AUTOMATED GENERATION OF CUSTOMIZED SCIENTIFIC SCATTER PLOTS AND NUMERICAL LABELS

Quentin Vanhaelen

Abstract

This document provides an overview of the technical implementation and capabilities of two complementary python programs. The first program can be used to generate sophisticated scientific scatter plots with various types of properties and visual characteristics. This program is tailored for the generation of large sets of synthetic scientific scatter plots that can be used for training machine learning model designed to extract numerical data and error values from various types of scatter plots commonly found in the scientific literature. The second program is a generator of customized numerical labels designed for creating realistic, diverse label images for data augmentation as well as for the training of machine learning methods designed for the automated extraction of labels from scientific plots. This program relies on advanced image processing techniques to generate thousands of unique label images with precise control over large categories of visual characteristics. These two python programs come with a user-friendly GUI and require limited computational resources. They have been designed to be self-contained and only use a restricted number of common and well-established Python libraries and modules as external dependencies. This makes the creation of the required virtual environment and the deployment of these programs straightforward.

Keywords: scientific scatter plot, error bars, logarithmic scale, vintage effects, numerical labels, synthetic data generation, PyQt6, Matplotlib, NumPy, SciPy, Pillow

Code Availability: The two python programs described herein as well as the instruction manuals can be accessed in the following repositories:

- `https://github.com/qvhaelen/scientific-scatter-plot-generator`
- `https://github.com/qvhaelen/numerical-label-generator`

Contents

1 Introduction **2**

2 Automated Generation of Customized Scientific Scatter Plots **3**

2.1 Introduction 3

2.2 Description of the main functionalities and practical examples 4

2.2.1 Generation of randomly scattered non-overlapping data points 4

2.2.2 Adding vertical error bars 5

2.2.3 Vertical line test 5

2.2.4 Displaying multiple series of data points 6

2.2.5 Generating data series following a predefined mathematical trend 7

2.2.6 Customization of the error bars and plot aesthetics 8

2.2.7 Black and white scatter plots and vintage effects 9

2.2.8 Scatter plots with semi-logarithmic scale 10

2.3 Discussion regarding the implementation of the GUI 12

3 Automated Generation of Customized Numerical Labels **14**

3.1 Introduction 14

3.2 Description of the main functionalities and practical examples 15

3.2.1 Overview of the workflow 15

3.2.2 Adding realism effects to mitigate the domain gap issue between synthetic and real images 17

3.2.3 About the configuration of the vintage effects 18

3.2.4 About the configuration of the image backgrounds 19

3.3 Description of the implementation of the GUI 21

1 Introduction

The extraction of information from scientific charts, such as scatter plots and bar plots, is different from other image recognition problems. Indeed, scientific charts have often a complex structure in the sense that they might contain different elements such as title, axes, legend, labels, and data points which may have different visual characteristics. More importantly, scientific charts are used to represent quantitative data and this means that the information conveyed by scientific charts is not only visual but also numerical and the graphical components (lines, axes, tick marks, legends, etc.) as well as the textual components (chart titles, axis labels, tick values, etc.) must be interpreted consistently to ensure the proper extraction and interpretation of the quantitative information encoded within the scientific charts. This complexity makes the identification of the different elements and the extraction of the information difficult.

In the past, various algorithmic workflows relying on a combination of OCR and deep learning techniques were proposed to extract data from charts of various kinds. For instance, ChartReader[1] presents itself as an automated end-to-end framework trained to extract data from bar plots in scientific literature. Beside the engine designed to extract the data from the chart, ChartReader uses a deep learning-based classifier to determine the chart type of a given chart image. It also uses a double-pass algorithm integrated with an OCR engine to improves text detection by noise reduction. An algorithm to parse legends arranged horizontally, vertically, or in a matrix with different colors, pixel values, and positions in the plot is also integrated in the workflow.

GraphMaster[2] is another automated end-to-end framework to extract data from scientific charts and other figures in scientific literature. One interesting feature is that it allows to separate lines of different color from each other and that it can perform axis detection. Once each line of a given color has been obtained on an isolated chart, the extraction of the data is performed. Text recognition is performed using the same text recognition than the one used by the workflow ChartReader.

Scatteract[3], is an algorithm that uses deep learning techniques and OCR to retrieve the chart coordinates of the data points in scatter plots. Scatteract is a method that detects chart components using a general detection model. It uses deep learning methods to identify the key components of a scatter plot. More precisely, three convolutional neural networks are trained to detect tick marks, tick values and points, respectively. For each tick mark, the closest tick value is found and sent to an OCR tool. Finally, a robust regression is performed to determine the mapping from pixel coordinates to chart coordinates.

It is worth emphasizing that these kinds of deep-learning based models are often trained and optimized to deal with a subset of types of scientific charts and unless they are retrained and fine-tuned with the appropriate training data, they might not perform equally well on other chart types. Furthermore, even if we focus on one type of scientific chart in particular, let us say, scatter plots, a quick overview of the plots found in the literature shows that the design and style of scatter plots can strongly vary. For instance, scatter plots may incorporate different types of special markers. The placement of the legends in charts that have multiple sets of data can vary all over the chart, creating a challenge for automated chart understanding techniques. Differences in font style and text orientation can also present difficulties for these methods.

Interestingly, Scatteract includes a system for generating a large training set of scatter plots, for which the ground truth is known, without requiring any manual labeling. This allows this system to be much more extensible than heuristic methods built around a set of assumptions. The plot distribution can be modified to accommodate new plot aesthetics.

[1] `https://github.com/Cvrane/ChartReader`

[2] `https://github.com/MasterAI-EAM/GraphMaster`

[3] Cliche, M. et al. (2017). Scatteract: Automated extraction of data from scatter plots. arXiv preprint arXiv:1704.06687.

The key challenge is that scatter plots often combine many visual characteristics which are not necessarily seen together on the charts used to build and test many of the existing methods for data extraction. This includes charts which combine line and data points with connecting lines using different patterns, existence of (possibly asymmetric) error bars, horizontal and vertical axes with different scales (for instance in the case of semi-logarithmic scatter plots), additional text located in the vicinity of the scatter plots, presence of multiple set of experimental points depicted with different symbols and colors.

Thus, although methods exist to handle various types of plots with different properties, it is not straightforward to find one existing method able to handle all features and visual characteristics encountered among scatter plots published in the scientific literature. Many methods are restricted in the sense that they handle a subset of the aforementioned characteristics. Thus, it might be necessary to either design an integrated workflow which can handle scatter plots incorporating all the possible visual characteristics or retrain and fine-tune already exiting workflows. In both cases, there is a need to have an access to tools that can be used to generate large sets of customized scatter plots that can be readily used for training and validation of these deep-learning based models.

2 Automated Generation of Customized Scientific Scatter Plots

2.1 Introduction

Automated scatter plot generation plays an important role in benchmarking machine learning systems for quantitative data extraction from scientific figures and recent advances in synthetic data generation have demonstrated the value of domain-specific augmentation for training robust machine learning models[4] In this context, the design of data generation algorithms for specific fields of sciences and techniques requires to pay attention on very specific requirements. For instance, pharmacological applications require specialized handling of measurement uncertainties (e.g., log-normal distributions in drug concentration data[5]) whereas, in scientific fields like pharmacokinetics, scatter plots often depict complex relationships such as dose-response curves, drug concentration-time profiles, or enzyme kinetics, with stringent requirements for error representation, scale fidelity, and visual clarity[6]. Existing visualization libraries[7] lack built-in support for these types of specific requirements. On the other hand, traditional methods for creating training sets rely on manual curation, which is time-consuming and lacks the diversity needed to train robust models. This gap motivates the development of generators of customized scatter plots that emulate domain-specific aesthetics while providing ground-truth data for training machine learning models.

Existing tools, such as the Scatteract system, have demonstrated the value of generating synthetic scatter plots for the training of machine learning models designed for the extraction of data from chart plots. The Scatteract's generator[8] focuses on random point distributions and standard styling variations but lacks features which are important to pharmacological applications such as semi-logarithmic scales and pharmacokinetic mathematical profiles. Indeed, Scatteract's linear/quadratic distributions cannot replicate time-dependent drug metabolism profiles, and the absence of error bars limits realism for experimental data simulation.

