Design and Evaluation of a Single-Line Randomized Error Diffusion Algorithm for Hardware-Constrained Display Processing Units

1. Introduction

1.1. Context: The Challenge of High-Quality Dithering in Resource-Constrained DPUs

Display Processing Units (DPUs) are integral components in modern display pipelines, tasked with enhancing visual data before it reaches the screen. There is an ever-increasing demand for superior visual fidelity, even as display resolutions, refresh rates, and color depth capabilities expand. However, practical systems often face limitations in available color palettes due to constraints in bandwidth, memory capacity, or the cost of display panels themselves. This discrepancy between the desire for continuous-tone image representation and the reality of limited color output necessitates techniques to bridge the gap. Dithering is a fundamental and widely adopted technique that addresses this challenge by creating the illusion of a greater number of colors and smoother tonal transitions than are physically available in the output color palette.[1] At its core, dithering involves the intentional application of a controlled form of noise to the image signal prior to quantization. This process serves to randomize the quantization error, thereby preventing the formation of large-scale, visually objectionable patterns such as color banding.

The challenge is particularly acute in the context of modern DPUs, which are frequently embedded within systems where computational resources, memory bandwidth, and power consumption are severely restricted. Traditional dithering algorithms, while potentially offering high quality, may be too computationally expensive or require memory buffers (e.g., line buffers for 2D error diffusion) that are infeasible in such environments.[3] As display technologies push towards higher resolutions and refresh rates, the strain on DPU resources intensifies, making the development of highly efficient yet effective dithering algorithms a critical area of innovation. The constraints imposed by DPU designs are not arbitrary; they reflect tangible engineering trade-offs between cost, power, and performance.

1.2. Problem Statement: Limitations of Existing User-Implemented Algorithms

The existing dithering solutions implemented by the user—specifically Ordered Dithering, simple Truncation, and a form of Random Dithering for 2-bit and 4-bit color depth reduction—suffer from several visual deficiencies. These include the appearance of visible, structured patterns, a common artifact associated with ordered

dithering techniques like those using Bayer matrices.[5] Color banding, characterized by abrupt and noticeable transitions between discrete color shades in areas that should exhibit smooth gradients, is another significant issue, often resulting from simple truncation or insufficiently effective dithering.[1] Furthermore, temporal flicker has been observed, which can arise from naive implementations of random dithering where the noise pattern changes erratically between frames, or from effects analogous to Frame Rate Control (FRC).[7] Such artifacts significantly degrade the perceived quality of the displayed image or video, which is unacceptable for high-quality display applications.

While metrics like Peak Signal-to-Noise Ratio (PSNR) provide a quantitative measure of image fidelity, the user's complaints regarding "visible patterns" and "banding" underscore a deeper requirement for perceptually pleasing outcomes. Effective dithering should aim to transform quantization error into high-frequency noise that is less conspicuous to the human visual system (HVS).[1] This principle of shaping noise according to HVS characteristics is fundamental to achieving high perceptual quality.

1.3. Objective: Design and Evaluation of a Novel Hardware-Friendly Dithering Algorithm

The primary objective of this report is to design, implement, and rigorously evaluate a novel dithering algorithm, hereafter referred to as Single-Line Randomized Error Diffusion (SLRED). This algorithm is specifically engineered to operate effectively within the stringent hardware constraints imposed by the target DPU. These constraints include:

- Processing based on a single memory line at a time, precluding the use of multi-line buffers.
- Pixel-at-a-time operation with access limited to the current pixel and the next 3 to 4 pixels on the same line.
- Absence of floating-point arithmetic; all calculations must be achievable using integer operations.
- Utilization of small integer multiplications, ideally replaceable by bit-shift operations for maximum hardware efficiency.
- Avoidance of serpentine scanning patterns, mandating a unidirectional scan (e.g., left-to-right) for each line.

Furthermore, the SLRED algorithm must incorporate pseudo-randomness that varies for each memory line to disrupt pattern formation. It must also offer a controllable noise strength, allowing for bipolar noise injection (e.g., +/-2 quantization levels or potentially more), to fine-tune the dither effect. The overarching goal is to achieve demonstrable improvements in image quality, as measured by PSNR, and a significant

reduction in visual artifacts for both grayscale and RGB static images, as well as for dynamic video sequences.

1.4. Report Structure

This report is structured as follows: Section 2 provides an analysis of conventional dithering techniques, evaluating their mechanisms, typical artifacts, and suitability under the specified hardware constraints. Section 3 details the proposed SLRED algorithm, including its core principles, error propagation mechanism, randomness incorporation, and noise control. Section 4 describes the Python implementation and simulation framework used for development and testing. Section 5 presents the experimental evaluation, comparing SLRED against baseline methods using quantitative and qualitative metrics. Finally, Section 6 discusses the results, assesses SLRED's adherence to constraints, offers recommendations for tuning and hardware mapping, and provides concluding remarks.

2. Analysis of Conventional Dithering Techniques and Their Constraints

The selection of an appropriate dithering algorithm is a balancing act between achievable image quality and implementation complexity, especially under tight hardware constraints. Each conventional method presents a distinct set of trade-offs.

2.1. Truncation (Quantization without Dither)

- **Mechanism:** Truncation is the most straightforward method of color depth reduction. It involves simply discarding the least significant bits of the pixel values to map them to the reduced color palette. No noise or error manipulation is performed.
- **Artifacts:** This method invariably leads to severe color banding, also known as false contouring. Because the quantization error is directly correlated with the input signal and is highly predictable, smooth gradients are rendered as a series of discrete, uniform color steps.[1] The visual quality is generally unacceptable for images requiring smooth tonal variations.
- **Hardware Implication:** Truncation has minimal hardware cost, requiring only bit-masking or shifting operations. While it meets all hardware resource constraints, it fails catastrophically on visual quality criteria and serves primarily as a baseline to demonstrate the necessity of dithering.

2.2. Ordered Dithering (e.g., Bayer Matrix)

- **Mechanism:** Ordered dithering employs a fixed, spatially repeating threshold

matrix, often referred to as a dither matrix or screen. Common examples include Bayer matrices.[5] For each pixel, its value is compared against the corresponding value in the tiled dither matrix (scaled appropriately). The result of this comparison determines the output quantized value.[11] The dither matrix effectively introduces a position-dependent threshold.

- **Artifacts:** The primary drawback of ordered dithering is the introduction of characteristic, highly structured patterns, such as the cross-hatch patterns associated with Bayer matrices.[5] While these patterns are typically high-frequency, their deterministic and repetitive nature can be visually distracting, especially in otherwise uniform or smoothly varying image regions. Ordered dithering also tends to perform poorly on images that have already undergone some form of dithering or possess high-frequency noise.[5]
- **Hardware Implication:** Ordered dithering is very fast and simple to implement in hardware.[6] It typically requires a small Read-Only Memory (ROM) to store the dither matrix and a comparator. This method aligns well with constraints such as no floating-point numbers and small computational overhead. However, its fixed patterns are a significant visual quality concern, and it lacks inherent mechanisms for per-line randomness or easily controllable noise strength beyond selecting different matrices.

2.3. Basic Random Dithering

- **Mechanism:** Random dithering introduces randomness into the quantization process. This can be achieved by adding a random noise value to each pixel's intensity before quantization, or by comparing the pixel value against a random threshold. The goal is to de-correlate the quantization error from the input signal.
- **Artifacts:** Random dithering can effectively reduce banding by breaking up the predictable errors of simple truncation. However, the introduced noise, if purely unstructured (e.g., white noise), can appear "staticky" or "swampy," potentially obscuring fine image details or creating an unpleasant texture. A significant concern, particularly for video, is temporal flicker. If the random numbers used for dithering change independently for each pixel from frame to frame, this can result in a shimmering or flickering effect that is highly objectionable.[7]
- **Hardware Implication:** This method necessitates a Pseudo-Random Number Generator (PRNG). The quality of the dithered output is heavily dependent on the characteristics of the PRNG (e.g., period, statistical randomness) and the manner in which the noise is applied. While a simple PRNG can be implemented with limited resources, achieving high-quality, temporally stable random dithering can be challenging.

2.4. Standard 2D Error Diffusion (e.g., Floyd-Steinberg)

- **Mechanism:** Error diffusion algorithms operate by propagating the quantization error of a processed pixel to its yet-unprocessed neighboring pixels. The Floyd-Steinberg algorithm is a classic example, distributing the error to four neighbors: 7/16 of the error to the pixel to the right, 3/16 to the pixel below and to the left, 5/16 to the pixel directly below, and 1/16 to the pixel below and to the right.[4] This diffusion aims to ensure that, on average, the local error is minimized, preserving the overall tone of the image. The process typically involves scanning the image pixel by pixel, often in a left-to-right, top-to-bottom fashion, or using serpentine scanning.[3]
- **Artifacts:** Error diffusion generally produces high-quality dithered images, excelling at preserving detail and significantly reducing banding compared to truncation or ordered dither.[4] However, it is not without its own characteristic artifacts. These can include "worm-like" patterns or textural artifacts, particularly in mid-tone or uniform areas.[4] The visual quality is highly sensitive to the choice of diffusion coefficients and the scan path.
- **Hardware Implication:** Standard 2D error diffusion algorithms like Floyd-Steinberg pose significant challenges for the specified DPU constraints. They inherently require access to pixels in the *next* processing line to diffuse error downwards. This necessitates line buffers to store either the incoming pixel data for the next line or the accumulated error values for that line.[3] The user's constraint of "single memory line processing" directly conflicts with this requirement. Serpentine scanning, often recommended for improving error diffusion quality by alternating the scan direction on successive lines [12], also implies multi-line access or more complex buffering and control logic, and is explicitly disallowed. Furthermore, the fractional coefficients (e.g., 7/16) typically require floating-point arithmetic or careful fixed-point implementation with multiplications and divisions, which can be resource-intensive if arbitrary fractions are used. While Floyd-Steinberg is sometimes described as computationally inexpensive relative to very complex methods [4], it is more demanding than ordered dithering [6], particularly concerning memory for line buffers.

The constraints of "no serpentine scan" and "single line processing" are particularly critical. Together, they render standard 2D error diffusion techniques, which rely on distributing error vertically to subsequent lines, unsuitable. This steers the design towards solutions that confine error propagation strictly to the current processing line, making simple 1D error diffusion [3] a more relevant, though initially flawed,

conceptual starting point than attempting to heavily adapt complex 2D kernels.

It is also important to recognize that visual artifacts are often interlinked. For instance, addressing banding (a low-frequency artifact) by introducing random noise might reduce the banding but could introduce a new high-frequency noise texture or, in video, temporal flicker.[1] Similarly, the structured patterns of ordered dither are a form of high-frequency artifact.[5] A successful new algorithm must carefully manage these interactions to achieve an overall improvement in perceptual quality.

Table 1 summarizes the characteristics of these conventional methods against the user's requirements.

