Algorithms of Scientific Computing

Fast Fourier Transform (FFT)

Michael Bader

Summer Term 2012
The Pair DFT/IDFT as Matrix-Vector Product

DFT and IDFT may be computed in the form

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

or as matrix-vector products

\[ F = \frac{1}{N} W^H f, \quad f = W F, \]

with a complexity of \( O(N^2) \).

Further result (req. separate computation):

\[ \text{DFT}(f) = \frac{1}{N} \text{IDFT}(f). \]

A fast computation is possible via the \textit{divide-and-conquer} approach.
Fast Fourier Transform for \( N = 2^p \)

Basic idea: sum up even and odd indices separately in IDFT:

first for \( n = 0, 1, \ldots, \frac{N}{2} - 1 \):

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

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 \( \frac{N}{2} \) coefficients:

\[
x_n = \sum_{k=0}^{N-1} X_k \omega_n^{nk} = \sum_{k=0}^{\frac{N}{2}-1} Y_k \omega_{N/2}^{nk} + \omega_n^n \sum_{k=0}^{\frac{N}{2}-1} Z_k \omega_{N/2}^{nk}.
\]

\[
\begin{align*}
&\underbrace{Y_n}_{:= y_n} + \omega_n^n \underbrace{Z_n}_{:= z_n}.
\end{align*}
\]
Fast Fourier Transform (FFT)

Do the same even vs. odd separation for indices $\frac{N}{2}, \ldots, N - 1$:

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

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

$$x_n = y_n + \omega^N_n z_n$$
$$x_{n + \frac{N}{2}} = y_n - \omega^N_n z_n$$
Fast Fourier Transform – Butterfly Scheme

\[(x_0, x_1, \ldots, x_{N-1}) = \text{IDFT}(X_0, X_1, \ldots, X_{N-1})\]
\[\downarrow\]
\[(y_0, y_1, \ldots, y_{N/2-1}) = \text{IDFT}(X_0, X_2, \ldots, X_{N-2})\]
\[(z_0, z_1, \ldots, z_{N/2-1}) = \text{IDFT}(X_1, X_3, \ldots, X_{N-1})\]
Recursive Implementation of the FFT

rekFFT(\(X\)) \rightarrow x

(1) Generate vectors \(Y\) and \(Z\):

for \(n = 0, \ldots, \frac{N}{2} - 1\):

\[Y_n := X_{2n}\text{ und } Z_n := X_{2n + 1}\]

(2) compute 2 FFTs of half size:

rekFFT(\(Y\)) \rightarrow y and rekFFT(\(Z\)) \rightarrow z

(3) combine with “butterfly scheme”:

for \(k = 0, \ldots, \frac{N}{2} - 1\):

\[
\begin{align*}
x_k & = y_k + \omega_N^k z_k \\
x_{k + \frac{N}{2}} & = y_k - \omega_N^k z_k
\end{align*}
\]
Observations on the Recursive FFT

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

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

- Algorithm splits up in 2 phases:
  - resorting of input data
  - combination following the “butterfly scheme”

$\Rightarrow$ Anticipation of the resorting enables a simple, iterative algorithm without additional memory requirements.
Observation:

- even indices are sorted into the upper half, 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 \textit{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 $X$)

```java
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 $O(p) = O(\log N)$ operations

⇒ Sorting results also in a complexity of $O(N \log N)$
⇒ Sorting may consume up to 10–30\% of the CPU time!
Iterative Implementation of the “Butterflies”
Iterative Implementation of the “Butterflies”

\[
\{\text{Loop over the size of the IDFT}\}
\]

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

- \text{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^j_L \cdot X[k+j+L/2]
            X[k+j+L/2] ← X[k+j] - z
            X[k+j] ← X[k+j] + z
        }

**Advantage:** consecutive access to data in field \(X\)
⇒ good cache performance
⇒ suitable for vector computers

**Disadvantage:** multiple computations of \(\omega^j_L\)
Iterative Implementation – Variant 2

for(int L=2; L<=N; L*=2) 
    for(int j=0; j<L/2; j++) { 
        w ← ω^j_L 
        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 ω^j_L only computed once 

**Disadvantage:** “stride-L”-access to the array X 
⇒ worse cache performance 
⇒ not suitable for vector computers
Separate Computation of $\omega^j_L$

- necessary: $N - 1$ factors

\[ \omega_0^0, \omega_0^2, \omega_0^4, \ldots, \omega_L^0, \ldots, \omega_L^{L/2-1}, \ldots, \omega_N^0, \ldots, \omega_N^{N/2-1} \]

- are computed in advance, and stored in an array $w$, e.g.:

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

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