User Guide

Automated Generation of Customized Labels

Abstract

The Automated Label Generator is a Python-based program designed for creating realistic, customizable label images for data augmentation, machine learning training, and synthetic dataset generation. This program combines advanced image processing techniques with a user-friendly GUI to generate thousands of unique label images with precise control over visual characteristics. The program addresses the need for diverse, realistic label images in scenarios where collecting real-world data is impractical or insufficient. By incorporating multiple visual effects, font variations, and realistic distortions, the generator produces labels that closely resemble real-world specimens, making it ideal for training OCR systems, computer vision models, and document analysis algorithms.

Technical Documentation and Installation Manual

Document Version: January 1, 2026

Contents

1 Introduction

A wide ecosystem of software exists for designing and printing physical labels for products, shipping, or organization. Leading commercial tools such as Avery Design & Print, Canva, and Adobe Express provide user-friendly, template-based interfaces that enable individuals and businesses to create professional-looking labels for physical products and packaging [1][2][3]. For more demanding business needs, solutions like Label LIVE offer collaborative features and batch printing from spreadsheets, while enterprise-grade platforms such as BarTender provide automation, extensive barcode support, and deep integration with inventory and logistics systems for high-volume, compliance-focused environments [4][5]. In contrast, the Automated Label Generator described in this documentation serves a different and specialized purpose. It is not a tool for designing a specific label to be mass-printed. Instead, it is a synthetic data generation engine engineered to produce large, diverse, and realistic datasets of label images. This capability is crucial for training and evaluating machine learning models, particularly in the domains of Optical Character Recognition (OCR), document analysis, and computer vision. The software automates the creation of thousands of unique label images, each with randomized text (including scientific notation), fonts, styles, orientations, and controlled visual effects like aging and distortion.

An important feature of this program lies in its focus on automated variation and controlled realism. While traditional label software gives users full design control over a single output, this generator relinquishes direct control over individual labels to algorithmically ensure broad statistical diversity across a dataset. It simulates real-world conditions—such as inconsistent printing, paper textures, lighting variations, and minor damage—that are critical for building robust machine learning models but are difficult and expensive to collect from the physical world. Therefore, this tool fills a distinct gap, positioned not as a competitor to graphic design or industrial labeling suites, but as a specialized utility for AI/ML practitioners and researchers needing high-quality, customizable synthetic training data. Important features include the possibility to generate customizable text labels with scientific notation support, the implementation handles realistic visual effects including vintage appearance, gamma distortion, and JPEG artifacts. The GUI provides the users with flexible font and style configuration options, variable rotation and scaling options. The software also generates metadata for the generated labels and provides multiple options to export in multiple image formats (PNG, JPG, TIFF, BMP).

2 Overview of GUI Capabilities

The graphical user interface (GUI) provides intuitive control over all label generation parameters through a tabbed interface. Each panel corresponds to a specific aspect of label generation, allowing users to fine-tune the visual characteristics of the output labels.

2.1 General Settings

The General Settings panel controls the fundamental parameters of the generation process:

- **Number of Labels**: Specifies the total quantity of labels to generate (1-10,000 range)
- **Output Format**: Selects the image format (PNG, JPG, JPEG, TIFF, BMP)
- **Output Directory**: Sets the destination folder for generated labels and metadata

[1]https://www.avery.com/software/design-and-print/
[2]https://www.canva.com/create/labels/
[3]https://www.adobe.com/express/create/label
[4]https://learn.g2.com/best-label-printing-software
[5]https://selectedfirms.co/blog/best-label-printing-software

Important Note: JPG format does not support transparency. When selecting JPG output, transparent backgrounds will be converted to white or specified background colors.

2.2 Text Content

This panel manages the textual content displayed on labels:

- **Label Text Options**: List of base text values that can appear on labels. Multiple selections allowed.
- **Custom Text Addition**: Ability to add new text options during runtime
- **Scientific Notation**: Controls the probability (0.0-1.0) of converting numbers to scientific notation format

Special Functionality: When numbers with commas (e.g., "$10,000$") are selected and scientific notation is enabled, they are automatically converted to proper scientific format (e.g., "1.0×10^4").

2.3 Options for Units

Manages unit specifications and separators for label text:

- **Unit Selection**: Comprehensive list of measurement units (mg, mL, μ g, %, ppm, etc.)
- **Unit Management**: Interface for adding custom units and removing existing ones
- **Unit Separators**: Configurable spacing between text and units (spaces, dashes, custom separators)
- **Probability Control**: 70% default chance of appending units to text

Compatibility Note: Empty units (no unit) can be selected if text-only labels are required. Multiple separators can be selected simultaneously, with random selection during generation.

2.4 Font & Style

Controls typographic aspects of label generation:

- **Font Size**: Base size with configurable variation range ($\pm$ variation)
- **Font Families**: Multi-selection from system-available fonts with common fonts preselected
- **Font Weights**: Selection of normal, bold, and italic styles
- **Text Colors**: Color palette management with preset options and custom color picker

Visual Features: Color list displays visual swatches with automatic text contrast adjustment for readability.

2.5 Vintage & Background Effects

Combined panel for controlling visual effects and background properties:

2.5.1 Vintage Effects

- **Effect Probability**: Chance that any vintage effects are applied to a given label
- **Intensity**: Global multiplier for all vintage effect strengths
- **Texture File**: Path to background texture image (e.g., old paper)
- **Noise Intensity**: Controls grain/noise addition
- **Blur Intensity**: Gaussian blur application strength

2.5.2 Realism Effects

- **Enable Realism**: Toggle for advanced realism enhancements
- **Realism Intensity**: Controls strength of gamma distortion, scaling artifacts, and JPEG compression

2.5.3 Background Settings

- **Background Brightness**: Minimum brightness for generated backgrounds (ensures light backgrounds)
- **Transparent Background**: Probability of labels having transparent (no color) backgrounds

Critical Compatibility Note: Transparent backgrounds are incompatible with realism effects. When realism effects are enabled, transparent backgrounds will be converted to colored backgrounds to apply distortions properly.

2.6 Rotation Effects

Manages label orientation and rotation:

- **Rotation Enable/Disable**: Master toggle for rotation functionality
- **Allowed Angles**: Multi-selection of preset rotation angles (0°, 30°, 45°, 60°, 90°, 270°, 315°)
- **Custom Angle Step**: Granularity for random angle generation when "customize" option is selected

Implementation Detail: Rotation is applied after text rendering, with automatic canvas expansion to prevent cropping. Background color is preserved during rotation.

2.7 Size & Resolution

Controls physical dimensions and quality parameters:

- **Custom Size Toggle**: Enables/disables custom dimension ranges
- **Dimensions**: Minimum and maximum width/height ranges when custom size is enabled
- **DPI Settings**:
  - DPI range for custom-sized labels
  - Fixed DPI for non-custom labels

- **Text Padding**: Minimum space between text and label edges

Scaling Behavior: When custom dimensions are enabled, labels maintain aspect ratio while fitting within specified ranges. DPI settings affect the metadata embedded in image files.

3 GUI Control and Status Section

The bottom section of the GUI contains essential control elements, status indicators, and output displays that manage the label generation process. This section is divided into three main areas: status/progression tracking, console output display, and control buttons.