This section describes a python program for the generation of scientific scatter plots with a large set of domain-specific capabilities. This includes different pharmacokinetic Profiles (Bi-exponential curves

[4]Shorten, C. et al. (2021). Survey on Image Data Augmentation. *ACM Computing Surveys* 54(6).

[5]Limpert, E. et al. (2001). Log-normal Distributions. *BioScience* 51(5), 341-352.

[6]Gabrielsson, J., Weiner, D. (2012). *Pharmacokinetic and Pharmacodynamic Data Analysis*, 5th ed.

[7]Hunter, J.D. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering* 9(3), 90-95.

[8]See the Scatteract repository: `https://github.com/bloomberg/scatteract`

modeling drug absorption/elimination[9]), scale-aware errors handling including absolute errors for linear scales and multiplicative errors for semi-logarithmic scales with configurable error bars and markers, structured noise with experimental variability through different configurable distributions (normal, uniform, triangular), an automated font and grid management and vintage styling which allows for realistic aging effects to simulate historical lab equipment outputs.

2.2 Description of the main functionalities and practical examples

2.2.1 Generation of randomly scattered non-overlapping data points

In these sections we discuss the step-by-step implementation of the different functionalities of the Python program that generates batches of scientific scatter plots based on user-defined parameters, making possible to quickly generate large datasets that can be used to train and validate machine learning programs designed for automated data extraction from scatter plots extracted from scientific articles. We started with the fundamental functionality that allowed the user to specify the number of images, point markers, minimum and maximum number of points to be randomly positioned in a predefined two dimensional frame with the condition that these points should not overlap with each other. In terms of output, each plot has a X axis and a Y axis with labeled scales (we shall later that this will include not only axis labels but also the legend, the main ticks and intermediate ticks). The original plots were literally scatter plots in the sense that the points were simply randomly placed within the specified coordinate ranges without following any specific mathematical function or pattern, see Fig. 1 for some examples.

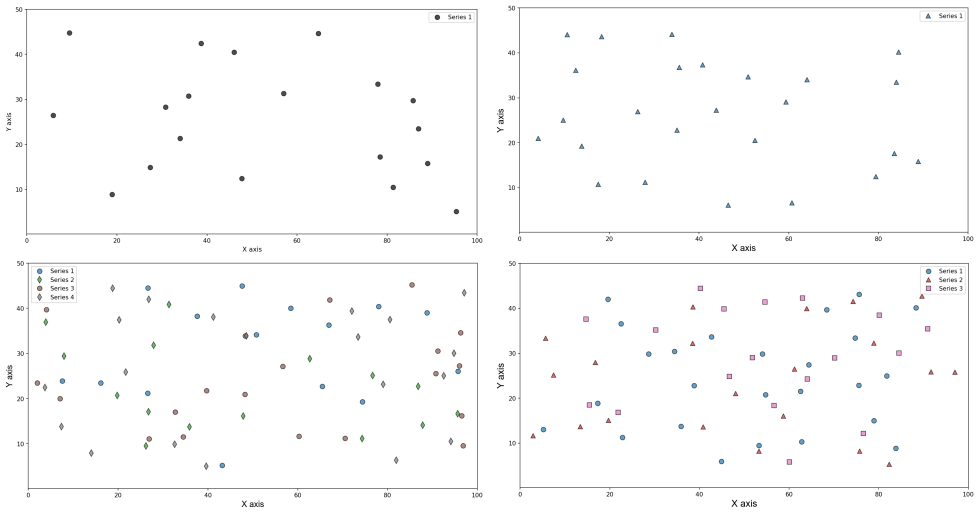

Figure 1: Examples of simple scatter plots that can easily created by the program. (1) the top left and right figures depict a single series of data points which are randomly positioned across the frame following a uniform distribution. (2) The bottom figures contain multiple data series each being represented by a unique combination of color and marker. The figure on the bottom-left contains several points of different series overlapping with each others, the algorithm to generate non-overlapping points was added to avoid this type of situation. The allowed number of data points for each series was set within the range $[15 - 25]$ and the X and Y coordinates were allowed to vary within the range $[0 - 100]$ and $[0 - 50]$, respectively.

One of the first challenge was the fact that one needs to have the possibility to generate points that do not overlap while ensuring that the coordinates of each point remain within the predefined ranges. To achieve this, we needed a method to verify that each new point is at a certain minimum distance from all others. Thus, the program generates for each point the random X and Y coordinates within the predefined

[9]Wagner, J.G. (1975). *Fundamentals of pharmacokinetics*, Drug Intelligence Publ.

ranges, then it checks against all existing points. If a point is too close from the already positioned points, it should be discarded and if the distance is above a threshold (the threshold might depends on the scale), the point is accepted. When a point is discarded, another attempt is made but to avoid infinite loops, the method includes a maximum number of attempts per point. If it is not possible to find a non-overlapping point after the predefined number of attempts, the program would create a plot with fewer points. For simplicity, the first version of the script used a brute-force approach with a reasonable distance threshold and maximum attempts.

Thus, there is one function[10] whose purpose is to ensure that the program generates random non-overlapping points within specified ranges. Another function[11] takes the generated object as an input and creates a single scatter plot with the chosen parameters. These two functions are complemented by the main function to generate all scatter plots. The Python script works with the matplotlib library[12] for the creation of the plot and the python random module[13] to generate the points for the scatter plots.

2.2.2 Adding vertical error bars

A second step was to generate the errors for each Y coordinate point and add the vertical error bars for each point on the plot using the functionalities of the matplotlib library. The original idea was to have vertical error bars, one lower bar and one upper bar, for each point whose size would be randomly (using a uniform distribution) determined within a certain range defined with parameters manually configured by the users[14]. The issues similar to the ones encountered with the need to impose constraint to have non-overlapping points within specified range must also be considered in this context because one wanted to make sure that the error values do not cause the points plus errors to go out of the Y-axis range, see Fig. 2 for some example of scatter plots with error bars.

2.2.3 Vertical line test

When generating the data points, we want to establish conditions to ensure that these points could be interpreted as the points following the curve of a mathematical function. To verify if a curve on a two-dimensional plot represents a mathematical function, one can use the vertical line test[15], which is a fundamental criterion for determining whether a curve defines Y as a function of X. Conceptually, the vertical test is rather simple: A curve represents a function $Y = f(X)$ if and only if every vertical line intersects the curve at most once. In practice, the users could choose to have the random points drawn following a particular distribution while ensuring that these points can be seen as following a function in the mathematical sense of the term. In other word, if one wants to draw a line connecting all generated points, this line could be interpreted as the curve of a mathematical function $Y = f(X)$. This means that one needs to ensure that all X-values are unique. This can be done by generating X-values in a way that they do not repeat. Since X is a continuous range, the program can generate X-values randomly but ensure they are unique by checking against existing points. Checking that the set of points follows this condition requires the implementation of a programmatic version of the vertical line test when the random data points are given. So, once the random data points are generated this function verifies if the vertical line test is successful or not.

