter is fed with a single impulse of strength one, corresponding to input  $\mathbf{e}_1$  (first unit vector).

We study a *finite linear time-invariant causal channel (filter)*:  
(widely used model for digital communication channels, e.g. in wireless communication theory)

**finite**: impulse response of finite duration  $\Rightarrow$  it can be described by a vector  $\mathbf{h}$  of finite length  $n$ .

**time-invariant**: when input is shifted in time, output is shifted by the same amount of time.

**linear**: input  $\mapsto$  output-map is linear

$$\text{output}(\mu \cdot \text{signal 1} + \lambda \cdot \text{signal 2}) = \mu \cdot \text{output}(\text{signal 1}) + \lambda \cdot \text{output}(\text{signal 2}) .$$

**causal** (or physical, or nonanticipative): output depends only on past and present inputs, not on the future.

$\blacktriangleright$  The output for finite length input  $\mathbf{x} = (x_0, \dots, x_{n-1})^T \in \mathbb{R}^n$  is  
a superposition of  $x_j$ -weighted  $j\Delta t$ -shifted impulse responses

channel is causal!

$$y_k = \sum_{j=0}^{n-1} h_{k-j} x_j, \quad k = 0, \dots, 2n-2 \quad (h_j := 0 \text{ for } j < 0 \text{ and } j \geq n) . \quad (6.1.1)$$

$\mathbf{x} = (x_0, \dots, x_{n-1})^T \in \mathbb{R}^n \hat{=}$  input signal  $\mapsto \mathbf{y} = (y_0, \dots, y_{2n-2})^T \in \mathbb{R}^{2n-1} \hat{=}$  output signal.

Matrix notation of (6.1.1):

$$\begin{pmatrix} y_0 \\ \vdots \\ y_{2n-2} \end{pmatrix} = \begin{pmatrix} h_0 & 0 & \dots & 0 \\ h_1 & & & \\ \vdots & \ddots & \ddots & \\ h_{n-1} & & & 0 \\ 0 & & & h_1 & h_0 \\ \vdots & & & \vdots & \vdots \\ 0 & & & 0 & h_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} . \quad (6.1.2)$$

Example 6.1.2 (Multiplication of polynomials).

$$p(z) = \sum_{k=0}^{n-1} a_k z^k, \quad q(z) = \sum_{k=0}^{n-1} b_k z^k \quad \blacktriangleright \quad (pq)(z) = \sum_{k=0}^{2n-2} \underbrace{\left( \sum_{j=0}^k a_j b_{k-j} \right)}_{=: c_k} z^k \quad (6.1.3)$$

$\blacktriangleright$  coefficients of product polynomial by **discrete convolution** of coefficients of polynomial factors!

Both in (6.1.1) and (6.1.3) we recognize the same pattern of a particular *bi-linear* combination of

- discrete signals in Ex. 6.1.1,
- polynomial coefficient sequences in Ex. 6.1.2.

**Definition 6.1.1** (Discrete convolution).

Given  $\mathbf{x} = (x_0, \dots, x_{n-1})^T \in \mathbb{K}^n$ ,  $\mathbf{h} = (h_0, \dots, h_{n-1})^T \in \mathbb{K}^n$  their *discrete convolution* (ger.: *diskrete Faltung*) is the vector  $\mathbf{y} \in \mathbb{K}^{2n-1}$  with components

$$y_k = \sum_{j=0}^{n-1} h_{k-j} x_j, \quad k = 0, \dots, 2n-2 \quad (h_j := 0 \text{ for } j < 0). \quad (6.1.4)$$

Notation for discrete convolution (6.1.4):  $\mathbf{y} = \mathbf{h} * \mathbf{x}$ .

Defining  $x_j := 0$  for  $j < 0$ , we find that *discrete convolution is commutative*:

$$y_k = \sum_{j=0}^{n-1} h_{k-j} x_j = \sum_{l=0}^{n-1} h_l x_{k-l}, \quad k = 0, \dots, 2n-2, \quad (\text{that is, } \mathbf{h} * \mathbf{x} = \mathbf{x} * \mathbf{h}),$$

obtained by index transformation  $l \leftarrow k - j$ .

**Remark 6.1.3** (Convolution of sequences).

The notion of a discrete convolution of Def. 6.1.1 naturally extends to sequences  $\mathbb{N}_0 \mapsto \mathbb{K}$ : the

(discrete) convolution of two sequences  $(x_j)_{j \in \mathbb{N}_0}$ ,  $(y_j)_{j \in \mathbb{N}_0}$  is the sequence  $(z_j)_{j \in \mathbb{N}_0}$  defined by

$$z_k := \sum_{j=0}^k x_{k-j} y_j = \sum_{j=0}^k x_j y_{k-j}, \quad k \in \mathbb{N}_0.$$

**Example 6.1.4** (Linear filtering of periodic signals).

*n*-periodic signal ( $n \in \mathbb{N}$ ) = sequence  $(x_j)_{j \in \mathbb{Z}}$  with  $x_{j+n} = x_j \quad \forall j \in \mathbb{Z}$

$n$ -periodic signal  $(x_j)_{j \in \mathbb{Z}}$  fixed by  $x_0, \dots, x_{n-1} \leftrightarrow$  vector  $\mathbf{x} = (x_0, \dots, x_{n-1})^T \in \mathbb{R}^n$ .

Whenever the input signal of a time-invariant filter is *n*-periodic, so will be the output signal. Thus, in the *n*-periodic setting, a causal *linear* time-invariant filter will give rise to a *linear* mapping  $\mathbb{R}^n \mapsto \mathbb{R}^n$  according to

$$y_k = \sum_{j=0}^{n-1} h_{k-j} x_j \quad \text{for some } h_0, \dots, h_{n-1} \in \mathbb{R}. \quad (6.1.5)$$

Note:  $h_0, \dots, h_{n-1}$  is no longer the impulse response of the filter.

Matrix notation:

$$\begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix} = \underbrace{\begin{pmatrix} h_0 & h_{n-1} & h_{n-2} & \cdots & \cdots & h_1 \\ h_1 & h_0 & h_{n-1} & \cdots & \cdots & \vdots \\ h_2 & h_1 & h_0 & \cdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & \cdots & \cdots & \cdots & h_1 & h_0 \end{pmatrix}}_{=\mathbf{H}} \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix}. \quad (6.1.6)$$

$(\mathbf{H})_{ij} = h_{i-j}, 1 \leq i, j \leq n$ , with  $h_j := h_{j+n}$  for  $1-n \leq j < 0$ .

**Definition 6.1.2** (Discrete periodic convolution).

The *discrete periodic convolution* of two *n*-periodic sequences  $(x_k)_{k \in \mathbb{Z}}$ ,  $(y_k)_{k \in \mathbb{Z}}$  yields the *n*-periodic sequence

$$(z_k) := (x_k) *_{n} (y_k), \quad z_k := \sum_{j=0}^{n-1} x_{k-j} y_j = \sum_{j=0}^{n-1} y_{k-j} x_j, \quad k \in \mathbb{Z}.$$

notation for discrete periodic convolution:  $(x_k) *_{n} (y_k)$

Since *n*-periodic sequences can be identified with vectors in  $\mathbb{K}^n$  (see above), we can also introduce the discrete periodic convolution of vectors:

Def. 6.1.2  $\triangleright$  discrete periodic convolution of vectors:  $\mathbf{z} = \mathbf{x} *_{n} \mathbf{y} \in \mathbb{K}^n, \quad \mathbf{x}, \mathbf{y} \in \mathbb{K}^n$ .

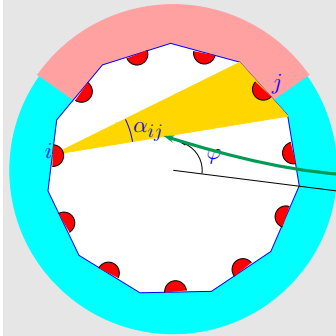
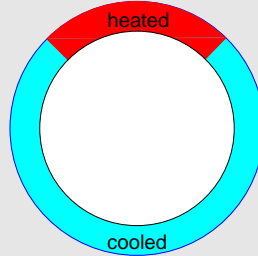
**Example 6.1.5** (Radiative heat transfer).

Beyond signal processing discrete periodic convolutions occur in mathematical models:

An engineering problem:

- cylindrical pipe,
- heated on part  $\Gamma_H$  of its perimeter ( $\rightarrow$  prescribed heat flux),
- cooled on remaining perimeter  $\Gamma_K$  ( $\rightarrow$  constant heat flux).

Task: compute local heat fluxes.



**Modeling** (discretization):

- approximation by regular  $n$ -polygon, edges  $\Gamma_j$ ,
- isotropic radiation of each edge  $\Gamma_j$  (power  $I_j$ ),

radiative heat flow  $\Gamma_j \rightarrow \Gamma_i: P_{ji} := \frac{\alpha_{ij}}{\pi} I_j$ ,

opening angle:  $\alpha_{ij} = \pi \gamma_{|i-j|}, 1 \leq i, j \leq n$ ,

$$\text{power balance: } \underbrace{\sum_{i=1, i \neq j}^n P_{ji}}_{=I_j} - \sum_{i=1, i \neq j}^n P_{ij} = Q_j. \quad (6.1.7)$$

$Q_j \hat{=}$  heat flux through  $\Gamma_j$ , satisfies

$$Q_j := \int_{\frac{2\pi}{n}(j-1)}^{\frac{2\pi}{n}j} q(\varphi) d\varphi, \quad q(\varphi) := \begin{cases} \text{local heating} & , \text{ if } \varphi \in \Gamma_H, \\ -\frac{1}{|\Gamma_K|} \int_{\Gamma_H} q(\varphi) d\varphi & (\text{const.}), \text{ if } \varphi \in \Gamma_K. \end{cases}$$

$$(6.1.7) \Rightarrow \text{LSE: } I_j - \sum_{i=1, i \neq j}^n \frac{\alpha_{ij}}{\pi} I_i = Q_j, \quad j = 1, \dots, n.$$

$$n = 8: \begin{pmatrix} 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 \\ -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 \\ -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 \\ -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 \\ -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 \\ -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 \\ -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 \\ -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ I_5 \\ I_6 \\ I_7 \\ I_8 \end{pmatrix} = \begin{pmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \\ Q_5 \\ Q_6 \\ Q_7 \\ Q_8 \end{pmatrix}. \quad (6.1.8)$$

This is a linear system of equations with symmetric, singular, and (by Thm. 4.1.3,  $\sum \gamma_i \leq 1$ ) positive semidefinite ( $\rightarrow$  Def. ??) system matrix.

Note: matrices from (6.1.6) and (6.1.8) have the same structure !

Observe: LSE from (6.1.8) can be written by means of the discrete periodic convolution ( $\rightarrow$  Def. 6.1.2) of vectors  $\mathbf{y} = (1, -\gamma_1, -\gamma_2, -\gamma_3, -\gamma_4, -\gamma_3, -\gamma_2, -\gamma_1)$ ,  $\mathbf{x} = (I_1, \dots, I_8)$

$$(6.1.8) \leftrightarrow \mathbf{y} * \mathbf{x} = (Q_1, \dots, Q_8)^T.$$

**Definition 6.1.3** (Circulant matrix).

A matrix  $\mathbf{C} = (c_{ij})_{i,j=1}^n \in \mathbb{K}^{n,n}$  is **circulant** (ger.: zirkulant)

$$:\Leftrightarrow \exists (u_k)_{k \in \mathbb{Z}} \text{ } n\text{-periodic sequence: } c_{ij} = u_{i-j}, 1 \leq i, j \leq n.$$

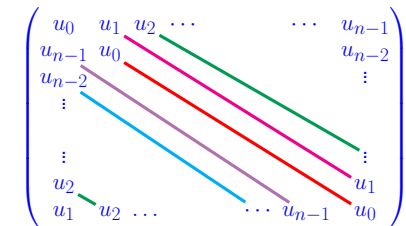
• Circulant matrix has constant (main, sub- and super-) diagonals (for which indices  $i - j = \text{const.}$ ).

• columns/rows arise by *cyclic permutation* from first column/row.