3.1 Status and Progression Panel

Located on the left side of the bottom section, this panel provides real-time feedback about the generation process:

- **Current Status Label**: Displays dynamic messages indicating the current state of the generator:
  - `"Ready to proceed."`: Initial state before generation begins
  - `"Generating label X/Y..."`: Active generation with progress indicator (where X is current label, Y is total labels)
  - `"Stopping generation..."`: When the stop button has been pressed
  - `"Generation completed"`: When all labels have been successfully generated
- **Progress Bar**: Visual indicator showing completion percentage of the current batch:
  - Range: 0-100% with text overlay showing exact percentage
  - Updates in real-time as each label is generated
  - Resets to 0% when starting a new generation batch
- **Generation Behavior**: The system generates labels sequentially from 1 to N, where N is specified in the General Settings tab. If generation is stopped before completion, the process cannot be resumed from the interruption point - it must be restarted from the beginning. All partially completed labels before the stop point are saved to disk, but the batch will be incomplete.

3.2 Prompt Outputs Panel

The right-side panel displays a live console output stream that captures all generation messages and system feedback:

- **Content Displayed**:
  - **Generation Logs**: Each generated label filename (e.g., `"Generated label: label_001.png"`)
  - **Metadata Confirmation**: Messages confirming metadata file creation (`"Metadata saved to: labels_metadata.csv and labels_metadata.txt"`)
  - **Completion Summary**: Final summary showing total labels generated and output directory
  - **Error Messages**: Any runtime errors or warnings encountered during generation
  - **System Messages**: Other Python console output redirected to this display

- **Technical Implementation**: This text area captures both `stdout` (standard output) and `stderr` (standard error) streams using a custom `OutputStream` class, providing a unified display of all console activity.
- **User Interaction Features**:
  - **Read-only Display**: Users cannot edit the output text, only view and copy
  - **Automatic Scrolling**: New messages automatically scroll into view
  - **Text Selection**: Users can select and copy text for troubleshooting or documentation
  - **Clear on Restart**: The output panel is automatically cleared when starting a new generation batch
- **Troubleshooting Utility**: This panel is essential for diagnosing issues such as:
  - Missing texture files or invalid paths
  - Font loading errors
  - Permission issues with output directories
  - Image format conversion problems

3.3 Control Buttons Section

The bottom row contains four action buttons that control the generation process and settings management:

3.3.1 Start Label Generation

- **Icon**: Green power symbol indicating activation
- **Function**: Initiates the label generation process with current settings
- **Pre-action Validation**: Before starting, the system performs comprehensive checks:
  - Output directory existence and write permissions
  - Texture file availability (if specified)
  - Valid parameter ranges (e.g., min $\leq$ max for dimensions)
  - Required selections (at least one text option, font family, etc.)
- **User Confirmation**: Displays a confirmation dialog showing the number of labels to generate
- **State Changes**: When clicked:
  - Start button becomes disabled
  - Stop button becomes enabled
  - All settings tabs become disabled (cannot modify settings during generation)
  - Output panel is cleared
  - Progress bar resets to 0%
- **Threaded Execution**: Generation runs in a separate thread to prevent GUI freezing

3.3.2 Stop Label Generation

- **Icon**: Red stop square indicating termination
- **Function**: stops the ongoing generation process
- **Behavior**:
  - Sets a stop flag that the generation thread checks between labels
  - Does not interrupt the current label being generated (completes it)
  - Does not delete already-generated labels
  - Saves metadata for all completed labels up to the stop point
- **Limitation**: Cannot be used to pause and resume - stopping requires starting over from the beginning
- **State Changes**: When clicked:
  - Stop button becomes disabled
  - Start button becomes enabled (for restarting)
  - Settings tabs remain disabled until generation fully stops

3.3.3 Save Settings

- **Icon**: Blue skip-forward symbol indicating forward movement/saving
- **Function**: Exports all current GUI settings to a JSON configuration file
- **File Format**: JSON (JavaScript Object Notation) with human-readable structure
- **Saved Parameters Include**:
  - **General Settings**: Number of labels, output format, output directory
  - **Text Content**: Selected text options, scientific notation probability
  - **Units**: Selected units, unit separators
  - **Font & Style**: Font families, sizes, weights, colors
  - **Vintage & Background**: All effect probabilities, intensities, paths
  - **Rotation**: Enabled state, allowed angles, custom step
  - **Size & Resolution**: Dimension ranges, DPI settings, padding
- **File Dialog**: Opens a standard file save dialog for choosing location and filename
- **Output Format Example**:

```
{
    "num_labels": 1000,
    "output_format": "png",
    "output_dir": "./my_labels",
    "label_text_options": ["1.0", "2.5", "10,000"],
    "units": ["mg", "mL", "%"],
    "vintage_effect_prob": 0.7,
    "realism_intensity": 0.5,
    "font_families": ["Arial", "Times New Roman"],
    "text_colors": ["#000000", "#333333"],
    "min_width": 200,
    "max_width": 400,
    "rotation_allowed": true,
    "rotation_angle_allowed": ["0", "45", "90"]
}
```

- **Use Cases**:
  - Saving frequently used configurations as presets
  - Backing up settings before experimentation
  - Sharing configurations between team members
  - Creating different configurations for different label types

3.3.4 Load Settings

- **Icon**: Orange skip-backward symbol indicating backward movement/loading
- **Function**: Imports settings from a previously saved JSON configuration file
- **Compatibility Features**:
  - **Backward Compatibility**: Handles older format files that might not have all current settings
  - **Missing Parameter Handling**: Uses default values for any parameters not in the loaded file
  - **Unit List Management**: Preserves custom units added to the available units list
- **Load Process**:
  1. Opens file selection dialog to choose JSON settings file
  2. Validates JSON format and required structure
  3. Applies loaded settings to all GUI controls
  4. Updates calculated properties (dimensions, etc.)
  5. Refreshes all tabs to reflect loaded values
  6. Preserves current tab selection
- **File Dialog**: Opens a standard file open dialog for selecting existing settings files
- **Validation**: Basic JSON parsing validation; invalid files display error messages
- **Use Cases**:
  - Restoring previous working configurations
  - Applying standardized settings across multiple generation sessions
  - Switching between different label generation presets
  - Recovering from accidental setting changes
- **Important Notes**:
  - Loading settings does **not** automatically start generation
  - Settings are applied immediately but can be further modified before generation
  - File paths (output directory, texture file) are loaded as-is and may need adjustment if moved to a different computer
  - Font selections are validated against available system fonts; unavailable fonts are ignored

3.4 Integration and Workflow

The control section elements work together to support a typical workflow:

1. **Configuration**: User sets desired parameters across all tabs
2. **Optional Save**: User saves settings as a preset (optional but recommended)
3. **Validation and Start**: User clicks "Start Label Generation," system validates settings, user confirms
4. **Monitoring**: User watches progress bar and output messages during generation
5. **Optional Stop**: User can stop generation if needed (restart required to continue)
6. **Completion**: System displays completion message and saves metadata
7. **Review**: User can review output files and console messages

3.5 Troubleshooting Common Issues

- **Start Button Disabled**: Ensure all required fields are filled and valid; check output directory permissions
- **No Output Messages**: Generation may not have started; check for hidden error dialogs
- **Settings Not Loading**: Verify JSON file format hasn't been manually edited incorrectly
- **Progress Bar Stuck**: Generation may be paused on a problematic label; check output panel for error messages
- **Font Issues After Loading**: Some saved fonts may not be available on the current system; check output panel for warnings

This comprehensive control section provides users with complete visibility and control over the label generation process, from configuration through execution to output management.

