DOI: 10.35598/mcfpga.2025.021

Discrete Fourier Transform: Analysis, Algorithms, Software and Hardware Implementation

Oleksandr Vorgul
Scientific adviser
ORCID 0000-0002-7659-8796
dept.Microprocessor Technologies and System
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine
oleksandr.vorgul@nure.ua

Maksym Skorbatiuk
dept.Microprocessor Technologies and System
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine
maksym.skorbatiuk@nure.ua

***Abstract*— In this paper, a comprehensive analysis of the discrete Fourier transform (DFT) and fast Fourier transform (FFT) is performed with an emphasis on computational complexity, implementation features, and practical applications. Algorithms for directly computing the DFT and optimized FFT with time decimation are investigated. Comparative modeling is performed in the Octave and Python environments, including error analysis and optimization methods. A hardware implementation on FPGA using VHDL is presented, describing the structure of basic modules and operations with complex numbers. Particular attention is paid to the problem of processing signals with a length that is not a power of two and alternative approaches (Bluestein algorithm). The results demonstrate the effectiveness of the FFT for real-time spectral analysis tasks.**

Keywords*— Discrete Fourier Transform, FFT, Algorithm, Computational Complexity, FPGA, Octave, VHDL, Spectral Analysis.

I. Introduction

The Fourier transform algorithm is one of the most important tools for signal analysis in digital data processing [1] . This algorithm allows transforming time sequences into frequency spectra, which allows identifying and analyzing frequency components of signals. The Fourier transform is used in a wide range of fields of science and technology, including digital signal processing, image analysis, telecommunications, and many others [1,2] .

This paper discusses the implementation of the Fourier transform (FT) algorithm using the Octave and Python programming languages , as well as its implementation on programmable logic integrated circuits (FPGAs) using the VHDL language. The topic of computational complexity and the use of complex arithmetic for efficient execution of the algorithm is also touched upon [3] .

The discrete Fourier transform (DFT) algorithm plays a key role in digital signal processing and allows transforming a sequence of time samples into the frequency domain. The standard fast Fourier transform (FFT) algorithm is effective if the number of signal samples N is a power of two. However, in practice, it is often necessary to calculate the DFT for N points that are not a power of two. In such cases, more The discrete Fourier transform (DFT) transforms a sequence x [n] of N points into a spectral sequence X[k] with the universal algorithms are used, for example, the Bluestein algorithm.

II. Theoretical foundations of DFT and FFT

same number of elements, possibly complex ones. The DFT [1,2] can be calculated using the formula

$$X[k]=\sum_{n=0}^{N-1}x[n]\cdot W_N{}^{-n\cdot k} \tag{1}$$

where x [n] is the original sequence, X[k] is its frequency representation, i.e. spectral components, W[n,k] is the transform kernel. In theory, the kernel of the Fourier transform is often effectively represented in complex form:

$$W_N{}^{-n\cdot k}=\exp\left(-j\frac{2\pi}{N}\cdot n\cdot k\right) \tag{2}$$

If the computer allows, then (1) can be implemented in complex form. But if not, then, using trigonometric form, a real computer will allow (1) to be implemented as a set of real and imaginary parts from (2) [1,2]:

$$W_N{}^{-n\cdot k}=\cos\left(\frac{2\pi}{N}\cdot n\cdot k\right)-j\sin\left(\frac{2\pi}{N}\cdot n\cdot k\right) \tag{3}$$

Then, in the case of using (3), the set of complex spectral coefficients will be represented by a pair of real numbers. Their use will no longer require support for complex calculations, but will require modification of the algorithms formulated for complex quantities.

The direct approach to computing the DFT has a computational complexity of order O(N2), which can be quite expensive for large values of N.

The FFT algorithm proposed by Cooley and Tukey (1965) reduces the complexity to O(N log N) by recursively dividing the sequence into subsequences [3]. For bases 2 and 4 this requires a signal length of $N = 2^k$, otherwise zero padding or Bluestein's algorithm is used [1, 3].

III. DFT IMPLEMENTATION IN OCTAVE AND PYTHON

Octave and Python are convenient environments for modeling Fourier transform algorithms due to the presence of built-in libraries and support for operations with complex numbers.

1) Implementation in Octave

Matlab and Octave provide very similar built-in and standard library functions for performing DFT, such as fft (fast Fourier transform). However, for the purposes of modeling a direct implementation of DFT, one can write one's own function using complex arithmetic:

```
function X = dft (x)
N = length(x);
X = zeros( 1, N);
  for k = 1:N
    for n = 1:N
     X( k)=X(k)+x(n)*exp(-1i*2*pi*(n-1)*
(k-1)/N);
    end
  end
end
```

This code is quite functional and performs a direct discrete Fourier transform for a given sequence x . Here, each iteration of the loop computes a complex value for the corresponding frequency component of X[k]. The code does not use Octave's capabilities for efficient loop computation and is provided for computational complexity analysis.

For Octave , this code is far from optimal and looks strange, since it is not formulated in matrix notation. If the signal vector x from (1) is a row, then in matrix form (1) looks like this:

$$\mathbf{X} = \mathbf{x} \cdot \mathbf{W_N} \qquad (4)$$

Now (3) is obtained from (4) by multiplying the row x by the columns W_N one by one and then summing. Now we are ready to consider the algorithm and its optimization.

2) Preliminary calculation of tables of sines and cosines W_N

Usually quantity N counts is permanent meaning , therefore calculation tables W_N Maybe be done preliminary . In the cycle itself these tables should not be calculate , but only use . Table structure W_N enough is specific because column 0 and row 0 are represented units , 1st line or 1st column is main , rest columns or rows can be obtained from 1 by permutation without computing . Limitations productivity can arise because of necessary speed , time of changeover, cost and time memory access .

