2.8 Fast Fourier Transformation (FFT)

Digital signals

- examples of signals
  - images in medicine and biology, astronomy, ...
  - videos in control of autonomous vehicles, television, remote sensing
  - sound in music and speech
  - spatial, temporal, spectral and combinations

Model of (sampled) digital signal: a sequence of real numbers

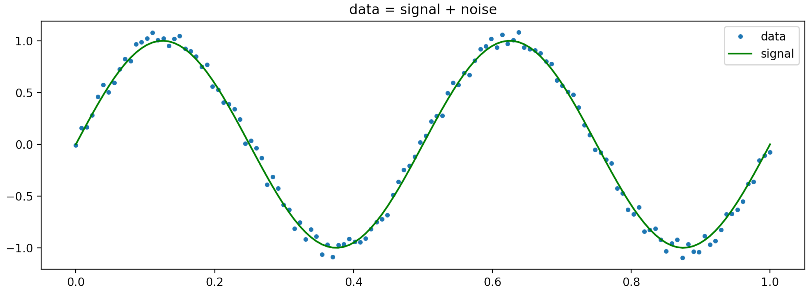
- *fundamental object – sequence* $(x_k)_{k\in\mathbb{Z}}$ of real numbers

$$\ldots, x_{-2}, x_{-1}, x_0, x_1, x_2, \ldots$$

  - is obtained by sampling (measuring) a time-dependent physical quantity at equidistant times
    - eg every year, month, day, hour, second or microsecond
- *computationally feasible object – vector* $x = (x_0, \ldots, x_{n-1})^T$
  1. *truncate sequence* to finite length: $x_0, \ldots, x_{n-1}$ gives vector $x \in \mathbb{R}^n$
  2. *extend vector* to infinite periodic sequence:

$$\ldots, x_{n-2}, x_{n-1}, x_0, x_1, \ldots, x_{n-1}, x_0, x_1, \ldots$$

```
n = 128; t = np.linspace(0, 1.0, n) # sample points
xs = np.sin(4*math.pi*t); # signal
x = xs + 0.2*rn.random(n)-0.1; # data = signal + white nois
plt.plot(t,x,'.',label='data'); plt.plot(t,xs,'-g',label='s
plt.legend(); plt.title('data = signal + noise');
```



Signal processing

- applications:
  - artificial intelligence, machine learning and data mining
  - image enhancement, noise removal, feature detection and object recognition, classification
  - areas include agriculture, medicine, astrophysics, self-driving vehicles
  - signal transmission and storage is based on signal compression

- fundamental tool of signal processing: *linear filter*, modeled as matrix vector product
$$y = Ax$$
  - straight application for noise removal
  - typically a component of more complex applications
- our aim: compute Ax and $A^{-1}x$

The shift (or translation) matrix S

- a simple filter given by the matrix

$$S = \begin{bmatrix} 0 & & & 1 \\ 1 & 0 & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{bmatrix}$$

