Fast Fourier Transforms

Steffen Maass

February 2023

Abstract

Fast implementations of discrete Fourier transforms are needed in most scientific software. The best algorithms are highly optimized for the specific hardware but are, consequently, more difficult to maintain. We implemented high-level code in modern C++using only standard compilers for the hardware-specific optimization. The source code for the project is hosted in a repository on `GitHub` at `https://github.com/smaasz/scicomp-project/`.

1 Introduction

The Fast Fourier Transform (FFT) is an algorithm to compute the Discrete Fourier Transform (DFT) of a finite, uniformly sampled signal.

In general, *Fourier analysis* studies periodic functions by representing them as sums of trigonometric functions of the same period (*harmonics*) called *Fourier series*. The summands are called the *frequency components* of the function. By calculating the Fourier transform of a sound signal for example, the frequency spectrum of the oscillating sound origin can be determined.

Originally, the Fourier transform was used by *Joseph Fourier* in 1807 to study solutions of the heat equation that models the diffusion of heat by a partial differential equation[1]. Nowadays, applications of Fourier analysis are ubiquitous in almost every area of science. A particularly prominent role does Fourier analysis play in signal processing where the decomposition of periodic signals into frequency components simplifies further processing.

If a function is uniformly sampled over some interval in space or time, a sequence of data points is obtained. To study the frequency components of the original function, a discrete version of the Fourier transform is used. This discrete Fourier transform is mathematically equivalent to computing a unitary change of basis in the complex vector space where the data points fit. In 1965, *James Cooley* and *John Tukey* independently formulated algorithms for computing the discrete Fourier transform of a signal that is faster than the obvious unitary change of basis calculation by matrix-vector multiplication. This class of algorithms called Fast Fourier Transform (FFT) splits the DFT into smaller DFTs that can be solved recursively and fitted back together by another DFT.

The details of implementing an optimized FFT algorithm are highly dependent on the computer's hardware as memory accesses mostly dominate the performance of the algorithms. A major obstacle is the necessity for discontiguous memory accesses in Cooley-Tukey algorithms. Highly optimized FFT codes therefore try to improve the temporal and spatial locality to use the most out

[1]Trigonometric series were already used earlier e.g. in astronomy.

of the hardware's memory technologies such as caching, prefetching and pipelining. A particularly wide-spread implementation is called Fastest Fourier Transform in the West (FFTW) by Matteo Frigo and Steven G. Johnson.

The second section provides the necessary mathematical background and formally introduces the DFT. Next, the idea of FFTs as developed by Cooley and Tukey is outlined. In the fourth section we document our implementation of two FFT algorithms and highlight the parameters of the implementations. The fifth section includes numerical experiments that analyze the parameters of the implementation and compare our implementations to FFTW with respect to their runtime performances. Finally, we outline some details of FFTW and aim to explain their superior performance.

2 Mathematical Background

Fourier analysis provides the rigorous setting when a complex-valued, periodic function $f : [0,1] \to \mathbf{C}$ can be written as a convergent series of harmonic functions

$$\sum_{n\in\mathbf{Z}} f_k \exp(2\pi i k x). \tag{1}$$

In this case the series is called the *Fourier series* of the function f and the coefficients $f_k \in \mathbf{C}$ are called the *Fourier coefficients* of f. The Fourier coefficients can be calculated by the explicit formula

$$f_k = \int_0^1 f(x)\exp(-2\pi i k x)\mathrm{d}x$$

for $k \in \mathbf{Z}$.

In the case that only a finite sequence of data points $(y_0, y_1, \ldots, y_{n-1}) \in \mathbf{C}^n$ that are uniformly sampled (step size $1/n$) of the function is available, the Fourier coefficients are approximated by

$$f_k \approx \frac{1}{n}\sum_{j=0}^{n-1} y_j \exp(-2\pi i/nkj) = \frac{1}{n}\sum_{j=0}^{n-1} y_j \omega_n^{-k\cdot j} \tag{2}$$

where $\omega_n = \exp(2\pi i/n)$ is a primitive root of unity.