[10] This function is called `generate_non_overlapping_points(num_points)` in the script.
[11] This function is named `create_scatter_plot` in the script.
[12] `https://matplotlib.org/`
[13] `https://docs.python.org/3/library/random.html`
[14] the creation of these errors was to be handled in the function `create_scatter_plot())` by a custom function
[15] `https://mathworld.wolfram.com/VerticalLineTest.html`

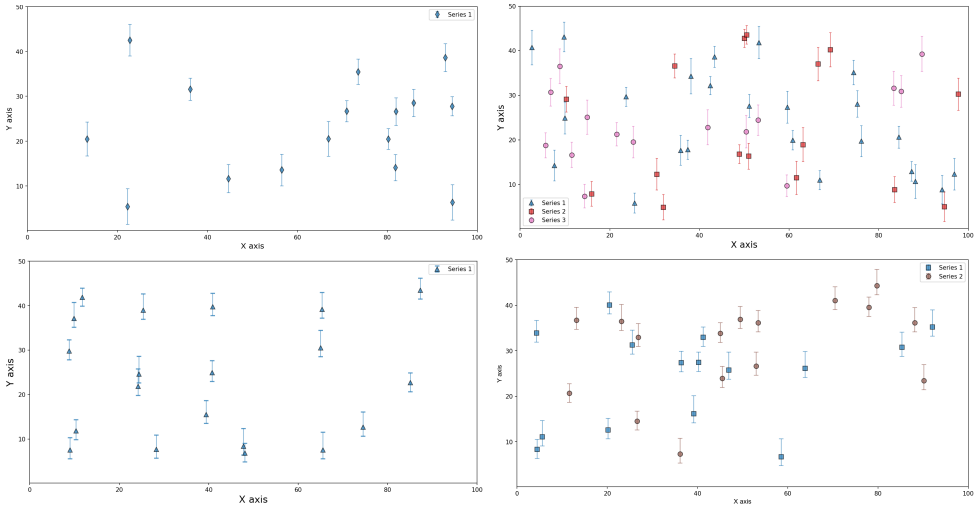

Figure 2: Examples of simple scatter plots that can easily created by the program. The configuration settings are the same than for the scatter plots depicted in Fig. 1, with the addition of the error bars. The two figures on the top show examples of scatter plots with symmetric error bars while the two scatter plots at the bottom are examples with asymmetric error bars.

2.2.4 Displaying multiple series of data points

A natural extension is to allow for multiple data series to be displayed. This corresponds to the situations where the data points from multiple experimental settings are displayed together to show how the behavior of the underlying system change in function of some parameters or because of different initial conditions for instance.

The presence of multiple data series in the same plot create additional difficulties for automated data extraction because the program should not only be able to identify all data points precisely but also to be able to distinguish the different data series by using for instance the fact that each data series can be represented using a different symbol and/or color scheme. This motivate the extension of the capabilities of the program to ensure that it could handle multiple data series. One needs to add parameters so the users can determine how many data series should be represented, then functionalities need to be added so the program can draw each data series using a specific color and/or a specific symbol.

One also needs to consider the existence of the error bars, the symbols and color schemes used should be consistent across the different data series. Algorithms already implemented such as the vertical line test should work for each data series independently if drawing data series following some function patterns is requested by the user, in that case this constraint would apply to all data series on a given plot.

To increase the diversity of the scatter plots generated by the program, we can easily implement the followings. First, in the original setting, the number of data series is the same for all images. Rather than having one parameter fixing the number of data series for all plots, one replaces it by two parameters, one defining the minimum and one defining the maximum number of data series authorized for each plot. The program can then randomly select the number of data series to be generated for each plot. Additionally, one can have the script randomly selecting what font and size to use for the text elements (label, tick marks, etc.) of the X and Y axis with the requirement that the font and size for the text elements of the two axis should be chosen to be the same for the two axis. In the same vein, the program is also adapted to randomly select a font and size for the title of the plot and for the legend.[16]

[16] these options were implemented using the module `font_manager` from the matplotlib library.

In terms of outputs, the program generate data files with the numerical values of the data points generated for each plot and create a separate file for each data series appearing on each plot. When the program generates the error bars, a function also generates the list of values of the error associated with each data point of each series in each plot. Each of these error files has two columns. The first column contains the upper error values and the second one contains the lower error value. When symmetric error bars are generated, the two columns contain the same numbers.[17]

2.2.5 Generating data series following a predefined mathematical trend

Another expansion of the capabilities of the program is to implement different data distributions and mathematical trends with an emphasis on data distributions mimicking time series data. For now, the program only generates scattered points with no inherent pattern with equal probability across the X and Y ranges. To create time-series-like patterns while retaining randomness, one needs to define a base mathematical function and add controlled noise to the Y-values. From a practical perspective, the standard random Python module and the numpy.random library gives access to the Uniform distribution, the Gaussian/normal distribution[18] and the triangular distribution[19]. The numpy.random library also provides an access to the Normal, exponential (Rapid growth and decay pattern), Logistic and polynomial distributions. Additionally, the python library scipy.stats[20] has specialized distributions such as Beta, Gamma, Weibull, Poisson, Binomial and other custom distributions. Depending on the types of processes to be modeled, some distributions might be more adapted than others.

In addition to the mathematical trend, it is possible to consider different approaches to model the statistical noises. One would typically use additive noise which model constant variance, multiplicative noise which models variance growing with the trend or Gaussian noise which are suitable for natural variation simulation. In practice, adding a simple additive noise would generate synthetic data points with a clear directional pattern (upward/downward trend). It would also ensure that the points are clustered around the predefined theoretical curve and enforce a controlled deviation from the perfect mathematical behavior. All these temporal patterns are assumed to represent the dynamics of non-negative quantities over time so among the constraints to be enforced when generating these data points, it is important to ensure that all point coordinates remain positive. We also need all functions generating mathematical trends to be designed so that the Y coordinate of the data points approaches a steady state as X increases while remaining non-negative.

Among the mathematical trends implemented, some of particular interest are as follows. The mathematical functions following a linear saturated pattern are good to model dose-response relationship, a function following the Logistic trend with its sigmoidal pattern is well-suited to represent population growth while an exponential shape is appropriate to model drug clearance kinetics. We also considered the asymmetric Bell curve with steady state which is a custom trend combining growth and decay. In the script, there are functions to implement these different trends and another function which generates the statistical noise, specifically encoded to prevent negative Y-values. It is important to notice that the choice of of the mathematical functions and the noise distribution functions is the same for all data point series of a given scatter plot (but the choice will randomly vary from one plot to another).

To increase the diversity of the shapes of the series of data points, the parameters characterizing the mathematical trends are allowed to take different values (within predefined reasonable ranges) from one

[17] It must be said that if the `parameter add_error_bars` is set to false, there is no error bar generated and no output files for the error bars.

[18] `https://numpy.org/doc/2.1/reference/random/generated/numpy.random.normal.html`

[19] `https://numpy.org/doc/2.1/reference/random/generated/numpy.random.triangular.html`

[20] `https://docs.scipy.org/doc/scipy/reference/stats.html`

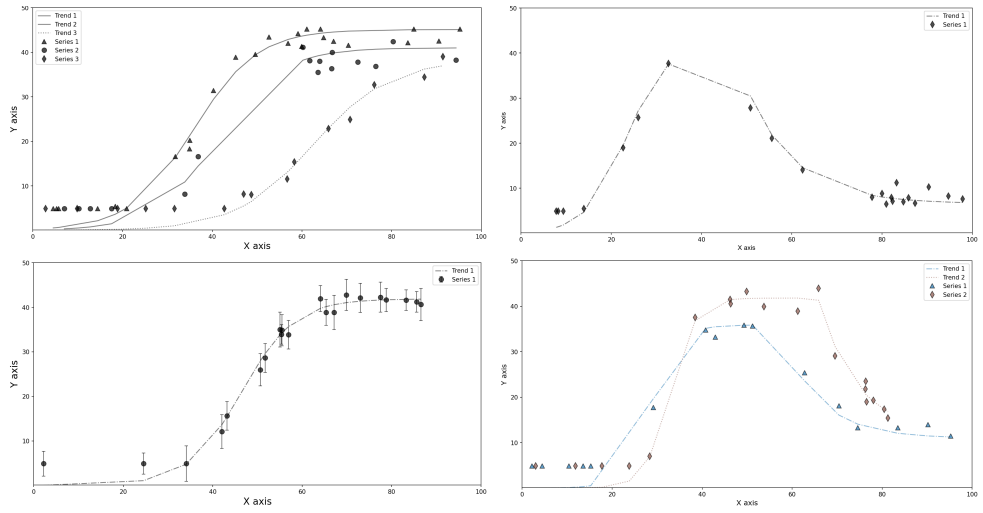

Figure 3: Examples of scatter plots whose data series follow predefined mathematical patterns. In these examples, the main mathematical curve is represented using various types of styles. When the data series are represented in color, the program chooses a suitable combination of color and line styles to ensure that each mathematical curve is clearly distinguishable. When curves are represented, the corresponding line style is also listed in the legend together with the combination of markers and colors used to represent the data points.

curve to another one and similarly, the parameters of functions modeling the noise distributions can also randomly vary from one data point series to another one to ensure that the different data point series of a given scatter plot are not strictly identical. There are functions within the script which verify that when varying the parameters, the resulting curve remain realistic and within the boundaries of the plot, see Fig. 3 for some examples.