  - Sx is the vector $(x_{n-1}, x_0, x_1, \dots, x_{n-2})^T$, i.e., the elements are moved down by one element
- shifts by k elements have matrix S^k
- the set $\{I, S, S^2, \dots S^{n-1}\}$ is a commutative *matrix group* (a representation of the cyclic group)
  - in particular $S^n = I$
  - inverse $S^{-1} = S^{n-1}$

```
n = 10; I = np.eye(n)
S = np.zeros((n,n))
S[1:,:] = I[:-1,:]
S[0 ,:] = I[-1, :]
print(" shift matrix S =\n {}".format(S))
```

```
 shift matrix S =
 [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```

Translational invariant filters

- motivation: processing time invariant – today's computation should give the same results as yesterday's given the same data
- this is modelled with the shift matrix and a filter A is *translational invariant* if

$$AS = SA$$

  - or $ASx = SAx$ for all x, which means that applying a filter to some data in the past and moving it to the present gives the same at the result obtained by first moving the data to the present and then applying the filter

- show that all matrices of the form

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

  define translational invariant filters, in fact every translational invariant filter is of this form
- this set is called the *group algebra* defined by the matrix group and the matrices of this form are *circulant*

Question for you

Show that

$$p(S)e_1 = a$$

where $a = (a_0, \ldots, a_{n-1})^T$, $e_1 = (1, 0, \ldots, 0)^T$ and

$$p(S) = a_0 I + a_1 S + \cdots + a_{n-1} S^{n-1}$$

and I is the identity matrix (only nonzeros are ones on the diagonal)

Examples

- A_1 computing the average of the nearest neighbors

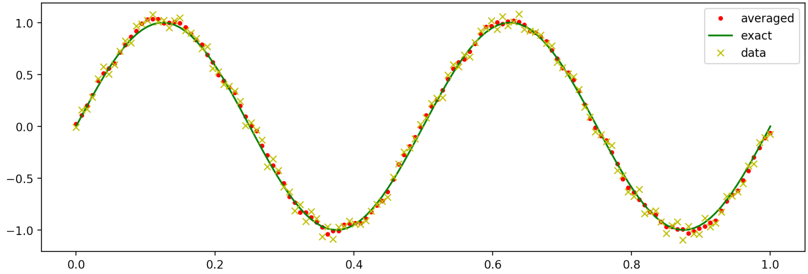
$$A_1 = \frac{1}{3}(S^{-1} + I + S) = \frac{1}{3}\begin{bmatrix} 1 & 1 & & & 1 \\ 1 & 1 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 1 & 1 \\ 1 & & & 1 & 1 \end{bmatrix}$$

- A_2 computing the surplus, i.e., the difference between the value and the average of the neighbors

$$A_2 = \frac{1}{2}(-S^{-1} + 2I - S) = \frac{1}{2}\begin{bmatrix} 2 & -1 & & & -1 \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ -1 & & & 2 & -1 \end{bmatrix}$$

```
# Example A1 -- implementation of a convolution
#               for local averaging
y = np.zeros(len(x))
y[0]    = (x[-1]+x[0]+x[1])/3
y[1:-1] = (x[:-2]+x[1:-1]+x[2:])/3
y[-1]   = (x[-2]+x[-1]+x[0])/3
```

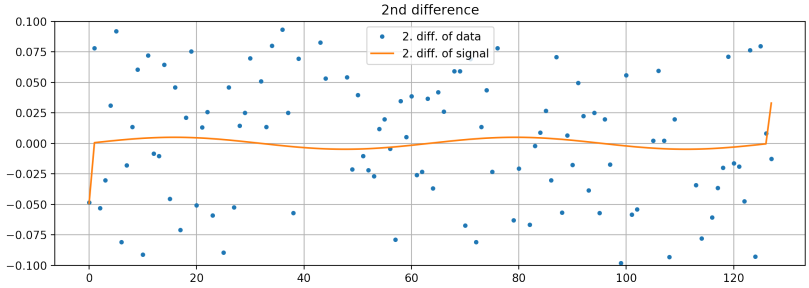
```
plt.plot(t, y,'.r',label='averaged'); plt.plot(t, xs, 'g-',
plt.plot(t, x, 'xy',label='data'); plt.legend();
Ax = y.copy()
```



```
# Example A2 -- implementation of a convolution
#                2nd difference -- curvature
y2 = np.zeros(len(x))
y2[0]    = (-x[-1]+2*x[0]-x[1])/2
y2[1:-1] = (-x[:-2]+2*x[1:-1]-x[2:])/2
y2[-1]   = (-x[-2]+2*x[-1]-x[0])/3

ys2 = np.zeros(len(x))
ys2[0]    = (-xs[-1]+2*xs[0]-xs[1])/2
ys2[1:-1] = (-xs[:-2]+2*xs[1:-1]-xs[2:])/2
ys2[-1]   = (-xs[-2]+2*xs[-1]-xs[0])/3
```