Care has to be taken in the choice of n also called the *sampling rate*. If the initial function f has frequency components higher than $n/2$, these frequency components cannot be resolved by sampling at a rate of n because $k' \equiv k \mod n$ implies

$$\omega_n^{k'\cdot j} = \omega_n^{k\cdot j} \qquad \text{for all } j = 0, 1, \ldots, n-1.$$

This effect is called *aliasing* and $n/2$ is the *Nyquist frequency.*

To summarize, when sampling at a rate of n it is assumed that the function has no frequency components with frequencies greater or equal than the Nyquist frequency. In this case the Fourier coefficients for $k = -n/2+1, \ldots, -1, 0, 1, \ldots, n/2$ if n is even or $-(n-1)/2, \ldots, -1, 0, 1, \ldots, (n-1)/2$ if n is odd are approximated by Equation 2. The map from $(y_j)_{j=0,1,\ldots,n-1}$ to the approximated Fourier coefficients is an one-to-one correspondence and is called the Discrete Fourier Transform of the signal.

The DFT is a linear map DFT : $\mathbf{C}^n \to \mathbf{C}^n$. Represented in the standard basis for n even it takes the form:

$$\begin{bmatrix} \hat{f}_0 \\ \hat{f}_1 \\ \vdots \\ \hat{f}_{n/2} \\ \hat{f}_{-n/2+1} \\ \vdots \\ \hat{f}_{-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega_n^{-1\cdot 1} & \dots & \omega_n^{-1\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{-n/2\cdot 1} & \dots & \omega_n^{-n/2\cdot(n-1)} \\ 1 & \omega_n^{-(n/2+1)\cdot 1} & \dots & \omega_n^{-(n/2+1)\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{-(n-1)\cdot 1} & \dots & \omega_n^{-(n-1)\cdot(n-1)} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n/2} \\ y_{n/2+1} \\ \vdots \\ y_{n-1} \end{bmatrix}. \tag{3}$$

Note again that $\omega_n^{-(n/2+1)} = \omega_n^{-(-n/2+1)}, \dots, \omega_n^{-(n-1)} = \omega_n^{-(-1)}$. The case when n is odd is analogous. This linear map can also be interpreted as a change of basis from the standard basis to the orthogonal basis defined by

$$\phi_k = \begin{bmatrix} 1 & \omega_n^{(k-1)\cdot 1} & \omega_n^{(k-1)\cdot 2} & \dots & \omega_n^{(k-1)\cdot(n-1)}. \end{bmatrix}^T$$

Notice that the entries of the $k+1$th basis vector are exactly the values of the pure harmonic $\exp(2\pi\ k \cdot i/n)$ evaluated at the sampled points i/n for $i = 0, 1, \dots, n-1$.

3 Fast Fourier Transforms

Algorithms to calculate DFTs quickly were rediscovered several times in history but they finally became public knowledge with the advent of digital computers. The first scientists to use fast Fourier transforms on digital computers were *James Cooley* and *John Tukey* in 1965.

They noted that DFTs can be solved recursively if the size n is a composite $n = mr$. For $j, k \in \{0, 1, \dots, n-1\}$ write $j = j_1 r + j_0$ with $j_0 \in \{0, 1, \dots, r-1\}$ and $j_1 \in \{0, 1, \dots, m-1\}$ and $k = k_1 m + k_0$ with $k_0 \in \{0, 1, \dots, m-1\}$ and $k_1 \in \{0, 1, \dots, r-1\}$. Then

$$\begin{aligned} \mathrm{DFT}(y)[k] = \hat{f}[k] &= \sum_{j=0}^{n-1} y[j]\omega_n^{-k\cdot j} \\ &= \sum_{j_0=0}^{r} \Bigg[\underbrace{\left(\sum_{j_1=0} y[j_1 r + j_0]\omega_m^{-k_0\cdot j_1} \right)}_{\mathrm{DFT}(y[j_0\ :\ r\ :\ j_0+(m-1)r])[k_0]=:v[k_0,j_0]} \omega_n^{-k_0\cdot j_0} \Bigg] \omega_r^{-k_1\cdot j_0} \\ &= \mathrm{DFT}(v[k_0,0]\omega_n^{-k_0\cdot 0}, v[k_0,1]\omega_n^{-k_0\cdot 1}, \dots, v[k_0,r-1]\omega_n^{-k_0\cdot(r-1)})[k_1] \end{aligned}$$

where we replaced most subscripts by the indexing operator more common in programming languages. The factors of the v's are also called *twiddle factors*.