Similar to the case of banded matrices ( $\rightarrow$  Sect. ??):

"information content" of circulant matrix  $\mathbf{C} \in \mathbb{K}^{n,n} = n$  numbers  $\in \mathbb{K}$ . (obviously, one vector  $\mathbf{u} \in \mathbb{K}^n$  enough to define circulant matrix  $\mathbf{C} \in \mathbb{K}^{n,n}$ )

Structure of circulant matrix



**Remark 6.1.6** (Reduction to periodic convolution).

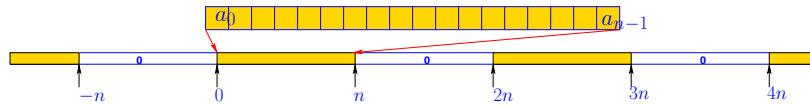
Discrete convolution ( $\rightarrow$  Def. 6.1.1) of  $\mathbf{a} = (a_0, \dots, a_{n-1})^T \in \mathbb{K}^n$ ,  $\mathbf{b} = (b_0, \dots, b_{n-1})^T \in \mathbb{K}^n$ :

$$(\mathbf{a} * \mathbf{b})_k = \sum_{j=0}^k a_j b_{k-j}, \quad k = 0, \dots, 2n - 2.$$

Expand  $a_0, \dots, a_{n-1}$  and  $b_0, \dots, b_{n-1}$  to  $2n - 1$ -periodic sequences by **zero padding**:

$$x_k := \begin{cases} a_k & , \text{ if } 0 \leq k < n, \\ 0 & , \text{ if } n \leq k < 2n - 1 \end{cases}, \quad y_k := \begin{cases} b_k & , \text{ if } 0 \leq k < n, \\ 0 & , \text{ if } n \leq k < 2n - 1, \end{cases} \quad (6.1.9)$$

and periodic extension:  $x_k = x_{2n-1+k}$ ,  $y_k = y_{2n-1+k}$  for all  $k \in \mathbb{Z}$ .



$$\blacktriangleright (\mathbf{a} * \mathbf{b})_k = (\mathbf{x} *_{2n-1} \mathbf{y})_k, \quad k = 0, \dots, 2n-2. \quad (6.1.10)$$

Matrix view of reduction to periodic convolution, cf. (6.1.2)

$$\begin{pmatrix} y_0 \\ \vdots \\ y_{2n-2} \end{pmatrix} = \underbrace{\begin{pmatrix} h_0 & 0 & \dots & 0 & h_{n-1} & h_0 \\ h_1 & & & & 0 & \\ & \ddots & & & & \\ & & 0 & & & \\ h_{n-1} & h_1 & h_0 & 0 & 0 & h_{n-1} \\ 0 & & h_1 & h_0 & & \\ & & & & & \\ & & & & & \\ 0 & & & & 0 & h_{n-1} \\ & & & & h_1 & h_0 \end{pmatrix}}_{\mathbf{a} (2n-1) \times (2n-1) \text{ circulant matrix!}} \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

△

## 6.2 Discrete Fourier Transform (DFT)

Example 6.2.1 (Eigenvectors of circulant matrices).

Code 6.2.2: Eigenvectors of random circulant matrices

```
function circeig
n = 8;
C = gallery('circul',rand(n,1)); [V1,D1] = eig(C);

for j=1:n
figure; bar(1:n,[real(V1(:,j)),imag(V1(:,j))],1,'grouped');
title(sprintf('Circulant_matrix_1_eigenvector_%d',j));
xlabel('\bf_vector_component_index','fontsize',14);
ylabel('\bf_vector_component_value','fontsize',14);
legend('real_part','imaginary_part','location','southwest');
print('-depsc2',sprintf('.../PICTURES/circeig1ev%d.eps',j));
end

C = gallery('circul',rand(n,1)); [V2,D2] = eig(C);

for j=1:n
figure; bar(1:n,[real(V2(:,j)),imag(V2(:,j))],1,'grouped');
title(sprintf('Circulant_matrix_2_eigenvector_%d',j));
xlabel('\bf_vector_component_index','fontsize',14);
ylabel('\bf_vector_component_value','fontsize',14);
legend('real_part','imaginary_part','location','southwest');
print('-depsc2',sprintf('.../PICTURES/circeig2ev%d.eps',j));
end

figure; plot(1:n,real(diag(D1)),'r+',1:n,imag(diag(D1)),'b+',...
1:n,real(diag(D2)),'m*',1:n,imag(diag(D2)),'k*');
ax = axis; axis([0 n+1 ax(3) ax(4)]);
xlabel('\bf_index_of_eigenvalue','fontsize',14);
ylabel('\bf_eigenvalue','fontsize',14);
legend('C_1:_real(ev)','C_1:_imag(ev)','C_2:_real(ev)','C_2:_imag(ev)','location','northeast');

print -depsc2 '.../PICTURES/circeigev.eps';
```

Random  $8 \times 8$  circulant matrices  $C_1, C_2$  ( $\rightarrow$  Def. 6.1.3)

eigenvalues  $\triangleright$

Generated by MATLAB-command:

```
C = gallery('circul',rand(n,1));
```

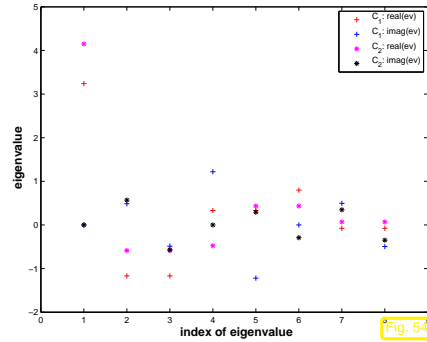
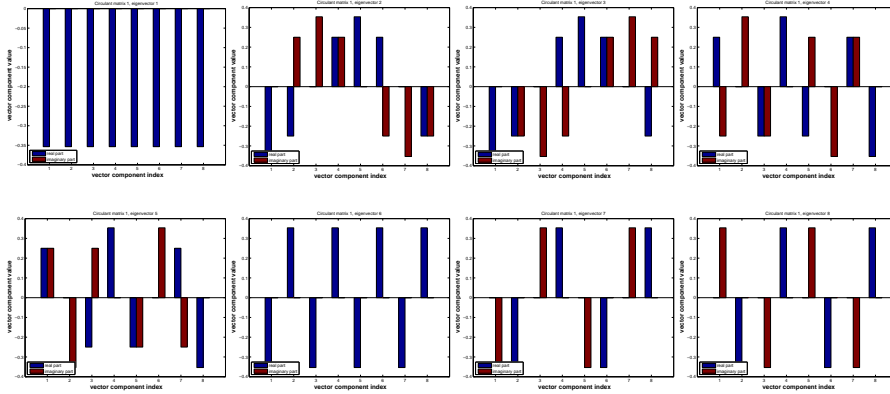


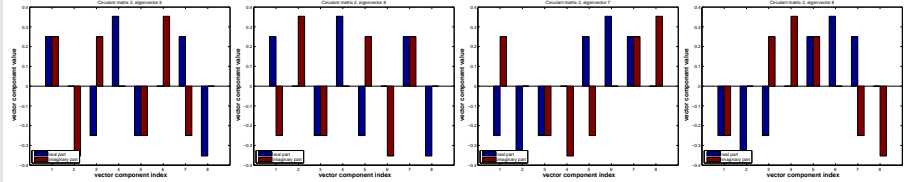
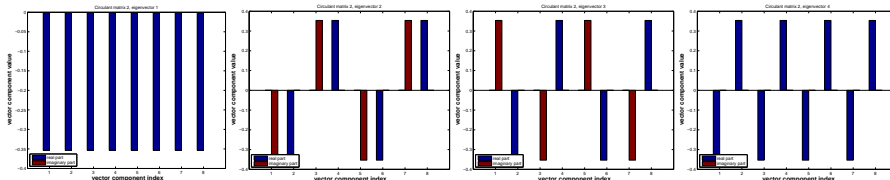
Fig. 54

Little relationship between (complex!) eigenvalues can be observed, as can be expected from random matrices with entries  $\in [0, 1]$ .

Eigenvectors of matrix  $C_1$ :

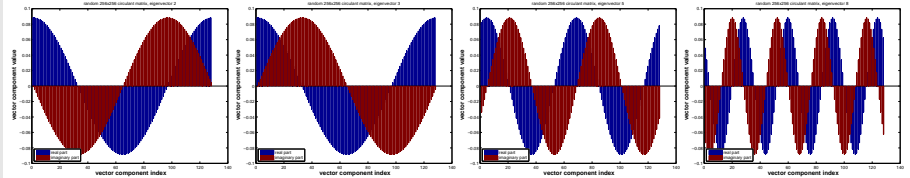


Eigenvectors of matrix  $C_2$



Observation: the different random circulant matrices have the **same eigenvectors!**

Eigenvectors of  $C = \text{gallery}('circul',(1:128));$ :



The eigenvectors remind us of sampled *trigonometric functions*  $\cos(k/n), \sin(k/n), k = 0, \dots, n-1!$

Remark 6.2.3 (Why using  $\mathbb{K} = \mathbb{C}$ ?).

Ex. 6.2.1: *complex* eigenvalues/eigenvectors for general circulant matrices.

Recall from analysis: unified treatment of trigonometric functions via *complex exponential function*

$$\exp(it) = \cos(t) + i \sin(t), \quad t \in \mathbb{R}.$$

**C!** The field of complex numbers  $\mathbb{C}$  is the *natural framework* for the analysis of linear, time-invariant filters, and the development of algorithms for circulant matrices.

notation: *n*th root of unity  $\omega_n := \exp(-2\pi i/n) = \cos(2\pi/n) - i \sin(2\pi/n), \quad n \in \mathbb{N}$

$$\text{satisfies } \bar{\omega}_n = \omega_n^{-1}, \quad \omega_n^n = 1, \quad \omega_n^{n/2} = -1, \quad \omega_n^k = \omega_n^{k+n} \quad \forall k \in \mathbb{Z}, \quad (6.2.1)$$

$$\sum_{k=0}^{n-1} \omega_n^{kj} = \begin{cases} n, & \text{if } j = 0, \\ 0, & \text{if } j \neq 0. \end{cases} \quad (6.2.2)$$

(6.2.2) is a simple consequence of the geometric sum formula

$$\sum_{k=0}^{n-1} q^k = \frac{1-q^n}{1-q} \quad \forall q \in \mathbb{C} \setminus \{1\}, \quad n \in \mathbb{N}. \quad (6.2.3)$$

$$\Rightarrow \sum_{k=0}^{n-1} \omega_n^{kj} = \frac{1-\omega_n^{nj}}{1-\omega} = \frac{1-\exp(-2\pi i j)}{1-\exp(-2\pi i/n)} = 0,$$

because  $\exp(-2\pi i j) = \omega_n^{nj} = (\omega_n^n)^j = 1$  for all  $j \in \mathbb{Z}$ .

Now we want to confirm the conjecture gleaned from Ex. 6.2.1 that vectors with powers of roots of unity are eigenvectors for any circulant matrix. We do this by simple and straightforward computations:

Consider:  $\mathbf{C} \in \mathbb{C}^{n,n}$  circulant matrix ( $\rightarrow$  Def. 6.1.3),  $c_{ij} = u_{i-j}$ , for  $n$ -periodic sequence  $(u_k)_{k \in \mathbb{Z}}, u_k \in \mathbb{C}$

$\mathbf{v}_k \in \mathbb{C}^n$  with  $\mathbf{v}_k := (\omega_n^{jk})_{j=0}^{n-1} \in \mathbb{C}^n, \quad k \in \{0, \dots, n-1\}$ .

$(u_{j-l}\omega_n^{lk})_{l \in \mathbb{Z}}$  is  $n$ -periodic!

$$(\mathbf{C}\mathbf{v}_k)_j = \sum_{l=0}^{n-1} u_{j-l}\omega_n^{lk} = \sum_{l=j-n+1}^j u_{j-l}\omega_n^{lk} \quad (6.2.4)$$

$$= \sum_{l=0}^{n-1} u_l\omega_n^{(j-l)k} = \omega_n^{jk} \sum_{l=0}^{n-1} u_l\omega_n^{-lk} = \lambda_k \cdot \omega_n^{jk} = \lambda_k \cdot (\mathbf{v}_k)_j.$$

change of summation index independent of  $j$ !

$\mathbf{v}$  is eigenvector of  $\mathbf{C}$  to eigenvalue  $\lambda_k = \sum_{l=0}^{n-1} u_l\omega_n^{-lk}$ .