4 Detailed Description of Vintage and Background Effects

4.1 Vintage Effects: Functional Architecture

The vintage effects system employs a multi-layered approach to simulate aged or distressed label appearances. This system is structured with two primary control mechanisms:

4.1.1 Probability-Based Application

- **Vintage Effect Probability (0.0-1.0)**: This parameter determines the likelihood that *any* vintage effects will be applied to an individual label. For example:
  - Setting to 0.3: 30% of generated labels will receive vintage effects
  - Setting to 0.7: 70% of labels will appear aged
  - Setting to 0.0: No vintage effects applied to any labels
- **Conditional Application**: Each specific vintage effect (blur, texture, sepia, noise) has its own independent probability threshold, creating a layered application system. For instance, even if vintage effects are selected for a label, individual effects like texture overlay or noise addition may not be applied based on their specific probability checks.

4.1.2 Intensity-Based Modulation

- **Vintage Intensity (0.0-1.0)**: Acts as a global multiplier that scales the strength of all vintage effects uniformly. This parameter:
  - At 0.0: Vintage effects are completely disabled
  - At 0.5: All vintage effects operate at half their maximum strength
  - At 1.0: Vintage effects operate at their maximum configured strength
- **Individual Intensity Controls**: Several effects have dedicated intensity parameters that work in conjunction with the global vintage intensity:
  - **Blur Intensity**: Controls Gaussian blur radius (0.0-2.0 range)
  - **Noise Intensity**: Governs the standard deviation of Gaussian noise added to pixel values
  - These individual intensities are multiplied by the global vintage intensity for final effect strength

4.2 Vintage Effect Components

4.2.1 Blur Application

- **Purpose**: Simulates out-of-focus camera capture or physical degradation
- **Control Logic**:
  - 80% probability of application when vintage effects are active
  - Final blur radius = `blur_intensity × vintage_intensity`
  - Uses Gaussian blur filter with configurable radius
- **Visual Impact**: Higher values create more pronounced softening, mimicking poor focus or motion blur

4.2.2 Texture Overlay

- **Purpose**: Adds physical texture patterns (e.g., paper grain, fabric weave)
- **Control Logic**:
  - 60% probability of application
  - Requires valid texture file (e.g., `old_paper.png`)
  - Texture converted to grayscale and blended at 10% opacity
  - Automatically resized to match label dimensions
- **Technical Note**: If texture file is missing or invalid, this effect silently fails without affecting other vintage effects

4.2.3 Sepia Toning

- **Purpose**: Creates warm, brownish tones characteristic of aged photographs
- **Control Logic**:
  - 70% probability of application
  - Color transformation matrix dynamically adjusted based on vintage intensity

  - Enhanced red and green channels to create warm tones
  - Blue channel attenuated for aged appearance

- **Mathematical Basis**: RGB color transformation using a 3×4 matrix with intensity-dependent coefficients

4.2.4 Noise Addition

- **Purpose**: Simulates film grain or digital sensor noise
- **Control Logic**:
  - 50% probability of application
  - Noise magnitude = 20 × `noise_intensity × vintage_intensity`
  - Gaussian noise distribution with zero mean
  - Pixel values clipped to valid RGB range [0, 255]
- **Implementation Detail**: Noise is added independently to each RGB channel, creating subtle color variations

4.2.5 Brightness Reduction

- **Purpose**: Simulates fading or light exposure over time
- **Control Logic**:
  - Always applied when vintage effects are active
  - Brightness reduction factor = 0.2 × `vintage_intensity`
  - Applied through multiplicative brightness enhancement (0.8-1.0 range)
- **Visual Effect**: Creates overall darkening while preserving contrast relationships

4.3 Realism Effects: Technical Architecture

4.3.1 Realism Effect Toggle

The "Enable Realism" checkbox activates a suite of advanced image processing effects designed to simulate real-world imaging artifacts and distortions. When enabled, the following effects become available for random application:

- **Realistic Scaling** (Probability: `realism_intensity`)
  - Random scaling between 0.5× and 2.0× original size
  - Scale range expands with higher realism intensity
  - Random selection of interpolation methods (nearest, bilinear, bicubic)
- **JPEG Artifacts** (Probability: `realism_intensity`)
  - Simulates compression artifacts from JPEG encoding
  - Quality range: 20-90, inversely related to realism intensity
  - Implementation: Encode/decode cycle with configurable quality
- **Subpixel Shift** (Probability: `realism_intensity`/2)
  - Creates chromatic aberration by shifting color channels

    - Random horizontal shift of ±1 pixel per channel
    - Separate handling for RGB vs RGBA image modes

- **Gamma Distortion** (Probability: `realism_intensity`)
    - Applies non-linear gamma correction ($\gamma = 1.5 - 3.0$)
    - Gamma range expands with realism intensity
    - Special handling for transparent images

- **Complex Background** (Probability: `realism_intensity`)
    - Adds grid patterns and random noise to background
    - Grid spacing decreases with higher realism intensity
    - Noise density increases with realism complexity

- **Font Rendering Variation** (Probability: `realism_intensity`/3)
    - Simulates printer/font rendering variations
    - Random morphological operations (erosion/dilation)
    - Kernel size increases with realism intensity

4.3.2 Critical Compatibility Constraint: Transparency and Realism Effects

Fundamental Incompatibility: Transparent backgrounds (alpha channel) cannot coexist with several realism effects due to fundamental image processing limitations:

- **Core Issue**: Many realism effects require opaque backgrounds for proper operation:
    - **Gamma Distortion**: Requires numerical operations on all pixels, including background
    - **Complex Background**: Adds new background elements that would overwrite transparency
    - **Subpixel Shift**: Channel separation assumes opaque background for clean shifting

- **Technical Implementation Review**:

```
In RealismEnhancer.apply_mode_aware():
    if is_transparent and original_mode == 'RGBA':
        # Separate alpha channel
        r, g, b, a = img.split()
        rgb_image = Image.merge('RGB', (r, g, b))

        # Apply effects to RGB only
        ... [effects applied to rgb_image] ...

        # Merge back with original alpha
        r2, g2, b2 = rgb_image.split()
        return Image.merge('RGBA', (r2, g2, b2, a))
```

- **Problematic Cases**:

  - `add_complex_background()`: Creates a new opaque white background with grid, completely replacing transparency
  - `apply_gamma_distortion()`: Mathematical operations on RGBA images may produce unexpected alpha values
  - `add_jpeg_artifacts()`: JPEG format doesn't support alpha channels, requiring conversion

- **User Workflow Requirement**:

  1. To use transparent backgrounds: Set `transparent_bg_prob > 0` AND disable realism effects (`add_realism = False`)
  2. To use realism effects: Set `transparent_bg_prob = 0` AND enable realism effects (`add_realism = True`)
  3. The software *does not* automatically disable realism when transparency is requested - this must be done manually by the user

- **Error Prevention**: If both are enabled simultaneously, the generation process will attempt to:

  - Apply realism effects to RGBA images
  - Potentially create visual artifacts or incorrect alpha channel values
  - May produce runtime errors in specific effect combinations

4.3.3 Realism Intensity Parameter

The realism intensity parameter serves multiple functions:

- **Probability Modulator**: Higher intensity increases the likelihood of each realism effect being applied

- **Effect Strength Controller**: Scales the magnitude of individual effects

  - Gamma range: [2.4 - intensity, 2.4 + intensity]
  - JPEG quality: 90 - (intensity $\times$ 50)
  - Background complexity: intensity $\times$ 5

