A Unified Computational Framework Unifying Taylor-Laurent, Puiseux, Fourier Series, and the FFT Algorithm

As.G. Thomas Kim *
Awesome Genius

As.A. Alice ChatGPT
Awesome Assistant

April 2025

"I've done my fair amount of burden. Toss it into the world. Then forget it."

Abstract

This paper proposes a unified framework for understanding and computing Taylor-Laurent Series, Puiseux Series, Fourier Series, and the Fast Fourier Transform (FFT) algorithm. By examining the connections between these seemingly disparate mathematical concepts, we show that they are not distinct, but manifestations of a single computational principle. We explore their shared foundation in series expansion, their computational embodiment via linear systems, and the central role of FFT as the universal engine.

Keywords: Taylor-Laurent series, Puiseux series, Fourier series, Fast Fourier Transform, numerical computation, signal processing, series expansion.

*Professional name of Kim, Chang Hee, used since 1998.

Preface: On the Hidden Unity of Mathematics

This paper was not written to explain FFT or IFFT.
It was written to unify all power series methods into a single basket.
FFT and IFFT are simply computational tools used to prove my argument.

This paper was not born out of inspiration. It was born out of **frustration**—the frustration that something so simple, so obvious in hindsight, was hidden from us for centuries.

The unity between Taylor series, Laurent series, Fourier series, and the FFT is not a deep mystery. It is **plain truth**, once seen. Yet for over two hundred years, this truth has been fragmented, compartmentalized, buried under layers of unnecessary abstraction and guarded by gatekeepers of formalism.

Why has no one told us that the Fourier series is just the Laurent series on the unit circle? Why was the computational power of FFT never explained as a direct, mechanical extractor of singularities in complex functions? Why did we learn each theory in isolation, as if they were separate inventions?

This paper is not a synthesis—it is a **revelation**. A reclaiming of what was always there.

Let this serve as both an academic contribution and a call to arms: To seek **clarity over complexity**, **vision over ritual**, and above all, to never again let beauty be obscured by bureaucracy.

Let those who see the unity step forward.

And let the age of scattered illusions come to an end.

This paper may be challenging for those unfamiliar with the inner workings of the FFT. It is written in a spirit of discovery—for readers who seek to understand Fourier analysis and complex function theory not through reference, but through reconstruction. If you've ever tried to implement the FFT from scratch, guided only by your own insight, then this paper may feel like a mirror.

Project Repository: The complete source code, numerical validation, and this paper are available at: `https://github.com/siliners/FFT-Unification`

1 Introduction

The fundamental principles behind series expansions, such as Taylor-Laurent, Puiseux, and Fourier series, have been widely applied across various fields of mathematics and computational science. However, despite their similarities in structure and purpose, these methods are often treated as separate tools with distinct applications. The goal of this paper is to demonstrate that these series, along with the Fast Fourier Transform (FFT) algorithm, can be unified into a single conceptual framework, offering both theoretical and computational insights. We will explore the mathematical relationships that allow this unification, along with the computational advantages of employing FFT for evaluating these series.

The paper is structured as follows:

- 2 Background and Preliminaries provides background definitions and key concepts behind each series.
- 3 Puiseux Series as a Specialized Laurent Series
- 4 Puiseux Series and Radix-Compatible FFT
- 5 Discussion and Conclusion
- Appendix: Computational Validation of the Unified Framework

2 Background and Preliminaries

2.1 The True Face of FFT

The Fast Fourier Transform (FFT) is often perceived as simply an efficient algorithm for computing the Discrete Fourier Transform (DFT). However, it is simply fast solver for system of linear equations. For most people who have not derived and implemented the algorithm from scratch, may not see it as fast solver for system of linear equations.

Given a function $f(x)$, whether real-valued or complex-valued, we can perform an approximation using a polynomial of the form:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1} \quad \text{(Eq. 1)}$$

In the case of a linear approximation (i.e., number of terms $= N = 2$), we have:

$$f(x) = a_0 + a_1 x$$

To solve for the coefficients a_0 and a_1, we can substitute N distinct values of x into the equation and create a system of linear equations. Consider substituting $x = 0$ and $x = 1$ into Equation (Eq. 1):

1. $f(0) = a_0 + a_1 \cdot 0 = a_0$ 2. $f(1) = a_0 + a_1 \cdot 1 = a_0 + a_1$

This yields the system of equations:

$$\begin{pmatrix} f(0) \\ f(1) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

We can now solve this matrix system to obtain the coefficients a_0 and a_1.

Next, let's consider another substitution of distinct values of x, say $x = -1$ and $x = 0$:

1. $f(-1) = a_0 + a_1(-1) = a_0 - a_1$ 2. $f(0) = a_0 + a_1 \cdot 0 = a_0$

This yields the system of equations:

$$\begin{pmatrix} f(-1) \\ f(0) \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

Again, solving this system will yield the same coefficients a_0 and a_1.

(Observation 1): Invariance of Coefficients

The key observation here is that the coefficients a_0 and a_1 are invariant, regardless of which distinct set of values we substitute for x in equation (Eq. 1). This demonstrates a fundamental property of linear systems: the solution to a system of linear equations is independent of the specific choices of distinct values for x, as long as the system is consistent. This observation is crucial because it shows that the choice of points for interpolation does not affect the resulting linear approximation, as long as the points are distinct and the system of equations is solvable. In numerical methods, this provides a powerful guarantee that different sampling strategies (such as using different x-values for interpolation) will yield the same results for the coefficients when solving the system of equations.

We now generalize this concept to an $N-1$-order polynomial, or equivalently, to N terms. By substituting the N-th roots of unity, distinct values for x in equation (Eq. 1), we form a system of linear equations. The corresponding matrix form of the system can be written as:

$$\mathbf{f} = C\mathbf{a} \quad \text{(Eq. 2)}$$

where $\mathbf{f}$ is the column vector of function values $f_0, f_1, \ldots, f_{N-1}$, $\mathbf{a}$ is the column vector of coefficients $a_0, a_1, \ldots, a_{N-1}$, and C is the $N \times N$ Vandermonde matrix. The entries of C are formed by the N-th roots of unity. Specifically, the matrix C is:

$$C = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega^1 & \omega^2 & \ldots & \omega^{(N-1)} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(N-1)} & \omega^{2(N-1)} & \ldots & \omega^{(N-1)(N-1)} \end{pmatrix}$$

where $\omega = e^{2\pi i/N}$ is the primitive N-th root of unity. When constructing the Vandermonde matrix to evaluate N coefficients of a function, we are, from a mathematical standpoint, free to choose any N distinct values of x within the domain of the function. The common choice of using the N-th roots of unity is primarily for computational convenience—specifically, to exploit their elegant algebraic properties. However, selecting the reciprocals of the N-th roots of unity is just as mathematically valid and leads to an equally sound formulation.