```
plt.plot(y2,'.', label='2. diff. of data'); plt.plot(ys2,'-
plt.axis(ymin=-0.1,ymax=0.1); plt.legend();
plt.grid(True); plt.title('2nd difference');
```



Convolution of two vectors

- consider three vectors x, y and z such that the circulant (matrix) corresponding to z is equal to the matrix product of the two circulants corresponding to x and y, i.e.,

$$\sum_{k=0}^{n-1} z_k S^k = \left(\sum_{k=0}^{n-1} x_k S^k\right)\left(\sum_{k=0}^{n-1} y_k S^k\right)$$

- show that

$$z_k = \sum_{j=0}^{n-1} x_{k-j} y_j$$

which is the convolution of x and y written as

$$z = x * y$$

Eigenvectors and eigenvalues of the shift matrix

- as $S^n = I$ one has for all eigenvalues $\lambda_k^n = 1$ and one can see that
$$\lambda_k = \exp(-2\pi ik/n) = \omega_n^{-k}$$
- the eigenvectors are of the form
$$v_k = (1, \omega_n^k, \ldots, \omega_n^{k(n-1)})^T$$
- the *discrete Fourier transform matrix* is then the matrix $F_n = [v_0, v_1, \ldots, v_{n-1}]$ and by definition one has
$$SF_n = F_n W_n$$
where $W_n = \mathrm{diag}(\lambda_0, \ldots, \lambda_{n-1})$

Examples of the Fourier transform matrix

$$F_n = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \cdots & \omega_n \end{bmatrix}$$

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

where $\omega_n = \exp(-2i\pi/n)$ and $i = \sqrt{-1}$

```
nn=5; fn = scft.fft(np.eye(nn));
np.set_printoptions(precision=2,
  formatter={'complexfloat' : lambda x : \
    ((" {0:4.1f}  ","{0:4.1f}+{1:3.1f}i", \
    "{0:4.1f}-{1:3.1f}i")[int(np.sign(x.imag))])\
             .format(x.real,abs(x.imag))})
print('Fourier trsf matrix F{} = \n {}'.format(nn,fn))

Fourier trsf matrix F5 =
 [[   1.0         1.0         1.0         1.0         1.0   ]
 [    1.0     0.3-1.0i -0.8-0.6i -0.8+0.6i  0.3+1.0i]
 [    1.0    -0.8-0.6i  0.3+1.0i  0.3-1.0i -0.8+0.6i]
 [    1.0    -0.8+0.6i  0.3-1.0i  0.3+1.0i -0.8-0.6i]
 [    1.0     0.3+1.0i -0.8+0.6i -0.8-0.6i  0.3-1.0i]]
```

Properties of the Fourier transform matrix

- $F_n \overline{F}_n = nI$
- $F_n^T = F_n$
- $F_n^4 = I$

The Gauss Relationship

Proposition

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

- this result is used to show various properties of the Fourier matrix F_n

Eigenvectors and eigenvalues of the circulant matrices

- The circulant matrix

$$A = \sum_{j=0}^{n-1} a_j S^j$$

  has the same eigenvectors v_k as the shift matrix S
  - thus all circulant matrices share the same eigenvectors!
- Furthermore, this circulant matrix A has the eigenvalues

$$\hat{a}_k = \sum_{j=0}^{n-1} a_j \omega_n^{-kj}$$

  - the vector of eigenvalues is thus a Fourier transform of the vector of coefficients a

convolution by padding

- infinite sequence convolutions

$$z_i = \sum_{k=-\infty}^{\infty} x_k y_{i-k}$$

- extend finite dimensional vector as infinite sequence by setting other elements to zero

$$x = (\dots, 0, x_0, x_1, \dots, x_{n-1}, 0, 0, \dots)$$

- do the same for y
- the nonzero elements of z are $z_0, z_1, \dots, 2n-2$, can be written as matrix vector product:

$$\begin{bmatrix} x_0 & & & \\ x_1 & x_0 & & \\ & \ddots & \ddots & \\ x_{n-1} & & \ddots & x_0 \\ & x_{n-1} & & x_1 \\ & & \ddots & \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

Example – convolution in numpy

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad y = \begin{bmatrix} 5 \\ 7 \\ 4 \end{bmatrix}$$

- (padded) convolution used from numpy as (Toeplitz) matrix-vector product

$$\begin{bmatrix} 1 & & \\ 2 & 1 & \\ 3 & 2 & 1 \\ & 3 & 2 \\ & & 3 \end{bmatrix} \begin{bmatrix} 5 \\ 7 \\ 4 \end{bmatrix} = \begin{bmatrix} 5 \\ 17 \\ 33 \\ 29 \\ 12 \end{bmatrix}$$

Example – extend $x_e = (x_0, \dots, x_{n-1}, x_0, \dots, x_{n-2})^T$

- extended x is

$$\begin{bmatrix} 2 \\ 3 \\ 1 \\ 2 \\ 3 \end{bmatrix}$$

- numpy's convolution is now

$$\begin{bmatrix} 2 & & \\ 3 & 2 & \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 3 & 2 & 1 \\ & 3 & 2 \\ & & 3 \end{bmatrix} \begin{bmatrix} 5 \\ 7 \\ 4 \end{bmatrix} = \begin{bmatrix} 10 \\ 29 \\ 34 \\ 29 \\ 33 \\ 29 \\ 12 \end{bmatrix}$$

- rows 3 to 5 are now just the circular convolution (use numpy's 'valid' keyword)

