Databending as a Target for Genetic Improvement

Erik M. Fredericks
frederer@gvsu.edu
Grand Valley State University
Allendale, Michigan, USA

Byron DeVries
devrieby@gvsu.edu
Grand Valley State University
Allendale, Michigan, USA

Reihaneh Hariri
haririr@gvsu.edu
Grand Valley State University
Allendale, Michigan, USA

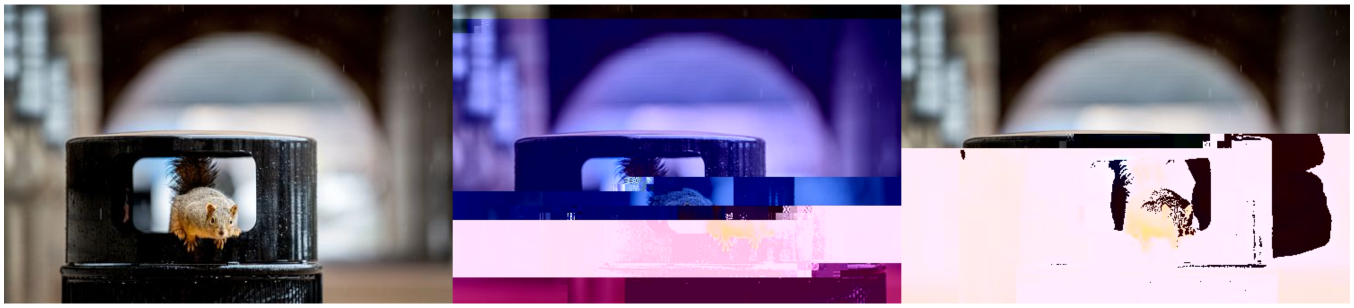

Figure 1: Databending images with *PYGGI* - (left) Original - (middle) Test Set - (right) *PYGGI*-Generated. Original photo taken by GVSU University Communications' Kendra Stanley-Mills.

Abstract

Genetic improvement (GI) is typically used as an approach for optimizing source code for a particular task, where examples include minimizing runtime, improving energy efficiency, and reducing memory footprint. As with any evolutionary algorithm, GI will optimize for a desired fitness objective and can be coerced into objectives that are not strictly for the improvement of software. This paper presents in-progress work towards using GI as a method for generating glitch art, or artwork that is created by intentionally corrupting files (i.e., databending) to provide an aesthetic or emotional experience. We use the *PYGGI* framework to explore how code patches can impact an existing glitch art framework, with the goal being to find new and interesting outputs that are different from what can be found with random search. Our proof of concept did not find a significant different between randomly-generated and *PYGGI*-generated images, and as such we present future avenues of research to improve results.

CCS Concepts

• **Software and its engineering → Search-based software engineering**; **Genetic programming**; *Software libraries and repositories*; *Automatic programming*.

Keywords

genetic improvement, evolutionary computation, glitch art, databending, software engineering

ACM Reference Format:
Erik M. Fredericks, Byron DeVries, and Reihaneh Hariri. 2026. Databending as a Target for Genetic Improvement. In *15th International Workshop on Genetic Improvement (GI '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786162.3793235

1 Introduction

Databending is a glitch art technique that intentionally corrupts data files to produce an intentionally-corrupted aesthetic, often resulting in warped or stretched colors [3, 21]. Typically these techniques involve manually modifying the bytes within an image file or deleting specific frames within a movie file. The second and third images in Figure 1 represent samples of glitched images. A number of tools and filters have been developed to make this process easier, however they are often designed as proof of concept programs with room for improvement. This paper presents early work on using genetic improvement (GI) to discover patches for an existing open source glitching tool to yield new and interesting results.

PYGGI (Python General Framework for Genetic Improvement) is a Python-based framework for applying GI via source code manipulation and managing patches [1]. While written in Python, *PYGGI* is able to improve programs in multiple languages including both Python and Java. *PYGGI* provides a cohesive framework to apply a variety of GI methods regardless of the language targeted, simplifying the difficulty in parsing multiple languages and applying the desired GI modifications. *PYGGI*, and GI in general, have been shown to be capable of both automatic program repair when bugs exist in the target software as well as non-functional improvement (e.g., runtime speed, energy usage) [4, 5]. We focus on using GI to improve existing software, but use a fitness metric that aims to intentionally corrupt output image files to obtain an aesthetic or emotional response rather than a typical program improvement (e.g., reduce execution time).