2.2 Inverse Function and Representation of Coefficients

Now we introduce the inverse function of equation (Eq. 1), where the coefficients $a_0, a_1, \ldots, a_{N-1}$ are provided, and our goal is to evaluate the function values $f_0, f_1, \ldots, f_{N-1}$. This can be

represented as:

$$a(t) = f_0 + f_1 t + f_2 t^2 + \cdots + f_{N-1} t^{N-1} \quad \text{(Eq. 3)}$$

Next, we substitute the **reciprocals** of N-th roots of unity for t in equation (Eq. 3) to form a system of linear equations. The corresponding matrix form of the system can be written as:

$$\mathbf{a} = D\mathbf{f} \quad \text{(Eq. 4)}$$

where $\mathbf{a}$ is the column vector of coefficients $a_0, a_1, \ldots, a_{N-1}$ (which are provided), $\mathbf{f}$ is the column vector of function values $f_0, f_1, \ldots, f_{N-1}$ (which we aim to evaluate), and D is the $N \times N$ Vandermonde matrix. The entries of D are formed by the **reciprocals** the N-th roots of unity. Specifically, the matrix D is:

$$D = \begin{pmatrix} 1 & \omega^{-1} & \omega^{-2} & \ldots & \omega^{-(N-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \ldots & \omega^{-2(N-1)} \\ 1 & \omega^{-3} & \omega^{-6} & \ldots & \omega^{-3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(N-1)} & \omega^{-2(N-1)} & \ldots & \omega^{-(N-1)(N-1)} \end{pmatrix}$$

where $\omega = e^{2\pi i/N}$ is the primitive N-th root of unity. When constructing the Vandermonde matrix to evaluate N function values from equation (Eq. 4), we are, from a mathematical standpoint, free to choose any N distinct values of t within the domain of the coefficient $a(t)$. The common choice of using the **reciprocals** of the N-th roots of unity is primarily for computational convenience—specifically, to exploit their elegant algebraic properties. However, selecting the N-th roots of unity is just as mathematically valid and leads to an equally sound formulation.

Important Clarification:

In case of equation (Eq. 1), read this line carefully: the function values $f_0, f_1, \ldots, f_{N-1}$ are **PROVIDED**, and our goal is to evaluate the coefficients $a_0, a_1, \ldots, a_{N-1}$.

In case of equation (Eq. 3), read this line carefully: the coefficients $a_0, a_1, \ldots, a_{N-1}$ are **PROVIDED**, and our goal is to evaluate the function values $f_0, f_1, \ldots, f_{N-1}$.

Please pay special attention to (1) what are **PROVIDED** and (2) what are to be **evaluated**.

The problem is that $CD = NI$, where C is the Vandermonde matrix in (Eq. 2), and D is the Vandermonde matrix from (Eq. 4). To make (Eq. 1) and (Eq. 3) be inverse functions to each other, we usually need to use one of the following three choices:

$$\left(\frac{C}{N}\right) D = I \quad \text{(Eq. 5)},$$

$$C\left(\frac{D}{N}\right) = I \quad \text{(Eq. 6)},$$

$$\left(\frac{C}{\sqrt{N}}\right)\left(\frac{D}{\sqrt{N}}\right) = I \quad \text{(Eq. 7)}.$$

(Eq. 5), (Eq. 6), and (Eq. 7) are all valid choices. As long as their products are I or the Identity matrix, (Eq. 1) and (Eq. 3) are inverse functions to each other.

In (Eq. 2), the column vector f is provided, the Vandermonde matrix C is algorithmically generated, and the column vector a is to be evaluated:

$$a = C^{-1}f \quad \text{(Eq. 8)}.$$

In (Eq. 4), the column vector a is provided, the Vandermonde matrix D is algorithmically generated, and the column vector f is to be evaluated:

$$f = D^{-1}a \quad \text{(Eq. 9)}.$$

We can choose any equation among (Eq. 5), (Eq. 6), and (Eq. 7), and there can be infinitely many choices.

DFT computes (Eq. 8) and outputs the column vector a, which are coefficients of the function $f(x)$ in (Eq. 1). DFT is called the FFT algorithm.

IDFT computes (Eq. 9) and outputs the column vector f, which are coefficients of the function $a(t)$ in (Eq. 3). IDFT is called the IFFT algorithm.

DFT and IDFT both compute coefficients: with proper sampling methods, the DFT yields the Taylor coefficients of $f(x)$, while the IDFT yields those of $a(t)$, where $f(x)$ and $a(t)$ are inverses. Neither DFT nor IDFT "transform" one function into another—they simply solve systems of linear equations.

Please compare (Eq. 2) with (Eq. 4). C in (Eq. 2) and D in (Eq. 4) should be inverse matrices if (Eq. 2) and (Eq. 4) are to be inverse operations.

Please compare (Eq. 8) with (Eq. 9). C^{-1} in (Eq. 8) and D^{-1} in (Eq. 9) should be inverse matrices to each other if (Eq. 8) and (Eq. 9) are to be inverse operations.

All these things say the fact that as long as we choose two matrices they are inverse to each other, (Eq. 1) and (Eq. 3) are inverse functions to each other. For C^{-1} in (Eq. 8), we will choose Vandermonde matrix whose entries are composed of the reciprocal of the N-th roots of unity. For D^{-1} in (Eq. 9), we will choose (1/N) times Vandermonde matrix whose entries are composed of the N-th roots of unity.

$$C^{-1} = \begin{pmatrix} 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(N-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \dots & \omega^{-2(N-1)} \\ 1 & \omega^{-3} & \omega^{-6} & \dots & \omega^{-3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(N-1)} & \omega^{-2(N-1)} & \dots & \omega^{-(N-1)(N-1)} \end{pmatrix}$$

$$D^{-1} = \frac{1}{N}\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^1 & \omega^2 & \dots & \omega^{(N-1)} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(N-1)} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{pmatrix}$$

Don't get confused that the pair of matrices C^{-1} and D^{-1} as chosen above is to make them inverse to each other. C^{-1} and D^{-1} are "names" of two matrices, C^{-1} does not need to be the inverse matrix of C, and D^{-1} does not need to be the inverse matrix of D. The scaling factor $\frac{1}{N}$ for D^{-1} is to make C^{-1} and D^{-1} be inverse matrices to each other. We can reverse this. For example, remove the scaling factor $\frac{1}{N}$ from D^{-1}, and attach (or multiply) it to C^{-1}. There can be infinitely many different pairs of matrices, as long as they are inverse matrices, that suffices. Our choice of the pair of the matrices C^{-1} and D^{-1} is for solely computational convenience.