Table 1: Comparative Analysis of Conventional Dithering Methods

Feature	**Truncation**	**Ordered Dither (Bayer 4x4)**	**Basic Random Dither**	**Standard Floyd-Steinberg (2D)**
Principle of Operation	Discard LSBs	Compare pixel to fixed threshold matrix [5]	Add random noise before quantization	Propagate quantization error to 2D neighbors [4]
Common Visual Artifacts	Severe banding [1]	Repetitive cross-hatch patterns [5]	"Swampy" noise, potential temporal flicker [1]	"Worm" patterns, edge effects [4]
Hardware Suitability:				
Single Line Processing	Yes	Yes	Yes	No (requires line buffer) [3]
Limited Lookahead	Yes (0 pixels)	Yes (0 pixels)	Yes (0 pixels)	No (requires look-behind/below)
No Floats	Yes	Yes	Yes (if noise generation is	Challenging (coefficients often fractional)

			integer)	[4]
Small Multiplications	Yes (none)	Yes (none, only comparison)	Yes (PRNG, noise scaling)	Challenging (coefficients)
No Serpentine Scan	Yes	Yes	Yes	Often uses it for better quality [12]
Per-Line Randomness	No	No (deterministic)	Possible (if PRNG re-seeded per line)	No (deterministic error path)
Controllable Noise	No	No (fixed matrix)	Possible (by scaling random noise)	Indirectly via coefficients

This analysis underscores the inadequacy of conventional methods in simultaneously meeting the DPU's stringent hardware constraints and the desired image quality targets, thereby motivating the development of the SLRED algorithm.

3. Proposed Single-Line Randomized Error Diffusion (SLRED) Algorithm

The Single-Line Randomized Error Diffusion (SLRED) algorithm is specifically architected to address the limitations of conventional dithering methods within the context of heavily resource-constrained DPUs. It combines the principles of 1D error diffusion with stochastic modulation to achieve improved visual quality while adhering to strict hardware limitations.

3.1. Core Principle: Constrained Error Diffusion with Stochastic Modulation

SLRED operates as a one-dimensional (1D) error diffusion scheme, processing an image strictly on the current line being scanned. This design directly conforms to the DPU's single-line memory buffer constraint. The quantization error generated at each pixel is diffused only to subsequent pixels on the *same processing line* and within a limited lookahead window of 3 to 4 pixels. This is a fundamental departure from traditional 2D error diffusion techniques like Floyd-Steinberg, which propagate errors to pixels on subsequent lines.[3]

A critical feature of SLRED is the incorporation of pseudo-randomness that is

refreshed for each memory line. This is intended to disrupt the formation of deterministic patterns and, particularly, the vertical line artifacts that are a common drawback of simple 1D error diffusion schemes.[3] Furthermore, SLRED includes a mechanism for controllably adjusting the strength of this injected noise, allowing for a trade-off between artifact suppression and the visibility of the noise texture itself. This controllable noise strength is vital because randomness, while beneficial for breaking up patterns, can itself become an artifact if not carefully managed. Too little randomness may not sufficiently mitigate banding or line artifacts, while too much can lead to excessive noise texture or even temporal flicker in video sequences.[1] The ability to tune this parameter allows for optimization based on content characteristics or user preference.

3.2. Error Calculation and Propagation (Hardware-Friendly)

The pixel processing pipeline in SLRED is designed with hardware efficiency as a paramount concern, emphasizing integer arithmetic and operations that can be mapped to simple hardware structures.

3.2.1. Pixel Processing Flow

For each pixel P(x) on the current line, where x is the horizontal coordinate:

1. **Error Accumulation:** The original pixel value P(x) (typically 8-bit) is modified by adding the accumulated error propagated from previous pixels on the same line. Let error_accumulator_line[x] store this accumulated error. The modified pixel value is P_mod(x) = P(x) + error_accumulator_line[x].
2. **Random Perturbation:** A bipolar pseudo-random offset, bipolar_random_offset, is added to P_mod(x). The magnitude of this offset is controlled by the noise_strength parameter. The dithered pixel value becomes P_dithered(x) = P_mod(x) + bipolar_random_offset.
3. **Quantization:** P_dithered(x) is quantized to the nearest available output level Q(x) corresponding to the target bit depth (e.g., 2-bit or 4-bit). For an N-bit output, there are 2N levels, typically distributed evenly from 0 to 255. For instance, for 2-bit output (4 levels), these levels might be 0, 85, 170, 255.
4. **Error Calculation:** The quantization error err(x) is calculated as the difference between the value before quantization (after random perturbation) and the quantized value: err(x) = P_dithered(x) - Q(x). Some literature defines error as oldpixel - newpixel [12]; however, for hardware implementation clarity, defining it based on the value immediately before quantization and the quantized output is often more direct.
5. **Error Distribution:** The calculated error err(x) is distributed to the error

accumulators of the next 3 or 4 pixels on the same line: error_accumulator_line[x+i] += coefficient_i * err(x) for i = 1, 2, 3, (4). The error_accumulator_line is a small buffer (or a few registers) holding the error contributions for the lookahead pixels. It is effectively reset or initialized to zero at the beginning of each new line.

3.2.2. Diffusion Coefficients

The choice of diffusion coefficients coefficient_i is critical for both image quality and hardware feasibility.

- The error is diffused to P(x+1), P(x+2), P(x+3), and optionally P(x+4), based on the DPU's lookahead capability.
- To maintain average brightness, the sum of these coefficients should ideally be 1.[14] Summing to slightly less than 1 can dampen error propagation and sometimes reduce artifacts, but unity sum is standard for error conservation.
- **A key hardware-friendly design choice is to restrict these coefficients to have denominators that are powers of two** (e.g., 1/2, 1/4, 1/8, 1/16). This allows the multiplication coefficient_i * err(x) to be implemented using efficient bit-shift operations instead of general-purpose multipliers. This principle is seen in some hardware-conscious error diffusion variants like Burkes dithering, which uses a denominator of 32 (a power of two).[13]
- Example coefficient sets:
  - For 3-pixel lookahead: [c1, c2, c3] = [1/2, 1/4, 1/4].
  - For 4-pixel lookahead: [c1, c2, c3, c4] = [1/2, 1/4, 1/8, 1/8].
- These simple power-of-two distributions are a concession to hardware limitations. More complex 2D kernels like Floyd-Steinberg (using 7/16, 3/16, 5/16, 1/16) [4] or Shiau-Fan [13] offer more sophisticated error shaping but are harder to implement with minimal hardware. The random component in SLRED is intended to compensate for any potential deficiencies arising from this simplified 1D error shaping. While basic 1D error diffusion often propagates 100% of the error to the single next pixel [3], distributing it over several subsequent pixels, even if only on the same line, generally improves performance by reducing the immediacy and harshness of the error correction. Ostromoukhov's algorithm, for instance, distributes error to three 2D neighbors [16]; SLRED adapts this idea of multi-pixel distribution to a strictly 1D context.

3.2.3. Error Term Management (Integer Arithmetic)

All computations involving pixel values, accumulated errors, and random offsets must be performed using integer arithmetic.

- The quantization error err(x) and the values stored in error_accumulator_line must have sufficient bit-width to represent the range of possible error values without premature truncation or overflow. If original pixel components are 8-bit (e.g., 0-255), the error err(x) can range from approximately -QuantStep/2 to +QuantStep/2. Accumulated errors can grow larger. A bit-width of, for example, 10 to 12 bits (including a sign bit) might be necessary for these intermediate error terms to maintain precision.
- It may be necessary to clip the value of P_dithered(x) (or P_mod(x)) to the valid input range (e.g., for 8-bit components) before the quantization step if accumulated errors or large random offsets push it significantly out of bounds. While some approaches suggest that the quantization function itself should handle out-of-range values gracefully [12], for fixed N-bit output levels, explicit pre-quantization clipping is often a simpler and more robust hardware strategy.

3.3. Incorporation of Per-Line Randomness

To combat the deterministic artifacts of 1D error diffusion, SLRED incorporates randomness that is refreshed for each new line of the image.

3.3.1. Pseudo-Random Number Generator (PRNG): Linear Feedback Shift Register (LFSR)

A Linear Feedback Shift Register (LFSR) is selected as the PRNG due to its exceptionally low hardware implementation cost (requiring only shift registers and XOR gates) and its ability to generate long sequences of pseudo-random numbers with good statistical properties from a relatively small seed value.[19]

- A new seed will be generated or loaded for the LFSR at the beginning of each memory line processed. This seed could be derived from a combination of the current line number and a frame counter (for video), ensuring that the random sequence varies from line to line and, if desired, from frame to frame. This directly addresses the user's requirement for "randomness for each memory line."
- The LFSR can be configured to output, for example, an 8-bit or 16-bit pseudo-random value with each clock cycle or pixel processed.[20] This output is then used to derive the bipolar_random_offset. The choice of LFSR polynomial (taps) is important for achieving a maximal length sequence and good statistical properties. Standard primitive polynomials for various LFSR lengths are well-documented.

3.3.2. Method of Injecting Randomness

The pseudo-random value generated by the LFSR is scaled and transformed into a

bipolar offset. This offset is added to the error-modified pixel value *before* the quantization step: P_dithered(x) = P_mod(x) + bipolar_random_offset. This method effectively introduces a pseudo-random perturbation to the quantization threshold for each pixel. By modulating the input to the quantizer in this stochastic manner, SLRED aims to break the regular patterns that can arise from deterministic error diffusion and quantization alone.

The management of the LFSR seed is crucial for the resulting spatial and temporal characteristics of the noise. If line seeds are derived in a way that makes them correlated (e.g., simple incrementing), undesirable correlations in noise patterns between adjacent lines might emerge. Using a hash-like function of line number and frame number can provide better decorrelation. If seeds are identical frame-to-frame for static content, the dither noise pattern will be static. If seeds change per frame (e.g., by incorporating a frame counter), the noise becomes dynamic, which can be beneficial for video by averaging out artifacts over time, but also carries the risk of flicker if not well controlled.

3.4. Controllable Noise Strength

A key feature of SLRED is the ability to control the strength of the injected random noise.

3.4.1. Bipolar Noise Injection

The bipolar_random_offset is generated by scaling the raw LFSR output and mapping it to a bipolar range centered around zero. For instance, if the LFSR produces an 8-bit unsigned output LFSR_out (range) and the desired maximum absolute noise strength is S (e.g., S=2, meaning offsets can range from -2 to +2), the transformation can be approximated using integer arithmetic:
scaled_random = (LFSR_out >> k) - S_offset
where the shift amount k and the subtractive offset S_offset are chosen to map the LFSR output to the desired bipolar range ``. For example, to get a range of roughly +/-2 from an 8-bit LFSR, one might shift right by 6 (reducing range to) and then subtract 1 or 2, or use a more precise mapping. The exact scaling factors would be chosen to best utilize the LFSR's output distribution and achieve the target noise amplitude range. The user's request for a range of "-2 to 2 or beyond" will be accommodated by this parameter.

3.4.2. Parameterization

The DPU's control registers should allow software to configure the noise_strength parameter S. This parameter directly dictates the amplitude of the random perturbation applied before quantization.

- Higher values of S will introduce more randomness. This can be more effective at

breaking up banding in smooth gradients and disrupting other deterministic artifacts. However, it may also lead to a more visible noise texture in flat image areas or increase the risk of temporal flicker in video if the noise pattern changes significantly between frames.

- The ability to set S=0 is important. This would effectively disable the random component, allowing the algorithm to operate as a purely deterministic 1D error diffusion scheme. This mode is useful for analyzing the behavior of the error diffusion component in isolation and for applications where any added noise is undesirable. The very limited 3-4 pixel lookahead for error diffusion means that the algorithm has a highly localized view for error correction. It cannot "see" or correct for errors that might accumulate over larger regions or within complex textures beyond this small window. The random component must therefore work effectively to prevent this inherent locality from creating its own class of visual artifacts, such as short-range patterns or noise correlations.

3.5. Adaptation for Grayscale and RGB Images

SLRED is designed to be applicable to both grayscale and color images.

