FRACTAL ANALYSIS OF GOOD PROGRAMMING STYLE

Ron Coleman and Pritesh Gandhi

Computer Science Department, Marist College, Poughkeepsie, New York, USA
roncoleman@marist.edu pritesh.gandhi1@marist.edu

ABSTRACT

This paper studies a new, quantitative approach using fractal geometry to analyse basic tenets of good programming style. Experiments on C source of the GNU/Linux Core Utilities, a collection of 114 programs or approximately 70,000 lines of code, show systematic changes in style are correlated with statistically significant changes in fractal dimension ($P \leq 0.0009$). The data further show positive but weak correlation between lines of code and fractal dimension ($r=0.0878$). These results suggest the fractal dimension is a reliable metric of changes that affect good style, the knowledge of which may be useful for maintaining a code base.

KEYWORDS

Good programming style, fractal dimension, GNU/Linux Core Utilities

1. INTRODUCTION

Good programming style is a way of writing source code. Although different style guides have different conventions, a survey of contemporary texts [1] [2] [3] [4] finds general agreement on three basic rules: use proper indentation, include documentation, and choose meaningful or mnemonic names. While style guides stress the importance of good style, especially for maintenance purposes, "good" is a value word and "style" connotes, among other things, form and taste. In other words, we propose source has potential elegance as a work of art like a painting or photograph and indeed, any given programming style, including an indecorous one, may be readily accessible without an in-depth understanding of how the code works or even what it does. In this view, source has aesthetic or *sensori-emotional* qualities.

We are not suggesting aesthetic appeal in code should be an overarching goal of software, only that it plays a role in crafting and maintaining code as a best practice. Yet aesthetics present challenges. According to a modernist, Kantian view [5], aesthetics in general and notions of beauty and matters of taste in particular are thought to be subjective, relative, and presumably beyond the pale of automation. However, software engineers have sidestepped these dilemmas, asking not what *is* beauty in source but rather what is *knowable* about such beauty (e.g., good programming style), which can be incorporated in programs like the GNU/Linux command, *indent* [6], which beautifies C source by refactoring indentation, comments, and spacing. Tools like *indent* are a staple of modern software engineering. Unfortunately, these tools do not quantify the value of their beautifying regimes, as a consequence developers have had to resort to anecdotal arguments rather than metrics to reason about aesthetic outcomes, and no research effort heretofore has investigated the problem or its opportunities.

In this paper, we study a new, quantitative way to analyse basic tenets good programming style using fractal geometry [7]. Fractals are often associated with beauty in nature and human designs [8]. Furthermore since fractals are self-similar and scale-invariant, we hypothesized a fractal approach might be inherently robust for handling distributions of source sizes.

Experiments with the C source code of the GNU/Linux Core Utilities [9], 114 commands of the Linux shell or about 70,000 lines of code (LOC), show systematic changes in programming style are correlated with statistically significant changes ($P \leq 0.0002$) in fractal dimension [10]. The data further show that while the baseline sizes of C source files vary widely, there is a positive but weak correlation with fractal dimension ($r=0.0878$). These data suggest the fractal dimension is a reliable metric of changes in source that affect good style, the knowledge of which may be useful for maintaining a code base.

2. Related Work

Aesthetic value in source is not the same as readability [11] [12], although the two are related. The latter is more about comprehending code whereas the former, appreciating it, *l'art pour l'art*. Beauty in source is also not the same as functional complexity [13]. Complexity relates to design and efficiency in algorithms and data structures, which may have appeal in a conceptual, though not necessarily a visual sense, although here again there is overlap. *Beautiful Code* [14] explores just this sort of conceptual aesthetic, not only in source but also in debugging and testing which are not subjects we consider. Gabriel [15] argues against clarity and conceptual beauty as primary goals of software in favour of what the author calls "habitability." Yet comfort with the code is independent of style since programmers might forgo style best practices as long as they can live with it, whereas our starting point *is* good style. The fractal dimension has been applied to a wide range of disciplines, though not software development [16]. Our code depends on Fractop [17], a Java library originally developed to categorize neural tissue. We have reused this library to analyze source code. Some researchers have employed the fractal dimension to study paintings of artists [18]; others working in a similar vein have used the fractal dimension to authenticate Jackson Pollack's "action paintings" [19] [20]. Still others have used the fractal dimension to examine aesthetic appeal in artificially intelligent path finding in videogames [21] [22] [23]. An investigation of Scala repositories on GitHub.com found sources are organized according to power-law distributions [24] [25] but that effort did not consider style. Kokol, et al, [26] [27] [28] reported evidence of fractal structure and long-range correlations in source; however, they were investigating not style but fine details, character, operator, and string patterns in a small sample of randomly generated Pascal programs. We study style in a moderate size sample of highly functional C programs.

