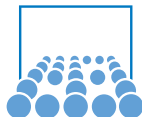


# Algorithms of Scientific Computing

## Fast Fourier Transform (FFT)

Tobias Neckel, Dirk Pflüger

Summer Term 2011



# The Pair DFT/IDFT as Matrix-Vector Product

DFT and IDFT may be computed in the form

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \omega_N^{-nk} \quad f_n = \sum_{k=0}^{N-1} F_k \omega_N^{nk}$$

or as matrix-vector products

$$\mathbf{F} = \frac{1}{N} \mathbf{W}^H \mathbf{f}, \quad \mathbf{f} = \mathbf{W} \mathbf{F},$$

with a **complexity of**  $\mathcal{O}(N^2)$ .

Note that

$$\text{DFT}(f) = \frac{1}{N} \overline{\text{IDFT}(\bar{f})}.$$

A fast computation is possible via the **divide-and-conquer** approach.

# Fast Fourier Transform for $N = 2^p$

**Basic idea:** sum up even and odd indices separately in IDFT:  
first for  $n = 0, 1, \dots, \frac{N}{2} - 1$  :

$$x_n = \sum_{k=0}^{N-1} X_k \omega_N^{nk} = \sum_{k=0}^{\frac{N}{2}-1} \left( X_{2k} \omega_N^{2nk} + X_{2k+1} \omega_N^{(2k+1)n} \right) .$$

We set  $Y_k := X_{2k}$  and  $Z_k := X_{2k+1}$ , use  $\omega_N^{2nk} = \omega_{N/2}^{nk}$ , and get a sum of two IDFT on  $\frac{N}{2}$  data:

$$x_n = \sum_{k=0}^{N-1} X_k \omega_N^{nk} = \underbrace{\sum_{k=0}^{\frac{N}{2}-1} Y_k \omega_{N/2}^{nk}}_{:= y_n} + \omega_N^n \underbrace{\sum_{k=0}^{\frac{N}{2}-1} Z_k \omega_{N/2}^{nk}}_{:= z_n} .$$

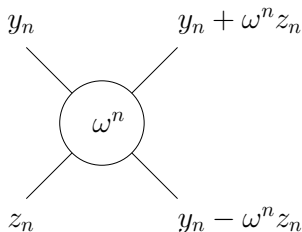
# Fast Fourier Transform (FFT)

Analogue separation even vs. odd for indices  $\frac{N}{2}, \dots, N-1$  :

$$x_{n+\frac{N}{2}} = y_{n+\frac{N}{2}} + \omega_N^{(n+\frac{N}{2})} z_{n+\frac{N}{2}}$$

Since  $\omega_N^{(n+\frac{N}{2})} = -\omega_N^n$  and  $y_n$  and  $z_n$  have a period of  $\frac{N}{2}$ , we obtain the so-called **butterfly scheme**:

$$\begin{aligned} x_n &= y_n + \omega_N^n z_n \\ x_{n+\frac{N}{2}} &= y_n - \omega_N^n z_n \end{aligned}$$



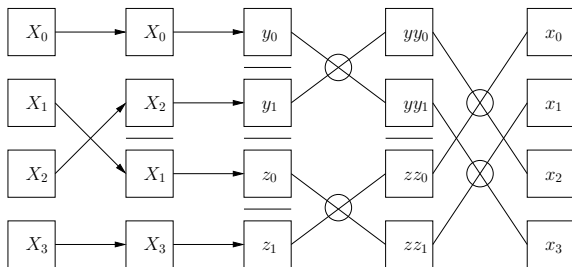
# Fast Fourier Transform – Butterfly Scheme

$$(x_0, x_1, \dots, x_{N-1}) = \text{IDFT}(X_0, X_1, \dots, X_{N-1})$$

$$\Downarrow$$

$$(y_0, y_1, \dots, y_{\frac{N}{2}-1}) = \text{IDFT}(X_0, X_2, \dots, X_{N-2})$$

$$(z_0, z_1, \dots, z_{\frac{N}{2}-1}) = \text{IDFT}(X_1, X_3, \dots, X_{N-1})$$



# Recursive Implementation of the FFT

$\text{rekFFT}(\mathbf{X}) \rightarrow \mathbf{x}$

(1) Generate vectors  $\mathbf{Y}$  and  $\mathbf{Z}$ :

$$\text{for } n = 0, \dots, \frac{N}{2} - 1: \quad Y_n := X_{2n} \quad \text{und} \quad Z_n := X_{2n+1}$$

(2) compute 2 FFTs of half length:

$$\text{rekFFT}(\mathbf{Y}) \rightarrow \mathbf{y} \quad \text{and} \quad \text{rekFFT}(\mathbf{Z}) \rightarrow \mathbf{z}$$

(3) combine with “butterfly scheme”:

$$\text{for } k = 0, \dots, \frac{N}{2} - 1: \quad \begin{cases} X_k &= y_k + \omega_N^k Z_k \\ X_{k+\frac{N}{2}} &= y_k - \omega_N^k Z_k \end{cases}$$