- **Threshold Adjuster**: Some effects (font rendering variation) use reduced probability (`intensity/3`) for more subtle application

4.4 Background Generation System

4.4.1 Light Background Generation Algorithm

The background color generation system ensures visually pleasing, light-colored backgrounds through controlled HSV space manipulation:

1. **Hue Selection**: Random hue value [0, 1] across full color spectrum

2. **Saturation Control**: Limited to [0.0, 0.3] to maintain pastel/light appearance

3. **Brightness Guarantee**: Value (brightness) constrained to [min_background_brightness, 1.0]

4. **Conversion**: HSV $\rightarrow$ RGB conversion with 8-bit quantization

Mathematical Representation:

$$\begin{aligned}
&h \sim U(0, 1)\\
&s \sim U(0.0, 0.3)\\
&v \sim U(\text{min\_brightness}, 1.0)\\
&(r, g, b) = \text{HSVtoRGB}(h, s, v)\\
&\text{hex\_color} = \#\{r : 02x\}\{g : 02x\}\{b : 02x\}
\end{aligned} \tag{1}$$

4.4.2 Transparent Background System

The transparent background functionality operates through a probabilistic system:

- **Probability Control**: `transparent_bg_prob` determines the chance of generating labels with transparent (alpha = 0) backgrounds
- **Image Mode Selection**:
  - Transparent: RGBA mode with (0,0,0,0) background
  - Opaque: RGB mode with generated background color
- **Rendering Considerations**: Text must be drawn with RGBA color tuples when using transparent backgrounds

4.5 Effect Application Pipeline

The complete effect application follows a specific sequence to ensure proper visual results:

1. **Base Label Creation**: Text rendering on generated background (RGB or RGBA)
2. **Rotation Application**: Canvas expansion and rotation with proper fillcolor handling
3. **Vintage Effects Application**: Conditional application based on probability settings
   - Mode conversion: RGBA $\rightarrow$ RGB for vintage effects (alpha preserved)
   - Effect application to RGB channels
   - Alpha channel re-application
4. **Realism Effects Application**: Only if enabled and compatible with current image mode
   - `apply_mode_aware()` handles transparent vs opaque differentiation
   - Effects applied in randomized order based on probabilities
5. **Final Format Conversion**: Based on output format requirements
   - JPG: Conversion from RGBA to RGB with background color fill
   - PNG: Preservation of alpha channel if present

4.6 Technical Implementation Details

4.6.1 Image Mode Management

The software maintains rigorous image mode tracking throughout the processing pipeline:

- **Mode Transitions**:

```
Transparent workflow: RGBA → [effects with alpha separation] → RGBA
Opaque workflow: RGB → [effects] → RGB
JPG output: Any mode → RGB → JPG
```

- **Alpha Channel Preservation**: When effects require RGB processing on RGBA images:

```
def apply_effect_to_rgba(image, effect_function):
    r, g, b, a = image.split()
    rgb = Image.merge('RGB', (r, g, b))
    rgb_processed = effect_function(rgb)
    r2, g2, b2 = rgb_processed.split()
    return Image.merge('RGBA', (r2, g2, b2, a))
```

4.6.2 Probability-Based Randomization System

Each effect employs independent random checks, creating natural variation:

```
def apply_vintage_effects(image):
    if random.random() < vintage_probability:
        # Apply vintage effects with individual probabilities
        if random.random() < 0.8: apply_blur()
        if random.random() < 0.6: apply_texture()
        if random.random() < 0.7: apply_sepia()
        if random.random() < 0.5: add_noise()
        apply_brightness_reduction()  # Always applied
    return image
```

4.6.3 Intensity Scaling Mathematics

Effect parameters are dynamically scaled based on intensity settings:

- **Linear Scaling**: $parameter_{final} = base_{value} \times intensity$
- **Range Scaling**: $range = [base - intensity, base + intensity]$
- **Inverse Scaling**: $quality = 90 - (50 \times intensity)$ for JPEG artifacts

4.7 Practical Usage Guidelines

4.7.1 Recommended Parameter Combinations

Use Case	Vintage Intensity	Realism Intensity	Transparency
Clean modern labels	0.0-0.3	0.0-0.2	0.0
Aged document labels	0.7-1.0	0.3-0.5	0.0
Transparent overlays	0.0	0.0	0.5-1.0
Realistic scanned labels	0.4-0.6	0.6-0.8	0.0
Subtle variations	0.2-0.4	0.1-0.3	0.0

Table 1: Recommended parameter combinations for common use cases

4.7.2 Troubleshooting Common Issues

- **Black backgrounds in transparent mode**: Ensure text colors are specified with full opacity (e.g., #000000FF for RGBA)
- **Vintage effects not appearing**: Check both vintage probability AND individual effect probabilities
- **Realism effects creating artifacts**: Reduce realism intensity or disable specific effects in code
- **Transparency lost in JPG output**: JPG format doesn't support transparency - use PNG for transparent backgrounds

4.7.3 Performance Considerations

- **Processing Time**: Each effect adds computational overhead. Complex combinations may significantly increase generation time
- **Memory Usage**: Large label counts with multiple effects may require substantial RAM
- **Disk Space**: High-quality effects with large dimensions produce larger file sizes

4.8 Advanced Customization Options

For advanced users willing to modify the source code:

- **Effect Order Modification**: Change the sequence in `RealismEnhancer.apply()` method
- **Custom Effect Creation**: Add new effect functions following the existing pattern
- **Probability Curve Adjustment**: Modify probability distributions for more controlled randomization
- **Intensity Mapping Functions**: Replace linear scaling with logarithmic or exponential functions

This detailed analysis provides both user-level understanding and technical implementation insights for the vintage and background effects system, enabling effective configuration and troubleshooting.

5 Description of Main Functions and Algorithms

5.1 Label Generation Pipeline

The label generation follows a sequential pipeline:

1. **Text Generation**: Random selection from text options with scientific notation conversion
2. **Font Selection**: Random choice from enabled font families, sizes, and weights
3. **Canvas Creation**: Dynamic canvas sizing based on text dimensions and padding
4. **Background Generation**: Color generation in HSV space with guaranteed brightness
5. **Text Rendering**: Precise centering with font metric calculations
6. **Rotation Application**: Canvas expansion and rotation with background preservation
7. **Visual Effects**: Sequential application of vintage and realism effects
8. **Format Conversion**: Final conversion to output format with DPI embedding

5.2 Scientific Notation Algorithm

The scientific notation conversion uses logarithmic calculations:

$$\text{exponent} = \lfloor \log_{10}(\text{value}) \rfloor \tag{2}$$

$$\text{coefficient} = \frac{\text{value}}{10^{\text{exponent}}} \tag{3}$$

Implementation Details:

- For comma-separated numbers (e.g., "10,000"): Convert to 1.0×10^4 format
- Coefficient precision: Random decimal places (1-3)
- Unicode superscript characters used for visual accuracy
- Alternative formats: "1.23E4" or "1.23e4" for variety

5.3 Realism Enhancement Algorithms

5.3.1 Gamma Distortion

$$I_{\text{out}} = I_{\text{in}}^{\gamma} \quad \text{where} \quad \gamma \in [1.5, 3.0] \tag{4}$$

Gamma correction simulates monitor calibration differences and aging effects.

5.3.2 JPEG Artifact Simulation

Controlled quality degradation (20-90 quality range) followed by re-encoding simulates compression artifacts common in real-world images.

