Algorithms of Scientific Computing

Fast Fourier Transform (FFT)

Michael Bader
Summer Term 2014
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

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

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

Note that

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

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

$\rightarrow$ 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 $\omega_N^{2nk} = \omega_N^{nk}$, and get a sum of two IDFT on $\frac{N}{2}$ coefficients:

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

Note: this formula is actually valid for all $n = 0, \ldots, N - 1$; however, the IDFTs of size $\frac{N}{2}$ will only deliver the $y_n$ and $z_n$ for $n = 0, \ldots, \frac{N}{2} - 1$ (but: $y_n$ and $z_n$ are periodic!)
Fast Fourier Transform (FFT)

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

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

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:

\[
x_n = y_n + \omega_N^n z_n
\]

\[
x_{n + \frac{N}{2}} = y_n - \omega_N^n z_n
\]
Fast Fourier Transform – Butterfly Scheme

\[(x_0, x_1, \ldots, x_{N-1}) = \text{IDFT}(X_0, X_1, \ldots, X_{N-1})\]

\[\downarrow\]

\[(y_0, y_1, \ldots, y_{N/2-1}) = \text{IDFT}(X_0, X_2, \ldots, X_{N-2})\]

\[(z_0, z_1, \ldots, z_{N/2-1}) = \text{IDFT}(X_1, X_3, \ldots, X_{N-1})\]
Fast Fourier Transform – Butterfly Scheme (2)

The diagram illustrates the butterfly scheme for the Fast Fourier Transform (FFT). Each step in the scheme involves a pair of inputs (X0, X1) and produces a pair of outputs (X0, X1). The arrows represent the flow of data through the butterfly stages, with each stage using a twiddle factor (W) to compute the transform.

- The top row shows the stages of the FFT, with each stage involving a pair of inputs and producing a pair of outputs.
- The diagram is color-coded to highlight the different stages and their corresponding outputs.
- The butterfly scheme is recursive, with each stage consisting of multiple butterfly operations.

This diagram is a visual representation of the FFT algorithm, which is widely used in digital signal processing and other fields requiring efficient computation of discrete Fourier transforms.
Recursive Implementation of the FFT

\textbf{rekFFT}(X) \rightarrow x

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

for \( n = 0, \ldots, \frac{N}{2} - 1 \) :
\[
Y_n := X_{2n} \quad \text{und} \quad Z_n := X_{2n+1}
\]

\textbf{(2)} compute 2 FFTs of half size:

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

\textbf{(3)} combine with “butterfly scheme”:

for \( k = 0, \ldots, \frac{N}{2} - 1 \) :
\[
\begin{cases}
  x_k &= y_k + \omega_N^k Z_k \\
  x_{k + \frac{N}{2}} &= y_k - \omega_N^k Z_k
\end{cases}
\]
Observations on the Recursive FFT

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

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

- 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 reversed (i.e. mirrored) binary representation compared to the original index.
Sorting of a Vector ($N = 2^p$ Entries, Bit Reversal)

/** FFT sorting phase: reorder data in array X */
for(int n=0; n<N; n++) {
    // Compute p−bit 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) {
        complex<double> h;
        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”

```c
// 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:
            complex<
double>
                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

```c
/** FFT butterfly phase: 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++) {
            complex<double> 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 to data in field $X$

⇒ suitable for vectorisation

⇒ good cache performance due to prefetching (stream access) and usage of cache lines

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

/** FFT butterfly phase: variant 2 */

for (int L=2; L<=N; L*=2) 
  for (int j=0; j<L/2; j++) {
    complex<double> w = omega(L,j);
    for (int k=0; k<N; k+=L) {
      complex<double> 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 \( \omega_j^L \) only computed once