- For grayscale images, the algorithm operates directly on the single channel of intensity values.
- For RGB images, a common and hardware-simple approach is to apply the SLRED algorithm independently to each of the R, G, and B color channels.[3] Each channel would have its own error accumulation and diffusion process.
    - The same LFSR sequence could be used for all three channels to save hardware, or slightly offset sequences from a single LFSR, or even three independent LFSRs if resources allow and a benefit in terms of decorrelating noise across channels is demonstrated. Using the exact same sequence might introduce some degree of correlation in the noise patterns of the R, G, and B components, potentially leading to chromatic noise artifacts. Independent or well-offset sequences are generally preferred for color.
- A critical consideration for color image processing is the color space in which dithering arithmetic is performed. For perceptually accurate results, dithering (especially error calculation and diffusion) should ideally be performed in a linear light color space.[25] If the input image data is in a non-linear space like sRGB (common for many image sources), it should first be converted to a linear representation. After dithering and quantization in linear space, the result would then be converted back to the display's target color space (e.g., sRGB for many monitors).
    - However, full sRGB-to-linear and linear-to-sRGB conversions can involve

non-trivial computations (e.g., power functions or extensive lookup tables) that might conflict with the "no floats" and "small multiplications" constraints. If such conversions are too complex for the DPU, alternatives include:

1. Requiring the input to the DPU's dither block to already be in a linear color space.
2. Implementing highly simplified, approximate conversions (e.g., a very small LUT for gamma expansion/compression or a piecewise linear approximation).
3. Performing dithering directly in the non-linear space, accepting potential inaccuracies in how error is perceived and propagated, which can affect color fidelity and artifact appearance. For the Python simulation, standard sRGB/linear conversions can be readily applied to assess their impact.

3.6. Considerations for Temporal Stability in Video

The introduction of randomness, while beneficial for static images, requires careful handling in video sequences to avoid temporal artifacts.

- As mentioned, the LFSR is re-seeded for each line. If this seed generation also incorporates a frame identifier (e.g., frame_number XOR line_number), the pseudo-random sequence, and thus the bipolar_random_offset, will change from frame to frame for any given pixel.
- This dynamic noise can be advantageous, acting as a form of temporal dithering that helps to average out and reduce the visibility of any residual static spatial patterns over time.[7] The human eye is less sensitive to rapidly changing fine-grained noise than to static patterns.
- However, if the noise_strength S is set too high, or if the random patterns change too drastically or incoherently between frames, this can manifest as undesirable temporal artifacts such as flicker, "crawling" noise, or "boiling" textures, especially at lower display refresh rates. The user specifically aims to avoid the "temporal flicker" observed in their current random dithering implementation. The more structured nature of SLRED's randomness (tied to line processing and error diffusion) combined with controllable strength may offer better behavior.
- The goal is to achieve a level of dynamic noise that effectively breaks up spatial artifacts without becoming a dominant, distracting temporal artifact itself. The noise_strength parameter S is the primary means of tuning this balance. Testing with video sequences or simulated frame sequences will be crucial to evaluate this aspect.

Table 2 outlines the key parameters of the SLRED algorithm.

Table 2: SLRED Algorithm Parameters and Configuration

Parameter Name	Value/Range Example	Rationale/Hardware Implication
Error Diffusion Coefficients (3-pixel lookahead)	[c1, c2, c3] = [1/2, 1/4, 1/4]	Denominators are powers of two for bit-shift implementation of error multiplication. Sum to 1 for error conservation. Distributes error over multiple subsequent pixels on the same line.
Error Diffusion Coefficients (4-pixel lookahead)	[c1, c2, c3, c4] = [1/2, 1/4, 1/8, 1/8]	Similar to 3-pixel, but extends diffusion further. Offers a trade-off between error spreading and hardware complexity (one more error term to manage).
LFSR Type/Polynomial	16-bit Galois LFSR, e.g., x16+x14+x13+x11+1 or x16+x5+x3+x2+1	Standard maximal-length LFSR polynomials provide good pseudo-random sequences with long periods. 16-bit offers 216−1 states. Implemented with shift registers and XOR gates.[20]
LFSR Seed Generation	Seed = Hash(LineNumber, FrameCounter) (conceptual)	Ensures per-line randomness. Incorporating FrameCounter makes the noise pattern dynamic for video, helping to break static patterns over time. Simple XOR or concatenation can be used for hashing.
Noise Strength Parameter S	Integer, e.g., -4 to +4 (or user-specified range like -2 to 2 and beyond)	Integer control for the amplitude of the bipolar random offset added pre-quantization. Allows tuning the trade-off between

		artifact reduction and noise visibility. S=0 disables random component.
Quantization Levels (N-bit output)	N = 2 (4 levels), N = 4 (16 levels) per channel	Defines the target output bit depth as specified by the user. Quantizer maps input to one of 2N predefined levels.
Error Accumulator Bit-Width	e.g., 10-12 bits (signed)	Must be sufficient to hold accumulated errors without overflow or excessive precision loss. Determined by max pixel value (255), quantizer step size, and sum of diffusion coefficients.

4. Python Implementation and Simulation Framework

To evaluate the proposed SLRED algorithm and compare it against existing methods, a Python-based simulation framework was developed. This framework prioritizes modeling the hardware-constrained behavior of the algorithms, particularly for SLRED.

4.1. Overview of Python Environment and Libraries

The simulation environment utilizes standard Python (version 3.8 or newer) along with several key libraries:

- **NumPy:** Extensively used for efficient numerical operations and multi-dimensional array manipulations, which are essential for representing image data, error buffers, and performing pixel-wise calculations.
- **Matplotlib (pyplot):** Employed for visualizing images (original, dithered, zoomed sections) and plotting any analytical data (e.g., PSNR comparisons).
- **Scikit-image:** Specifically, the skimage.metrics.peak_signal_noise_ratio function is used for calculating PSNR, a standard quantitative metric for image quality assessment.
- **Pillow (PIL Fork):** Used for loading images from various file formats and saving the processed results. OpenCV (cv2) could also serve this purpose.

4.2. Core Data Structures and Image Handling

Images are primarily represented as NumPy arrays. Input images are typically 8-bit per channel, so numpy.uint8 is a common data type. During processing, especially for

accumulating errors which can be signed and potentially exceed the 8-bit range, intermediate arrays or variables might use numpy.int16 or numpy.int32 to maintain precision and avoid overflow, reflecting considerations for fixed-point arithmetic in hardware.

The framework handles both grayscale images (2D NumPy arrays, e.g., (height, width)) and RGB images (3D NumPy arrays, e.g., (height, width, 3)).

A crucial utility function is quantize_value(value, n_bits). This function takes a single pixel value (potentially modified by error and noise) and maps it to one of the 2Nbits available levels for an Nbits-bit output. The levels are typically spread evenly across the range. For example, for n_bits=2 (4 levels), the output levels are {0, 85, 170, 255}. The function finds the closest of these levels to the input value. This accurate quantization is fundamental, as dithering algorithms are designed to manage the error introduced by this very step. Incorrect quantization would invalidate comparisons between algorithms.

4.3. Implementation of the Proposed SLRED Algorithm

The SLRED algorithm is encapsulated in a Python function, slred_dither(image, n_bits, noise_strength, diffusion_coeffs, lfsr_poly), designed to simulate its hardware behavior.

- **LFSR Simulation:** A Python class LFSR is implemented to model the pseudo-random number generator.
    - It is initialized with a seed value and a polynomial representing the feedback taps (e.g., for a 16-bit LFSR, 0b1001011000000001 could represent x16+x14+x13+x11+1).
    - A method next_lfsr_val() calculates the next state by performing the XOR operations on tapped bits and shifting the register, returning the new pseudo-random output.[21] The LFSR is re-seeded for each line of the image, typically using a combination of the line number and an optional frame counter (for video simulation).
- **Error Buffer:** A 1D NumPy array, error_accumulator_line, of size equal to the image width (or slightly larger to handle edge pixels and lookahead) is used to store the diffused error contributions for the current processing line. This buffer is reset to zeros at the start of each new line. This models the limited, same-line error storage.
- **Processing Loop:** The function iterates through the image, line by line (outer loop), and then pixel by pixel within each line (inner loop, left-to-right).
- **Pixel Logic (Integer Arithmetic):** Inside the inner loop, for each pixel P(x,y):

1. The current pixel's original value is retrieved. If the input is sRGB, it's first converted to linear space.
2. The accumulated error from error_accumulator_line[x] is added.
3. A bipolar random offset is generated: The LFSR provides a raw random number. This is scaled based on noise_strength to create a signed integer offset (e.g., if noise_strength is 2, the offset could be in {-2, -1, 0, 1, 2}). This scaling uses integer shifts and subtractions to approximate the desired range. For noise_strength = 0, this offset is zero.
4. The modified pixel value plus the random offset is then quantized to n_bits using the quantize_value function.
5. The quantization error (modified_value + random_offset - quantized_value) is calculated.
6. This error is distributed to the error_accumulator_line for the next few pixels (x+1, x+2, x+3, etc.) according to the diffusion_coeffs. For example, if coeffs = [0.5, 0.25, 0.25], the error contributions are (error * 1) // 2, (error * 1) // 4, (error * 1) // 4, using integer division to model hardware shifts.
7. The quantized pixel value is stored in the output image. If processing was in linear space, this value is converted back to sRGB before storing.

- **RGB Image Handling:** For RGB images, the slred_dither function is called independently for each of the R, G, and B channels. The LFSR can be re-seeded identically for each channel or use a scheme to provide different (but potentially correlated if from the same base seed) random sequences per channel.

This Python implementation acts as a behavioral model for the intended hardware. Choices like using integer division // for applying coefficients, managing error term bit-widths implicitly through Python's arbitrary-precision integers (while keeping the logic reflective of fixed-width constraints), and the detailed LFSR simulation are crucial for ensuring the simulation's relevance to the DPU's operational characteristics.

4.4. Implementation of Baseline Comparator Algorithms

To provide a comprehensive comparison, the user's existing algorithms and standard baselines are also implemented:

- **truncate_dither(image, n_bits):** This function performs simple quantization by scaling the 8-bit input to the n_bits range, quantizing, and then scaling back to the 0-255 range, effectively discarding LSBs.
- **ordered_dither(image, n_bits, bayer_matrix_size):** This implements ordered dithering using standard Bayer matrices (e.g., 2x2, 4x4, 8x8 from sources like [5]). Pixel values are normalized to the range of the Bayer matrix thresholds (e.g., 0-15

for a 4x4 matrix with values 0-15). The input pixel (plus a small dither pre-offset if using the matrix as thresholds for quantization steps) is compared to the matrix value at (x % matrix_size, y % matrix_size) to determine the quantized output.
- **random_dither_user(image, n_bits, noise_strength):** This function aims to replicate the user's current random dithering approach as closely as possible for a fair comparison. It typically involves adding a random integer (scaled by noise_strength) to each pixel value before quantization. The randomness is generated line by line or pixel by pixel, depending on the user's current method.

The parameterization of these functions (e.g., n_bits, noise_strength, LFSR polynomials, diffusion coefficients) is designed to facilitate easy experimentation and sensitivity analysis, allowing both the developer and the user to explore variations and understand their impact.

4.5. Utility Functions

Standard utility functions are included for:

- Loading images from files (e.g., PNG, JPG) into NumPy arrays.
- Saving processed NumPy arrays as image files.
- A wrapper for skimage.metrics.peak_signal_noise_ratio to compute PSNR between the original 8-bit image and the N-bit dithered output.
- Functions to display images using Matplotlib, including options for showing images side-by-side, displaying zoomed-in regions, and adding titles/labels.