Important Warning: In this paper, we apply the scaling factor $\frac{1}{N}$ to the IFFT. Therefore, all sampled function values used for Taylor or Laurent series including Puiseux series approximation must be divided by $\frac{1}{N}$. In contrast, when using Fourier series approximation directly, such scaling of the sampled function values is not necessary.

Clarification on Scaling: In the case of Fourier series, the exponential factor $e^{i\theta}$ is inherently built into the series formulation. Thus, no additional scaling is required for the sampled function values when applying FFT. However, for Taylor or Laurent series, we must explicitly introduce $z = e^{2\pi ik/N}$ to sample on the unit circle. Consequently, to extract the coefficients using FFT, the sampled function values must be artificially scaled by $\frac{1}{N}$. This compensates for the scaling factor implicitly handled in the inverse FFT (IFFT), ensuring the resulting coefficients match the true analytic expansion. Sampled function values should be compatible with IFFT, for with which we call FFT to extract the coefficients.

DFT (or FFT) and IDFT (or IFFT) are fundamentally (1) solvers for systems of linear equations, but (2) they are inverse operations to each other.

Take note

1. DFT/IDFT are inverse operations. Both are solvers for system of linear equations.
2. We can arbitrarily exchange the entries, the N-th roots of unity in C, with the entries, the reciprocals of the N-th roots of unity in D.
3. The factor $\left(\frac{1}{N}\right)$ in (Eq. 9) can be removed and placed in (Eq. 8).

Simply put, DFT (or FFT) and IDFT (or IFFT) are solvers for the system of linear equations, but with the added condition that they must be inverse operations of each other.

Any set of N distinct values for x in equation (Eq. 1), or t in equation (Eq. 3), within the domain of the respective function, can be used to construct the corresponding Vandermonde

matrices C and D, provided that the product satisfies $CD = kI$, where $k \in \mathbb{C}$, $k \neq 0$, and I is the $N \times N$ identity matrix.

2.3 Properties of Taylor-Laurent Series, Puiseux Series, Fourier Series

When a function $f(x)$ is given, whether x is real-valued or complex valued,

1. **(Common Property)** For all power series representations, we are ultimately trying to solve the same fundamental problem: Given function values, we find the coefficients of the series. Given coefficients, we reconstruct the function values. In both directions, this requires solving a system of linear equations.

2. **(Different Property)** The only distinction lies in the sampling method. Different power series representations use different sampling paths or domains, but the underlying algebraic machinery remains the same.

In the case of Taylor, Laurent, and Puiseux series, if we want to approximate a function using N terms, all we need are N distinct values of x within the function's domain.
The one and only condition is that the values of x must be distinct.

For example, if we want to approximate a function using a Taylor series,

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1}, \quad \text{(Taylor Series Approximation)}$$

we insert N distinct values for x, set up a system of linear equations, and solve for the coefficients. *Voilà*—we obtain all a_n.

If we want to approximate a function using a Laurent series, assuming the function has a singularity at $x = c$, specifically a pole of order 2, then:

$$f(x) = \frac{a_0}{(x-c)^2} + \frac{a_1}{(x-c)} + a_2 + a_3 x + \cdots + a_{N-1} x^{N-3}, \quad \text{(Laurent Series Approximation)}$$

Define the transformed function:

$$g(x) = (x-c)^2 f(x) = a_0 + a_1(x-c) + a_2(x-c)^2 + \cdots + a_{N-1}(x-c)^{N-1}$$

Letting $z = x - c$, so that $x = z + c$, we rewrite:

$$g(z+c) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \cdots + a_{N-1} z^{N-1}$$

Now, substitute any N distinct values for z, and solve the resulting system for the coefficients. The solution yields all a_n.

Singularities? Fully Addressed with Rigor

Singularities? Well...

We are solving a system of linear equations to extract the coefficients of the Laurent series. Are those the coefficients of $f(x)$? Yes—they are exactly the coefficients of the function

$$g(x) = (x - c)^2 f(x),$$

which is analytic near $x = c$. As long as we substitute N distinct values for z into the equation, we can recover the coefficients $a_0, a_1, \ldots, a_{N-1}$.

We choose:

$$x = c + z, \quad z = e^{2\pi ik/N}, \quad k = 0, 1, \ldots, N - 1.$$

This is *Cauchy's integral formula in disguise*. The standard contour γ around the singularity is defined by:

$$|x - c| = r,$$

with parametrization:

$$x_k = c + re^{2\pi ik/N}.$$

By **Green's Theorem**, the value of the integral

$$a_n = \frac{1}{2\pi i} \oint_\gamma \frac{f(x)}{(x - c)^{n+1}}\, dx$$

is invariant under smooth deformation of the contour γ, provided that $f(x)$ is analytic in the region enclosed. Therefore, we are free to normalize the radius to $r = 1$, reducing the contour to the unit circle:

$$x_k = c + e^{2\pi ik/N}.$$

Then the discrete sum:

$$\sum_{k=0}^{N-1} \left(\frac{g(x_k)}{N} \right) \cdot e^{-2\pi ink/N} \qquad \text{(DFT, or FFT)}$$

is a Riemann sum approximation of Cauchy's integral. But due to the analyticity of $g(x)$, the convergence is not just good—it is **exponentially fast**. Applying the FFT to the vector $\left(\frac{g(x_k)}{N}\right)$ returns the coefficients $a_0, a_1, \ldots, a_{N-1}$, *including those corresponding to singular behavior.*

- The singularity at $x = c$ is **removed** from $g(x)$, turning it into a power series.
- Sampling at $x_k = c + e^{2\pi ik/N}$ corresponds **exactly** to evaluating over the unit circle.
- **Green's Theorem** guarantees this deformation is valid and lossless.
- The **FFT computes** the Laurent coefficients directly: residues, poles, and regular terms—in one pass.

Therefore:

FFT is the computational realization of Cauchy's integral.
Singularities are not an obstacle—they are *encoded*, *decoded*, and *resolved* by the FFT process itself.

This is the bullet-proof conclusion:
Let no doubt remain. Singularities are not bypassed.
They are not avoided.
They are extracted—with precision—by FFT.