3. Methods

We use a multi-phase operation to process a single source file: 1) beautify or de-beautify the source style, if necessary; convert the result to an in-memory representation called an *artefact*; 3) calculate the fractal dimension of the artefact.

To beautify the source in phase 1, we use a combination of the GNU/Linux *indent* command and a kit we developed called *Mango* [29] (see below). The *indent* manual page [6] gives input options for beautifying the source according to four distinct C styles: GNU, K&R (Kernighan and Ritchie), Berkeley, and Linux (kernel). They affect indentation, spacing, and comments and differences can be found in the manual page. The command, *indent*, does not, however, change mnemonics.

Mango is a kit written in Scala, C, and to drive the experiments, Korn shell scripts. During the first phase of processing, Mango mostly does the reverse of *indent*: it "mangles" or de-beautifies C source and outputs new source as we discuss below.

3.1. Base lining measurements

To get baseline measurements of the source, Mango skips phase 1 and sends the unmodified source directly to phases 2 and 3 to generate the artefact and calculate the fractal dimension, respectively.

3.2. De-beautifying source

When de-beautifying source in phase 1, Mango does one of the following: remove indentation, randomize indentation, remove comments, or make the names of variables, functions, macros, and labels less mnemonic. To remove indentation, Mango trims each line of spaces. To randomize the indentation, Mango inserts a random number of spaces to the beginning of the line. To remove comments, Mango strips the file of both block (/* … */) and line (//) comments. Finally, to make names less mnemonic, Mango shortens them according to the algorithm below.

3.3. Non-mnemonic algorithm

The algorithm to shorten names requires two passes over the source. During the first pass Mango filters key words, compiler directives, library references, names with less than a minimum length (l=3), and names appearing less than a minimum frequency (n=3). For names that get through these filters, Mango calculates new, non-mnemonic names as follows. If a name has at least one under bar ("_"), Mango splits the name along the under bar and recombines the first letter of each subsequent sub-name with the whole first sub-name followed by an under bar. If a name is uppercase name, Mango uses every other letter to reform the name, effectively, cutting the name in half. If a name is neither of these, it shortens the name by half. Mango puts the old name and the new name in a database for lookup and substitution back into the source during the second pass. The table below gives some examples of how the algorithm works.

Table 1. Example changes by non-mnemonic algorithm

Old name	New name
i	i
T_FATE_INIT	T_FI
NOUPDATE	NUDT
linkname	link

3.4. Mnemonic algorithm

Mango also has a beautify mode of phase 1 to make names more mnemonic. Mango does not, of course, know the intention of programmers or semantics of names. However, it can simulate these by lengthening names. The algorithm to lengthen names is similar to the one to shorten them. During the first pass Mango collects appropriately filtered candidate names of a maximum length (*l*=3) and with a minimum frequency (*n*=3). Mango makes these names a maximum of length of four by repeating the letters in the name or adding an under bar after the name. The table below gives some examples of how the algorithm works.

Table 2. Example changes by the mnemonic algorithm

Old name	New name
loop	loop
foo	foo_
go	gogo
i	iiii

3.5. Artefact generation

Phase 2 of Mango converts an input source file it to an artefact, which has one of two types of encodings: *literal* and *block.*

With literal encoding, the flat text of the source is written to a buffered image using a graphics context. The text is Courier New, ten-point, plain style, and black foreground over a white background with ten-point line height. In this case, the artefact looks identical the flat text except it's in bitmap form.