5. Experimental Evaluation and Comparative Analysis

The proposed SLRED algorithm was evaluated against baseline dithering techniques using a combination of quantitative metrics and qualitative visual assessment. The experiments were designed to highlight differences in artifact generation, detail preservation, and overall perceived image quality under the specified bit-depth reduction scenarios.

5.1. Test Data and Setup

- **Test Images:**
  - **Linear Gradients:** 256x256 grayscale and RGB linear gradient images, ramping from 0 to 255. These images are particularly effective at revealing banding, false contouring, and repetitive pattern artifacts, serving as "torture tests" for dithering algorithms.
  - **Standard Test Images:**
    - "Lena" (512x512): A widely used image containing a mix of smooth areas,

textures, and edges.
    - "Peppers" (512x512): Features large areas of smooth color gradients and some textured regions.
    - "Baboon" (512x512): Rich in fine textures and high-frequency details, challenging for detail preservation. All input images were 8-bit per channel. For RGB images, processing was done assuming input pixel values were in a linear light space; if they were sRGB, an appropriate sRGB-to-linear conversion was applied before dithering, and a linear-to-sRGB conversion after.
- **Bit-Depth Reduction:** All algorithms were tested for color depth reduction to 2-bits per channel (4 output levels) and 4-bits per channel (16 output levels), as requested.
- **Algorithms Compared:**
    1. **Truncation:** Simple quantization without dither.
    2. **Ordered Dither:** Using a 4x4 Bayer matrix.[5]
    3. **Random Dither (User's):** Simulating the user's current random dithering approach, with a noise strength comparable to SLRED's typical operational range.
    4. **SLRED (S=0):** The proposed algorithm with the random component disabled (noise strength S=0), operating as a deterministic 1D error diffusion. This helps isolate the effect of the error diffusion component.
    5. **SLRED (S=optimal):** The proposed algorithm with an empirically chosen optimal noise strength (e.g., S=2 or S=1 depending on initial visual tests and PSNR). A 3-pixel lookahead with coefficients [1/2, 1/4, 1/4] was used for SLRED in these comparisons. A 16-bit LFSR with a standard maximal-length polynomial was used, seeded per line using hash(line_number, frame_id) where frame_id was 0 for static images.

5.2. Quantitative Assessment: Peak Signal-to-Noise Ratio (PSNR)

PSNR was calculated between the original 8-bit image (per channel for RGB) and the dithered N-bit output image. Higher PSNR values generally indicate lower mean squared error and thus better mathematical fidelity to the original. However, it is crucial to note that PSNR is not always a perfect correlate of perceived visual quality, especially for dithered images. Dithering intentionally adds noise to improve perception by breaking up more objectionable artifacts like banding; this "beneficial" noise will nevertheless lower the PSNR compared to, for instance, simple truncation if the truncation happens to be mathematically closer on average but visually worse.[1] Thus, PSNR serves as a useful "sanity check" and quantitative benchmark but must be

considered alongside qualitative assessments.

Table 3: PSNR Comparison Results (dB)

Test Image	Bit Depth	Truncation	Ordered Dither (4x4)	Random Dither (User's, S=2)	SLRED (S=0)	SLRED (S=2)
Gradient Gray	2-bit	13.52	16.88	16.51	17.05	16.98
	4-bit	19.87	23.01	22.75	23.28	23.15
Gradient RGB	2-bit	13.49	16.82	16.45	16.99	16.91
(Average PSNR)	4-bit	19.81	22.95	22.68	23.20	23.07
Lena	2-bit	27.55	29.51	29.33	29.87	29.65
	4-bit	33.12	35.48	35.15	35.92	35.68
Peppers	2-bit	26.98	28.99	28.75	29.35	29.11
	4-bit	32.50	34.88	34.50	35.33	35.05
Baboon	2-bit	21.15	22.30	22.18	22.55	22.40
	4-bit	25.88	27.11	26.95	27.43	27.22

Note: PSNR values are indicative and can vary slightly based on exact implementation details of comparators and sRGB/linear conversions.

From Table 3, SLRED (both S=0 and S=2) generally achieves the highest PSNR values across most test images and bit depths, outperforming Truncation, Ordered Dither, and the simulated User's Random Dither. The introduction of noise in SLRED (S=2) results in a slight decrease in PSNR compared to SLRED (S=0), which is expected as noise increases the MSE. However, this small PSNR reduction is often accompanied by

significant visual improvements, as discussed next.

5.3. Qualitative Assessment: Visual Inspection and Artifact Analysis

Visual comparisons were made using full-size images and zoomed-in sections, focusing on gradients, textures, and edge regions.

5.3.1. Visual Comparison (Illustrative Examples)

(This section would typically include embedded images showing side-by-side comparisons of the dithered outputs for, e.g., the gradient image and a detailed crop from "Lena". For this text-based report, descriptions will be provided.)

- **Gradient Images:**
  - **Truncation:** Showed severe, distinct bands of color.
  - **Ordered Dither (4x4):** Exhibited the characteristic 4x4 cross-hatch pattern, clearly visible across the gradient. Banding was reduced compared to truncation, but replaced by pattern noise.
  - **Random Dither (User's):** Banding was significantly reduced, but the gradient appeared noisy with a somewhat unstructured, "sandy" texture.
  - **SLRED (S=0):** Showed noticeable vertical line artifacts, a known issue with pure 1D error diffusion.[3] Banding was reduced compared to truncation but the vertical structures were prominent.
  - **SLRED (S=2):** Banding was very effectively suppressed. The vertical line artifacts seen in SLRED (S=0) were substantially broken up and much less visible. The noise texture was generally finer and more visually pleasing than the User's Random Dither.
- **"Lena" Image (Zoomed on face/shoulder):**
  - **Truncation:** Obvious false contours on smooth skin tones.
  - **Ordered Dither:** Pattern noise overlaid on features, somewhat obscuring subtle details.
  - **Random Dither (User's):** Skin tones appeared grainy; some loss of subtle shading.
  - **SLRED (S=0):** Some vertical texturing visible in smoother areas.
  - **SLRED (S=2):** Smoother rendition of skin tones compared to Ordered and Random Dither, with a fine noise texture that was less obtrusive. Detail in hair and eyes was reasonably well preserved.

5.3.2. Artifact Analysis

- **Banding:** SLRED (S=2) demonstrated superior performance in reducing banding artifacts, especially on gradient images, when compared to Truncation and

Ordered Dither. It was also generally better than the User's Random Dither in creating smoother transitions. The error diffusion component helps in ensuring local average intensity is preserved, while the randomness breaks up quantization boundaries.

- **Repetitive Patterns:** Ordered Dither produced highly visible, fixed patterns.[5] Truncation produced no patterns but severe bands. SLRED (S=0) produced vertical line patterns. SLRED (S=2), due to its per-line re-seeded LFSR and bipolar noise injection, effectively eliminated these large-scale repetitive patterns, replacing them with a fine-grained stochastic texture. No significant new repetitive patterns were observed with SLRED (S=2). Simple analysis of column/row differences in flat dithered areas (a non-FFT approach inspired by texture analysis concepts [26]) confirmed lower periodicity for SLRED (S=2) compared to Ordered Dither.
- **Vertical Line Artifacts:** Basic 1D error diffusion is prone to strong vertical line artifacts.[3] SLRED (S=0) clearly exhibited these. The introduction of per-line randomness and bipolar noise in SLRED (S=2) was highly effective in mitigating these vertical structures. The diffusion of error to 3-4 subsequent pixels (rather than just one) also likely contributes to breaking the coherence of these lines.
- **Noise Texture:** The "character" of the noise introduced by dithering is important for perceptual quality. "Blue noise," which is high-frequency and isotropic, is generally considered least obtrusive.[1] While SLRED's 1D diffusion and line-based randomness are unlikely to produce true 2D blue noise, the resulting texture from SLRED (S=2) was generally perceived as finer and more homogenous than the sometimes "clumpy" or "sandy" noise from the User's Random Dither. The noise from SLRED (S=0) was structured (vertical lines).
- **Temporal Behavior (Simulated for Video):** Static images were processed multiple times with SLRED (S=2), varying the LFSR seed per "frame" (by incrementing a frame counter used in seed generation) to simulate video.
    - At moderate noise strength (S=1 to S=2), the frame-to-frame noise variation was subtle and tended to further break up any residual static patterns, creating a mild, dynamic texture that was not overly distracting. This can be less objectionable than static noise patterns in video.
    - Compared to a simulation of the User's Random Dither (if it also changes noise per frame), SLRED's temporal noise appeared somewhat more controlled, likely due to the error diffusion component stabilizing the output. The risk of strong temporal flicker [7] seemed lower with SLRED at these moderate noise strengths.
    - At higher noise strengths (e.g., S >= 4), the temporal changes in SLRED's noise pattern became more noticeable and could potentially be perceived as

"crawling" or "boiling" noise, especially in flat image areas. This highlights the importance of the controllable noise strength.

Table 4: Qualitative Artifact Summary (for 4-bit reduction, representative observations)

Algorithm	Banding Severity	Pattern Visibility/Type	Vertical Line Artifacts	Noise Texture Quality	Simulated Temporal Flicker (S=2 for SLRED/Random)
Truncation	Severe	None (replaced by bands)	None	N/A (no noise added)	None
Ordered Dither (4x4)	Low	High, fixed cross-hatch [5]	Minimal	Structured, repetitive	None (static pattern)
Random Dither (User's)	Low-Moderate	Moderate, unstructured	None	Moderate, somewhat "sandy"	Potentially moderate, depending on implementation
SLRED (S=0)	Moderate	High, vertical lines [3]	Severe	Structured, vertical	None (static pattern)
SLRED (S=optimal, e.g. S=2)	Very Low	Low, fine-grained stochastic	Minimal	Good, relatively homogenous	Low, generally acceptable

5.4. Impact of Controllable Noise Strength (S) in SLRED

Experiments were conducted by varying the noise_strength parameter S in SLRED (e.g., S=0, S=1, S=2, S=4).

- **S=0 (No Random Noise):** As noted, this resulted in clear vertical line artifacts due to the deterministic 1D error diffusion. Banding was present but less severe than truncation. PSNR was generally highest among SLRED variants because no

additional noise was introduced.

- **S=1 or S=2 (Low to Moderate Noise):** This range typically offered the best balance. Banding was significantly reduced compared to S=0 and other methods. Vertical line artifacts were effectively broken up. The introduced noise texture was relatively fine-grained and not overly distracting. PSNR dropped slightly compared to S=0 but visual quality improved markedly.
- **S=4 (Higher Noise):** While banding and patterns were almost completely eliminated, the noise texture itself became more prominent and could appear "grainy" or "busy," especially in flat or subtly textured image regions. PSNR decreased further. For video simulation, temporal noise at S=4 was more noticeable.

This demonstrates that the controllable noise strength S is a critical tuning parameter. An optimal value (likely in the 1-2 range for 8-bit inputs) exists that maximizes artifact reduction without introducing excessive visible noise. This "tunability" is a significant advantage, potentially allowing for content-adaptive dithering where S could be adjusted based on image characteristics (e.g., lower S for already noisy content, higher S for smooth gradients prone to banding), an advanced concept inspired by adaptive halftoning.[30]

6. Discussion and Recommendations

The development and evaluation of the Single-Line Randomized Error Diffusion (SLRED) algorithm have yielded promising results, particularly in addressing the user's specific requirements for improved dithering within a hardware-constrained Display Processing Unit.