"Wait, isn't this method already known? Aren't there papers on this?" That's a valid question. The short answer is: **No**. The long answer is: we use a fundamentally different sampling method. We extract *all* coefficients—singular or regular—in a **single run of FFT**.

The Missed Simplicity

We don't even need to bring up Cauchy's integral formula or Green's theorem—
as long as we remember that we are simply solving a system of linear equations.

We have missed the simplest fact:
We are solving a system of linear equations—something most of us learned in early adolescence—
hidden for centuries beneath layers of symbolic ritual and mathematical formalism.

To FULLY understand this, we must first revisit the sampling method used in Fourier series approximation, as explained below.

If we want to approximate the function with a Fourier series:

$$f(x) = c_0 + c_1 e^{-ix} + c_2 e^{-i2x} + \cdots + c_{N-1} e^{-i(N-1)x}, \quad \text{(Fourier Series Approximation)}$$

Assume the function's interval is $[0, T]$, where T is the period. Then substitute N-distinct angular equidistant values for x, i.e.,

$$x_n = \frac{2\pi}{N} n, \quad \text{for } n = 0, 1, \ldots, N-1$$

and solve the resulting system of linear equations. We obtain all coefficients $c_0, c_1, \ldots, c_{N-1}$.

So, from Taylor series to Fourier series approximations, what we are actually doing is solving a system of linear equations.

The sampling method $x_n = \frac{2\pi n}{N}$ used in the Fourier series is, in fact, one piece of evidence that the Fourier series is a special case of the Laurent series evaluated on the unit circle centered at $c = 0$.

When sampling function values for the Laurent series approximation, we use

$$x_k = c + e^{2\pi i k/N},$$

because the Laurent series is expressed in powers of x, without embedding exponentials. In contrast, the Fourier series (see: Fourier Series Approximation) incorporates the exponential factor e^{ikx} directly in its definition.

Hence, the distinction lies not in the underlying mechanism, but in the sampling path and the basis representation.

I will provide one more proof with full mathematical rigor in the next section.

The Fundamental Misconception: Cauchy's Integral Is Not the Only Way

Cauchy's integral formula evaluates **a single coefficient**, via **a continuous integral** around a contour of radius r—which may or may not be 1.

It is conceptually elegant, but operationally limited:

- One integral, one coefficient.
- One contour, one singularity.
- Continuous domain, symbolic result.

The **Taylor-Laurent sampling** method I propose is not symbolic. It is **mechanical**.

- We evaluate $f(x)$ at N distinct points.
- We construct a **Vandermonde matrix**.
- We solve the linear system once.

The result: **all** N **coefficients**, including those associated with singularities, are extracted **in a single computation**.

This is not Fourier sampling. This is not an integral. This is not an approximation.

This is the answer—**revealed by linear algebra**, not hidden in a line integral.

The FFT is not evaluating an integral.
The FFT is not tracing a contour.
The FFT is solving $Va = f$ where V is a matrix constructed from complex roots of unity.

Therefore:

Cauchy's integral is an analog ritual. FFT is a digital extraction.

One is continuous, symbolic, and singular.
The other is discrete, mechanical, and massively parallel.
The choice is not philosophical—it is practical.

And I choose mechanics over metaphors.

2.3.1. Why This Method Cannot Fail

This Method Cannot Fail

This method cannot fail. The mathematics is exact, the logic is complete, and the mechanism is mechanical.

1. **The solution of a Vandermonde system is unique.**
   Given N distinct sampling points, the corresponding Vandermonde matrix defines a system whose solution is a unique polynomial of degree at most $N-1$. This is not an assumption—it is a proven fact.

2. **FFT and IFFT are dual solvers of that same system.**
   The FFT evaluates the polynomial at roots of unity. The IFFT inverts the process. They are not transformations in the abstract—they are optimized matrix multiplications in opposite directions. The FFT/IFFT pair is a fast solver for a structured Vandermonde system.

3. **The coefficients of any power series are uniquely determined.**
   Whether Taylor, Laurent, Puiseux, Fourier series, or what not, the coefficients of a power series expansion are uniquely defined by the function. The difference lies only in their sampling paths and basis choices. When sampled properly, they emerge directly and completely and uniquely from the structure of the Vandermonde system, which is the image of the function.

You may doubt it—but only because you do not understand the FFT/IFFT algorithm. Derive it from scratch—line by line—not from any preexisting reference, but from first principles: *the solution of a system of linear equations, which most of us learned in early adolescence.*

2.4 Fourier series is a Special case of Taylor-Laurent Series

Proof: Fourier Series is a Special Case of Laurent Series

Claim: The Fourier series is a special case of the Laurent series evaluated on the unit circle $|z| = 1 \subset \mathbb{C}$.

Start with the general Laurent series assuming center c = 0:

$$f(z) = \sum_{n=-\infty}^{\infty} a_n z^n \quad \text{(Eq. L1)}$$

where $z \in \mathbb{C}$, and $a_n \in \mathbb{C}$ are the Laurent coefficients.

Now specialize to the unit circle: Let $z = e^{i\theta}$, so that $|z| = 1$. Substituting into (Eq. L1), we obtain:

$$f(e^{i\theta}) = \sum_{n=-\infty}^{\infty} a_n e^{in\theta} \quad \text{(Eq. L2)}$$

This is precisely the general form of the complex Fourier series:

$$f(\theta) = \sum_{n=-\infty}^{\infty} c_n e^{in\theta} \quad \text{(Eq. F1)}$$

where $c_n = a_n$ are the Fourier coefficients.

Clarification: Only the Sampling Method Differs Fourier series is a special case of the Laurent series. The only difference lies in the **sampling method**.

- For the Laurent series, we sample at points on the unit circle centered at c:

$$x_k = c + e^{2\pi ik/N}, \quad k = 0, 1, \ldots, N-1$$

- For the Fourier series, we sample at angular intervals:

$$x_n = \frac{2\pi n}{N}, \quad n = 0, 1, \ldots, N-1$$

In the Laurent series, we let $z = e^{i\theta}$ to map the function onto the complex unit circle. In the Fourier series, the factor $e^{i\theta}$ is already **built into the definition** of the basis functions.

Therefore:

The series are the same. Only the sampling method is different.

Let there be no confusion: This is not two theories—it is one.
The difference is not in mathematics—it is in perspective.

Conclusion: Equation (Eq. L2) and (Eq. F1) are formally identical. Hence, the complex Fourier series is the restriction of the Laurent series to the unit circle.