5.3.3 Subpixel Shifting

Random channel offset by ±1 pixel creates chromatic aberration effects:

$$R_{\text{shifted}} = \text{offset}(R, \Delta x, 0) \tag{5}$$

$$G_{\text{shifted}} = \text{offset}(G, 0, 0) \quad \text{(reference)} \tag{6}$$

$$B_{\text{shifted}} = \text{offset}(B, -\Delta x, 0) \tag{7}$$

5.3.4 Complex Background Generation

Grid pattern with configurable spacing:

$$\text{grid\_size} = 40 - (6 \times \text{complexity}) \tag{8}$$

Noise point density increases with complexity parameter.

5.4 Color Space Operations

Background color generation uses HSV space for controlled brightness:

- Hue (H): Random [0,1]
- Saturation (S): Limited to [0, 0.3] for pastel colors
- Value (V): Limited to [min_brightness, 1.0]

Conversion to RGB:

$$(r, g, b) = \text{hsv\_to\_rgb}(h, s, v) \tag{9}$$

5.5 Image Mode Management

Sophisticated mode handling ensures compatibility:

- RGBA for transparent backgrounds
- RGB for colored backgrounds
- Automatic conversion for JPG output
- Alpha channel preservation during transformations

6 Installation and Configuration

6.1 Installation of Environment and Required Dependencies

This section provides a step-by-step procedure to set up the working environment[6] and install the Python distribution. Follow these steps to set up the required software environment on Windows 10/11:

6.1.1 Install Anaconda Distribution

Anaconda is a free package manager that simplifies Python installation and dependency management.

1. Visit the official Anaconda website: https://www.anaconda.com/products/distribution
2. Download the **64-bit Graphical Installer** for Windows
3. Double-click the downloaded .exe file and follow the installation wizard:
   - Use default settings for all installation options
   - Check "Add Anaconda3 to my PATH environment variable"
   - Select "Register Anaconda3 as my default Python 3.10"

Verify the installation by opening the **Anaconda Prompt** from the Start Menu and running:

```
conda --version
```

This should display the installed conda version (e.g., `conda 23.1.0`).

6.1.2 Create a Virtual Environment

Create a virtual environment specifically for the label generator:

```
conda create -n labelgen python=3.10
```

Activate the environment using:

```
conda activate labelgen
```

After activation, your command prompt should show `(labelgen)` at the beginning.

[6] https://www.anaconda.com/distribution/

6.1.3 Install Required Packages

Install the following essential libraries in your virtual environment:

```
conda install numpy matplotlib pillow scipy pyqt -c conda-forge
```

The installed packages serve specific purposes:

- **NumPy**[7]: Numerical computing for array operations and mathematical functions
- **Matplotlib**[8]: Font management and system font detection
- **Pillow**[9]: Core image processing, text rendering, and effects
- **SciPy**[10]: Scientific computing functions
- **PyQt6**[11]: Graphical user interface framework

Verify package installation using:

```
conda list
```

6.1.4 Prepare the Workspace

1. Create a project folder (e.g., `C:\LabelGenerator`)
2. Place these files in the folder:
   - `label_generator_core.py`
   - `label_generator_gui.py`
   - `old_paper.png` (optional texture file)
   - Icon files (if available): `control-power.png`, `control-stop-square.png`, etc.
3. Open Anaconda Prompt and navigate to your project:

   ```
   cd C:\LabelGenerator
   ```

6.1.5 Troubleshooting Tips

- If you encounter ”DLL load failed” errors, install Microsoft Visual C++ Redistributable
- For font rendering issues, ensure system fonts are properly installed
- Update packages with `conda update --all` if experiencing version conflicts
- Refer to Anaconda documentation[12] for advanced configuration

7 API Documentation

7.1 Core Module: label_generator_core.py

The core module implements the fundamental label generation algorithms, visual effects, and configuration management. This module operates independently of the GUI and can be imported programmatically for batch processing or integration into other applications.

[7] https://numpy.org/doc/stable/
[8] https://matplotlib.org/stable/users/index.html
[9] https://pillow.readthedocs.io/en/stable/
[10] https://docs.scipy.org/doc/scipy/
[11] https://www.riverbankcomputing.com/software/pyqt/
[12] https://docs.anaconda.com/

7.1.1 Main Classes and Architecture

Class Structure Overview

The module implements three primary classes:

- `LabelGeneratorSettings`: Configuration container with validation
- `LabelGenerator`: Main generation engine with text, rendering, and export
- `RealismEnhancer`: Specialized effects processor for realistic artifacts

Data Flow: Settings → Generator → Image Creation → Effects Application → Export

7.1.2 LabelGeneratorSettings Class

The `LabelGeneratorSettings` class encapsulates all configurable parameters for label generation. It provides intelligent defaults and validation logic.

__init__() - Settings Initialization

```
def __init__(self):
    # Basic configuration
    self.num_labels = 2000
    self.output_format = 'png'
    self.output_dir = './test_labels-5-270'

    # Visual effects
    self.vintage_intensity = 0.7
    self.add_realism = True
    self.realism_intensity = 0.7

    # Text content parameters
    self.label_text_options = ["0.5", "1.0", "2.7", ...]
    self.scientific_notation_prob = 0.35

    # Units configuration with comprehensive list
    self.available_units = ["", "  mg", "  mL", "  μg", ...]
    self.units = self.available_units[:]  # All units by default

    # Font and styling with system font detection
    self.font_families = self.get_safe_fonts()

    # Calculated properties
    self.update_calculated_properties()
```

Initialization Strategy: Three-phase initialization - default values, system detection (fonts), calculated properties

update_calculated_properties() - Dynamic Configuration

```python
def update_calculated_properties(self):
    """Update calculated properties based on current settings"""
    max_font_size = self.base_font_size + self.font_size_variation

    if self.customized_size_resolution:
        self.image_width = random.randint(self.min_width, self.max_width)
        self.image_height = random.randint(self.min_height, self.max_height)
    else:
        self.image_width = max(150, int(max_font_size * 8))
        self.image_height = max(60, int(max_font_size * 3))
```

Purpose: Dynamically computes dependent properties (image dimensions) based on font settings and resolution flags. Ensures consistency between configuration parameters.

get_safe_fonts() - System Font Discovery

```python
def get_safe_fonts(self):
    """Get list of safe fonts that can render basic text"""
    system_fonts = fm.findSystemFonts()
    safe_fonts = []
    for fpath in system_fonts:
        try:
            font = fm.get_font(fpath)
            if font.style.find('Regular') != -1 and font.variant.find('normal')
            ↪  != -1:
                safe_fonts.append(font.name)
        except:
            continue
    return list(set(safe_fonts)) + ['DejaVu Sans', 'Arial', 'Verdana', 'Times
    ↪  New Roman']
```

Algorithm: Scans system font directories, validates each font can render basic text, filters for regular/normal variants, and adds common fallback fonts.

generate_light_background() - Color Generation

```python
def generate_light_background(self):
    """Generate a light background color with guaranteed brightness"""
    h = random.random()  # Hue: random [0,1]
    s = random.uniform(0.0, 0.3)  # Saturation: low for pastel
    v = random.uniform(self.min_background_brightness, 1.0)  # Value: controlled
    ↪  brightness

    r, g, b = colorsys.hsv_to_rgb(h, s, v)
    r, g, b = int(r * 255), int(g * 255), int(b * 255)
    return f"#{r:02x}{g:02x}{b:02x}"
```

Color Space Strategy: Uses HSV space for intuitive control over hue, saturation, and brightness. Guarantees minimum brightness through the `min_background_brightness` parameter. Returns hex color string for consistent formatting.