Prior work in this field involved applying GI to generate drawing programs comprising different techniques and parameterizations [11–13], with results heavily relying upon the suite of input drawing functions. For this paper we apply *PYGGI* [1] to *glitch-tool* [19], an open source proof of concept program developed for demonstrating the capability of glitching various image types with Python file manipulation. We configured *PYGGI* to maximize the overall differences of generated images with a suite of randomly-generated test images, along with maximizing the number of patched lines of code, with the overall goal being to find software patches that yielded new and interesting forms of databending (i.e., code that yields new image results).

Experimental results suggest that applying GI to a glitching program, with the aim of maximizing the novelty of generated images, is feasible. However, current results do not significantly differ from a randomly-generated test set, leading to the indication that further study and improvements to the search procedure are required. The rest of this paper is organized as follows. Section 2 presents background information and related work. Section 3 presents our intended approach for extending *PYGGI* for databending and maximizing novelty in outputs. Section 4 then details our in-progress results and Section 5 discusses our findings and presents future directions.

2 Background and Related Work

This section details background and related work on purposeful glitching or databending images for aesthetic or artful purposes work within the contexts of both GI and generative art.

Image glitching, or databending, alters images by purposefully applying methods that corrupt the original image file to alter the appearance of the image [3]. In 2011, Rosa Menkman wrote *The Glitch Moment(um)* [21] that described the art-based impact of image glitching as more than an aberrant technical process. That is, image glitching, or databending, produces an artistic expression intended to impact the viewer. From a technical perspective, glitched images are created by altering the encoding of an image file in such a way that it creates visual artifacts similar to those in Figure 1 and can be produced by purpose-built applications, including *glitch-tool* [19].

Typically, GI is used to enable a wide range of improvements across both functional and non-functional properties [5] including optimizing performance, repairing programs, and parallelization [8, 14, 15, 25]. Importantly, the search-space of GI methods allow for a variety of search methods to be employed, though it is not clear if an individual search method is most effective [24]. Recent fundamental work in this area has ranged from LLM-based mutations in GI [7], maintaining diversity [22], and improving GI operator selection [6]. GI has yielded impressive results on SAT solvers (e.g., MiniSAT [26]), image processing software (e.g., ImageMagick [27]), and problem-specific optimizations for LLVM compilers [17]. This work differs from the existing work by applying GI to a specific application area that, uniquely, does not necessarily improve the target software.

Prior work in the domain of creating aesthetically pleasing visuals via evolution and exploring generative art techniques includes generating art via grammatical evolution with GI [12, 13]. Further, a comparison investigating the impact of the different fitness functions was performed using many-objective search [11]. Though, these approaches only used GI in the lightest sense (i.e., creating a list of parameterized function calls). Typically, GI that is used to generate images is focused on improving an aspect of the software rather than new or different resulting images. For example, work on improving shaders for Graphics Processing Units (GPUs) focused on reducing code footprint while maintaining an existing visual output [28] or transplanting code between image processing applications [27]. Another evolutionary image glitching project used genetic programming to create glitch art, though it focused on corrupting the image itself rather than the program [10]. Finally, the EvoMusArt conference series has explored how evolutionary computation (among other search/optimization algorithms) can be applied to the domains of music and art generation [20].

3 Approach

We now discuss our approach for incorporating databending into *PYGGI*. In searching for techniques for corrupting images we found a blog post [18] detailing *glitch-tool* [19]. This post was particularly interesting as the tool was specifically developed as a proof of concept with no intention of maintenance or becoming production-ready and was instead made to show the feasibility of corrupting images with Python. *glitch-tool* itself modifies the bytes of a source image file to yield a glitched result, based on a set of input parameters. This type of project seemed highly amenable to GI for discovering improvements to the source code, given that (to us) it is an interesting and useful tool that is no longer maintained and may be improved via GI. Furthermore, we were especially interested in expanding on the typical GI applications based on existing work in optimizing drawing programs [11–13].

We used the `improve_python` example in *PYGGI* and extended the fitness calculation to maximize the difference in images corrupted by *glitch-tool*. As a basis, we extended *glitch-tool* to generate a configurable number of test images for comparison to *PYGGI*-generated images.

