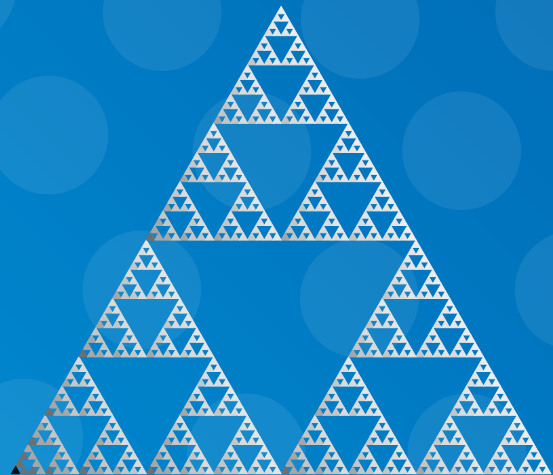
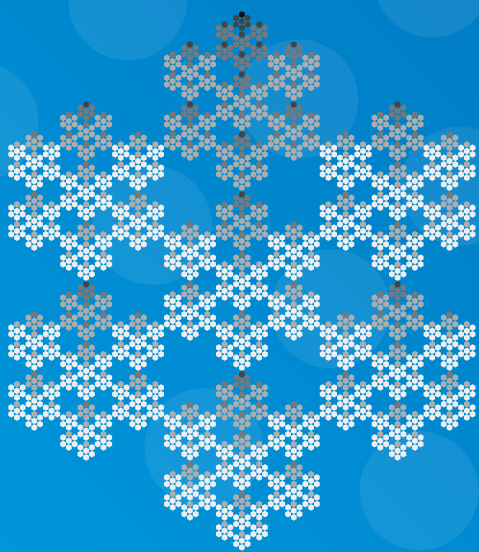
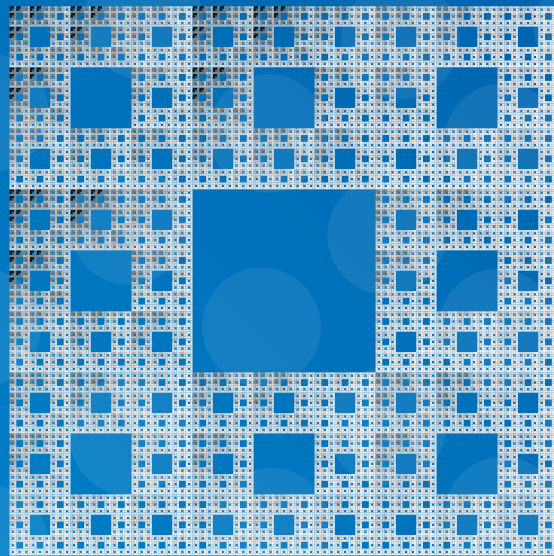


# Inhomogeneous Fractals as a Martin Boundary

Stefan Kohl





# Inhomogeneous Fractals as a Martin Boundary

Von der Fakultät Mathematik und Physik der Universität Stuttgart  
zur Erlangung der Würde eines Doktors der Naturwissenschaften  
(Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von  
**Stefan Gabriel Kohl**  
aus Göppingen

Hauptberichterin: Prof. Dr. Uta Renata Freiberg  
Mitberichter: apl. Prof. Dr. Jens Wirth  
Prof. Dr. Sze-Man Ngai

Tag der mündlichen Prüfung: 2. Dezember 2021

Institut für Stochastik und Anwendungen der Universität Stuttgart  
2022



*In beloved memory of my Mom (1950 – 2016) and my father-in-law (1960 – 2020).*



# Contents

<b>Acknowledgments</b>	<b>vii</b>
<b>Abstract</b>	<b>viii</b>
<b>Zusammenfassung</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Mathematical background</b>	<b>13</b>
2.1 Basic definitions . . . . .	13
<b>3 Martin boundary theory in context of fractals</b>	<b>23</b>
3.1 Neighboring cells on fractals . . . . .	24
3.2 Idea of the transition probability and its consequences . . . . .	31
3.3 The Martin kernel . . . . .	43
3.4 The Martin boundary . . . . .	48
3.5 The minimal Martin boundary . . . . .	52
<b>4 Reduction to a finite problem with useful properties</b>	<b>57</b>
4.1 Simplifying Condition (B2) . . . . .	58
4.2 Sierpiński gasket as the simplest example . . . . .	61
4.3 Some general facts . . . . .	66
<b>5 Investigate the essential word spaces of a fractal with two algorithms</b>	<b>81</b>
5.1 Equivalence classes of essential word spaces . . . . .	83
5.2 Calculate free weights . . . . .	86
5.2.1 With a constructive algorithm . . . . .	87
5.2.2 With an existential algorithm . . . . .	104
5.3 Summarized results of both the algorithms . . . . .	111

<b>6 Outlook and further research</b>	<b>119</b>
<b>A Source code</b>	<b>125</b>
<b>Bibliography</b>	<b>181</b>



# Acknowledgments

First and foremost I would like to thank my doctoral supervisor Prof. Dr. Uta Freiberg. She introduced me to the beautiful world of fractals with her captivating lectures and I am very grateful for all her support on every aspect during my PhD studies. Without her dedication, her trust and her faith in me this would not have been possible. I am also thankful to her for helping me develop my mathematical and personal skills.

I am very grateful being part of our working group consisting of Tim Ehnes, Elias Hauser, Nico Heizmann, Lenon Minorics and Klemens Taglieber. In particular I am thankful for many discussions on mathematics, but also non-mathematical topics and all the experiences we made together, also on several conferences in combination with nice journeys.

Moreover, I would like to thank all members of the ISA, which has always been a home for me. I received a lot of assistance in all matters during my whole PhD studies, especially by Simon Fischer.

Beside the ISA, I would like to thank the IADM and in particular Prof. Dr. Timo Weidl. It was a pleasure for me to work with him on the entrance test within the FES<sub>t</sub>4 project. I'm also thankful to Jan Köllner for the collaboration, some amusing hours while copying test booklets and all the fruitful discussions.

Furthermore, I want to express my gratitude to the whole CUS team (now ITAP team) for a very nice time while introducing C@MPUS. I acquired a lot of soft skills during this time.

Beyond that, I thank all my colleagues at the University of Stuttgart which accompanied me during all my different stations in the last five years.

I want to express my deepest gratitude to my friends and family. They supported me unconditionally in all aspects in life. On the same time it fills me with sadness that my mother and my father-in-law will no longer be able to witness this day.

Last but not least, I thank my wife Vanessa for her motivation and her encouragement during all these times and in particular for her infinite love.

# Abstract

In this thesis we consider inhomogeneous fractals and represent them as Martin boundaries. The representation of fractals by Martin boundary theory is one of the important tools for analysis on fractals, but has been so far only considered for homogeneous fractals. In particular, we examine self-similar fractals generated by iterated function systems (IFSs), where we can introduce a mass function which generates a self-similar measure on the fractal.

In the first part we introduce a Markov chain on the word space, where the word space can be associated to the fractal. We adapt the transition probabilities of the Markov chain according to the mass function. As a consequence of this, the transition probabilities can get arbitrarily small, hence the results on homogeneous fractals can not be applied, and the Martin boundary theory on fractals has to be revised. We set up three conditions, whereas (B2) is the central condition. We are able to prove that the Martin boundary of the Markov chain is homeomorphic to the weighted fractal under these conditions.

In the second part we take a closer look at Condition (B2). We reverse the condition in some sense and examine, in which cases (B2) can be fulfilled. We elaborate that the fulfillment of (B2) mainly depends on the chosen iterated function system for a fixed fractal. Further we disprove that (B2) can be induced by a well-known property of fractals like “nested” or “p.c.f.”. We give a general example which shows that (B2) is satisfied with inhomogeneous mass function for IFSs with any cardinality.

The last part of the thesis is a more practical work: we develop a computer algorithm which determines for which inhomogeneous mass functions a fractal fulfills (B2). The algorithm computes the maximum number of weights that can be chosen. To this end we give two modes of the algorithm: a constructive way and existential way. The constructive way calculates exactly which weights can be freely chosen, whereas the existential algorithm calculates the exact number only. The benefit of the existential algorithm is a shorter calculating time. We apply the algorithm to some common fractals and observe that the number of free weights varies a lot. For the sake of completeness, the complete source code is appended.

# Zusammenfassung

Wir betrachten in dieser Arbeit inhomogene Fraktale und repräsentieren diese als Martin-Ränder. Die Martin-Rand-Theorie ist ein wichtiger Zugang für Analysis auf Fraktalen, wurde jedoch bisher nur für homogene Fraktale betrachtet. Genauer betrachten wir selbstähnliche Fraktale, welche durch iterierte Funktionensysteme (IFSe) generiert werden und führen eine Gewichtsfunktion ein, welche ein selbst-ähnliches Maß auf dem Fraktal erzeugt.

Im ersten Abschnitt definieren wir eine Markovkette auf dem Wortraum, wobei der Wortraum mit dem Fraktal identifiziert werden kann. Die Übergangswahrscheinlichkeiten der Markovkette werden dabei an die Gewichtsfunktion angelehnt. Diese können dadurch beliebig klein werden, die bisherigen Resultate für homogene Fraktale sind nicht anwendbar und wir überarbeiten die Martin-Rand-Theorie auf Fraktalen unter diesem neuen Gesichtspunkt. Dazu führen wir drei Bedingungen ein, wobei (B2) von zentraler Bedeutung ist. Dabei können wir beweisen, dass der Martin-Rand unter diesen Bedingungen homöomorph zum Fraktal ist.

Im zweiten Teil untersuchen wir die Bedingung (B2) eingehender. Dabei drehen wir in gewisser Weise die Fragestellung um und untersuchen, wann (B2) erfüllt ist. Dabei erarbeiten wir, dass das Erfüllen von (B2) vom gewählten IFS abhängt, falls wir ein fixiertes Fraktal betrachten. Wir zeigen, dass (B2) nicht durch eine andere, bekannte Bedingung wie „nested“ oder „p.c.f.“ induziert wird. Wir geben zudem ein allgemeines, inhomogenes Beispiel an, welches (B2) erfüllt und IFSe von beliebiger Kardinalität zulässt.

Der letzte Teil der Arbeit ist eher praktischer Natur: Wir entwickeln einen Computeralgorithmus, welcher bestimmt, welche inhomogenen Gewichtsfunktionen auf einem Fraktal (B2) erfüllen. Der Algorithmus berechnet dabei die maximale Anzahl an inhomogenen Gewichten durch zwei unterschiedliche Modi: der konstruktive Algorithmus bestimmt dabei explizit, welche Gewichte frei gewählt werden können. Im Gegensatz dazu berechnet der existenzielle Algorithmus lediglich deren genaue Anzahl mit einer deutlich kürzeren Rechenzeit. Wir wenden abschließend beide Algorithmen auf einige bekannte Fraktale an und beobachten, dass die Anzahl an freien Gewichten sich sehr unterscheidet. Der Vollständigkeit halber ist der komplette Quellcode zum Algorithmus angehängt.



# Chapter 1

## Introduction

Nature is the most interesting and at the same time one of the most astonishing topics in this world. Almost all objects appear very smooth and elegant, but irregular and rough on closer inspection. So, for example, a trunk of a tree appears to be round. But if we take a closer look, the bark is not a smooth surface but is interspersed with grooves and unevenness. Let us explore the grooves in more detail. These grooves are not simple, straight lines, and not all grooves are equal. Every groove is unique in some sense, and the shape is harsh and crenate. Furthermore, sometimes the groove splits up into some smaller grooves which behave in the same way as the bigger grooves. So, the grooves are in some sense comparable to each other up to scaling.

We can observe such a scaling in other parts in the nature, too. This time we stay on trees and take a look at the branches. Such a branch starts at the trunk. On his way to the sky the branch splits into further branches and another branch starts. This branching follows no strict rule like, for example, a splitting every 10 cm. Nevertheless, such a branching appears with some regularity. This small branch behaves like his big ancestor: it branches out again very regularly, this time with shorter rates. It looks as if this is continued ad infinitum. However, this is not possible since the tree does not branch infinitely many times. By the way: the same structure can be observed at the roots of the tree. Such a behavior is called self-similarity, since the small branches behave like small copies of the big ones.

These phenomena are described mathematically by so-called fractals. Fractals are a very powerful tool to describe everything that is in some sense irregular and at the same time recursively structured. Classical mathematical tools fail to describe such things in an adequate way in most cases. To return to the example from above: the profile of the trunk can be described in fractal terminology as a very irregular curve which is roughly

shaped like a circle.

The first examples of fractals were introduced by mathematicians like Karl Weierstraß, Georg Cantor and Felix Hausdorff in the late 19th century and at the beginning of the 20th century. At this time, fractals served mainly as counterexamples in analysis. Later they became more popular, especially by the famous book from Benoît Mandelbrot [Man82] but also by the works of Kenneth Falconer [Fal90] and Michael Barnsley [Bar93]. For a good introduction to the topic of fractals we refer the reader to one of these books depending on the reader's intention and prior knowledge.

The concept of fractals can be extended further by a mass function. This means that some parts of the fractal have more mass than others although these parts have the same size. Such fractals are called weighted fractals here. Obviously, unweighted fractals are a special case of weighted fractals by choosing a suitable mass function. Such weighted fractals are much more suitable for modeling and we can describe the trunk from above even in the case where some parts are heavier (or more dense). This can happen, for example, due to weather influences (i.e. weather side of the tree) or other influences during the growth process of the tree.

Our main interest is the correct and appropriate representation of weighted fractals. This is indeed not an easy task, since there are a lot of fractional mathematical objects with different characteristics. Because of this, it seems to be impossible to describe all fractals in the same way. But in many cases fractals can be separated into different classes and we are able to represent fractals of such a whole class by the same methods. Such classes are for example graphs of functions like Weierstraß functions or the Brownian motion. The attractors of complex dynamical systems, so called Julia sets, are another class. The Mandelbrot set, which classifies the Julia sets, is one of the most popular fractals. In this thesis, we will not study these kind of fractals. Instead, we consider weighted fractals generated by so called iterated function systems, which we introduce in Chapter 2. Such fractals are a big and popular class of fractals and can be represented in a way which allows a deeper analysis.

Beside the different classes of fractals, there is another way to distinguish between fractals: the mathematical approach of representation. Almost all fractals are represented in a graphical way, since this gives a good first impression of the fractal (besides a nice picture). Such a representation can already indicate the basic properties of a fractal. For example, the picture of a Mandelbrot set shows the chaotic and dynamic structure of the fractal in a very illustrative way. Another possibility is a topological representation which may be more useful to determine topological or analytic properties.

A suitable representation is necessary to do analysis on fractals. This subject is of high interest within the fractal community, since it allows the investigation of fractals in a physical context. This includes the study of the diffusion of heat through the heat equation

$$\frac{\partial u}{\partial t} = \Delta u.$$

In this case, the function  $u = u(t, x)$  models the temperature at time  $t$  and position  $x$  and  $\Delta$  is the Laplacian on the fractal. The classical definition of a Laplacian fails on fractals due to multiple reasons, which means that the concept of a Laplacian on fractals needs to be revised.

One approach to do this, is the consideration of harmonic functions, which get mapped to zero by the Laplacian. These harmonic functions can be simulated numerically based on the method of averages. These simulations can provide a very good first impression of a Laplacian. In particular, we simulate harmonic functions with appropriate boundary values. This is an easy and joyful task and Figure 1.1 shows three simulations. In Figure 1.1(a) we consider the Sierpiński gasket with uniform weights. The left part of the figure shows the Sierpiński gasket and the gray color indicates the mass distribution. All weights are equal and thus all shades of gray are equal. The right part shows the simulated harmonic function. The boundary points are the three vertices of the big triangle. In this example we choose the boundary values to be 0 at the bottom left, 0.5 at the top and 1 at the bottom right. The coloring is in accordance to a temperature scale: blue means low values, red means high values and a color of green or yellow corresponds to a value in-between.

In contrast to this, we consider the weighted Sierpiński gasket in Figure 1.1(b). The weight at the top equals 0.31, at the bottom left side 0.31 and at the bottom right 0.38. Again, the left part shows the weighted Sierpiński gasket. The different cells have different weight and therefore a different shades of gray. We choose for the harmonic function the same boundary values as in Figure 1.1(a). If we compare the harmonic functions in Figures 1.1(a) and 1.1(b), we see that the mass function has an influence on the distribution of the heat. We deduce from this observation that the mass function has a great impact on harmonic functions (and hence on the Laplacian).

In addition to this we simulate a harmonic function on the Sierpiński carpet, shown in Figure 1.1(c). We consider the inhomogeneous Sierpiński carpet and choose a weight of 0.01 at the bottom left side, a weight of 0.24 at the top right side and a weight of 0.125 at all other of the eight cells. The right part of Figure 1.1(c) shows the weights of each cell and lighter colors correspond to lighter weight. We can simulate a harmonic function on

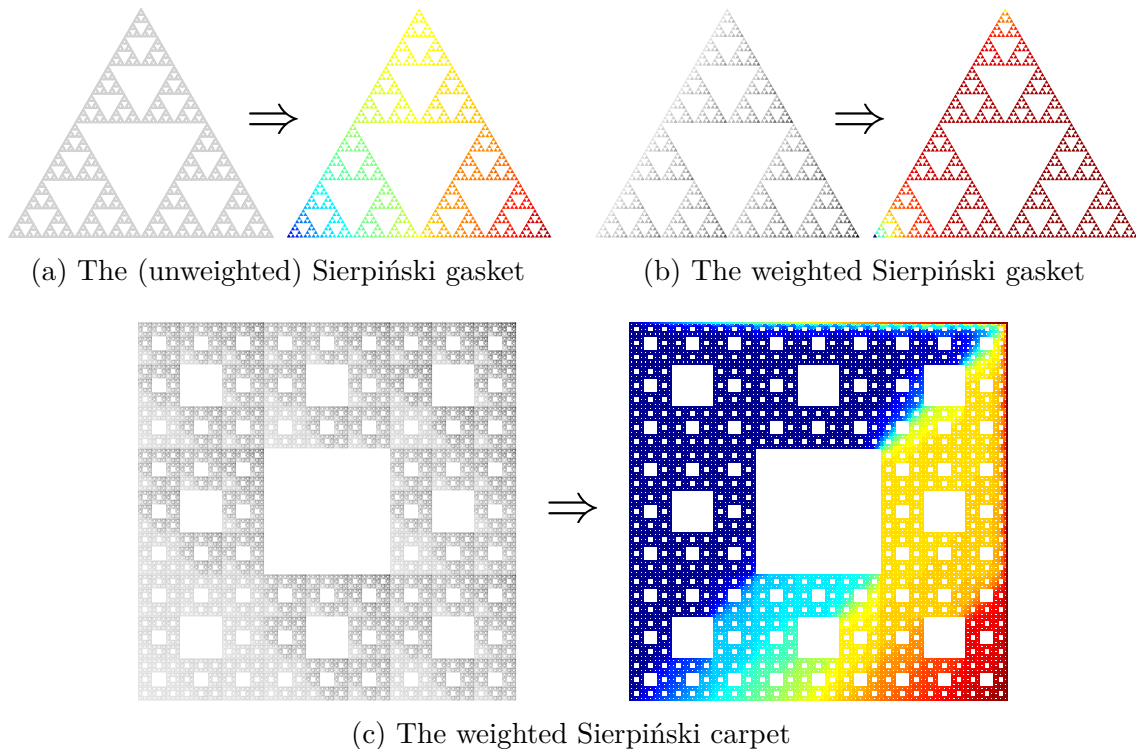


Figure 1.1: Some simulations to harmonic functions on two common fractals.

this weighted Sierpiński carpet, where the boundary is the boundary of the surrounding square, see [Bar+10]. We choose the function  $f(x, y) = x$  as the boundary function. The influence of the mass function is clearly visible, since the coloring in the homogeneous case would be blue to red from left to right, independent of the  $y$ -coordinate.

In the literature there are basically four different classical approaches to define a Laplacian on fractals. We will briefly introduce each approach and list the advantages and disadvantages of each.

The first way is given by using Brownian motion. In this case we consider a sequence of graphs  $F_n$  approximating the fractal  $F$ . Then we define a random walk on  $F_n$ . Those random walks converge, under suitable renormalization, to a Brownian motion on  $F$ . The infinitesimal generator of this Brownian motion is then the Laplacian on  $F$ . This was first done by Goldstein [Gol87] and Kusuoka [Kus87] and later by Barlow and Perkins [BP88] for the Sierpiński gasket. Lindstrøm generalized this to nested fractals in [Lin90]. Barlow and Bass considered Brownian motion on the Sierpiński carpet in [BB89] and harmonic analysis on the Sierpiński carpet in [BB99].

The approach allows the consideration of nested fractals and also of (at least some) non-p.c.f. fractals like the Sierpiński carpet, where p.c.f. stands for post critically finite. We



will introduce the term p.c.f. later in detail, see Definition 2.1.1. At the same time this approach has been only used for unweighted fractals and it is hard to calculate some harmonic functions explicitly.

The second and very common approach is done by graph approximation. For this, we fix a certain fractal and a sequence of graphs  $(V_n)$  approaching the fractal. We define a discrete Laplacian  $\Delta_n$  as a difference operator on  $(V_n)$ , where the difference operator is thereby defined over neighboring vertices. This discrete Laplacians converges under a suitable renormalization to a Laplacian on  $F$ . This was introduced by Kigami in [Kig89] for the Sierpiński gasket and extended in [Kig93] to p.c.f. self-similar sets. Among other things, this approach allows to study the spectral asymptotics of this Laplacian. This was firstly done by Fukushima and Shima in [FS92] and later by Kigami and Lapidus in [KL93]. The whole approach was generalized by Kigami in [Kig01] using Dirichlet forms  $(\mathcal{E}_n)$  on  $(V_n)$ . If these Dirichlet forms are compatible to each other,  $(\mathcal{E}_n)$  converge to a Dirichlet form on  $F$ , which gains access to the Laplacian on  $F$ .

By construction, the Laplacian depends on a measure on the fractal. At the same time we can only consider p.c.f. self-similar sets, which excludes for example the Sierpiński carpet from Figure 1.1(c).

The third approach uses function spaces and was introduced by Triebel in [Tri97] and others. This approach includes weighted fractals, but generates a Laplacian which cannot compared directly to the two previous and the following approaches.

The last approach is done via Martin boundary theory, which we will follow later. For now we take a closer look at this idea and also at the history of Martin boundary theory.

The general idea of the Martin boundary was first introduced by Martin in [Mar41] and was then extended by Doob in [Doo59] and Hunt in [Hun60] by investigating the behavior of Markov chains in the limit. Further articles and books [Doo01; Dyn69; KSK76; Saw97; Woe00; Woe09] followed this idea and tied the connection to harmonic analysis. Thereby, we consider the word space  $\mathcal{W}$ , which represents the mathematical object and harmonic functions  $h$  on  $\mathcal{W}$ , which are invariant under applying the Markov operator on  $h$ . These harmonic functions should not be confused with the harmonic functions that are introduced by numerical simulations, which are defined in another way. The harmonic functions have an integral representation by

$$h(w) = \int_{\mathcal{M}} k(w, \xi) d\mu(\xi), \quad w \in \mathcal{W},$$

where  $\mathcal{M}$  is the Martin boundary (based on  $\mathcal{W}$ ),  $k$  is the Martin kernel and  $\mu$  is a Borel measure supported on  $\mathcal{M}$ .

Denker and Sato came up with the idea to describe fractals through Martin boundary theory. In several papers [DS99; DS01; DS02] they studied the description of the Sierpiński gasket and proved that the Sierpiński gasket is homeomorphic to the Martin boundary of a Markov chain on the corresponding word space. They defined so called strongly harmonic functions and an analogue of the Laplacian on the Martin boundary. They compared their results with the graph approximation introduced by Kigami and showed that both definitions of harmonic functions coincide. Another fractal represented as Martin boundary was the Pentagasket, which was done by Imai in [Ima02]. The idea was picked up by Lau and co-authors in [JLW12; LN12]. Lau and Wang extended in [LW15] the Martin boundary theory to fractals satisfying five assumptions on the Markov chain and the OSC.

The question arises whether we can extend this idea to fractals that are less regular as in the case of a non-isotropic modification of the Markov chain. This was done by Kesseböhmer, Samuel and Sender in [KSS20] for the Sierpiński gasket, firstly published in 2017 on arXiv. They modified the transition probability of the Markov chain in such a way that there exist two different transition probabilities and showed that the Martin boundary is still homeomorphic to the fractal.

In this thesis we extend the Martin boundary theory to weighted fractals, including non-p.c.f. fractals. This closes the gap between the graph approximation approach, which allows weighted p.c.f. fractals and the Brownian motion approach, which allows non-p.c.f. fractals, but only if they are not weighted. Our approach is thereby applicable to all weighted, self-similar fractals that are generated by iterated function systems fulfilling the OSC and three newly introduced conditions (A), (B1) and (B2). The condition (B2) is introduced for a compatibility of the self-similarity and the mass function. This is the first part of the thesis. In the second part we consider some properties of (B2), and in the third part we verify (B2) on some common fractals with the help of a computer algorithm.

At the beginning, we introduce all required basic terminologies in Chapter 2. This includes the definition of iterated function systems and the representation of fractals by an alphabet  $\mathcal{A}$  and the word space  $\mathcal{W}$ . We also define the p.c.f.-property of a fractal and further introduce nested fractals, which form a specific subclass of p.c.f.-fractals. Then, we introduce the (in-)homogeneous mass function on the alphabet. For the mass function on the word space we first consider a Bernoulli measure and then a self-similar measure on the fractal. The mass function on  $\mathcal{W}$  is then set up in Definition 2.1.8. We close the chapter with two plots of some inhomogeneous fractals, in particular the Sierpiński gasket and the Sierpiński carpet.

After the necessary preparations, we consider the Martin boundary theory in Chapter 3.

We start with the consideration of fractals and the terminology of cells. Cells are basically the parts which build up the pre-fractals and have an one-to-one correspondence to finite words  $w$  from the word space  $\mathcal{W}$ . These cells can be visualized in a very illustrative way. Based on this we introduce in Definition 3.1.1 a relation  $\sim$ , which clarifies in which case two words are said to be equivalent. In fact, this relation need not to be an equivalence relation. Nevertheless we will speak of two equivalent words. We clarify in Remark 3.1.2 that the property of p.c.f. does not imply that the relation is an equivalence relation. At the same time we are able to show in Proposition 3.1.8 that for all nested fractals the introduced relation is always an equivalence relation. Therefore, we introduce the condition

**(A)** The relation  $\sim$  is an equivalence relation,

which allows us later to refer to this fact.