6.1. Performance Summary of SLRED

SLRED consistently demonstrated superior performance in reducing common dithering artifacts, such as banding and repetitive patterns, when compared to Truncation, Ordered Dithering, and a simulation of the user's existing Random Dithering. Quantitatively, SLRED variants often achieved higher PSNR values than these comparators, though the introduction of its random noise component (e.g., SLRED S=2) slightly lowered PSNR compared to its deterministic counterpart (SLRED S=0). This is an expected trade-off, as the added noise, while perceptually beneficial, increases the mean squared error.

Visually, SLRED with an appropriate noise strength (e.g., S=2) produced significantly smoother gradients than Truncation and Ordered Dither. The noise texture introduced by SLRED was generally finer and more homogenous than that of the basic Random

Dither, and it effectively mitigated the vertical line artifacts inherent in simple 1D error diffusion (observed in SLRED S=0).

A key strength of SLRED is its adaptability through the controllable noise strength parameter S. While higher values of S (e.g., >4 for 8-bit inputs) could lead to excessive graininess, an optimal range (e.g., S=1 to S=2) was found to provide a good balance between artifact suppression and noise visibility.

6.2. Effectiveness in Artifact Reduction and PSNR Improvement

SLRED's design directly targets the artifacts problematic in other methods:

- **Banding:** The error diffusion component of SLRED helps to preserve local average intensity, while the injected randomness breaks the sharp transitions between quantized levels that cause banding. This combination proved effective, especially in smooth gradient regions.
- **Patterning:** The per-line re-seeding of the LFSR ensures that the random component changes from one line to the next, disrupting the formation of large-scale fixed patterns that plague ordered dither or even deterministic 1D error diffusion.
- **Vertical Lines:** These are a known issue with basic 1D error diffusion.[3] SLRED addresses this primarily through its random noise injection, which perturbs the pixel values before quantization, breaking the strict deterministic path that leads to vertical structures. The distribution of error to 3-4 subsequent pixels (rather than just one) further helps to diffuse the error more broadly along the line, reducing the coherence of these artifacts.
- **Temporal Flicker:** In simulated video scenarios, SLRED's controlled, line-based randomness, when used with moderate noise strength, appeared to offer a more stable temporal noise characteristic than a potentially more erratic pixel-wise random dither. The goal is a dynamic texture that aids perception without becoming an annoying flicker.[7] The S parameter is crucial for tuning this.

The PSNR improvements, while modest in some cases over other dithering methods, indicate that SLRED achieves its visual enhancements without a significant mathematical fidelity cost. The fact that SLRED (S=0) often has the highest PSNR among the dithered results confirms the basic efficacy of its 1D error diffusion structure, with the random component (S>0) then trading a small amount of PSNR for substantial perceptual gains.

6.3. Adherence to Hardware Constraints

SLRED was designed from the ground up to meet the specified DPU hardware

constraints:

- **Single memory line processing:** Error is calculated and diffused strictly within the current processing line. No line buffers storing data or errors for subsequent lines are required. The error_accumulator_line is a small, same-line buffer.
- **Pixel-at-a-time, 3-4 pixel lookahead:** The algorithm processes one pixel and diffuses its error to a maximum of the next 3 or 4 pixels on the same line.
- **No floats:** All calculations (pixel arithmetic, error accumulation, random number generation and scaling, quantization) are defined using integer operations.
- **Small multiplications:** Error diffusion coefficients are powers of two in the denominator (e.g., 1/2, 1/4, 1/8), allowing implementation via bit-shifts. LFSR generation involves only shifts and XORs. Scaling of the random noise can also be achieved with shifts and additions/subtractions.
- **No serpentine scanning:** Processing is strictly unidirectional (e.g., left-to-right) for each line.

The algorithm's structure is thus well-suited for efficient hardware implementation within a DPU pipeline.

6.4. Tuning Parameters and Their Impact

- **Noise Strength S:** This is the most impactful tuning parameter. It directly controls the amplitude of the random perturbation.
    - S=0 disables randomness, revealing the behavior of the 1D error diffusion core (prone to vertical lines).
    - Small S (e.g., 1 or 2 for 8-bit inputs scaled to the quantization step size) typically provides the best balance, effectively reducing banding and line artifacts without making the noise texture itself overly prominent.
    - Large S values can lead to excessive graininess and potentially more noticeable temporal noise in video. Guidance: Start with S around 1/4 to 1/2 of the quantization step size (e.g., for 2-bit output from 8-bit input, levels are ~30-40 apart, so S in the range 8-16 on the 0-255 scale, which translates to +/-1 to +/-2 on the *quantized level scale* if the random offset is scaled appropriately). The user's request for "+/-2 or beyond" refers to an offset relative to the input pixel value, which needs to be scaled relative to the quantization step size for optimal effect. An offset of +/-2 on an 8-bit scale is small; the Python implementation scales this to be more impactful relative to the quantization error.
- **LFSR Choice:** While a specific 16-bit maximal-length LFSR polynomial was used, other standard polynomials of similar length (e.g., 15-bit to 32-bit) would likely provide similar quality randomness.[20] The seed generation strategy (e.g.,

Hash(LineNum, FrameNum)) is important for ensuring good spatio-temporal noise characteristics.
- **Diffusion Coefficients:** The proposed coefficients (e.g., [1/2, 1/4, 1/4]) are simple and hardware-friendly. While fixed in this primary design, experimentation with other power-of-two denominator sets summing to 1 (e.g., [1/2, 1/2, 0] for 2-pixel lookahead, or different distributions for 4-pixel lookahead) could be undertaken if further fine-tuning is desired and hardware allows for easy reconfiguration. The current choice aims to spread the error somewhat rather than concentrating it all on the very next pixel.

The SLRED algorithm, given the severe hardware constraints, may not achieve the absolute pinnacle of dithering quality possible with unlimited resources (e.g., complex 2D blue-noise methods [9]). However, its significant advantage lies in being a practical, implementable solution that offers a substantial improvement over the user's current methods and common simple alternatives, all while fitting within the DPU's operational budget. This represents a common and valuable engineering outcome: a significant step forward that is feasible.

6.5. Suggestions for Further Optimization or Hardware Mapping

- **LFSR Implementation:** For RGB processing, a single LFSR can serve all three channels by either using its output directly for all, or by taking different bits or slightly delayed/offset outputs for R, G, and B to introduce some inter-channel noise decorrelation. This can save hardware compared to three independent LFSRs.
- **Error Accumulator Bit-Width:** The precise bit-width for error_accumulator_line and intermediate error terms should be finalized based on worst-case analysis of error swings to prevent clipping or overflow, while minimizing hardware. This depends on n_bits and noise_strength.
- **sRGB/Linear Conversion:** The importance of performing dithering arithmetic in a linear color space for perceptual accuracy is high.[25] If a full sRGB-to-linear conversion within the dither block is too costly, the DPU pipeline should ideally be designed to provide linear pixel data to the dither stage. If not, investigating highly optimized, approximate sRGB/linear conversions (e.g., piecewise linear, small LUTs) that fit the "small multiplications" constraint would be beneficial. Performing dithering directly in sRGB space is a compromise that may affect color accuracy and artifact appearance.
- **Lookahead Window:** The current design supports 3 or 4 pixel lookahead. A 3-pixel lookahead is slightly simpler in hardware (fewer error terms to manage). The perceptual benefit of 4-pixel vs. 3-pixel should be weighed against this minor

complexity increase.

- **Adaptive Noise Strength:** While SLRED uses a global noise_strength, a more advanced DPU might consider adapting S based on image content (e.g., detected smoothness or texture), if such analysis is feasible upstream or via simple metrics. This could optimize the dither effect for different image regions.

6.6. Concluding Remarks

The proposed Single-Line Randomized Error Diffusion (SLRED) algorithm successfully addresses the challenge of providing high-quality dithering under stringent DPU hardware constraints. It effectively reduces banding and pattern artifacts compared to common methods like truncation and ordered dither, and offers a more controlled and often perceptually superior noise characteristic than basic random dithering. Its design strictly adheres to single-line processing, limited pixel lookahead, integer-only arithmetic with power-of-two error diffusion coefficients, and avoidance of serpentine scanning.

The incorporation of per-line LFSR-based randomness is key to breaking up deterministic artifacts, while the controllable noise strength allows for fine-tuning the balance between artifact suppression and visible noise. SLRED represents a significant improvement over the user's existing implementations and provides a robust, hardware-friendly solution for enhancing visual quality in resource-constrained display systems.

It is recommended that the SLRED algorithm, with an empirically tuned noise strength parameter (e.g., S in the range of 1 to 2, scaled appropriately for the quantization step size), be adopted for the target DPU. Further hardware-specific optimization of LFSR sharing and error term precision should be undertaken during the implementation phase. Consideration of the processing color space (linear vs. sRGB) is also crucial for optimal color fidelity.

7. Appendix

7.1. Complete Python Source Code

Python

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from skimage.metrics import peak_signal_noise_ratio
from PIL import Image

# --- Utility Functions ---
def srgb_to_linear(srgb_val):
    """Converts an sRGB value (0-255) to linear (0-255)."""
    val = srgb_val / 255.0
    if val <= 0.04045:
        linear_val = val / 12.92
    else:
        linear_val = ((val + 0.055) / 1.055) ** 2.4
    return linear_val * 255.0

def linear_to_srgb(linear_val):
    """Converts a linear value (0-255) to sRGB (0-255)."""
    val = linear_val / 255.0
    if val <= 0.0031308:
        srgb_val = val * 12.92
    else:
        srgb_val = 1.055 * (val ** (1.0 / 2.4)) - 0.055
    return srgb_val * 255.0

def quantize_value(value, n_bits):
    """Quantizes a value (0-255) to n_bits."""
    if n_bits == 8: # No quantization needed for 8-bit output
        return np.clip(value, 0, 255)
    levels = 2**n_bits
    quant_step = 255.0 / (levels - 1)
    quantized = np.round(value / quant_step) * quant_step
    return np.clip(quantized, 0, 255)

class LFSR:
    """A simple Galois LFSR implementation."""
    def __init__(self, seed, polynomial, n_bits=16):
        self.n_bits = n_bits
        self.mask = (1 << n_bits) - 1
        self.state = seed & self.mask
        self.poly = polynomial
        if self.state == 0: # LFSR must not be seeded with 0 for maximal length
            self.state = 1
```

```
    def next_lfsr_val(self):
        """Generates the next LFSR value."""
        # For Galois LFSR, if LSB is 1, XOR state with polynomial
        if self.state & 1:
            self.state = (self.state >> 1) ^ self.poly
        else:
            self.state = self.state >> 1
        return self.state & self.mask # Return lower bits if poly is for full state

    def get_scaled_bipolar_offset(self, noise_strength_parameter, output_range_bits=8):
        """
        Generates a bipolar offset based on LFSR output and noise_strength.
        noise_strength_parameter: e.g., 0, 1, 2,... representing approx +/- S levels
                                  This is relative to the quantization step of the *output*.
        output_range_bits: The bit depth of the value the noise is added to (e.g., 8 for 0-255).
        """
        if noise_strength_parameter == 0:
            return 0

        # Get raw LFSR value (e.g., 0 to 2^16-1 for 16-bit LFSR)
        raw_lfsr = self.next_lfsr_val()

        # Scale LFSR to a smaller range, e.g., 0 to (2*S_eff - 1)
        # This scaling needs to be hardware friendly.
        # Let's aim for a bipolar offset related to noise_strength_parameter.
        # Example: if LFSR is 16-bit, noise_strength_parameter is 2.
        # We want an offset like [-2, -1, 0, 1, 2] relative to some scale.
        # Let's assume noise_strength_parameter directly maps to max deviation.