Each of the available parameters within *glitch-tool* were encoded in a *PYGGI* test script to enable execution, with the exception of the `--amount` and `output-iterations` parameters. These parameters specify the number of images to generate and changes between each iteration for a *glitch-tool* execution and for this work we only wanted one output per execution. Each of the other parameters were randomly instantiated, within constraints empirically chosen by the authors, per *PYGGI* evaluation. Table 1 lists out the *glitch-tool* parameters and their defined constraints for this paper. Note that we do not include required parameters that do not influence the glitching process as they are assumed to be included (e.g., input file to glitch, output directory). We also prevent the parameterization of *glitch-tool* from being optimized by *PYGGI* as well to ensure that only the source code itself is patched.

To calculate the difference between two images, we use a hashing algorithm (using an average hash algorithm from the Python library `imagehash` [9]) to quantify each image and then take the Hamming distance (i.e,. h_diff) as shown in Equation 1. We note that there are a significant number of approaches that can be used to calculate the difference between images, including thresholding [23] and machine learning [11]. For this paper, we preferred a lightweight approach to minimize time spent within the evolutionary loop calculating fitness. Fredericks *et al.* [12, 13] used pixel differences as a metric for calculating distance, however slight pixel differences

Parameter	Constraint
File change mode (-m)	`random(mode)`
Amount of random changes (-c)	`random(0,100)`
Amount of bytes to change (-b)	`random(0,100)`
Amount of bytes to repeat (-r)	`random(0,100)`
`mode options`	`[change, reverse, repeat, remove zero, insert replace, move]`

Table 1: *glitch-tool* parameters [19] and defined constraints.

Parameter	Value
Number of replicates	25
Type and Search	*Improve Python*, *Tabu Search*
Mode	*Tree*
Iterations	10
Epochs	1000
Number of *generated* test images	5000
Number of *valid* test images	2094
Image comparison algorithm	*Average hash*
Fitness	*See Equation 2*

Table 2: *PYGGI* configuration.

were indistinguishable from noise and not overly helpful in the realm of glitch art.

We calculate individual fitness by combining the averaged Hamming distance between the newly created image and a suite of randomly-generated test images with the number of lines in the patched file, as is demonstrated in Equation 2. The aim of this calculation is to **maximize** the overall difference between the generated image and test images as well as to induce the search process to "provide more" patches, where the intention is to generate new images not previously discovered by random generation. As such, we were intending that "more patches" would potentially discover new avenues for glitching an image.

$$h\_diff = |avg\_hash(new\_image) - avg\_hash(test\_image)| \quad (1)$$

$$fitness = \left(\frac{\sum_{i=0}^{|test\_set|} h\_diff(new\_image, test\_set[i])}{|test\_set|} + |lines_{patched\_file}| \right) \quad (2)$$

4 In-Progress Results

4.1 Configuration

We now present current results from our early-stage efforts. We used the current version of *PYGGI* (i.e., commit $14361bb$) [2] to enable GI processes and the current version of *glitch-tool* (i.e., commit $7fced34$) [19] for our target tool. To compare images we used an average hash algorithm, provided by the *ImageHash* Python library [9]. Comparing image hashes provides a relatively lightweight approach for determining image similarity and can be embedded within an evolutionary algorithm. Our code may be found on our GitHub repository[1] and our generated output images can be found on Zenodo.[2] We configured *PYGGI* as follows in Table 2:

We experimented with different configurations for *PYGGI*, including testing both *Line* and *Tree* modes, using local and Tabu search, and using different values for the number of iterations and epochs. Ultimately the values we settled on in Table 2 provided a search process that subjectively appeared to provide more patches and visual differences in generated files. As part of the test set creation process, 5000 images were generated however over half were found to be corrupted and removed from the test set, leaving 2094 images to be used for comparison. The corrupted images were not considered as attempting to use them in the image hashing fitness calculation would cause errors in parsing the image data (either leading the program to halt or ignore those images as part of `try/catch` blocks).

4.2 Results

As this is early-stage research, we leave a full empirical evaluation for future work. However, we present our current results from initial *PYGGI* runs in generating patches that yield digitally-glitched images. Our initial *research question* was to determine if *PYGGI* is suitable for automatically patching a glitching program to create new and interesting glitch art that could not be found by running the normal program, where the overall intent was not necessarily to *improve* a program, but to generate *diversity* in its outputs. To evaluate this question, we generated a set of test images using randomly-instantiated parameters (see Table 1) to the *glitch-tool* program and then performed a fitness evaluation per Equation 2 to determine the overall difference from each test image to all others in the test set. We then instrumented *PYGGI* with the ability to patch the *glitch-tool* program, leveraging the same random forms of selection for its input parameters (see Table 1), and executed it per Table 2.