With block encoding, each character in the input is written to the graphics context as "blocks" or 8×10 (pixels) black filled rectangles over a white background with two pixels between each rectangle. Spaces are 10×10 pixels. A block artifact resembles the source but in digital outline.
Block encoding has two advantages. It makes the artefact more robust, more independent language. Similarly, it makes the mnemonic and non-mnemonic algorithms more robust. In fact, for these algorithms with block encoding, only the length of the name is relevant, not the name itself.

The figure below is an example of a simple C program.

```
#include <stdio.h>
int main(int argc, char** argv) {
  printf("Hello, world!");
  return 0;
}
```

Figure 1. Simple C file which is identical to its literal artefact encoding except in bitmap form

A literal artifact looks identical to the figure above except it is a bitmap.

The figure below shows the same C program as a block artifact.

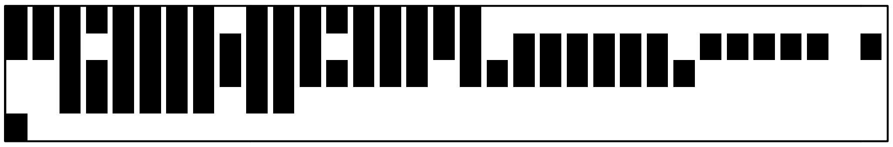

Figure 2. Same C file as an artefact with block encoding

As the reader can see from the figure above, all the language details have been "blocked". Only the digital outline persists.

3.6. Fractal dimension calculation

The third and final phase of Mango measures the fractal dimension of the artefact. Mandelbrot [9] described fractals as geometric objects, which are no-where differentiable, that is, textured, and self-similar at different scales. We use the geometric interpretation based on reticular cell counting or the box counting dimension. We choose this method for two reasons. Firstly, the box counting dimension is conceptually and computationally straightforward. Secondly, Fractop [x] provides a tested, high quality implementation.

Mandelbrot also said fractal objects have fractional dimension, *D*, namely, a non-whole number called the fractional dimension. Mathematically, *D* is given by the Hausdorff dimension [15]:

$$D(S) = \lim_{\varepsilon \to \infty} \frac{log N_\varepsilon}{\log\left(\frac{1}{\varepsilon}\right)} \tag{1}$$

where *S* represents a set of points on a surface (e.g., coastlines, brush strokes, source lines of code, etc.), ε is the size of the measuring tool or ruler and $N_\varepsilon(S)$ is the number of self-similar objects or subcomponents covered by the measuring tool. For fractal objects, log $N_\varepsilon(S)$ will be greater than log (1/ε) by a fractional amount. If the tool is a uniform grid of square cells, then a straight line passes through twice as many cells if the cell length is reduced by a factor of two. A fractal object passes through more than twice as many cells.

The artefact is *S* from Equation 1. Mango uses the Fractop default grid sizes of 2, 3, 4, 6, 8, 12, 16, 32, 64, and 128 measured in pixels for ε. For any given input artefact, Mango returns *D*, which is the slope of the line of the log proportion of cells intersected by the surface increases as log cell size decreases.

4. EXPERIMENT DESIGN

The GNU/Linux Core Utilities version 8.10 [8] comprise 114 dot C source files. First, we generated descriptive statistics for this test bed for number of files and LOC.

We then ran three experiments as follows

1. Established baseline *D* using the original, unmodified C files with literal and block artefact encodings.

2. Treat the source with de-beautifying regimes using Mango to i) remove indentation, ii) randomize indentation by 0-20 spaces, iii) randomize indentation by 0-40 spaces, iv) make names non-mnemonic, and v) remove comments.

3. Treat the source with beautifying regimes using Mango to i) make names more and using GNU/Linux *indent* to refactor the source with ii) GNU, iii) K&R, iv) Berkeley, and v) Linux style settings.

We observed the frequency and direction in which D changes relative to the baseline. We computed the percentage change and the one-tailed P-value using the Binomial test [30]. We also measured the rank correlation coefficient, Spearman's rho [30], between the baseline D and lines of code over all source files.

5. RESULTS

The table below gives the test bed summary statistics. The range of LOC is fairly wide, from files with just two lines to several thousand lines.