A procedural understanding of the idea is as follows (see Figure 3):

1. Interpret the data vector y as a $m \times r$-matrix in *row-major format*.
2. To each of the r columns apply a DFT of length m. The resulting matrix will be denoted by v.

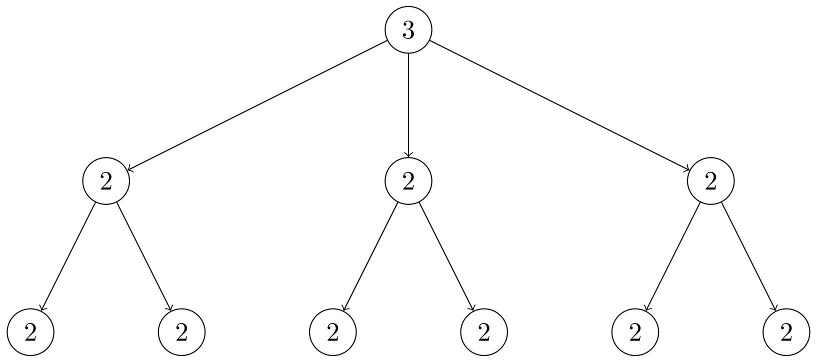


Figure 1: Recursion tree for a DFT of size $n = 2 \cdot 2 \cdot 3$

3. Multiply each entry of v by its corresponding twiddle factor.
4. To each of the m rows apply a DFT of length r.
5. Reinterpreting the $m \times r$-matrix in *column-major format* is the data vector $\hat{f}$.

If either the inner or outer DFTs are solved recursively, the procedure is called decimation in time (DIT) or decimation in frequency (DIF), respectively. Here only DIT will be considered. In this case the outer DFT is called *butterfly* and the value r is called *radix*. Suppose that $n = r_p \cdot \ldots \cdot r_1 r_1$. The recursion yields a rooted tree of DFTs where the nodes at level $i < p$ represent the $r_p \cdot \cdots \cdot r_{i+1}$ butterfly DFTs of size r_i. The leaves represent the *base case* DFTs of size r_p. Note that at level i there are $r_{i-1} \cdot \ldots \cdot r_1$ nodes and each node has r_i children (see Figure 1).

4 Implementation of a Fast Fourier Transform

We implemented two versions of FFTs in `C++`. FFTs are commonly implemented for sampled data of length being a power-of-2 (2^k for $k > 0$) as this case can be very efficiently implemented. Such implementations are called *radix-2* FFT. In this case care has to be taken that the number of samples in the measurement are a power-of-2. Otherwise, the data can also be *zero-padded* to a power-of-2 that is artificial zeros are appended to the actual measurements.

To retain the flexibility of using the measurements without the need of post-processing and to implement algorithms that cannot be found in all textbooks, we decided to implement *mixed-radix* FFTs where the radices in different (recursion) steps need not be 2 nor do they have to be the same. In this case an initialization step to choose appropriate radices for a given input size n is needed. We implemented three *radix generation* strategies:

1. Radices are chosen to be the prime factors of n in increasing order.
2. Radices are chosen to be the prime factors of n in decreasing order.
3. Group prime factors of n as long as their product does not exceed a chosen threshold.