Listing 1 presents two sample (truncated) patches generated by *PYGGI* that were considered "the best" at the end of a run. The first patch includes a transplanted variable (i.e., `iteration`) that is not used within the current code block. The second patch transplants a function `changeBytes` that is used as a global function for modifying image bytes, however its inclusion within the conditional statement does significantly not impact program behavior.

```
*** before: glitch_tool.py
--- after: glitch_tool.py
***************
*** 21,26 ****
--- 21,27 ----
          )
        if not args.quiet:
            print('Writing␣file␣to␣' + outPath)
+       iteration = 1
        open(outPath, 'wb').write(\
        bytes(fileByteList))
***************
*** 40,45 ****
--- 41,53 ----
    if (args.output_iterations > 0 and \
        iteration % args.output_iterations \
          == 0):
        writeFile(newByteList, fileNum, \
```

[1]See https://github.com/efredericks/pyggi/tree/gi2026.
[2]See https://zenodo.org/records/17407891.

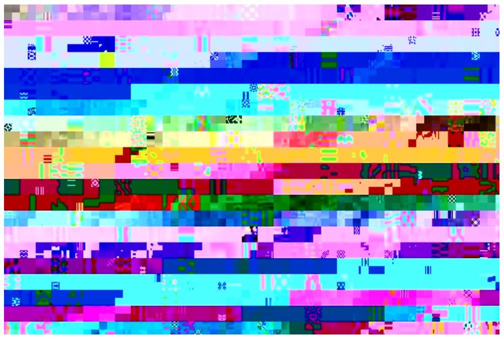

Figure 2: Test set image with highest averaged distance.

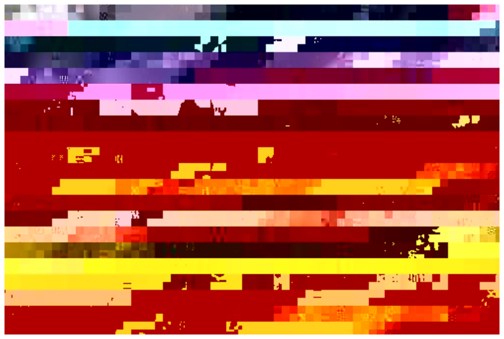

Figure 3: *PYGGI*-image with highest averaged distance.

```
        iteration , bytesToChange , seed)
+       def changeBytes(byteList , \
          bytesToChange):[40/1848]
+           global args
+           pos = random.randint(0, \
              len(byteList) - bytesToChange)
+           chunk = [random.randint(0, 255) \
                for i in range(bytesToChange)]
+           byteList[pos:pos + bytesToChange] \
                = chunk
+           return byteList
    writeFile(newByteList , fileNum , iteration ,\
        bytesToChange , seed)
```

Listing 1: Generated patches.

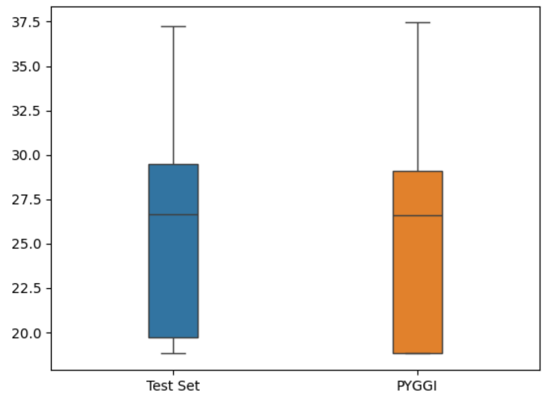

Figure 4: Average Hamming distances comparisons between test set images and *PYGGI*-generated images.

Figures 2 and 3 demonstrate the two images with the largest Hamming distance (see Equation 1), averaged over comparisons to all test set images, between the test set and *PYGGI*, respectively. Their scores are relatively close (37.235 and 38.887, respectively) and visually the main differences appear to be the color palette. Note that we don't include the patch size as part of the score here, as the test set does not include any patching. Additionally, each of the *PYGGI*-generated images that we included in this paper would not render as they were corrupted – the images included are post-processed and saved as PNGs to enable LaTeX to render them (i.e., opened in Krita and exported as PNG).