Table 3. Test bed summary statistics

Files	114
Total LOC	69,722
Median LOC	356
Maximum LOC	4,733
Minimum LOC	2

The table below gives the baseline fractal dimension values for literal and block encodings.

Table 4. Baseline analysis

	Literal	Block
Median D	1.4592	1.6500
Maximum D	1.5448	1.7176
Minimum D	0.9836	1.4011
r (LOC v. D)	0.0878	0.0878

5.1 De-beautifying treatments

The tables below give the direction and the frequency of changes D decreases in relation to the baseline. As the reader can see the fractal dimension decreases in each case with a small difference between literal and block encoded artefacts. Removing indents is statistically significant, however, as a contrarian indicator. In other words, rather than decreasing D, it increases it in relation to the baseline. We explore this matter further below.

Table 5. Changes in *D* in relation to the baseline with literal encoding

Treatment	Dir.	Freq.	Rate	P
Random indents 0-20	down	112	98%	<0.0001
Random indents 0-40	down	109	96%	<0.0001
Remove indents	up	107	94%	<0.0001
Remove comments	down	82	72%	<0.0001
Non-mnemonic	down	104	91%	<0.0001

Table 6. Changes in *D* in relation to the baseline with block encoding

Treatment	Dir.	Freq.	Rate	P
Random indents 0-20	down	113	99%	<0.0001
Random indents 0-40	down	113	99%	<0.0001
Remove indents	up	107	94%	<0.0001
Remove comments	down	112	98%	<0.0001
Non-mnemonic	down	106	93%	<0.0001

5.2 Beautifying treatments

The tables below give the direction and the frequency of changes *D* decreases in relation to the baseline.

Table 7. Changes in *D* in relation to the baseline with literal encoding

Treatment	Dir.	Freq.	Rate	P
GNU style	up	100	88%	<0.0001
K&R style	up	105	92%	<0.0001
Berkeley style	up	74	65%	0.0009
Linux style	up	106	93%	<0.0001
Mnemonic	up	97	85%	<0.0001

Table 8. Changes in *D* in relation to the baseline with block encoding

Treatment	Dir.	Freq.	Rate	P
GNU style	up	112	98%	<0.0001
K&R style	up	104	91%	<0.0001
Berkeley style	up	78	68%	<0.0001
Linux style	up	105	92%	<0.0001
Mnemonic	up	99	87%	<0.0001

5.3 No indentation as contrarian indicator

The experimental results in section 5.1, "De-beautifying treatments," removed indentation on all the source lines and we found *D* increased. We hypothesized that if removing indentation were a contrary indicator, we expect *D* to rise from the baseline (0% rate) to complete indentation removal (100% rate). The null hypothesis is no change in *D* is affected by the removal rate. To test the null hypothesis, namely, no change in *D* with change in removal rate, we examined several files and found we could reject the null, at least on a subset of typical size files. For instance, mktemp.c has 358 LOC, which is very close to the median size file. We removed the indentation on randomly selected lines at 75%, 50%, and 25% rates and measured D in ten trials using literal encoding. The data for mktemp.c is in the table below is typical for other programs we examined.

Table 9 D for different random remove rates over ten trials for mktemp.c

	Indentation removal rate		
Trial	25%	50%	75%
1	1.468205428	1.470438295	1.476648907
2	1.46463698	1.472219091	1.47721244
3	1.465692458	1.470056954	1.475848552
4	1.465102815	1.47256331	1.479550183
5	1.464691894	1.469024252	1.477846232
6	1.464413407	1.470376845	1.480434004
7	1.465313286	1.474732486	1.481568639
8	1.466252928	1.470800863	1.480060737
9	1.469609632	1.470203698	1.474179211
10	1.467231153	1.468487205	1.480865379
Median	1.465502872	1.47040757	1.478698207

The chart below shows the plot with the median values for 25%, 50%, and 75% removal rates, the baseline (0%), and complete removal (100%).