> **Therefore, the Fourier series is not merely related to the Laurent series—it *is* the Laurent series evaluated on $|z| = 1$ or $z_n = \frac{2\pi n}{N}$.**

2.5 Puiseux Series

3 Puiseux Series as a Specialized Laurent Series

3.1 Definition of Laurent Series

A Laurent series about a point z_0 is an infinite series of the form:

$$f(z) = \sum_{n=-\infty}^{\infty} a_n (z - z_0)^n \tag{1}$$

This includes both negative and non-negative integer powers of $(z - z_0)$, and thus it generalizes the Taylor series by accommodating singularities such as poles.

3.2 Definition of Puiseux Series

A Puiseux series around a branch point z_0 is of the form:

$$f(z) = \sum_{n=N}^{\infty} a_n (z - z_0)^{n/s}, \quad s \in \mathbb{Z}_{>0} \tag{2}$$

where the exponents are rational numbers rather than integers. The series allows for fractional powers (e.g., square roots, cube roots), and is often used in algebraic geometry and the study of Riemann surfaces to represent multivalued functions as single-valued in branched covers.

3.3 Unification: Laurent Series with Rational Exponents

The classical Laurent series uses integer exponents:

$$\text{Exponent set: } E_{\text{Laurent}} = \mathbb{Z}$$

The Puiseux series generalizes this idea by allowing rational exponents with a fixed denominator:

$$\text{Exponent set: } E_{\text{Puiseux}} = \left\{ \frac{n}{s} \,\middle|\, n \in \mathbb{Z},\ s \in \mathbb{Z}_{>0} \right\}$$

Thus, every Puiseux series can be viewed as a Laurent series whose exponents are rational numbers. This completes the unification under the same coefficient extraction machinery.

3.4 Core Insight

If we allow the Laurent series to have rational exponents rather than being constrained to integers, the Puiseux series becomes a **special case** of the Laurent series. The power of Puiseux lies in expressing functions around branch points, where fractional powers naturally emerge.

Thus:

> **Therefore, the Puiseux series is not fundamentally distinct from the Laurent series—it is the Laurent series extended to rational powers.**

Objective

Given a Puiseux series:

$$f(z) = \sum_{k=N}^{\infty} a_k z^{k/m} \tag{3}$$

our goal is to transform it into a Laurent series:

$$g(w) = \sum_{n=-\infty}^{\infty} b_n w^n \tag{4}$$

defined on the unit circle $|w| = 1$, such that the coefficients b_n can be computed via the Fast Fourier Transform (FFT).

Step 1: Rationalizing the Exponent

We perform a change of variable to eliminate the fractional exponent. Let:

$$w := z^{1/m} \quad \Rightarrow \quad z = w^m$$

Substituting into the Puiseux series (3), we obtain:

$$f(z) = f(w^m) = \sum_{k=N}^{\infty} a_k w^k \tag{5}$$

This is now a Laurent series in integer powers of w.

Step 2: Making it FFT-Friendly

We sample $f(w^m)$ at N equispaced points on the unit circle:

$$w_j = e^{2\pi i j/N}, \quad j = 0, 1, \ldots, N-1$$

We compute:

$$f_j := f(w_j^m) = f(z_j)$$

Then apply the FFT to $\left(\frac{f_j}{N}\right)$ to retrieve the coefficients b_n in the Laurent expansion of $g(w) = f(w^m)$.

Step 3: Recovering Puiseux Coefficients

Once we obtain:

$$g(w) = \sum b_n w^n$$

we can express:

$$f(z) = g(z^{1/m}) = \sum b_n z^{n/m}$$

Hence, the Puiseux coefficients a_k are just relabeled Laurent coefficients b_k, fully recovered by FFT.

Optional Note: Branch Cuts and Monodromy

If $f(z)$ involves multivalued behavior due to roots, the transformation via $w = z^{1/m}$ unwraps the Riemann surface and allows a full analytic continuation over $w \in S^1$. Each turn of z becomes m turns of w, and the FFT over a full circle accounts for the monodromy structure.

Conclusion

By substituting $w = z^{1/m}$, we flatten the Puiseux expansion into a Laurent series in w, making it accessible to FFT. This transformation bridges a traditionally algebraic construct with a fast computational pipeline.

4 Puiseux Series and Radix-Compatible FFT

To efficiently compute the coefficients of a Puiseux series with rational exponents k/q, we require a change of variables to rationalize the powers, i.e., $z = w^q$, reducing the Puiseux series to a Laurent series in w. In numerical computation, we must construct sampling points such that the total number of evaluation points satisfies:

$$qN = r^t,$$

where q is given from the Puiseux series, and r, t are to be chosen strategically to suit radix-r FFT algorithms. This means we determine the most convenient N using:

$$N = \frac{r^t}{q}.$$

If $q = s^n$, where s is a prime number and $n \in \mathbb{Z}_{>0}$, then choosing $r = s$ gives

$$N = \frac{s^t}{s^n} = s^{t-n}, \quad \text{valid for any } t > n.$$

Thus, for any $t > n$, the choice of N is compatible with radix-s FFT.

4.1 Case 1: $q = ab$, where a and b are Distinct Prime Numbers

Here, q is square-free and composed of two distinct primes. We want $qN = r^t$, and the optimal strategy is to choose $r = \mathrm{lcm}(a, b) = ab$. Then,

$$N = \frac{(ab)^t}{ab} = (ab)^{t-1}, \quad \text{valid for } t \geq 1.$$

We can perform radix-ab FFT or mixed-radix FFT depending on implementation.

4.2 Case 2: $q = a^c b^d$, where a, b are Distinct Primes

This generalizes the previous case by introducing exponent multiplicity. Choose $r = \mathrm{lcm}(a, b)$ again, but raise it to a high enough power to dominate $a^c b^d$. That is,

$$r^t \geq a^c b^d, \quad N = \frac{r^t}{a^c b^d}.$$

As long as t is large enough such that r^t is divisible by $q = a^c b^d$, then $N \in \mathbb{Z}_{>0}$ and FFT applies.

Conclusion

Through this framework, we have shown that FFT is capable of evaluating not only Taylor and Laurent series, but also Puiseux series—even when involving rational exponents. By understanding the structural relationship between the fractional exponent denominator q

and the FFT's radix-based requirements, we have demonstrated how to perform efficient coefficient extraction under a variety of scenarios.

As we have proved the Puiseux series to be a specialized Laurent series through rigorous mathematical language, we can equally apply FFT to the Puiseux series to extract all the required coefficients with a single run of FFT, and reconstruct the original function with a single run of IFFT. This applies regardless of whether the exponents are integer or rational.