7.1.3 LabelGenerator Class

The main engine class responsible for orchestrating label generation, text creation, image rendering, and file export.

__init__() - Generator Initialization

```
def __init__(self, settings):
    self.settings = settings
    self.settings.update_calculated_properties()
    self.metadata = []
```

Initialization Pattern: Accepts a `LabelGeneratorSettings` instance, ensures calculated properties are up-to-date, and initializes an empty metadata store for tracking generation results.

to_superscript() - Unicode Conversion

```
def to_superscript(self, num):
    """Convert numbers to Unicode superscript characters"""
    superscript_map = {
        '0': '⁰', '1': '¹', '2': '²', '3': '³', '4': '⁴',
        '5': '⁵', '6': '⁶', '7': '⁷', '8': '⁸', '9': '⁹',
        '-': '⁻', '+': '⁺'
    }
    return ''.join(superscript_map.get(char, char) for char in str(num))
```

Mapping Strategy: Uses a direct character-to-character mapping for mathematical superscript symbols. Supports negative and positive exponents through '-' and '+' characters.

generate_label_text() - Text Generation Algorithm

```
def generate_label_text(self):
    """Generate label text with proper scientific notation"""
    # 1. Base text selection
    base_text = random.choice(self.settings.label_text_options)

    if base_text == "customize":
        base_text = str(round(random.uniform(1, 60), 4))

    # 2. Scientific notation conversion (comma-separated numbers)
    if random.random() < self.settings.scientific_notation_prob and ',' in
    ↪   base_text:
        value = float(base_text.replace(',', ''))
        exponent = int(np.floor(np.log10(value)))
        coefficient = value / (10 ** exponent)

        dec_places = random.randint(1, 3)
        coef_str = format(coefficient,
        ↪   f'.{dec_places}f').rstrip('0').rstrip('.')
        exp_str = self.to_superscript(exponent)
        base_text = f"{coef_str} × 10{exp_str}"

    # 3. Alternative scientific notation (E-notation)
    elif random.random() < self.settings.scientific_notation_prob:
        value = round(random.uniform(1, 60), 3)
        exp_str = random.randint(-14, 14)
        if random.uniform(1, 2) < 1.5:
            base_text = str(value) + "E" + str(exp_str)
        else:
            base_text = str(value) + "e" + str(exp_str)

    # 4. Unit attachment (70% probability)
    if random.random() > 0.3 and self.settings.units:
        separator = random.choice(self.settings.unit_separator)
        unit = random.choice(self.settings.units)
        return base_text + separator + unit

    return base_text
```

Multi-Stage Text Generation:

1. Base selection from predefined or random custom values
2. Scientific notation for comma-separated numbers (e.g., 10000 → 1.0×10^4)
3. Alternative E-notation for variety
4. Unit attachment with configurable separators

Precision Control: Random decimal places (1-3) for scientific coefficients

determine_rotation_angle() - Angle Selection Logic

```python
def determine_rotation_angle(self):
    """Determine rotation angle based on settings"""
    if not self.settings.rotation_allowed:
        return 0

    angle_type = random.choice(self.settings.rotation_angle_allowed)

    if angle_type == 'customize':
        # Generate custom angle (multiple of step between 0-90)
        angle = random.randrange(5, 80, self.settings.custom_angle_step)
        # Avoid duplicating preset angles
        while angle in [0, 45, 90]:
            angle = random.randrange(5, 80, self.settings.custom_angle_step)
        return angle
    else:
        return int(angle_type)
```

Rotation Strategy: Two-tier system:

- Preset angles: Direct mapping from string to integer
- Custom angles: Generates random angles in 5°-80° range with configurable step, avoiding common presets

Angle Validation: Ensures custom angles don't duplicate preset values (0°, 45°, 90°)

apply_vintage_effects() - Multi-Effect Pipeline

```python
def apply_vintage_effects(self, image, intensity=0.7):
    """Apply vintage effects to label images"""
    try:
        # 1. Gaussian Blur (80% probability)
        if random.random() < 0.8:
            image = image.filter(ImageFilter.GaussianBlur(
                radius=self.settings.blur_intensity * intensity
            ))

        # 2. Texture Overlay (60% probability)
        if random.random() < 0.6:
            try:
                texture = Image.open(self.settings.texture_file).convert('L')
                texture = texture.resize(image.size)
                image = Image.blend(image.convert('RGB'),
                ↪  texture.convert('RGB'), 0.1)
            except:
                pass  # Silently fail if texture missing

        # 3. Sepia Toning (70% probability)
        if random.random() < 0.7:
            sepia_filter = (
                0.393 + 0.1*intensity, 0.769, 0.189, 0,
                0.349, 0.686 + 0.1*intensity, 0.168, 0,
                0.272, 0.534, 0.131 + 0.1*intensity, 0
            )
            image = image.convert('RGB', matrix=sepia_filter)

        # 4. Gaussian Noise (50% probability)
        if random.random() < 0.5:
            arr = np.array(image).astype(np.float32)
            noise = np.random.normal(0, 20*intensity, arr.shape)
            noisy = np.clip(arr + noise, 0, 255).astype(np.uint8)
            image = Image.fromarray(noisy)

        # 5. Brightness Reduction (always applied)
        enhancer = ImageEnhance.Brightness(image)
        return enhancer.enhance(1 - 0.2*intensity)

    except Exception as e:
        print(f"Error applying vintage effects: {str(e)}")
        return image  # Graceful degradation
```

Layered Effects Pipeline: Five distinct vintage effects applied conditionally with independent probabilities. Each effect intensity scales with the global `intensity` parameter. **Error Handling**: Graceful degradation - if any effect fails, the function returns the original or partially processed image with error logging.

create_label_image() - Complete Generation Pipeline

```python
def create_label_image(self, label_idx):
    """Create a single label image with metadata - FIXED VERSION"""
    # Phase 1: Text Generation and Styling
    label_text = self.generate_label_text()
    rotation_angle = self.determine_rotation_angle()

    # Phase 2: Font and Color Selection
    font_family = random.choice(self.settings.font_families)
    font_size = self.settings.base_font_size + random.randint(
        -self.settings.font_size_variation,
        self.settings.font_size_variation
    )
    font_weight = random.choice(self.settings.font_weights)
    text_color = random.choice(self.settings.text_colors)

    # Phase 3: Canvas Creation and Text Measurement
    generated_bg_color = self.settings.generate_light_background()
    use_transparent_bg = random.random() < self.settings.transparent_bg_prob

    # Phase 4: Text Rendering with proper mode handling
    if use_transparent_bg:
        canvas = Image.new('RGBA', (canvas_width, canvas_height), (0, 0, 0, 0))
    else:
        canvas = Image.new('RGB', (canvas_width, canvas_height),
        ↪  generated_bg_color)

    # Phase 5: Rotation Application
    if rotation_angle != 0:
        # Canvas expansion, rotation, and cropping
        canvas = self._apply_rotation(canvas, rotation_angle,
                                      generated_bg_color, use_transparent_bg)

    # Phase 6: Vintage Effects (with mode awareness)
    apply_vintage = random.random() < self.settings.vintage_effect_prob
    if apply_vintage:
        canvas = self._apply_vintage_with_mode(canvas, use_transparent_bg)

    # Phase 7: Realism Effects (if enabled)
    if self.settings.add_realism:
        realism = RealismEnhancer(self.settings)
        canvas = realism.apply_mode_aware(canvas.copy(), use_transparent_bg)

    # Phase 8: Metadata Collection
    metadata = {
        "label_id": label_idx,
        "text": label_text,
        "rotation_angle": rotation_angle,
        # ... additional metadata fields
    }

    return canvas, metadata
```