```
# convolutions in numpy
xx = [1, 2, 3]
yy = [5, 7, 4]

z1 = np.convolve(xx, yy)
print('numpy convolution of x and y : {}'.format(z1))

z2 = np.convolve(xx[1:] + xx, yy) # note: + is concatenati
print('numpy convolution of extended x with y : {}'.format

z3 = np.convolve(xx[1:] + xx, yy, 'valid')
print('circular convolution of x and y : {}'.format(z3))

numpy convolution of x and y : [ 5 17 33 29 12]
numpy convolution of extended x with y : [10 29 34 29 33 2
circular convolution of x and y : [34 29 33]
```

```
xxf = scft.fft(xx)
yyf = scft.fft(yy)
print('(Hadamard) product of Fourier transforms: {}'.format
z3f = scft.fft(z3)
print('Fourier transform of circular convolution: {}'.forma
print(scft.ifft(xxf*yyf))
```

```
(Hadamard) product of Fourier transforms: [   96.0    3.0+3
Fourier transform of circular convolution: [   96.0    3.0-
[   34.0      29.0      33.0  ]
```

The convolution theorem

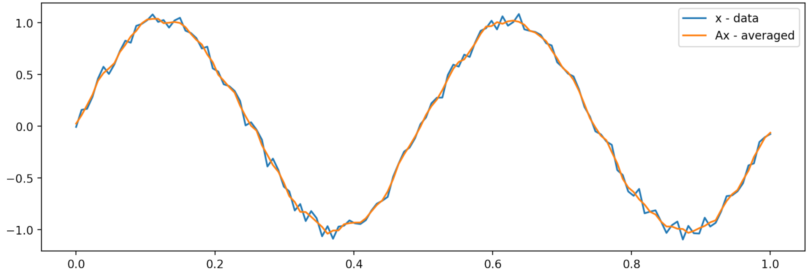
Theorem
The Fourier transformation of the convolution of two vectors x and y is equal to the product of the Fourier transforms of the vectors x and y.
Proof (sketch)

- The convolution has been obtained from the product of the circulant matrices corresponding to vectors x and y.
- As all the circulant matrices have the same eigen vectors, the Fourier transform provides the eigen values of the the two factors and the product.