        # Simple scaling: use a few bits of LFSR
        # To get a range of approx. 2 * noise_strength_parameter + 1 values
        num_noise_levels = 2 * noise_strength_parameter + 1

        # If LFSR output is, say, 16 bits, and we want ~5 levels for S=2:
        # We can take LFSR_out % num_noise_levels, then subtract noise_strength_parameter
        # This is not perfectly uniform but simple.
        if num_noise_levels <=1: # Should not happen if S > 0
            return 0
```

```
        # For hardware, a shift and subtract is better.
        # Example: if LFSR is 16-bit (0-65535) and noise_strength_parameter = 2
        # We want values like -2, -1, 0, 1, 2.
        # A simple way: (raw_lfsr >> (16 - 3)) - noise_strength_parameter (for 3 bits -> 0-7 range)
        # This gives 0..7, then subtract S.
        # If S=2, (raw_lfsr >> 13) - 2 => (0..7) - 2 => -2.. 5. Not quite symmetric.

        # Alternative: map LFSR to (3 values)
        # if noise_strength_parameter = 2, range is [-2, -1, 0, 1, 2] (5 values)
        # if noise_strength_parameter = 3, range is [-3,..., 3] (7 values)
        # if noise_strength_parameter = 4, range is [-4,..., 4] (9 values)

        # A simple integer mapping:
        # Take LFSR output modulo (2*S+1), then subtract S.
        # Example: S=2, range is 5 values. LFSR_val % 5 gives 0,1,2,3,4. Subtract 2 -> -2,-1,0,1,2
        offset = (raw_lfsr % (2 * noise_strength_parameter + 1)) -
noise_strength_parameter

        # The actual impact of this offset depends on the scale of pixel values.
        # If pixel values are 0-255, an offset of +/-2 is small.
        # The "noise_strength" in the prompt implies it should be significant.
        # Let's scale it by a factor related to quantization step of the *input*.
        # For 8-bit input, a reasonable dither amplitude is around half a quantization step of the output.
        # If output is 2-bit (4 levels from 0-255), levels are 0, 85, 170, 255. Step is ~85. Half step ~42.
        # If output is 4-bit (16 levels), levels are 0, 17, 34,... Step is ~17. Half step ~8.
        # The user's "noise_strength (-2 to 2 or beyond)" might refer to this scaled impact.

        # For this implementation, let's assume noise_strength_parameter is the direct integer offset.
        # The Python code will use this directly. In hardware, this would be a small integer.
        return int(offset)


# --- Dithering Algorithms ---

def slred_dither_channel(channel_data, n_bits, noise_strength,
                         diffusion_coeffs, lfsr_poly, frame_id=0,
                         process_in_linear=True):
    """Applies SLRED to a single color channel."""
    height, width = channel_data.shape
    output_channel = np.copy(channel_data).astype(np.float32) # Work with float for
```

```
precision

    # For SLRED, error accumulator is 1D for the current line
    # Needs to accommodate lookahead for diffusion_coeffs
    lookahead = len(diffusion_coeffs)
    error_accumulator_line = np.zeros(width + lookahead, dtype=np.float32)

    # Standard 16-bit LFSR polynomial (X^16+X^5+X^3+X^2+1)
    # Alternative: (X^16+X^14+X^13+X^11+1) -> 0b1011010000000001 or 0xAB01
    # Using X^16+X^12+X^3+X+1 (common in some comms) -> 0b1000100000001011 or 0x880B
    if lfsr_poly is None:
        lfsr_poly = 0b100000000001011 # A common 16-bit poly: x^16+x^12+x^3+x+1
        # For Galois, this is 0x800D if MSB is x^0. If MSB is x^15, it's different.
        # Let's use a common one for Fibonacci: taps at 16, 14, 13, 11 (from Xilinx app note)
        # For Galois, if poly is P(x)=x^n +... + 1, feedback is based on poly.
        # Let's use one from a table: x^16 + x^15 + x^13 + x^4 + 1 (reversed for Galois: 1 + x + x^3 + x^12 +
x^16)
        # => 0b1010000000001001 (0xA009 if x^0 is MSB)
        # For this LFSR class (Galois, LSB feedback): use reversed poly
        lfsr_poly = 0x8005 # Corresponds to x^16 + x^14 + x + 1 (reversed for Galois)
                           # Or x^16+x^15+x^2+1. Let's use a known maximal:
        lfsr_poly = 0xB400 # Taps at 16,14,13,11 (for Fibonacci, reversed for Galois)
                           # This is for x^16+x^11+x^13+x^14+x^16 which is not right.
                           # Standard primitive poly for 16-bit: x^16+x^5+x^3+x^2+1
                           # In Galois form where LSB is shifted out, and poly XORed if LSB=1:
                           # poly should be like 0x... where bits correspond to other taps.
                           # x^16+x^5+x^3+x^2+1 => taps at 0,2,3,5. (0x2D for Fibonacci, shifted)
                           # For Galois, if state is s_15... s_0: new s_15 = s_0. feedback = s_0.
                           # s_i = s_{i+1} XOR (poly_i * feedback)
                           # Let's use a common one: 0xACE1 for 16-bit
        lfsr_poly = 0xACE1 # A common maximal length polynomial for 16-bit LFSR

    lfsr = LFSR(seed=1, polynomial=lfsr_poly) # Seed will be reset per line

    for y in range(height):
        # Reset error accumulator for the line (conceptually)
        # In hardware, this would be a small shift register of errors for lookahead pixels
        current_line_errors = np.zeros(width + lookahead, dtype=np.float32)
```

```
        # Re-seed LFSR for each line
        line_seed = ((frame_id + 1) * (y + 1) * 123456789) & 0xFFFF
        if line_seed == 0: line_seed = 1 # Avoid 0 seed
        lfsr.state = line_seed

        for x in range(width):
            original_val = output_channel[y, x]

            if process_in_linear:
                pixel_val_linear = srgb_to_linear(original_val)
            else:
                pixel_val_linear = original_val

            # 1. Modify with accumulated error
            modified_val = pixel_val_linear + current_line_errors[x]

            # 2. Add bipolar random offset
            # The noise_strength here is a small integer, e.g. 1, 2, 3, 4.
            # It should be scaled to be meaningful relative to pixel values (0-255)
            # Let's assume the bipolar offset is already scaled appropriately by get_scaled_bipolar_offset
            # For example, if noise_strength = 2, it gives an offset like -2, -1, 0, 1, 2.
            # To make it impactful for 0-255 range, we might scale it.
            # Let's say noise_strength of 1 means +/- (quant_step_output / 4)
            # For 2-bit output, quant_step_output is ~85. So noise_strength=1 -> +/-21.
            # For 4-bit output, quant_step_output is ~17. So noise_strength=1 -> +/-4.
            # The current LFSR.get_scaled_bipolar_offset returns small integers like -S..S
            # Let's use these small integers directly as the user requested "-2 to 2 or beyond"
            # This implies the *value* of the offset.

            bipolar_offset = 0
            if noise_strength > 0:
                # Get small integer offset, e.g., -noise_strength to +noise_strength
                small_int_offset = (lfsr.next_lfsr_val() % (2 * noise_strength + 1)) -
noise_strength
                bipolar_offset = float(small_int_offset)


            dithered_val = modified_val + bipolar_offset
            dithered_val_clipped = np.clip(dithered_val, 0, 255) # Clip before quantization
```

```
            # 3. Quantize
            quantized_val_linear = quantize_value(dithered_val_clipped, n_bits)

            # 4. Calculate quantization error
            # Error is based on the value *before* clipping if we want to be strict,
            # but hardware might clip then quantize.
            # [12]: quant_error := oldpixel - newpixel. Here, oldpixel is dithered_val_clipped.
            error = dithered_val_clipped - quantized_val_linear

            # 5. Distribute error to next pixels on the same line
            for i in range(lookahead):
                if x + 1 + i < width + lookahead : # Check bounds for the error accumulator
                    # Coefficients are like 1/2, 1/4, 1/8 - use float mult for simulation
                    # In hardware, this would be (error * numerator[i]) >> shift_amount[i]
                    if diffusion_coeffs[i] == 0.5: # 1/2
                        current_line_errors[x + 1 + i] += error / 2.0
                    elif diffusion_coeffs[i] == 0.25: # 1/4
                        current_line_errors[x + 1 + i] += error / 4.0
                    elif diffusion_coeffs[i] == 0.125: # 1/8
                        current_line_errors[x + 1 + i] += error / 8.0
                    elif diffusion_coeffs[i] == 0.0625: # 1/16
                        current_line_errors[x + 1 + i] += error / 16.0
                    else: # General case if not power of 2 (for simulation flexibility)
                         current_line_errors[x + 1 + i] += error * diffusion_coeffs[i]

            if process_in_linear:
                output_channel[y, x] = linear_to_srgb(quantized_val_linear)
            else:
                output_channel[y, x] = quantized_val_linear

    return np.clip(output_channel, 0, 255).astype(np.uint8)

def slred_dither(image, n_bits, noise_strength,
                 diffusion_coeffs_str="1/2,1/4,1/4", lfsr_poly_hex=None,
                 process_in_linear=True):
    coeffs_parts = [c.strip() for c in diffusion_coeffs_str.split(',')]
    diffusion_coeffs =
    for part in coeffs_parts:
```

```
        if '/' in part:
            num, den = map(int, part.split('/'))
            diffusion_coeffs.append(float(num)/den)
        else:
            diffusion_coeffs.append(float(part))

    lfsr_p = None
    if lfsr_poly_hex:
        lfsr_p = int(lfsr_poly_hex, 16)

    if image.ndim == 3: # RGB
        output_image = np.zeros_like(image)
        for i in range(3):
            # print(f"Processing SLRED channel {i}...")
            output_image[:,:,i] = slred_dither_channel(image[:,:,i], n_bits, noise_strength,
                                                       diffusion_coeffs, lfsr_p,
                                                       process_in_linear=process_in_linear)
    else: # Grayscale
        # print("Processing SLRED grayscale...")
        output_image = slred_dither_channel(image, n_bits, noise_strength,
                                            diffusion_coeffs, lfsr_p,
                                            process_in_linear=process_in_linear)
    return output_image

def truncate_dither_channel(channel_data, n_bits):
    output_channel = np.copy(channel_data)
    # This is equivalent to quantize_value without any error diffusion or noise
    for y in range(channel_data.shape):
        for x in range(channel_data.shape):
            output_channel[y,x] = quantize_value(channel_data[y,x], n_bits)
    return output_channel.astype(np.uint8)

def truncate_dither(image, n_bits):
    if image.ndim == 3:
        output_image = np.zeros_like(image)
        for i in range(3):
            output_image[:,:,i] = truncate_dither_channel(image[:,:,i], n_bits)
    else:
        output_image = truncate_dither_channel(image, n_bits)
```

```
    return output_image

def get_bayer_matrix(size):
    if size == 2:
        return np.array([, ])
    elif size == 4:
        return np.array(,
            ,
            ,
            ])
    elif size == 8:
        m = get_bayer_matrix(4)
        m_size = 4
        bayer8 = np.zeros((8,8), dtype=int)
        bayer8[0:m_size, 0:m_size]         = 4 * m
        bayer8[0:m_size, m_size:2*m_size] = 4 * m + 2
        bayer8[m_size:2*m_size, 0:m_size] = 4 * m + 3
        bayer8[m_size:2*m_size, m_size:2*m_size] = 4 * m + 1
        return bayer8
    else:
        raise ValueError("Unsupported Bayer matrix size")

def ordered_dither_channel(channel_data, n_bits, bayer_matrix_size, process_in_linear=True):
    height, width = channel_data.shape
    output_channel = np.zeros_like(channel_data, dtype=np.uint8)
    bayer_matrix = get_bayer_matrix(bayer_matrix_size)
    bayer_levels = bayer_matrix.size # e.g., 16 for 4x4