In the next step we introduce a transition probability  $p$  on the word space. This transition probability, defined in Definition 3.2.1, is based on the considered mass function on the fractal. It is one of the key components of this work, since this definition has an influence on the whole theory developed within this thesis. With the help of  $p$  we define a Markov operator  $P$  which acts on the word space  $\mathcal{W}$  (resp. on the fractal). For a very good visual representation we refer to Figure 3.3, which shows the action of  $p$  in a very nice way.

In Remark 3.2.4 we discuss the case of a homogeneous mass function and the consequences for our definition of the transition probability  $p$ . We compare the literature with the transition probability in the homogeneous case and see that  $p$  is comparable with other definitions in the literature, especially with the work of Lau and Wang in [LW15]. In this comparison, we also note that the results of [LW15] are not applicable to the introduced transition probability in the inhomogeneous case. Therefore, we need to revisit the Martin boundary theory for the inhomogeneous case.

We define the Green function  $g(v, w)$ , which is basically the  $n$ -step transition probability between two words  $v$  to  $w$ . We are able to prove that the  $n$ -step transition probability from the empty word to a certain word equals exactly the mass of the word. This justifies in some sense the correct definition of  $p$ . At the same time we are unable to calculate the Green function in an explicit way for arbitrary  $v$  and  $w$ . But we can notice a specific pattern in  $g$ , which we extract into the function  $q$ . We set up two conditions:

**(B1)** The Martin kernel in the homogeneous case exists.

**(B2)** For all  $w \in \mathcal{W}$  either

$$m(w) = m(\tilde{w}), \quad \forall \tilde{w} \sim w,$$

or

$$w^- \sim (\tilde{w})^-, \quad \forall \tilde{w} \sim w,$$

holds.

Here,  $w^-$  describes the word  $w$  where the last letter is dropped.

A nice consequence of (B2) is the fact that we are able to calculate  $q$  in a recursive way. This can be done independently of the mass function  $m$ , see Theorem 3.2.12. We define the Martin kernel  $k$  in Definition 3.3.1, which is in some sense the renormalized Green function. The Martin kernel can then be computed in the inhomogeneous case by the Martin kernel of the homogeneous case by using  $q$ . This is the key to calculate the Martin boundary for inhomogeneous fractals. Roughly speaking, this allows to apply the results of [LW15] to inhomogeneous fractals. For this, we introduce a Martin metric  $\rho$  on  $\mathcal{W}$ . Based on  $\rho$  we can set up the  $\rho$ -completion of  $\mathcal{W}$ , which is the Martin space  $\overline{\mathcal{W}}$ . The Martin boundary  $\mathcal{M}$  is then in some way the difference between the Martin space and the word space, namely

$$\mathcal{M} := \overline{\mathcal{W}} \setminus \mathcal{W}.$$

After some technical lemmata we are finally able to prove in Corollary 3.4.5 that the Martin boundary is homeomorphic to the fractal  $K$ , i.e.

$$K \cong \mathcal{W}^* / \sim \cong \mathcal{M}_{\text{hom}} = \mathcal{M}.$$

We conclude the chapter with the so called minimal Martin boundary  $\mathcal{M}_{\text{min}}$ , which is also known as “space of exits”. We prove that the minimal Martin boundary coincides with the (regular) Martin boundary.

Chapter 4 is dedicated to a deeper understanding of the required pre-conditions (A), (B1) and (B2). We discuss that the conditions (A) and (B1) can not be replaced. As a consequence of this we consider (B2). First of all we investigate the case where the condition (B2) is fulfilled by all words automatically. This is the first step to isolate the quintessence of (B2). Then we investigate when (B2) is not fulfilled automatically by all words and set up the framework that must hold for those special words. It turns out that this has something to do with the mass function and moreover with the equivalence of the parental words.

Based on these observations we define the so called essential word space  $\mathfrak{W}$ , which is a subset of the word space. One of the benefits of the essential word space  $\mathfrak{W}$  is the fact, that  $\mathfrak{W}$  is usually finite. In contrast to this, the word space  $\mathcal{W}$  is always infinite. The finiteness of  $\mathfrak{W}$  makes it easier to check condition (B2). In fact, we show in Theorem 4.1.4 that we only have to check for all words  $w \in \mathfrak{W}$  whether they satisfy

$$m(w) = m(\tilde{w}) \quad \forall \tilde{w} \sim w,$$

which is basically the first part of (B2).

We consider in Section 4.2 the classical Sierpiński gasket and apply the results of the previous section to it. Further, we consider the fact that the Sierpiński gasket can be generated by different iterated function systems. This follows from the fact that small copies can be rotated and possibly reflected. This leads to a whole collection of iterated function systems with the same attractor. These different IFSs generate different word spaces, which also lead to different essential word spaces. This has a big impact on Condition (B2) and on the mass function, respectively. We elaborate for the Sierpiński gasket different cases, where the mass function fulfills (B2) only if a certain number of weights are fixed. This especially includes an iterated function system which fulfills (B2) only with a homogeneous mass function.

The example of the Sierpiński gasket motivates the search for further examples. We set up the terminology of a “minimal fractal example”. Of course, this example should be not a “pathological” example. This means that we exclude cases like a straight line, which can be in fact described as an attractor of an IFS but is no fractal in the common sense. The minimization is related to the length of the alphabet. We find a minimal example which fulfills (B2) and one example, which can fulfill (B2) only with a homogeneous mass function. Both examples generate the Koch curve.

The minimal example raises the question of a maximal fractal example fulfilling (B2) with an inhomogeneous mass function. To be precise, we maximize in this case according to the number of equivalent words. First, we build up an example with three equivalent words and an inhomogeneous mass function. This can be achieved with the so called 3–level Sierpiński gasket in Example 4.3.4. In the second step we build up a general example with  $n$  equivalent words where  $n \geq 4$ . This example has an inhomogeneous mass function and fulfills on the same time (B2). The key to this example is mainly a center with  $n$  copies meeting in one point. We orientate these copies in the right way and prove that an inhomogeneous mass function fulfills (B2).

In the third part, Chapter 5, we develop a computer algorithm. This is motivated

by the observation that a general answer to the question, which fractals fulfill (B2), seems to be impossible. Additionally, we have shown in the previous Chapter 4 that it depends more on the particular iterated function system itself than on the fractal, if (B2) is fulfilled. Because of this, we develop a computer algorithm, which basically fixes an (fractal) attractor and varies the iterated function systems. The algorithm determines which iterated function system allows an inhomogeneous mass function and how many weights can be chosen independently of each other. This calculation can be done for every IFS separately.

The algorithm can be applied to many fractals. However, in this thesis we only consider a few examples to demonstrate the application of the algorithm and the results. In a first step we reduce the number of iterated function systems that have to be considered. The purpose of this is the observation that a calculation to a certain iterated function system needs a relatively long computation time. We can cluster different iterated function systems in equivalence classes with the same number of free weights and we will later calculate only the number of free weights to one representative of each equivalence class. Such an equivalence class consists of all iterated function systems that can be generated from each other by rotation or reflection. We will call this step preprocessing, since it is performed before the main task. This preprocessing reduces the number of cases by approximately 90%. However, a general factor cannot be given, since this depends on the fractal.

The main part of Chapter 5 is the calculation of the number of free weights. We give two different methods of calculating this number. The first method is a constructive way, see Subsection 5.2.1. Here, the algorithm determines exactly which weights can be chosen and give those back to the user. At the beginning the algorithm sets up some equations which have to be fulfilled. We introduce some (common) methods to reduce the equations and collect these in Method 5.2.6, which states how and when these equations can be simplified. After this we formulate how the number of free parameters corresponds to these equations. Using the implicit function theorem, we can calculate the exact free parameters in Theorem 5.2.9. We extend the algorithm to the case where the implicit function theorem is not directly applicable. Finally we give two examples, where we exercise the algorithm by hand.

In contrast to this constructive algorithm, the algorithm in Subsection 5.2.2 only considers the existence of the exact number of free weights. The idea for this existential algorithm comes from the extension of the constructive algorithm. We formulate the problem in mathematical words again and apply the implicit function theorem to the problem in Theorem 5.2.15. Further we give an example of the algorithm applied to a special

realization of the filled Hexagasket, also known as Lindstøm snowflake. The complete implementation of both the algorithms is provided in Appendix A.

We consider at the end of Chapter 5 the (complete) algorithm applied to some common fractals. This includes especially the Sierpiński gasket. Further, we consider the 3–level and 4–level Sierpiński gasket, the Vicsek fractal, the (filled) Pentagasket, the (filled) Hexagasket, the three–dimensional Sierpiński tetrahedron and the Sierpiński carpet. We compare the results between the fractals as well as within the constructive and existential algorithm.

Finally, in Chapter 6, we provide a brief outlook on open problems and further fields of study.

This thesis is partially based on the following papers (see [FK19] and [Koh20]):

- U. Freiberg, S. Kohl, *Martin boundary theory on inhomogeneous fractals*, Preprint, 2019.
- S. Kohl, *Topological conditions on inhomogeneous fractals in Martin boundary theory and their algorithmic testing*, Preprint, 2020.





# Chapter 2

## Mathematical background

### 2.1 Basic definitions

In this thesis we consider fractals which are generated by iterated function systems, sometimes also called iterated function schemes. Iterated function systems (IFSs) are a very powerful and convenient way to represent and characterize a wide class of fractals. These fractals consist of smaller copies of itself on a non-empty closed subset  $D \subseteq \mathbb{R}^d$ . Such a small copy is generally the image of a contraction and is denoted by

$$S_i : D \rightarrow D \text{ with } |S_i(x) - S_i(y)| \leq c_i|x - y|,$$

where  $c_i \in (0, 1)$  is a contraction ratio and  $i$  denotes the particular number of the contraction.

An iterated function system (IFS) is a family of contractions

$$\{S_1, \dots, S_N\}$$

on  $D$  with  $N \geq 2$  finite. A nice property of such an IFS is the fact that by Hutchinson's theorem [Hut81] there exists a unique non-empty compact invariant subset  $K \subset \mathbb{R}^d$  satisfying

$$K = S(K) := \bigcup_{i=1}^N S_i(K).$$

The set  $K$  is called attractor (or invariant set) of the IFS  $\{S_i\}_{i=1}^N$  and we assume for the whole thesis that  $K$  is connected. If  $K$  is not connected, we can still do the whole calculus, but it would be quite uninteresting.

In this thesis we only consider the special case where the contractions have uniform

contraction ratio. Such special contractions are called similarities (by some authors similitudes) and are characterized by

$$S_i : D \rightarrow D \text{ with } |S_i(x) - S_i(y)| = c_i|x - y| \quad \forall x, y \in D \text{ and for some } c_i \in (0, 1).$$

The attractors of IFSs consisting of similarities are called self-similar and build a wide class of fractals. Beside self-similar fractals there are also self-affine attractors which are generated by a family of affine mappings. But we will only examine self-similar fractals.

Additionally we assume for the whole thesis that the IFS satisfies the open set condition, abbreviated by OSC. The OSC states that there exists a non-empty bounded open set  $\mathcal{O} \subset \mathbb{R}^d$  such that  $\bigcup_{i=1}^N S_i(\mathcal{O}) \subset \mathcal{O}$  with the union disjoint.

We now introduce the so called word space which sometimes is called code space. The word space allows us to handle fractals which are generated by an IFS in a topological (and very elegant) way. The main idea is to identify images of  $K$  under composed  $S_i$  by a single word (also called code) which consists of multiple letters. Then, each letter represents a single mapping.

In detail we consider a certain IFS  $\{S_1, \dots, S_N\}$  and define the corresponding alphabet

$$\mathcal{A} := \{1, \dots, N\}.$$

The letter  $i \in \mathcal{A}$  corresponds to the mapping  $S_i$ . We have by the open set condition  $S_i(\mathcal{O}) \subseteq \mathcal{O}$  and thus the “small image” is a subset of the whole fractal. At the same time the fractal can be decomposed into multiple cells and because of this we will sometimes also use the term “cell” for a letter or word.

We tie up the connection between image and letter under composed mappings. Therefore, we define the word space  $\mathcal{W}$  by

$$\mathcal{W} := \bigcup_{n \geq 1} \mathcal{A}^n \cup \{\emptyset\}$$

where  $\emptyset$  is the empty word. The connection to composed mappings is established by

$$S_{w_1}(\dots(S_{w_n}(E))) = S_{w_1} \circ \dots \circ S_{w_n}(E) =: S_w(E)$$

for  $E \subset \mathbb{R}^d$  and  $w = w_1 \dots w_n \in \mathcal{W}$ . A cell  $w$  has thereby the form  $S_w(K)$ .

We denote by  $\mathcal{W}^* = \mathcal{A}^{\mathbb{N}}$  the set of all infinite  $\mathcal{A}$ -valued sequences  $w = w_1 w_2 \dots$  and by  $w|_n = w_1 \dots w_n$  the restriction to the first  $n$  letters of  $w \in \mathcal{W}^*$ . For any  $n \in \mathbb{N}_0$ , the function  $\cdot|_n$  assigns to every sequence  $w \in \mathcal{W}^*$  a word in  $\mathcal{W}$ . For any  $w = w_1 w_2 \dots w_n \in \mathcal{W}$

we define the reverse assignment  $\mathcal{W}_w^* := \{v = v_1v_2\cdots \in \mathcal{W}^* : v_1v_2\cdots v_n = w_1w_2\cdots w_n\}$ . This set is often called cylinder set.

For  $w = w_1\cdots w_n \in \mathcal{W}$  we define  $\tau(w) := w_n$  which is the last letter of the word  $w$ , the parent of  $w$  by  $w^- := w_1\cdots w_{n-1}$ , a child of  $w$  by  $wa$  with  $a \in \mathcal{A}$  and the length of  $w$  by  $|w| = n$ . For two words  $v, w \in \mathcal{W}$  we define  $d(v, w) := |w| - |v|$ . This function is thereby no metric, but gives us the difference of the length of two words. The function can be negative, if the word  $w$  is shorter than  $v$ .

The product  $vw$  of two words  $v = v_1v_2\cdots v_m \in \mathcal{W}$  and  $w = w_1w_2\cdots w_n \in \mathcal{W}$  is defined by

$$vw := v_1v_2\cdots v_mv_1w_2\cdots w_n,$$

and for the empty word  $\emptyset$  it holds that  $|\emptyset| = 0$  and  $w\emptyset = \emptyset w = w$  for any  $w \in \mathcal{W}$ .

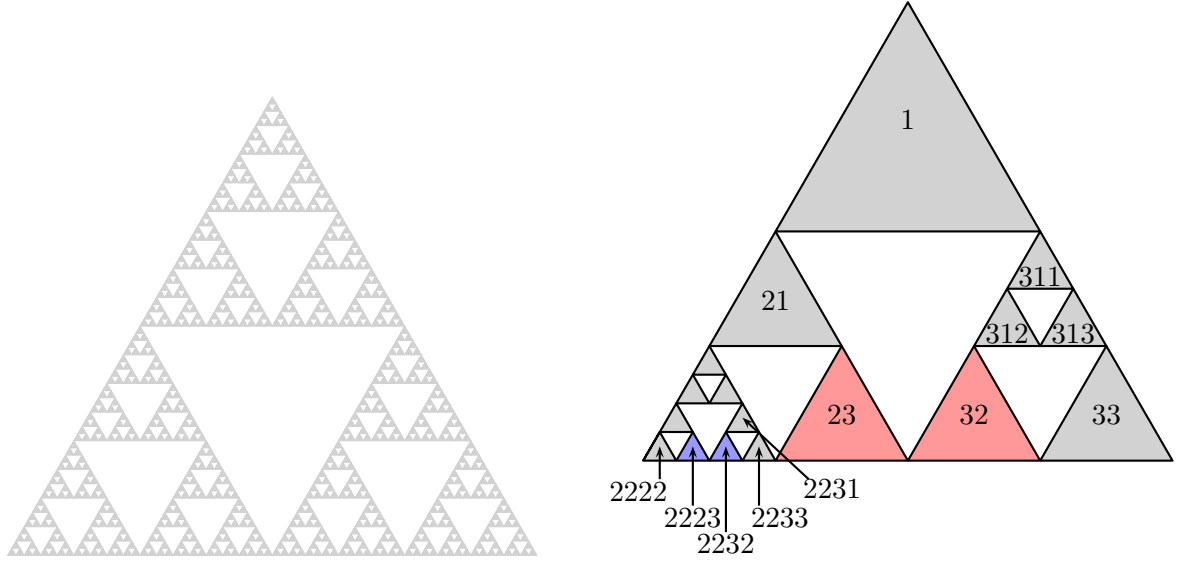
We apply this notation to the Sierpiński gasket, which is a textbook example of a fractal. The Sierpiński gasket is constructed by the IFS consisting of the three similarities

$$\begin{aligned} S_1 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{2}y + \left(\frac{1}{4}, \frac{\sqrt{3}}{4}\right)^T \\ S_2 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{2}y \\ S_3 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{2}y + \left(\frac{1}{2}, 0\right)^T, \end{aligned}$$

usually applied to an equilateral triangle. The attractor  $K$  of this IFS is displayed in Figure 2.1(a), whereas Figure 2.1(b) depicts some words  $w \in \mathcal{W}$ , where the coloring comes into account later, when we will consider equivalent words.

A lot of recent articles and books consider special types of fractals. This comes from the fact that such fractals can be handled in much more detail. Kigami, for example, developed in [Kig01] a whole analytical calculus for so called p.c.f. fractals (see definition below) and introduced a Laplacian on such fractals. The book of Strichartz [Str06], which discusses also differential equations on (p.c.f.) fractals, aims in the same direction. Furthermore, Ju, Lau and Wang also considered p.c.f. fractals in context of Martin boundary theory (see [JLW12]). For this reason we introduce two common classes of fractals: post critically finite fractals and nested fractals.

**Definition 2.1.1** (p.c.f. structure, following [Kig01, Chapter 1.3]). *Let  $K$  be a compact metrizable topological space,  $\mathcal{A} = \{1, \dots, N\}$  with  $N < \infty$  and  $S_i : D \rightarrow D, i = 1, \dots, N$*



(a) The attractor  $K$  of the Sierpiński gasket. (b) Different cells of the Sierpiński gasket. The coloring gets relevant in a later part of the thesis.

Figure 2.1: The attractor and some cells of the Sierpiński gasket.

contractions. Further we define the right shift operators

$$\sigma_i : \mathcal{W}^* \rightarrow \mathcal{W}^*, \sigma_i(w_1 w_2 w_3 \dots) = i w_1 w_2 w_3 \dots$$

for all  $i \in \mathcal{A}$  and the left shift operator

$$\sigma : \mathcal{W}^* \rightarrow \mathcal{W}^*, \sigma(w_1 w_2 w_3 \dots) = w_2 w_3 \dots$$

The triple  $\mathcal{L} = (K, \mathcal{A}, \{S_i\}_{i \in \mathcal{A}})$  is called self-similar structure if there exists a continuous surjection

$$\pi : \mathcal{W}^* \rightarrow K \text{ such that } S_i \circ \pi = \pi \circ \sigma_i$$

holds for every  $i \in \mathcal{A}$ .

Additionally we define the critical set

$$\mathcal{C} := \pi^{-1} \left( \bigcup_{\substack{i, j \in \mathcal{A} \\ i \neq j}} (S_i(K) \cap S_j(K)) \right) \subset \mathcal{W}^*$$

(depending on  $\pi$ ) and the post critical set

$$\mathcal{P} := \bigcup_{n \geq 1} \sigma^n(\mathcal{C}) \subset \mathcal{W}^*$$

(also depending on  $\pi$ ). The self-similar structure  $\mathcal{L}$  is then called post critically finite (or *p.c.f.* for short) if and only if the post critical set  $\mathcal{P}$  is finite.

Another very common class of fractals is based on similar ideas, but topologically motivated.

**Definition 2.1.2** (nested fractal, [Ham00, Def. 2.1]). Let  $\{S_1, \dots, S_N\}$  be a IFS with similitudes  $S_i : D \rightarrow D, i = 1, \dots, N$  and attractor  $K$ . A 1-cell is defined as  $S_i(K)$  for  $i = 1, \dots, N$ . We denote by  $F'_0 := \{q_i : S_i(q_i) = q_i\}$  the set of all fixed points of the similarities  $S_i$ . Further we define the set of all essential fixed points  $F_0$  by  $F_0 := \{x \in F'_0 : \exists i, j \in \mathcal{A}, y \in F'_0, x \neq y \text{ such that } S_i(x) = S_j(y)\}$ . A fractal  $K$  is then called nested, if it satisfies:

1. *Connectivity:* For any 1-cells  $C$  and  $C'$ , there is a sequence  $\{C_i : i = 0, \dots, n\}$  of 1-cells such that  $C_0 = C, C_n = C'$  and  $C_{i-1} \cap C_i \neq \emptyset, i = 1, \dots, n$ .
2. *Symmetry:* If  $x, y \in F_0$ , then reflection in the hyperplane  $H_{xy} = \{z : |z - x| = |z - y|\}$  maps  $S^n(F_0)$  to itself.
3. *Nesting:* If  $v, w \in \mathcal{W}$  with  $v \neq w$ , then

$$S_v(K) \cap S_w(K) = S_v(F_0) \cap S_w(F_0)$$

4. *Open set condition OSC:* There is a non-empty, bounded, open set  $\mathcal{O}$  such that the  $\{S_i(\mathcal{O})\}$  are pairwise disjoint and  $\bigcup_{i=1}^N S_i(\mathcal{O}) \subseteq \mathcal{O}$ .

In this thesis, we will not restrict ourselves to either of these classes. Nevertheless, we show in Proposition 3.1.8 that nested fractals have some nice properties and are relatively easy to handle.

In the following we consider fractals and self-similar measures at the same time. To do so, we first introduce some basic notations from measure theory. Then, we introduce a mass function  $m$  on the alphabet  $\mathcal{A}$ . This induces the so called Bernoulli measure  $\mu^m$  on  $\mathcal{W}^*$ . With help of the Bernoulli measure we are able to define a self-similar measure  $\nu^m$  on the attractor  $K$ . This self-similar measure depends on the mass function  $m$ , which is the reason for the superscript in the notation of the measure.

The rest of this section is roughly based on [Kig01, §1.4], since Kigami introduces the self-similar measure with high accuracy and in a very structured way. Nevertheless we use our own notation and expand it according to our requirements.

We start with the definition of a measure and some properties.

**Definition 2.1.3** (measurable space and measure). *The tuple  $(\Omega, \mathfrak{A})$  is called measurable space if  $\Omega$  is a non-empty set and  $\mathfrak{A}$  is a  $\sigma$ -algebra consisting of subsets of  $\Omega$ .  $\mu$  is called measure on the measurable space  $(\Omega, \mathfrak{A})$  if  $\mu$  is a non-negative,  $\sigma$ -additive function defined on  $\mathfrak{A}$ .*

**Definition 2.1.4** (Borel  $\sigma$ -algebra, [Kig01, Def. 1.4.1]). *Let  $(\Omega, d)$  be a metric space. Further let  $\mu$  be a measure on a measurable space  $(\Omega, \mathfrak{A})$ .*

1. *The Borel  $\sigma$ -algebra  $\mathcal{B}(X, d)$  is the minimal  $\sigma$ -algebra which contains all open sets of  $\Omega$  and an element of  $\mathcal{B}(X, d)$  is called a Borel set. If no confusion can occur, we write  $\mathcal{B}(X)$ .*
2. *The measure  $\mu$  is called a Borel measure if  $\mathcal{B}(X) \subset \mathfrak{A}$ .*
3. *The measure  $\mu$  is called a Borel regular measure if it is a Borel measure and for any  $A \in \mathfrak{A}$  exists a  $B \in \mathcal{B}(X)$  such that  $A \subseteq B$  and  $\mu(A) = \mu(B)$ .*
4. *The measure  $\mu$  is called complete if any subset of a null set is measurable. In other words it must hold that  $B \in \mathfrak{A}$  if  $B \subseteq A \in \mathfrak{A}$  and  $\mu(A) = 0$ .*
5.  *$\mu$  is called a probability measure if and only if  $\mu(X) = 1$ .*

**Definition 2.1.5** (mass function). *The function  $m : \mathcal{A} \rightarrow (0, 1)$  defined on an alphabet  $\mathcal{A} = \{1, \dots, N\}$  with  $N$  finite is called mass function (on  $\mathcal{A}$ ), if*

$$\sum_{a \in \mathcal{A}} m(a) = 1$$

*holds. The value  $m(a)$  is also called weight of a for  $a \in \mathcal{A}$ .*

*The mass function is called homogeneous, if  $m(a) = \frac{1}{N}$  holds for all  $a \in \mathcal{A}$ . Otherwise the mass function is called inhomogeneous.*

The mass function is one of the key elements in this thesis. We especially discuss the inhomogeneous mass function and its consequences to fractals and their representation as Martin boundaries. But for now, we continue with the general measure theory and introduce a measure on  $\mathcal{W}^*$  which is based on a mass function  $m$ .

**Proposition 2.1.6** (Bernoulli measure, cf. [Kig01, Prop. 1.4.3]). *Consider a mass function  $m$  on  $\mathcal{A} = \{1, \dots, N\}$ . Then there exists an unique complete Borel measure  $\mu^m$  on  $(\mathcal{W}^*, \mathfrak{A}^\mu)$  that satisfies*

$$\mu^m(\mathcal{W}_w^*) = m(w_1)m(w_2) \cdot \dots \cdot m(w_n) \tag{2.1}$$

for any  $w = w_1 w_2 \dots w_n \in \mathcal{W}$ .

The measure  $\mu^m$  is called the *Bernoulli measure* on  $\mathcal{W}^*$  with mass function  $m$ .

The Bernoulli measure  $\mu^m$  on  $\mathcal{W}^*$  with mass function  $m$  can also be defined as the unique regular probability measure satisfying

$$\mu^m(A) = \sum_{a \in \mathcal{A}} m(a) \mu^m(\sigma_a^{-1}(A)) \quad (2.2)$$

for any Borel set  $A \subset \mathcal{W}^*$ .

Based on this Bernoulli measure we can define a measure on the attractor  $K$  of the IFS. This links the measure on  $\mathcal{W}^*$  and the geometric object.

**Theorem 2.1.7** (self-similar measure, cf. [Kig01, Prop. 1.4.4]). *Consider the self-similar structure  $\mathcal{L} = (K, \mathcal{A}, \{S_i\}_{i \in \mathcal{A}})$  with a continuous surjection  $\pi : \mathcal{W}^* \rightarrow K$ . Further we consider a mass function  $m$  on  $\mathcal{A}$ .*

We define  $\nu^m$  by

$$\nu^m(A) = \mu^m(\pi^{-1}(A))$$

for  $A \in \mathfrak{A}^\nu = \{A : A \subseteq K, \pi^{-1}(A) \in \mathfrak{A}^\mu\}$ , which is a  $\sigma$ -algebra. It follows that  $\nu^m$  is a Borel regular measure on  $(K, \mathfrak{A}^\nu)$  and  $\nu^m$  is called the self-similar measure on  $K$  with mass function  $m$ .