Orthogonal trigonometric basis of  $\mathbb{C}^n =$  eigenvector basis for circulant matrices

$$\left\{ \begin{pmatrix} \omega_n^0 \\ \vdots \\ \omega_n^0 \end{pmatrix}, \begin{pmatrix} \omega_n^0 \\ \omega_n^1 \\ \vdots \\ \omega_n^{n-1} \end{pmatrix}, \dots, \begin{pmatrix} \omega_n^0 \\ \omega_n^{n-2} \\ \omega_n^{2(n-2)} \\ \vdots \\ \omega_n^{(n-1)(n-2)} \end{pmatrix}, \begin{pmatrix} \omega_n^0 \\ \omega_n^{n-1} \\ \vdots \\ \omega_n^{(n-1)^2} \end{pmatrix} \right\}.$$

(6.2.2)  $\Rightarrow$  orthogonality of basis vectors:

$$\mathbf{v}_k := (\omega_n^{jk})_{j=0}^{n-1} \in \mathbb{C}^n: \quad \mathbf{v}_k^H \mathbf{v}_m = \sum_{j=0}^{n-1} \omega_n^{-jk} \omega_n^{jm} = \sum_{j=0}^{n-1} \omega_n^{(m-k)j} \stackrel{(6.2.2)}{=} 0, \quad \text{if } k \neq m. \quad (6.2.5)$$

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

$$\mathbf{F}_n = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \dots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \dots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \dots & \omega_n^{2(n-2)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \dots & \omega_n^{(n-1)^2} \end{pmatrix} = (\omega_n^{lj})_{l,j=0}^{n-1} \in \mathbb{C}^{n,n}. \quad (6.2.6)$$

**Lemma 6.2.1** (Properties of Fourier matrix).

The scaled Fourier-matrix  $\frac{1}{\sqrt{n}}\mathbf{F}_n$  is unitary ( $\rightarrow$  Def. 2.1.1):  $\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^H = \frac{1}{n}\overline{\mathbf{F}_n}$ .

*Proof.* The lemma is immediate from (6.2.5) and (6.2.2), because

$$(\mathbf{F}_n \mathbf{F}_n^H)_{l,j} = \sum_{k=0}^{n-1} \omega_n^{(l-1)k} \overline{\omega_n^{(j-1)k}} = \sum_{k=0}^{n-1} \omega_n^{(l-1)k} \omega_n^{-(j-1)k} = \sum_{k=0}^{n-1} \omega_n^{k(l-j)}, \quad 1 \leq l, j \leq n.$$

**Remark 6.2.4** (Spectrum of Fourier matrix).

$$\frac{1}{n^2}\mathbf{F}_n^4 = \mathbf{I} \Rightarrow \sigma\left(\frac{1}{\sqrt{n}}\mathbf{F}_n\right) \subset \{1, -1, i, -i\},$$

because, if  $\lambda \in \mathbb{C}$  is an eigenvalue of  $\mathbf{F}_n$ , then there is an eigenvector  $\mathbf{x} \in \mathbb{C}^n \setminus \{0\}$  such that  $\mathbf{F}_n \mathbf{x} = \lambda \mathbf{x}$ , see Def. 4.1.1.

**Lemma 6.2.2** (Diagonalization of circulant matrices ( $\rightarrow$  Def. 6.1.3)).

For any circulant matrix  $\mathbf{C} \in \mathbb{K}^{n,n}, c_{ij} = u_{i-j}, (u_k)_{k \in \mathbb{Z}}$   $n$ -periodic sequence, holds true

$$\mathbf{C}\overline{\mathbf{F}_n} = \overline{\mathbf{F}_n} \text{diag}(d_1, \dots, d_n), \quad \mathbf{d} = \mathbf{F}_n(u_0, \dots, u_{n-1})^T.$$

Proof. Straightforward computation, see (6.2.4). □

Conclusion (from  $\bar{\mathbf{F}}_n = n\mathbf{F}_n^{-1}$ ):  $\mathbf{C} = \mathbf{F}_n^{-1} \text{diag}(d_1, \dots, d_n) \mathbf{F}_n$ . (6.2.7)

Lemma 6.2.2, (6.2.7)  $\triangleright$  multiplication with Fourier-matrix will be crucial operation in algorithms for circulant matrices and discrete convolutions.

Therefore this operation has been given a special name:

**Definition 6.2.3** (Discrete Fourier transform (DFT)).

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

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj}, \quad k = 0, \dots, n-1. \quad (6.2.8)$$

Recall the convention also relevant for the discussion of the DFT: vector indexes range from 0 to  $n-1$ !

Terminology:  $\mathbf{c} = \mathbf{F}_n \mathbf{y}$  is also called the (discrete) Fourier transform of  $\mathbf{y}$

MATLAB-functions for discrete Fourier transform (and its inverse):

DFT:  $\mathbf{c} = \text{fft}(\mathbf{y}) \leftrightarrow$  inverse DFT:  $\mathbf{y} = \text{ifft}(\mathbf{c})$ ;

### 6.2.1 Discrete convolution via DFT

Recall discrete periodic convolution  $z_k = \sum_{j=0}^{n-1} u_{k-j} x_j$  ( $\rightarrow$  Def. 6.1.2),  $k = 0, \dots, n-1$   
 $\uparrow$   
 multiplication with circulant matrix ( $\rightarrow$  Def. 6.1.3)  $\mathbf{z} = \mathbf{C}\mathbf{x}$ ,  $\mathbf{C} := (u_{i-j})_{i,j=1}^n$ .

Idea: (6.2.7)  $\triangleright \mathbf{z} = \mathbf{F}_n^{-1} \text{diag}(\mathbf{F}_n \mathbf{u}) \mathbf{F}_n \mathbf{x}$

Code 6.2.6: discrete periodic convolution: straightforward implementation

```
1 function z=pconv(u,x)
2 n = length(x); z = zeros(n,1);
3 for i=1:n, z(i)=dot(conj(u),
4   x([i:-1:1,n:-1:i+1]));
5 end
```

Code 6.2.8: discrete periodic convolution: DFT implementation

```
1 function z=pconvfft(u,x)
2 z = ifft(fft(u).*fft(x));
```

Rem. 6.1.6: discrete convolution of  $n$ -vectors ( $\rightarrow$  Def. 6.1.1) by *periodic* discrete convolution of  $2n-1$ -vectors (obtained by zero padding):

Implementation of discrete convolution ( $\rightarrow$  Def. 6.1.1) based on periodic discrete convolution  $\triangleright$

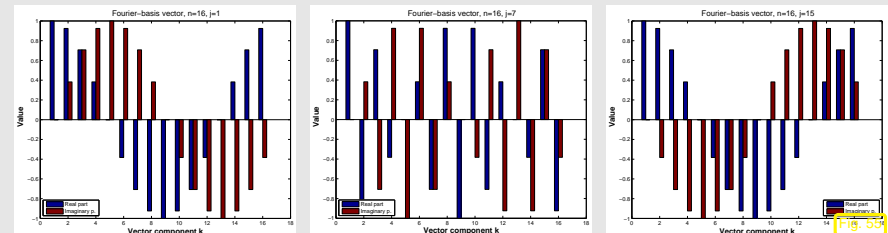
Built-in MATLAB-function:  
 $\mathbf{y} = \text{conv}(\mathbf{h}, \mathbf{x});$

Code 6.2.9: discrete convolution: DFT implementation

```
1 function y = myconv(h,x)
2 n = length(h);
3 %Zero padding
4 h = [h;zeros(n-1,1)]; x =
5   [x;zeros(n-1,1)];
6 %Periodic discrete convolution of length 2n-1
7 y = pconvfft(h,x);
```

### 6.2.2 Frequency filtering via DFT

The trigonometric basis vectors, when interpreted as time-periodic signals, represent harmonic oscillations. This is illustrated when plotting some vectors of the trigonometric basis ( $n = 16$ ):



“slow oscillation/low frequency” “fast oscillation/high frequency” “slow oscillation/low frequency”

$\blacktriangleright$  Dominant coefficients of a signal after transformation to trigonometric basis indicate dominant frequency components.

Terminology: coefficients of a signal w.r.t. trigonometric basis = signal in **frequency domain** (*ger.*: Frequenzbereich), original signal = **time domain** (*ger.*: Zeitbereich).

Example 6.2.10 (Frequency identification with DFT).

Extraction of characteristic frequencies from a distorted discrete periodical signal:

```
1 t = 0:63; x = sin(2*pi*t/64)+sin(7*2*pi*t/64);
2 y = x + randn(size(t)); %distortion
```

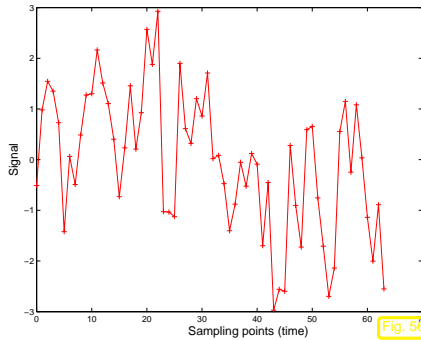


Fig. 56

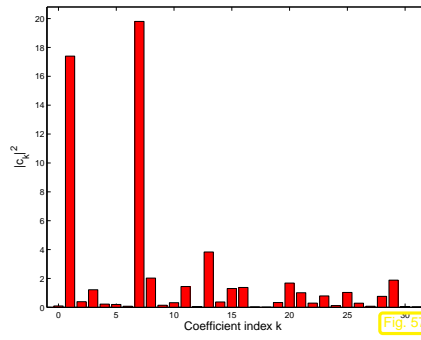
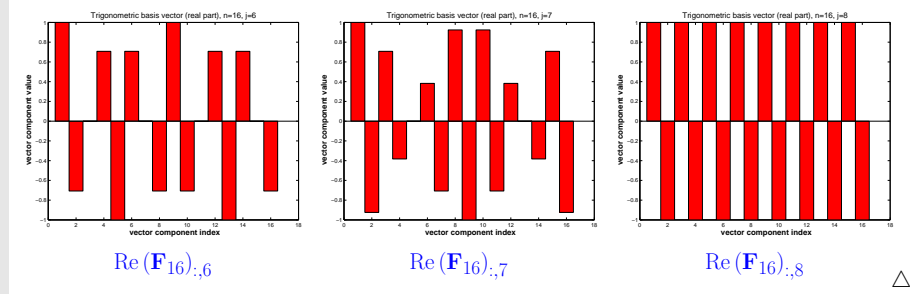
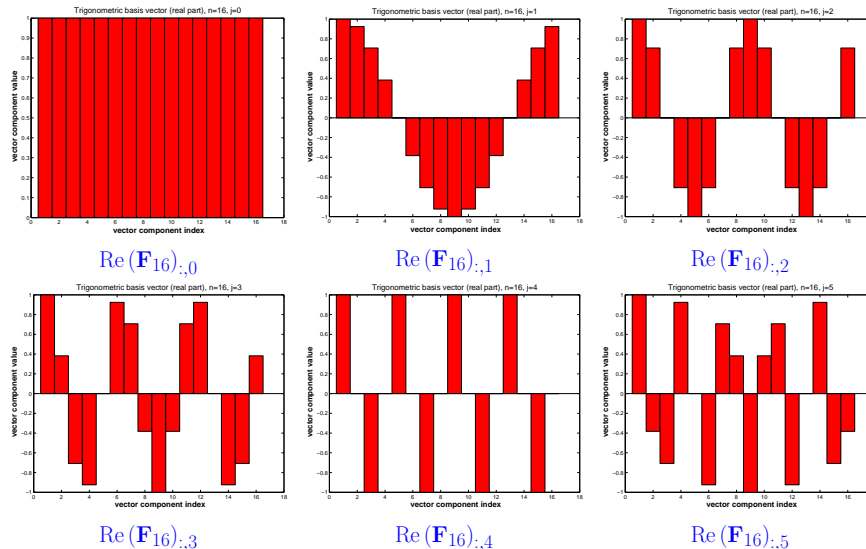


Fig. 57

Frequencies present in unperturbed signal become evident in frequency domain.

Remark 6.2.11 (“Low” and “high” frequencies).

Plots of real parts of trigonometric basis vectors  $(VF_n)_{:,j}$  (= columns of Fourier matrix),  $n = 16$ .



By elementary trigonometric identities:  

$$\text{Re}(F_n)_{:,j} = \left( \text{Re} \omega_n^{(j-1)k} \right)_{k=0}^{n-1} = \left( \text{Re} \exp(2\pi i(j-1)k/n) \right)_{k=0}^{n-1} = \left( \cos(2\pi(j-1)x) \right)_{x=0, \frac{1}{n}, \dots, 1-\frac{1}{n}}$$
  
 Slow oscillations/low frequencies  $\leftrightarrow j \approx 1$  and  $j \approx n$ .  
 Fast oscillations/high frequencies  $\leftrightarrow j \approx n/2$ .