# Observations on the Recursive FFT

- Computational effort  $C(N)$  ( $N = 2^p$ ) given by recursion equation

$$C(N) = \begin{cases} \mathcal{O}(1) & \text{für } N = 1 \\ \mathcal{O}(N) + 2C\left(\frac{N}{2}\right) & \text{für } N > 1 \end{cases} \Rightarrow C(N) = \mathcal{O}(N \log N)$$

- Algorithm splits up in 2 phases:
    - resorting of input data
    - combination following the “butterfly scheme”
- ⇒ Anticipation of the resorting enables a simple, iterative algorithm without additional memory requirements.

# Sorting Phase of the FFT – Bit Reversal

## Observation:

- even indices are sorted into the upper, odd indices into the lower half.
  - distinction even/odd based on least significant bit
  - distinction upper/lower based on most significant bit
- ⇒ An index in the sorted field has the **reversed** (i.e. mirrored) binary representation compared to the original index.



# Sorting of a Vector ( $N = 2^p$ Entries, Bit Reversal)

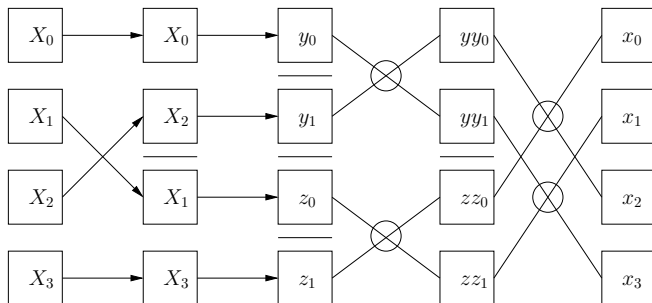
**Java-Code:** (data in field  $X$ )

```
for(int n=0; n<N; n++) {
    // Compute p-valued bit reversal of n in j
    int j=0; int m=n;
    for(int i=0; i<p; i++) {
        j = 2*j + m%2; m = m/2;
    }
    // if j>n exchange X[j] and X[n]
    if (j>n) { double h=X[j]; X[j] = X[n]; X[n] = h; }
}
```

Bit reversal needs  $\mathcal{O}(p) = \mathcal{O}(\log N)$  operations

- ⇒ Sorting results also in a complexity of  $\mathcal{O}(N \log N)$
- ⇒ Sorting may consume up to 10–30% of the CPU time!

# Iterative Implementation of the “Butterflies”



# Iterative Implementation of the “Butterflies”

```

{Loop over the size of the IDFT}
for(int L=2; L<=N; L*=2)
    {Loop over the IDFT of one level}
    for(int k=0; k<N; k+=L)
        {perform all butterflies of one level}
        for(int j=0; j<L/2; j++) {
            {complex computation:}
            z ←  $\omega_L^j$  * X[k+j+L/2]
            X[k+j+L/2] ← X[k+j] - z
            X[k+j] ← X[k+j] + z
        }
    }

```

- k-loop und j-loop are “commutable”!
- How and when are the  $\omega_L^j$  computed?

# Iterative Implementation – Variant 1

```
for(int L=2; L<=N; L*=2)
  for(int k=0; k<N; k+=L)
    for(int j=0; j<L/2; j++) {
      z ←  $\omega_L^j$  * X[k+j+L/2]
      X[k+j+L/2] ← X[k+j] - z
      X[k+j] ← X[k+j] + z
    }
```

**Advantage:** consecutive access of data in field X

- ⇒ good cache performance
- ⇒ suitable for vector computers

**Disadvantage:** multiple computations of  $\omega_L^j$

## Iterative Implementation – Variant 2

```
for(int L=2; L<=N; L*=2)
  for(int j=0; j<L/2; j++) {
    w ←  $\omega_L^j$ 
    for(int k=0; k<N; k+=L) {
      z ← w * X[k+j+L/2]
      X[k+j+L/2] ← X[k+j] - z
      X[k+j] ← X[k+j] + z
    }
  }
```

**Advantage:** each  $\omega_L^j$  only computed once

**Disadvantage:** “stride-2”-access of the field X

⇒ worse cache performance

⇒ not suitable for vector computers

## Separate Computation of $\omega_L^j$

- necessary:  $N - 1$  factors

$$\omega_2^0, \omega_4^0, \omega_4^1, \dots, \omega_L^0, \dots, \omega_L^{L/2-1}, \dots, \omega_N^0, \dots, \omega_N^{N/2-1}$$

- are computed beforehand and stored in a field  $w$ , e.g.:

```
for(int L=2; L<=N; L*=2)
  for(int j=0; j<L/2; j++)
    w[L-j-1] ←  $\omega_L^j$ ;
```

- Variant 2: access on  $w$  in sequential order
- Variant 1: access on  $w$  local (but repeated)