The two implemented versions can be characterized as *depth-first* and *breadth-first*, respectively (see recursion tree e.g. Example 1). The former case implements the Cooley-Tukey algorithm starting with the DFT on the full data and uses the *divide-and-conquer* approach to first solve the DFT of each column (see step 3) before moving to the DFT of the next column. In contrast the breadth-first version computes all leaves of the recursion tree first and works its way up to the root level by level. Both implementations are templates in order to use different complex data types (e.g. `std::complex<float>` and `std::complex<double>`). The implementation (without the testing interface) was written in about 500 lines.

Note that the calculation of each butterfly can be interpreted as an evaluation of a polynomial in $\omega_n^{-k_0}\omega_r^{-k_1}$ in the notation above. Thus, we implemented the evaluation using *Horner's method* and evaluated $\omega_n^{-k_0}$ and $\omega_r^{-k_1}$ in `long double` precision using the function `std::polar` from the `complex` standard library.

4.1 Accuracy

To obtain a very rough bound on the accuracy of the algorithm assume that the input is scaled such that all values will be in absolute value less than 1. Observe that for the calculation of each Fourier coefficient in a DFT of length n requires $2n$ complex floating-point operations. Neglecting the error in the phase and twiddle factors because they are computed with `long double` precision, every such complex floating-point operation (addition and multiplication by a phase/twiddle factor) introduces an additional absolute error of at most $\epsilon = |\epsilon_{\text{machine}} + i\epsilon_{\text{machine}}| = 2^{1/2-m}$ where m is the precision of the data type i.e. 24 for single and 53 for double precision. Given the assumptions the error of a Fourier coefficient is at most $n2^{1/2-m}$ so that at most $\log_{10} n$ digits of accuracy are lost.

4.2 Runtime Complexity

Neglecting the calculation of the phase and twiddle factors, the base case evaluation (e.g. in the butterfly) of a DFT of length $r \geq 2$ using Horner's scheme uses $2r^2$ floating point operations. The runtime complexity $\ell(n)$ of the mixed-radix Cooley-Tukey algorithm for a DFT of length $n = r_p \cdot \ldots \cdot r_2 \cdot r_1$ with $r_i \geq 2$ satisfies the recursive relation:

$$\begin{aligned} \ell(r_p) &= 2r_p^2 \\ \ell\left(r_p \cdot \ldots \cdot r_{i+1} \cdot r_i\right) &= \frac{n}{r_i \cdot \ldots \cdot r_2 \cdot r_1} \cdot 2r_i^2 + r_i \ell\left(r_p \cdot \ldots \cdot r_{i+1}\right) \end{aligned}$$

which is easily seen to be solved by

$$\ell(n) = 2n(r_1 + r_2 + \ldots + r_p) = 2np\frac{r_1 + r_2 + \ldots + r_p}{p} \geq 4n \log_2 n \tag{4}$$

where the last inequality is strict if and only if $n = 2^p$.

The proof of the first equality follows by induction on the number of factors. The last inequality can be proven using the arithmetic mean-geometric mean inequality:

$$\frac{r_1 + r_2 + \ldots + r_p}{p} \geq (r_1 \cdot r_2 \cdot \ldots \cdot r_p)^{1/p} = n^{1/p} =: r$$

size	time (ms)	accuracy (max-norm)	accuracy (two-norm)
840	0.25	0.000023016964	0.000096100957
2145	0.63	0.000122070305	0.000454294670
1776	0.69	0.000097934695	0.000658761011
27000	12.32	0.000976562442	0.003745999187

Table 1: Accuracy test of the iterative method, single precision

size	time (ms)	accuracy (max-norm)	accuracy (two-norm)
840	0.27	0.000000000000	0.000000000000
2145	0.70	0.000000000000	0.000000000001
1776	0.77	0.000000000000	0.000000000001
27000	14.24	0.000000000005	0.000000000019

Table 2: Accuracy test of the iterative method, double precision

which is strict if and only if $r_1 = r_2 = \ldots = r_p = r \geq 2$. Note that $p = \log_r n$. Therefore

$$\begin{aligned}\ell(n) &= 2n(r_1 + r_2 + \ldots + r_p) = 2np\frac{r_1 + r_2 + \ldots + r_p}{p} \\ &\geq 2npr = 2nr\log_r p \\ &= 2n\log_2 n\frac{r}{\log_2 r} \\ &\geq 4n\log_2 n\end{aligned}$$

where the last inequality is strict for $r \geq 2$ if and only if $r = 2$.