2.2.6 Customization of the error bars and plot aesthetics

We also looked at what can be done at the level of the error bars in the sense that we wanted to have more advanced features that would include support for asymmetric error bars and different error types, see Fig. 2 for some simple examples of scatter plots with asymmetric error bars. This can be done using the implementation of the matplotlib's errorbar function[21] which can handle asymmetric errors.

There are different approaches to model the errors that can be implemented like percentage errors, logarithmic, or even error bands. These variations can be interesting for more nuanced data representation. By default, the program generates absolute errors only, but one can also consider logarithmic errors which are relevant for logarithmic scale plots (when the error is proportional to the magnitude).

Regarding the graphical representation of the error bars, it is feasible to customize the length and the style of the error bars (using arrows, circles, etc.). Asymmetric error bars can be used to represent confidence intervals with different probabilities above and below the measurements. In pharmacokinetics, asymmetric errors are common because drug concentration measurements might have different variability above and below the mean. It is important to maintain coherence within each plot, especially with plots with more than one data series. In practice if one data series of a scatter plot has symmetric error bars then this should also be the case for all data series of this plot because it would be strange if a data series has asymmetric bars while another data series on this same plot has only symmetric bars.

[21] `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.errorbar.html`

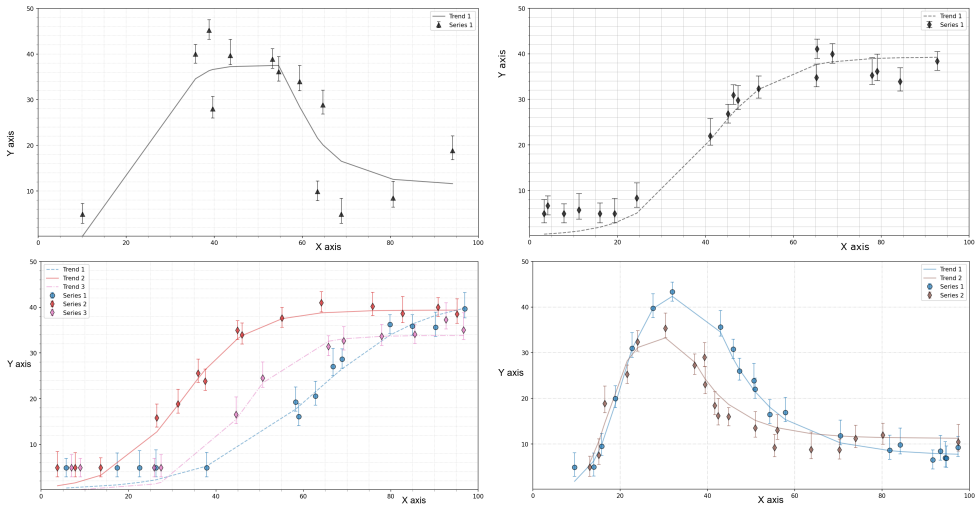

Figure 4: Examples of scatter plots whose data series follow predefined mathematical patterns. In these examples, the main mathematical curve is represented using various types of styles. It is also possible to see the different grids used. It can also be seen that the axis labels are displayed in different position along their corresponding axis. When curves are represented, the corresponding line style is also listed in the legend together with the combination of markers and colors used to represent the data points.

In terms of plot aesthetics, we wanted to provide the users with more control over the sizes of the markers used to represent the data points and the styles of the grid that are displayed in the plot. For the grid styles it is possible to allow the users to enable or disable the presence of major and minor grids and change the line styles. This is done by adding configuration parameters to control the grid visibility and the different styles of lines. Under some circumstances, one can consider dashed or dotted grids which can reduce visual clutter. In the current implementation, the markers can have a fixed size but it is possible to have markers with variable size where the size of the markers would be scaled according to the data values. It is worth emphasizing that when the program generates a scatter plot with multiple data series in black and white, the use of various types of markers is necessary to distinguish the different data series and in that specific case, the number of data series drawn in a given plot cannot exceed the number of different point markers available.

The grid styles include different line styles, colors, or densities. There are options for adding minor grid lines or custom grid intervals. It is important to handle the randomization of these graphical features while ensuring the absence of overlaps and maintain the clarity of the scatter plots to avoid overcrowded plots with too many annotations or complex error representations. In particular, asymmetric errors require adjusting the data generation to produce separate upper and lower errors and the generation of markers with variable sizes would involve generating a size parameter for each series and applying it during the plotting. While the users have the option to enable the customization, it is the program that, during the generation of the different scatter plots, randomly select the exact type of grids for each plot, see Fig 4 for some examples.

2.2.7 Black and white scatter plots and vintage effects

We also considered the possibility to add effects to black and white scatter plots to make them look like plots scanned from old, degraded scientific articles. Old articles often have lower print quality, so adding some noise or blur could replicate that. Also, the paper texture might have yellowed or degraded over time, so simulating that texture can also be considered.

The implementation of these effects required the python library PIL/Pillow[22]. The effects are optional and controlled by a set of parameters managed by the user. The addition of these vintage effects do not affect other existing functionalities. In practice, the addition of these vintage effects is done as follows. the program first generates the scatter plot as usual and save it as a high-resolution image, then the program loads the high-resolution image to apply the vintage effects selected by the user and the program finally saves the modified image. There are various effects that can be considered:

1. Scan Artifacts: Adding scan lines or JPEG compression artifacts. It is done by overlaying horizontal lines with reduced opacity or using noise patterns. It is feasible with PIL by drawing lines on the image.
2. Paper Texture: Applying a background texture that looks like old paper. It is done using a semi-transparent image overlay or generating a procedural texture.
3. Ink Bleed: Making lines slightly blurry or imperfect. Gaussian blur on the plot elements can be appropriate, but matplotlib does not support this so one needs to use PIL and post-process the original image.
4. Halftone Patterns: Old prints used halftone dots. Converting gray-scale to halftone patterns is possible but it requires dithering algorithms[23], which are available in image libraries.
5. Color Shifting: Making the image slightly yellowish or sepia tone. This requires to adjust the color channels to add a yellowish tint. PIL can do this with the color matrix transformations. The most effective way to shift colors in Python with the Pillow (PIL) library is to convert the image from the RGB color space to the HSV (Hue, Saturation, Value) color space, manipulate the hue channel, and then convert it back to RGB.
6. Random Noise: Adding graininess to simulate film grain or print noise. PIL can apply this with pixel-wise manipulations.

Some of these effects are implemented in a separate function[24]. An additional way to customize the scatter plots is to allow randomized figure size and randomized DPI. These options are also available and can be controlled by the user with the parameter `customized_size_resolution`. Some examples of scatter plots generated with vintage effects enabled are depicted in Fig. 5.

2.2.8 Scatter plots with semi-logarithmic scale

Another useful addition is to consider the possibility to generate scatter plots with semi-logarithmic scale (called semilog scale for short: Y-axis follows a logarithmic scale while the X-axis follows a linear scale). In the program, the user has access to a Boolean parameter to enable this option. When this Boolean parameter is set to true, the program can randomly choose to use a semi-logarithmic scale for the generation of the scatter plots.