► Frequency filtering of real discrete periodic signals by suppressing certain “Fourier coefficients”.

Code 6.2.12: DFT-based frequency filtering

```
1 function [low, high] =
2   freqfilter(y,k)
3 m = length(y)/2; c = fft(y);
4 clow = c; clow(m+1-k:m+1+k) = 0;
5 chigh = c-clow;
6 high = real(ifft(chigh));
```

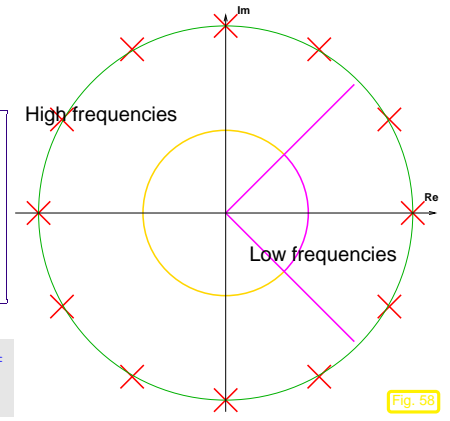


Fig. 58

(can be optimised exploiting  $y_j \in \mathbb{R}$  and  $c_{n/2-k} = \bar{c}_{n/2+k}$ )

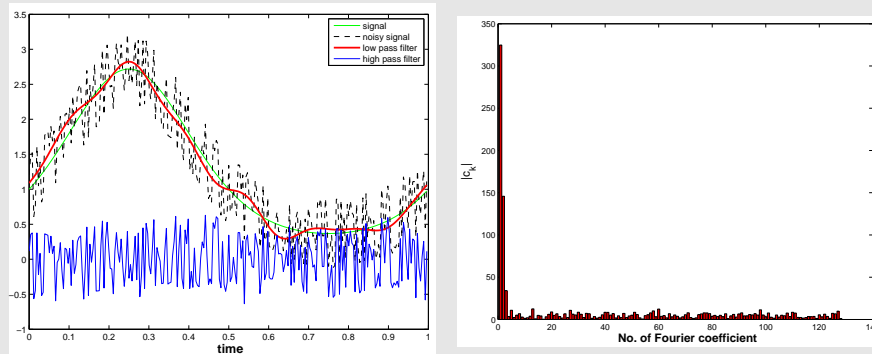
Map  $y \mapsto \text{low}$  (in Code 6.2.11)  $\hat{=}$  low pass filter (ger.: Tiefpass).  
 Map  $y \mapsto \text{high}$  (in Code 6.2.11)  $\hat{=}$  high pass filter (ger.: Hochpass).

Example 6.2.13 (Frequency filtering by DFT).

Noisy signal:

```
n = 256; y = exp(sin(2*pi*((0:n-1)')/n)) + 0.5*sin(exp(1:n)');
```

Frequency filtering by Code 6.2.11 with  $k = 120$ .



Low pass filtering can be used for *denoising*, that is, the removal of high frequency perturbations of a signal.

Example 6.2.14 (Sound filtering by DFT).

Code 6.2.15: DFT based sound compression

```
1 % Sound compression by DFT
2 % Example: ex:soundfilter
3
4 % Read sound data
5 [y,freq,nbits] = wavread('hello.wav');
6
7 n = length(y);
8 fprintf('Read_wav_File:_%d_samples,_%d_freq,_%d_nbits,_%d\n',
9         n,freq,nbits);
10
11 k = 1; s{k} = y; leg{k} = 'Sampled_signal';
12
13 c = fft(y);
14
15 figure('name','sound_signal');
16 plot((22000:44000)/freq,s{1}(22000:44000),'r-');
17 title('samples_sound_signal','fontsize',14);
18 xlabel('\bf_time[s]','fontsize',14);
19 ylabel('\bf_sound_pressure','fontsize',14);
```

Numerical  
Methods  
401-0654

```
18 grid on;
19
20 print -depsc2 '../PICTURES/soundsignal.eps';
21
22 figure('name','sound_frequencies');
23 plot(1:n,abs(c).^2,'m-');
24 title('power_spectrum_of_sound_signal','fontsize',14);
25 xlabel('\bf_index_k_of_Fourier_coefficient','fontsize',14);
26 ylabel('\bf_|c_k|^2','fontsize',14);
27 grid on;
28
29 print -depsc2 '../PICTURES/soundpower.eps';
30
31 figure('name','sound_frequencies');
32 plot(1:3000,abs(c(1:3000)).^2,'b-');
33 title('low_frequency_power_spectrum','fontsize',14);
34 xlabel('\bf_index_k_of_Fourier_coefficient','fontsize',14);
35 ylabel('\bf_|c_k|^2','fontsize',14);
36 grid on;
37
38 print -depsc2 '../PICTURES/soundlowpower.eps';
39
40 for m=[1000,3000,5000]
```

V.  
Gradinaru  
D-ITET,  
D-MATL

6.2  
p. 317

Numeric  
Methods  
401-0654

V.  
Gradina  
D-ITET,  
D-MATL

6.2  
p. 31



Numerical  
Methods  
401-0654

```
11 %Low pass filtering
12 cf = zeros(1,n);
13 cf(1:m) = c(1:m); cf(n-m+1:end) = c(n-m+1:end);
14
15
16 %Reconstruct filtered signal
17 yf = ifft(cf);
18 wavwrite(yf,freq,nbits,sprintf('hello%d.wav',m));
19
20 k = k+1;
21 s{k} = real(yf);
22 leg{k} = sprintf('cutt-off_=%d',m');
23 end
24
25 %Plot original signal and filtered signals
26 figure('name','sound_filtering');
27 plot((30000:32000)/freq,s{1}(30000:32000),'r-',...
28      (30000:32000)/freq,s{2}(30000:32000),'b-',...
29      (30000:32000)/freq,s{3}(30000:32000),'m-',...
30      (30000:32000)/freq,s{2}(30000:32000),'k-');
31 xlabel('\bf_time[s]','fontsize',14);
32 ylabel('\bf_sound_pressure','fontsize',14);
33 legend(leg,'location','southeast');
```

V.  
Gradinaru  
D-ITET,  
D-MATL

6.2  
p. 318

Numeric  
Methods  
401-0654

V.  
Gradina  
D-ITET,  
D-MATL

6.2  
p. 32

```

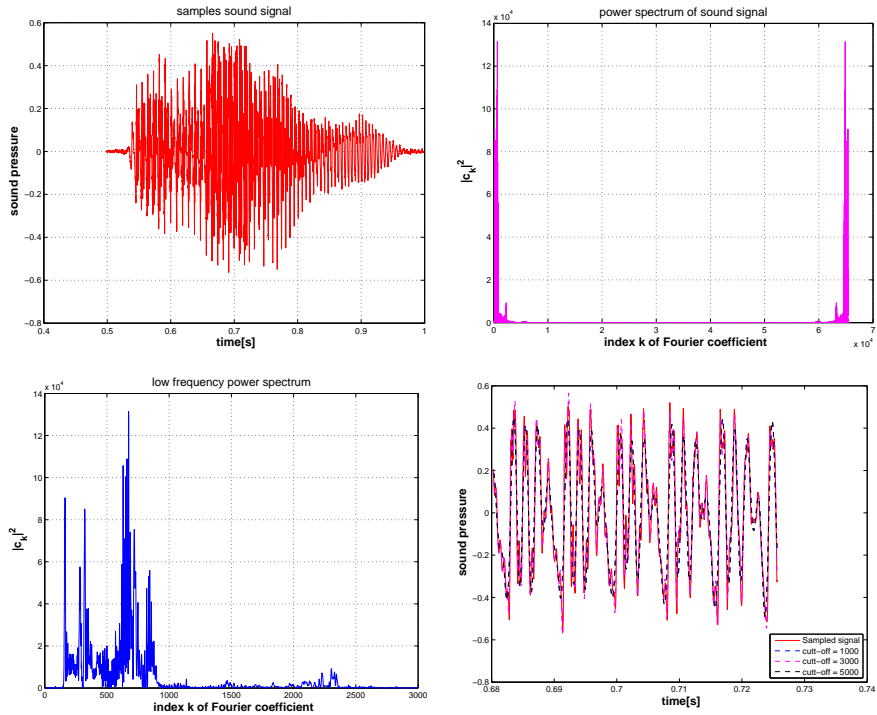
34
35 print -depsc2 '../PICTURES/soundfiltered.eps';

```

```

_____ DFT based low pass frequency filtering of sound _____
[y,sf,nb] = wavread('hello.wav');
c = fft(y); c(m+1:end-m) = 0;
wavwrite(ifft(c),sf,nb,'filtered.wav');

```



The **power spectrum** of a signal  $y \in \mathbb{C}^n$  is the vector  $(|c_j|^2)_{j=0}^{n-1}$ , where  $c = F_n y$  is the discrete Fourier transform of  $y$ .

◇

### 6.2.3 Real DFT

Signal obtained from sampling a time-dependent voltage: a **real** vector.

Aim: efficient DFT (Def. 6.2.3)  $(c_0, \dots, c_{n-1})$  for *real* coefficients  $(y_0, \dots, y_{n-1})^T \in \mathbb{R}^n, n = 2m, m \in \mathbb{N}$ .

If  $y_j \in \mathbb{R}$  in DFT formula (6.2.8), we obtain redundant output

$$\omega_n^{(n-k)j} = \overline{\omega_n^{kj}}, \quad k = 0, \dots, n-1,$$

$$\Rightarrow \bar{c}_k = \sum_{j=0}^{n-1} y_j \overline{\omega_n^{kj}} = \sum_{j=0}^{n-1} y_j \omega_n^{(n-k)j} = c_{n-k}, \quad k = 1, \dots, n-1.$$

**Idea:** map  $y \in \mathbb{R}^n$ , to  $\mathbb{C}^m$  and use DFT of length  $m$ .

$$h_k = \sum_{j=0}^{m-1} (y_{2j} + iy_{2j+1}) \omega_m^{jk} = \sum_{j=0}^{m-1} y_{2j} \omega_m^{jk} + i \cdot \sum_{j=0}^{m-1} y_{2j+1} \omega_m^{jk}, \quad (6.2.9)$$

$$\bar{h}_{m-k} = \sum_{j=0}^{m-1} y_{2j} + iy_{2j+1} \overline{\omega_m^{j(m-k)}} = \sum_{j=0}^{m-1} y_{2j} \omega_m^{jk} - i \cdot \sum_{j=0}^{m-1} y_{2j+1} \omega_m^{jk}. \quad (6.2.10)$$

$$\Rightarrow \sum_{j=0}^{m-1} y_{2j} \omega_m^{jk} = \frac{1}{2}(h_k + \bar{h}_{m-k}), \quad \sum_{j=0}^{m-1} y_{2j+1} \omega_m^{jk} = -\frac{1}{2}i \omega_n^k (h_k - \bar{h}_{m-k}).$$

Use simple identities for roots of unity:

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{jk} = \sum_{j=0}^{m-1} y_{2j} \omega_m^{jk} + \omega_n^k \cdot \sum_{j=0}^{m-1} y_{2j+1} \omega_m^{jk}. \quad (6.2.11)$$

$$\Rightarrow \begin{cases} c_k = \frac{1}{2}(h_k + \bar{h}_{m-k}) - \frac{1}{2}i \omega_n^k (h_k - \bar{h}_{m-k}), & k = 0, \dots, m-1, \\ c_m = \text{Re}\{h_0\} - \text{Im}\{h_0\}, \\ c_k = \bar{c}_{n-k}, & k = m+1, \dots, n-1. \end{cases} \quad (6.2.12)$$

MATLAB-Implementation  
(by a DFT of length  $n/2$ ):

```

Code 6.2.16: DFT of real vectors
1 function c = fftreal(y)
2 n = length(y); m = n/2;
3 if (mod(n,2) ~= 0), error('n_must_be_even'); end
4 y = y(1:2:n)+i*y(2:2:n); h = fft(y); h = [h;h(1)];
5 c = 0.5*(h+conj(h(m+1:-1:1))) - ...
6     (0.5*i*exp(-2*pi*i/n).^((0:m)')) .* ...
7     (h-conj(h(m+1:-1:1)));
8 c = [c; conj(c(m:-1:2))];

```

(Note: not really optimal  
MATLAB implementation)

## 6.2.4 Two-dimensional DFT

A natural analogy:

- one-dimensional data ("signal")  $\longleftrightarrow$  vector  $\mathbf{y} \in \mathbb{C}^n$ ,
- two-dimensional data ("image")  $\longleftrightarrow$  matrix.  $\mathbf{Y} \in \mathbb{C}^{m,n}$

Two-dimensional trigonometric basis of  $\mathbb{C}^{m,n}$ :

$$\text{tensor product matrices } \left\{ (\mathbf{F}_m)_{:,j} (\mathbf{F}_n)_{:,l}^T, 1 \leq j \leq m, 1 \leq l \leq n \right\}. \quad (6.2.13)$$

Basis transform: for  $y_{j_1, j_2} \in \mathbb{C}, 0 \leq j_1 < m, 0 \leq j_2 < n$  compute (nested DFTs !)

$$c_{k_1, k_2} = \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} y_{j_1, j_2} \omega_m^{j_1 k_1} \omega_n^{j_2 k_2}, \quad 0 \leq k_1 < m, 0 \leq k_2 < n.$$

MATLAB function: `fft2(Y)`.