Finally, it is easy to observe that $\ell(n) \leq 2n^2$.

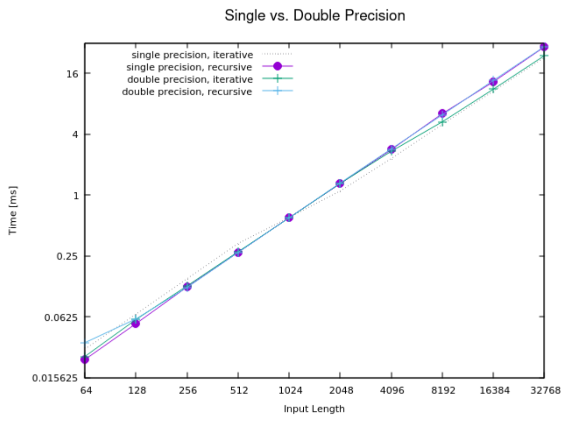
5 Numerical Experiments

The implementation was compiled with `gcc` (version 9) on a Intel Core i5-3320M with a clock rate of 2.6 GHz.

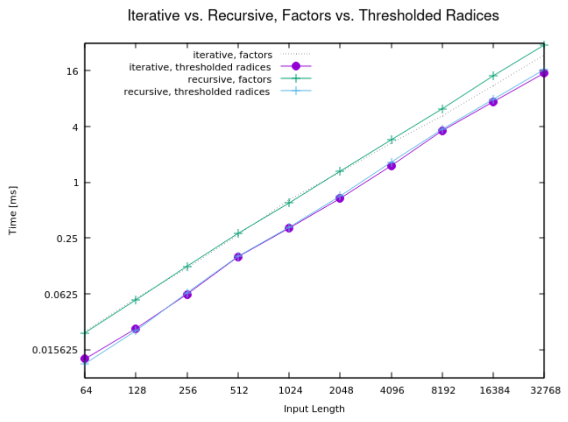
We started with testing the accuracy of our implementations. The data of the test instances is randomly generated. The results of the test for single and double precision and the iterative method are shown in tables 1 and 2 (the results for the recursive method are similar).

Next we conducted numerical experiments to analyze the performance. Since the length of the measured data is known a priory, the initialization step is not included in the timing. For the compilation we used `gcc` (version 9) with the `-Ofast` flag. We did three experiments:

1. Performance comparison of our algorithms using single vs. double precision (see Figure 2a).
2. Performance comparison of our algorithms with different radix generation strategies (see Figure 2b).
3. Performance comparison of our best performing algorithm with FFTW (see Figure 2c and 2d).

