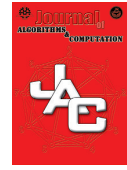
Journal of Algorithms and Computation

journal homepage: http://jac.ut.ac.ir



Remarks on a rendering method for limit sets of Kleinian groups

Alessandro Rosa*[1]

[1]Software Developer, Brindisi, Italy.

ABSTRACT

We revised our technique for generating graphical renderings of the limit sets of Kleinian groups. The algorithm, relying on numerical base conversion, has been improved to shorten computation times. This method easily applies to any family of Kleinian groups with an arbitrary number of generators. Following an overview of this problem, we present the implementation guidelines in the form of pseudo-code.

Keyword: Kleinian groups, limit sets, rendering, coding, Möbius transformations.

ARTICLE INFO

Article history:
Research paper
Received 05, August 2022
Received in revised form 18, November 2022
Accepted 03, December 2022
Available online 30, December 2022

AMS subject Classification: 05C78.

1 Introduction

The subject of this article has been dragged into the caravan of fractal shapes since the beginning of their modern revival through the computer graphics in the late 1970s. The intricate shapes of Julia sets stunned the audience first; mathematicians went on to render another family of similar limit sets generated by Kleinian groups later on. Both environments come up from researches conducted around the turn of the last two centuries using processes involving infinitely many compositions of maps in one complex variable z. Julia's are limit sets coming from the iteration of any of such functions $f(z)$ of non-linear

*Corresponding author:A. Rosa. Email: alessandro.a.rosa@gmail.com

degree, whereas those for Kleinian groups[1] are generated by the combinations of linear fractional transformations (or 'Möbius maps') in the form $M(z) = (az + b)/(cz + d)$. The imagery of these two mathematical environments evolved from hand-made pictures to the high quality offered by modern computer graphics. While Julia sets cracked to the top and became a popular topic, Kleinian groups remained on the sidelines. The reason that stopped them entering the mainstream is the entry level, far higher than it is required to display Julia sets.

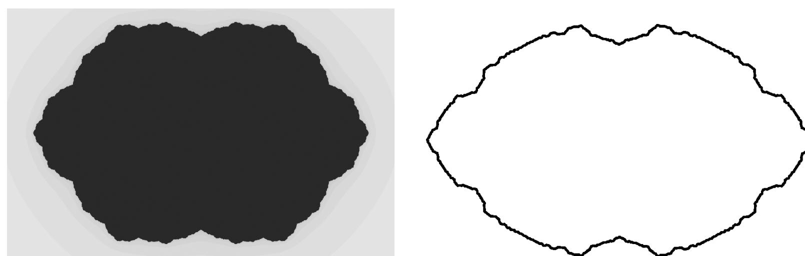


Figure 1: An example of similar shapes between Julia sets (left) and Kleinian groups (right).

1 INTRODUCTION

Both renderings basically calculate the *combinations* generated from the composition procedure being peculiar to the environments: whereas this problem is trivial for iteration as there is only one input element coming into play, namely a function $f(z)$ – combinations will be f, ff, fff, $\ldots$ – it becomes more difficult for Kleinian groups because of multiple group elements generating an infinite variety of arbitrarily long combinations: for example, the maps $M_n(z)$ in the group can be tagged by letters in order to obtain combinations like a, ab, $aaaabbb$, $abababbaaabb$; thus, this variety would intuitively suggest to use some storage tool to keep track of them all. The management of the combinations is the greatest complication that wraps any approach to the rendering of Kleinian groups. The classic method was devised long ago by Robert Fricke, Felix Klein [1] and Friedrich Schottky [5] in the end of the XIX$^{\text{th}}$ century and its design was not naturally geared to modern computational features, such as speed, efficiency, and memory resource optimization. The book *Indra's Pearls* [3] contains an in-depth presentation of this approach. The above scenario may have been too technically demanding for the non-mathematical audience, who was more concerned with displaying their shapes.[2] We attempted to lessen this issue as much as we could. The goal of this paper is to improve the implementation of an approach that we previously published in this same journal [4]. Then we will not revisit the underlying theory here; we will just resume the problem in the next section. Readers may refer to the bibliography for delving the concepts discussed further.