The self-similar measure satisfies a similar property as the Bernoulli measure on  $\mathcal{W}^*$  in (2.2). To be more precise, the measure  $\nu^m$  is the unique Borel regular measure on  $K$  satisfying

$$\nu^m(A) = \sum_{i \in \mathcal{A}} m(i) \nu^m(S_i^{-1}(A))$$

for any Borel set  $A \subset K$ . For a proof we refer the reader to the original source [Hut81, Theorem 4.4(1)] or to [Fal97, Theorem 2.8].

We revisit the mass function and extend its definition to all words  $w \in \mathcal{W}$ . This is consistent to the idea that the weight splits up to smaller cells in the same way as we split up the weight on the alphabet.

**Definition 2.1.8** (extended mass function, inhomogeneous fractal). *We extend the mass function  $m$  on  $\mathcal{A}$  to all words  $w = w_1 w_2 \dots w_n \in \mathcal{W}$  by*

$$m(w) := \mu^m(\mathcal{W}_w^*) = m(w_1) m(w_2) \cdot \dots \cdot m(w_n).$$

For the empty word  $\emptyset$  we set  $m(\emptyset) = 1$  which preserves the multiplicative structure of  $m$ . If no confusion can occur, we will drop the preposition “extended”.

We call the pair  $(\mathcal{W}, m)$  inhomogeneous, if we consider a word space together with an inhomogeneous mass function, otherwise we call the pair homogeneous. If no confusion can occur, we call the word space inhomogeneous, if we consider  $(\mathcal{W}, m)$  with an inhomogeneous mass function.

Further we call the fractal inhomogeneous, if we consider a fractal with an inhomogeneous mass function. Otherwise the fractal is called homogeneous fractal or just fractal.

This definition of  $m$  immediately implies that

$$\sum_{a \in \mathcal{A}} m(wa) = m(w) \quad (2.3)$$

holds for all  $w \in \mathcal{W}$ . Further the multiplicity of the mass function

$$m(vw) = m(v)m(w) \quad (2.4)$$

holds for any  $v, w \in \mathcal{W}$ . Both equations (2.3) and (2.4) will be used excessively in this thesis.

**Remark 2.1.9.** Normally, the terminology “homogeneous fractal” is used with an other meaning for the mass function and describes a fractal, where each part has the same density. This is induced by the mass function defined by

$$m(i) = c_i^d \text{ for } i \in \mathcal{W}, \quad (2.5)$$

where  $c_i$  is the contraction ratio of the similarity  $S_i$  and  $d$  the Hausdorff dimension of the fractal. In this case it follows that  $\nu$  is the normalized,  $d$ -dimensional Hausdorff measure. For an iterated function system  $\{S_1, \dots, S_N\}$  with contraction ratios  $c_i \equiv c \in (0, 1)$  and OSC follows by (2.5) for the mass function

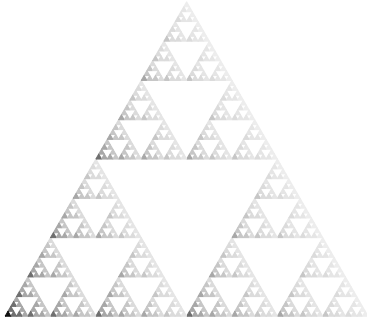
$$m(i) = \frac{1}{N},$$

since  $c_i^d = c^d = \frac{1}{N}$  hold. This coincide with our definition in 2.1.5.

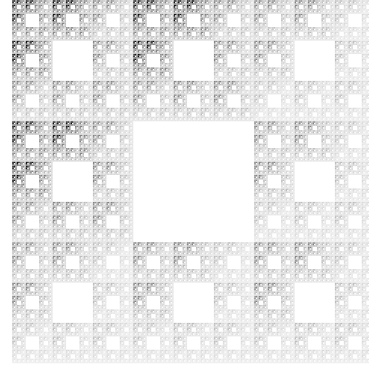
This comes from the fact that the dimension  $d$  solves the equation  $\sum_i^N c_i^d = 1$ . In the case of  $c_i \equiv c \in (0, 1)$  follows then  $d = \frac{\ln N}{-\ln c}$ .

**Remark 2.1.10** (probabilistic IFS). The mass function can also be interpreted in a probabilistic way: the weight  $m(a) \in (0, 1)$  represents the probability to choose the mapping  $S_a$ , for example if we plot the IFS with a random algorithm. This leads to so called





(a) The weighted Sierpiński gasket.



(b) The weighted Sierpiński carpet.

Figure 2.2: Two examples of fractals with a mass function. A darker color means more weight/mass, a brighter color corresponds to less weight/mass.

*probabilistic iterated function schemes. For further information about (probabilistic) iterated function schemes we refer the reader to [Fal90; Fal97].*

We can illustrate inhomogeneous fractals, if we plot the attractor of the IFS. Then we add shades of gray which correspond to the weight of each cell. Of course, cells with the same weight should get the same color. In Figure 2.2 are two examples of inhomogeneous fractals (in fact these are only pre-fractals). In these plots heavier cells are painted in darker color, lighter cells are painted brighter. The Sierpiński gasket has a weight of 0.5 in the bottom left triangle and the two other triangles have a weight of 0.25. The Sierpiński carpet has a mass function where the three upper left squares have a weight of 0.2 and all other squares have a weight of 0.08.



# Chapter 3

## Martin boundary theory in context of fractals

In this chapter we develop a connection between a fractal and the Martin boundary of a Markov chain. This idea was first considered by Denker and Sato in [DS99; DS01; DS02] for the Sierpiński gasket. A group around Ka–Sing Lau extended this idea in several papers (see [LN12; JLW12; LN14; LW17]) and finally for fractals which fulfill the open set condition beside some other conditions in [LW15].

We add a new aspect to this theory: the mass function on a fractal. So far this has not been done in the context of Martin boundary theory and the results of the previous papers are not applicable since they only considered fractals without a mass function.

The main idea is to adjust the transition probability of the Markov chain according to the mass function. For this purpose, we start with some basic ideas of neighboring cells in Section 3.1. In this section we identify words which are basically the same, but have a different coding. These equivalent words describe in the finer structure the same geometrical point. As it turns out, this can be very challenging since the corresponding relation is in general not an equivalence relation. Luckily, for all nested fractals it is an equivalence relation.

In Section 3.2 we introduce the modified transition probability with respect to the mass function. Further, we introduce at the same time the typical notation of the Martin boundary theory for fractals. We compare the modified transition probability with the literature and notice that the results of [LW15] can not be applied. This implies that we consider the Martin boundary theory from this new point of view. For this, we establish two Assumptions (B1) and (B2). Assumption (B1) is quite harmless and states that we only consider the inhomogeneous Martin boundary if the homogeneous Martin kernel

exists. Since the homogeneous fractal can also be understood as inhomogeneous fractal with mass function  $m(a) = \frac{1}{N}$  for  $a \in \mathcal{A}$  the assumption (B1) is in fact no limitation. The situation is completely different for (B2). This assumption is very difficult to treat, hence we continue with the Martin boundary theory and take a closer look at (B2) in Chapter 4.

In Section 3.3 we define the Martin kernel  $k$  and observe some useful properties of  $k$ . An essential part of this section (and for the whole thesis) is Theorem 3.3.3, which gives us the opportunity to calculate the Martin kernel in the inhomogeneous case in terms of the Martin kernel of the homogeneous case. Based on  $k$  we are able to define a Martin metric  $\rho$  on the word space, which enables us to define in Section 3.4 the Martin boundary. In Theorem 3.4.4 we show that the Martin boundaries in the inhomogeneous and the homogeneous case are equal. From this it follows that the inhomogeneous fractal is homeomorphic to the Martin boundary of the homogeneous case.

### 3.1 Neighboring cells on fractals

In this first step of the whole Martin boundary theory we identify different cells as equivalent. This comes from the idea that these cells represent the same geometric object. This should also hold for longer words and in particular if the geometric distance of two equivalent words gets small.

In the literature several definitions for equivalent words in context of Martin boundary are considered, see for example [DS01; LW15]. All those definitions have some advantages and disadvantages and our aim is to combine all their advantages. The result is the following definition which specifies when two words are said to be equivalent. Thereby it is indeed not always guaranteed that we consider an equivalence relation. Since we consider only the case where this is an equivalence relation (see Condition (A)), we hold on the term “equivalent”.

**Definition 3.1.1.** *The words  $v, w \in \mathcal{W}$  are said to be equivalent, denoted by  $v \sim w$ , if and only if  $|v| = |w|$ ,  $S_v(K) \cap S_w(K) \neq \emptyset$  and  $v^- \neq w^-$ . Additionally, we define that  $v$  is equivalent to itself, i.e.  $v \sim v$  holds for all  $v \in \mathcal{W}$ .*

*For  $v, w \in \mathcal{W}^*$  with  $v = v_1v_2\dots$  and  $w = w_1w_2\dots$  we extend this relation such that  $v \sim w$  if and only if there exists  $n_0 \in \mathbb{N}$  such that  $v|_n \sim w|_n$  holds for all  $n \geq n_0$ .*

*Further, we define the number of equivalent words of  $w \in \mathcal{W}$  by  $R(w) := \#\{v \in \mathcal{W} : v \sim w\}$ .*

For a better understanding illustrates Figure 2.1 some equivalent words highlighted by the same color.

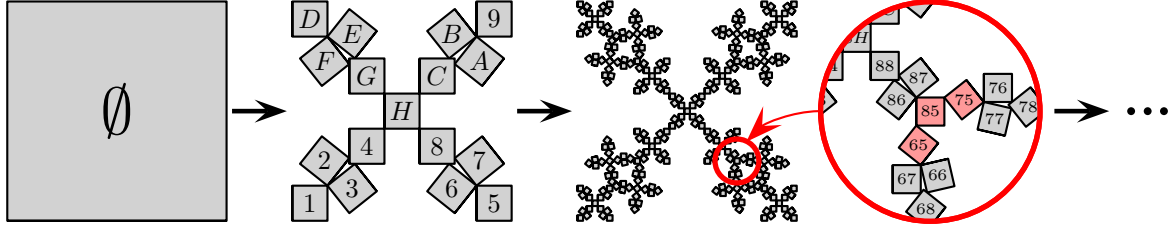


Figure 3.1: Construction of the Vicsek-snowflake in the first two steps and a part of the word space of length 2. In red are three cells highlighted, which disprove that our relation is always transitive.

**Remark 3.1.2.** *Definition 3.1.1 seems to have the disadvantage that it does not guarantee that our relation  $\sim$  is transitive. It is a natural question, if this could be induced by some other, more common condition like p.c.f., see Definition 2.1.1. Unfortunately, this is not the case.*

We construct an example to prove this. The fractal is generated as in Figure 3.1. This IFS contains 17 similarities and for better understanding we define the alphabet as  $\mathcal{A} = \{1, \dots, 9, A, \dots, H\}$ . Each similarity has a contraction ratio of  $c \approx 0.1601$ , which is the real solution of

$$-c^2 + c(5 + \sqrt{2 - c^2}) - 1 = 0.$$

This fractal satisfies the open set condition and is p.c.f.. Since the fractal has somehow the structure of the Vicsek fractal, we call this fractal the Vicsek-snowflake.

We consider the cells 65, 85 and 75, which are highlighted in Figure 3.1. It holds that the intersection of the cells 65 and 85 is non-empty. Furthermore, the intersection of 85 and 75 is non-empty and the parents of the three words are different. Thus, it follows by Definition 3.1.1 that  $65 \sim 85$  and  $85 \sim 75$ . On the other hand it holds that  $S_{65}(K) \cap S_{75}(K) = \emptyset$  and thus  $65 \not\sim 75$ , which shows that the relation is not transitive. This fractal is therefore a counterexample that the transitivity is not induced by p.c.f..

The following three lemmata are very useful tools which state some (more or less) basic facts about fractals fulfilling the OSC. In particular, we examine that the number of equivalent words is finite and that a similarity only contains exactly one fixed point.

**Lemma 3.1.3.** *If the fractal  $K$  fulfills the OSC, then  $R(v) < \infty$  for all  $v \in \mathcal{W}$ .*

*Proof.* This follows directly by [BK91, Prop. 11]. □

As a first observation we consider some properties of the fixed points  $q_i$ .

**Lemma 3.1.4** ([BK91, Corollary in §9]). *Let  $S_1, \dots, S_N$  be an IFS with OSC and attractor  $K$ . Let  $q_i$  be the fixed point of  $S_i$ . Then  $q_i$  belongs exactly to one  $S_j(K)$  with  $j = i$ .*

**Lemma 3.1.5.** *Let  $S_1, \dots, S_N$  be an IFS with OSC and attractor  $K$ . Let  $q_i$  be the fixed point of  $S_i$ . If  $q_i = q_j$  holds, then  $i = j$  follows.*

*Proof.* Let  $i, j \in \{1, \dots, N\}$  and assume that  $q_i = q_j$ . By Lemma 3.1.4 it follows that  $q_i \in S_i(K)$  and at the same time  $q_j = q_i \in S_i(K)$  holds. Using again Lemma 3.1.4,  $i = j$  follows.  $\square$

We now analyze the intersection of some cells. The first lemma considers the intersection with a child, the second lemma the intersection of two cells of words with the same length.

**Lemma 3.1.6.** *Let  $u \in \mathcal{W}$  and  $a \in \mathcal{A}$ . Then, it holds that the cell  $S_{ua}(K)$  contains only one element from  $S_u(F_0)$ , namely  $S_u(q_a)$ . In other words:*

$$S_{ua}(K) \cap S_u(F_0) = S_u(q_a).$$

*Proof.* Let us consider  $q_k \in F_0'$ . It holds that  $S_k(q_k) = q_k$  is true and consequently  $S_u(q_k) = S_{uk}(q_k)$  holds, hence

$$S_{ua}(K) \cap S_u(q_k) = S_{ua}(K) \cap S_{uk}(q_k) \tag{3.1}$$

follows.

Our goal is to show that (3.1) is empty for  $k \neq a$  and consists of one point, if  $k = a$ . So let us take a look at those two cases.

We first consider  $k = a$ . It follows that

$$S_{ua}(K) \cap S_{uk}(q_k) = S_{ua}(K) \cap S_{ua}(q_a) = \{x\}$$

holds and therefore the intersection consists of one single point.

Now let us consider  $k \neq a$ . Assume that (3.1) is non-empty. In particular we assume that

$$\{y\} = S_{ua}(K) \cap S_{uk}(q_k) = S_{ua}(p) \cap S_{uk}(q_k)$$

with  $p \in K$ . Since  $ua \neq uk$  holds by assumption and our fractal is nested, we conclude that  $p \in F_0$  must hold by the nesting property.

Further, we observe that in general  $S_{ua}(F_0) \cap S_{vb}(F_0) \subseteq S_u(F_0) \cap S_v(F_0)$  holds. Using this it follows that

$$\begin{aligned} \emptyset \neq S_{ua}(K) \cap S_{uk}(K) &= S_{ua}(F_0) \cap S_{uk}(F_0) \subseteq S_u(F_0) \cap S_u(F_0) = \\ &= S_u(F_0) = \{S_u(q_{b_1}), \dots, S_u(q_{b_n})\} = \{S_{ub_1}(q_{b_1}), \dots, S_{ub_n}(q_{b_n})\} \end{aligned}$$

holds, where  $n = |F_0|$  and  $b_i \in F_0$  with  $b_i \neq b_j$  for  $i \neq j$ .

Let us denote

$$\{y_m\} = S_{ua}(q_{c_m}) \cap S_{uk}(q_{d_m}) \quad \text{for some } m$$

with  $q_{c_m}, q_{d_m} \in F_0$ . At the same time  $\{y_m\} = S_{ub_m}(q_{b_m})$  holds.

Therefore it follows that  $c_m = a$  and  $d_m = k$  holds, or more precisely  $S_{ua}(q_a) = S_{uk}(q_k)$ . We can reformulate this into

$$S_u(q_a) = S_{ua}(q_a) = S_{uk}(q_k) = S_u(q_k)$$

and see that  $q_a = q_k$  must hold. By Lemma 3.1.5 it follows that  $a = k$ , which contradicts our assumption.  $\square$

**Lemma 3.1.7.** *For nested fractals and words  $u, v \in \mathcal{W}$  with  $|u| = |v|$ ,  $u \neq v$  and  $u^- \neq v^-$  it holds that  $S_u(K) \cap S_v(K)$  consists of at most a single point.*

*Proof.* Let us only consider words  $u, v \in \mathcal{W}$ , where  $S_u(K) \cap S_v(K) \neq \emptyset$  holds. As a first step we note that the intersection consists of points, since

$$S_u(K) \cap S_v(K) = S_u(F_0) \cap S_v(F_0) =: \{x_1, \dots, x_n\}$$

holds true by the nesting property.

We now consider  $u, v \in \mathcal{W}$  with  $|u| = |v|$ ,  $u \neq v$  and  $u^- \neq v^-$ . We denote the intersection points by  $x_i$  and therefore, we can represent them as

$$x_i = S_u(q_{a_i}) \cap S_v(q_{b_i})$$

where  $q_{a_i} \neq q_{a_j}$  and  $q_{b_i} \neq q_{b_j}$  for  $i \neq j$  must hold.

At the same time we can represent this point by

$$x_i = S_{u^-}(q_{A_i}) \cap S_{v^-}(q_{B_i}),$$

again with  $q_{A_i} \neq q_{A_j}$  and  $q_{B_i} \neq q_{B_j}$  for  $i \neq j$ .

Consequently it follows that  $S_u(q_{a_i}) = x_i = S_{u^-}(q_{A_i})$ . We can now apply Lemma 3.1.6 and it follows that  $A_i = \tau(u)$  for all  $i$  and hence  $n \leq 1$  follows.  $\square$

We can use the previous statements and prove the following proposition.

**Proposition 3.1.8.** *For nested fractals it holds that the relation  $\sim$  defined in Definition 3.1.1 is transitive and therefore defines an equivalence relation.*

*Proof.* It is obvious that  $\sim$  is reflexive and symmetric.

The only critical part is the transitivity of  $\sim$ . Therefore we show that  $u \sim v$  and  $v \sim w$  imply  $u \sim w$ . For the sake of understanding we study  $uU \sim vV$  and  $vV \sim wW$  with  $u, v, w \in \mathcal{W}$  and  $U, V, W \in \mathcal{A}$  and show  $uU \sim wW$ .

Without loss of generality we consider  $uU, vV, wW$  pairwise different, since otherwise the statement is trivial.

Let us now prove that  $uU \sim wW$  holds. Thus we have to prove that  $|uU| = |wW|$ ,  $S_{uU}(K) \cap S_{wW}(K) \neq \emptyset$  and  $u \neq w$ .

The first part is very easy, since  $|uU| = |vV| = |wW|$  holds.

In the second step we show that

$$S_{uU}(K) \cap S_{wW}(K) = S_{uU}(F_0) \cap S_{wW}(F_0) \neq \emptyset$$

since the fractal is nested. It holds by Lemma 3.1.7 that

$$S_{uU}(K) \cap S_{vV}(K) = \{x_1\}$$

since  $uU \sim vV$  and consequently  $u \neq v$ . Further, it holds that

$$x_1 \in S_u(K) \cap S_v(K) = S_u(F_0) \cap S_v(F_0).$$

Or in other words:  $x_1 \in S_u(F_0)$  and  $x_1 \in S_{uU}(K)$ . By Lemma 3.1.6 it follows that

$$x_1 = S_u(q_U) = S_{uU}(K) \cap S_u(F_0), \quad (3.2)$$

and in the same way

$$x_1 = S_v(q_V) = S_{vV}(K) \cap S_v(F_0).$$

On the other hand it follows with the same argumentation for  $S_{vV}(K) \cap S_{wW}(K) = \{x_2\}$  that  $x_2 = S_v(q_V) = S_{vV}(K) \cap S_v(F_0)$  and

$$x_2 = S_w(q_W) = S_{wW}(K) \cap S_w(F_0) \quad (3.3)$$

hold. In particular we obtain  $x_2 = S_v(q_V) = x_1$  and we conclude that

$$x_1 = x_2 \in S_{uU}(F_0) \cap S_{wW}(F_0) = S_{uU}(K) \cap S_{wW}(K)$$

holds. Hence the intersection of  $S_{uU}(K)$  and  $S_{wW}(K)$  is non-empty.



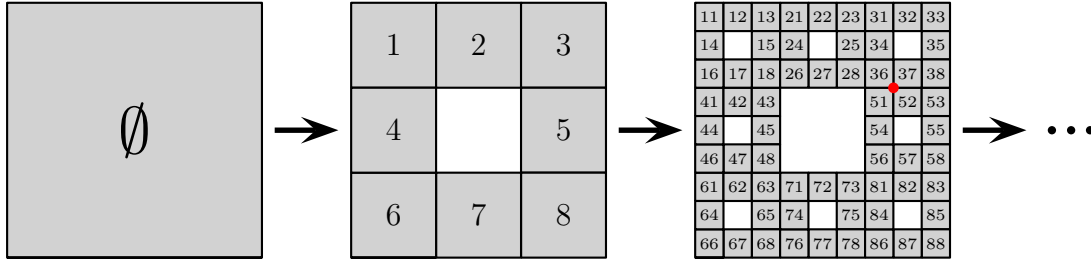


Figure 3.2: Construction of the Sierpiński carpet in the first two steps and words up to length 2. The red point highlights the intersection of the cells 36 and 52.

In our last part we prove that  $u \neq w$  holds. Therefore, we assume  $u = w$  and deduce a contradiction. Since  $u = w$  must hold, it follows that  $U \neq W$ , otherwise we would have  $uU = wW$ . We know by Equation (3.2) and (3.3) that

$$S_u(q_U) = x_1 = x_2 = S_w(q_W) = S_u(q_W)$$

holds, where the last equality follows by  $u = w$ . This implies that  $q_U = q_W$  and by Lemma 3.1.5  $U = W$  must hold, which contradicts our assumptions. Consequently  $u \neq w$  follows and we obtain in general that  $\sim$  is transitive.  $\square$

**Remark 3.1.9.** *A general definition of  $\sim$  such that  $\sim$  is transitive is quite complicated or maybe impossible. In some cases it can be necessary to adapt the definition of  $\sim$  to the IFS.*

*For example, if we take a short look at the Sierpiński carpet depicted in Figure 3.2, we can see that the definition regarding 3.1.1 will not form an equivalence relation. This is due to the fact that  $S_{36}(K) \cap S_{52}(K)$  is non empty (highlighted by the red point). It seems unnatural that  $36 \sim 52$  should hold. Instead it should hold that 36, 51 and 28 are equivalent words. The definition of  $\sim$  can be adapted to the Sierpiński carpet which we will do in the following Example 3.1.10.*

*The three words 28, 36 and 51 provide another interesting fact. It holds that the dimension of these intersections have different Hausdorff dimension:  $\dim_H(S_{28}(K) \cap S_{36}(K)) = 1$ , but  $\dim_H(S_{28}(K) \cap S_{51}(K)) = 0$ . It is possible, to adjust the transition probability with respect to this fact, see Chapter 6. By now it is unclear, if this has an effect and what this effect is.*

**Example 3.1.10** (Modified relation for SC). *The previous Remark 3.1.9 discussed the fact that the relation  $\sim$  is applicable for the Sierpiński carpet, but not in a satisfying way. For this reason we introduce in this example an alternative definition of  $\sim$ . We are*

unable to do this in an universal notation (otherwise we could obtain a better, more general definition of  $\sim$ ) and we define “by hand” which words should be equivalent. Of course, this definition follows mainly the idea of the original relation and we will use the same symbol  $\sim$  to describe the equivalence of words.

In this example we consider the Sierpiński carpet in the classical way. This means that the similarities have neither a flipping nor a rotation. The IFS consists of the similarities

$$\begin{aligned}
S_1 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{3}y + \left(0, \frac{2}{3}\right)^T \\
S_2 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{3}y + \left(\frac{1}{3}, \frac{2}{3}\right)^T \\
S_3 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{3}y + \left(\frac{2}{3}, \frac{2}{3}\right)^T \\
S_4 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{3}y + \left(0, \frac{1}{3}\right)^T \\
S_5 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{3}y + \left(\frac{2}{3}, \frac{1}{3}\right)^T \\
S_6 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{3}y \\
S_7 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{3}y + \left(\frac{1}{3}, 0\right)^T \\
S_8 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, & y &\mapsto \frac{1}{3}y + \left(\frac{2}{3}, 0\right)^T
\end{aligned}$$

and Figure 3.2 illustrates the word space graphically.

We consider the word space  $\mathcal{W}$  in the usual way. First we say that  $\sim$  is transitive. Then we define, for  $u \in \mathcal{W}$  and  $k \geq 1$ , the relation  $\sim$  as given in Table 3.1. This list of equivalent words should for example be read as follows. Let us consider

$$u15^k \sim u27^k \sim u83^k. \quad (3.4)$$

This relation states, when we choose  $u = \emptyset$  and  $k = 1$ , that the words 15, 27 and 83 are equivalent. At the same time the words 444155, 444277 and 444833 are equivalent, if we choose  $u = 444$  and  $k = 2$ . We can of course also choose other words  $u \in \mathcal{W}$  and  $k \geq 1$  in (3.4). All other relations can be read in the same way.

The so defined relation is transitive (by definition) and forms an equivalence relation. This later allows us to investigate the Sierpiński carpet in terms of the Martin boundary.

We can use a modified version of Table 3.1 for an IFS where some similarities have a rotation or a flipping.

$u13^k \sim u21^k,$	$u23^k \sim u31^k,$
$u17^k \sim u81^k,$	$u35^k \sim u43^k,$
$u87^k \sim u71^k,$	$u45^k \sim u53^k,$
$u75^k \sim u67^k,$	$u65^k \sim u57^k,$
$u15^k \sim u27^k \sim u83^k,$	$u25^k \sim u37^k \sim u41^k,$
$u85^k \sim u73^k \sim u61^k,$	$u63^k \sim u47^k \sim u51^k,$
$u135^k \sim u143^k \sim u217^k \sim u281^k,$	$u145^k \sim u153^k \sim u287^k \sim u271^k,$
$u235^k \sim u243^k \sim u317^k \sim u381^k,$	$u245^k \sim u253^k \sim u387^k \sim u371^k,$
$u175^k \sim u167^k \sim u813^k \sim u821^k,$	$u165^k \sim u157^k \sim u823^k \sim u831^k,$
$u875^k \sim u867^k \sim u713^k \sim u721^k,$	$u865^k \sim u857^k \sim u723^k \sim u731^k,$
$u375^k \sim u367^k \sim u413^k \sim u421^k,$	$u365^k \sim u357^k \sim u423^k \sim u431^k,$
$u475^k \sim u467^k \sim u513^k \sim u521^k,$	$u465^k \sim u457^k \sim u523^k \sim u531^k,$
$u735^k \sim u743^k \sim u617^k \sim u681^k,$	$u745^k \sim u753^k \sim u687^k \sim u671^k,$
$u635^k \sim u643^k \sim u517^k \sim u581^k,$	$u645^k \sim u653^k \sim u587^k \sim u571^k.$

Table 3.1: The equivalence relation defined on the Sierpiński carpet.