Figure 4 shows the maximum Hamming distance (see Equation 1) found between images generated for our test set (and compared to all other test images in that set) and for those images that were generated via *PYGGI*. The boxplot on the left (test set differences) shows our baseline Hamming distances and the boxplot on the right shows the comparison of *PYGGI*-generated images to the test set images. While there is a disparate number of values per boxplot (2094 for the test set and 74714 for *PYGGI*), there is presently no significant difference between fitness values.

We also noted that most of the generated patches at the end of a run appear to be fairly trivial. For example, we found in results summaries that patches often resolved to moving around variable declarations. Our initial assumption is that either our solution space is too limited or that we are getting stuck in a local optima. Regardless, further study is required.

Figures 5 and 6 present screenshots of the image thumbnails from our test image set and one replicate of a *PYGGI* run. While visually there do not appear differences in the glitched outputs, we note that subjectively there appear to be "more" glitches in the *PYGGI*-generated files. Notably, there are more files that are "corrupted" (denoted by the blank image icon) and there seems to be more that are "glitchier" than those in the test set. However, when verifying with our image difference calculations there does not appear to be a significant difference in scores.

Our **future plans** to extend these initial results are to incorporate other hashing algorithms (e.g., perceptual hashing, wavelet hashing) to determine if one approach induces more useful patches than another. We also plan to compare our *PYGGI* results with those found with *GenerativeGI* [11–13] by extending its current suite of drawing/glitching techniques to incorporate a *base image layer* to corrupt. Further, we plan to incorporate novelty search [16] into *PYGGI* with the aim of improving its searching potential for diversity in patches.

5 Discussion

This paper has presented early work towards optimizing a program for generating a wide variety of outputs using GI. Our early results suggest that, while interesting, our implementation performs on par with random generation. These results indicate that further study is warranted to determine where our fitness calculations are plateauing. Our initial thoughts are that: (1) our implementation of *PYGGI* is not modifying the base program enough to merit a difference in outputs, (2) our search procedure has hit a local optima that our current fitness calculations and evolutionary operators do not move beyond, and (3) that the base *glitch-tool* program requires

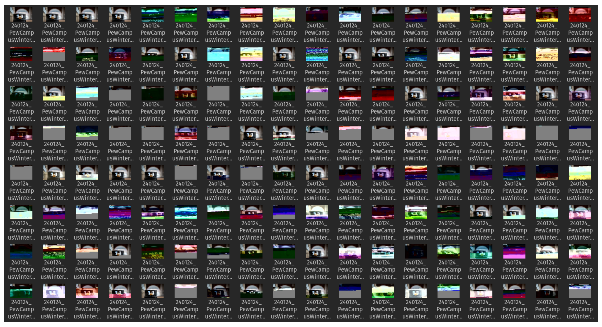

Figure 5: Screenshot of test image previews.

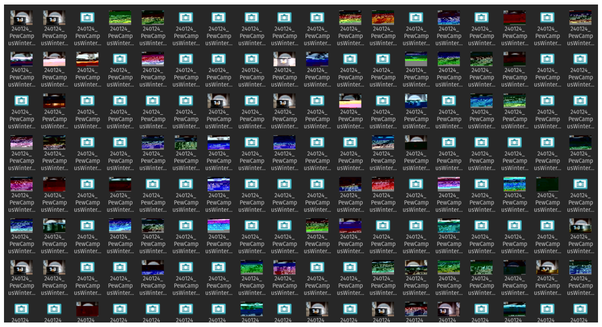

Figure 6: Screenshot of generated image previews.

additional extensions to enable *PYGGI* to provide more nuance in its patches.

Future work for this project includes a deeper empirical analysis of databending with *PYGGI*, including exploring the impact on other art creation tools and manually-corrupting image and video files. Additionally, we aim to include other image comparison techniques within our fitness calculation to determine if other methods improve the results set, including histogram comparisons, pixel differences (as was previously explored in [12, 13]) and using machine learning techniques [11]. Finally, we intend to explore other unique optimization problems, including procedurally-generating content for video game environments and guided fuzzing for safety-critical systems.

Acknowledgments

We gratefully acknowledge the support of Grand Valley State University for this project.

References

[1] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 1100–1104. doi:10.1145/3338906.3341184

[2] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 1100–1104. doi:10.1145/3338906.3341184

[3] Ilias Bergstrom and R Beau Lotto. 2015. Code Bending: A new creative coding practice. *Leonardo* 48, 1 (2015), 25–31.