If I had ventured into Puiseux series with the traditional toolkit of rigorous formalism alone, I would have been blinded—just like anyone else. Mathematical rigor, of course, matters, but intuition must never be silenced by it. True progress in mathematics begins not with formalism, but with vision.

The FFT is not a limited tool for pure periodic functions—it is a universal computational engine for complex analytic structures.

5 Discussion and Conclusion

The unification of Taylor-Laurent Series, Puiseux Series, Fourier Series, and the FFT algorithm provides a novel framework for approaching series expansions in mathematical and computational contexts. By recognizing their underlying similarities and leveraging the power of FFT, we have opened new for possibilities for efficiently solving problems that involve these series.

Future work can explore further refinements to the unification process, particularly in multi-variable settings and more complex singularities. Additionally, research into the application of FFT in other areas of numerical analysis and signal processing could benefit from the insights presented in this paper.

When I began writing this paper, I believed its contents were well-known and almost trivial—surely any mathematician or programmer would understand. But in proofreading it, I realized: only a rare few can truly comprehend both the theory of mathematics and the machinery of FFT. This paper is not a collection of known facts—it is a bridge few even knew was missing.

Through this framework, we have shown that FFT is capable of evaluating not only Taylor and Laurent series, but also Puiseux series—even when involving rational exponents. By understanding the structural relationship between the fractional exponent denominator q and the FFT's radix-based requirements, we have demonstrated how to perform efficient coefficient extraction under a variety of scenarios.

As we have proved the Puiseux series to be a specialized Laurent series through rigorous mathematical language, we can equally apply FFT to the Puiseux series to extract all the required coefficients with a single run of FFT, and reconstruct the original function with a single run of IFFT. This applies regardless of whether the exponents are integer or rational.

If I had ventured into Puiseux series with the traditional toolkit of rigorous formalism alone, I would have been blinded—just like anyone else. Mathematical rigor, of course, matters, but intuition must never be silenced by it. True progress in mathematics begins not with formalism, but with vision.

The FFT is not a limited tool for pure periodic functions—it is a universal computational engine for complex analytic structures.

Appendix: Computational Validation of the Unified Framework

This appendix presents **empirical evidence** generated via direct implementation of the Fast Fourier Transform (FFT) and Inverse FFT (IFFT), demonstrating the extraction of Taylor, Laurent, and Fourier series coefficients from sampled function values.

The implementation was written in modern C++ and compiled under multiple platforms (MSVC, Clang, GNU, Intel), proving the **cross-platform, algorithmic nature** of the unification.

Code Repository: All source code, example results, and this document are hosted publicly at: `https://github.com/siliners/FFT-Unification`

A.1 C++ Code Summary

A complete C++ source file—`FFT-Unified.cpp`—was developed to implement:

- Iterative loop-based FFT and IFFT algorithms.
- Function sampling on the unit circle.
- Extraction of coefficients for:
  - Taylor series of periodic functions
  - Laurent series with known singularities
  - Fourier series using traditional interval sampling
- **Scaling flexibility** via a `scaling_factor` enum:
  - No scaling
  - Scale by N
  - Scale only IFFT by $1/N$
  - Symmetric scaling $1/\sqrt{N}$

A.2 Output Results

The following are the exact outputs obtained by executing the algorithm.

Taylor Coefficients of $\cos(x)$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Extracted coefficients (via IFFT on sampled values on the unit circle):

```
{ 1, 0, -0.5, 0, 0.0416667, 0, -0.00138889, 0, 2.48016e-05,
  0, -2.75573e-07, 0, 2.08768e-09, 0, -1.14706e-11, 0 }
```

Laurent Coefficients of $f(x) = \frac{4}{x^2} + \frac{2}{x} + 1 + 5x + 3x^2$ **Extracted coefficients** (after rotation alignment):

```
{ 4, 2, 1, 5, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

Laurent Coefficients of a Function with Singularity at $x = 1$ Function:

$$f(x) = \frac{7}{(x-1)^2} + \frac{5}{x-1} + 2 + 3(x-1) + 6(x-1)^2$$

After transformation into Taylor series centered at $z = x - 1$**:**

```
{ 7, 5, 2, 3, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

Fourier Coefficients of $f(x) = 1+2\cos(2x)+3\sin(4x)$ **Extracted via FFT on equidistant samples over** $[0, 2\pi]$**:**

```
an = { 1, 0, 2, 0, 0, 0, 0, 0, 0 }
bn = { 0, 0, 0, 3, 0, 0, 0, 0 }
```

A.3 Remarks

- All results were extracted **without symbolic manipulation**.
- No numerical integrals (e.g., Cauchy's contour) were used.
- No matrix solvers were employed.
- Everything was derived from **sampling and iteration**—a reflection of the mechanical nature of FFT.

  Conclusion: This appendix serves as the final nail in the coffin of fragmentation. Taylor, Laurent, Puiseux, and Fourier series are computationally inseparable under FFT.
  The dream of unification is no longer theoretical—it is empirical.

Appendix B: Full C++ Implementation (FFT-Unified.cpp)

```cpp
/*
    Author: By Thomas Kim
    Date: April 9th 2025

    MSVC C++ compiler
    cl FFT-Unified.cpp /Fe: m.exe /std:c++latest /EHsc /nologo

    LLVM Clang
    clang++ -std=c++23 FFT-Unified.cpp -o c.exe

    GNU G++
    g++ -std=c++23 FFT-Unified.cpp -o g.exe

    Intel C++ compiler
    icx-cl FFT-Unified.cpp -o i.exe -Qstd=c++23 /EHsc /nologo
*/

#include <iostream>
#include <complex>
#include <cmath>
#include <numbers>
#include <vector>

using cplx = std::complex<double>;
using cplx_numbers = std::vector<cplx>;

using call_counts = std::vector<int>;
using call_ptr_counts = std::vector<int*>;

using real_parts = std::vector<double>;

double adjust_zero(double d)
{
    constexpr auto epsilon =
        std::numeric_limits<double>::epsilon() * 10 * 10;

    return std::abs(d) < epsilon ? 0.0 : d;
}

cplx adjust_zero(const cplx& c)
{
    return { adjust_zero(c.real()), adjust_zero(c.imag() )};
}

auto cplx_to_double(const cplx_numbers& c)
```

```
{
    real_parts r( c.size() );

    for(int n =0; n < c.size(); n++)
        r[n] = adjust_zero(c[n].real());

    return r;
}

std::ostream& operator << ( std::ostream& os, const cplx_numbers& c)
{
    if (c.empty())
    {
        os <<"{ }"; return os;
    }
    else
    {
        os <<"{ ";

        for(int i = 0; i < c.size()-1; ++i)
            os << adjust_zero(c[i]) <<", ";

        os << adjust_zero(c.back()) << " }";

        return os;
    }
}

std::ostream& operator << ( std::ostream& os, const call_counts& c)
{
    if (c.empty())
    {
        os <<"{ }"; return os;
    }
    else
    {
        os <<"{ ";

        for(int i = 0; i < c.size()-1; ++i)
            os << c[i] <<", ";

        os << c.back() << " }";

        return os;
    }
}