Two-dimensional DFT by *nested one-dimensional DFTs* (6.2.8):

$$\text{fft2}(Y) = \text{fft}(\text{fft}(Y)')'$$

Here: `.'` simply transposes the matrix (no complex conjugation)

Example 6.2.17 (Deblurring by DFT).

Gray-scale pixel image  $\mathbf{P} \in \mathbb{R}^{m,n}$ , actually  $\mathbf{P} \in \{0, \dots, 255\}^{m,n}$ , see Ex. ??.

$(p_{l,k})_{l,k \in \mathbb{Z}} \hat{=}$  periodically extended image:

$$p_{l,j} = (\mathbf{P})_{l+1,j+1} \quad \text{for } 1 \leq l \leq m, 1 \leq j \leq n, \quad p_{l,j} = p_{l+m,j+n} \quad \forall l, k \in \mathbb{Z}.$$

**Blurring** = pixel values get replaced by weighted averages of near-by pixel values  
(effect of distortion in optical transmission systems)

$$c_{l,j} = \sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} p_{l+k,j+q}, \quad \begin{matrix} 0 \leq l < m, \\ 0 \leq j < n, \end{matrix} \quad L \in \{1, \dots, \min\{m, n\}\}. \quad (6.2.14)$$

blurred image      point spread function

Usually:  $L$  small,  $s_{k,m} \geq 0, \sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} = 1$  (an averaging)

Used in test calculations:  $L = 5$

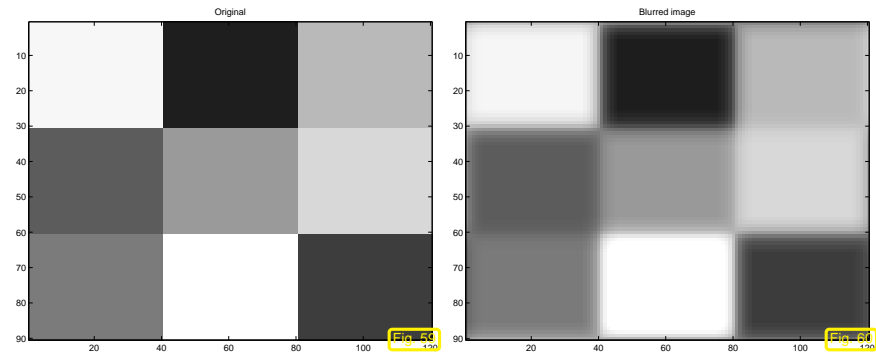
$$s_{k,q} = \frac{1}{1+k^2+q^2}.$$

Code 6.2.18: point spread function

```

1 function S = psf(L)
2 [X,Y] = meshgrid(-L:1:L,-L:1:L);
3 S = 1./(1+X.*X+Y.*Y);
4 S = S/sum(sum(S));

```



Code 6.2.19: MATLAB deblurring experiment

```

1 % Generate artificial "image"
2 P = kron(magic(3), ones(30,40)) * 31;
3 col = [0:1/254:1]' * [1,1,1];
4 figure; image(P); colormap(col); title('Original');
5 print -depsc2 '../PICTURES/dborigimage.eps';
6 % Generate point spread function
7 L = 5; S = psf(L);
8 % Blur image
9 C = blur(P,S);

```

```

10 figure; image(floor(C)); colormap(col); title('Blurred_image');
11 print -depsc2 '../PICTURES/dblurredimage.eps';
12 %Deblur image
13 D = deblur(C,S);
14 figure; image(floor(real(D))); colormap(col);
15 fprintf('Difference_of_images_(Frobenius_norm):_%.1f\n',norm(P-D));

```

►  $V_{\nu,\mu} := \left( \omega_m^{\nu k} \omega_n^{\mu q} \right)_{k,q \in \mathbb{Z}}, 0 \leq \mu < m, 0 \leq \nu < n$  are the eigenvectors of  $\mathcal{B}$ :

$$\mathcal{B}V_{\nu,\mu} = \lambda_{\nu,\mu} V_{\nu,\mu}, \quad \text{eigenvalue } \lambda_{\nu,\mu} = \underbrace{\sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} \omega_m^{\nu k} \omega_n^{\mu q}}_{\text{2-dimensional DFT of point spread function}} \quad (6.2.15)$$

Code 6.2.20: blurring operator

```

1 function C = blur(P,S)
2 [m,n] = size(P); [M,N] = size(S);
3 if (M ~= N), error('S_not_quadratic'); end
4 L = (M-1)/2; C = zeros(m,n);
5 for l=1:m, for j=1:n
6     s = 0;
7     for k=1:(2*L+1), for q=1:(2*L+1)
8         kl = l+k-L-1;
9         if (kl < 1), kl = kl + m; end
10        if (kl > m), kl = kl - m; end
11        jm = j+q-L-1;
12        if (jm < 1), jm = jm + n; end
13        if (jm > n), jm = jm - n; end
14        s = s+P(kl,jm)*S(k,q);
15    end, end
16    C(l,j) = s;
17 end, end

```

Note:

(6.2.14) defines a linear operator  $\mathcal{B}: \mathbb{R}^{m,n} \mapsto \mathbb{R}^{m,n}$  ("blurring operator")

Note: more efficient implementation via MATLAB function `conv2(P,S)`

Inversion of blurring operator  
⇕  
componentwise scaling in "Fourier domain"

Code 6.2.21: DFT based deblurring

```

1 function D = deblur(C,S,tol)
2 [m,n] = size(C); [M,N] = size(S);
3 if (M ~= N), error('S_not_quadratic'); end
4 L = (M-1)/2; Spad = zeros(m,n);
5 %Zero padding
6 Spad(1:L+1,1:L+1) = S(L+1:end,L+1:end);
7 Spad(m-L+1:m,n-L+1:n) = S(1:L,1:L);
8 Spad(1:L+1,n-L+1:n) = S(L+1:end,1:L);
9 Spad(m-L+1:m,1:L+1) = S(1:L,L+1:end);
10 % Inverse of blurring operator
11 SF = fft2(Spad);
12 % Test for invertibility
13 if (margin < 3), tol = 1E-3; end
14 if (min(min(abs(SF))) < tol * max(max(abs(SF))))
15     error('Deblurring_impossible');
16 end
17 % DFT based deblurring
18 D = fft2( ifft2(C) ./ SF );

```

Recall: derivation of (6.2.4) and Lemma 6.2.2. Try this in 2D!

$$\left( \mathcal{B} \left( \left( \omega_m^{\nu k} \omega_n^{\mu q} \right)_{k,q \in \mathbb{Z}} \right) \right)_{l,j} = \sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} \omega_m^{\nu(l+k)} \omega_n^{\mu(j+q)} = \omega_m^{\nu l} \omega_n^{\mu j} \sum_{k=-L}^L \sum_{q=-L}^L s_{k,q} \omega_m^{\nu k} \omega_n^{\mu q}.$$

### 6.3 Fast Fourier Transform (FFT)

At first glance (at (6.2.8)): DFT in  $\mathbb{C}^n$  seems to require asymptotic computational effort of  $O(n^2)$  (matrix×vector multiplication with dense matrix).

Example 6.3.1 (Efficiency of fft).

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

(MATLAB V6.5, Linux, Mobile Intel Pentium 4 - M CPU 2.40GHz, minimum over 5 runs)

Code 6.3.2: timing of different implementations of DFT

```

1 res = [];
2 for n=1:1:3000, y = rand(n,1); c = zeros(n,1);
3   t1 = realmax; for k=1:5, tic;
4     omega = exp(-2*pi*i/n); c(1) = sum(y); s = omega;
5     for j=2:n, c(j) = y(n);
6       for k=n-1:-1:1, c(j) = c(j)*s+y(k); end
7       s = s*omega;
8     end
9   t1 = min(t1, toc);
10 end
11 [I,J] = meshgrid(0:n-1,0:n-1); F = exp(-2*pi*i*I.*J/n);
12 t2 = realmax; for k=1:5, tic; c = F*y; t2 = min(t2,toc); end
13 t3 = realmax; for k=1:5, tic; d = fft(y); t3 = min(t3,toc); end
14 res = [res; n t1 t2 t3];
15 end
16
17 figure('name','FFT_timing');
18 semilogy(res(:,1),res(:,2),'b-',res(:,3),'k-',
19          res(:,4),'r-');
19 ylabel('\bf_run_time_[s]','FontSize',14);
20 xlabel('\bf_vector_length_n','FontSize',14);
21 legend('loop_based_computation','direct_matrix_multiplication','MATLAB_
22        fft_function',1);
23 print -deps2c './PICTURES/ffttime.eps'

```

Numerical Methods 401-0654

V. Gradinaru D-ITET, D-MATL

6.3 p. 333

Numerical Methods 401-0654

V. Gradinaru D-ITET, D-MATL

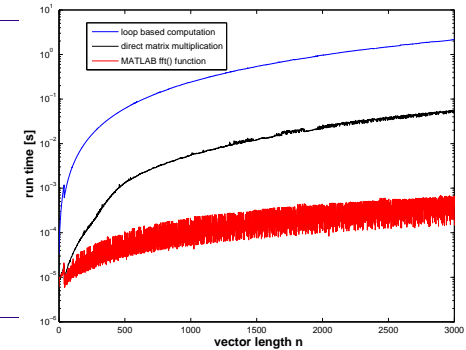
6.3 p. 334

MATLAB-CODE naive DFT-implementation

```

c = zeros(n,1);
omega = exp(-2*pi*i/n);
c(1) = sum(y); s = omega;
for j=2:n
  c(j) = y(n);
  for k=n-1:-1:1
    c(j) = c(j)*s+y(k);
  end
  s = s*omega;
end

```



Numerical Methods 401-0654

V. Gradinaru D-ITET, D-MATL

6.3 p. 33

Numerical Methods 401-0654

V. Gradinaru D-ITET, D-MATL

6.3 p. 33

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

The secret of MATLAB's `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 (6.2.8) for  $n = 2m$ ,  $m \in \mathbb{N}$ :

$$\begin{aligned}
 c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n} jk} \\
 &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n} (2j+1)k} \\
 &= \sum_{j=0}^{m-1} y_{2j} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\tilde{c}_k^{\text{even}}} + e^{-\frac{2\pi i}{n} k} \cdot \sum_{j=0}^{m-1} y_{2j+1} \underbrace{e^{-\frac{2\pi i}{m} jk}}_{=\tilde{c}_k^{\text{odd}}}
 \end{aligned}
 \tag{6.3.1}$$

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

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

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

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

(6.3.1): 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 6.3.3: Recursive FFT

```

1 function c = fftrec(y)
2 n = length(y);
3 if (n == 1), c = y; return;
4 else
5     c1 = fftrec(y(1:2:n));
6     c2 = fftrec(y(2:2:n));
7     c = [c1; c2] +
8         (exp(-2*pi*i/n).^((0:n-1)'))
9         .* [c2; c2];
10 end
    
```

Computational cost of fftrec:



Code 6.3.2: each level of the recursion requires  $O(2^L)$  elementary operations.

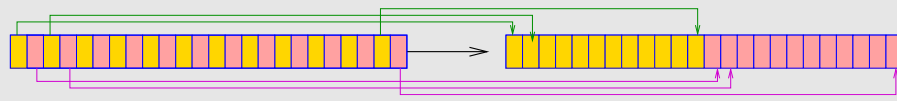
Asymptotic complexity of FFT algorithm,  $n = 2^L$ :  $O(L2^L) = O(n \log_2 n)$

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

Remark 6.3.4 (FFT algorithm by matrix factorization).