**Assumption.** *The Remarks 3.1.9 and 3.1.2 as well as Proposition 3.1.8 show that  $\sim$  can be, depending on the definition of  $\sim$ , an equivalence relation or not. This is an essential part of the thesis and we make the following assumption for the rest of the thesis:*

(A) *The relation  $\sim$  is an equivalence relation.*

This assumption is of great significance and is a geometric property. We can consider multiple IFSs which are comparable to each other in the sense, that they generate the same attractor but one or more similarity has a rotation or flipping. This geometric property means that we can verify (A) for all comparable IFSs if we can verify (A) for one specific IFS. So, for example, we can rotate one or more cells in Figure 2.1(b) and we already know that the relation is still an equivalence relation.

## 3.2 Idea of the transition probability and its consequences

As a first step we define a Markov chain on  $\mathcal{W}$ . In order to do so, we have to specify a transition probability  $p$  on  $\mathcal{W} \times \mathcal{W}$ . Our goal is to define the transition probabilities from

a cell to its children with respect to the mass function on the alphabet  $\mathcal{A}$ .

Therefore we entertain the idea that the probability of going from  $v$  to its child  $w$  should be equal to the quotient of the weight of  $w$  and the weight of the word we start from, which is the weight of  $v$ . This means that we consider

$$p'(v, w) = \frac{m(w)}{m(v)} \quad \text{where } w \text{ is a child of } v. \quad (3.5)$$

The problem of this definition is that we do not obtain a probability, since in general we have  $\sum_{w \in \mathcal{W}} p'(v, w) \neq 1$ . We therefore scale Equation (3.5) and get:

$$p(v, w) = \frac{p'(v, w)}{\sum_{x \in \mathcal{W}} p'(v, x)} \quad (3.6)$$

We now describe conditions under which  $w$  is a child of  $v$ . First, if we have  $v \in \mathcal{W}$ , then the words  $vi$  with  $i \in \mathcal{A}$  should be children of  $v$ . Second, if we have an equivalent word  $\tilde{v}$  from  $v$  we identify  $\tilde{v}$  and  $v$  as the same. Therefore the children of  $\tilde{v}$ , namely by the first thought  $\tilde{v}i$ , should also be children of  $v$ .

In total we get that all children of  $v$  are of the form  $w = \tilde{v}i$  with  $\tilde{v} \sim v$  and  $i \in \mathcal{A}$ .

Then, using Equation (3.5), it follows for Equation (3.6):

$$\begin{aligned} p(v, w) &= \frac{\frac{m(w)}{m(v)}}{\sum_{\tilde{v} \sim v} \sum_{i \in \mathcal{A}} \frac{m(\tilde{v}i)}{m(v)}} \\ &= \frac{m(w)}{\sum_{\tilde{v} \sim v} \sum_{i \in \mathcal{A}} m(\tilde{v})m(i)} \\ &= \frac{m(w)}{\sum_{\tilde{v} \sim v} m(\tilde{v}) \sum_{i \in \mathcal{A}} m(i)}. \end{aligned}$$

Since  $m$  is a mass function on  $\mathcal{A}$ , it holds that  $\sum_{i \in \mathcal{A}} m(i) = 1$ . So it follows:

$$p(v, w) = \frac{m(w)}{\sum_{\tilde{v} \sim v} m(\tilde{v})}.$$

In the other case, when  $w$  is not a child of  $v$ , we set  $p(v, w) = 0$ .

This motivates the following definition.

**Definition 3.2.1.** We define the transition probability  $p : \mathcal{W} \times \mathcal{W} \rightarrow [0, 1]$  by

$$p(v, w) := \begin{cases} \frac{m(w)}{\sum_{\hat{v} \sim v} m(\hat{v})}, & \text{if } w = \hat{v}i \text{ where } \hat{v} \sim v \text{ and } i \in \mathcal{A}, \\ 0, & \text{else.} \end{cases}$$

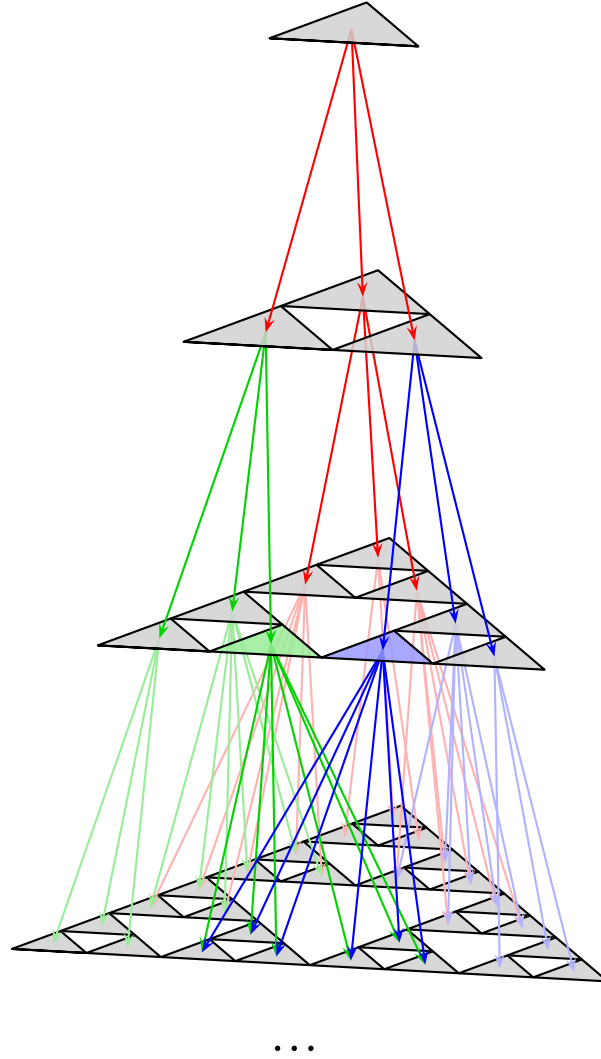


Figure 3.3: The positive transition probabilities for words up to length 2 on the Sierpiński gasket.

We denote the Markov chain on  $\mathcal{W}$ , which is generated by  $p$ , by  $(X_n)_{n \geq 0}$ . In Figure 3.3 the positive transition probabilities on the Sierpiński gasket can be seen for words up to length 2.

Further, we define the associated Markov operator  $P$  by

$$(Pf)(v) := \sum_{w \in \mathcal{W}} p(v, w) f(w), \quad v \in \mathcal{W},$$

for any non-negative function  $f$  on  $\mathcal{W}$ . We call a function  $f : \mathcal{W} \rightarrow \mathbb{R}$   $P$ -harmonic, if

$$(Pf)(v) = f(v), \quad \text{for all } v \in \mathcal{W}$$

holds.

In order to understand the definition of the transition probability we first make an observation on a basic property of  $p$  in the following lemma.

**Lemma 3.2.2.** *For  $v, \tilde{v}, w \in \mathcal{W}$  and  $\tilde{v} \sim v$  it holds that*

$$p(v, w) = p(\tilde{v}, w).$$

*Proof.* The lemma follows by definition and the fact that  $\sum_{u \sim v} m(u) = \sum_{x \sim \tilde{v}} m(x)$  holds for  $\tilde{v} \sim v \in \mathcal{W}$ .  $\square$

In the next logical step we study the transition probability between two arbitrary words  $v$  and  $w$ . The transition probability from  $v$  to  $w$  is of course only positive, if there is a path between  $v$  and  $w$  and if  $w$  is a so called successor of  $v$  (defined below).

In particular, the  $n$ -step transition probability from  $v \in \mathcal{W}$  to  $w \in \mathcal{W}$  can be calculated recursively by

$$p_n(v, w) := \sum_{u \in \mathcal{W}} p_{n-1}(v, u)p(u, w), \quad n \geq 1$$

where  $p_0(v, w) := \delta_v(w)$  (and where  $\delta_v(w)$  is the Kronecker delta function). We set  $p_n(v, w) = 0$  for  $n < 0$ . For obvious reasons it holds that  $p_n(v, w) > 0$  only if  $d(v, w) = n$  and the following definition is consistent.

**Definition 3.2.3.** *The Green function  $g : \mathcal{W} \times \mathcal{W} \rightarrow \mathbb{R}$  is defined by*

$$g(v, w) := \sum_{n=0}^{\infty} p_n(v, w) \quad v, w \in \mathcal{W}$$

*which equals  $p_{|w|-|v|}(v, w)$  for  $|v| \leq |w|$  and equals 0 otherwise.*

Based on the Green function we can determine, if a word  $v \in \mathcal{W}$  is an ancestor of  $w \in \mathcal{W}$ . We say that  $v$  is an ancestor of  $w$ , denoted by  $v \ll w$ , if  $g(v, w) > 0$  and at the same time we say that  $w$  is a successor of  $v$ . Further,  $v$  is a  $k$ -ancestor of  $w$ , if  $g(v, w) > 0$  and  $d(v, w) = k$ . The set of all  $k$ -ancestors of  $w$  is then defined by

$$\text{Anc}_k(w) := \{v \in \mathcal{W} : g(v, w) > 0 \text{ and } d(v, w) = k\}.$$

For  $w \in \mathcal{W}^*$  we define the set of all ancestors by  $\text{Anc}(w) := \bigcup_{n=1}^{\infty} \bigcup_{k=0}^n \text{Anc}_k(w \upharpoonright_n)$ .

These additional notations enable us to compare our definition of the transition probability with those in the literature, especially with [LW15]. Since the homogeneous

case has been already treated in the literature, we start with a short remark about this case.

**Remark 3.2.4.** *The definition of the transition probability in the homogeneous case is covered by Definition 3.2.1. All weights are equal and therefore  $m(a) = \frac{1}{N}$  for all  $a \in \mathcal{A}$ . It follows that*

$$p(v, w) = \begin{cases} \frac{1}{N \cdot R(v)}, & \text{if } w = \hat{v}i \text{ with } \hat{v} \sim v \text{ and } i \in \mathcal{A}, \\ 0, & \text{else,} \end{cases} \quad (3.7)$$

since  $m(w) = N^{-|w|}$  holds.

For a better understanding we adapted the notation of [LW15] to our notation. A Markov chain is of DS-type (where DS stands for “Denker–Sato”), if the following five assumptions hold:

- (LW1)  $p(v, w) > 0$  if  $v = w^-$ ;
- (LW2)  $p(v, w) > 0$  implies that either  $v = w^-$  or  $S_v(K) \cap S_{w^-}(K) \neq \emptyset$ ;
- (LW3)  $p(v, w) > 0$  for any  $w$  such that  $w^- \sim v$  and  $w \sim vk$  for some  $k \in \mathcal{W}$ ;
- (LW4)  $\inf\{p(v, w) : p(v, w) > 0, v, w \in \mathcal{W}\} =: a > 0$ ;
- (LW5) there exists a constant  $C_0 \geq 1$  such that

$$\frac{g(\emptyset, w_1)}{g(\emptyset, w_2)} \leq C_0$$

for any  $v \in \mathcal{W}^*$  and all  $w_1, w_2 \in \text{Anc}(v)$  with  $|w_1| = |w_2|$ .

We point out that Lau and Wang use the symbol “ $\sim$ ” differently than we do here and they denote by  $v \sim w$  that the two words  $v$  and  $w$  are neighbors.

The Markov chain with transition probability (3.7) is in the homogeneous case of DS-type, since all of the assumptions above are satisfied. This allows us to apply the results of [LW15, Theorem 1.2] in the homogeneous case. It follows that the (homogeneous) Martin boundary is homeomorphic to the self-similar set  $K$ .

We can not apply the results of [LW15] in a general setting, since our Markov chain is in general not a DS-type Markov chain. The reason for this is that our transition probability can get arbitrarily small and thus does not fulfill Assumption (LW4).

Therefore we have to consider the Martin boundary theory as a whole and we start with two basic statements.

**Lemma 3.2.5** ([DS01, Lemma 2.3]). *For any  $v, w \in \mathcal{W}$  and  $1 \leq k \leq d(v, w)$  we have*

$$g(v, w) = \sum_{\substack{d(v, u)=k, \\ v \ll u \ll w}} g(v, u)g(u, w).$$

This lemma was proven first by Denker and Sato and is very useful for us. As a special case we get:

**Corollary 3.2.6.** *For any  $v, w \in \mathcal{W}$  it follows*

$$g(v, w) = \sum_{u \sim w^-} g(v, u)p(u, w) = p(w^-, w) \sum_{u \sim w^-} g(v, u).$$

*Proof.* The first step of the corollary follows directly by Lemma 3.2.5 with  $k = d(v, w) - 1$ , thus the sum is over all  $u \sim w^-$ . The second step follows by using Lemma 3.2.2.  $\square$

We now take a look at the  $n$ -step transition probabilities, or equivalently the Green function. We consider the mass function, which has a multiplicative structure in the sense that

$$m(w_1 \dots w_n) = m(w_1) \cdots m(w_n)$$

holds. This multiplicative structure can be used to determine the value of the Green function from the empty word  $\emptyset$  to  $w \in \mathcal{W}$  based on  $m$ . The next theorem proves this connection between the Green function  $g$  and the mass function  $m$ .

**Theorem 3.2.7.** *Under Assumption (A) it holds for all  $w \in \mathcal{W}$*

$$g(\emptyset, w) = m(w). \tag{3.8}$$

*Proof.* We will show the statement by induction over  $|w|$  with  $w \in \mathcal{W}$ .

Consider the case  $|w| = 0$ . The only word with length 0 is the empty word  $\emptyset$ , hence

$$g(\emptyset, \emptyset) = p_0(\emptyset, \emptyset) = 1$$

holds. Furthermore holds  $m(\emptyset) = 1$  and the statement of the theorem holds for  $|w| = 0$ .

We now take a look at the induction step. Assume that Equation (3.8) is true for all words of length  $l$ . We choose  $u \in \mathcal{W}$  with  $|u| = l$  and  $i \in \mathcal{A}$ . We consider  $w = ui$  with  $|w| = |ui| = l + 1$  and applying Corollary 3.2.6 we get

$$g(\emptyset, ui) = p(u, ui) \sum_{\hat{u} \sim u} g(\emptyset, \hat{u}).$$



By induction and definition of  $p(u, ui)$  it follows

$$g(\emptyset, ui) = \frac{m(ui)}{\sum_{\hat{u} \sim u} m(\hat{u})} \sum_{\hat{u} \sim u} m(\hat{u}) = m(ui).$$

This proves the statement.  $\square$

It seems to be quite hard to calculate  $g(v, w)$  for arbitrary  $v, w \in \mathcal{W}$ . If we calculate some values of  $g(v, w)$  we realize that there is some kind of inner structure. This motivates us to define the function  $q$ .

**Definition 3.2.8.** We define the function  $q : \mathcal{W} \times \mathcal{W} \rightarrow [0, 1]$  by

$$q(v, w) = \begin{cases} \frac{g(v, w)}{m(w)} \sum_{\hat{v} \sim v} m(\hat{v}), & \text{if } v \neq w, \\ 1, & \text{if } v = w. \end{cases}$$

The function  $q$  measures in some sense, how the Green function differs from the quotient of the weight of two points. At first glance, this does not seem to be beneficial. Nevertheless we examine some properties of  $q$ . As we will see later, we are able to calculate the value of  $q(v, w)$  recursively without using  $g(v, w)$ .

**Lemma 3.2.9.** Under assumption (A), the function  $q$  satisfies for all  $v, w \in \mathcal{W}$  and all  $i, j \in \mathcal{A}$  the following properties:

a)  $g(v, w) = q(v, w) \sum_{\hat{v} \sim v} \frac{m(w)}{m(\hat{v})}$  if  $v \neq w$ ,

b)  $q(v, \hat{v}i) = 1$  for  $\hat{v} \sim v$ ,

c)  $q(v, wi) = \frac{\sum_{\hat{w} \sim w} q(v, \hat{w})m(\hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})}$  if  $v \not\sim w$ ,

d)  $q(v, wj) = q(v, wi) = q(v, \hat{w}i) = q(v, \hat{w}j)$  for  $\hat{w} \sim w$ .

*Proof.* Statement a) follows directly by definition and allows us to express  $g$  based on  $q$ .

Now, we take a look at the other statements.

We prove  $q(v, \hat{v}i) = 1$  by considering  $v, \hat{v} \in \mathcal{W}$  with  $v \sim \hat{v}$  and  $i \in \mathcal{A}$ . It holds that

$$g(v, \hat{v}i) = p(v, \hat{v}i) = \frac{m(\hat{v}i)}{\sum_{u \sim v} m(u)}.$$

If we insert this into the definition of  $q$  it follows that

$$\begin{aligned} q(v, \hat{v}i) &= \frac{g(v, \hat{v}i) \sum_{u \sim v} m(u)}{m(\hat{v}i)} \\ &= \frac{\frac{m(\hat{v}i)}{\sum_{u \sim v} m(u)} \sum_{u \sim v} m(u)}{m(\hat{v}i)} = 1 \end{aligned}$$

which proves Property b).

Let us now prove Assertion c). For this we consider  $w \in \mathcal{W}$  and  $w \not\sim v$ . By definition of  $q$  and Corollary 3.2.6 follows

$$\begin{aligned} q(v, wi) &= g(v, wi) \frac{1}{m(wi)} \sum_{\hat{v} \sim v} m(\hat{v}) \\ &= p(w, wi) \sum_{\hat{w} \sim w} g(v, \hat{w}) \frac{1}{m(wi)} \sum_{\hat{v} \sim v} m(\hat{v}). \end{aligned}$$

We insert the definition of  $p(w, wi)$  and use Lemma 3.2.9 a) for  $g(v, \hat{w})$  and receive

$$\begin{aligned} &= \frac{m(wi)}{\sum_{\hat{w} \sim w} m(\hat{w})} \sum_{\hat{w} \sim w} \frac{q(v, \hat{w}) m(\hat{w})}{\sum_{\hat{v} \sim v} m(\hat{v})} \frac{1}{m(wi)} \sum_{\hat{v} \sim v} m(\hat{v}) \\ &= \frac{\sum_{\hat{w} \sim w} q(v, \hat{w}) m(\hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})}. \end{aligned}$$

Property d) follows for  $v \not\sim w$  immediately with Property c) and for  $v \sim w$  with Property b).  $\square$

In order to prove a strong result on  $q$  in Theorem 3.2.12 we first need the following Proposition.

**Proposition 3.2.10.** *Let  $v, w, \tilde{w} \in \mathcal{W}$  and  $\tilde{w} \sim w$ . If*

$$w^- \sim (\tilde{w})^- \tag{3.9}$$

*holds, then*

$$q(v, w) = q(v, \tilde{w})$$

*follows.*

*Proof.* For simplicity we write  $u := w^-$  whenever  $w = u\tau(w)$  holds. For  $\tilde{w} \sim w$  we write  $\tilde{u} = (\tilde{w})^-$  such that  $\tilde{w} = \tilde{u}\tau(\tilde{w})$ . By Precondition (3.9) it follows that  $\tilde{u} \sim u$ .

By definition of  $q$  it follows that

$$q(v, w) = \frac{g(v, w)}{m(w) \sum_{\hat{v} \sim v} m(\hat{v})}$$

and with Corollary 3.2.6 it follows that

$$\begin{aligned} &= p(u, w) \sum_{\hat{u} \sim u} g(v, \hat{u}) \frac{1}{m(w) \sum_{\hat{v} \sim v} m(\hat{v})} \\ &= \frac{m(w)}{\sum_{\hat{u} \sim u} m(\hat{u})} \sum_{\hat{u} \sim u} g(v, \hat{u}) \frac{1}{m(w) \sum_{\hat{v} \sim v} m(\hat{v})} \\ &= \frac{1}{\sum_{\hat{u} \sim u} m(\hat{u})} \sum_{\hat{u} \sim u} g(v, \hat{u}) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})} \end{aligned}$$

holds. Since by Precondition (3.9)  $\tilde{u} \sim u$  holds, it follows

$$= \frac{1}{\sum_{\hat{u} \sim \tilde{u}} m(\hat{u})} \sum_{\hat{u} \sim \tilde{u}} g(v, \hat{u}) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})}$$

and by doing the same calculus as before, we get

$$\begin{aligned} &= \frac{m(\tilde{w})}{\sum_{\hat{u} \sim \tilde{u}} m(\hat{u})} \sum_{\hat{u} \sim \tilde{u}} g(v, \hat{u}) \frac{1}{m(\tilde{w}) \sum_{\hat{v} \sim v} m(\hat{v})} \\ &= p(\tilde{u}, \tilde{w}) \sum_{\hat{u} \sim \tilde{u}} g(v, \hat{u}) \frac{1}{m(\tilde{w}) \sum_{\hat{v} \sim v} m(\hat{v})} \\ &= \frac{g(v, \tilde{w})}{m(\tilde{w}) \sum_{\hat{v} \sim v} m(\hat{v})} \\ &= q(v, \tilde{w}). \end{aligned}$$

□

We already made a short precondition in Proposition 3.2.10 and we introduce a second precondition, so that for all  $w \in \mathcal{W}$  one of both preconditions hold. We assume this for the rest of the thesis and for a clear structure, we put them into an assumption. The following two assumptions do not stand in a direct connection to Condition (A) and to differ those assumptions from [LW15], we denote them with a “B” at the beginning.

**Assumption.** *We make the following assumptions for the rest of the thesis:*

**(B1)** *The Martin kernel in the homogeneous case exists.*

(B2) For all  $w \in \mathcal{W}$  either

$$m(w) = m(\tilde{w}) \quad \forall \tilde{w} \sim w$$

or

$$w^- \sim (\tilde{w})^- \quad \forall \tilde{w} \sim w$$

holds.

**Remark 3.2.11.** Sometimes we use the terminology that (B2) is fulfilled by  $v \in \mathcal{W}$ . With this terminology we mean that one of the conditions of (B2) is fulfilled for this particular word, i.e.  $m(v) = m(w)$  for all  $w \sim v$  or  $v^- \sim w^-$  for all  $w \sim v$ .

We can see easily that these assumptions are fulfilled by the Sierpiński gasket and his higher-dimensional analogue, see also Example 3.2.13 below. This is an important observation. Otherwise it could be possible that our assumptions are too restrictive and therefore are not fulfilled by any fractal.

We can use those assumptions on the structure and are now able to state and prove the following Theorem.

**Theorem 3.2.12.** Under (A) and (B2) it holds that

$$q(v, wi) = \frac{1}{R(w)} \sum_{\tilde{w} \sim w} q(v, \tilde{w}). \quad (3.10)$$

In particular  $q$  is independent of  $m$ .

*Proof.* Let us consider  $v, w \in \mathcal{W}$  with  $v \not\sim w$ . We split the proof into two parts, based on which of the relations of Condition (B2) is fulfilled.

First, we consider  $w \in \mathcal{W}$  and assume that all  $\tilde{w} \sim w$  fulfill  $m(w) = m(\tilde{w})$ , which is the first condition of (B2). It follows by Lemma 3.2.9 c):

$$\begin{aligned} q(v, wi) &= \frac{\sum_{\hat{w} \sim w} q(v, \hat{w})m(\hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})} \\ &= \frac{m(w)}{\sum_{\hat{w} \sim w} m(w)} \sum_{\hat{w} \sim w} q(v, \hat{w}) \\ &= \frac{1}{R(w)} \sum_{\hat{w} \sim w} q(v, \hat{w}). \end{aligned}$$

Let us now consider the other case, when the second part of Condition (B2) is fulfilled. For this we consider  $w \in \mathcal{W}$  and  $w^- \sim (\tilde{w})^-$  holds for all  $\tilde{w} \sim w$ . By definition it holds that

$$q(v, wi) = \frac{\sum_{\hat{w} \sim w} q(v, \hat{w})m(\hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})}$$

We can apply Proposition 3.2.10 and it follows that  $q(v, w) = q(v, \tilde{w})$  holds for all  $\tilde{w} \sim w$ . Using this, it follows:

$$\begin{aligned} &= \frac{q(v, w) \sum_{\hat{w} \sim w} m(\hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})} \\ &= q(v, w) = \frac{1}{R(w)} \sum_{\hat{w} \sim w} q(v, \hat{w}). \end{aligned}$$

The independence of  $q$  and  $m$  follows immediately through the representation of  $q$  by Equation (3.10).  $\square$

The fact that  $q$  is independent of  $m$  is very important and allows us later to calculate the Martin kernel in the inhomogeneous case.

For now, we continue with an example which fulfills the introduced conditions and especially Condition (B2).