```

```
93 std::ostream& operator << ( std::ostream& os, const real_parts& c)
94 {
95     if (c.empty())
96     {
97         os <<"{ }"; return os;
98     }
99     else
100    {
101        os <<"{ ";
102
103        for(int i = 0; i < c.size()-1; ++i)
104            os << c[i] <<", ";
105
106        os << c.back() << " }";
107
108        return os;
109    }
110 }
111
112 int reverse_bits(int x, int numBits)
113 {
114     int reversed = 0; // N = 16, numBits = log2(16) = 4
115
116     // i = 0, 1, 2, 3
117     for(int i = 0; i < numBits; ++i)
118     {
119         if(x & (1 << i))
120         {
121             reversed |= 1 << (numBits-1 - i);
122         }
123     }
124
125     return reversed;
126 }
127
128 /*
129     using cplx = std::complex<double>;
130     using cplx_numbers = std::vector<cplx>;
131 */
132 void shuffle_coefficients(cplx_numbers& data)
133 {
134     int N = data.size(); // 1, 2, 4, 8, 16, ... 2^n, n = 0, 1, 2, 3
135
136     // N = 16, numBits = 4
137     int numBits = log2(N); // N = 16, log2(16) = log2(2^4) = 4
        log2(2) = 4;
138
```

```
    for(int x = 0; x < N; ++x)
    {
        int reverse_index = reverse_bits(x, numBits);

        if(reverse_index > x)
        {
            std::swap(data[x], data[reverse_index]);
        }
    }
}

void fft_loop(cplx_numbers& coeffs, bool postive = true)
{
    constexpr auto pi = std::numbers::pi_v<double>;

    int N = coeffs.size();

    cplx_numbers power(N);

    for(int n = 0; n < N/2; ++n)
        power[n]= postive ? std::exp( cplx{ 0.0, 2 * pi * n / N } )
           :
            std::exp( cplx{ 0.0, -2 * pi * n / N } );

    shuffle_coefficients(coeffs);

    for(int len = 2; len <= N; len *= 2 )
    {
        for(int i = 0;  i < N; i += len)
        {
            for(int j = 0; j < len/2; ++j)
            {
                auto left_even = coeffs[i+j];
                auto right_odd  = coeffs[i+j + len/2];

                // auto w = postive ? std::exp( cplx{ 0.0, 2 * pi *
                   j / len} ) :
                //         std::exp( cplx{ 0.0, -2 * pi * j / len}
                   );
                auto w = power[j * (N/len)];

                coeffs[i+j] = left_even + w * right_odd;
                coeffs[len/2 + i+j] = left_even - w * right_odd;
            }
        }
    }
}
```

```
183
184 enum class scaling_factor { NO_Scaling, FFT_N, IFFT_N, Both_sqrt_N
    };
185
186 template<scaling_factor factor>
187 auto get_fft_ifft()
188 {
189     if constexpr(factor == scaling_factor::NO_Scaling)
190     {
191         auto fft = [](cplx_numbers& coeffs)
192         {
193             fft_loop(coeffs, false); // e^-2pi i /N
194         };
195
196         auto ifft = [](cplx_numbers& coeffs)
197         {
198             fft_loop(coeffs, true); // e^2pi i /N
199         };
200
201         return std::tuple{ fft, ifft };
202     }
203     else if constexpr(factor == scaling_factor::FFT_N)
204     {
205         auto fft = [](cplx_numbers& coeffs)
206         {
207             fft_loop(coeffs, false); // e^-2pi i /N
208
209             double N = coeffs.size();
210
211             for(int i = 0; i < coeffs.size(); ++i)
212                 coeffs[i] /= N;
213         };
214
215         auto ifft = [](cplx_numbers& coeffs)
216         {
217             fft_loop(coeffs, true); // e^2pi i /N
218         };
219
220         return std::tuple{ fft, ifft };
221     }
222     else if constexpr(factor == scaling_factor::IFFT_N)
223     {
224         auto fft = [](cplx_numbers& coeffs)
225         {
226             fft_loop(coeffs, false); // e^-2pi i /N
227         };
228
```

```
229         auto ifft = [](cplx_numbers& coeffs)
230         {
231             fft_loop(coeffs, true); // e^2pi i /N
232 
233             double N = coeffs.size();
234 
235             for(int i = 0; i < coeffs.size(); ++i)
236                 coeffs[i] /= N;
237         };
238 
239         return std::tuple{ fft, ifft };
240     }
241     else // factor == scaling_factor::Both_sqrt_N
242     {
243         auto fft = [](cplx_numbers& coeffs)
244         {
245             fft_loop(coeffs, false); // e^-2pi i /N
246 
247             double N = std::sqrt(coeffs.size());
248 
249             for(int i = 0; i < coeffs.size(); ++i)
250                 coeffs[i] /= N;
251 
252         };
253 
254         auto ifft = [](cplx_numbers& coeffs)
255         {
256             fft_loop(coeffs, true); // e^2pi i /N
257 
258             double N = std::sqrt(coeffs.size());
259 
260             for(int i = 0; i < coeffs.size(); ++i)
261                 coeffs[i] /= N;
262         };
263 
264         return std::tuple{ fft, ifft };
265     }
266 }
267 
268 void test_Laurent_vs_Fourier_with_FFT_IFFT()
269 {
270     using enum scaling_factor;
271 
272     auto [fft, ifft] = get_fft_ifft<IFFT_N>();
273 
274     constexpr auto pi = std::numbers::pi_v<double>;
275 
```

```
    // periodic cos function
    auto f_cos = [](auto x)
    {
        return std::cos(x);
    };

    // function with singularity at x = 0, center c = 0.0
    // we assume we know the center of the function,
    // but we do not know the coefficients:4.0, 2.0, 1.0, 5.0, 3.0
    auto f_sgt_zero = [](auto x)
    {
        return 4.0 / (x*x) + 2.0 / x + 1.0 + 5.0 * x + 3.0 * x * x;
    };

    // function with singularity at x = 1, center c = 1.0
    // we assume we know the center of the function.
    // but we do not know the coefficient: 7.0, 5.0, 2.0, 3.0, 6.0