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

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



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

$$\text{permutation of rows } P_m^{\text{OE}} \mathbf{F}_n = \begin{pmatrix} & \mathbf{F}_m & & \mathbf{F}_m \\ \mathbf{F}_m \begin{pmatrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n/2-1} \end{pmatrix} & & \mathbf{F}_m \begin{pmatrix} \omega_n^{n/2} & & & \\ & \omega_n^{n/2+1} & & \\ & & \ddots & \\ & & & \omega_n^{n-1} \end{pmatrix} \\ & & & \\ \mathbf{F}_m & & & \mathbf{F}_m \\ & & & \\ & & \mathbf{I} & \mathbf{I} \\ & & \omega_n^0 & -\omega_n^0 \\ & & \omega_n^1 & -\omega_n^1 \\ & & \ddots & \ddots \\ & & \omega_n^{n/2-1} & -\omega_n^{n/2-1} \end{pmatrix}$$

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

$$P_5^{\text{OE}} \mathbf{F}_{10} = \begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 \\ \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 & \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 \\ \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 & \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 \\ \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 & \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 & \omega^8 & \omega^9 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^2 & \omega^5 & \omega^8 & \omega^1 & \omega^4 & \omega^7 \\ \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 \\ \omega^0 & \omega^7 & \omega^4 & \omega^1 & \omega^8 & \omega^5 & \omega^2 & \omega^9 & \omega^6 & \omega^3 \\ \omega^0 & \omega^9 & \omega^8 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix}, \quad \omega := \omega_{10}.$$

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 → Ex. 6.3.1.

*Remark 6.3.5* (FFT based on general factorization).

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

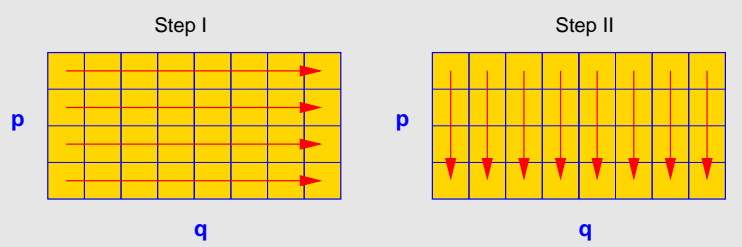
$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{jk} \stackrel{[j=:lp+m]}{=} \sum_{m=0}^{p-1} \sum_{l=0}^{q-1} y_{lp+m} e^{-\frac{2\pi i}{pq}(lp+m)k} = \sum_{m=0}^{p-1} \omega_n^{mk} \sum_{l=0}^{q-1} y_{lp+m} \omega_q^{l(k \bmod q)} \quad (6.3.2)$$

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

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

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

and hence  $q$  DFTs of length  $p$  give all  $c_k$ .



*Remark 6.3.6* (FFT for prime  $n$ ).

When  $n \neq 2^L$ , even the Cooley-Tukey algorithm of Rem. 6.3.5 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:

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

► permutation  $P_{p,g}: \{1, \dots, p-1\} \mapsto \{1, \dots, p-1\}, P_{p,g}(k) = g^k \bmod p,$   
 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.

6.3 Example for  $p = 13$ :  
 p. 341

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

$$\mathbf{F}_{13} \rightarrow \begin{matrix} \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 \\ \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{matrix}$$

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

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

Asymptotic complexity of discrete periodic convolution/multiplication with circulant matrix, see Code 6.2.6:

$$\text{Cost}(z = \text{pconvfft}(u, x), u, x \in \mathbb{C}^n) = O(n \log n).$$

Asymptotic complexity of discrete convolution, see Code 6.2.8:

$$\text{Cost}(z = \text{myconv}(h, x), h, x \in \mathbb{C}^n) = O(n \log n).$$

## 6.4 Trigonometric transformations

Keeping in mind  $\exp(2\pi i x) = \cos(2\pi x) + i \sin(2\pi x)$  we may also consider the real/imaginary parts of the Fourier basis vectors  $(F_n)_{\cdot, j}$  as bases of  $\mathbb{R}^n$  and define the corresponding basis transformation. They can all be realized by means of `fft` with an asymptotic computational effort of  $O(n \log n)$ .

Details are given in the sequel.

### 6.4.1 Sine transform

Another trigonometric basis transform in  $\mathbb{R}^{n-1}$ ,  $n \in \mathbb{N}$ :

$$\left\{ \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \dots, \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} \right\} \leftarrow \left\{ \begin{pmatrix} \sin(\frac{\pi}{n}) \\ \sin(\frac{2\pi}{n}) \\ \vdots \\ \sin(\frac{(n-1)\pi}{n}) \end{pmatrix}, \begin{pmatrix} \sin(\frac{2\pi}{n}) \\ \sin(\frac{4\pi}{n}) \\ \vdots \\ \sin(\frac{2(n-1)\pi}{n}) \end{pmatrix}, \dots, \begin{pmatrix} \sin(\frac{(n-1)\pi}{n}) \\ \vdots \\ \sin(\frac{(n-1)^2\pi}{n}) \end{pmatrix} \right\}$$

Standard basis of  $\mathbb{R}^{n-1}$                       "Sine basis"

Basis transform matrix (sine basis  $\rightarrow$  standard basis):  $S_n := (\sin(jk\pi/n))_{j,k=1}^{n-1} \in \mathbb{R}^{n-1, n-1}$ .

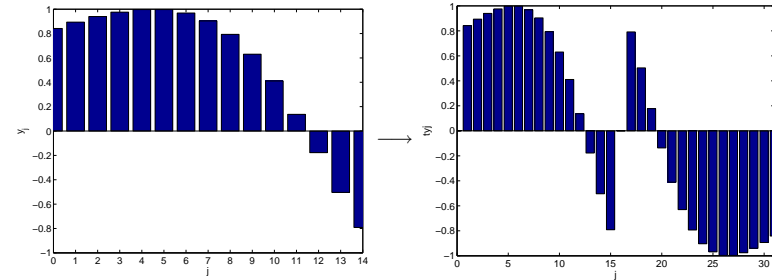
**Lemma 6.4.1** (Properties of the sine matrix).

$$\sqrt{2/n} S_n \text{ is real, symmetric and orthogonal } (\rightarrow \text{Def. 2.1.1})$$

Sine transform:  $s_k = \sum_{j=1}^{n-1} y_j \sin(\pi j k/n)$ ,  $k = 1, \dots, n-1$ . (6.4.1)

DFT-based algorithm for the sine transform ( $\hat{=}$   $S_n \times$  vector):

"wrap around":  $\tilde{y} \in \mathbb{R}^{2n}$ :  $\tilde{y}_j = \begin{cases} y_j & , \text{ if } j = 1, \dots, n-1, \\ 0 & , \text{ if } j = 0, n, \\ -y_{2n-j} & , \text{ if } j = n+1, \dots, 2n-1. \end{cases}$  ( $\tilde{y}$  "odd")



$$\begin{aligned} (F_{2n} \tilde{y})_k &\stackrel{(6.2.8)}{=} \sum_{j=1}^{2n-1} \tilde{y}_j e^{-\frac{2\pi}{2n}kj} \\ &= \sum_{j=1}^{n-1} y_j e^{-\frac{\pi}{n}kj} - \sum_{j=n+1}^{2n-1} y_{2n-j} e^{-\frac{\pi}{n}kj} \\ &= \sum_{j=1}^{n-1} y_j (e^{-\frac{\pi}{n}kj} - e^{\frac{\pi}{n}kj}) \\ &= -2i (S_n \mathbf{y})_k, \quad k = 1, \dots, n-1. \end{aligned}$$

```
function c = sinetrans(y)
n = length(y)+1;
yt = [0, y, 0, -y(end:-1:1)];
ct = fft(yt);
c = -ct(2:n)/(2*i);
```

MATLAB-CODE sine transform

**Remark 6.4.1** (Sine transform via DFT of half length).

Step ①: transform of the coefficients

$$\tilde{y}_j = \sin(j\pi/n)(y_j + y_{n-j}) + \frac{1}{2}(y_j - y_{n-j}), \quad j = 1, \dots, n-1, \quad \tilde{y}_0 = 0.$$

Step ②: real DFT ( $\rightarrow$  Sect. 6.2.3) of  $(\tilde{y}_0, \dots, \tilde{y}_{n-1}) \in \mathbb{R}^n$ : 
$$c_k := \sum_{j=0}^{n-1} \tilde{y}_j e^{-\frac{2\pi i}{n}jk}$$

Hence 
$$\begin{aligned} \operatorname{Re}\{c_k\} &= \sum_{j=0}^{n-1} \tilde{y}_j \cos(-\frac{2\pi i}{n}jk) = \sum_{j=1}^{n-1} (y_j + y_{n-j}) \sin(\frac{\pi j}{n}) \cos(\frac{2\pi i}{n}jk) \\ &= \sum_{j=0}^{n-1} 2y_j \sin(\frac{\pi j}{n}) \cos(\frac{2\pi i}{n}jk) = \sum_{j=0}^{n-1} y_j \left( \sin(\frac{2k+1}{n}\pi j) - \sin(\frac{2k-1}{n}\pi j) \right) \\ &= s_{2k+1} - s_{2k-1} . \\ \operatorname{Im}\{c_k\} &= \sum_{j=0}^{n-1} \tilde{y}_j \sin(-\frac{2\pi i}{n}jk) = -\sum_{j=1}^{n-1} \frac{1}{2}(y_j - y_{n-j}) \sin(\frac{2\pi i}{n}jk) = -\sum_{j=1}^{n-1} y_j \sin(\frac{2\pi i}{n}jk) \\ &= -s_{2k} . \end{aligned}$$

Step ③: extraction of  $s_k$

$s_{2k+1}$ ,  $k = 0, \dots, \frac{n}{2} - 1 \rightarrow$  from recursion  $s_{2k+1} - s_{2k-1} = \operatorname{Re}\{c_k\}$ ,  $s_1 = \sum_{j=1}^{n-1} y_j \sin(\pi j/n)$ ,  
 $s_{2k}$ ,  $k = 1, \dots, \frac{n}{2} - 2 \rightarrow s_{2k} = -\operatorname{Im}\{c_k\}$ .

MATLAB-Implementation (via a `fft` of length  $n/2$ ):

```

MATLAB-CODE Sine transform
function s = sinetrans(y)
n = length(y)+1;
sinevals = imag(exp(i*pi/n).^(1:n-1));
yt = [0 (sinevals.*(y+y(end:-1:1)) + 0.5*(y-y(end:-1:1)))]';
c = fftreal(yt);
s(1) = dot(sinevals,y);
for k=2:N-1
if (mod(k,2) == 0), s(k) = -imag(c(k/2+1));
else, s(k) = s(k-2) + real(c((k-1)/2+1)); end
end

```

Application: diagonalization of local translation invariant linear operators.

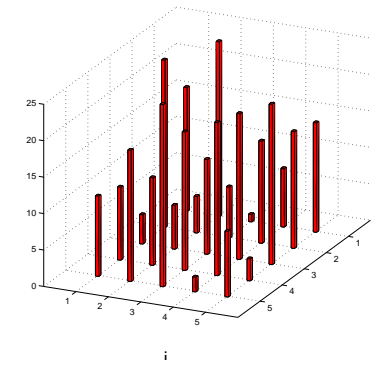
5-points-stencil-operator on  $\mathbb{R}^{n,n}$ ,  $n \in \mathbb{N}$ , in grid representation:

$$T : \mathbb{R}^{n,n} \mapsto \mathbb{R}^{n,n}, \quad \mathbf{X} \mapsto T(\mathbf{X})$$

$$(T(\mathbf{X}))_{ij} := cx_{ij} + cy_{x_{i,j+1}} + cy_{x_{i,j-1}} + cx_{x_{i+1,j}} + cx_{x_{i-1,j}}$$

with  $c, c_y, c_x \in \mathbb{R}$ , convention:  $x_{ij} := 0$  for  $(i, j) \notin \{1, \dots, n\}^2$ .