To be able to handle this option correctly, the function controlling the generation of the data points and the function computing the upper and lower errors and the one drawing the corresponding error bars need

[22] `https://pypi.org/project/pillow/`

[23] Dithering algorithms are image processing techniques used to simulate higher color depth and smoother gradients by arranging limited colors in specific patterns. They create the illusion of more shades (or grayscale) using only a few colors by creating a speckled or grainier pattern that the human eye mixes

[24] The function is called `apply_vintage_effects` in the script

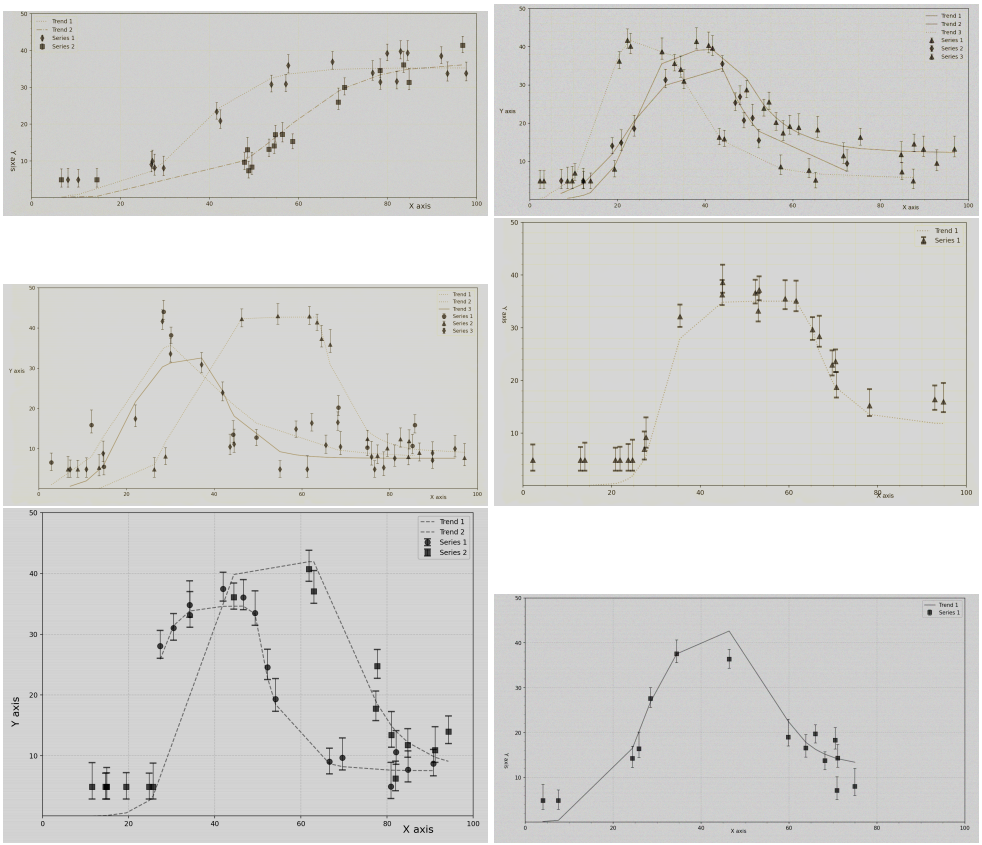

Figure 5: Examples of scatter plots whose data series follow predefined mathematical patterns. In these examples, the main mathematical curve is represented using various types of styles. It is also possible to see the different grids used and the fact that the axis labels are displayed in different position along their corresponding axis. These scatter plots show what can be created when enabling the vintage effects fo the generation of images in black and white. For the creation of these scatter plots, the customization of the position and orientation of the axis labels was enabled as well as the customization of the resolution and size of the images.

to be adapted for logarithmic scales. For instance, when generating the Y values (especially for trends like exponential decay), the program needs to ensure that the Y coordinates are (strictly) positive. For the error bars, since the logarithmic scale is multiplicative, the errors need to be handled differently than in the linear case[25].

In practice, the existing function for the error generation needs to be adjusted to be compatible with logarithmic scales by converting the additive errors to multiplicative factors, because in the case of logarithmic scales error bars should be multiplicative rather than additive[26]. Thus, in the error generation function, when the semi-logarithmic scales are considered, the errors should be generated as multiplicative relative values rather than absolute values, so the error bars scale appropriately with the logarithmic axis.

Another aspect to take into account is that the module computing the errors uses an exponential function to generate the errors in the linear case, which makes them scale with the magnitude of the data point coordinates. When generating error bars for data points in the semi-logarithmic scale, one needs to

[25] Stuve, E. M. (2004). Estimating and plotting logarithmic error bars. Retrieved September, 19, 2013.

[26] The reason is that on a logarithmic axis, distances represent ratios, not differences. An additive error bar applied to a log plot will appear skewed and misleading, whereas multiplicative error bars correctly reflect the proportional nature of variability across multiple orders of magnitude

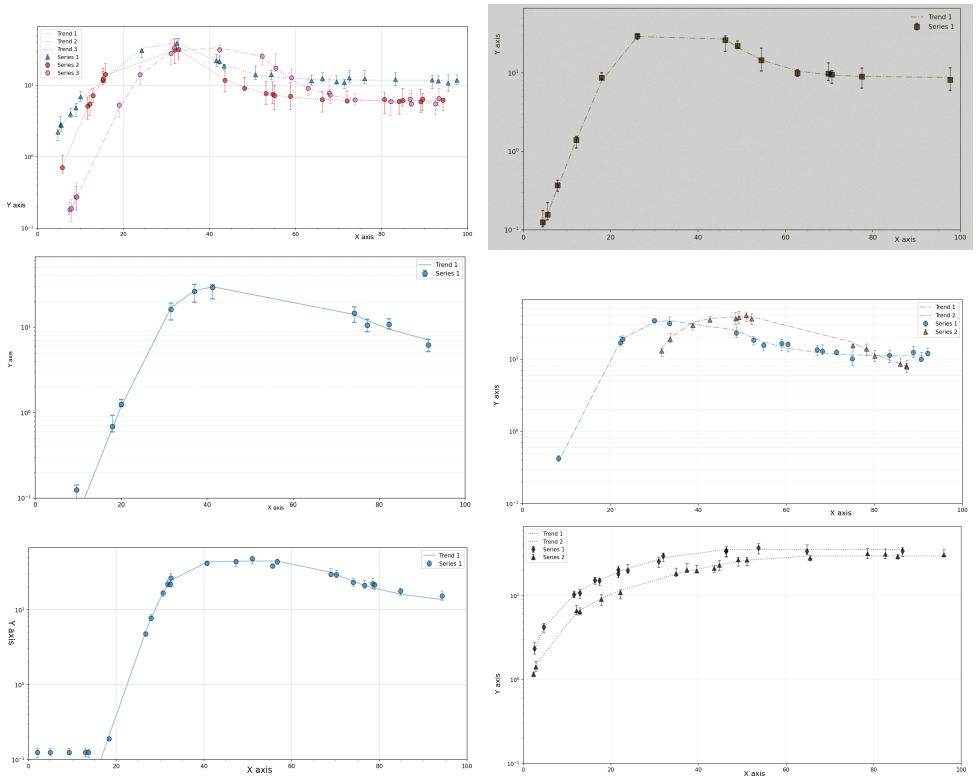

Figure 6: Examples of scatter plots whose data series follow predefined mathematical patterns and whose Y coordinates follow the logarithmic scale. In these examples, the main mathematical curve is represented using various types of styles. It is also possible to see the different grids used and the fact that the axis labels are displayed in different position along their corresponding axis. These scatter plots show what can be created when enabling the vintage effects for the generation of images in black and white.

adjust the error generation to use relative percentages. For example, a 20% error in log scale becomes a multiplier of 1.2, ensuring errors are consistent across different data magnitudes.

The buffer calculation for the Y-axis, the method used to handle data points near the borders of the plot, is also impacted when generating plots in the semi-logarithmic scale. For linear scales, the buffer is additive (buffers are added by subtracting a fixed value), but the buffer calculation for the Y-axis should be multiplicative to correctly handle logarithmic scales. Instead of subtracting a fixed value, one needs to multiply or divide the limits by a buffer factor to keep points within the visible bounds. To handle these different scenarios, the program contains separate functions to compute the buffers for the linear and the logarithmic scales. See Fig. 6 for some examples of scatter plots in semi-logarithmic scale.