3) Errors in calculations

The problems with computational errors with the approach used in Octave or Python, that is, using all calculations at maximum bit depth, will be minor and for practical purposes are theoretical.

From a theoretical point of view, since there are multiplication and addition operations on floating or fixed point numbers, there will be errors, but they will be acceptable.

Calculating sines and cosines in a range of angles over one revolution of a circle gives a non-uniform error, and the errors of cosine and sine are different.

The peculiarity, as can be seen from (1) and clearly seen from the code for Octave, is the operation of the “multiply and accumulate” type, which will lead to the accumulation of calculation errors.

There is an interesting idea [3] how to normalize errors when calculating the table (2) or (3). It consists in the fact that only the sine of the minimum angle is calculated using the trigonometric formula. The cosine of the minimum angle is determined using the obtained result, the rest of the table is built using the reduction formulas. For the case of calculations with a fixed point of 16 bits, this leads to normalization and uniform distribution of the error over the entire field of change of the argument of trigonometric functions.

4) Python implementation

Python also has built - in tools for working with Fourier transforms, such as the NumPy library . However, like Octave , a direct implementation of the DFT can be done in a simple way:

```
import numpy as np

def dft (x):
N = len (x)
X = np.zeros ( N, dtype = complex)
  for k in range(N):
   for n in range(N):
X[k ]+ =x[n]* np.exp (-2*j* np.pi *k*n/N
)
  return X
```

`dft` function does the same thing as its Octave counterpart, but uses the NumPy library to handle arrays and complex numbers. Python and Octave allow for efficient computations with complex arithmetic, which is important for Fourier transform algorithms.

All modifications of the direct algorithm in Octave are also applicable to Python.

IV. IMPLEMENTATION OF DFT IN FPGA IN VHDL LANGUAGE

Implementation of DFT on programmable logic integrated circuits (FPGA) requires taking into account the architecture and features of the VHDL hardware description language. The main difficulty of implementation on FPGA is that all calculations, including operations with complex numbers, must be performed at the hardware level.

1) Implementation structure

To perform DFT on FPGA, it is necessary to organize the calculation of sums for each frequency coefficient X[k]. In VHDL, this can be described using modules that perform complex multiplications and additions.

2) VHDL code example

Below is an example of implementation of part of the DFT algorithm in VHDL:

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all ;

entity dft is
```

```
 port (
  clk : in std_logic ;
  reset : in std_logic ;
  start : in std_logic ;
  x_real , x_ imag : in signed(15 downto
 0);
-- Real and Imag In Data
   x_valid : in std_logic ;
   X_real , X_ imag : out signed(31 down
to 0);
-- Real an Imag Out Data
   done : out std_logic
);
end dft ;


architecture behavioral of dft is
-- internal signals, registers and pipel
ines
begin
    process( clk )
    begin
        if rising_edge ( clk ) then
            if reset = '1' then
-- RESET and
            elsif start = '1' then
-- DPF calculations
            end if;
        end if;
    end process;
end behavioral;
```

This implementation represents the structure of a process that will perform the DFT calculation in hardware. Implementing complex number operations requires creating multiplication and addition units to handle complex numbers.

A fairly effective development of the Fourier transform calculation can be achieved by implementing it at a high level (High Level Synthesis) in the C language with subsequent implementation in VHDL of the code of the implemented IP core.

And once again, in the standard library (not IP cores, there are some) of mathematics for HLS there is a function FFT. And also, as for Ocvave , it is possible to implement by the enumeration method, which will not be optimal.

V. Complexity of calculation

As mentioned earlier, a simple implementation of DFT has $O(N^2)$ computational complexity. This means that for each value of X[k], the sum of N complex products must be calculated. Thus, the algorithm requires N×N complex multiplications and N×N complex additions to complete, which becomes problematic for large values of N. For real-valued computations, this will result in more than a doubling of the complexity, since different forms of the two complex values will be required to perform the addition and multiplication

VI. Conclusions

The discrete Fourier transform remains a key tool for spectral analysis, despite the computational complexity of the direct algorithm being $O(N^2)$ [1]. The implementation of the FFT reduces the complexity to O(N log N), allowing efficient processing in Octave / Python and FPGA hardware systems [5, 6]. Critical aspects are:

Optimizing memory access and precomputing coefficients

Accounting for errors when working with a fixed point

Bluestein's algorithm to signals of arbitrary length [3].

Research prospects include hybrid CPU-FPGA architectures and adaptive algorithms for non-stationary signals [5].

References

[1] Oppenheim AV, Schafer RW, Buck JR Discrete-Time Signal Processing . – 3rd ed . – Pearson , 2009. – 1120 p .

[2] Rabiner LR, Gold B. Theory and Application of Digital Signal Processing . – Reprint . – Dover Publications , 2021. – 777 p .

[3] Blahut RE Fast Algorithms for Signal Processing . – Cambridge University Press , 2010. – 448 p .

[4] Lutz M. Learning Python : Powerful Object-Oriented Programming – 5th ed ., Vol . 1. – O'Reilly Media , 2013. – 1542 p .

[5] Kastner R., Matai J., Neuendorffer S. Parallel Programming for FPGAs [Electronic resource]. – La Jolla , 2018. – URL: https://kastner.ucsd.edu/wp-content/uploads/ 2018/03/admin/ pp4fpgas.pdf (accessed: 01.05.2025).

[6] Octave.org: About GNU Octave [Electronic resource]. – URL: https://www.octave.org (date of access: 01.05.2025).