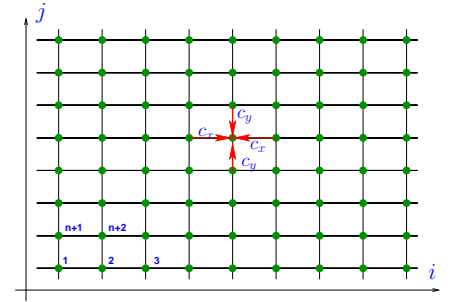
$\mathbf{X} \in \mathbb{R}^{n,n}$   
 $\updownarrow$   
 grid function  $\in \{1, \dots, n\}^2 \mapsto \mathbb{R}$



Identification  $\mathbb{R}^{n,n} \cong \mathbb{R}^{n^2}$ ,  $x_{ij} \sim \tilde{x}_{(j-1)n+i}$  gives matrix representation  $\mathbf{T} \in \mathbb{R}^{n^2, n^2}$  of  $T$ :

$$\mathbf{T} = \begin{pmatrix} \mathbf{C} & c_y \mathbf{I} & 0 & \dots & \dots & 0 \\ c_y \mathbf{I} & \mathbf{C} & c_y \mathbf{I} & & & \vdots \\ 0 & \dots & \dots & \dots & & \\ \vdots & & & c_y \mathbf{I} & \mathbf{C} & c_y \mathbf{I} \\ 0 & \dots & \dots & 0 & c_y \mathbf{I} & \mathbf{C} \end{pmatrix} \in \mathbb{R}^{n^2, n^2},$$

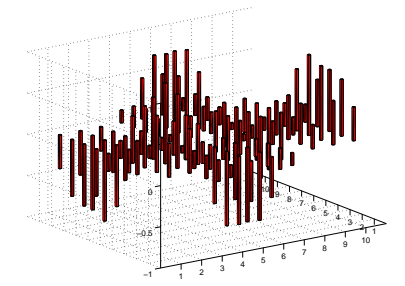
$$\mathbf{C} = \begin{pmatrix} c & c_x & 0 & \dots & \dots & 0 \\ c_x & c & c_x & & & \vdots \\ 0 & \dots & \dots & \dots & & \\ \vdots & & & c_x & c & c_x \\ 0 & \dots & \dots & 0 & c_x & c \end{pmatrix} \in \mathbb{R}^{n,n}.$$



Sine basis of  $\mathbb{R}^{n,n}$ :

$$\mathbf{B}^{kl} = \left( \sin(\frac{\pi}{n+1}ki) \sin(\frac{\pi}{n+1}lj) \right)_{i,j=1}^n \quad (6.4.2)$$

$n = 10$ : grid function  $\mathbf{B}^{2,3}$



$$(T(\mathbf{B}^{kl}))_{ij} = c \sin(\frac{\pi}{n}ki) \sin(\frac{\pi}{n}lj) + c_y \sin(\frac{\pi}{n}ki) (\sin(\frac{\pi}{n+1}l(j-1)) + \sin(\frac{\pi}{n+1}l(j+1))) + c_x \sin(\frac{\pi}{n}lj) (\sin(\frac{\pi}{n+1}k(i-1)) + \sin(\frac{\pi}{n+1}k(i+1)))$$

$$= \sin(\frac{\pi}{n}ki) \sin(\frac{\pi}{n}lj) (c + 2c_y \cos(\frac{\pi}{n+1}l) + 2c_x \cos(\frac{\pi}{n+1}k))$$

Hence  $\mathbf{B}^{kl}$  is **eigenvector** of  $T \leftrightarrow \mathbf{T}$  corresponding to eigenvalue  $c + 2c_y \cos(\frac{\pi}{n+1}l) + 2c_x \cos(\frac{\pi}{n+1}k)$ .

Algorithm for basis transform:

$$\mathbf{X} = \sum_{k=1}^n \sum_{l=1}^n y_{kl} \mathbf{B}^{kl} \Rightarrow x_{ij} = \sum_{k=1}^n \sin(\frac{\pi}{n+1}ki) \sum_{l=1}^n y_{kl} \sin(\frac{\pi}{n+1}lj)$$

MATLAB-CODE two dimensional sine tr.

```
function C = sinft2d(Y)
[m,n] = size(Y);
C = fft([zeros(1,n); Y;...
        zeros(1,n);...
        -Y(end:-1:1,:)]);
C = i*C(2:m+1,:)' / 2;
C = fft([zeros(1,m); C;...
        zeros(1,m);...
        -C(end:-1:1,:)]);
C = i*C(2:n+1,:)' / 2;
```

Hence nested sine transforms ( $\rightarrow$  Sect. 6.2.4) for rows/columns of  $\mathbf{Y} = (y_{kl})_{k,l=1}^n$ .

Here: implementation of sine transform (6.4.1) with "wrapping"-technique.

MATLAB-CODE FFT-based solution of local

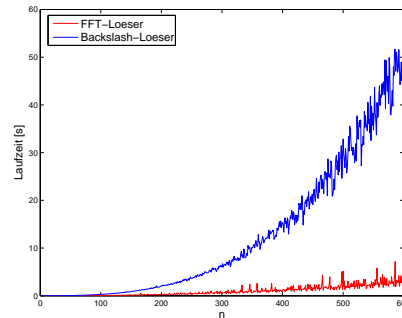
```
function X = fftsolve(B,c,cx,cy)
[m,n] = size(B);
[I,J] = meshgrid(1:m,1:n);
X = 4*sinft2d(sinft2d(B)...
    ./ (c+2*cx*cos(pi/(n+1)*I)+...
        2*cy*cos(pi/(m+1)*J)))...
    / ((m+1)*(n+1));
```

translation invariant linear operators

Example 6.4.2 (Efficiency of FFT-based LSE-solver).

tic-toc-timing (MATLAB V7, Linux, Intel Pentium 4 Mobile CPU 1.80GHz)

```
A = gallery('poisson',n);
B = magic(n);
b = reshape(B,n*n,1);
tic;
C = fftsolve(B,4,-1,-1);
t1 = toc;
tic; x = A\b; t2 = toc;
```



Diagonalization of  $\mathbf{T}$  via 2D sine transform

efficient algorithm

for solving linear system of equations  $T(\mathbf{X}) = \mathbf{B}$

computational cost  $O(n^2 \log n)$ .

## 6.4.2 Cosine transform

Another trigonometric basis transform in  $\mathbb{R}^n$ ,  $n \in \mathbb{N}$ :

standard basis of  $\mathbb{R}^n$  "cosine basis"

$$\left\{ \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \dots, \begin{pmatrix} 0 \\ \vdots \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} \right\} \leftarrow \left\{ \begin{pmatrix} 2^{-1/2} \\ \cos(\frac{\pi}{2n}) \\ \cos(\frac{2\pi}{2n}) \\ \vdots \\ \cos(\frac{(n-1)\pi}{2n}) \end{pmatrix}, \begin{pmatrix} 2^{-1/2} \\ \cos(\frac{3\pi}{2n}) \\ \cos(\frac{6\pi}{2n}) \\ \vdots \\ \cos(\frac{3(n-1)\pi}{2n}) \end{pmatrix}, \dots, \begin{pmatrix} 2^{-1/2} \\ \cos(\frac{(2n-1)\pi}{2n}) \\ \cos(\frac{2(2n-1)\pi}{2n}) \\ \vdots \\ \cos(\frac{(n-1)(2n-1)\pi}{2n}) \end{pmatrix} \right\}$$

Basis transform matrix (cosine basis  $\rightarrow$  standard basis):

$$\mathbf{C}_n = (c_{ij}) \in \mathbb{R}^{n,n} \quad \text{with} \quad c_{ij} = \begin{cases} 2^{-1/2} & , \text{ if } i = 1, \\ \cos((i-1)\frac{2j-1}{2n}\pi) & , \text{ if } i > 1. \end{cases}$$

Lemma 6.4.2 (Properties of cosine matrix).

$\sqrt{2/n} \mathbf{C}_n$  is real and orthogonal ( $\rightarrow$  Def. 2.1.1).

6.4  
p. 353

6.4  
p. 35

Note:  $\mathbf{C}_n$  is not symmetric

cosine transform: 
$$c_k = \sum_{j=0}^{n-1} y_j \cos(k\frac{2j+1}{2n}\pi), \quad k = 1, \dots, n-1, \quad (6.4.3)$$

$$c_0 = \frac{1}{\sqrt{2}} \sum_{j=0}^{n-1} y_j$$

MATLAB-implementation of  $\mathbf{C}_y$  ("wrapping"-technique):

MATLAB-CODE cosine transform

```
function c = costrans(y)
n = length(y);
z = fft([y,y(end:-1:1)]);
c = real([z(1)/(2*sqrt(2)), ...
        0.5*(exp(-i*pi/(2*n)).^(1:n-1)).*z(2:n)]);
```

MATLAB-implementation of  $\mathbf{C}_n^{-1} \mathbf{y}$  ("Wrapping"-technique):

6.4  
p. 354

6.4  
p. 35

MATLAB-CODE : Inverse cosine transform

```
function y=icostrans(c)
n = length(c);
y = [sqrt(2)*c(1), (exp(i*pi/(2*n)).^(1:n-1)).*c(2:end))];
y = ifft([y,0,conj(y(end:-1:2))]);
y = 2*y(1:n);
```

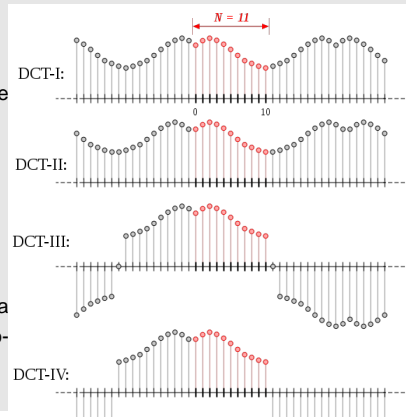
Remark 6.4.3 (Cosine transforms for compression).

The cosine transforms discussed above are named DCT-II and DCT-III.

Various cosine transforms arise by imposing various boundary conditions:

- DCT-II: even around  $-1/2$  and  $N - 1/2$
- DCT-III: even around  $0$  and odd around  $N$

DCT-II is used in JPEG-compression while a slightly modified DCT-IV makes the main component of MP3, AAC and WMA formats.

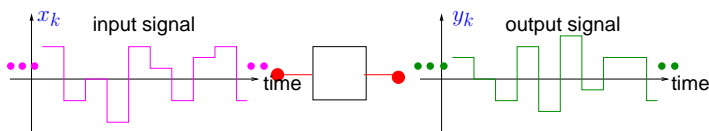


## 6.5 Toeplitz matrix techniques

Example 6.5.1 (Parameter identification for linear time-invariant filters).

- $(x_k)_{k \in \mathbb{Z}}$   $m$ -periodic discrete signal = known input
- $(y_k)_{k \in \mathbb{Z}}$   $m$ -periodic known output signal of a linear time-invariant filter, see Ex. 6.1.1.
- Known: impulse response of filter has maximal duration  $n\Delta t$ ,  $n \in \mathbb{N}$ ,  $n \leq m$

cf. (6.1.1)  $\exists \mathbf{x} = (h_0, \dots, h_{n-1})^T \in \mathbb{R}^n$ ,  $n \leq m$ :  $y_k = \sum_{j=0}^{n-1} h_j x_{k-j}$ . (6.5.1)



Numerical Methods 401-0654

Parameter identification problem: seek  $\mathbf{h} = (h_0, \dots, h_{n-1})^T \in \mathbb{R}^n$  with

$$\|\mathbf{A}\mathbf{h} - \mathbf{y}\|_2 = \left\| \begin{pmatrix} x_0 & x_{-1} & \cdots & \cdots & x_{1-n} \\ x_1 & x_0 & x_{-1} & & \vdots \\ \vdots & x_1 & x_0 & \ddots & \\ \vdots & & \ddots & \ddots & x_{-1} \\ x_{n-1} & & & x_1 & x_0 \\ \vdots & x_n & x_{n-1} & & x_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{m-1} & \cdots & \cdots & \cdots & x_{m-n} \end{pmatrix} \begin{pmatrix} h_0 \\ \vdots \\ h_{n-1} \end{pmatrix} - \begin{pmatrix} y_0 \\ \vdots \\ y_{m-1} \end{pmatrix} \right\|_2 \rightarrow \min.$$

> Linear least squares problem,  $\rightarrow$  Ch. 3 with Toeplitz matrix  $\mathbf{A}$ :  $(\mathbf{A})_{ij} = x_{i-j}$ .

V. Gradinaru D-ITET, D-MATL

System matrix of normal equations ( $\rightarrow$  Sect. 3.1)

$$\mathbf{M} := \mathbf{A}^H \mathbf{A}, \quad (\mathbf{M})_{ij} = \sum_{k=1}^m x_{k-i} x_{k-j} = z_{i-j} \quad \text{due to periodicity of } (x_k)_{k \in \mathbb{Z}}.$$

>  $\mathbf{M} \in \mathbb{R}^{n,n}$  is a matrix with constant diagonals & s.p.d. ("constant diagonals"  $\Leftrightarrow$   $(\mathbf{M})_{i,j}$  depends only on  $i - j$ )

6.4 p. 357

Example 6.5.2 (Linear regression for stationary Markov chains).