2.3 Discussion regarding the implementation of the GUI

The original python program controlling the core of the functionalities has a lot of configuration options, and we want the users to be able to configure these configuration settings through a GUI instead of manually modifying the options within the code. The idea was to design the GUI around the original code so it would act as a wrapper. For user convenience, we wanted the GUI to be able to update the users about the status of the generation in real-time and also display the scatter plots as they are generated. We have assembled a GUI around the original python program that generates the scientific scatter plots using the python library PyQt[27]. PyQt was chosen for this task because it is a robust python framework for

[27] `https://pypi.org/project/PyQt6/`

designing and implementing self-contained desktop applications. The approach described herein can be considered as "minimalist" in the sense that the implementation of the GUI only relies on a restricted number of python libraries, that is, mainly PyQt6 for the GUI itself with modules from Matplotlib[28], NumPy[29], SciPy[30], and Pillow[31] which are python libraries already used by the original script. Beside these main libraries, there is no need to use additional external dependencies. This makes the creation of the virtual environment easy and the deployment and installation of the program itself is straightforward.

The implementation of the GUI is a step-by-step process. To create a prototype of the GUI and design its layout, we started with the identification and extraction of all the configurable parameters, that is, all variables that the user should have access to. These include numeric inputs, dropdowns (markers, trend types), checkboxes, and file paths. In total, the original script contains around 40 configuration options that must be logically integrated into different groups and made accessible to the user through the panels and menus of the GUI. Then one needs to build a mapping between these parameters that should be manually configured by the users and the corresponding widget suitable for each of them.

These parameters must be organized into the interface and to that end, the main window is divided into sections to group the related settings into panels and configurable menus. We have designed a GUI made of multiple pages (also called panels) which are to be incorporated within the main windows. The separate panels include a panel entitled "General Settings" where the user can configure the number of images and number of series per image to be generated. The user can also enable the vintage effects and customization options. A second panel is called "Representation of the Data Points". There, the users can select the makers to be used to represent the data points, enable and configure the error bars and select the function to model the noise distribution. The third panel is called "Axis, Scale and Grid Configuration" where it is possible to configure the axis and the associated scales as well as the font and color to be used. It is on this panel that the users can configure the grid and customize the positioning and orientation of the labels and ticks. The last panel is named "Data Generation Methods" and that is where the users can select the method for generating the data points.

The user can navigate from one page to another one to configure the parameters and the options that can be customized in many ways. A point to consider is that many options in the original script are made of two separated parts. Firstly, there is a Boolean parameter to enable/disable the option itself and secondly, there is a vector containing the list of possibilities available for this option that can be randomly selected by the program. The user should ideally has access to both as this not only allows to enable or disable a given option but also to control, or modify, what are the possibilities that could be randomly selected.

We have designed a multiple windows GUI but beside the main window we wanted to keep the number of additional pop-up windows restricted to warning messages and confirmation required from the user before executing a task which is irreversible. This includes shutting down the application, starting image generation, or error message displayed if the user has selected inappropriate/incompatible settings. This could be, for instance, if the directory where the generated images should be created happens to be incorrect, or cannot be found. When the user clicks the button to begin to generate the scatter plots, it initiates a process which first validates the chosen settings (for instance, checking that the output directory exists, that the options are selected within the appropriate predefined ranges and so on) and if some of the selected options are invalid, the program will show a warning in a message box, and on the other hand, if the selected options are valid, the program will display a confirmation dialog box and upon confirmation of the user, it starts the generation in a separate thread.

[28] `https://matplotlib.org/`
[29] `https://numpy.org/`
[30] `https://pypi.org/project/scipy/`
[31] `https://pypi.org/project/pillow/`

From a more technical perspective, we want to integrate the original script within the one coding the GUI itself, considered in this context as a wrapper embedding the complete program. To this end, the original script containing the core functionalities needs to be refactored to accept parameters from the GUI instead of using the global variables. This requires the creation of a new configuration class and the extraction of the core of the image generation logic into a separate class that can be called from the GUI. Once the configuration parameters are correctly encapsulated into this new class, the core functions of the original script can be adapted to accept these parameters as arguments instead of relying on global variables. Once the script is refactored, it is possible to properly connect the input widgets to the configuration parameters and implement the buttons controlling the generation and allowing the users to browse the file directory as well as signals and slots to update the GUI when the images are generated or errors occur.

Additionally, we have incorporated within the GUI a system handling a preview area and progress indicators so the users can monitor in real-time the progress and the type of images being generated. The display of the images in real-time requires the script to emit signals or provide callbacks when an image is saved. The GUI loads the image, updates the panel designed for the preview accordingly and displays the generated scatter plots immediately as they are created. We have also found useful to add a progress bar and a menu for the messages displayed by the console to provide the users with continuous real-time information about the status of the generation. To that end, the existing print statements in the original script are redirected to a QTextEdit in the GUI.

Another important aspects to take into account from the viewpoint of the performance of the program when implementing the GUI is that the generation of large batches of scatter plots can take a significant amount of time. Thus, in order to prevent the GUI from freezing, the program makes use QThread to ensure that the main GUI thread remains responsive, and that the user can monitor the progress being made and terminate the generation whenever deemed necessary. To allow this, the GUI is designed so that the area where the images are being displayed in real time and the buttons used to start or stop the generation of the images are placed on the main page of the GUI which always visible to the user whatever other panels and options are displayed. For user convenience, the GUI includes saving and loading presets which allow the users to save their configurations as presets for future use. This is implemented by serializing the settings to a JSON file and loading them back when needed.

3 Automated Generation of Customized Numerical Labels

3.1 Introduction

The automated label generator described in this section is a synthetic data generation engine which can 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 vintage effects and distortion.

An important feature of this program lies in its focus on automated variation and controlled realism. While the aforementioned label software gives users full design control over a single output, this generator relinquishes direct control over individual labels to 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).

3.2 Description of the main functionalities and practical examples

3.2.1 Overview of the workflow

The label generation follows a sequential workflow which can be summarized as follows:

1. Text Generation: The program randomly selects the text to be displayed using the text options available. It also takes into account the appropriate scientific notation if this option is selected.
2. Font selection: This is the first step to customize the label where the program randomly selects the visual characteristics of the text among a large choice of enabled font families[32], sizes, and weights.
3. Canvas creation: The program creates the frame where the text is to be added. To take into account that the size of the text can vary from one label to another one, the program uses a dynamic canvas sizing approach based on text dimensions and padding.
4. Background generation: The program designs the background of the image. It generates the color in the HSV space while guaranteeing a minimum level of brightness.
5. Text Rendering: This step ensures that the text is correctly positioned in the middle of the frame. The method ensures precise centering with font metric calculations.
6. Rotation: When the option to rotate the text is selected by the user, the program rotates the label within the frame. This might requires to expand the canvas while preserving the background properties and visual characteristics.
7. Visual Effects: This is the last step for the customization of the image. Depending on the selections made by the user, the program proceeds with a sequential application of vintage and realism effects.
8. Format Conversion: The program ensures the proper conversion of the generated label image to the appropriate output format required by the user with proper DPI embedding.

The program is structured so that all functions are included in one of the three following classes. The first one is named the `LabelGeneratorSettings` class which contains the main settings container with approximately 30 parameters which are classified in different categories: General Settings, Resolution/Sizing, Text Content, Font/Text Styling, Rotation, Visual Effects, Layout. The second class is called the `LabelGenerator` class and it contains the core of the generation logic with various methods including the principal ones which are `generate_label_text()` which creates the text with scientific notation, `create_label_image()` which creates the single label image, `generate_all_labels()` which is the main generation loop and the function `save_metadata()` which saves the metadata in

[32]For the fonts, the system fonts are used

`.csv` and `.txt` formats. The third class is the `RealismEnhancer` class. It applies the image distortions. There is also the function called `encode_scientific_notation()` which is used for text formatting. The reader is invited to consult the technical documentation for detailed information about these functions and how they operate. In what follows, we provide a step-by-step description of how the program proceeds to generate customized label images. We also provide some examples of images of customized labels that can be generated using this program.

For each customized label image, a function determines the text content, randomly selects one color, determines the rotation angle as described, and generate the images (using the PIL Python library[33] for image generation and rotation) following a a multi-step process. It first creates a transparent background image of sufficient size (it estimates the size of the text but there is a method that can be used to expand the size as needed), the method also uses PIL to draw the text with the chosen font, size, and color. The label content generation procedure supports numerical values with optional scientific notation, random unit attachment with separators and predefined text options with various options for customization. The system also provides the user with typography options such as a system font detection, font size variation around the base size, font weight options (for instance normal, bold, italic) as well as text and background color selection.