8-Phase Generation Pipeline:

1. **Text Generation**: Content creation with scientific notation
2. **Styling**: Font, color, and rotation decisions
3. **Canvas Setup**: Mode-aware background creation
4. **Text Rendering**: Precise font metric calculations
5. **Rotation**: Canvas expansion and transformation

generate_all_labels() - Batch Processing Engine

```
def generate_all_labels(self):
    """Generate all labels and save with metadata"""
    os.makedirs(self.settings.output_dir, exist_ok=True)

    for i in range(self.settings.num_labels):
        label_idx = i + 1

        # 1. Image Generation
        image, metadata = self.create_label_image(label_idx)

        # 2. DPI Determination
        if self.settings.customized_size_resolution:
            dpi = random.randint(self.settings.min_dpi, self.settings.max_dpi)
        else:
            dpi = self.settings.fixed_dpi

        # 3. Format-Specific Processing
        img_filename =
        ↪  f"label_270_{label_idx:03d}.{self.settings.output_format}"
        img_path = os.path.join(self.settings.output_dir, img_filename)

        # 4. JPG Special Handling
        if self.settings.output_format.lower() in ['jpg', 'jpeg']:
            if image.mode in ['RGBA', 'LA']:
                bg_color = metadata['background']
                if bg_color == "transparent":
                    bg_color = "#FFFFFF"
                background = Image.new('RGB', image.size, bg_color)
                background.paste(image, mask=image.split()[3] if image.mode ==
                ↪  'RGBA' else None)
                image = background
            save_params['quality'] = 95

        # 5. Image Saving with Metadata
        image.save(img_path, **save_params)

        # 6. Metadata Accumulation
        metadata["image_filename"] = img_filename
        self.metadata.append(metadata)

    # 7. Final Metadata Export
    csv_path, txt_path = self.save_metadata()
```

Batch Processing Strategy: Sequential generation of all configured labels with consistent per-label processing:

- **Directory Management**: Ensures output directory exists
- **Format Adaptation**: Special handling for JPG transparency conversion
- **Progressive Metadata**: Accumulates metadata during generation
- **Final Export**: Saves both CSV and TXT metadata formats

Error Resilience: Individual label failures don't stop batch processing

save_metadata() - Dual-Format Export

```python
def save_metadata(self):
    """Save metadata to both CSV and TXT files"""
    # CSV file (preserves Unicode, Excel-compatible)
    csv_path = os.path.join(self.settings.output_dir, "labels_metadata.csv")
    fieldnames = [
        "label_id", "image_filename", "text", "rotation_angle",
        "font_family", "font_size", "font_weight", "text_color",
        "background", "vintage_applied", "vintage_intensity"
    ]

    with open(csv_path, 'w', newline='', encoding='utf-8-sig') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(self.metadata)

    # TXT file (plain-text with encoded scientific notation)
    txt_path = os.path.join(self.settings.output_dir, "labels_metadata.txt")
    with open(txt_path, 'w', encoding='utf-8') as txtfile:
        headers = "\t".join(fieldnames)
        txtfile.write(f"{headers}\n")

        for row in self.metadata:
            encoded_row = row.copy()
            encoded_row['text'] = encode_scientific_notation(row['text'])
            line = "\t".join(str(encoded_row[field]) for field in fieldnames)
            txtfile.write(f"{line}\n")

    return csv_path, txt_path
```

Dual-Format Strategy:

- **CSV Format**: Uses UTF-8 with BOM (`utf-8-sig`) for Excel compatibility
- **TXT Format**: Tab-separated values with scientific notation encoding

Scientific Notation Encoding: Converts Unicode superscript (e.g., 10^4) to plain-text representation (e.g., 10000) for maximum compatibility

7.1.4 RealismEnhancer Class

Specialized class for applying realistic imaging artifacts and distortions. Implements mode-aware processing for transparent images.

apply_mode_aware() - Transparency-Compatible Processing

```
def apply_mode_aware(self, image, is_transparent=False):
    """Apply realism effects with mode awareness"""
    img = image.copy()  # Work on copy to avoid closed file issues
    intensity = self.settings.realism_intensity

    if is_transparent and original_mode == 'RGBA':
        # Separate alpha channel for RGB-only processing
        r, g, b, a = img.split()
        rgb_image = Image.merge('RGB', (r, g, b))

        # Apply effects to RGB only
        if random.random() < intensity:
            rgb_image = self.apply_realistic_scaling(rgb_image)
        if random.random() < intensity:
            rgb_image = self.add_jpeg_artifacts(rgb_image)
        # ... additional effects

        # Merge back with original alpha
        r2, g2, b2 = rgb_image.split()
        return Image.merge('RGBA', (r2, g2, b2, a))
    else:
        # Apply normally for RGB images
        return self.apply(img)
```

Transparency Handling Strategy: For RGBA images, separates alpha channel, applies effects to RGB only, then recombines. This preserves transparency while allowing realism effects. **Critical Limitation**: Some realism effects (like `add_complex_background`) are incompatible with transparency and will override the alpha channel.

apply_realistic_scaling() - Random Resampling

```
def apply_realistic_scaling(self, image):
    # Scale range based on intensity
    min_scale = max(0.5, 1.0 - self.settings.realism_intensity/2)
    max_scale = min(2.0, 1.0 + self.settings.realism_intensity)

    scale_factor = random.uniform(min_scale, max_scale)
    new_size = (int(image.width * scale_factor),
                int(image.height * scale_factor))

    methods = [
        Image.Resampling.NEAREST,  # Pixelated
        Image.Resampling.BILINEAR,  # Smooth
        Image.Resampling.BICUBIC   # High-quality
    ]
    method = random.choice(methods)

    return image.resize(new_size, method)
```

Dynamic Scaling Range: Scale factor range expands with realism intensity. Higher intensity = more extreme scaling variations. **Resampling Variety**: Random selection of interpolation methods simulates different scaling algorithms found in real-world software.

add_jpeg_artifacts() - Compression Simulation

```
def add_jpeg_artifacts(self, image):
    # Quality range based on intensity (lower quality = more artifacts)
    min_quality = max(20, 90 - int(self.settings.realism_intensity * 50))
    max_quality = 90
    quality = random.randint(min_quality, max_quality)

    buffer = io.BytesIO()
    image.save(buffer, format='JPEG', quality=quality)
    buffer.seek(0)
    return Image.open(buffer).convert(image.mode)
```

Inverse Quality Relationship: Higher realism intensity produces lower JPEG quality (more artifacts). Quality range: 20-90. **Memory-Efficient Processing**: Uses in-memory buffer for JPEG encode/decode cycle without disk I/O.

apply_subpixel_shift() - Chromatic Aberration

```
def apply_subpixel_shift(self, image):
    shift = random.randint(-1, 1)
    if image.mode in ['RGB', 'RGBA']:
        r, g, b, *a = image.split()
        channels = [
            ImageChops.offset(ch, shift, 0)
            for ch in (r, g, b)
        ]
        if a:
            channels.append(a[0])
        return Image.merge(image.mode, channels)
    return image
```