**Disadvantage:** “stride-L”-access to the array \( x \)
  
  ⇒ worse cache performance (inefficient use of cache lines)
  ⇒ not suitable for vectorisation
L-Oriented Implementation – Illustration

\[X_0 \rightarrow X_0 \rightarrow X_0 \rightarrow X^{(0)}_0 \rightarrow X^{(0,4)}_0 \rightarrow X^{(0-6)}_0 \rightarrow X_0\]

\[X_1 \rightarrow X_2 \rightarrow X_4 \rightarrow X^{(4)}_4 \rightarrow X^{(0,4)}_4 \rightarrow X^{(0-6)}_4 \rightarrow X_1\]

\[X_2 \rightarrow X_4 \rightarrow X_2 \rightarrow X^{(2)}_2 \rightarrow X^{(2,6)}_2 \rightarrow X^{(0-6)}_2 \rightarrow X_2\]

\[X_3 \rightarrow X_6 \rightarrow X_6 \rightarrow X^{(6)}_6 \rightarrow X^{(2,6)}_6 \rightarrow X^{(0-6)}_6 \rightarrow X_3\]

\[X_4 \rightarrow X_1 \rightarrow X_1 \rightarrow X^{(1)}_1 \rightarrow X^{(1,5)}_1 \rightarrow X^{(1-7)}_1 \rightarrow X_4\]

\[X_5 \rightarrow X_3 \rightarrow X_5 \rightarrow X^{(5)}_5 \rightarrow X^{(1,5)}_5 \rightarrow X^{(1-7)}_5 \rightarrow X_5\]

\[X_6 \rightarrow X_5 \rightarrow X_3 \rightarrow X^{(3)}_3 \rightarrow X^{(3,7)}_3 \rightarrow X^{(1-7)}_3 \rightarrow X_6\]

\[X_7 \rightarrow X_7 \rightarrow X_7 \rightarrow X^{(7)}_7 \rightarrow X^{(3,7)}_7 \rightarrow X^{(1-7)}_7 \rightarrow X_7\]
Separate Computation of $\omega_L^j$

- necessary: $N - 1$ factors
  
  $\omega_2^0, \omega_4^0, \omega_4^1, \ldots, \omega_L^0, \ldots, \omega_L^{L/2-1}, \ldots, \omega_N^0, \ldots, \omega_N^{N/2-1}$

- are computed in advance, and stored in an array $w$, e.g.:
  
  ```
  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)
Cache Efficiency – Variant 1 Revisited

`/** FFT butterfly phase: variant 1 */`

```c
for (int L=2; L<=N; L*=2)
    for (int k=0; k<N; k+=L)
        for (int j=0; j<L/2; j++) {
            complex< double > z = w[L−j−1] * X[k+j+L/2];
            X[k+j+L/2] = X[k+j] − z;
            X[k+j] = X[k+j] + z;
        }
```

**Observation:**

- each L-loop traverses entire array X
- in the ideal case \((N \log N)/B\) cache line transfers
  
  \((B\) the size of the cache line)

**Compare with recursive scheme:**

- if \(L < M\) (\(M\) the cache size), entire FFT of size \(L\) could be computed in cache
- ideal case then only \(L/(MB)\) cache line transfers
Butterfly Phase with Loop Blocking

```c
/** FFT butterfly phase: loop blocking for k */
for(int L=2; L<=N; L*=2)
    for(int kb=0; kb<N; kb+=M)
        for(int k=kb; k<kb+M; k+=L)
            for(int j=0; j<L/2; j++) {
                complex<double> z = w[L−j−1] * X[k+j+L/2];
                X[k+j+L/2] = X[k+j] − z;
                X[k+j] = X[k+j] + z;
            }
```

**Question:** can we make the L-loop an inner loop?

- kb-loop and L-loop may be swapped, if $M > L$
- however, we assumed that $N > M$ (“data does not fit into cache”)
- we thus need to split the L-loop into a phase $L=2..M$ (in cache) and a phase $L=2*..N$ (out of cache)
Butterfly Phase with Loop Blocking (2)

/** perform all butterfly phases of size \(L \leq M\) */
for(int kb=0; kb<N; kb+=M)
  for(int L=2; L<=M; L*=2)
    for(int k=kb; k<kb+M; k+=L)
      for(int j=0; j<L/2; j++) {
        complex<double> z = w[L−j−1] * X[k+j+L/2];
        X[k+j+L/2] = X[k+j] − z;
        X[k+j] = X[k+j] + z;
      }

/** perform remaining butterfly levels of size \(L>M\) */
for(int L=2*M; L<=N; L*=2)
  for(int k=0; k<N; k+=L)
    for(int j=0; j<L/2; j++) {
      complex<double> z = w[L−j−1] * X[k+j+L/2];
      X[k+j+L/2] = X[k+j] − z;
      X[k+j] = X[k+j] + z;
    }
Loop Blocking and Recursion – Illustration