There is a background color selection to ensure that only light backgrounds are used and additional settings are available to configure the maximum darkness of the background. In practice, the program generates light backgrounds with RGB values greater or equal to 200 with the option to create a variety of light colors (white, light grays, pastels with low saturation). To guarantee that light backgrounds are properly generated, the program uses a dynamic light color generation and it uses the HSV color space to ensure that brightness is also always chosen to larger or equal to the parameter `min_background_brightness` and it calculates the luminance[34] using the perceptual formula: $0.299\,R + 0.587\,G + 0.114\,B$. This relationship is known as the Legacy NTSC Formula, other formulations can be used as well. Finally, it returns the colors meeting the minimum brightness threshold.

The next step is to rotate the image by the chosen angle. The rotation is performed within a fixed square frame[35] and occurs around the center. The rotation operation is enabled and controlled with the parameter `rotation_allowed` which acts as a Boolean switch. When enabled, the selection of the rotation angle is done from the predefined list of allowed angles or using custom angles which are chosen as multiples of 5 degrees, in the $[0 - 90]$ degrees range. The fact that the rotation is made within a fixed square frame requires several steps. Firstly, the program determines a fixed square size (max of width/height with padding), then it creates the base image at this fixed size and creates a temporary image for the text (without rotation). After, it rotates the text image and pastes it onto the center of the base image. When the image creation uses a fixed square canvas and rotates only the text, not the entire image while maintaining consistent frame dimensions regardless of rotation, it ensures that the frame remains consistent regardless of rotation.

The last step of the image generation is applying vintage and realism effects. The program applies vintage effects only if they are enabled by the user. Vintage effects include paper texture overlay, sepia tone filters, Gaussian blur, noise effects, brightness adjustment and configurable intensity. The program also handles the customization of the size and resolution of the generated label images. The dimension can be managed using the parameters `min_width` and `max_width` which defines the range of allowed

[33] `https://pypi.org/project/pillow/`

[34] Luminance, often referred to as perceptual brightness or relative luminance is calculated by applying non-linear, weighted coefficients to RGB color values to account for the human eye's higher sensitivity to green and lower sensitivity to blue

[35] When labels are rotated with the complete rectangular frame being rotated, this might cause an issue when using these images to train an image recognition and extraction machine learning model to recognize rotated labels because it could associate rotated labels exclusively with the existence of a rotated frame. Thus here the rotation is made within a fixed frame.

values for the width of the images. Additionally, the parameters `min_height` and `max_height` are used to calibrate the range of allowed values for the height of the images. The system ensures that the aspect ratio is maintained during resizing. The resolution can also be adapted. The parameters `min_dpi` and `max_dpi` control the DPI range when the option for customized resolution is enabled and the parameter called `fixed_dpi` maintains the DPI at a unique default value when the option for customization is disabled.

After the completion of these steps, the program saves the images[36] in the output directory defined by the user and adds the record in metadata including `image_filename`, `text`, `rotation_angle`, `font_name`, `font_size`, `color`, `vintage_intensity` (if applied), `unit` (if any), and any other parameters. The final outputs also include metadata in `.csv` format.

Figure 7: Several examples of generated labels. It provides an overview of the different properties that can be customized to obtain large sets of diverse numerical labels.

3.2.2 Adding realism effects to mitigate the domain gap issue between synthetic and real images

An important aspect of the generation of the synthetic data that should be taken into account is the phenomenon of "domain gap" between the synthetic and real data which leads to the following common

[36]The program handles various types of format including the most common ones like JPEG and PNG. The program includes functionalities for automatic filename numbering.

issue in machine learning, that is, the model trained on synthetic data does not generalize well to real data. There are many differences between the synthetic label images and the typical real images extracted from articles that might cause a significant decrease in performance. Fortunately, many of these differences can be mitigated using appropriate algorithmic solutions during the generation of the label images. Here we list several of them that were taken into account when designing the program.

A first aspect is related to the resolution and quality in the sense that synthetic images might have consistent high resolution, while real screenshots extracted from scientific publications could be of lower resolution or contain various compression artifacts. One can introduce downsampling, compression artifacts, and varying resolutions in synthetic data to solve this type of issue. Compression Artifacts are needed because the synthetic data are saved in the form of clean PNG with no compression while real screenshots contain JPEG compression artifacts, color banding. Thus, it is necessary to add a function to simulate compression artifacts. Another aspect to consider is anti-aliasing and rendering differences between synthetic label images which has perfect font rendering and the images extracted from real PDFs which contain subpixel rendering, font hinting variations. To handle this, one also adds subpixel shift simulation. There is also the need to add a feature to simulate the gamma correction to manage the discrepancies related to color space and gamma differences. These are caused by the fact that the synthetic images contain linear RGB and perfect colors while the real screenshots of images often include sRGB gamma and color profile conversions. Also, the background complexity differs between synthetic images which has uniform, often simple, backgrounds and real images which might include features such as grid lines, plot elements, noise behind labels, textures, etc. To simulate this type of effects, it is possible to add realistic background noise to simulate more complex backgrounds for the synthetic label images.

A second aspect to consider is the font variability. The generator of label images uses a limited set of fonts. Real labels might use fonts not seen during training of the machine learning models simply because these fonts are not present in the dataset of synthetic images. A solution is to expand the font library or use some kind of font augmentation techniques.

Another aspect relates to the label positioning and rotation. Often, labels in real images might have more diverse rotations and positions than the generated labels. A straightforward solution is to increase the variability in rotation and positioning during training.

One should also consider the effects of lighting and color variations because real images often have lighting variations, shadows, and color shifts that are absent in synthetic label images. To take this into account, one can add brightness, and contrast adjustments to the synthetic images. Another point to consider relates to the noise and distortions. Images extracted from articles might have noise, blur, or distortions from the screenshot process. To include these types of effects, one can add noise, blur, and distortion filters to the synthetic images. Finally, there is the existence of artifacts and occlusions. Real labels might be partially occluded or have artifacts (like ink smudges in scanned documents). A solution is to simulate occlusions and artifacts in the synthetic images.

Several of these features are integrated in the third class of the program called `RealismEnhancer`. This class contains the functions to apply various image distortion techniques and they can be enabled through the GUI using the parameter called `realism_intensity` located in the panel named "Vintage Background effects".

3.2.3 About the configuration of the vintage effects

It is important to emphasize the differences between the vintage effects, the intensity of these effects and what they actually do on the labels and images. Furthermore, there are also the realism effects which controls a different set of effects applied on the labels and images.

The GUI provides the user with an access to two different sets of vintage effects implemented in the code [37]. These are the vintage effects that are applied to the entire label image (background and text) as a post-processing step. They are controlled by the parameter `vintage_effect_prob` (probability that the vintage effects are applied to a given image) and the parameter `vintage_intensity`. This second parameter determines how strong the vintage effect is in the sense that it controls the overall intensity multiplier for all vintage effects applied on the image. Thus, these two parameters provide a general control for applying vintage effects, i.e., whether to apply these effects and how intense they should be. On the same panel, there is an entry to define the texture file, `old_paper.png` which is used in this vintage effect to add a paper-like texture.

A second set of parameters related to the vintage effects allows the user to fine-tune individual components which make the vintage effects. These are the parameters that control the amount of noise and blur in the vintage effect. More precisely, the first of these parameters is called `noise_intensity` and it can be used to determine how much random noise and grain should be added. The second parameter is named `blur_intensity` and it allows the user to configure how much Gaussian blur should be applied on the image. These are specific parameters that are used within the vintage effects function to specifically control the noise and blur applied during the vintage effect process.

In this panel, there is also a parameter called `realism_intensity` which controls a set of fundamentally different effects (JPEG artifacts, subpixel shifts, gamma distortion, etc.) which are implemented within the `RealismEnhancer` class and are discussed in more details in another section. These are separate effects from the vintage effects described above and they can be applied regardless of whether the vintage effects are enabled.

3.2.4 About the configuration of the image backgrounds

There are two settings controlling the background of the generated label images and the purpose of this section is to detail how they operate and what impact they have on the generated label images.