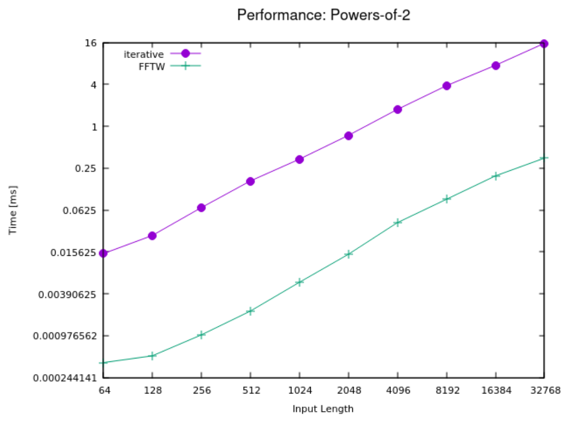


(a) Test 1

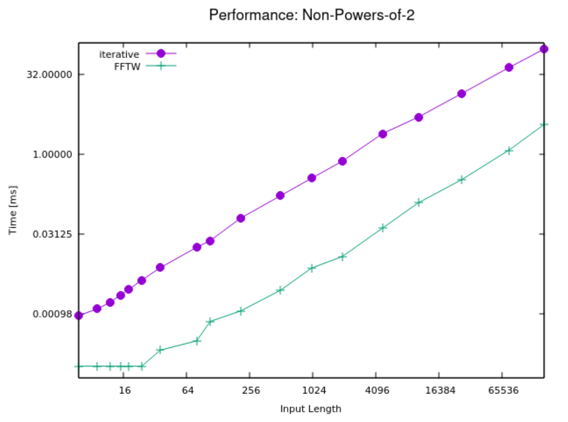


(b) Test 2

(c) and (d):



(c) Test 3: powers-of-2



(d) Test 3: non-powers-of-2

Figure 2: Visualization of the numerical experiments

The size of the tested instances was chosen in accordance with *benchFFT* (the benchmarking tool for FFTW).

From the first test we concluded that there is no difference in performance of our algorithms when using single or double precision. The second test illustrates that the iterative approach performances slightly better than recursive method. Moreover, the choice of the radix strategy (1 or 3) has a small impact on the performance: the third radix strategy with a threshold of 16 performs better for both the iterative and recursive approach. Finally, the third test shows that the FFTW implementation performs for both instances of length a power-of-2 and non-power-of-2 an order of magnitude better than our algorithm.

6 Details of the FFTW implementation

The software library FFTW (latest official version 3.3.10 as of February 2023) is one of the fastest free portable implementation of the FFT developed by Matteo Friga and Steven G. Johnson. In fact the acronym FFTW stands for "Fastest Fourier Transform in the West".

FFTW is composed of a collection of fast C routines for computing the DFT in one or more dimensions. It includes complex, real, symmetric, and parallel transforms, and can compute them in $O(n \log n)$ time for any length n including primes. FFTW is typically faster than other publicly-available FFT implementations, and is even competitive with vendor-tuned libraries.

Parallel shared-memory versions of FFTW exist for *Cilk*, a superset of C developed at MIT, and *Threads*. Starting with version 3.3, FFTW even includes a parallel distributed-memory version for *MPI*.

6.1 How it works

FFTW is used in the following way to compute a DFT:

1. The user specifies the type of problem they want to solve with FFTW. The *problem* is specified with a data structure describing the size of the array, dimensionality of the FFT, the input data type and in-place or out-of-place execution to name a few parameters.

2. The problem gets passed to the *FFTW planer*, which yields a plan, an executable data structure that accepts the input data and computes the desired DFT. This is done by measuring the the *runtime* of different plans in a subspace of all plans and selecting the fastest. The FFTW planer uses the recursive nature of the problem to find the fastest plan with a dynamic programming approach. Once the planner reaches a simple enough sub-problem, it calls a fragment of optimized code called *codelet* that solves the sub-problem directly. This entire process is **time-consuming** and will typically run **once**.

3. The plan can now be executed with data fitting the described problem. This process is **very fast** and will typically run **multiple times** with different data fitting the specified problem.

As we can see only a subset of all plans is searched in the second step. Furthermore the dynamic programming approach means that each sub-problem is optimized locally, independently of the larger context. One could possibly find a faster FFT by searching a larger space of plans for example by not using dynamic programming, but the second step would be even more time-consuming. As a trade-off the set of plans searched by FFTW must therefore be large enough to contain the fastest possible plans while being small enough to keep the planning time acceptable.

Furthermore the codelets have to be really fast, because all of the plans are composed of these. The codelets are generated by a special-purpose compiler called `genfft` from an abstract description. The codelets implement some combination of the *Cooley-Tukey algorithm* or it's *split-radix* variant as well as the *prime-factor algorithm* for co-prime factorisation of n or *Rader's algorithm* for prime n. Each algorithm is first expressed in a straightforward math-like notation, using complex numbers, before it gets simplified by certain means such as eliminating multiplications by 1, elimination of common sub-expression, precomputing the factorisation of n, unrolling and re-ordering recursions in the algorithm and breaking complex-number abstraction into real and imaginary components to name a few. The standard FFTW distribution works most efficiently for arrays whose size can be factored into small primes (2, 3, 5, and 7), but one can use `genfft` to create codelets for sizes with bigger prime factors.