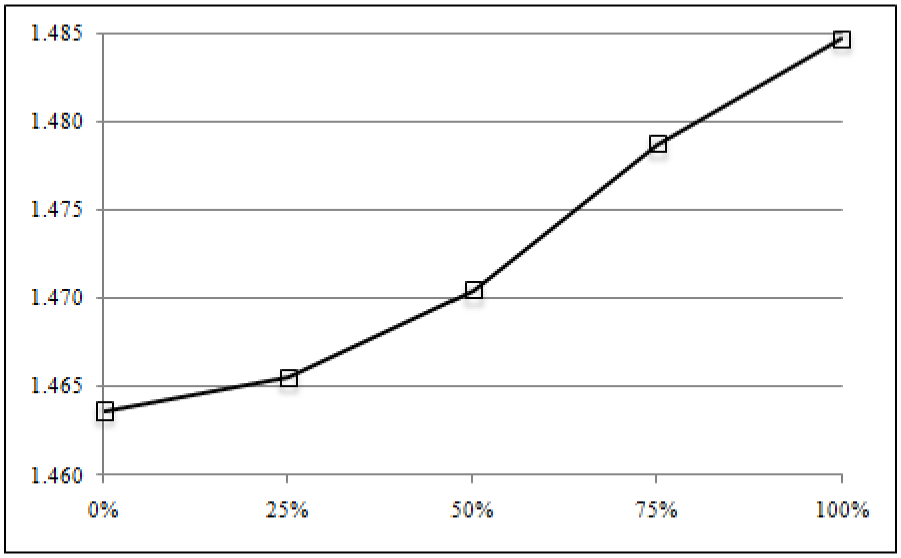

Figure 3 The rate of indentation removal rate vs. D for mktemp.c where 0% is the baseline and 100% is removal of all indentation.

6. Discussion

The first observation we make is generally $D^{\text{literal}} < D^{\text{block}}$. This makes sense since the block encoding covers more surface area, S, in the artefact than the literal encoding. Our preference is for block encoding because of its robustness we mentioned earlier. Nevertheless the pattern of results is consistent between literal and block encoding. When we de-beautify the source, *D* decreases; when we beautify the source, *D* increases.

The exception, we noted, is the removal of all indentation. Yet Figure 1 suggests that removing indentation is a contrarian indicator of style. We believe the contrariness is a peculiar property of the fractal dimension. That is, keeping in mind that D=2 means there is no texture and we have a completely covered surface of a solid colour, the larger *D* for removing indentation implies greater surface area. Thus, having all the text aligned on the left gives a more compact, and thus complete, surface.

All the beautifying treatments increase in *D*. The indent command programmed with Linux style is the most effective for raising *D* and Berkeley style, the least effective.

What is most interesting is that since the GNU/Linux Core Utilities were presumably written with the GNU style guide, the GNU style-beautifying regime nonetheless increases *D*. If changes in *D* are represent changes in style as the data suggests, then it appears there may be room yet for style improvements in the Core Utilities.

This observation offers insight into how to formulate a relative aesthetic value. Consider, for instance, the conflict between regimes that beautify code and increase *D* and the contrarian effect of removing all indentation, which de-beautify the code but also increase *D*. One way to resolve this is to randomly sample the removal of indentation at different rates, measure *D* for each rate as we did above, and test the slope of the line. If it is near zero, we assume there must be poor indentation. In fact, the slope might be the aesthetic value of the indentation. A similar process could be developed for documentation and mnemonics.

7. CONCLUSIONS

We have seen how systematic changes in the style of C programs affect the fractal dimension in a statistically significant manner. Future research may consider the nature of these changes, i.e., how much beauty was added or removed by a change in style as suggested in the discussion. Another useful avenue is confirming these results for programming languages other than C.