[1]The term 'group' refers here to a set of elements that are closed under composition and include one neutral element I. Hence, every element g is included together with its inverse g^{-1}, so that $g \circ g^{-1} = I$.

[2]While there exist many documental resources and computer programs for rendering Julia sets, only a few have been dedicated to Kleinian groups, such as 'OPTi' by Masaaki Wada and 'Kleinian' by Danny Calegary. There is also a work in progress by the author at `http://alessandrorosa.altervista.org/`

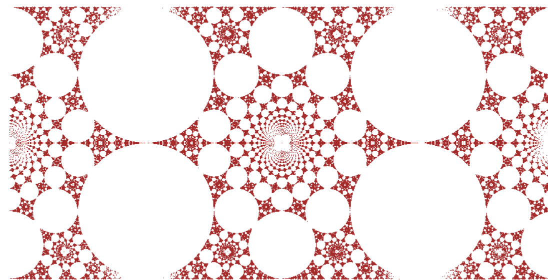


Figure 2: The limit set for a two-generators Picard group.

2 The problem

The process of building of group elements combinations up to bounded length consists first in assuming a subset of them, known as 'generators set' $\mathcal{G}$. The successive stage, where combinations are expressly produced, is extremely delicate and potentially prone to produce duplicate or null-action combinations, mainly due to the formal composition of generators and their inverses (see footnote 1); thus rigid checks are required. The classic approach binds each element of $\mathcal{G}$ to a unique alphabetical letter and runs the branching action of a tree-like structured model that keeps track of the formation of combinations, each represented by a string of letters: there is no risk of intersections between the branches and any combination shows as *only formally unique* [4, p. 54]: in fact, it shall be tested through a predefined set of rules for preventing duplicates or null-actions in practice. The run of elements in the combination is numerically equivalent to a sequence of values, known as 'orbit' in the literature of dynamical systems; orbits are proven to converge to some limit set or 'attractor'. Therefore the rendering simply reads every literal string/combination to obtain and display the equivalent numerical value.

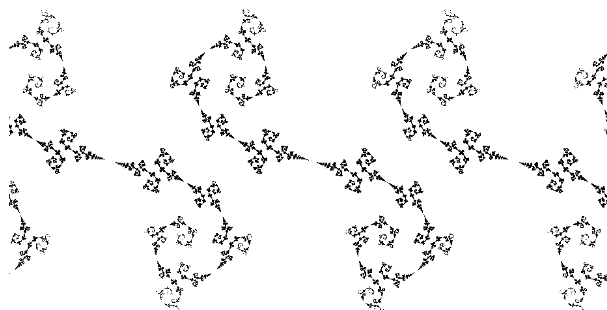


Figure 3: The limit set for a parabolic Kleinian group whose parameters lie on the boundary of the Riley slice.

High quality pictures of limit sets for Kleinian groups require a large number of combinations.[3] With regard to the classic approach, we shall consider that, though the conversion

---

`circles`.

[3]Such as hundreds of millions for the images here shown.

of each combination from letters to digits is a simple procedure that involves additional arrays that match generators with letters and the related indexes, the overall process eventually results long and laborious. Additionally, fine renderings want long combinations; and not to mention the huge memory resources that the classic approach needs to keep track of all combinations produced during the branching action of the tree. In response to these drawbacks, our approach aims to drastically reduce the amount of resources required by avoiding convoluted or unnecessary steps such dropping the tree model. The new paradigm is the *numerical base conversion* from a given integer to a base whose order matches with the generators set cardinality. This conversion simply rewrites the same numerical value as a concatenation of symbols, here digits not letters: the base conversion theorem assures the uniqueness of such string of digits ranging from 0 to the cardinality of the generators set minus one. No tracking is required: *the combination is hidden in the input number and it is unveiled and pulled out by the base conversion.*