6.2 The FFTW planer

We will now take a deeper look at the FFTW planner. As stated before the planner will compare the runtime of different plans and return the fastest. The plans are returned by ancillary entities called *solvers*, which are initialized by the FFTW planer. A solver is **problem-specific**, **plan-specific** and will either return a *pointer* to a plan or a *null pointer* if a plan can't be created by the solver. Each solver is specific to a subclass of plans. Here are some examples:

- The simplest type of plans are no-op plans, where nothing has to be done.
- Some plans will only perform a permutation of the input array to the output array
- Some plans solve the 1-dimensional DFT using codelets when the input has a length $n \in \{2, \ldots, 16, 32, 64\}$
- Other plans use Cooley-Tukey to split the problem into sub-problems using DIT or DIF while other plans reduce multidimensional DFT to sub-problems of lower dimensionality. All sub-problems are solved in the fastest time through a recursive call of the FFTW planer.

The Planner has to potentially evaluate the same sub-problem multiple times, because of the dynamic programming approach. **Memoization** is used to avoid unnecessary work by saving a 128-bit hash of the problem generated by MD5 together with a pointer to the solver that generated the plan. This approach avoids saving the large data structures, which describe either the problem or the plan.

References

[1] M. Frigo and S. G. Johnson. *benchFFT*. URL: `https://www.fftw.org/benchfft/` (visited on 02/16/2023).

[2] M. Frigo and S.G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231. DOI: `10.1109/JPROC.2004.840301`.