**Example 3.2.13.** *We take a look at the (higher-dimensional) Sierpiński gaskets which provide us with several examples. In  $\mathbb{R}^2$  this is the classical Sierpiński gasket, which we already introduced in Figure 2.1, in  $\mathbb{R}^3$  this is the so called Sierpiński tetrahedron. Figure 3.4 shows the construction of the Sierpiński tetrahedron, where the inner part of each tetrahedron is removed such that it consists of four tetrahedra connected only at the vertices.*

*We extend this to every ambient space  $\mathbb{R}^{N-1}$  with dimension  $N-1$  ( $N \geq 2$ ) and we follow mainly the construction in [DS01, §4]. For this, we consider the points  $p_1, \dots, p_N \in \mathbb{R}^{N-1}$ , which generate a non-degenerate regular simplex  $\Delta(p_1, \dots, p_N) \subset \mathbb{R}^{N-1}$ . This means that the vectors  $\overline{p_1 p_i}$  (with  $i = 2, \dots, N$ ) are linearly independent and the simplex is*

$$\Delta(p_1, \dots, p_N) = \left\{ x \in \mathbb{R}^{N-1} : x = p_1 + \sum_{i=2}^N \lambda_i \overline{p_1 p_i}, \lambda_i \geq 0, \sum_{i=2}^N \lambda_i \leq 1 \right\}.$$

*Further, we define the midpoint of  $p_i$  and  $p_j$  by  $p_{i,j} := \frac{p_i + p_j}{2}$  ( $= p_{j,i}$ ). As a next step we define the similarities of the IFS. For  $1 \leq k \leq N$  denote by*

$$S_k : \Delta(p_1, \dots, p_N) \rightarrow \Delta(p_{1,k}, \dots, p_{N,k})$$

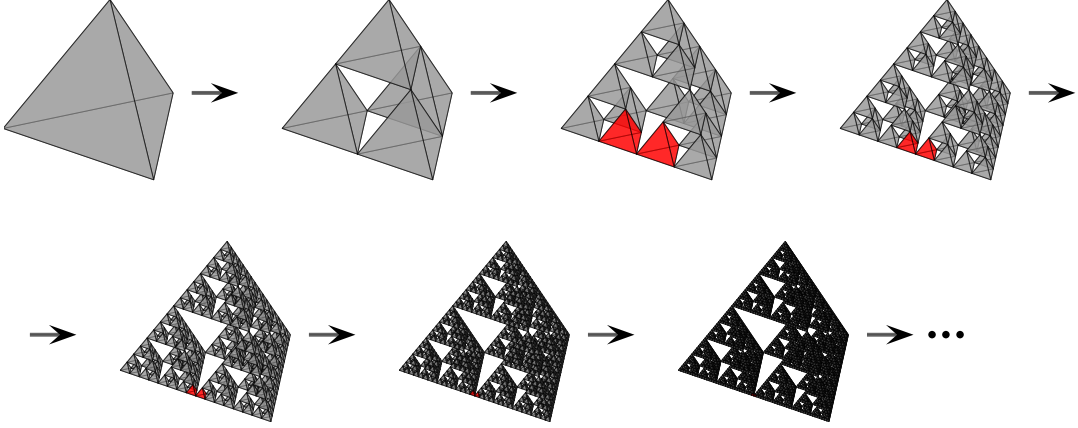


Figure 3.4: The construction of the Sierpiński tetrahedron, where two equivalent tetrahedra are highlighted.

the affine mapping onto the simplex generated by  $p_{1,k}, \dots, p_{N,k}$  which satisfies  $S_k(p_i) = p_{i,k}$ . Since  $S_k(p_k) = p_{k,k} = p_k$  holds,  $p_k$  is a fixed point of  $S_k$ . For a word  $w \in \mathcal{W}$  we define the iterations of the simplex by

$$\Delta(w) := S_w(\Delta(p_1, \dots, p_N)).$$

The Sierpiński gasket (associated to  $p_1, \dots, p_N$ ) is then defined by

$$\mathcal{S} := \bigcap_{m=1}^{\infty} \bigcup_{\substack{w \in \mathcal{W}, \\ |w|=m}} \Delta(w).$$

We can describe the topology of the Sierpiński gasket by an alphabet  $\mathcal{A} = \{1, \dots, N\}$  with  $N$  letters and the corresponding word space. For the equivalence relation we fix  $i$  and  $j$  and observe that

$$S_i(p_j) = p_{j,i} = p_{i,j} = S_j(p_i)$$

holds. In particular it holds that  $S_j \cap S_i$  is non-empty. As a consequence of this it follows that

$$uab^k \sim uba^k$$

holds for  $u \in \mathcal{W}$  and  $a, b \in \mathcal{A}$  with  $a \neq b$  and  $k \geq 1$ . We will consider the Sierpiński gasket in a later part of this thesis with different mappings.

Further, we consider an arbitrary mass function  $m$  as already introduced in Chapter 2 and we investigate, if the (higher-dimensional) Sierpiński gasket fulfills Assumptions (B1) and (B2).

Assumption (B1) is fulfilled if we apply the results of [DS01; LW15].

For Assumption (B2) we consider all possible shapes of a word. The simplest word is  $w = a^k$  with  $a \in \mathcal{A}$  and  $k \geq 1$ . In this case Assumption (B2) is trivial, since  $R(w) = 1$ .

In all other cases we consider words which can be expressed as  $w = uab^k$  with  $u \in \mathcal{W}$  and  $a, b \in \mathcal{A}$  with  $a \neq b$  and  $k \geq 1$ . The equivalent word  $\tilde{w} \sim w$  is thereby  $\tilde{w} = uba^k$ . Let us now observe, what happens for different values of  $k$ .

In the case  $k = 1$  we get that  $m(w) = m(u)m(a)m(b) = m(u)m(b)m(a) = m(\tilde{w})$  holds. Thus the first part of (B2) is fulfilled.

For  $k \geq 2$  we get that  $w^- = uab^{k-1} \sim uba^{k-1} = (\tilde{w})^-$  holds. Therefore  $w = uab^k$  fulfills the second part of (B2).

In total we obtain that (B2) is fulfilled for every  $N$ . Consequently the higher-dimensional Sierpiński gasket is a good example for a fractal, where we can introduce weights and the Assumptions (B1) and (B2) are fulfilled. As we will see later, this allows us to calculate the inhomogeneous Martin kernel and the inhomogeneous Martin boundary.

### 3.3 The Martin kernel

In the next step we define the Martin kernel. The Martin kernel is one essential part of the whole Martin boundary theory and is in some sense the regularized Green function.

**Definition 3.3.1.** The Martin kernel  $k : \mathcal{W} \times \mathcal{W} \rightarrow \mathbb{R}$  is defined by

$$k(v, w) = \frac{g(v, w)}{g(\emptyset, w)}, \quad v, w \in \mathcal{W}.$$

It is easy to see that we can express the Martin kernel by  $k(v, w) = \frac{g(v, w)}{m(w)}$  if we apply Theorem 3.2.7. Before we continue, we examine the Martin kernel and validate some properties of  $k(\cdot, \cdot)$ .

**Lemma 3.3.2.** Let  $v, w, \tilde{w} \in \mathcal{W}$  with  $\tilde{w} \sim w$  and  $i, j \in \mathcal{A}$ . Then, under (A), (B1) and (B2), it holds:

a)  $k(v, w) = q(v, w) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})}$  for  $v \neq w$ ,

b)  $k(w, \tilde{w}i) = \frac{1}{\sum_{\hat{w} \sim w} m(\hat{w})}$ ,

c)  $k(v, wi) = \frac{\sum_{\hat{w} \sim w} k(v, \hat{w})m(\hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})}$  if  $v \not\sim w$ ,

d)  $k(v, wi) = k(v, wj) = k(v, \tilde{w}j) = k(v, \tilde{w}i)$ ,

e) If  $w^- \sim (\tilde{w})^-$  for  $\tilde{w} \sim w$  and  $w \not\sim v$  holds, then it follows:

$$k(v, w) = k(v, \tilde{w}).$$

*Proof.* We consider  $v, w, \tilde{w} \in \mathcal{W}$  with  $\tilde{w} \sim w$  and  $i, j \in \mathcal{A}$  for the whole proof. We first prove Statement a) and use in the first step the definition of  $k$ . Then we apply Theorem 3.2.7 and Lemma 3.2.9 a) and it follows:

$$\begin{aligned} k(v, w) &= \frac{g(v, w)}{g(\emptyset, w)} \\ &= \frac{q(v, w) \sum_{\hat{v} \sim v} m(\hat{v})}{m(w)} \\ &= q(v, w) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})}, \end{aligned}$$

which we wanted to prove.

For Statement b) we use Statement a) for  $k(w, \tilde{w}i)$ . We get:

$$\begin{aligned} k(w, \tilde{w}i) &= q(w, \tilde{w}i) \frac{1}{\sum_{\hat{w} \sim w} m(\hat{w})} \\ &= \frac{1}{\sum_{\hat{w} \sim w} m(\hat{w})}, \end{aligned}$$

where we use in the second step Lemma 3.2.9 b), which states  $q(w, \tilde{w}i) = 1$ .

We now take a look at Statement c). For this, let  $v \not\sim w$ . We again use Statement a) and it follows

$$k(v, wi) = q(v, wi) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})}$$

and using Lemma 3.2.9 c) we get:

$$= \frac{\sum_{\hat{w} \sim w} q(v, \hat{w}) m(\hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})} \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})}$$

By the definition of  $q$  it follows:

$$= \frac{\sum_{\hat{w} \sim w} \frac{g(v, \hat{w})}{m(\hat{w})} \sum_{\hat{v} \sim v} m(\hat{v}) m(\hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})} \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})}$$



$$\begin{aligned}
&= \frac{\sum_{\hat{w} \sim w} g(v, \hat{w}) \cdot \sum_{\hat{v} \sim v} m(\hat{v})}{\sum_{\hat{w} \sim w} m(\hat{w})} \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})} \\
&= \frac{\sum_{\hat{w} \sim w} g(v, \hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})} \\
&= \frac{\sum_{\hat{w} \sim w} k(v, \hat{w})m(\hat{w})}{\sum_{\hat{w} \sim w} m(\hat{w})},
\end{aligned}$$

where we used in the last step  $g(v, \hat{w}) = k(v, \hat{w})m(\hat{w})$ . This proves Statement c). Statement d) also follows with the help of Statement a):

$$\begin{aligned}
k(v, wi) &= q(v, wi) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})} \\
&= q(v, \tilde{w}j) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})} \\
&= k(v, \tilde{w}j).
\end{aligned}$$

Thereby, we used in the second step Lemma 3.2.9 d) and in the last step Statement a) once more.

The proof of Statement e) also uses Statement a). So we consider:

$$\begin{aligned}
k(v, w) &= q(v, w) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})} \\
&= q(v, \tilde{w}) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})} \\
&= k(v, \tilde{w}).
\end{aligned}$$

In the second step we applied Proposition 3.2.10, which is possible due to the preconditions.  $\square$

In the following we consider only fractals which fulfill our Assumptions (A), (B1) and (B2). This means that the Martin kernel can be computed in the homogeneous case (for example in a similar way as in the work of [DS01] or [LW15]) and as a consequence of Theorem 3.2.12 the function  $q$  is independent of  $m$ . For now, the Sierpiński gasket is such a fractal and in Chapter 4 we will encounter more fractals. Under the Assumptions (A), (B1) and (B2) we can calculate the inhomogeneous Martin kernel through the homogeneous Martin kernel, which is basically the key to the inhomogeneous Martin boundary theory.

**Theorem 3.3.3.** *We denote by  $k_{\text{hom}}$  the homogeneous Martin kernel. It holds, under (A), (B1) and (B2), that*

$$k(v, w) = \begin{cases} k_{\text{hom}}(v, w) \frac{R(v) \cdot N^{-|v|}}{\sum_{\hat{v} \sim v} m(\hat{v})}, & \text{for } v \neq w, \\ \frac{1}{m(v)}, & \text{for } v = w. \end{cases}$$

for  $v, w \in \mathcal{W}$ .

*Proof.* First we consider the case with  $v \neq w$  for  $v, w \in \mathcal{W}$ . By Lemma 3.3.2 a) we get

$$q(v, w) = k(v, w) \sum_{\hat{v} \sim v} m(\hat{v}). \quad (3.11)$$

Since the function  $q$  is independent of the mass function  $m$ , Equation (3.11) holds for all mass functions and the value of  $q(v, w)$  will not change, if we change the mass function. Especially in the homogeneous case with  $m(\hat{v}) = m(v) = N^{-|v|}$  we have

$$\begin{aligned} q(v, w) &= k_{\text{hom}}(v, w) \sum_{\hat{v} \sim v} N^{-|\hat{v}|} \\ &= k_{\text{hom}}(v, w) R(v) N^{-|v|} \end{aligned}$$

We now can use this identity of  $q$  and insert it in the general definition of  $k$ . It follows:

$$\begin{aligned} k(v, w) &= q(v, w) \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})} \\ &= k_{\text{hom}}(v, w) R(v) N^{-|v|} \frac{1}{\sum_{\hat{v} \sim v} m(\hat{v})}. \end{aligned}$$

This proves the theorem in the case  $v \neq w$ .

We now take a short look at what happens in the case that  $w = v$ . It holds that

$$k(v, v) = \frac{g(v, v)}{m(v)},$$

and by the definition of  $g$  we have  $g(v, v) = \delta_v(v) = 1$ . Thus it follows:

$$= \frac{1}{m(v)}.$$

This completes the proof of Theorem 3.3.3. □

This theorem is one of the essential parts of this thesis. It later allows us to compare the homogeneous case with the inhomogeneous case. In order to do this, we first have to define a metric on  $\mathcal{W}$ .

**Definition 3.3.4.** A Martin metric  $\rho$  on  $\mathcal{W}$  is defined by

$$\rho(v, w) := \sum_{u \in \mathcal{W}} a(u) |k(u, v) - k(u, w)| \quad \text{for } v, w \in \mathcal{W}$$

with  $a(u) > 0$  for all  $u \in \mathcal{W}$  such that  $\sum_{u \in \mathcal{W}} \frac{a(u)}{g(\emptyset, u)} < \infty$ .

This is indeed a metric.  $\rho$  is non-negative, since

$$\rho(v, w) = \sum_{u \in \mathcal{W}} \underbrace{a(u)}_{>0} \underbrace{|k(u, v) - k(u, w)|}_{\geq 0} \geq 0$$

holds.

$\rho$  is furthermore zero if and only if  $v = w$ . For this we consider  $v = w$ . It holds

$$\rho(v, w) = \rho(v, v) = \sum_{u \in \mathcal{W}} a(u) |k(u, v) - k(u, v)| = \sum_{u \in \mathcal{W}} a(u) \cdot 0 = 0.$$

For the reverse conclusion we consider  $\rho(v, w) = 0$ . We have to deduce that  $v = w$  holds. It follows from  $\rho(v, w) = 0$  that  $|k(u, v) - k(u, w)| = 0$  must hold for all  $u \in \mathcal{W}$  and hence

$$k(u, v) = k(u, w) \text{ for all } u \in \mathcal{W}. \quad (3.12)$$

We now assume that  $v \neq w$  holds and deduce a contradiction. We can split this up into three cases, which depend on the length of the words  $v$  and  $w$ .

We first take a look at  $|v| < |w|$ . If we choose  $u = w$ , it follows that  $k(u, v) = k(w, v) = \frac{g(w, v)}{m(v)} = 0$  must hold, since  $g(w, v) = 0$  holds. On the other hand it holds that

$$k(u, w) = k(w, w) = \frac{g(w, w)}{m(w)} = \frac{1}{m(w)} \neq 0.$$

This contradicts Equation (3.12), since  $k(u, v) = 0 \neq k(u, w)$  holds.

Now consider the case where  $|v| = |w|$ . We again choose  $u = w$  and it follows that

$$k(u, v) = k(w, v) = \frac{g(w, v)}{m(v)} = \frac{\delta_w(v)}{m(v)} = 0$$

holds, since by assumption  $v \neq w$ . At the same time it holds that  $k(u, w) = k(w, w) =$

$\frac{1}{m(w)} \neq 0$ . We again get a contradiction to Equation (3.12), since  $k(u, v) = 0 \neq k(u, w)$  hold.

The last case is  $|v| > |w|$ . We can choose  $u = v$  in the same way as in the first case and it follows that  $k(v, v) = k(u, v) \neq k(u, w) = k(v, w)$ , which again contradicts Equation (3.12).

In total it holds that  $\rho(v, w) = 0$  if and only if  $v = w$ .

Further,  $\rho$  is symmetric, which can be easily seen since if we take a look at the definition of  $\rho(v, w)$ :

$$\rho(v, w) = \sum_{u \in \mathcal{W}} a(u) |k(u, v) - k(u, w)| = \sum_{u \in \mathcal{W}} a(u) |k(u, w) - k(u, v)| = \rho(w, v)$$

The third property is the triangle inequality. For  $\rho$  we can consider

$$\begin{aligned} \rho(v, w) &= \sum_{u \in \mathcal{W}} a(u) |k(u, v) - k(u, w)| \\ &= \sum_{u \in \mathcal{W}} a(u) |k(u, v) - k(u, x) + k(u, x) - k(u, w)| \\ &\leq \sum_{u \in \mathcal{W}} a(u) (|k(u, v) - k(u, x)| + |k(u, x) - k(u, w)|) \\ &= \sum_{u \in \mathcal{W}} a(u) |k(u, v) - k(u, x)| + \sum_{u \in \mathcal{W}} a(u) |k(u, x) - k(u, w)| \\ &= \rho(v, x) + \rho(x, w). \end{aligned}$$

Therefore,  $\rho$  is a metric on  $\mathcal{W}$ .

**Remark 3.3.5.** *The particular values of  $a(u)$  with  $u \in \mathcal{W}$  in Definition 3.3.4 are not really of interest. For example they can be defined by  $a(u) := m(u)^{|u|+1}$ , which fulfills both conditions on  $a(u)$ .*

## 3.4 The Martin boundary

We now take a look at the Martin boundary. For this, we need to define the completion of  $\mathcal{W}$ .

As a first step we devote ourselves to Cauchy sequences in  $\mathcal{W}$ . A sequence  $(w_n)_{n \in \mathbb{N}} \subseteq \mathcal{W}$  with  $|w_n| \rightarrow \infty$  is a  $\rho$ -Cauchy sequence if and only if

$$\lim_{n \rightarrow \infty} k(v, w_n) \text{ exists for all } v \in \mathcal{W}.$$

We use the Cauchy criteria to prove this and consider the sequence  $(k(v, w_n))_{n \in \mathbb{N}}$  for all  $v \in \mathcal{W}$ . If  $(w_n)_{n \in \mathbb{N}} \subseteq \mathcal{W}$  is a  $\rho$ -Cauchy sequence, we fix  $u \in \mathcal{W}$  and  $\varepsilon > 0$ . Since  $(w_n)_{n \in \mathbb{N}}$  is a  $\rho$ -Cauchy sequence, we can find a  $N \in \mathbb{N}$  such that

$$\rho(w_n, w_m) < \varepsilon a(u) \quad \text{holds for all } n, m > N,$$

where we used that  $0 < a(u) < \infty$  for all  $u \in \mathcal{W}$ .

At the same time we consider

$$\begin{aligned} |k(u, w_n) - k(u, w_m)| &\leq \frac{1}{a(u)} \sum_{v \in \mathcal{W}} |k(v, w_n) - k(v, w_m)| \\ &= \frac{1}{a(u)} \rho(w_n, w_m) < \frac{1}{a(u)} \varepsilon a(u) = \varepsilon \end{aligned}$$

for all  $n, m > N$  and conclude that  $(k(u, w_n))_{n \in \mathbb{N}}$  is a Cauchy sequence.

For the reverse direction we consider  $\varepsilon > 0$  and  $(k(u, w_n))_{n \in \mathbb{N}}$  as a Cauchy sequence for all  $u \in \mathcal{W}$ . Since  $\sum_{v \in \mathcal{W}} a(v) < \infty$  holds, there exists a  $N \in \mathbb{N}$  such that

$$|k(u, w_n) - k(u, w_m)| < \frac{\varepsilon}{\sum_{v \in \mathcal{W}} a(v)} \quad \text{holds for all } u \in \mathcal{W}, n, m > N.$$

For the sequence  $(w_n)_{n \in \mathbb{N}}$  it follows that

$$\begin{aligned} \rho(w_n, w_m) &= \sum_{u \in \mathcal{W}} a(u) |k(u, w_n) - k(u, w_m)| \\ &< \sum_{u \in \mathcal{W}} a(u) \frac{\varepsilon}{\sum_{v \in \mathcal{W}} a(v)} = \varepsilon \end{aligned}$$

and thus  $(w_n)_{n \in \mathbb{N}} \subseteq \mathcal{W}$  where  $|w_n| \rightarrow \infty$  is a  $\rho$ -Cauchy sequence.

We denote the set of all  $\rho$ -Cauchy sequences by  $\widehat{\mathcal{W}} := \left\{ (w_n) \subseteq \mathcal{W} : (w_n) \text{ is a } \rho\text{-Cauchy sequence} \right\}$  and we can define an equivalence relation  $\approx$  on  $\widehat{\mathcal{W}}$  by

$$(v_n) \approx (w_n) \quad \text{if and only if} \quad \lim_{n \rightarrow \infty} k(u, v_n) = \lim_{n \rightarrow \infty} k(u, w_n) \text{ for all } u \in \mathcal{W}.$$

For  $(w_n) \subseteq \widehat{\mathcal{W}}$  we indicate the equivalence class by  $\llbracket (w_n) \rrbracket$ . It holds that the space  $\overline{\mathcal{W}} = \widehat{\mathcal{W}} / \approx$  is the collection of all equivalence classes of Cauchy sequences of  $\mathcal{W}$  and is the  $\rho$ -completion of  $\mathcal{W}$ . This space is called the Martin space, which is a compact metric space [KSK76] and we still denote the metric on  $\overline{\mathcal{W}}$  by  $\rho$ . The set

$$\mathcal{M} := \overline{\mathcal{W}} \setminus \mathcal{W}$$

is called Martin boundary, which is also a compact metric space, since  $\mathcal{W}$  is open in  $\overline{\mathcal{W}}$ . For a fixed  $v \in \mathcal{W}$  we can extend every function  $w \mapsto k(v, w)$  to a continuous function on  $\mathcal{M}$ , which we denote by  $k(v, \cdot)$ . For this, let  $\xi \in \llbracket (w_n) \rrbracket \in \mathcal{M}$  and define

$$k(v, \xi) = \lim_{n \rightarrow \infty} k(v, w_n) \quad \text{for } v \in \mathcal{W}.$$

As a last point, we examine the Martin boundary  $\mathcal{M}$  in the inhomogeneous case. We compare this with the homogeneous case and for this, we need to distinguish between the two cases. Therefore, we denote by  $\mathcal{M}_{\text{hom}}$  the Martin boundary in the homogeneous case and in the same way  $k_{\text{hom}}(v, \cdot)$ ,  $\rho_{\text{hom}}(v, w)$ ,  $\widehat{\mathcal{W}}_{\text{hom}}$ ,  $\approx_{\text{hom}}$ ,  $\llbracket (w_n) \rrbracket_{\text{hom}}$  and  $\overline{\mathcal{W}}_{\text{hom}}$ . Of course, all properties of the Martin boundary are still valid in the homogeneous case.

As a preparation for Theorem 3.4.4 we show some useful statements. The first one is about the word space followed by a statement about  $\rho$ -Cauchy sequences and the equivalence relation.

**Lemma 3.4.1.** *The word space  $\mathcal{W}$  is equal to  $\mathcal{W}_{\text{hom}}$ .*

*Proof.* This is in fact very easy to see, since the definition of  $\mathcal{W}$  does not depend on the transition probability  $p$  and therefore not on the mass function  $m$ .  $\square$

**Lemma 3.4.2.** *Under (A), (B1) and (B2) every  $\rho$ -Cauchy sequence  $(w_n)$  is a  $\rho_{\text{hom}}$ -Cauchy sequence and vice versa. This implies that  $\widehat{\mathcal{W}} = \widehat{\mathcal{W}}_{\text{hom}}$  holds.*

*Proof.* We consider a  $\rho$ -Cauchy sequence  $(w_n) \subseteq \widehat{\mathcal{W}}$  and prove that  $(w_n) \subseteq \widehat{\mathcal{W}}_{\text{hom}}$ .

Since  $|w_n| \rightarrow \infty$  holds there exists a  $n_0 \in \mathbb{N}$  such that  $|w_n| > |u|$ ,  $u \in \mathcal{W}$  holds for all  $n \geq n_0$ . With Theorem 3.3.3 it follows for  $k(u, w_n)$  that

$$k_{\text{hom}}(u, w_n) = \frac{\sum_{\hat{u} \sim u} m(\hat{u})}{R(u) \cdot N^{-|u|}} k(u, w_n)$$

holds for all  $n \geq n_0$ . So it follows that

$$\begin{aligned} \lim_{n \rightarrow \infty} k_{\text{hom}}(u, w_n) &= \lim_{\substack{n \rightarrow \infty \\ n \geq n_0}} \frac{R(u) \cdot N^{-|u|}}{\sum_{\hat{u} \sim u} m(\hat{u})} k(u, w_n) \\ &= \frac{R(u) \cdot N^{-|u|}}{\sum_{\hat{u} \sim u} m(\hat{u})} \lim_{\substack{n \rightarrow \infty \\ n \geq n_0}} k(u, w_n) \quad \text{exists for all } u \in \mathcal{W}, \end{aligned}$$

since  $\lim_{n \rightarrow \infty} k(u, w_n)$  exists for all  $u \in \mathcal{W}$ . Consequently it follows  $(w_n) \subseteq \widehat{\mathcal{W}}_{\text{hom}}$ .

The reverse direction uses the same argument and completes the proof.  $\square$

**Lemma 3.4.3.** *Under (A), (B1) and (B2) the equivalence relations  $\approx$  and  $\approx_{\text{hom}}$  are identical.*

*Proof.* Let  $(v_n) \in \llbracket (w_n) \rrbracket \subseteq \widehat{\mathcal{W}}$ . Since  $(v_n) \approx (w_n)$  it follows for all  $u \in \mathcal{W}$  that

$$\lim_{n \rightarrow \infty} k(u, v_n) = \lim_{n \rightarrow \infty} k(u, w_n)$$

holds. Since  $|w_n| \rightarrow \infty$  and  $|v_n| \rightarrow \infty$  there exists a  $n_0 \in \mathbb{N}$  such that  $|w_n| > |u|$  and  $|v_n| > |u|$  holds for all  $n \geq n_0$ . It follows that

$$\frac{R(u) \cdot N^{-|u|}}{\sum_{\hat{u} \sim u} m(u)} \lim_{\substack{n \rightarrow \infty \\ n \geq n_0}} k_{\text{hom}}(u, v_n) = \frac{R(u) \cdot N^{-|u|}}{\sum_{\hat{u} \sim u} m(u)} \lim_{\substack{n \rightarrow \infty \\ n \geq n_0}} k_{\text{hom}}(u, w_n)$$

is valid, which we can reduce to

$$\lim_{\substack{n \rightarrow \infty \\ n \geq n_0}} k_{\text{hom}}(u, v_n) = \lim_{\substack{n \rightarrow \infty \\ n \geq n_0}} k_{\text{hom}}(u, w_n).$$

Consequently we get that  $(v_n) \approx_{\text{hom}} (w_n)$  respectively  $(v_n) \in \llbracket (w_n) \rrbracket_{\text{hom}}$  holds.

Using the same argument we can show that  $(v_n) \in \llbracket (w_n) \rrbracket_{\text{hom}}$  implies  $(v_n) \in \llbracket (w_n) \rrbracket$ . Overall it follows that  $\llbracket (w_n) \rrbracket = \llbracket (w_n) \rrbracket_{\text{hom}}$  holds for all  $(w_n) \subseteq \widehat{\mathcal{W}} (= \widehat{\mathcal{W}}_{\text{hom}})$ .  $\square$

We need all three statements to prove our following main result.