[4] Aymeric Blot and Justyna Petke. 2021. Empirical Comparison of Search Heuristics for Genetic Improvement of Software. *IEEE Transactions on Evolutionary Computation* 25, 5 (2021), 1001–1011. doi:10.1109/TEVC.2021.3070271

[5] Aymeric Blot and Justyna Petke. 2025. A Comprehensive Survey of Benchmarks for Improvement of Software's Non-Functional Properties. *Comput. Surveys* 57, 7 (2025), 1–36.

[6] Damien Bose, Carol Hanna, and Justyna Petke. 2025. Enhancing Software Runtime with Reinforcement Learning-Driven Mutation Operator Selection in Genetic Improvement. In *2025 IEEE/ACM International Workshop on Genetic Improvement (GI)*. IEEE, 27–34.

[7] Alexander EI Brownlee, James Callan, Karine Even-Mendoza, Alina Geiger, Carol Hanna, Justyna Petke, Federica Sarro, and Dominik Sobania. 2025. Large language model based mutations in genetic improvement. *Automated Software Engineering* 32, 1 (2025), 15.

[8] Bobby R Bruce, Justyna Petke, and Mark Harman. 2015. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 1327–1334.

[9] Johannes Buchner. 2025. ImageHash GitHub Repository. https://github.com/JohannesBuchner/imagehash Accessed: 2025-10-09.

[10] Eelco Den Heijer. 2013. Evolving glitch art. In *International Conference on Evolutionary and Biologically Inspired Music and Art*. Springer, 109–120.

[11] Erik M Fredericks, Denton Bobeldyk, and Jared M Moore. 2025. Crafting generative art through genetic improvement: Managing creative outputs in diverse fitness landscapes. In *Genetic Programming Theory and Practice XXI*. Springer, 321–335.

[12] Erik M Fredericks, Abigail C Diller, and Jared M Moore. 2023. Generative Art via Grammatical Evolution. In *Proceedings of the 12th International Workshop on Genetic Improvement*.

[13] Erik M Fredericks, Jared M Moore, and Abigail C Diller. 2024. GenerativeGI: creating generative art with genetic improvement. *Automated Software Engineering* 31, 1 (2024), 23.

[14] William B Langdon. 2020. Genetic improvement of genetic programming. In *2020 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1–8.

[15] William B Langdon and Mark Harman. 2014. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (2014), 118–135.

[16] Joel Lehman and Kenneth O Stanley. 2011. Novelty search and the problem with objectives. In *Genetic programming theory and practice IX*. Springer, 37–56.

[17] Shuyue Stella Li, Hannah Peeler, Andrew N Sloss, Kenneth N Reid, and Wolfgang Banzhaf. 2022. Genetic improvement in the Shackleton framework for optimizing LLVM pass sequences. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1938–1939.

[18] Tobias Løfgren. 2018. Glitch Art - Adventures in databending. https://tobloef.com/blog/glitch-art/ Accessed: 2025-10-09.

[19] Tobias Løfgren. 2019. glitch-tool GitHub Repository. https://github.com/tobloef/glitch-tool Accessed: 2025-10-09.

[20] Tiago Martins and João Correia. 2025. EvoMUSART Index. https://evomusart-index.dei.uc.pt/ Accessed on 2025-10-20.

[21] Rosa Menkman. 2011. *The glitch moment (um)*. Vol. 4. Institute of Network Cultures.

[22] Zsolt Németh, Penn Faulkner Rainford, and Barry Porter. 2025. Reaching Meaningful Diversity with Speciation-Novelty in Genetic Improvement for Software. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1017–1025.

[23] OpenCV. 2025. Image Thresholding. https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html Accessed on 2025-10-20.

[24] Justyna Petke, Brad Alexander, Earl T Barr, Alexander EI Brownlee, Markus Wagner, and David R White. 2019. A survey of genetic improvement search spaces. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1715–1721.

[25] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. 2017. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2017), 415–432.

[26] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. 2014. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *European Conference on Genetic Programming*. Springer, 137–149.

[27] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. 2017. Specialising software for different downstream applications using genetic improvement and code transplantation. *IEEE Transactions on Software Engineering* 44, 6 (2017), 574–594.

[28] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. 2011. Genetic programming for shader simplification. *ACM Transactions on Graphics (TOG)* 30, 6 (2011), 1–12.