The first setting is the parameter called `min_background_brightness`, whose default value is set to 0.8. It controls the minimum brightness of the randomly generated background colors of the label. This parameter is used in the function named `generate_light_background` located in the `LabelGeneratorSettings` class. The background color is generated in the HSV color space, and this setting ensures that the value (brightness) is always within the range from 0.0 (black) to 1.0 (white). So this parameter controls how light the background color will be. Selecting a higher values, i.e., closer to 1.0, means that the background will be lighter, since the value 1.0 corresponds to the maximum allowed brightness. Notice that this parameter is only relevant when the background is not transparent. In practice, the parameter `min_background_brightness` set to 0.9 will create very light pastels (almost white), if `min_background_brightness` is set to 0.5, it creates medium-light colors and if `min_background_brightness` is set to 0.2, it creates darker colors (but these colors are still visible).

The second setting is controlled by the parameter named `transparent_bg_prob` which is set to 0 by default. This parameter, ranging from 0.0 to 1.0, is the probability that the background of the label will be transparent, i.e., it generates an image without any background color, with just the text (label) itself. So this works as follows. When a label is generated, a random number is compared to the probability given by `transparent_bg_prob`. If the random number is less, then the label will have a transparent background. If on the other hand, the background is not transparent, then the

[37] all available in the panel named "Vintage Background effects"

`min_background_brightness` is used to generate a light background color. It is important to keep in mind that if the output format of the images is JPG, then the transparent backgrounds are converted to a white background because the JPG format does not support transparency. On the other hand, PNG format can have transparent background (the alpha channel is preserved).

Thus the process of assigning color to the text itself and to the background and the canvas of the image occurs as follows:

- Generate the background (either colored or transparent):
  `bg_color = self.settings.generate_light_background()` e.g., "#f0f0f0"
- Create the canvas which includes the background:
  `canvas = Image.new('RGB', (canvas_width, canvas_height), bg_color)`
- Select a random text color from the predefined list:
  `text_color = random.choice(self.settings.text_colors)` e.g., "#000000"
- Draw the text with the selected color:
  `draw.text((x, y), label_text, fill=text_color, font=font)`

The reader is invited to consult the technical documentation for more details about the aforementioned procedure and the definition of the functions. In practice these two parameters might take different values and their respective effects is combined together to generate images with various types of visual characteristics. One scenario would be the following one. We can generate an image with a non-transparent background (the choice by default in the GUI), so we set `transparent_bg_prob` to 0.0 and `min_background_brightness` to 0.8. This setting always generates light pastel backgrounds so that the colors are at least 80% bright (light colors). As a result, we obtain labels with soft, light-colored backgrounds. Another scenario could be, for instance, to use a type of mixed background by setting `transparent_bg_prob` to 0.3 and `min_background_brightness` to 0.9. This selection means that 30% of the generated labels will have a transparent background while 70% of the generated labels will have very light pastel backgrounds (with $\geq$ 90% brightness). This provides a mix of transparent and very light backgrounds as a result.

Thus, combining these different background properties allow the user to handle different cases:

- For OCR and dataset generation, one creates images with high contrast for better OCR with `min_background_brightness` set to 0.9 (this corresponds to very light backgrounds) and the parameter `transparent_bg_prob` set to 0.0 to guarantee that the generated labels always have a non-transparent background. This provides dark text on light backgrounds, a configuration optimized for for OCR.
- For Overlay and composite images, i.e., for labels that will be overlaid on other images. We select `min_background_brightness` set to 0.0 and `transparent_bg_prob` set to 1.0. With this choice of parameter values, the backgrounds are always transparent. This would give Labels containing text only without background.
- For vintage and aesthetic labels with soft vintage look, it is suitable to select `min_background_brightness` set to 0.7 for a medium-light background and choose `transparent_bg_prob` set to 0.0. In this case, the images have always a background and the generated images will be with soft pastel backgrounds which provides a vintage aesthetic.

3.3 Description of the implementation of the GUI

The GUI contains different panels and provides an intuitive control over all parameters that are available to configure the generation of the customized labels. Each panel corresponds to a specific aspect of the label generation and contains options and menus allowing the users to fine-tune the characteristics of the labels. The technical documentation provides the users with detailed information about how to proceed to start generating large sets of customized labels. So, in what follows, we summarize some aspects of the implementation of GUI to give an idea of the kind of workflow that can be managed by the user through the GUI.

One important step when planing the organization of the GUI was to map the approximately 35 parameters available to customize the labels to the appropriate PyQt6 widgets. Once this step was completed, it was possible to structure the different panels of the GUI. The structure and functionalities of the panels of the GUI can be summarized as follows. The panel called "General Settings" contains configuration and setting details to be provided by the user. This includes the number of labels, the image format and the path to output directory where the labels generated should be saved. The second panel is called "Text Content". It contains the label text options (static list which can be customized). The user can also specify if the use of scientific notation should be allowed, and set the probability accordingly. The third panel is named "Options for Units". It allows the user to manage unit suffix options and separators (%, mL, mg, etc.). The user can select the units among a predefined set or enter new customized units and symbols that should be used instead. The fourth panel is called "Font Style". The user has access to the font family selector[38]. The user can also determine the font size range (there is a base font size used as a reference and a second parameter to set the allowed variations from one label to another) and the font weight (it is possible to select multiple options including normal, bold and italic). There is also a menu to select the colors for the text of the labels. The fifth panel is called "Vintage Background effects". There, the user can enable the vintage effects, control the vintage intensity, the noise and blur levels and select the texture overlay and the paper aging intensity. The user has also access to the parameters to enable and adjust the realism effects. There is also the menu where the user can control the minimum background brightness and enable transparent background. The sixth panel is named "Rotation Effects". It gives the user the possibility to generate labels with rotated text. It is possible to select the rotation angles among a predefined list of angle. It is also possible to enable complete automated customization of the angles. The last panel is named "Size Resolution". The user has access to options to configure the allowed ranges of variation of the size of the images, the dimensions, DPI, and set the minimum text padding.

The GUI also possesses a real-time preview panel, a panel to monitor the progress of the label generation and to get real-time feedback and metadata from the console when running the image generation (They work following the same principle than in the case of the scatter plot generator described above).

When designing the GUI, another important aspect was the widget dependencies and conditional logic. for instance, if the user enable the option to customize the size and resolution of the images, then it should also automatically enable options to control the minimum and maximum width, the minimum and maximum high as well as the minimum and maximum dpi, whereas when the user decides to disable the option to customize the size and resolution, this should enable the fixed dpi option and disable range controls. The same logic applies when it comes to manage the rotations related options and parameters. When rotations are allowed by the user, the list of allowed rotation angles and the custom angle options should be made automatically accessible while if rotations are not allowed at all then all rotation related widgets should be automatically disabled. In the same vein, when the option to add realism is set to true,

[38] the font are selected among the font available within the system

this enables the slider controlling the realism intensity and when the realism effects are not selected, the slider controlling realism intensity should be disabled.

These dependencies must be properly considered because they translate into special widget requirements. Thus, the GUI assembly requires to implement each panel with the mapped widgets, then add the dependency logic for enabling and disabling these widgets. Finally, it is necessary to connect all widgets to the proper settings to implement the generation thread for label generation.

The different panels and functions of the GUI work together to allow the user to easily begin generating customized labels. A typical workflow would be as follows. In the configuration step, the users sets the desired parameters across the different panels. The GUI gives the users the possibility to save the settings as a preset[39]. Once the user has reviewed and validated the values selected for the different parameters, the generation can begin. To proceed, the user clicks the button to start the label generation and the system will ask the user to confirm the settings. As the generation progresses, the user can monitor the progress bar and the output messages which provide information about the characteristics of the labels being generated and saved. This panel also gives access to the outputs of the console in real-time. At any time, the users can stop the generation if needed. Upon completion of the generation of the labels, the system displays a completion message and saves the metadata. The user can review the output files and the messages displayed in the console.

[39] optional but recommended if the user has selected specific ranges of values for multiple parameters and wants to be able to generate multiple sets of labels using the same settings