**Theorem 3.4.4.** *Under (A), (B1) and (B2) the inhomogeneous Martin boundary coincides with the homogeneous Martin boundary, i.e.  $\mathcal{M} = \mathcal{M}_{\text{hom}}$ .*

*Proof.* We take a look at the inhomogeneous Martin boundary  $\mathcal{M}$ . By definition of the Martin boundary and the Martin space it follows that

$$\mathcal{M} = \overline{\mathcal{W}} \setminus \mathcal{W} = \left( \widehat{\mathcal{W}} /_{\approx} \right) \setminus \mathcal{W}$$

holds. Using Lemma 3.4.2, we get:

$$= \left( \widehat{\mathcal{W}}_{\text{hom}} /_{\approx} \right) \setminus \mathcal{W}$$

Now we can apply Lemma 3.4.3 and it follows that

$$\begin{aligned} &= \left( \widehat{\mathcal{W}}_{\text{hom}} /_{\sim_{\text{hom}}} \right) \setminus \mathcal{W} \\ &= \overline{\mathcal{W}}_{\text{hom}} \setminus \mathcal{W} \end{aligned}$$

holds. With Lemma 3.4.1 we get

$$= \overline{\mathcal{W}}_{\text{hom}} \setminus \mathcal{W}_{\text{hom}} = \mathcal{M}_{\text{hom}},$$

which proves the theorem. □

Finally we can compare the inhomogeneous Martin boundary with the attractor  $K$  of the IFS.

**Corollary 3.4.5.** *Under (A), (B1) and (B2) it holds that the attractor of the IFS and the Martin boundary are homeomorphic to each other. In total we have:*

$$K \cong \mathcal{W}^* /_{\sim} \cong \mathcal{M}_{\text{hom}} = \mathcal{M}.$$

*Proof.* We observed in Remark 3.2.4 that the Markov chain in the homogeneous case is of DS-type and therefore fulfills (LW1) – (LW5) from [LW15]. With similar considerations as in [LW15, Theorem 1.2] it follows that

$$K \cong \mathcal{W}^* /_{\sim} \cong \mathcal{M}_{\text{hom}}$$

holds.

The second part of the corollary follows with Theorem 3.4.4. □

## 3.5 The minimal Martin boundary

In this last section we investigate the minimal Martin boundary, also known as space of exits. In a first step we prove that the function  $v \mapsto k(v, \xi)$  is  $P$ -harmonic. For this, we prove the following helpful lemma.

**Lemma 3.5.1.** *For any  $v, w \in \mathcal{W}$  it holds that*

$$k(v, w) = \sum_{u \in \mathcal{W}} p(v, u) k(u, w).$$



*Proof.* Let  $v, w \in \mathcal{W}$ . By definition of the Martin kernel  $k$  we get

$$k(v, w) = \frac{1}{g(\emptyset, w)} g(v, w).$$

We can apply Lemma 3.2.5 with  $1 \leq l \leq d(v, w)$ , and we get:

$$= \frac{1}{g(\emptyset, w)} \sum_{\substack{d(v, u)=l \\ v \ll u \ll w}} g(v, u) g(u, w).$$

We choose  $l = 1$  and observe that in this case  $g(v, u) = p(v, u)$  holds. This leads to:

$$\begin{aligned} &= \frac{1}{g(\emptyset, w)} \sum_{u \in \mathcal{W}} p(v, u) g(u, w) \\ &= \sum_{u \in \mathcal{W}} p(v, u) k(u, w), \end{aligned}$$

which proves the lemma. □

**Proposition 3.5.2.** *The function  $v \mapsto k_\xi(v) := k(v, \xi)$  is  $P$ -harmonic for every  $\xi \in \mathcal{M}$ .*

*Proof.* Let us consider  $v \in \mathcal{W}$  and  $\xi = \llbracket (w_n) \rrbracket \in \mathcal{M}$ . We can apply the Markov operator to  $k_\xi$  and it follows:

$$\begin{aligned} (Pk_\xi)(v) &= \sum_{u \in \mathcal{W}} p(v, u) k_\xi(u) \\ &= \sum_{u \in \mathcal{W}} p(v, u) \lim_{n \rightarrow \infty} k(u, w_n). \end{aligned}$$

We notice that  $p(v, u)$  is only positive for  $u = \tilde{v}i$  with  $\tilde{v} \sim v, i \in \mathcal{A}$ . Since  $R(v) < \infty$  holds by Lemma 3.1.3, there are only finitely many summands positive and all limits exist. Therefore, we can interchange limits and summation and get:

$$= \lim_{n \rightarrow \infty} \sum_{u \in \mathcal{W}} p(v, u) k(u, w_n)$$

We can now apply Lemma 3.5.1:

$$= \lim_{n \rightarrow \infty} k(v, w_n) = k_\xi(v).$$

Thus, it follows that the function  $k_\xi(\cdot)$  is  $P$ -harmonic. □

We now take a look at the minimal Martin boundary. For this, we recall the Poisson–Martin integral representation, which is one of the nice properties of the Martin boundary. Any non–negative harmonic function  $h$  on  $\mathcal{W}$  can be described by

$$h(\cdot) = \int_{\mathcal{M}} k(\cdot, y) d\mu_h(y) \quad (3.13)$$

with a measure  $\mu_h$  on  $\mathcal{M}$ , called spectral measure of  $h$ , which may not be unique.

The mapping onto the Martin kernel  $v \mapsto k_\xi(v)$  (for a fixed  $\xi$ ) can be expressed by (3.13), since  $k_\xi(\cdot)$  is by Proposition 3.5.2 harmonic (and non–negative). For the sake of readability we denote a spectral measure of  $k_\xi(\cdot)$  by  $\mu_\xi$ , although this may not be unique.

**Definition 3.5.3.** *A harmonic function  $h$  on  $\mathcal{W}$  is called minimal harmonic, if  $h(\emptyset) = 1$  and  $0 \leq f(w) \leq h(w)$  for all  $w \in \mathcal{W}$  with  $f$  harmonic implies  $f = ch$  with  $c \geq 0$ .*

The minimal Martin boundary  $\mathcal{M}_{\min}$  is defined by

$$\mathcal{M}_{\min} := \{\xi \in \mathcal{M} : k(\cdot, \xi) \text{ is a minimal harmonic function}\}.$$

The minimal Martin boundary is sometimes called space of exits and equals

$$\mathcal{M}_{\min} = \{\xi \in \mathcal{M} : \mu_\xi = \delta_\xi\},$$

where  $\delta_\xi$  is the Dirac measure at point  $\xi$ .

The main purpose of the minimal Martin boundary is the following: the spectral measure in (3.13) can be chosen in such a way that it is supported on  $\mathcal{M}_{\min}$  and is unique. For further information see for example [Dyn69; Woe09].

**Theorem 3.5.4.** *Under Assumptions (B1) and (B2) the minimal Martin boundary  $\mathcal{M}_{\min}$  coincides with the Martin boundary  $\mathcal{M}$ .*

*Proof.* The proof is similar to the proof in [LW15]. Since we modified it slightly, we include it here.

Our aim is to prove that  $\mu_\xi$  is the Dirac measure at point  $\xi$  for every  $\xi \in \mathcal{M}$ .

For this, let  $\xi \in \mathcal{M}$ . In a first step we prove that

$$\mathcal{M} \setminus \{\xi\} = \bigcup_{u \in \mathcal{W} : k(u, \xi) = 0} \{\zeta \in \mathcal{M} : k(u, \zeta) > 0\}, \quad (3.14)$$

which we show by proving that each side is contained in the other.

The inclusion  $\supseteq$  is quite simple. We consider  $\eta$  to be an element of the right hand side

of (3.14), i.e.

$$\eta \in \bigcup_{u \in \mathcal{W}: k(u, \xi) = 0} \{\zeta \in \mathcal{M} : k(u, \zeta) > 0\}$$

and obviously  $\eta \in \mathcal{M}$ . Suppose now that  $\eta = \xi$  holds. It follows that there exists a  $u \in \mathcal{W}$  with  $0 < k(u, \eta) = k(u, \xi) = 0$ . This is contradiction and we conclude that  $\eta \in \mathcal{M} \setminus \{\xi\}$  must hold.

For the other direction we consider  $\eta \in \mathcal{M} \setminus \{\xi\}$ . We can choose  $v, w \in \mathcal{W}^*$  such that  $(v|_n) \rightarrow \xi$  and  $(w|_n) \rightarrow \eta$  since  $\mathcal{M}$  is homeomorphic to  $\mathcal{W}^* / \sim$ . By assumption it holds that  $\eta \neq \xi$  and because of this, there exists a  $k \in \mathbb{N}$  such that  $v|_k \not\sim w|_k$  holds. The index  $k$  marks in this case, where the two words begin to differ from each other.

We can choose  $u := w|_k$ . It holds that  $u \in \text{Anc}(w)$  and hence  $k(u, \eta) > 0$ . At the same time it holds that  $u \notin \text{Anc}(v)$  and  $k(u, \xi) = 0$ . It follows that  $\eta$  is part of the right hand side of (3.14).

As a second observation we note that  $\mu_\xi(\{\zeta \in \mathcal{M} : k(u, \zeta) > 0\}) = 0$  holds for all  $u \in \mathcal{W}$  with  $k(u, \xi) = 0$ . This follows from (3.13), where

$$0 = k_\xi(u) = \int_{\mathcal{M}} k(u, y) d\mu_\xi(y)$$

holds and  $k(u, y)$  is non-negative for all  $u$  and  $y$ .

In total we get

$$\mu_\xi(\mathcal{M} \setminus \{\xi\}) = \sum_{u \in \mathcal{W}: k(u, \xi) = 0} \mu_\xi(\{\zeta : k(u, \zeta) > 0\}) = 0$$

and therefore  $\mu_\xi$  is the Dirac measure at point  $\xi$ . □

The Martin boundary mainly describes the topology of the fractal. The mass function has no influence on the topology except the degenerated case with  $m(a) = 0$  for one letter  $a \in \mathcal{A}$ . We excluded this case from the beginning, since we could describe such a fractal using an alphabet with one letter less.

At the same time, the mass function has a great influence on the transition probability and harmonic functions. For this, we recall that we denoted the Markov chain on  $\mathcal{W}$  by  $(X_n)_{n \geq 0}$ . We denote by  $\mathbb{P}_v$  the probability measure concentrated at paths starting in  $v$  as in [Dyn69] and get

$$\mathbb{P}_v[X_0 = w_0, X_1 = w_1, \dots, X_n = w_n] = \delta(v, w_0)p(w_0, w_1) \cdot \dots \cdot p(w_{n-1}, w_n)$$

for  $v, w_i \in \mathcal{W}$ ,  $i \geq 1$ .

By [Dyn69, Theorem 9] follows that a bounded harmonic function  $h : \mathcal{W} \rightarrow \mathbb{R}$  can be expressed as

$$h(w) = \int_{\mathcal{M}_{\min}} k(w, \xi) \varphi(\xi) \mu_1(d\xi) \quad \text{for } w \in \mathcal{W}$$

with a bounded measurable function  $\varphi : \mathcal{M}_{\min} \rightarrow \mathbb{R}$ . Further, it holds

$$h(X_n) \rightarrow \varphi(X_\infty) \quad \mathbb{P}_w\text{-a.e.}$$

and

$$\mathbb{E}_w(\varphi(X_\infty)) = h(w).$$

It holds that  $\mu_1$  is the spectral measure associated to the constant function  $h \equiv 1$ . As a consequence of this it follows that harmonic functions depend on the mass function. This can already be seen in Figure 1.1(a), where we considered different mass functions on the Sierpiński gasket.

# Chapter 4

## Reduction to a finite problem with useful properties

In the previous chapter we expressed a weighted fractal as the Martin boundary of a certain Markov chain. We preserved the weights and took them into account at the transition probabilities. To do so, we introduced three pre-conditions, (A), (B1) and (B2), which state that

(A) The relation  $\sim$  is an equivalence relation.

(B1) The Martin kernel in the homogeneous case exists.

(B2) For all  $w \in \mathcal{W}$  either

$$m(w) = m(\tilde{w}) \quad \forall \tilde{w} \sim w$$

or

$$w^- \sim (\tilde{w})^- \quad \forall \tilde{w} \sim w$$

holds.

If these conditions are fulfilled, we can apply the results from Chapter 3. But up to now it is not clear, when these pre-conditions are fulfilled, nevertheless they are necessary to pursue our ansatz. The next logical step is to understand these conditions in more detail. Furthermore, we do not know, whether these conditions are too restrictive: it could be possible that Example 3.2.13 contains the only inhomogeneous fractals fulfilling the Conditions (A), (B1) and (B2) at the same time. In this case the results from Chapter 3 would be pointless.

Condition (A) is quite simple to understand and can not be easily loosened. We need the equivalence of multiple words, since we consider these words as the same. If  $\sim$  would be no equivalence relation, these words would in some sense represent the same part of the fractal, but at the same time they are incompatible with each other.

Condition (B1) is also important and at the same time not really restrictive. As we elaborated in Remark 3.2.4, the Condition (B1) is in the most cases fulfilled. Furthermore, it seems pointless to calculate the Martin boundary in the inhomogeneous case if the homogeneous does not exist. We do not expect that this can be done, since the homogeneous case is part of the inhomogeneous case. Therefore, we are holding on to Condition (A) and (B1).

Now we consider (B2). We do not find ad hoc an argument, why Condition (B2) can not be replaced by another condition and thus we consider in the remainder of this chapter Condition (B2) in more detail. First of all, we express Condition (B2) in an easier way, which allows us to find fractals which satisfies Condition (B2) in addition to conditions (A) and (B1). This will be discussed in Section 4.1. After this we consider the Sierpiński gasket in Section 4.2, which is maybe the simplest fractal we can think of. As we will see, the Sierpiński gasket can sometimes fulfill (B2) in the inhomogeneous case and sometimes not. We will elaborate why this happens.

In Section 4.3 we consider some general properties of (B2). First of all we consider two minimal examples. One fulfills (B2) and the other will not. After this we build up a general example and prove that the number of equivalent words does not restrict the fact that Condition (B2) can be fulfilled.

All in all we will see that (B2) is a necessary condition which is at the same time very abstract and hard to handle.

## 4.1 Simplifying Condition (B2)

In a first step we examine how we can express Condition (B2) in a different way. This should also help to detect iterated function systems where (B2) is fulfilled in the inhomogeneous case. In other words, we reverse the problem, start with Condition (B2) and examine an inhomogeneous mass function.

For this we formulate our first lemma, which exploits the self-similar structure of a fractal.

**Lemma 4.1.1.** *Let  $\bar{v} = uv \in \mathcal{W}$  with  $u, v \in \mathcal{W}$  and  $u \neq \emptyset$ . If all  $\bar{w} \sim \bar{v}$  can be expressed as  $\bar{w} = uw$  with  $v \sim w$  and  $v$  fulfills Condition (B2), then  $\bar{v}$  fulfills (B2) as well.*

*Proof.* We consider  $\bar{v} = uv \in \mathcal{W}$  with  $u, v \in \mathcal{W}$  and  $u \neq \emptyset$ . We suppose that we can express all equivalent words  $\bar{w} \sim \bar{v}$  by  $\bar{w} = uw$ . Furthermore  $v \sim w$  should satisfy (B2). We distinguish two cases, depending on which part of Condition (B2) is fulfilled by  $v \sim w$ .

In the first case we consider that  $v \sim w$  fulfills (B2) by  $m(v) = m(w)$ . This implies that

$$\begin{aligned} m(\bar{v}) &= m(uv) = m(u)m(v) \\ &= m(u)m(w) = m(uw) = m(\bar{w}) \end{aligned}$$

holds. Hence (B2) is fulfilled by the first part of Condition (B2).

We now consider the second case where  $v \sim w$  fulfills (B2) by the second property and thus  $v^- \sim w^-$  holds. We claim that  $\bar{v}^- \sim \bar{w}^-$  holds. To verify this, we consider the definition of the equivalence relation. The two words  $\bar{v}^-$  and  $\bar{w}^-$  have the same length since  $|\bar{v}| = |\bar{w}|$  holds. Additionally  $v^- \neq w^-$  follows by the fact that  $v \sim w$  holds. Since  $v \sim w$  fulfills (B2) by the second condition (i.e.  $v^- \sim w^-$ ) it follows

$$S_{v^-}(K) \cap S_{w^-}(K) \neq \emptyset. \quad (4.1)$$

We now apply  $S_u(\cdot)$  to (4.1) and we obtain  $S_{\bar{v}^-}(K) \cap S_{\bar{w}^-}(K) \neq \emptyset$ .

As a last part it remains to be shown that  $(\bar{v}^-)^- \neq (\bar{w}^-)^-$  holds. By  $v^- \sim w^-$  and  $v^- \neq w^-$  follows  $(v^-)^- \neq (w^-)^-$ . This implies  $(uv^-)^- \neq (uw^-)^-$  respectively  $(\bar{v}^-)^- \neq (\bar{w}^-)^-$ .

Therefore we obtain that  $\bar{v}^- \sim \bar{w}^-$  holds which implies that  $\bar{v} \sim \bar{w}$  fulfills (B2) by the second condition.

This completes the proof. □

As a next step we distinguish between two cases on equivalent words. In the first case we can find an alternative formulation of (B2), in the second one (B2) is fulfilled automatically.

**Corollary 4.1.2.** *Let  $v = v_0v_1 \dots v_n \in \mathcal{W}$ . Define  $A_v := \{w \in \mathcal{W}, w \sim v, w = w_0 \dots w_n, w_0 \neq v_0\}$  as the set of all equivalent words of  $v$  which first letter differs from  $v$ . If  $v$  should satisfy (B2), then:*

- *if  $v^- \not\sim w^-$  for at least one  $w \in A_v$ , then  $m(v) = m(u)$  must hold for all  $u \in A_v$  in order to fulfill (B2) for  $v$ .*
- *if conversely  $v^- \sim w^-$  is satisfied for all  $w \in A_v$ , then  $v$  fulfills (B2) automatically.*

*Proof.* By Lemma 4.1.1 we only have to consider words which differ already in the first

letter. Therefore we consider  $v \in \mathcal{W}$  and all equivalent words  $w^i \sim v$  fulfilling  $w_0^i \neq v_0$ . Furthermore (B2) should hold for  $v$ .

In the first case there exists an equivalent word  $w^i$  with  $(w^i)^- \not\sim v^-$ . Since (B2) holds for  $v$ , the first condition must be fulfilled and consequently

$$m(w^i) = m(v) \quad \text{for all } w^i \sim v.$$

In the second case all equivalent words  $w^i$  fulfill  $(w^i)^- \sim v^-$ . It follows that (B2) is fulfilled by the second part of Condition (B2).  $\square$

For our main result we define a subset of the whole word space  $\mathcal{W}$ , which consists basically of all relevant words which we have to consider in context of (B2).

**Definition 4.1.3.** *We define the essential word space  $\mathfrak{W} \subseteq \mathcal{W}$  by*

$$\mathfrak{W} := \left\{ v \in \mathcal{W} : R(v) > 1 \text{ and} \right. \\ \left. \begin{array}{l} \exists w \in \mathcal{W} \text{ with } w \sim v : w_0 \neq v_0 \text{ and} \\ \exists w \in \mathcal{W} \text{ with } w \sim v : w^- \not\sim v^- \end{array} \right\}.$$

We note that this is only a subset of  $\mathcal{W}$ . Nevertheless we hold on the term “word space” since it is part of the word space.

Therefore, the essential word space contains all words, which have at least an equivalent word where the first letter differs and contains an equivalent word where both parents are not equivalent. This set is of great interest, as we can apply the following theorem.

**Theorem 4.1.4.** *If all  $w \in \mathfrak{W}$  fulfill  $m(\tilde{w}) = m(w)$  for all  $\tilde{w} \sim w$ , then (B2) holds.*

*Proof.* We consider  $v \in \mathcal{W}$  and have to prove that (B2) is fulfilled. We will split this up into the two cases which are based on the fact if  $v$  is contained in  $\mathfrak{W}$  or not.

First, we consider  $v \in \mathfrak{W}$ . By the precondition of the theorem it holds that

$$m(\tilde{v}) = m(v) \quad \text{for all } \tilde{v} \sim v.$$

Consequently  $v$  fulfills (B2) by the first condition.

Now we consider  $v \notin \mathfrak{W}$ . If  $R(v) = 1$  holds, then  $v$  fulfills (B2) since there are no equivalent words and we are done.

It remains the case with  $v \notin \mathfrak{W}$  and  $R(v) > 1$ . We deduce that all equivalent words  $w \sim v$  must fulfill  $w_0 = v_0$  or all equivalent words  $w \sim v$  must fulfill  $w^- \sim v^-$ . We consider



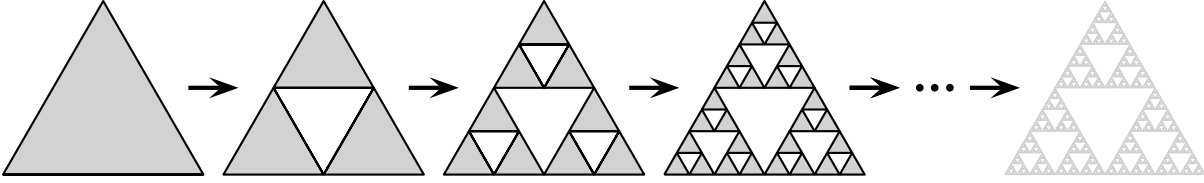


Figure 4.1: How the Sierpiński gasket is generated in a descriptive way.

as a sub-case  $w_0 = v_0$  and apply Lemma 4.1.1. It holds that we can represent  $v$  and all equivalent words  $w \sim v$  by  $v = u\bar{v}$  and  $w = u\bar{w}$  with  $u, \bar{v}, \bar{w} \in \mathcal{W}$ . By recursion it follows that  $v$  fulfills (B2). In the second sub-case we consider  $w^- \sim v^-$  for all  $w \sim v$ . By definition of (B2) it follows immediately that  $v$  fulfills (B2).

This completes the proof of the theorem.  $\square$

In the following we will use this theorem extensively, since it allows us to check only a small amount of words in  $\mathfrak{W} \subseteq \mathcal{W}$  in order to verify Condition (B2). So, instead of considering an infinite amount of words, Theorem 4.1.4 allows us to check only finitely many words because of self-similarity. Later, we will see that this amount of words is very small. For example we only have to check three relations on the Sierpiński gasket, which will be demonstrated in the following section.

## 4.2 Sierpiński gasket as the simplest example

We apply the results from Section 4.1, in particular Theorem 4.1.4. For this we consider one of the simplest fractals one can think of: the Sierpiński gasket. We know the Sierpiński gasket already from Example 3.2.13, where we considered the Sierpiński gasket in every ambient space  $\mathbb{R}^n$ .

In this section, we consider the Sierpiński gasket in the plane and abbreviate it by SG. The SG is generated using three similarities. Each similarity maps a big triangle to a smaller triangle and the small triangles are arranged in such a way that they form again the starting triangle with a hole. This is done infinitely many times and the resulting figure is called Sierpiński gasket. Figure 4.1 illustrates this procedure. Of course, we could also specify the three similarities with contraction ratio  $\frac{1}{2}$  in a mathematical way, but this would only distract ourselves from the topology. Instead we analyze in which way each similarity can be arranged. For now, let us focus on one single mapping. This mapping can be rotated by  $0, \frac{2}{3}\pi$  or  $-\frac{2}{3}\pi$  and may be flipped over (or not). In combination we get six possible ways to map the big triangle onto the small triangle. Those six ways are

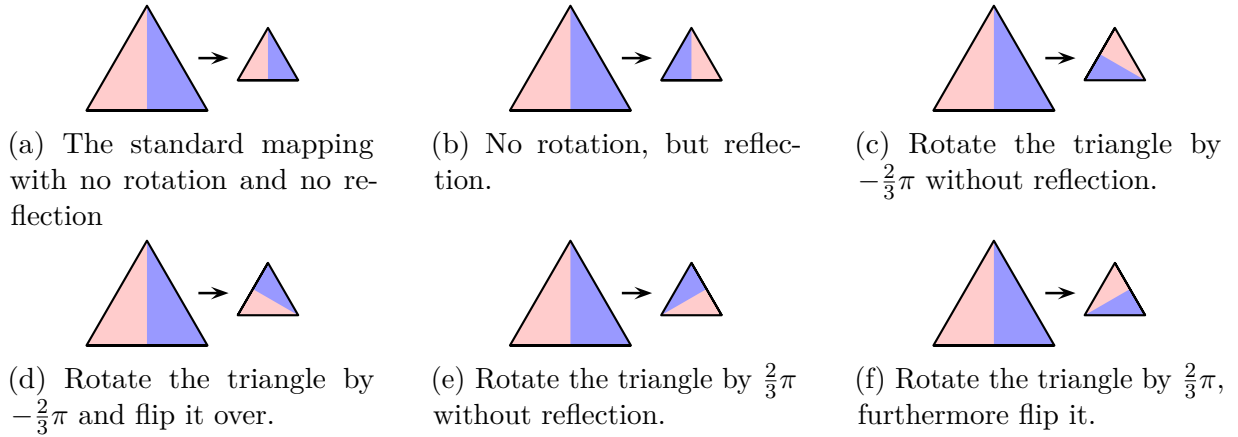


Figure 4.2: Six possible mappings to map the big triangle to the smaller triangles at the Sierpiński gasket. The coloring should help to understand the orientation and the reflection.

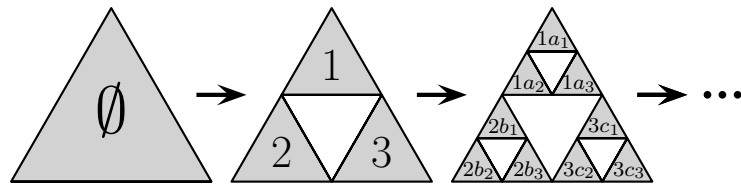


Figure 4.3: The general word space of the Sierpiński gasket. It must hold, that  $a_i$  (resp.  $b_i$  and  $c_i$ ) are pairwise different and  $a_i, b_i, c_i \in \{1, 2, 3\}$ .

illustrated in Figure 4.2. We can apply one of the six mappings independently on all three copies, therefore we get in total  $6^3 = 216$  possible IFSs. This number initially appears small. But if we consider other fractals the number of possible IFSs increases rapidly. Because of this, we should consider all IFSs in a general way.