REFERENCES

[1] Vermeulen, Allan & Ambler, Scott W., (2000) The Elements of Java Style, Cambridge
[2] Oulline, S., (1992) C Elements of Style: The Programmer's Style Manual for Elegant C and C++ Programs, M&T, 1992
[3] Google, Inc., (2015) "google-styleguide", http://code.google.com/p/google-styleguide/, accessed 11-May-2015
[4] NOAA National Weather Service, National Weather Service Office of Hydrologic Development, (2007) "General Software Development Standards and Guidelines Version 3.5"
[5] Kant, Immanuel, (1978) The Critique of Judgment (1790), translation by J. C. Meredith, Oxford University Press
[6] Free Software Foundation, (2015) http://linux.die.net/man/1/indent, access 13-May-2015
[7] Mandelbrot, Benoit, (1967) "How long is the coast of Britain? Statistical self-similarity and fractional dimension," Science, vol. 156 (3775), p. 636-638
[8] Peltgen, Heinz-Otto & Richter, P.H., (1986) The Beauty of Fractals, Springer, 1986

[9] Free Software Foundation (2015) http://www.gnu.org/software/coreutils/coreutils.html, accessed 11-May-2015
[10] Mandelbrot, Benoit, (1982) Fractal Geometry of Nature, Freeman, 1982
[11] Posnett, Daryl, Hindle, Abram & Devanbu, Prem, (2011) "A Simpler Model of Software Readability", MSR '11 Proceedings of the 8th Working Conference on Mining Software Repositories
[12] Buse, Raymond P.L., & Weimer, Westley R., (2008) "A metric for software readability," ISSTA '08 Proceedings of the 2008 international symposium on Software testing and analysis
[13] Tran-Cao, De, Lévesque, Ghislain, & Meunier, Jean-Guy, (2004) "A Field Study of Software Functional Complexity Measurement," Proceedings of the 14th International Workshop on Software Measurement
[14] Oram, Andy & Wilson, Greg, eds. (2007) Beautiful Code, O'Reilly
[15] Gabriel, Richard, (1996) Patterns of Software, Oxford
[16] Schroeder, M., (2009) Fractals, Chaos, and Power Laws, Dover, 2009
[17] Cornforth, David, Jelinek, Herbert, Peichl, Leo, (2002) "Fractop: A Tool for Automated Biological Image Classification," Proceedings of the Sixth Australia-Japan Joint Workshop on Intelligent and Evolutionary Systems, p. 1-8
[18] Gerl, Peter, Schönlieb, Carola, Wang, Kung Cheih, (2004) "The Use of Fractal Dimension in Arts Analysis," Harmonic and Fractal Image Analysis, 2004, p. 70-73
[19] Coddington, Jim, Elton, John, & Rockmore, Daniel, Wang, Yang, (2008) "Multifractal analysis and authentication of Jackson Pollock paintings" Proc. SPIE 6810, Computer Image Analysis in the Study of Art, 68100F; doi: 10.1117/12.765015
[20] Taylor, R.P, Micolich, A.P., Jonas, D., (1999) "Fractal analysis of Pollock's drip paintings," Nature, vol. 399, June 1999
[21] Coleman, R, (2009) "Long-Memory of Pathfinding Aesthetics," International Journal of Computer Games Technology, Volume 2009, Article ID 318505
[22] Coleman, R., (2009) "Fractal Analysis of Stealthy Pathfinding," International Journal of Computer Games Technology, Special Issue on Artificial Intelligence for Computer Games, Volume 2009, Article ID 670459
[23] Coleman, R., (2008) "Fractal Analysis of Pathfinding Aesthetics," International Journal of Simulation Modeling, Vol. 7, No. 2
[24] Coleman, Ron, Johnson, Matthew, (2014) "A Study of Scala Repositories on Github", International Journal of Advanced Computer Science Applications, vol. 5, issue 7, August 2014
[25] Coleman, Ron, Johnson, Matthew, (2014) "Power-Laws and Structure in Functional Programs," Proceedings of 2014 International Conference on Computational Science & Computational Intelligence, Las Vegas, NV, IEEE Computer Society
[26] P. Kokol, J. Brest, and V. Zumer, "Long-range correlations in computer programs," Cybernetics and systems, 28(1), 1997, p43-57
[27] P. Kokol, J. Brest, "Fractal structure of random programs," SIGPLAN notices 33(6), 1998, p33-38
[28] P. Kokol "Searching for fractal structure in computer programs," SIGPLAN 29(1), 1994
[29] Coleman, R., Pretty project, (2015) http://github.com/roncoleman125/Pretty, accessed 11-May-2015
[30] Conover, W.J., (1999) Practical Non-Parametric Statistics, Wiley