Appendix

Acronyms

FFT Fast Fourier Transform

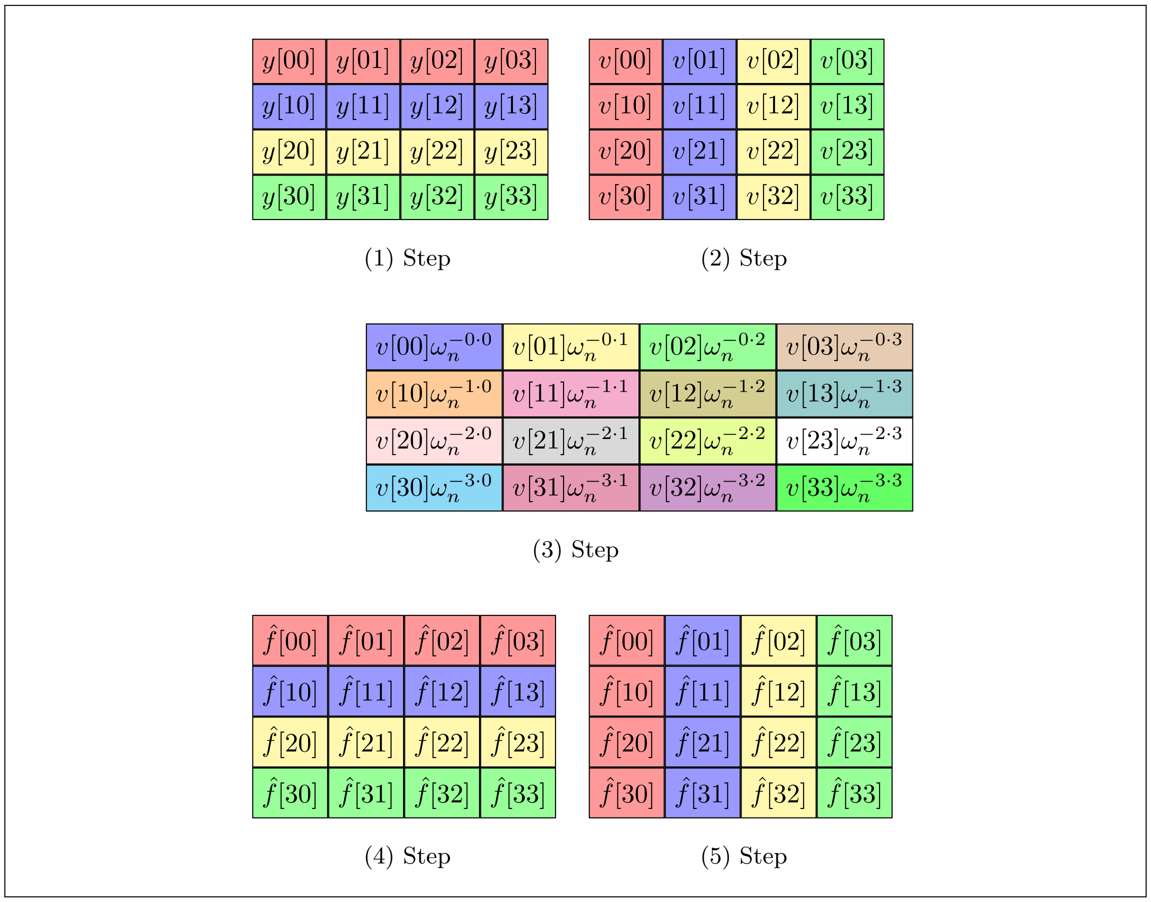
DFT Discrete Fourier Transform

DIT decimation in time

DIF decimation in frequency

FFTW Fastest Fourier Transform in the West

Tables



$y[00]$	$y[01]$	$y[02]$	$y[03]$
$y[10]$	$y[11]$	$y[12]$	$y[13]$
$y[20]$	$y[21]$	$y[22]$	$y[23]$
$y[30]$	$y[31]$	$y[32]$	$y[33]$

(1) Step

$v[00]$	$v[01]$	$v[02]$	$v[03]$
$v[10]$	$v[11]$	$v[12]$	$v[13]$
$v[20]$	$v[21]$	$v[22]$	$v[23]$
$v[30]$	$v[31]$	$v[32]$	$v[33]$

(2) Step

$v[00]\omega_n^{-0\cdot 0}$	$v[01]\omega_n^{-0\cdot 1}$	$v[02]\omega_n^{-0\cdot 2}$	$v[03]\omega_n^{-0\cdot 3}$
$v[10]\omega_n^{-1\cdot 0}$	$v[11]\omega_n^{-1\cdot 1}$	$v[12]\omega_n^{-1\cdot 2}$	$v[13]\omega_n^{-1\cdot 3}$
$v[20]\omega_n^{-2\cdot 0}$	$v[21]\omega_n^{-2\cdot 1}$	$v[22]\omega_n^{-2\cdot 2}$	$v[23]\omega_n^{-2\cdot 3}$
$v[30]\omega_n^{-3\cdot 0}$	$v[31]\omega_n^{-3\cdot 1}$	$v[32]\omega_n^{-3\cdot 2}$	$v[33]\omega_n^{-3\cdot 3}$

(3) Step

$\hat{f}[00]$	$\hat{f}[01]$	$\hat{f}[02]$	$\hat{f}[03]$
$\hat{f}[10]$	$\hat{f}[11]$	$\hat{f}[12]$	$\hat{f}[13]$
$\hat{f}[20]$	$\hat{f}[21]$	$\hat{f}[22]$	$\hat{f}[23]$
$\hat{f}[30]$	$\hat{f}[31]$	$\hat{f}[32]$	$\hat{f}[33]$

(4) Step

$\hat{f}[00]$	$\hat{f}[01]$	$\hat{f}[02]$	$\hat{f}[03]$
$\hat{f}[10]$	$\hat{f}[11]$	$\hat{f}[12]$	$\hat{f}[13]$
$\hat{f}[20]$	$\hat{f}[21]$	$\hat{f}[22]$	$\hat{f}[23]$
$\hat{f}[30]$	$\hat{f}[31]$	$\hat{f}[32]$	$\hat{f}[33]$

(5) Step

Figure 3: Schematic procedure of a Cooley-Tukey FFT