- The eigenvalues of the product of two circulants is the product of the eigenvalues of the factors.

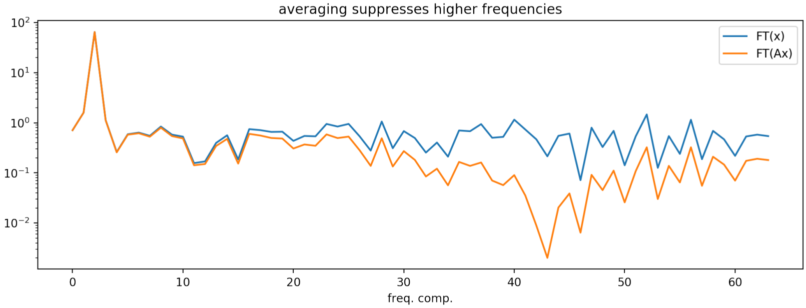
Computing the matrix vector product AB if A and B have the same (known) eigenvectors

- let $A = TD_A T^{-1}$ and $B = TD_B T^{-1}$
- here T is the matrix where the eigenvectors of A and B are the columns of T as
$$AT = TD_A$$
- then the product is
$$AB = TD_A D_B T^{-1}$$
- this leads to a fast method, if the multiplications with T and T^{-1} can be done fast

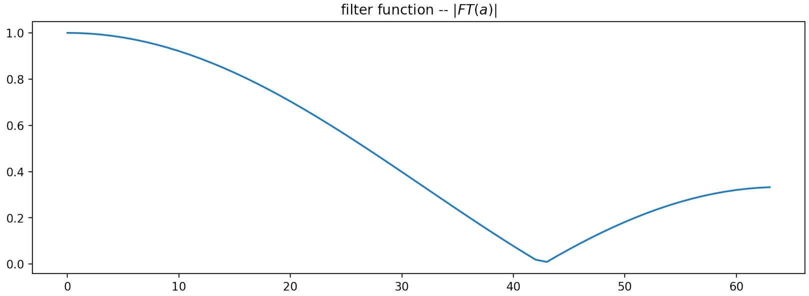
```python
# Fourier transforms
plt.plot(t,x,label='x - data');
plt.plot(t,Ax,label='Ax - averaged')
plt.legend();
```



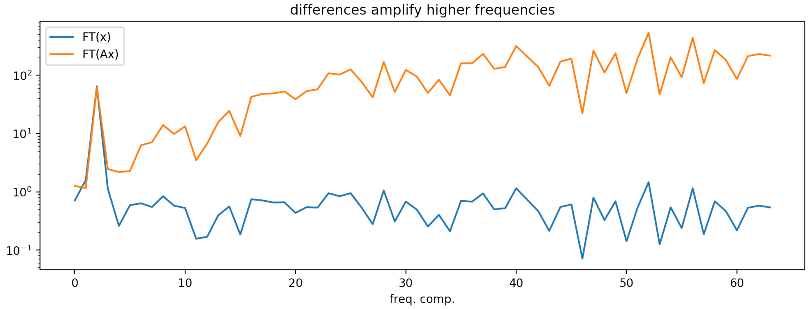
```
xf = scft.fft(x)
yf = scft.fft(y)
plt.semilogy(abs(xf[:64]),label='FT(x)'); plt.xlabel('freq
plt.semilogy(abs(yf[:64]),label='FT(Ax)');
plt.legend(); plt.title("averaging suppresses higher freque
```



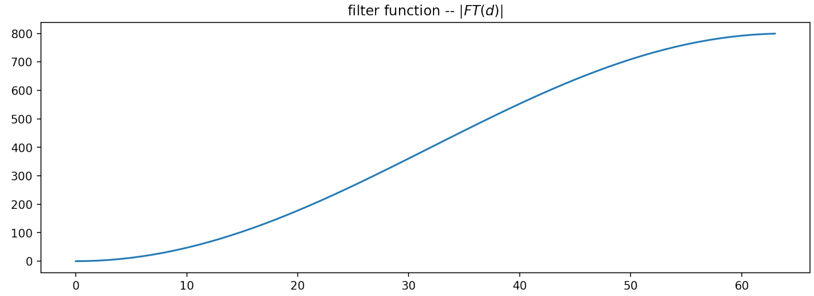
```
# Computing how the error is suppressed
# ... compare with previous plot
a = np.zeros(len(x)); a[-1]=a[0]=a[1]=1.0/3
af = scft.fft(a);
plt.plot(abs(af[:64])); plt.title('filter function -- $|FT(
```



```
xf = scft.fft(x)
y2f =200*scft.fft(y2)
plt.semilogy(abs(xf[:64]),label='FT(x)'); plt.xlabel('freq
plt.semilogy(abs(y2f[:64]),label='FT(Dx)');
plt.legend(); plt.title("differences amplify higher frequen
```