For the Sierpiński gasket we write down the general word space. This can be seen in Figure 4.3 for words of length 2. We choose the coding in such a way that the first letter is fixed and thus the rotation and reflection is not taken into account for the first letter. For the second letter the variation of the IFSs have to be taken into account and consequently the second letter has to be variable. We can denote the children of 1 by  $1a_i$  and similarly the children of 2 (resp. 3) by  $b_i$  ( $c_i$ ). It must hold that  $a_1, a_2, a_3 \in \{1, 2, 3\}$  and that they are pairwise different. The same must hold for  $b_i$  respectively  $c_i$ . By the definition of the equivalence relation it follows, that  $1a_2 \sim 2b_1$ ,  $1a_3 \sim 3c_1$  and  $2b_3 \sim 3c_2$  hold.

We now take a look at the essential word space. It turns out that

$$\mathfrak{W} = \{1a_2, 2b_1, 1a_3, 3c_1, 2b_3, 3c_2 \in \mathcal{W} : 1a_2 \sim 2b_1, 1a_3 \sim 3c_1 \text{ and } 2b_3 \sim 3c_2\}.$$

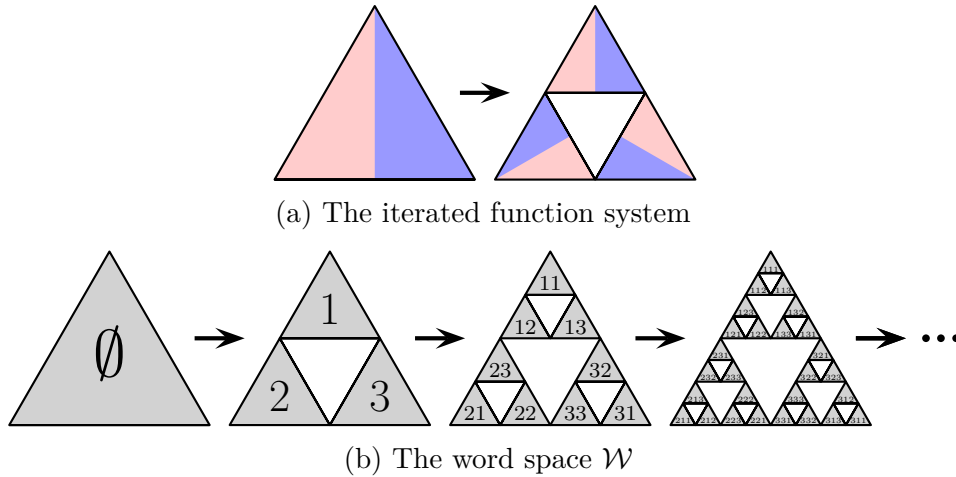


Figure 4.4: An IFS where no weights can be chosen. The coloring should help to understand the orientation of the contractions. It must hold, that  $m(1) = m(2) = m(3)$  and thus  $m(a) = \frac{1}{3}$  for  $a \in \mathcal{A}$ .

By Theorem 4.1.4 it is sufficient, to consider only those three relations. Therefore, if (B2) is fulfilled, the equations

$$\begin{aligned}
 m(1a_2) &= m(2b_1) \\
 m(1a_3) &= m(3c_1) \\
 m(2b_3) &= m(3c_2)
 \end{aligned}
 \tag{4.2}$$

must hold. Remember, that  $m$  is multiplicative, so  $m(1a_2) = m(1)m(a_2)$ . Since  $m$  is a mass function,  $m$  must fulfill

$$\begin{aligned}
 m(1) + m(2) + m(3) &= 1 \\
 m(a) &> 0 \text{ for all } a \in \mathcal{A}
 \end{aligned}
 \tag{4.3}$$

In total we get four algebraic equations in three positive variables  $m(1)$ ,  $m(2)$  and  $m(3)$ .

The values of  $a_1, \dots, c_3$  are a big problem, since they differ with each IFS and thus generate always different equations which have to be fulfilled. For this reason we consider three different examples below to get a deeper insight in the dependency of selecting small mappings fulfilling (4.2) and (4.3).

**Example 4.2.1.** We consider the IFS with  $a_1 = 1, a_2 = 2, a_3 = 3, b_1 = 3, b_2 = 1, b_3 = 2, c_1 = 2, c_2 = 3, c_3 = 1$ . Figure 4.4 shows this IFS in a graphical way, moreover the word space is indicated in the same figure.

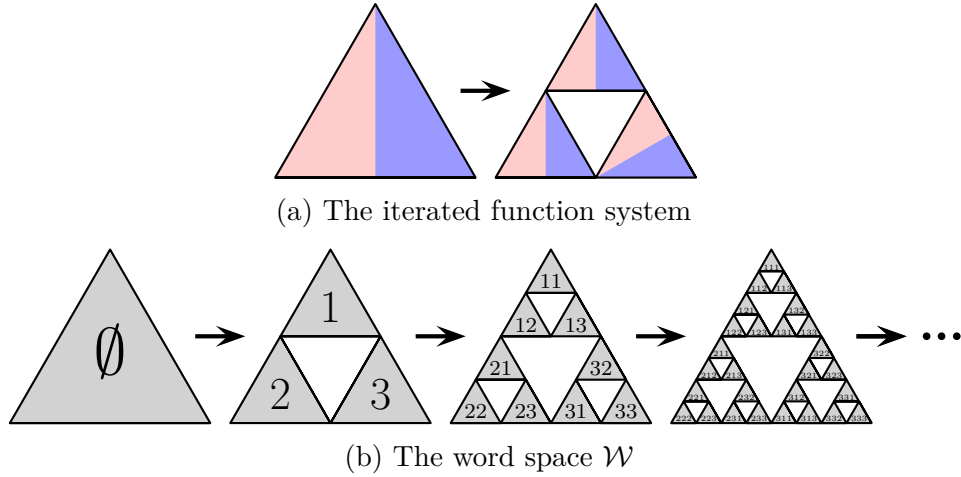


Figure 4.5: One weight can be chosen. In this case, we can choose  $m(3) \in (0, 1)$ . It follows, that  $m(1) = m(2) = \frac{1-m(3)}{2}$  must hold.

The Equations (4.2) and (4.3) turn into

$$\begin{aligned}
 m(1)m(1) &= m(2)m(3) \\
 m(1)m(3) &= m(3)m(2) \\
 m(2)m(2) &= m(3)m(2) \\
 m(1) + m(2) + m(3) &= 1 \\
 m(a) &> 0 \text{ for all } a \in \mathcal{A}.
 \end{aligned}$$

Those equations can be easily solved unlike the general equations in (4.2). We get that  $m(1) = m(2) = m(3) = \frac{1}{3}$  must hold. We conclude therefore, that this IFS only fulfills (B2) in the homogeneous case, which is relatively uninteresting (for us).

The next example is much more interesting, since we get a weighted example of the Sierpiński gasket fulfilling (B2).

**Example 4.2.2.** Let  $a_1 = 1, a_2 = 2, a_3 = 3, b_1 = 1, b_2 = 2, b_3 = 3, c_1 = 2, c_2 = 1, c_3 = 3$ . This iterated function system and the word space are illustrated in Figure 4.5.

The corresponding equations, which need to be fulfilled in order to satisfy (B2), are now

$$\begin{aligned}
m(1)m(2) &= m(2)m(1) \\
m(1)m(3) &= m(3)m(2) \\
m(2)m(3) &= m(3)m(1) \\
m(1) + m(2) + m(3) &= 1 \\
m(a) &> 0 \text{ for } a \in \mathcal{A}.
\end{aligned} \tag{4.4}$$

From the third equation of (4.4) we can immediately see that  $m(1) = m(2)$  must hold. At the same time, the equations make no restrictions on  $m(3)$ , except from  $2m(1) + m(3) = 1$ . This implies that we either can select  $m(3) \in (0, 1)$  with  $m(1) = m(2) = \frac{1-m(3)}{2}$  or we can choose  $m(1) \in (0, \frac{1}{2})$  with  $m(3) = 1 - 2m(1)$  (and of course  $m(2) = m(1)$ ).

This means, we can select the weights such that they are inhomogeneous and at the same time (B2) is fulfilled. Since we can choose one weight, we call this IFS a case with one free weight or one free parameter.

The previous two examples show that in the inhomogeneous case the validity of (B2) depends on the IFS. The question arises whether it is possible to choose more than one weight arbitrarily while (B2) is still satisfied. The next example answers this question.

**Example 4.2.3.** We consider the original Sierpiński gasket without rotations or reflections, which we already know from Example 3.2.13. As a short recall, Figure 4.6 contains the generating iterated function system and the word space. In this case we have  $a_1 = 1, a_2 = 2, a_3 = 3, b_1 = 1, b_2 = 2, b_3 = 3, c_1 = 1, c_2 = 2$  and  $c_3 = 3$  with

$$\begin{aligned}
m(1)m(2) &= m(2)m(1) \\
m(1)m(3) &= m(3)m(1) \\
m(2)m(3) &= m(3)m(2) \\
m(1) + m(2) + m(3) &= 1 \\
m(a) &> 0 \text{ for all } a \in \mathcal{A}.
\end{aligned} \tag{4.5}$$

The first three equations of (4.5) are always fulfilled and only  $m(1) + m(2) + m(3) = 1$  remains. Thus we can choose two weights and the third is determined by this equation. For example we can choose in a first step  $m(1) \in (0, 1)$ . Then we choose  $m(2) \in (0, 1 - m(1))$  and  $m(3) = 1 - m(1) - m(2)$ .

We could proceed in the same way, if we choose two other weights independently of each other. In total we have two free parameters on this particular IFS. At the same time

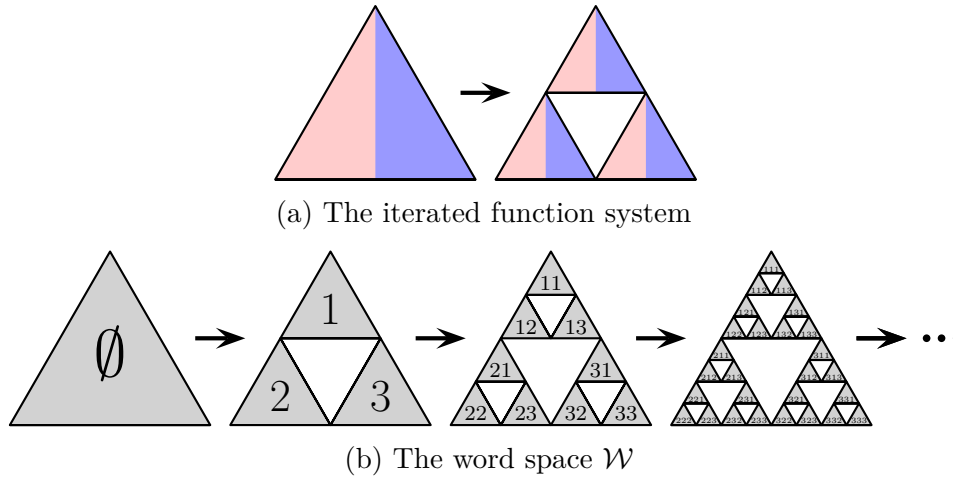


Figure 4.6: Two weights can be chosen. For example we can choose  $m(1) \in (0, 1)$  and furthermore  $m(2) \in (0, 1 - m(1))$ . This implies  $m(3) = 1 - m(1) - m(2)$ .

*this is the maximal number of free parameters.*

Those three examples show that the number of free weights depends purely on the topology of the IFS. In other words, we can consider a certain fractal and may look at all iterated function systems generating this fractal. The number of free parameters depends on the topology of each iterated function system. For example we can consider the remaining 213 IFSs which generate the Sierpiński gasket. As we will elaborate later (see Chapter 5) there are in total 194 IFSs with zero free parameters, 21 IFSs with one free parameter and exactly one (!) IFS with two free parameters. This is the iterated function system from Example 4.2.3.

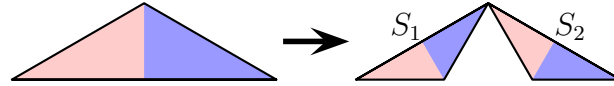
Because of this, we establish in Chapter 5 an algorithm which analyzes all possible IFSs to a certain fractal and returns the number of free parameters.

### 4.3 Some general facts

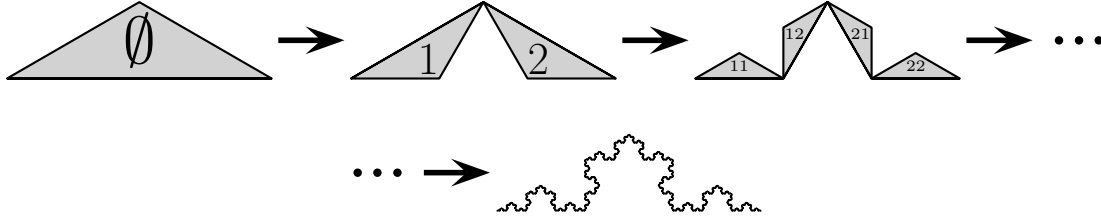
In this section we collect some facts about (inhomogeneous) fractals and the Condition (B2). As a first topic we answer the question if there is a “minimal” inhomogeneous fractal which fulfills (B2). Later we set up a minimal example which won’t fulfill (B2).

These examples are minimal in the following sense:

- The length of the alphabet  $\mathcal{A}$  respectively the number of similarities should be minimal.



(a) The IFS generating the Koch curve.



(b) The word space up to length 2 and the attractor of the Koch curve.

Figure 4.7: The Koch curve generated with two similarities and (B2) is fulfilled with an inhomogeneous mass function.

- The open set condition holds.
- All those examples have to be fractals in the common sense. We refer to [Fal90, Introduction] for a definition of a fractal by a list of properties. The intention of this property is the fact that we can describe the interval  $[0, 1]$  as a self-similar (pseudo-)fractal with two or more similarities. Since the unit interval is no fractal in the classical sense we exclude it with this property.
- The examples contain at least a word  $w$  with  $R(w) > 1$ . If we would consider a fractal where all words  $w$  fulfill  $R(w) = 1$ , then (B2) would be fulfilled automatically, also in the inhomogeneous case. This would make the example meaningless.

In other words, our minimal example should be an inhomogeneous fractal, minimal in the sense of the number of similarities and should contain a word  $w$  with  $R(w) > 1$ .

**Example 4.3.1** (Minimal inhomogeneous fractal fulfilling (B2)). *Our minimal example needs at least two small similarities, otherwise the alphabet consists of a single letter and  $R(w) > 1$  can not be fulfilled. In fact this is possible with the Koch curve, which is named after the Swedish mathematician Helge von Koch (1870 – 1924) who introduced this fractal in 1904 [Koc04].*

*The typical IFS which generates the Koch curve uses four similarities and arranges them in such a way that the similarities form the typical tent-form. Nevertheless the following IFS consisting of only two mappings and has the Koch curve as its attractor as*

well:

$$\begin{aligned}
S_1 : \mathbb{R}^2 \rightarrow \mathbb{R}^2, \quad y \mapsto & \frac{1}{\sqrt{3}} \begin{pmatrix} -\cos\left(\frac{7\pi}{6}\right) & -\sin\left(\frac{7\pi}{6}\right) \\ -\sin\left(\frac{7\pi}{6}\right) & \cos\left(\frac{7\pi}{6}\right) \end{pmatrix} y = \\
& \frac{1}{\sqrt{3}} \begin{pmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{\sqrt{3}}{2} \end{pmatrix} y \\
S_2 : \mathbb{R}^2 \rightarrow \mathbb{R}^2, \quad y \mapsto & \frac{1}{\sqrt{3}} \begin{pmatrix} -\cos\left(\frac{5\pi}{6}\right) & -\sin\left(\frac{5\pi}{6}\right) \\ -\sin\left(\frac{5\pi}{6}\right) & \cos\left(\frac{5\pi}{6}\right) \end{pmatrix} y + \begin{pmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{6} \end{pmatrix} = \\
& \frac{1}{\sqrt{3}} \begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} \end{pmatrix} y + \begin{pmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{6} \end{pmatrix}.
\end{aligned}$$

For a better understanding, the two mappings are also presented in Figure 4.7(a) in a graphical way. The coloring of the mappings should help to understand how the mappings are arranged. Figure 4.7(b) shows the word space up to length 2 and furthermore the attractor of the IFS.

The alphabet obviously is given  $\mathcal{A} = \{1, 2\}$  and the word space of all words consists of  $w = w_1 w_2 \dots w_n$  with  $w_i \in \mathcal{A}$ . It follows that the essential word space equals

$$\mathfrak{W} = \{12, 21 \in \mathcal{W} : 12 \sim 21\}.$$

We apply Theorem 4.1.4 and hence the equation

$$m(12) = m(21) \tag{4.6}$$

has to be fulfilled. Since  $m$  is multiplicative, it follows that (4.6) is always fulfilled. So, the only restriction is the fact that  $m$  has to be a mass function with

$$\begin{aligned}
m(1) + m(2) &= 1 \\
m(a) &> 0 \text{ for all } a \in \mathcal{A}.
\end{aligned}$$

Therefore it follows that we can choose  $m(1) \in (0, 1)$  and  $m(2) = 1 - m(1)$ . Of course we could also choose  $m(2) \in (0, 1)$  and determine  $m(1)$ .

Thus we found an inhomogeneous fractal which fulfills (B2). The attractor is also a “real” fractal, since the Hausdorff dimension equals  $\frac{\ln 4}{\ln 3} \approx 1.262$ . We also note that this fractal is p.c.f..

As a logical consequence we ask for a minimal example which does not fulfill (B2). This is presented in the next example.



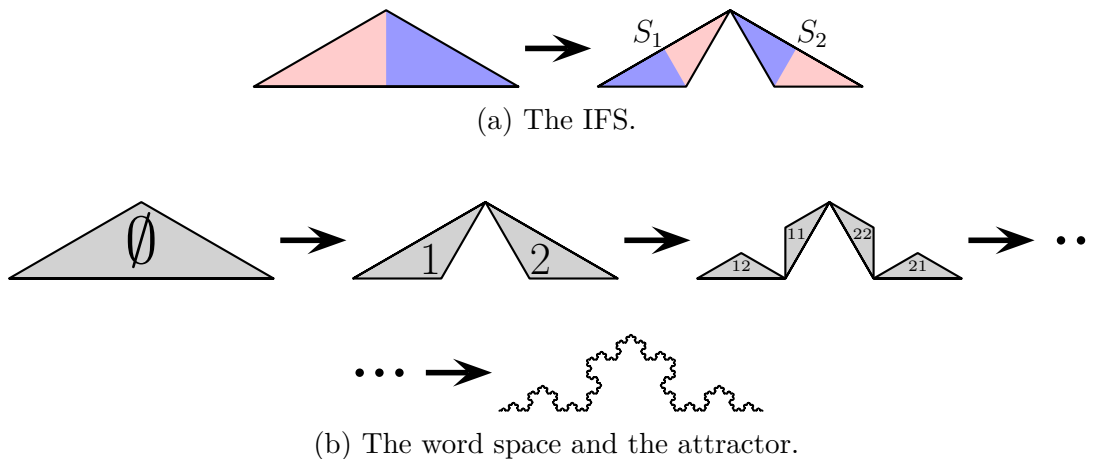


Figure 4.8: An IFS (and its word space) generating the Koch curve, but which fulfill (B2) only in the homogeneous case.

**Example 4.3.2** (Minimal inhomogeneous fractal not fulfilling (B2)). *For this minimal counter-example we use again the Koch curve, which we already introduced in the previous Example 4.3.1. For the counter-example we arrange the mappings in a different way. We choose the similarities as*

$$\begin{aligned}
 S_1 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto -\frac{1}{\sqrt{3}} \begin{pmatrix} \cos\left(\frac{\pi}{6}\right) & -\sin\left(\frac{\pi}{6}\right) \\ \sin\left(\frac{\pi}{6}\right) & \cos\left(\frac{\pi}{6}\right) \end{pmatrix} y + \begin{pmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{6} \end{pmatrix} = \\
 &\quad -\frac{1}{\sqrt{3}} \begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix} y + \begin{pmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{6} \end{pmatrix} \\
 S_2 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto -\frac{1}{\sqrt{3}} \begin{pmatrix} \cos\left(-\frac{\pi}{6}\right) & -\sin\left(-\frac{\pi}{6}\right) \\ \sin\left(-\frac{\pi}{6}\right) & \cos\left(-\frac{\pi}{6}\right) \end{pmatrix} y + \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \\
 &\quad -\frac{1}{\sqrt{3}} \begin{pmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix} y + \begin{pmatrix} 1 \\ 0 \end{pmatrix},
 \end{aligned}$$

which is also illustrated in Figure 4.8(a). The second Figure 4.8(b) contains the corresponding word space and again the attractor. The intention of this IFS is the fact that the cells “11” and “22” touch each other and are therefore equivalent.

We set up the essential word space as

$$\mathfrak{W} = \{11, 22 \in \mathcal{W} : 11 \sim 22\}$$

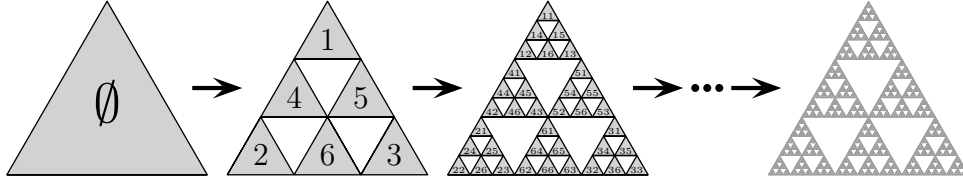


Figure 4.9: The 3-level Sierpiński gasket, which is generated in a same way as the Sierpiński gasket, but with six similarities. This  $SG_3$  has no rotations or reflections.

and by Theorem 4.1.4 the equation

$$m(11) = m(22) \tag{4.7}$$

has to be fulfilled. Since  $m(1) > 0$  and  $m(2) > 0$  must hold, Equation 4.7 is only fulfilled in the case of  $m(1) = m(2)$ . This means that an inhomogeneous mass function on this IFS can not fulfill (B2).

This example is a valid minimal example using the same arguments as in Example 4.3.1 and is also p.c.f..

In total we receive two examples, which fulfill our criteria of a minimal example. Thereby, one example fulfills (B2) and the other example does not fulfill (B2).

We elaborated in Proposition 3.1.8 that the property “nested” implies the fact that  $\sim$  forms an equivalence relation. It would be very nice, if “nested” also implies that (B2) is automatically fulfilled in the inhomogeneous case. For example, the Sierpiński gasket in Example 4.2.3 is nested and fulfills (B2) with an inhomogeneous mass function. The following example is a counter-example, that “nested” does imply the fulfilling of (B2).

**Example 4.3.3.** We consider the so called 3-level Sierpiński gasket in  $\mathbb{R}^2$ , which is often shortened to  $SG_3$ . This is a modification of the classical Sierpiński gasket and consists of six similarities with contraction ratios  $\frac{1}{3}$ . The images of the similarities are arranged in such a way that they form again an equilateral triangle, as visualized in Figure 4.9. Again, we could flip and rotate the mappings, but for now we consider the case with neither flipping nor rotating, since the property of nested should hold. The six similarities

$S_i : \mathbb{R}^2 \rightarrow \mathbb{R}^2, i = 1, \dots, 6$  are in this case

$$\begin{aligned} S_1 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \left(\frac{1}{3}, \frac{\sqrt{3}}{3}\right)^T \\ S_2 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y \\ S_3 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \left(\frac{2}{3}, 0\right)^T \\ S_4 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \left(\frac{1}{6}, \frac{\sqrt{3}}{6}\right)^T \\ S_5 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \left(\frac{1}{3}, 0\right)^T \\ S_6 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \left(\frac{1}{2}, \frac{\sqrt{3}}{6}\right)^T. \end{aligned}$$

We can choose as one OSC-set the open triangle with side length 1, and the associated word space can be found in Figure 4.9.

It holds that with this IFS the  $SG_3$  is nested and the essential fixed points  $F_0$  consist of

$$F_0 = \{q_1, q_2, q_3\} = \left\{ \left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right)^T, (0, 0)^T, (1, 0)^T \right\}.$$

The fractal is connected which can be seen directly in Figure 4.9.

Further, the fractal is symmetric, since the hyperplanes  $H_{q_1q_2}$ ,  $H_{q_1q_3}$  and  $H_{q_2q_3}$  reflect the fractal to itself.

For the third nesting property consider the following: the six copies are touching each other in the vertices of the small triangle which are exactly the images of  $q_1, q_2$  or  $q_3$ . Thus the intersection of  $K$  under different similarities consists only of images of  $q_1, q_2$  or  $q_3$ . This states the third nesting property.

Obviously, the OSC is fulfilled and the  $SG_3$  is indeed nested. We set up the essential word space

$$\begin{aligned} \mathfrak{W} &= \{12, 41, 13, 51, 21, 42, 23, 62, 31, 53, 32, 63, 43, 52, 61 \in \mathcal{W} \text{ with} \\ &12 \sim 41, 13 \sim 51, 21 \sim 42, 23 \sim 62, 31 \sim 53, 32 \sim 63, 43 \sim 52 \sim 61\} \end{aligned}$$

and by Theorem 4.1.4 we only have to consider those words. The corresponding equations

are then

$$\begin{aligned}
m(12) &= m(41) \\
m(13) &= m(51) \\
m(21) &= m(42) \\
m(23) &= m(62) \\
m(31) &= m(53) \\
m(32) &= m(63) \\
m(43) &= m(52) = m(61) \\
m(1) + m(2) + m(3) + m(4) + m(5) + m(6) &= 1 \\
m(a) &> 0 \text{ for all } a \in \mathcal{A},
\end{aligned}$$

where the last two equations is due to the fact that we consider a mass function.

We apply the multiplicity of  $m(\cdot)$ , in particular  $m(ab) = m(a)m(b)$ . After shortening the equations, we receive

$$\begin{aligned}
m(2) &= m(4) \\
m(3) &= m(5) \\
m(1) &= m(4) \\
m(3) &= m(6) \\
m(1) &= m(5) \\
m(2) &= m(6) \\
m(43) &= m(52) = m(61) \\
m(1) + m(2) + m(3) + m(4) + m(5) + m(6) &= 1 \\
m(a) &> 0 \text{ for all } a \in \mathcal{A}.
\end{aligned}$$

We can solve these equations for all  $m(a)$  and it follows that  $m(a) = \frac{1}{6}$  must hold for all  $a \in \mathcal{A}$ . This implies that we are not able to choose an inhomogeneous mass function while at the same time (B2) is fulfilled.

Therefore this is a suitable counter example that the property “nested” can not imply the fact that (B2) is fulfilled in the inhomogeneous case.

In the following, we show that (B2) is not too restrictive. In particular we consider inhomogeneous fractals, which have words  $\hat{w}$  with  $R(\hat{w}) > 2$ . We split this into two examples, since the first example is a special example with  $R(\hat{w}) = 3$  and the second example is more general with  $R(\hat{w}) = n$  and  $n \geq 4$ .