3 The new implementation

In whatever version we intend to run the rendering of Kleinian groups, the main guideline consists in looping this block of three essential actions: (1) the *generation* of a combination, (2) its *validation*[4] and (3) *graphical rendering.*
The early version of our algorithm either relied on an auxiliary function for computing the base conversion [4, p. 58], performing a transformation from *quantitative values* to *concatenations of digits*, which represent the formal output of base conversion: it would be then inappropriate to claim that we simply convert from base 10 to base n. The run of that auxiliary function could be very time-consuming because it involves a large number of function calls in the overall economy of the rendering process, potentially lowering the specific benefits we intend to achieve through our approach.[5] The *first enhancement* affects the performance of base conversion, here tagged by `<convert-to-base>`: the aforementioned auxiliary function [4, p. 58] has been replaced by built-in function (inherently to the chosen programming language): this choice is due to its low level coding which

[4]Formally distinct combinations may refer to equivalent computations however. It is then necessary to prevent the processing of such duplicates using a set of composition rules specific to the given Kleinian group. This set could be implemented into a so called ‘Cayley Table’ or a ‘group presentation’ (also known as ‘defining relations’). The latter are writings in the form

$$\langle a, b, A, B \mid aA = Aa = bB = Bb = I\rangle. \tag{1}$$

On the left, we read the letters corresponding to the elements of $\mathcal{G}$. The transformations a, b and the related inverse A, B: this is a ‘2-generators group’, which counts elements only, not their inverse ones. On the right, we see a number of relations which shall be verified for dropping the combination or not. For theoretical details, see [2] and [3] respectively. Despite their same task, presentations cannot be simplistically intended as shorter versions of Cayley tables because the latter follows the composition of symbolic combinations step by step, which may not follow the same rules as the natural appending of consecutive symbols because of the specific algebraic nature of the group.

[5]The literature developed in the 80s and 90s referred to renderings taking several hours.

grants much faster performance than any higher level language implementation. The benefits of increased speed were immediate since the earliest tests. On this train of ideas, we shall also consider that an input number n, assumed in base 10, misses to generate all the combinations with leading zeros, which are *numerically but not computationally equivalent* to the original input value: given $n \in \mathbb{Z}$, we will also generate the family of strings '$\overline{0}n$' up to the desired length. This additional stage can be worked out by applying the same method to convert each input integer n and fill the resulting string with leading zeros up to the desired length. More technical information can be obtained directly from the pseudo-coding below.

Main code

```
1  /* We assumed a two-generators group with four elements. These are the global
2  variables required for the sub-routines too. The generators gn are objects
3  performing Möbius maps computations (i.e., composition, computation, ...) */
4
5  var _gens_objs = [ g1, g2, g3, g4 ], _gens_num = _gens_objs.length; var
6  _max_depth = 10, _max_value = power( _gens_num, _max_depth ); var _proc_str =
7  "", _zero_fill_proc_str = "", _index; var _str_length = 1, _rec_start = 0,
8  _rec_end = -1; var _b_length_change = 0, _b_crash_found = 0;
9
10 for( var _i = 0; _i < _max_value; _i++ ){
11     //base conversion and string generation
12     _proc_str = _i.<convert-to-base>( _gens_num );
13
14     //this test possibly triggers the leading zeros management
15     _b_length_change = _str_length != _proc_str.length;
16
17     //validation
18     if ( <call-to-sub-routine-#1.x: cancellation-rule-test_of_proc_str> ) continue;
19     <call-to-sub-routine-#2: process-the-numerical-string-in-base-n>
20     <call-to-sub-routine-#3: leading-zeros-management>
21 }
```

Readers will notice that this approach uses a single `for`-loop, instead of the nested two we applied in [4, p. 60]. The orbits resulting from combinations has been computed starting from one of the periodic points of the first element. The resulting strings of digits are read from right to left as it is conventionally assumed for the chains of function compositions. We will check the resulting string for compliance with the group rules[6] once the input index `_i` has been converted to the chosen numerical base. This task is resolved by `<sub-routine-#1.x>`, with regard to the Cayley Table

test via Cayley Table

```
1  //<sub-routine-#1.1>
2  function __check__cayley_table__( _digitized_word = "" ){
3      <let a boolean flag and set it to 0>
4
5      //we assume that the index has been converted into the required base
6      <split-the-digitized-word-into-an-array-of-single-digits->
7      <get-the-first-digit-in-the-word>
8      <get a reference pointer to the related row inside the table>
9
10     //we prevent to raise conditional if-statement in the loop
11     <remove the first digit from the word>
12
13     <for each digit in the of rest this array> //sequential read
```