```
# Computing how the error is amplified
# ... compare with previous plot
d = np.zeros(len(x)); d[-1]=d[1]=-1.0; d[0] = 2
df = 200*scft.fft(d);
plt.plot(abs(df[:64])); plt.title('filter function -- $|FT
```



Fast Fourier Transformation

Fast Fourier Transform

How should we calculate the DFT? Calculating

$$\xi^{kj} = \cos 2\pi kj/n + i \sin 2\pi kj/n$$

is expensive. It is possible to pre-calculate and store ξ^{kj} but n^2 multiplications remain.
Lets consider the case where $n = 4$. The discrete Fourier transform is

$$\begin{aligned}
\hat{x}_0 &= x_0\xi^0 + x_1\xi^0 + x_2\xi^0 + x_3\xi^0, \\
\hat{x}_1 &= x_0\xi^0 + x_1\xi^1 + x_2\xi^2 + x_3\xi^3, \\
\hat{x}_2 &= x_0\xi^0 + x_1\xi^2 + x_2\xi^4 + x_3\xi^6, \\
\hat{x}_3 &= x_0\xi^0 + x_1\xi^3 + x_2\xi^6 + x_3\xi^9.
\end{aligned}$$

16 multiplications and 12 additions are required.

Lets rewrite the discrete Fourier transformation as

$$\begin{aligned}
\hat{x}_0 &= (x_0 + \xi^0 x_2) + \xi^0(x_1 + \xi^0 x_3),\\
\hat{x}_1 &= (x_0 + \xi^2 x_2) + \xi^1(x_1 + \xi^2 x_3),\\
\hat{x}_2 &= (x_0 + \xi^4 x_2) + \xi^2(x_1 + \xi^4 x_3),\\
\hat{x}_3 &= (x_0 + \xi^6 x_2) + \xi^3(x_1 + \xi^6 x_3).
\end{aligned}$$

This rewrite reduces the number of operations to 12 multiplications and 12 additions/subtractions.
The above idea can be extended to larger problems.

If $n = 2^\alpha$, we can use the fast-Fourier transform (FFT). Specifically, suppose $n = 2m$, $k \leq m-1$. Then

$$\begin{aligned}
\hat{x}_k &= x_0 + x_1\xi^k + x_2\xi^{2k} + \cdots + x_{n-1}\xi^{(n-1)k} \\
&= x_0 + x_2(\xi^2)^k + x_4(\xi^2)^{2k} + \cdots + x_{2(m-1)}(\xi^2)^{(m-1)k} \\
&+ \xi^k(x_1 + x_3(\xi^2)^k + x_5(\xi^2)^{2k} + \cdots + x_{2m-1}(\xi^2)^{(m-1)k}).
\end{aligned}$$

Hence

$$\hat{x}_k = \hat{y}_k + \xi^k \hat{z}_k$$

where

$$\hat{y}_k = y_0 + y_1\zeta^k + y_2\zeta^{2k} + \cdots + y_{m-1}\zeta^{(m-1)k}$$

$$\hat{z}_k = z_0 + z_1\zeta^k + z_2\zeta^{2k} + \cdots + z_{m-1}\zeta^{(m-1)k}$$

and $\zeta = \xi^2$, $y_i = x_{2i}$ and $z_i = x_{2i+1}$.

An additional simplification can also be used. Since $\xi^{n/2} = -1$, we have

$$\begin{aligned} \hat{x}_k &= \hat{y}_k + \xi^k \hat{z}_k \\ \hat{x}_{k+m} &= \hat{y}_k - \xi^k \hat{z}_k, \end{aligned}$$

for $(0 \leq k \leq m - 1)$.

Applying the above simplification to the 4×4 example gives,

$$\begin{aligned}
\hat{x}_0 &= (x_0 + \xi^0 x_2) + \xi^0 (x_1 + \xi^0 x_3) = (x_0 + \xi^0 x_2) + \xi^0 (x_1 + \xi^0 x_3), \\
\hat{x}_1 &= (x_0 + \xi^2 x_2) + \xi^1 (x_1 + \xi^2 x_3) = (x_0 - \xi^0 x_2) + \xi^1 (x_1 - \xi^0 x_3), \\
\hat{x}_2 &= (x_0 + \xi^4 x_2) + \xi^2 (x_1 + \xi^4 x_3) = (x_0 + \xi^0 x_2) - \xi^0 (x_1 + \xi^0 x_3), \\
\hat{x}_3 &= (x_0 + \xi^6 x_2) + \xi^3 (x_1 + \xi^6 x_3) = (x_0 - \xi^0 x_2) - \xi^1 (x_1 - \xi^0 x_3),
\end{aligned}$$