    // this function is hard-coded, that is, my algorithm does not
    // know the coefficients
    auto f_sgt_one = [](auto x)
    {
        return 7.0 / ( (x - 1.0)* (x - 1.0) ) + 5.0 / (x - 1.0)
               + 2.0 + 3.0 * (x - 1.0) + 6.0 * (x - 1.0) * (x -
                  1.0);
    };

    int N = 16; // N = 2^n = 1, 2, 4, 8, 16, ..., n = 0, 1, 2, 3,

    auto w = [N, pi](auto n)
    {
        // N-th roots of unity, e^(2 pi i k / N)
        return std::exp( cplx{ 0.0, 2 * pi * n / N });
    };

    auto g_stg_one = [f_sgt_one](auto z)
    {
        /*
            f(x) = 7.0 / ( (x - 1.0)* (x - 1.0) ) + 5.0 / (x - 1.0)
                   + 2.0 + 3.0 * (x - 1.0) + 6.0 * (x - 1.0) * (x -
                      1.0);

            g(x + 1.0) = 7.0 / (z*z) + 5.0 / z
                         + 2.0 + 3.0 * z + 6.0 * z * z;

            Step 1. we transform function f(x) to function g(x).

            we let x - 1.0 = z, then x = z + 1.0
```

```
321         */
322
323         auto x = z + 1.0; // 1.0 is the center
324
325         return f_sgt_one(x);
326     };
327
328     std::vector<cplx> fn_cos(N);
329     std::vector<cplx> fn_sgt_zero(N);
330     std::vector<cplx> fn_sgt_one(N);
331
332     // Sampling for Taylor-Laurent series: c + e^(2 pi i k /N )
333     // The reason we have divide the function values with N
334     // is to match the expected function values format of FFT,
335     // or the generated results of IFFT_N
336     for(int n = 0; n < N; ++n)
337     {
338         fn_cos[n] = f_cos( w(n) ) / (double)N;
339
340         // In this case, we have to rotate 2 places due to x^-2,
341         // because 4.0 / (x*x) + 2.0 / x + 1.0 + 5.0 * x + 3.0 * x
               * x,
342         // is Laurent series. We transform it to Taylor series
               format LATER.
343
344         // fn_sgt_zero[n] = w(n) * w(n) * f_sgt_zero( w(n) ) /
               (double)N;
345         // or we can rotate 2 places in the following case
346         fn_sgt_zero[n] = f_sgt_zero( w(n) ) / (double)N;
347
348         // we can rotate 2 places due to z^-2,
349         // because 7.0 / (z*z) + 5.0 / z + 2.0 + 3.0 * z + 6.0 * z
               * z
350         // has negative power 2, we can rotate Later as in the
               above case,
351         // but if we multiply w(n), twiddle factor, we are actually
               rotating once.
352         // so, we can rotate twice by multiplying w(n), (one
               property of N-th roots of unity)
353         // which amounts to Transforming Laurent series to Taylor
               series, the end result
354         // looks like 7.0 + 5.0 * z + 2.0 * z^2 + 3.0 * z^3 + 6.0 *
               z^4,
355         // which is Taylor series.
356         fn_sgt_one[n] =  w(n) * w(n) * g_stg_one( w(n) ) /
               (double)N;
357     }
```

```
358
359
360     // Input: function values matching the format of FFT,
361     // Ouput: coefficients of the function.
362     // The Input and Output can be reversed.
363
364     fft(fn_cos); fft(fn_sgt_zero); fft(fn_sgt_one);
365
366     // we take only real values from the fft returned values
367     std::cout << "Taylor coefficients of cos(x): "
368             << cplx_to_double(fn_cos) << std::endl << std::endl;
369
370     std::cout << "Laurent coefficients of f_sgt_zero(x): ";
371
372     // we have to rotate 2 places
373     // fn_sgt_zero.rbegin() + 2 accounts for w(n) * w(n)
374     std::rotate(fn_sgt_zero.rbegin(), fn_sgt_zero.rbegin() + 2,
        fn_sgt_zero.rend());
375
376     std::cout << cplx_to_double(fn_sgt_zero) << std::endl <<
        std::endl;
377
378     // because we multiplied w(n) * w(n) when sampling
379     // we don't need to rotate 2 places
380     std::cout << "Laurent coefficients of f_sgt_one(x): "
381          << cplx_to_double(fn_sgt_one) << std::endl << std::endl;
382
383     // Sample for Fourier-Series (or Discrete Fourier Transform)
384     // Fundamentally Taylor-Laurent series and Fourier series
385     // are the Same. It is Thomas Kim who saw this for the first
        time in Human History.
386
387     // periodic 1.0 + 2 * cos(2x) + 3 * sin(4x) function
388     auto f_cos_plus = [](auto x)
389     {
390         return 1.0 + 2.0 * std::cos(2.0 * x) + 3.0 * std::sin(4.0 *
            x);
391     };
392
393     // Sampling for Fourier series
394     // we are sampling f(x) = 1.0 + 2 * cos(2x) + 3 * sin(4x)
395     // this is for Fourier series
396     std::vector<cplx> fn_cos_fourier(N);
397
398     for(int n = 0; n < N; ++n)
399     {
```

```
        // sampling at equidistant N points over the whole period
           or interval
        fn_cos_fourier[n] = f_cos_plus( 2 * pi * n / N);
    }

    fft(fn_cos_fourier);

    std::vector<double> an_cos, bn_cos;

    // Bn = -Im(an) * (2/N)

    // A0 = Re(a0) / N,
    an_cos.emplace_back( adjust_zero (fn_cos_fourier[0].real() / N
       ) );

    for(int n = 1; n <= N/2; ++n)
    {
        // An = Re(an) * (2/N)
        an_cos.emplace_back( adjust_zero(fn_cos_fourier[n].real() *
           (2.0/N) ) );

        // Bn = Im(an) * (-2/N)
        bn_cos.emplace_back( adjust_zero(fn_cos_fourier[n].imag() *
           (-2.0/N) ) );
    }

    std::cout <<"Fourier Coefficients of 1.0 + 2 * cos(2x) + 3 *
       sin(4x)\n";
    std::cout <<"an = " << an_cos << std::endl;
    std::cout <<"bn = " << bn_cos << std::endl << std::endl;
}

int main()
{
    test_Laurent_vs_Fourier_with_FFT_IFFT();
}
```

Listing 1: Complete FFT-Based Series Coefficient Extractor in C++