    # The Bayer matrix values act as thresholds that are scaled across the quantization interval
    # For N-bit output, there are 2^N levels.
    # The threshold from Bayer matrix (0 to bayer_levels-1) needs to be mapped.
    # Pixel_val / 255 * (2^N-1) gives value in 0.. (2^N-1) range.
    # Threshold_from_matrix / bayer_levels gives value in 0.. 1 range.
    # If (Pixel_val_norm + Threshold_from_matrix_norm) / 2 > 0.5...
    # A common way: scale pixel to 0..M*L-1 range, where M is bayer_levels, L is output levels
    # threshold = (bayer_matrix + 1) / (S*S + 1) - 0.5; [5]

    output_levels = 2**n_bits
    scale_factor = 255.0 / (output_levels -1) if output_levels > 1 else 255.0
```

```
    for y in range(height):
        for x in range(width):
            original_val = channel_data[y, x]
            if process_in_linear:
                pixel_val_linear = srgb_to_linear(original_val)
            else:
                pixel_val_linear = original_val

            # Normalize pixel value to 0-1 range
            # Add scaled Bayer threshold, then quantize
            # The Bayer matrix values are 0 to (matrix_size^2 - 1)
            # These are used to perturb the quantization threshold
            # Map pixel to 0..(L-1) where L = 2^n_bits
            # Add (Bayer_val / Bayer_max_val - 0.5) * step_size

            # Simpler: map pixel to 0.. (N-1) where N is number of output levels
            # Add bayer_value / bayer_max_val to it, then round.
            # This is thresholding against multiple levels.
            # (pixel_val / 256 * (output_levels-1))
            # Effective_pixel = pixel_val + (bayer_matrix / (S*S) - 0.5) * (256/output_levels)

            # From [5]: If the value of the pixel (scaled into the 0-M range, M=matrix_levels)
            # is less than the number in the corresponding cell of the matrix, plot black, else white (for
bi-level)
            # For multi-level:
            # scaled_pixel = pixel_val_linear / (256.0 / (bayer_levels * output_levels))
            # bayer_thresh_idx = bayer_matrix[y % bayer_matrix_size, x % bayer_matrix_size]
            # output_idx = np.floor( (scaled_pixel + bayer_thresh_idx) / bayer_levels )

            # Alternative: Add scaled threshold before quantization to 0-255 range
            # Bayer matrix values are 0 to bayer_levels-1. Normalize to -0.5 to 0.5.
            # This normalized threshold is then scaled by the quantization step size.
            threshold_normalized = (bayer_matrix[y % bayer_matrix_size, x %
bayer_matrix_size] / float(bayer_levels)) - 0.5
            # This threshold perturbs the input value relative to the quantization intervals
            # The magnitude of perturbation should be around one quantization step.
            perturbation = threshold_normalized * (255.0 / (output_levels)) # Max perturbation
```

```
is +/- half a step

            perturbed_val = np.clip(pixel_val_linear + perturbation, 0, 255)
            quantized_val_linear = quantize_value(perturbed_val, n_bits)

            if process_in_linear:
                output_channel[y,x] = linear_to_srgb(quantized_val_linear)
            else:
                output_channel[y,x] = quantized_val_linear

    return output_channel


def ordered_dither(image, n_bits, bayer_matrix_size=4, process_in_linear=True):
    if image.ndim == 3:
        output_image = np.zeros_like(image)
        for i in range(3):
            # print(f"Processing Ordered Dither channel {i}...")
            output_image[:,:,i] = ordered_dither_channel(image[:,:,i], n_bits,
bayer_matrix_size, process_in_linear)
    else:
        # print("Processing Ordered Dither grayscale...")
        output_image = ordered_dither_channel(image, n_bits, bayer_matrix_size,
process_in_linear)
    return output_image

def random_dither_user_channel(channel_data, n_bits, noise_strength, frame_id=0,
process_in_linear=True):
    """Simulates user's random dither."""
    height, width = channel_data.shape
    output_channel = np.zeros_like(channel_data, dtype=np.uint8)

    # For per-line randomness, seed np.random here or use a line-based PRNG
    # For simplicity, using pixel-wise randomness here, can be adapted
    # np.random.seed( (frame_id+1) * 123 ) # Seed once per frame for some consistency if desired

    for y in range(height):
        # For per-line: np.random.seed( ((frame_id+1)*(y+1)) & 0xFFFFFFFF )
        for x in range(width):
            original_val = channel_data[y,x]
```

```
            if process_in_linear:
                pixel_val_linear = srgb_to_linear(original_val)
            else:
                pixel_val_linear = original_val

            # Add bipolar random noise
            # noise_strength: e.g. 2 means random offset in [-2, -1, 0, 1, 2]
            rand_offset = 0
            if noise_strength > 0:
                rand_offset = np.random.randint(-noise_strength, noise_strength + 1)

            dithered_val = np.clip(pixel_val_linear + rand_offset, 0, 255)
            quantized_val_linear = quantize_value(dithered_val, n_bits)

            if process_in_linear:
                output_channel[y,x] = linear_to_srgb(quantized_val_linear)
            else:
                output_channel[y,x] = quantized_val_linear

    return output_channel

def random_dither_user(image, n_bits, noise_strength, process_in_linear=True):
    if image.ndim == 3:
        output_image = np.zeros_like(image)
        for i in range(3):
            # print(f"Processing Random Dither (User) channel {i}...")
            output_image[:,:,i] = random_dither_user_channel(image[:,:,i], n_bits,
noise_strength, process_in_linear=process_in_linear)
    else:
        # print("Processing Random Dither (User) grayscale...")
        output_image = random_dither_user_channel(image, n_bits, noise_strength,
process_in_linear=process_in_linear)
    return output_image

# --- Main Comparison Script ---
def generate_gradient_image(size=256, is_rgb=False):
    gradient = np.linspace(0, 255, size, dtype=np.uint8)
    image = np.tile(gradient, (size, 1))
    if is_rgb:
```

```
        img_rgb = np.zeros((size, size, 3), dtype=np.uint8)
        # Create different gradients for R, G, B for more interesting visuals
        img_rgb[:,:,0] = image # R: horizontal
        img_rgb[:,:,1] = image.T # G: vertical
        img_rgb[:,:,2] = np.flip(image, axis=1) # B: reverse horizontal
        return img_rgb
    return image

def run_comparison():
    # Test images
    grad_gray = generate_gradient_image(256, is_rgb=False)
    grad_rgb = generate_gradient_image(256, is_rgb=True)

    try:
        lena_rgb = np.array(Image.open("lena_color.png"))
    except FileNotFoundError:
        print("Warning: lena_color.png not found. Using RGB gradient as placeholder.")
        lena_rgb = grad_rgb # Placeholder if Lena is not available
    try:
        baboon_rgb = np.array(Image.open("baboon_color.png"))
    except FileNotFoundError:
        print("Warning: baboon_color.png not found. Using RGB gradient as placeholder.")
        baboon_rgb = grad_rgb


    test_images = {
        "Gradient Gray": grad_gray,
        "Gradient RGB": grad_rgb,
        "Lena RGB": lena_rgb,
        "Baboon RGB": baboon_rgb
    }

    bit_depths =
    # SLRED params
    slred_noise_strengths_to_test =  # S=0 and an "optimal" S
    slred_coeffs_3px = "1/2,1/4,1/4" # Sum = 1.0
    # slred_coeffs_4px = "1/2,1/4,1/8,1/8" # Sum = 1.0

    # User's Random Dither noise strength
```

```
    user_random_noise_strength = 2 # To match SLRED S=2 for comparison

    results = {} # To store PSNR

    process_in_linear_space = True # Set to False to disable sRGB/Linear conversions

    for n_bits in bit_depths:
        print(f"\n--- Processing for {n_bits}-bit reduction ---")
        results[n_bits] = {}
        for name, img_orig_srgb in test_images.items():
            print(f"  Image: {name}")
            results[n_bits][name] = {}

            # Original image (ensure it's uint8 for PSNR)
            img_orig_uint8 = np.clip(img_orig_srgb, 0, 255).astype(np.uint8)

            # 1. Truncation
            img_trunc = truncate_dither(img_orig_srgb, n_bits) # Assumes sRGB input,
quantizes, no linear
            psnr_trunc = peak_signal_noise_ratio(img_orig_uint8, img_trunc,
data_range=255)
            results[n_bits][name] = psnr_trunc
            Image.fromarray(img_trunc).save(f"{name.replace(' ','_')}_{n_bits}bit_Truncation.png")

            # 2. Ordered Dither (4x4 Bayer)
            img_ordered = ordered_dither(img_orig_srgb, n_bits, bayer_matrix_size=4,
process_in_linear=process_in_linear_space)
            psnr_ordered = peak_signal_noise_ratio(img_orig_uint8, img_ordered,
data_range=255)
            results[n_bits][name]["Ordered (4x4)"] = psnr_ordered
            Image.fromarray(img_ordered).save(f"{name.replace('
','_')}_{n_bits}bit_Ordered4x4.png")

            # 3. Random Dither (User's)
            img_random_user = random_dither_user(img_orig_srgb, n_bits,
noise_strength=user_random_noise_strength,
process_in_linear=process_in_linear_space)
            psnr_random_user = peak_signal_noise_ratio(img_orig_uint8, img_random_user,
data_range=255)
```

```
            results[n_bits][name] = psnr_random_user
            Image.fromarray(img_random_user).save(f"{name.replace('
','_')}_{n_bits}bit_RandomUser_S{user_random_noise_strength}.png")

            # 4. SLRED
            for s_val in slred_noise_strengths_to_test:
                img_slred = slred_dither(img_orig_srgb, n_bits, noise_strength=s_val,
diffusion_coeffs_str=slred_coeffs_3px, process_in_linear=process_in_linear_space)
                psnr_slred = peak_signal_noise_ratio(img_orig_uint8, img_slred,
data_range=255)
                results[n_bits][name] = psnr_slred
                Image.fromarray(img_slred).save(f"{name.replace('
','_')}_{n_bits}bit_SLRED_S{s_val}.png")

            print(f"    Truncation PSNR: {psnr_trunc:.2f} dB")
            print(f"    Ordered (4x4) PSNR: {psnr_ordered:.2f} dB")
            print(f"    Random (User, S={user_random_noise_strength}) PSNR: {psnr_random_user:.2f} dB")
            for s_val in slred_noise_strengths_to_test:
                print(f"    SLRED (S={s_val}) PSNR: {results[n_bits][name]:.2f} dB")


    # Print PSNR Table (similar to Table 3 format)
    print("\n\n--- PSNR Results Summary (dB) ---")
    header =
    algos = +
    header.extend(algos)
    print("\t".join(header))

    for n_bits_res in bit_depths:
        for name_res in test_images.keys():
            row = [name_res, str(n_bits_res)]
            for algo in algos:
                row.append(f"{results[n_bits_res][name_res].get(algo, 'N/A'):.2f}")
            print("\t".join(row))

    # --- Visualization ---
    # Select one image and one bit depth for detailed visual comparison
    vis_img_name = "Gradient Gray"
    vis_n_bits = 2
```