Sequence of scalar random variables:  $(Y_k)_{k \in \mathbb{Z}} =$  Markov chain

Assume: stationary (time-independent) correlation

$$\text{Expectation} \quad \mathcal{E}(Y_{i-j} Y_{i-k}) = u_{k-j} \quad \forall i, j, k \in \mathbb{Z}, \quad u_i = u_{-i}.$$

Model: finite linear relationship

$$\exists \mathbf{x} = (x_1, \dots, x_n)^T \in \mathbb{R}^n: Y_k = \sum_{j=1}^n x_j Y_{k-j} \quad \forall k \in \mathbb{Z}.$$

with unknown parameters  $x_j$ ,  $j = 1, \dots, n$ : for fixed  $i \in \mathbb{Z}$

$$\text{Estimator} \quad \mathbf{x} = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} E \left| Y_i - \sum_{j=1}^n x_j Y_{i-j} \right|^2$$

$$\rightarrow E|Y_i|^2 - 2 \sum_{j=1}^n x_j u_k + \sum_{k,j=1}^n x_k x_j u_{k-j} \rightarrow \min.$$

$$\rightarrow \mathbf{x}^T \mathbf{A} \mathbf{x} - 2\mathbf{b}^T \mathbf{x} \rightarrow \min \quad \text{with } \mathbf{b} = (u_k)_{k=1}^n, \quad \mathbf{A} = (u_{i-j})_{i,j=1}^n.$$

Lemma 5.2.2  $\Rightarrow$   $\mathbf{x}$  solves  $\mathbf{A}\mathbf{x} = \mathbf{b}$  (Yule-Walker-equation, see below)

6.5 p. 358

$\mathbf{A} \hat{=}$  Covariance matrix: s.p.d. matrix with constant diagonals.

Numerical Methods 401-0654

V. Gradinaru D-ITET, D-MATL

6.5 p. 35

Numerical Methods 401-0654

V. Gradinaru D-ITET, D-MATL

6.5 p. 36

Matrices with constant diagonals occur frequently in mathematical models. They generalize of circulant matrices → Def. 6.1.3.

Note: "Information content" of a matrix  $\mathbf{M} \in \mathbb{K}^{m,n}$  with constant diagonals, that is,  $(\mathbf{M})_{i,j} = m_{i-j}$ , is  $m + n - 1$  numbers  $\in \mathbb{K}$ .

**Definition 6.5.1** (Toeplitz matrix).  
 $\mathbf{T} = (t_{ij})_{i,j=1}^n \in \mathbb{K}^{m,n}$  is a **Toeplitz matrix**, if there is a vector  $\mathbf{u} = (u_{-m+1}, \dots, u_{n-1}) \in \mathbb{K}^{m+n-1}$  such that  $t_{ij} = u_{j-i}$ ,  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ .

$$\mathbf{T} = \begin{pmatrix} u_0 & u_1 & \cdots & \cdots & u_{n-1} \\ u_{-1} & u_0 & u_1 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & u_1 \\ u_{1-m} & \cdots & \cdots & u_{-1} & u_0 \end{pmatrix}$$

### 6.5.1 Toeplitz matrix arithmetic

$\mathbf{T} = (u_{j-i}) \in \mathbb{K}^{m,n} =$  Toeplitz matrix with generating vector  $\mathbf{u} = (u_{-m+1}, \dots, u_{n-1}) \in \mathbb{C}^{m+n-1}$

Task: efficient evaluation of matrix×vector product  $\mathbf{T}\mathbf{x}$ ,  $\mathbf{x} \in \mathbb{K}^n$

Note: this extended matrix is **circulant** (→ Def. 6.1.3)

$$\mathbf{C} = \begin{pmatrix} \mathbf{T} & \mathbf{S} \\ \mathbf{S} & \mathbf{T} \end{pmatrix} = \begin{pmatrix} u_0 & u_1 & \cdots & \cdots & u_{n-1} & 0 & u_{1-n} & \cdots & \cdots & u_{-1} \\ u_{-1} & u_0 & u_1 & & \vdots & u_{n-1} & 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & u_1 & \vdots & & \ddots & \ddots & u_{1-n} \\ u_{1-n} & \cdots & \cdots & u_{-1} & u_0 & u_1 & & \cdots & u_{n-1} & 0 \\ 0 & u_{1-n} & \cdots & \cdots & u_{-1} & u_0 & u_1 & \cdots & \cdots & u_{n-1} \\ u_{n-1} & 0 & \ddots & & \vdots & u_{-1} & u_0 & u_1 & & \vdots \\ \vdots & \ddots & \ddots & & \vdots & \vdots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & u_{1-n} & \vdots & & \ddots & \ddots & u_1 \\ u_1 & & \cdots & u_{n-1} & 0 & u_{1-n} & \cdots & \cdots & u_{-1} & u_0 \end{pmatrix}$$

This example demonstrates the case  $m = n$

In general:

```
T = toeplitz(u(0:-1:1-m), u(0:n-1));
S = toeplitz([0, u(n-1:-1:n-m+1)], [0, u(1-m:1:-1)]);
```

$$\mathbf{C} \begin{pmatrix} \mathbf{x} \\ 0 \end{pmatrix} = \begin{pmatrix} \mathbf{T}\mathbf{x} \\ \mathbf{S}\mathbf{x} \end{pmatrix}$$

► Computational effort  $O(n \log n)$  for computing  $\mathbf{T}\mathbf{x}$  (FFT based, Sect. 6.3)

### 6.5.2 The Levinson algorithm

Given: **S.p.d.** Toeplitz matrix  $\mathbf{T} = (u_{j-i})_{i,j=1}^n \in \mathbb{C}^{2n-1}$ , generating vector  $\mathbf{u} = (u_{-n+1}, \dots, u_{n-1}) \in \mathbb{C}^{2n-1}$   
 (Symmetry  $\leftrightarrow u_{-k} = u_k$ , w.l.o.g  $u_0 = 1$ )

Task: efficient solution algorithm for LSE  $\mathbf{T}\mathbf{x} = \mathbf{b}$ ,  $\mathbf{b} \in \mathbb{C}^n$   
 (Yule-Walker problem)

Recursive (inductive) solution strategy:

Define:   
 •  $\mathbf{T}_k := (u_{j-i})_{i,j=1}^k \in \mathbb{K}^{k,k}$  (left upper block of  $\mathbf{T}$ )  $\triangleright$   $\mathbf{T}_k$  is s.p.d. Toeplitz matrix,   
 •  $\mathbf{x}^k \in \mathbb{K}^k$ :  $\mathbf{T}_k \mathbf{x}^k = (b_1, \dots, b_k)^T \Leftrightarrow \mathbf{x}^k = \mathbf{T}_k^{-1} \mathbf{b}^k$ ,   
 •  $\mathbf{u}^k := (u_1, \dots, u_k)^T$

Block-partitioned LSE, cf. Rem. ??, Rem. ??

$$\mathbf{T}_{k+1} \mathbf{x}^{k+1} = \begin{pmatrix} \mathbf{T}_k & \begin{matrix} u_k \\ \vdots \\ u_1 \end{matrix} \\ \hline u_k \cdots u_1 & 1 \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{x}}^{k+1} \\ x_{k+1}^{k+1} \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_k \\ b_{k+1} \end{pmatrix} = \begin{pmatrix} \tilde{\mathbf{b}}^{k+1} \\ b_{k+1} \end{pmatrix} \quad (6.5.2)$$

Reversing permutation:  $P_k : \{1, \dots, k\} \mapsto \{1, \dots, k\}$ ,  $P_k(i) := k - i + 1$

$$\tilde{\mathbf{x}}_{k+1} = \mathbf{T}_k^{-1} (\tilde{\mathbf{b}}^{k+1} - x_{k+1}^{k+1} P_k \mathbf{u}^k) = \mathbf{x}^k - x_{k+1}^{k+1} \mathbf{T}_k^{-1} P_k \mathbf{u}^k, \quad (6.5.3)$$

$$x_{k+1}^{k+1} = b_{k+1} - P_k \mathbf{u}^k \cdot \tilde{\mathbf{x}}^{k+1} = b_{k+1} - P_k \cdot \mathbf{x}^k + x_{k+1}^{k+1} P_k \cdot \mathbf{T}_k^{-1} P_k \mathbf{u}^k.$$

Efficient algorithm by using *auxiliary* vectors:  $\mathbf{y}^k := \mathbf{T}_k^{-1} P_k \mathbf{u}^k$

$$\mathbf{x}^{k+1} = \begin{pmatrix} \tilde{\mathbf{x}}^{k+1} \\ x_{k+1}^{k+1} \end{pmatrix} \quad \text{with} \quad \begin{matrix} x_{k+1}^{k+1} = (b_{k+1} - P_k \mathbf{u}^k) / \sigma_k \\ \tilde{\mathbf{x}}^{k+1} = \mathbf{x}^k - x_{k+1}^{k+1} \mathbf{y}^k \end{matrix}, \quad \sigma_k := 1 - P_k \mathbf{u}^k \cdot \mathbf{y}^k. \quad (6.5.4)$$

### Levinson algorithm

(recursive,  $u_{n+1}$  not used!)

Linear recursion:

Computational cost  $\sim (n-k)$  on level  $k, k =$

$0, \dots, n-1$

➤ Asymptotic complexity  $O(n^2)$



```

MATLAB-CODE Levinson algorithm
function [x,y] = levinson(u,b)
k = length(u)-1;
if (k == 0)
    x=b(1); y = u(1); return;
end
[xk,yk] = levinson(u(1:k),b(1:k));
sigma = 1-dot(u(1:k),yk);
t= (b(k+1)-dot(u(k:-1:1),xk))/sigma;
x= [ xk-t*yk(k:-1:1);t];
s= (u(k+1)-dot(u(k:-1:1),yk))/sigma;
y= [yk-s*yk(k:-1:1); s];

```

Remark 6.5.3 (Fast Toeplitz solvers).

FFT-based algorithms for solving  $\mathbf{T}\mathbf{x} = \mathbf{b}$  with asymptotic complexity  $O(n \log^3 n)$  [49] !



[10, Sect. 8.5]: Very detailed and elementary presentation, but the discrete Fourier transform through trigonometric interpolation, which is not covered in this chapter. Hardly addresses discrete convolution.

[29, Ch. IX] presents the topic from a mathematical point of few stressing approximation and trigonometric interpolation. Good reference for algorithms for circulant and Toeplitz matrices.

[47, Ch. 10] also discusses the discrete Fourier transform with emphasis on interpolation and (least squares) approximation. The presentation of signal processing differs from that of the course.

There is a vast number of books and survey papers dedicated to discrete Fourier transforms, see, for instance, [15, 6]. Issues and technical details way beyond the scope of the course are treated there.

## Part II

# Interpolation and Approximation

## Introduction

Distinguish two fundamental concepts:

(I) **data interpolation** (point interpolation, also includes CAD applications):

Given: data points  $(\mathbf{x}_i, \mathbf{y}_i), i = 1, \dots, m, \mathbf{x}_i \in D \subset \mathbb{R}^n, \mathbf{y}_i \in \mathbb{R}^d$

Goal: reconstruction of a (continuous) function  $\mathbf{f} : D \mapsto \mathbb{R}^d$  satisfying **interpolation conditions**

$$f(\mathbf{x}_i) = \mathbf{y}_i, \quad i = 1, \dots, m$$

Additional requirements: • smoothness of  $\mathbf{f}$ , e.g.  $\mathbf{f} \in C^1$ , etc.

• shape of  $\mathbf{f}$  (positivity, monotonicity, convexity)

(II) **function approximation**:

Given: function  $\mathbf{f} : D \subset \mathbb{R}^n \mapsto \mathbb{R}^d$  (often in procedural form  $y=f_{\text{eval}}(x)$ )

Goal: Find a "simple" (\*) function  $\tilde{\mathbf{f}} : D \mapsto \mathbb{R}^d$  such that the difference  $\mathbf{f} - \tilde{\mathbf{f}}$  is "small" (♣)

(\*): "simple"  $\sim$  described by small amount of information, easy to evaluate (e.g. polynomial or piecewise polynomial  $\tilde{\mathbf{f}}$ )