Color Channel Manipulation: Shifts RGB channels independently by ±1 pixel horizontally, creating color fringing effects. **Alpha Preservation**: Maintains original alpha channel position during channel manipulation.

add_complex_background() - Grid Pattern Generation

```
def add_complex_background(self, image):
    complexity = int(self.settings.realism_intensity * 5)

    bg = Image.new('RGB', image.size, "#FFFFFF")
    draw = ImageDraw.Draw(bg)

    # Grid lines - spacing decreases with complexity
    grid_size = 40 - (complexity * 6)
    if grid_size < 5:
        grid_size = 5

    for i in range(0, image.width, grid_size):
        draw.line([(i, 0), (i, image.height)], fill="#EEEEEE")
    for i in range(0, image.height, grid_size):
        draw.line([(0, i), (image.width, i)], fill="#EEEEEE")

    # Random noise points for higher complexity
    if complexity > 2:
        for _ in range(complexity * 10):
            x = random.randint(0, image.width-1)
            y = random.randint(0, image.height-1)
            draw.point((x, y), fill="#DDDDDD")

    # Composite original image over background
    if image.mode == 'RGBA':
        bg.paste(image, (0, 0), image)
    else:
        bg.paste(image, (0, 0))
    return bg
```

Adaptive Grid System: Grid spacing inversely proportional to realism intensity. Higher intensity = denser grid. **Complexity-Based Effects**: Noise points only added at higher complexity levels (complexity ¿ 2). **Transparency Support**: Uses alpha channel for compositing when available.

7.1.5 Utility Functions

Standalone functions providing specialized operations used throughout the module.

encode_scientific_notation() - Text Normalization

```python
def encode_scientific_notation(text):
    """Convert Unicode superscript notation to plain-text representation"""
    superscript_map = {
        '⁰': '0', '¹': '1', '²': '2', '³': '3', '⁴': '4',
        '⁵': '5', '⁶': '6', '⁷': '7', '⁸': '8', '⁹': '9',
        '⁻': '-', '⁺': '+'
    }

    converted = []
    for char in text:
        converted.append(superscript_map.get(char, char))

    plain_text = ''.join(converted)
    if " × 10" in plain_text:
        return plain_text.replace(" × 10", " × 10^{") + "}"
    return plain_text
```

Unicode Normalization: Converts mathematical superscript characters to plain digits and symbols. **Format Standardization**: Wraps exponent in curly braces for LaTeX/-mathematical notation compatibility.

apply_gamma_distortion() - Non-linear Color Adjustment

```python
def apply_gamma_distortion(image, gamma=random.uniform(1.8, 2.4)):
    """Apply gamma correction to an image in a memory-safe way"""
    if image.mode == 'RGBA':
        r, g, b, a = image.split()
        rgb_image = Image.merge('RGB', (r, g, b))
        arr = np.array(rgb_image) / 255.0
        arr = np.power(arr, gamma)
        arr = (arr * 255).astype(np.uint8)
        distorted = Image.fromarray(arr).convert('RGB')
        r, g, b = distorted.split()
        return Image.merge('RGBA', (r, g, b, a))
    else:
        arr = np.array(image.copy()) / 255.0
        arr = np.power(arr, gamma)
        return Image.fromarray((arr * 255).astype(np.uint8))
```

Gamma Correction: Applies power-law transformation $I_{out} = I_{in}^{\gamma}$ for non-linear brightness adjustment. **Memory Safety**: Uses `copy()` to prevent memory issues with PIL image buffers. **Alpha Preservation**: Special handling for RGBA images - separates, processes RGB, recombines with original alpha.

7.1.6 Execution Entry Point

main() - Command-Line Interface

```python
def main():
    """Entry point for command-line execution"""
    settings = LabelGeneratorSettings()

    # Customize settings for CLI testing
    settings.num_labels = 10
    settings.output_dir = './test_labels_refactored'

    generator = LabelGenerator(settings)
    generator.generate_all_labels()

if __name__ == '__main__':
    main()
```

CLI Usage Pattern: Provides minimal configuration for command-line testing without GUI dependencies. **Import Safety**: The `__main__` guard ensures the module can be imported without automatic execution.

This comprehensive API documentation provides detailed insights into the core module's architecture, implementation strategies, and usage patterns. Each function box includes the actual code implementation along with explanatory notes about design decisions, algorithms, and special considerations.

7.2 GUI Module: label_generator_gui.py

7.2.1 Main Window Class

MainWindow Class

- `__init__()`: Sets up the main application window with all tabs
- `init_ui()`: Creates the complete GUI layout
- `start_generation()`: Initiates label generation in separate thread
- `stop_generation()`: Stops ongoing generation process
- `validate_settings()`: Validates all parameters before generation
- `save_settings() / load_settings()`: Configuration persistence

7.2.2 Tab Classes

Each settings tab extends `SettingsTab` base class:

- `GeneralSettingsTab`: Basic output configuration
- `TextContentSettingsTab`: Text and scientific notation settings
- `UnitsOptionsSettingsTab`: Unit management interface
- `FontStyleSettingsTab`: Typography controls
- `VintageBackgroundSettingsTab`: Visual effects and background settings

- `RotationEffectsSettingsTab`: Rotation configuration
- `SizeResolutionSettingsTab`: Dimension and DPI controls

7.2.3 Support Classes

GenerationThread Class (QThread subclass)

- `run()`: Executes generation in background thread
- `stop()`: Gracefully stops generation

UnitSelectionDialog Class (QDialog subclass)

- Provides interactive unit selection interface

OutputStream Class

- Redirects console output to QTextEdit widget

7.3 Threading Architecture

The GUI uses PyQt6's QThread to prevent interface freezing during generation:

- Main thread: Handles user interaction and UI updates
- Worker thread: Performs CPU-intensive image generation
- Signal-slot connections: Update progress and preview in real-time

7.4 Data Structures

7.4.1 Metadata Format

Each generated label includes the following metadata fields:

`label_id, image_filename, text, rotation_angle, font_family, font_size, font_weight, text_color, background, vintage_applied, vintage_intensity`

7.4.2 Settings Persistence

Settings are saved/loaded as JSON with the following structure:

```json
{
    "num_labels": 2000,
    "output_format": "png",
    "output_dir": "./output",
    "vintage_intensity": 0.7,
    "text_colors": ["#000000", "#333333"],
    "units": ["mg", "mL", "%"],
    "realism_intensity": 0.7,
    ...
}
```

References

[1] Munzner, T. (2014). *Visualization Analysis and Design*. CRC Press.

[2] FDA (2021). *Technical Specifications for Clinical Pharmacology Data*. https://www.fda.gov/media/151733/download

[3] Python Software Foundation. (2023). *Python 3.10 Documentation*. https://docs.python.org/3.10/

[4] Clark, A. (2023). *Pillow (PIL Fork) Documentation*. https://pillow.readthedocs.io/en/stable/

[5] Riverbank Computing. (2023). *PyQt6 Documentation*. https://www.riverbankcomputing.com/static/Docs/PyQt6/

[6] Harris, C. R., et al. (2023). *Array programming with NumPy*. Nature 585, 357–362. https://numpy.org/doc/stable/

[7] Bradski, G. (2023). *OpenCV Documentation*. https://docs.opencv.org/