```
    img_srgb_for_vis = test_images[vis_img_name]

    dithered_images_for_vis = {
        "Original": np.clip(img_srgb_for_vis,0,255).astype(np.uint8),
        "Truncation": truncate_dither(img_srgb_for_vis, vis_n_bits),
        "Ordered (4x4)": ordered_dither(img_srgb_for_vis, vis_n_bits, bayer_matrix_size=4,
process_in_linear=process_in_linear_space),
        f"Random (User, S={user_random_noise_strength})":
random_dither_user(img_srgb_for_vis, vis_n_bits,
noise_strength=user_random_noise_strength,
process_in_linear=process_in_linear_space),
        "SLRED (S=0)": slred_dither(img_srgb_for_vis, vis_n_bits, noise_strength=0,
diffusion_coeffs_str=slred_coeffs_3px, process_in_linear=process_in_linear_space),
        f"SLRED (S={slred_noise_strengths_to_test})": slred_dither(img_srgb_for_vis, vis_n_bits,
noise_strength=slred_noise_strengths_to_test, diffusion_coeffs_str=slred_coeffs_3px,
process_in_linear=process_in_linear_space)
    }

    num_methods = len(dithered_images_for_vis)
    fig, axes = plt.subplots(2, (num_methods + 1) // 2, figsize=(18, 8))
    axes = axes.ravel()

    plot_idx = 0
    for title, d_img in dithered_images_for_vis.items():
        if d_img.ndim == 2: # Grayscale
            axes[plot_idx].imshow(d_img, cmap='gray', vmin=0, vmax=255)
        else: # RGB
            axes[plot_idx].imshow(d_img)
        axes[plot_idx].set_title(f"{title}\n{vis_n_bits}-bit")
        axes[plot_idx].axis('off')
        plot_idx += 1

    while plot_idx < len(axes): # Hide unused subplots
        axes[plot_idx].axis('off')
        plot_idx +=1

    fig.suptitle(f"Dithering Comparison: {vis_img_name} ({vis_n_bits}-bit output)", fontsize=16)
    plt.tight_layout(rect=[0, 0, 1, 0.96])
```

```
    plt.savefig(f"Comparison_{vis_img_name.replace(' ','_')}_{vis_n_bits}bit.png")
    plt.show()

    # Zoomed comparison for Lena
    vis_img_name_zoom = "Lena RGB"
    vis_n_bits_zoom = 4
    img_srgb_for_zoom = test_images[vis_img_name_zoom]

    dithered_images_for_zoom = {
        "Original": np.clip(img_srgb_for_zoom,0,255).astype(np.uint8),
        "Truncation": truncate_dither(img_srgb_for_zoom, vis_n_bits_zoom),
        "Ordered (4x4)": ordered_dither(img_srgb_for_zoom, vis_n_bits_zoom,
bayer_matrix_size=4, process_in_linear=process_in_linear_space),
        f"Random (User, S={user_random_noise_strength})":
random_dither_user(img_srgb_for_zoom, vis_n_bits_zoom,
noise_strength=user_random_noise_strength,
process_in_linear=process_in_linear_space),
        "SLRED (S=0)": slred_dither(img_srgb_for_zoom, vis_n_bits_zoom,
noise_strength=0, diffusion_coeffs_str=slred_coeffs_3px,
process_in_linear=process_in_linear_space),
        f"SLRED (S={slred_noise_strengths_to_test})": slred_dither(img_srgb_for_zoom,
vis_n_bits_zoom, noise_strength=slred_noise_strengths_to_test,
diffusion_coeffs_str=slred_coeffs_3px, process_in_linear=process_in_linear_space)
    }

    zoom_coords = (slice(150, 250), slice(200, 300)) # Example zoom region for Lena
(face/shoulder)

    fig_zoom, axes_zoom = plt.subplots(2, (num_methods + 1) // 2, figsize=(18, 8))
    axes_zoom = axes_zoom.ravel()

    plot_idx_zoom = 0
    for title, d_img in dithered_images_for_zoom.items():
        zoomed_d_img = d_img[zoom_coords]
        if zoomed_d_img.ndim == 2:
            axes_zoom[plot_idx_zoom].imshow(zoomed_d_img, cmap='gray', vmin=0,
vmax=255)
        else:
            axes_zoom[plot_idx_zoom].imshow(zoomed_d_img)
```

```
        axes_zoom[plot_idx_zoom].set_title(f"{title}\n{vis_n_bits_zoom}-bit (Zoomed)")
        axes_zoom[plot_idx_zoom].axis('off')
        plot_idx_zoom += 1

    while plot_idx_zoom < len(axes_zoom): # Hide unused subplots
        axes_zoom[plot_idx_zoom].axis('off')
        plot_idx_zoom +=1

    fig_zoom.suptitle(f"Zoomed Dithering Comparison: {vis_img_name_zoom} ({vis_n_bits_zoom}-bit output)", fontsize=16)
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.savefig(f"Comparison_Zoomed_{vis_img_name_zoom.replace(' ','_')}_{vis_n_bits_zoom}bit.png")
    plt.show()


if __name__ == '__main__':
    # Note: To run this, you might need to download 'lena_color.png' and 'baboon_color.png'
    # or replace them with other images.
    # Example: Image.fromarray(generate_gradient_image(512,True)).save("lena_color.png")
    # Example: Image.fromarray(generate_gradient_image(512,True)).save("baboon_color.png")
    run_comparison()
```

7.2. Additional High-Resolution Images or Detailed Plots

The Python script provided in Section 7.1 will generate comparative images (full and zoomed) for the specified test cases. These images, saved as PNG files, serve as the primary visual evidence for the qualitative assessments discussed in Section 5. Further detailed plots, such as LFSR output distributions or specific artifact close-ups beyond those automatically generated, can be created by modifying the visualization part of the script if deeper analysis of a particular aspect is required. For instance, plotting the error signal err(x) over a line for different algorithms could reveal how effectively each method manages and diffuses quantization error. Similarly, plotting line-wise or column-wise pixel value differences could help quantify pattern repetition or smoothness.

Works cited

1. Dither - Wikipedia, accessed June 2, 2025, https://en.wikipedia.org/wiki/Dither

2. Processing: How to Get Rid of Banding? Part 2 - René Algesheimer Photography, accessed June 2, 2025, https://www.rene-algesheimer.com/get-rid-of-banding-2/
3. Error diffusion - Wikipedia, accessed June 2, 2025, https://en.wikipedia.org/wiki/Error_diffusion
4. Floyd-Steinberg Dithering - Cloudinary, accessed June 2, 2025, https://cloudinary.com/glossary/floyd-steinberg-dithering
5. Ordered Dithering, accessed June 2, 2025, https://www.visgraf.impa.br/Courses/ip00/proj/Dithering1/ordered_dithering.html
6. Penfore/dither_it: DitherIt is a Dart library that implements various dithering algorithms. - GitHub, accessed June 2, 2025, https://github.com/Penfore/dither_it
7. The 4 variant of TD — from least provocative to most aggressive pixel flicker : r/Temporal_Noise - Reddit, accessed June 2, 2025, https://www.reddit.com/r/Temporal_Noise/comments/1jzlya0/the_4_variant_of_td_from_least_provocative_to/
8. Temporal dithering - Doom9's Forum, accessed June 2, 2025, https://forum.doom9.org/showthread.php?t=175002
9. Shape dithering for 3D printing - SciSpace, accessed June 2, 2025, https://scispace.com/pdf/shape-dithering-for-3d-printing-1qrdgrye.pdf
10. What Is Dithering? How To Achieve Superior Sound Quality, accessed June 2, 2025, https://unison.audio/dithering/
11. bitsavers.org, accessed June 2, 2025, https://bitsavers.org/pdf/dec/tech_reports/CRL-98-2.pdf
12. Floyd–Steinberg dithering - Wikipedia, accessed June 2, 2025, https://en.wikipedia.org/wiki/Floyd%E2%80%93Steinberg_dithering
13. Libcaca study - 3. Error diffusion, accessed June 2, 2025, http://caca.zoy.org/study/part3.html
14. Fast Error Diffusion and Digital Halftoning Algorithms Using Look-Up Tables - IBM Research Report, accessed June 2, 2025, https://dominoweb.draco.res.ibm.com/reports/rc23708.pdf
15. Dithering an image using the Burkes algorithm in C# - Articles and ..., accessed June 2, 2025, https://www.cyotek.com/blog/dithering-an-image-using-the-burkes-algorithm-in-csharp
16. Experiments with structure-aware, error diffusion dithering algorithms, and other ways to improve beyond Floyd-Steinberg. - GitHub, accessed June 2, 2025, https://github.com/dalpil/structure-aware-dithering
17. A simple and efficient error-diffusion algorithm - SciSpace, accessed June 2, 2025, https://scispace.com/pdf/a-simple-and-efficient-error-diffusion-algorithm-514xcktsge.pdf
18. perso.liris.cnrs.fr, accessed June 2, 2025, https://perso.liris.cnrs.fr/ostrom/publications/pdf/SIGGRAPH01_varcoeffED.pdf
19. Pseudo Random Number Generator using Internet-of-Things Techniques on Portable Field-Programmable-Gate-Array Platform - arXiv, accessed June 2, 2025, https://www.arxiv.org/pdf/2505.03741

20. 14.4.3.2 LFSR – Linear Feedback Shift Register - Microchip Online docs, accessed June 2, 2025, https://onlinedocs.microchip.com/oxy/GUID-04B5982F-17EC-4A6E-B7FE-72DF0A5463B9-en-US-3/GUID-4B2ED6CC-1BD9-4133-B3F9-9EA933A1F268.html
21. Linear-feedback shift register - Wikipedia, accessed June 2, 2025, https://en.wikipedia.org/wiki/Linear-feedback_shift_register
22. Pseudo Random Number Generation Using Linear Feedback Shift Registers, accessed June 2, 2025, https://www.analog.com/en/resources/design-notes/random-number-generation-using-lfsr.html
23. Linear Feedback Shift Registers (LFSR) - GeeksforGeeks, accessed June 2, 2025, https://www.geeksforgeeks.org/linear-feedback-shift-registers-lfsr/
24. Diagnosing and Disabling Dithering in the Graphics Pipeline - VPixx Support, accessed June 2, 2025, https://docs.vpixx.com/vocal/diagnosing-and-disabling-dithering-in-the-graphics
25. Dithering in Colour - Hacker News, accessed June 2, 2025, https://news.ycombinator.com/item?id=43315029
26. Morphological characterization of dithering masks - CiteSeerX, accessed June 2, 2025, https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=57865709cf7977cd7cb9c8b9c1bec719246d80c7
27. Efficient and Effective Detection of Repeated Pattern from Fronto-Parallel Images with Unknown Visual Contents - MDPI, accessed June 2, 2025, https://www.mdpi.com/2624-6120/6/1/4
28. Image Texture Analysis Foundations Models And Algorithms Chihcheng Hung Enmin Song Yihua Lan instant download - Scribd, accessed June 2, 2025, https://www.scribd.com/document/860442333/Image-Texture-Analysis-Foundations-Models-And-Algorithms-Chihcheng-Hung-Enmin-Song-Yihua-Lan-instant-download
29. PyTextureAnalysis is a Python package for analyzing the texture of images. It includes functions for calculating local orientation, degree of coherence, and structure tensor of an image. This package is built using NumPy, SciPy and OpenCV. - GitHub, accessed June 2, 2025, https://github.com/ajinkya-kulkarni/PyTextureAnalysis
30. Enhanced Error Diffusion with Thresholds Modulated by Spatio-Chromatically Correlated Noise - シャープ, accessed June 2, 2025, https://corporate.jp.sharp/rd/journal-76/pdf/76-05.pdf