[6]See footnote 4.

```
      <get the next-index in the current row at the index expressed by the digit>
        <if the next-index is invalid, then (1) set the above flag to 1>
        <(2) break the loop and (3) skip this word processing>
      <get a reference pointer to the row inside the-table and related to the next-index>
    <end-of-for-loop>

    <return the flag>
}
```

or via group presentation

test via group presentation

```
//<sub-routine-#1.2>
function __check__group_presentation__( _digitized_word = "" ){
    <let a boolean flag and set it to 0>

    <for each entry inside the group presentation>
        <check if the input digitized word includes the current entry>
        <if so, set the above flag to 1 and break this loop>
    <end-of-for-loop>

    <return the boolean flag>
}
```

After the validation, we will process the input string and then draw the resulting value on the complex plane:

process-the-numerical-string-in-base-n

```
//<sub-routine-#2>
//right-to-left reading order
_index = <turn-the-symbol-to-integer>( _proc_str[ _proc_str.length-1 ] );
//initialization of the fixed point _fp
_fp = _gens_objs[ _index ].get_one_fixed_point();

//process the rest of the string
for( var _wr = _proc_str.length-2; _wr >= 0; _wr-- ){
   _index = <turn-the-symbol-to-integer>( _proc_str[ _wr ] );
   _fp = _gens_objs[ _index ].map_point( _fp );
}

<call-a-sub-routine-for-drawing-the-pixel-at-the-fixed-point-coordinates>
```

Finally, we compute the family $\overline{0}$`_i` of strings with leading zeros for each index value `_i`. This is the *second enhancement* that concludes this revision; the code in the next subroutine implements again two nested `for`-loops that runs the same actions as in the main block, but it resolves strings with leading zeros exclusively.

leading-zeros-management

```
//<sub-routine-#3>
if ( _b_length_change ) //refer to the main for-loop for the role of this flag
{
  _rec_end = _n - 1;
  for( var _r = _rec_start; _r <= _rec_end; _r++ )
  {
      _zero_fill_proc_str = _r.toString( _n_gens );
      for( var _filler = _zero_fill_proc_str.length; _filler <= _max_depth; _filler++ )
      {
          _zero_fill_proc_str = "0" + _zero_fill_proc_str;
          if ( <call-to-sub-routine-#1.x: cancellation-rule-test_of_proc_str> )
              continue;
```

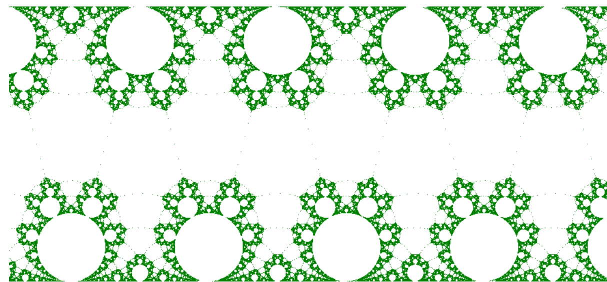


Figure 4: A limit set for a Jørgensen 2-generators group of parabolic type.

```
13
14            <call-to-sub-routine-#2.x: process-the-numerical-string-in-base-n>
15        }
16    }
17
18    _rec_start = _rec_end + 1;
19    _rec_end = -1;
20
21    //reset the flag to default state to track the next change
22    _b_length_change = 0;
23  }
```

References

[1] Fricke R., Klein F., *Vorlesungen über die Theorie der automorphen Functionen*, 1897, Leipzig, B.G. Teubner.

[2] Lyndon R.C., Schupp P.E., *Combinatorial Group Theory*, Springer, 2001.

[3] Mumford D., Series C., Wright D., *Indra's pearls: The Vision of Felix Klein*, Cambridge University Press, 2002 (reprinted in 2015).

[4] Rosa A., *A new indexed approach to render the attractors of Kleinian Groups*, Journal of Algorithms and Computation, 49, issue 2, December 2017, pp. 53–62.

[5] Schottky F., *Ueber die conforme Abbildung mehrfach zusammenhängender ebener Flächen*, Journal für die reine und angewandte Mathematik, 83, 1877, pp. 300–351.