which only requires 4 multiplications and 8 add/sub.

- in the next slides we will carefully reformulate the all the computations and derivate a factorisation of the Fourier transform matrix using 4 steps:

1. permutation
2. 2 Fourier transforms of size 2
3. multiplication with complex diagonal matrix
4. 2 Fourier transforms of size 2

Matrix Notation

$$\begin{aligned}
\hat{x}_0 &= (x_0 + \xi^0 x_2) + \xi^0(x_1 + \xi^0 x_3) = (x_0 + x_2) + \xi^0(x_1 + x_3),\\
\hat{x}_1 &= (x_0 + \xi^2 x_2) + \xi^1(x_1 + \xi^2 x_3) = (x_0 - x_2) + \xi^1(x_1 - x_3),\\
\hat{x}_2 &= (x_0 + \xi^4 x_2) + \xi^2(x_1 + \xi^4 x_3) = (x_0 + x_2) - \xi^0(x_1 + x_3),\\
\hat{x}_3 &= (x_0 + \xi^6 x_2) + \xi^3(x_1 + \xi^6 x_3) = (x_0 - x_2) - \xi^1(x_1 - \xi^0 x_3),
\end{aligned}$$

Step 1 – permutation

$$x^I = \begin{bmatrix} x_0 & x_2 & x_1 & x_3 \end{bmatrix}^T = Px$$

Step 2 – compute Fourier transforms of order two

$$x^{II} = \begin{bmatrix} x_0 + x_2 & x_0 - x_2 & x_1 + x_3 & x_1 - x_3 \end{bmatrix}^T = \begin{bmatrix} F_2 & \\ & F_2 \end{bmatrix} x^I$$

$$\begin{aligned}
\hat{x}_0 &= (x_0 + \xi^0 x_2) + \xi^0(x_1 + \xi^0 x_3) = (x_0 + x_2) + \xi^0(x_1 + x_3), \\
\hat{x}_1 &= (x_0 + \xi^2 x_2) + \xi^1(x_1 + \xi^2 x_3) = (x_0 - x_2) + \xi^1(x_1 - x_3), \\
\hat{x}_2 &= (x_0 + \xi^4 x_2) + \xi^2(x_1 + \xi^4 x_3) = (x_0 + x_2) - \xi^0(x_1 + x_3), \\
\hat{x}_3 &= (x_0 + \xi^6 x_2) + \xi^3(x_1 + \xi^6 x_3) = (x_0 - x_2) - \xi^1(x_1 - \xi^0 x_3),
\end{aligned}$$

Step 3 – multiply with complex diagonal matrix

$$x^{III} = \begin{bmatrix} x_0 + x_2 & x_0 - x_2 & \xi^0(x_1 + x_3) & \xi^1(x_1 - x_3) \end{bmatrix}^T = \begin{bmatrix} I_2 & \\ & \Theta_2 \end{bmatrix} X^{II}$$

Step 4 – compute Fourier transforms of order two

$$\hat{x} = \begin{bmatrix} I_2 & I_2 \\ I_2 & -I_2 \end{bmatrix} x^{III} = \begin{bmatrix} I_2 & I_2 \\ I_2 & -I_2 \end{bmatrix} \begin{bmatrix} I_2 & \\ & \Theta_2 \end{bmatrix} \begin{bmatrix} F_2 & \\ & F_2 \end{bmatrix} Px$$

Matrix Factorisation for $n = 2^k$

In general we can factor the nth Fourier matrix

$$F_n = (\xi^{kj})$$

into

$$F_n = B_n \begin{bmatrix} F_m & 0 \\ 0 & F_m \end{bmatrix} P_n$$

where B_n is the *butterfly matrix*

$$B_n = \begin{bmatrix} I_m & \Theta_m \\ I_m & -\Theta_m \end{bmatrix}.$$

I_m is the $m \times m$ identity matrix, and

$$\Theta = \operatorname{diag}(1, \xi_n^1, \xi_n^2, \cdots, \xi_n^{(m-1)}).$$

P_n is the permutation matrix that orders all of the even numbered entries first.

We can continue to factor the matrices. Two successive factorisations would look like

$$F_n = B_n \begin{bmatrix} B_{n/2} & 0 \\ 0 & B_{n/2} \end{bmatrix} \cdot \begin{bmatrix} F_{n/4} & 0 & 0 & 0 \\ 0 & F_{n/4} & 0 & 0 \\ 0 & 0 & F_{n/4} & 0 \\ 0 & 0 & 0 & F_{n/4} \end{bmatrix} \begin{bmatrix} P_{n/2} & 0 \\ 0 & P_{n/2} \end{bmatrix} P_n.$$