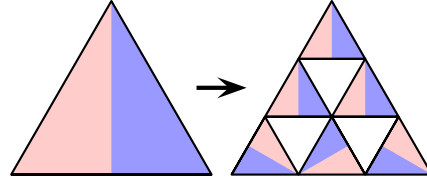
**Example 4.3.4.** We use a modification of the 3-level Sierpiński gasket as an example for a fractal with a word  $\hat{w}$  with  $R(\hat{w}) = 3$ . We already introduced the  $SG_3$  in the previous Example 4.3.3, but in contrast to this we rotate some mappings, but won't flip them. We rotate  $S_2$  and  $S_3$  by  $-\frac{2}{3}\pi$  and  $S_6$  by  $\frac{2}{3}\pi$ . The similarities  $S_i : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  are then given by:

$$\begin{aligned}
S_1 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \left(\frac{1}{3}, \frac{\sqrt{3}}{3}\right)^T \\
S_2 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} \cos\left(-\frac{2}{3}\pi\right) & -\sin\left(-\frac{2}{3}\pi\right) \\ \sin\left(-\frac{2}{3}\pi\right) & \cos\left(-\frac{2}{3}\pi\right) \end{pmatrix} y + \left(\frac{1}{6}, \frac{\sqrt{3}}{6}\right)^T = \\
&= \frac{1}{3} \begin{pmatrix} -0.5 & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & -0.5 \end{pmatrix} y + \left(\frac{1}{6}, \frac{\sqrt{3}}{6}\right)^T \\
S_3 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} \cos\left(-\frac{2}{3}\pi\right) & -\sin\left(-\frac{2}{3}\pi\right) \\ \sin\left(-\frac{2}{3}\pi\right) & \cos\left(-\frac{2}{3}\pi\right) \end{pmatrix} y + \left(\frac{5}{6}, \frac{\sqrt{3}}{6}\right)^T = \\
&= \frac{1}{3} \begin{pmatrix} -0.5 & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & -0.5 \end{pmatrix} y + \left(\frac{5}{6}, \frac{\sqrt{3}}{6}\right)^T \\
S_4 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \left(\frac{1}{6}, \frac{\sqrt{3}}{6}\right)^T \\
S_5 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \left(\frac{1}{3}, 0\right)^T \\
S_6 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} \cos\left(\frac{2}{3}\pi\right) & -\sin\left(\frac{2}{3}\pi\right) \\ \sin\left(\frac{2}{3}\pi\right) & \cos\left(\frac{2}{3}\pi\right) \end{pmatrix} y + \left(\frac{2}{3}, 0\right)^T = \\
&= \frac{1}{3} \begin{pmatrix} -0.5 & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & -0.5 \end{pmatrix} y + \left(\frac{2}{3}, 0\right)^T.
\end{aligned}$$

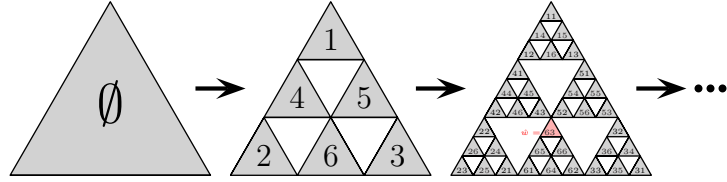
We receive the same shape of attractor as in Example 4.3.3, but with a different word space. This word space is illustrated in Figure 4.10 together with the IFS.

As a first observation we note that  $\sim$  forms still an equivalence relation. Further, we can choose the weight of the cell 1 with  $m(1) \in (0, 1)$ . For all other cells it holds, that  $m(a) = \frac{1-m(1)}{5}$  for  $a \in \{2, \dots, 6\}$ .

We claim, that (B2) is fulfilled for all  $w \in \mathcal{W}$ . For this, we can analyze words of length 2, which will not fulfill the property (B2) by  $w^- \sim (\tilde{w})^-$ . Consequently they must fulfill



(a) The iterated function system



(b) The word space  $\mathcal{W}$

Figure 4.10: The 3-level Sierpiński gasket with six similarities (some of them rotated), which has a cell  $\hat{w} = 63$  with  $R(\hat{w}) = 3$  and fulfills (B2) with inhomogeneous mass function.

$m(w) = m(\tilde{w})$  for all  $\tilde{w} \sim w$ , which gives the following list of equations:

$$\begin{aligned}
 m(12) &= m(41) \\
 m(42) &= m(22) \\
 m(21) &= m(61) \\
 m(62) &= m(33) \\
 m(32) &= m(53) \\
 m(51) &= m(13) \\
 m(43) &= m(52) = m(63).
 \end{aligned}$$

For a word of length greater 2 we can differ between two cases.

Either  $w$  fulfills  $w^- \sim (\tilde{w})^-$  for all  $\tilde{w} \sim w$ . In this case (B2) is fulfilled.

In the other case  $w$  can be expressed as  $w = uab$  and the conjugated word  $\tilde{w}$  by  $\tilde{w} = ucd$  with  $u \in \mathcal{W}$  and  $a, b, c, d \in \mathcal{W}$ . From the results for words of length 2, where  $m(ab) = m(cd)$  holds, we deduce that

$$m(w) = m(uab) = m(ucd) = m(\tilde{w})$$

holds and (B2) is fulfilled for all  $w \in \mathcal{W}$ .

Thus, we can choose the weight  $m(1) \in (0, 1)$  arbitrarily and all other weights follow.

For now it could be possible that this example can be modified in such a way, that we can choose the weight of two cells arbitrarily. As we will see in Chapter 5, there is no way to generate the attractor of the  $SG_3$  with six similarities and two or more free parameters.

**Example 4.3.5** ( $n$ -diamond propeller). *The previous Example 4.3.4 already gives a good idea, how we could construct an example (in  $\mathbb{R}^2$ ) with a cell  $\hat{w}$  and  $R(\hat{w}) = n$ . Basically, we just need  $n$  touching copies in one single point  $p$ . We can achieve this, if we choose our basic form to be diamond-like and arrange the copies around  $p$ . The copies around  $p$  look like a propeller and therefore we call this example the  $n$ -diamond propeller.*

*Since we do not want overlapping copies the angle in one corner of the diamond must be sufficiently small. For now we denote this angle by  $\tilde{\alpha}$ . We denote the angle with a tilde, because we adjust this angle in the process of the example and denote later the final angle by  $\alpha$ .*

*Let us fix  $n \in \mathbb{N}$  with  $n \geq 4$  as the number of touching copies in  $p$ . If we choose  $\tilde{\alpha} \leq \frac{\pi}{n}$ , we can arrange the copies in such a way that they only touch in  $p$ . As it turns out, it is even more practical, if we choose*

$$\alpha_0 := \pi \left( 4 \left\lceil \frac{n}{4} \right\rceil \right)^{-1} \leq \tilde{\alpha},$$

*where  $\lceil \cdot \rceil$  is the ceiling function.*

*This allows us that four copies form a cross of the shape “+” in  $p$ . Then, we can choose the contraction ratio  $c_0$  of all similarities as*

$$c_0 := \frac{1}{2} \tan \left( \frac{\alpha_0}{2} \right).$$

*This implies that two smaller copies rotated by  $\pm \frac{\pi}{2}$  just fit horizontally in the original diamond.*

*In the next step we add two additional copies at the top and at the bottom of the diamond, which guarantees us, that the shape of the OSC-set  $\mathcal{O}$  is diamond-like.*

*In the last construction step we connect those outer cells with the inner cells by a proper line of diamonds from top to bottom. For this, we have to choose the contraction ratio again smaller and its reciprocal should be a multiple of four, so that we set the new contraction ratio  $c$  as*

$$c := \left( 4 \left\lceil \frac{1}{4c_0} \right\rceil \right)^{-1}.$$

*As we still hold on the property that the smaller  $n$  copies fit exactly in the diamond, we also have to adjust  $\alpha$  and set*

$$\alpha := 2 \arctan(2c).$$

*This also implies that we have to modify our OSC-set  $\mathcal{O}$  slightly. We choose the union of the diamond and a circle around  $p$  with radius  $c$  as new set  $\mathcal{O}$ . The different steps of the*

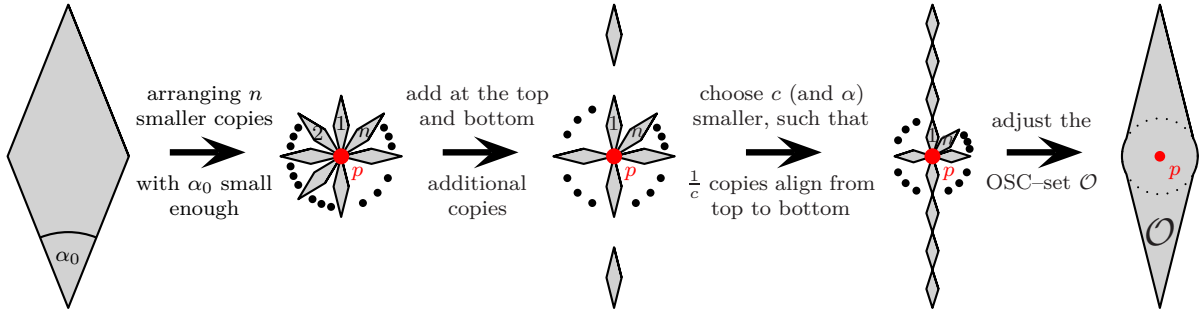


Figure 4.11: The idea of the construction of a fractal with a cell  $w$  and  $R(w) = n$  ( $n \geq 4$ ).

construction can also be seen in Figure 4.11 and should help to understand this process.

Let us now consider the word space. For this, we denote the cells in line from top to bottom by

$$1, \dots, n_0$$

with  $n_0 := \frac{1}{c}$ . The remaining cells arranged in a circle around  $p$  should be denoted by

$$n_0 + 1, \dots, N$$

where  $N = n_0 - 2 + n$  holds. We orientate the copies from top to bottom in such a way that two neighboring cells  $u$  and  $v$  touch in  $u_1$  and  $v_1$  and in  $u_{n_0}$  and  $v_{n_0}$  respectively, except for the cell  $n_0$ . The cell  $n_0$  is not rotated and it holds that  $n_0_1$  intersects with the cell  $(n_0 - 1)_{n_0}$ .

Since  $n_0$  is a multiple of four, it holds that the cells  $\frac{n_0}{2}$  and  $(\frac{n_0}{2} + 1)$  meet exactly in  $p$ . Moreover their orientation is in such a way, that they intersect in  $(\frac{n_0}{2})_1$  and  $(\frac{n_0}{2} + 1)_1$ . The remaining copies are accumulated around  $p$ . For such a copy  $w$  it holds that  $w_1$  intersects with the other cells. Figure 4.12 shows the word space up to words of length 2.

The relation  $\sim$  is defined as in Definition 3.1.1 and we claim that  $\sim$  forms an equivalence relation. For this we can quickly notice that  $\sim$  is reflexive and symmetric. For the transitivity we consider two types of touching cells. Either the cells touch in  $p$  (or iterations of it) or they touch in the line from top to bottom (and also in iterations of it). Both cases are harmless. In the first case we get that all copies intersect in  $p$  and  $\sim$  is transitive. For such a cell  $w$  it further holds  $R(w) = n$ .

In the other case we only have cells with  $R(w) = 2$ , which immediately guarantees that  $\sim$  is transitive. In both cases it follows that  $\sim$  is an equivalence relation.

Our aim is that this fractal fulfills (B2) and the mass function is inhomogeneous. If



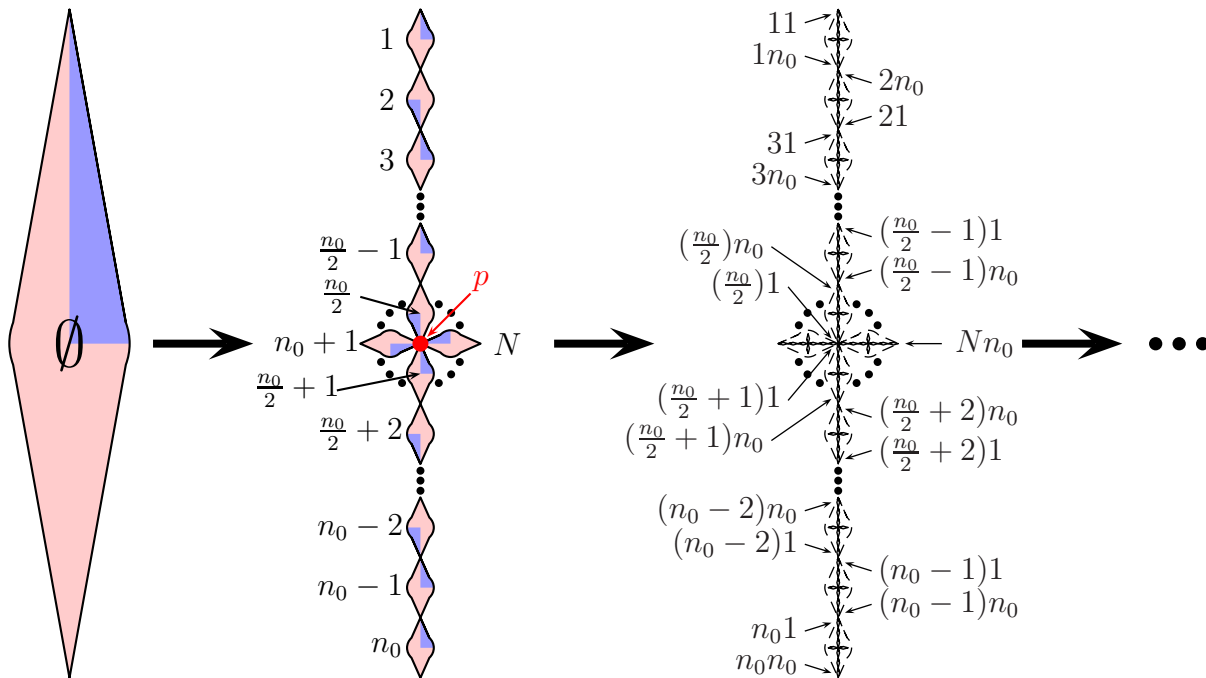


Figure 4.12: The fractal with its word space  $\mathcal{W}$ . The coloring should help to understand how the similarities are orientated. They are not flipped over, but could be flipped.

we choose a mass function arbitrarily with  $m(1), \dots, m(N) \in (0, 1)$  and  $\sum_{a=1}^N m(a) = 1$  this is in general not the case. It turns out, that some relations must hold on the mass function. For this we take a closer look at words of length 2.

First, let us consider cells around  $p$ . All equivalent words are coded by  $a1$  and in particular we consider two cells

$$w_1 = a_1 1 \quad \text{and} \quad w_2 = a_2 1$$

with  $a, a_1, a_2 \in \left\{ \frac{n_0}{2}, \frac{n_0}{2} + 1, n_0 + 1, \dots, N \right\}$ .

It holds that  $w_1 \sim w_2$  and  $w_1^- \not\sim w_2^-$  is true. Consequently  $w_1$  and  $w_2$  have to fulfill the first condition of (B2) which implies

$$m(a_1 1) = m(w_1) = m(w_2) = m(a_2 1).$$

From this follows  $m(a_1) = m(a_2)$ .

This has to be valid for all words around  $p$  and we get that

$$m(a) = m(N)$$

must hold for all  $a \in \{\frac{n_0}{2}, \frac{n_0}{2} + 1, n_0 + 1, \dots, N\}$ .

As a second case we consider cells from top to bottom. The intersection of these words are coded by

$$b1 \quad \text{or} \quad bn_0$$

with  $b \in \{1, \dots, n_0\}$ .

By construction it holds that two cells  $v_1$  and  $v_2$  intersect in the way that

$$v_1 = b_1d \quad \text{and} \quad v_2 = (b_1 + 1)d$$

with  $b_1 \in \{1, \dots, n_0 - 2\}$  and  $d \in \{1, n_0\}$  holds.

Again the first condition of (B2) has to be fulfilled, which implies

$$m(b_1d) = m(v_1) = m(v_2) = m((b_1 + 1)d)$$

and therefore  $m(b_1) = m((b_1 + 1))$  must hold. By induction it follows, that  $m(b_1) = m(1)$  must hold for  $b_1 \in \{1, \dots, n_0 - 1\}$ .

We are now nearly finished. The last two equivalent words we have to consider are

$$u_1 = (n_0 - 1)n_0 \quad \text{and} \quad u_2 = n_01.$$

By the previous results we have, that

$$m(u_1) = m((n_0 - 1)n_0) = m((n_0 - 1))m(n_0) = m(1)m(n_0)$$

must hold. This implies also that  $m(u_1) = m(u_2)$  holds independent of the choice of  $m(1)$  and  $m(n_0)$  and (B2) is fulfilled.

Therefore we can choose

$$m(n_0) \in (0, 1) \quad \text{and} \quad m(b_1) = \frac{1 - m(n_0)}{N - 1}$$

for  $b_1 \in \{1, \dots, n_0 - 1\}$  and at the same time (B2) holds. Remember that  $N = n_0 - 2 + n$ . This gives us an example with an inhomogeneous mass function and cells with  $R(w) = n \in \mathbb{N}$  for  $n \geq 4$ .

The examples of this section showed some interesting facts and allowed us to understand the Condition (B2) in more detail. At the same time those examples demonstrated clearly that (B2) is actually a suitable condition, but in practice very complex.



## Chapter 5

# Investigate the essential word spaces of a fractal with two algorithms

As we have already seen in Section 4.2, we can create the Sierpiński gasket with various iterated function systems with a fixed number of similarities. Depending on the chosen IFS it varies, how many weights can be chosen and how many weights are fixed. This depends purely on the topology, while the attractor of the IFS stays the same although the mass function  $m$  differ.

This fact raises the question how many weights we can choose, if we fix the attractor and the number of similarities, but vary the iterated function systems. Moreover we can ask how the number of free weights is distributed. It is clear that the number of free weights is bounded by  $N - 1$  parameters, since the weights have to sum up to 1 and thus the last chosen weight  $m(a)$  has to be

$$m(a) = 1 - \sum_{i=1, i \neq a}^N m(i).$$

This implies that  $N - 1$  free weights are the optimal case and up to now it is not clear, that this can be achieved for every attractor of an IFS. For the Sierpiński gasket we already know this, see Example 4.2.3.

It seems to be impossible to answer those questions in general for all possible fractals (or a wide class of fractals like all p.c.f.-fractals or all nested fractals) as this depends on the fractals topology. We can answer this question for a given fractal in some cases. For the Sierpiński gasket Section 4.2 answers this question positive, since we found for every number of free parameters an example. If we take a look at the 3-level Sierpiński gasket (also noted as  $SG_3$ ) those questions seem to be more challenging. We get a lot of

equations which have to be fulfilled and we can not solve them easily. So it seems to be quite hopeless to find easily suitable examples for every number of free weights despite the fact that they may not exist. The big problem is that we would have to check every possible IFS by hand and calculate how many free weights are possible. This may be possible for the Sierpiński gasket where we have to handle  $6^3 = 216$  different IFSs, but if we take a look at the  $SG_3$ , this is an extensive task since there are already  $6^6 = 46\,656$  possible iterated function systems. This problem gets even harder, if we consider other fractals with more possible mappings.

For this reason we develop a computer algorithm implemented in `Python 3.6` (see [Pyt]), which calculates us the number of free parameters to all possible IFSs with the same attractor and fixed length of the alphabet. Based on this, we are also able to gather how the free parameters are distributed.

Let us briefly introduce the main idea of our algorithm. We fix a particular fractal and put all IFSs with the same equations into equivalence classes. This will be done in Section 5.1. Afterwards, we go through all equivalence classes and pick one representative. For each representative we set up the essential word space  $\mathfrak{W}$ . Then we apply Theorem 4.1.4 and set up all equations which have to be fulfilled. There are (at least) two possible methods:

1. We can calculate the number of free parameters in a constructive way, which is discussed in Subsection 5.2.1 below. For this, we simplify the equations up to a certain point. Then, we apply the implicit function theorem to find a function from a sub-space to the full space. It holds that the dimension of the sub-space and the number of free weights are equal and therefore we save the dimension (e.g. in a file). Since we can determine with this method which weights can be chosen, this is called the constructive algorithm.
2. The second way is in an existential way, see Subsection 5.2.2 below. In contrast to the constructive algorithm we do not know afterwards which weights can be chosen. The general idea is that we set up a suitable function and determine the (partial) Jacobian matrix at the homogeneous point. The number of free weights equals the length of the alphabet  $\mathcal{A}$  substituted by the rank of the (partial) Jacobian.

Both methods determine the same number of free weights. We can save the number of free weights for example into a file and continue with the next IFS until we are finished.

At the end of this chapter we will have two different algorithms. The advantage of the constructive algorithm is that it additionally provides the free weights and the advantage of the existential algorithm is that it is faster than the constructive algorithm.

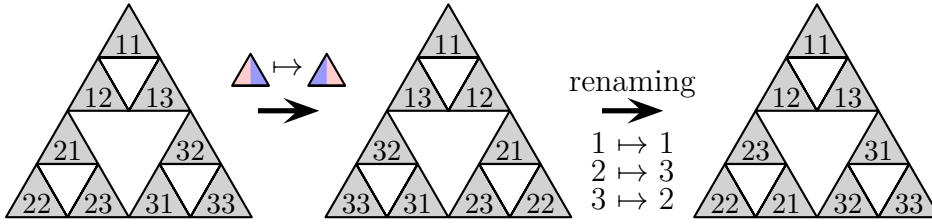


Figure 5.1: Consider a certain word space of the Sierpiński gasket. We map this word space by reflect the whole fractal (middle picture). We can rename the word space in that way, that the top triangle is called “1”, the left triangle “2” and the right triangle “3”. With this we receive the right figure which represents an other word space as the starting word space.

## 5.1 Equivalence classes of essential word spaces

In a first step we fix a certain fractal with  $N$  similarities and an alphabet  $\mathcal{A} = \{1, \dots, N\}$ , for example the Sierpiński gasket or the 3-level Sierpiński gasket. We have to check by hand, if  $\sim$  (as defined in Definition 3.1.1) is in fact an equivalence relation according to Assumption (A). Luckily, we only have to check this for one particular realization, since all other realizations are permutations of the word space which will not effect the property of equivalence.

Furthermore we have to check, if the homogeneous case solves the problem. This comes from Condition (B1), where we stated that the Martin kernel in the homogeneous case exists. This has to be also checked only once, since the weights are equal and the equations stay the same under renaming of the variables.

In the next step we put all IFSs into equivalence classes, since this will save a lot of calculating time. We notice for this, that the number of free parameters stay the same if we consider the rotated and/or flipped fractal. To get all equivalence classes to a particular IFS we can therefore consider the fractal flipped and/or rotated in such a way that the fractal is mapped to itself. In most cases we have to rename the word space. The new fractal can then be represented in the classical way and consequently we found another IFS with the same number of free parameters. Sometimes we find an invariant rotation and reflection and we receive the starting IFS.

Let us illustrate this procedure with an example.

**Example 5.1.1.** *Let us consider the Sierpiński gasket. We take a closer look at the iterated function system from Example 4.2.2. In this case the top triangle gets shrank, the bottom left is also shrank and the bottom right triangle is rotated by  $\frac{2}{3}\pi$  and flipped. The word*

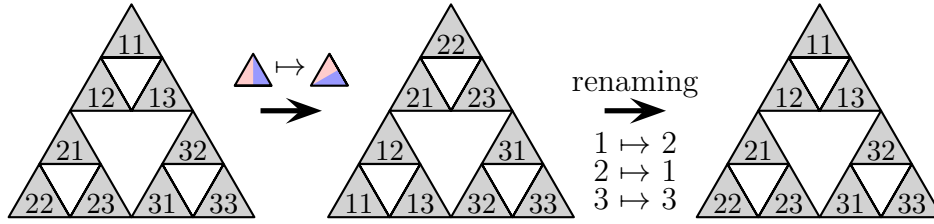


Figure 5.2: Consider a certain word space of the Sierpiński gasket (left picture). We map this word space by reflecting the whole fractal and rotate it by  $\frac{2}{3}\pi$  (middle picture). We can rename the word space in that way, that the top triangle is called “1”, the left triangle “2” and the right triangle “3”. This gives us another word space (right picture).

space (of length 2) can be found in Figure 5.1 and we represent this word space by the list

$$[[11, 12, 13], [21, 22, 23], [32, 31, 33]].$$

In this list, the first entry  $[11, 12, 13]$  represents the mapped word space of the letter “1”, which is again a list. The three words of the list represent and determine how the mapping “1” looks like. The same holds for the second and third entry of the list.

Let us now consider the fractal flipped along the  $y$ -axis without any rotation. After this, the word space can be represented by

$$[[11, 13, 12], [32, 33, 31], [21, 23, 22]].$$

We can rename the labels in such a way that the top triangle starts with “1”, the bottom left with “2” and the bottom right with “3”. To be more precise, we rename “2” to “3” and “3” to “2”. The label of “1” stays the same. We receive

$$[[11, 12, 13], [23, 22, 21], [31, 32, 33]].$$

This is illustrated in Figure 5.1. As a result, we receive a different word space. However, the equations are the same for the new word space. For this reason we only have to find the number of free parameters of the first word space to receive the number of free parameters of the second word space.

For a better understanding let us consider another transformation of the whole word space.



**Example 5.1.2.** *Again, we consider the Sierpiński gasket and use the same word space*

$$[[11, 12, 13], [21, 22, 23], [32, 31, 33]],$$

*as in Example 5.1.1 as starting word space. This time we rotate the fractal by  $\frac{2}{3}\pi$  and flip the fractal over. We receive*

$$[[22, 21, 23], [12, 11, 13], [31, 32, 33]].$$

*Analogously to Example 5.1.1, we rename the word space. To be more precise, we rename “1” to “2”, “2” to “1” and “3” does not change. After the renaming we get*

$$[[11, 12, 13], [21, 22, 23], [32, 31, 33]],$$

*which is illustrated in Figure 5.2. If we compare this with our starting word space, we see that they are identical. Thus this mapping is invariant on the starting word space and will not generate an other representative in the equivalence class.*

*Indeed this equivalence class consists in total of three different word spaces. The last missing one is*

$$[[11, 13, 12], [21, 22, 23], [31, 32, 33]],$$

*which we would receive if we rotate the starting word space by  $\frac{2}{3}\pi$  but without reflection.*

*From Example 4.2.2 we know that all these three different word spaces have exactly one free parameter.*

The Examples 5.1.1 and 5.1.2 give already a clue how we can put all word spaces into equivalence classes. We simply have to apply all possible rotations and reflections onto a starting word space and rename it afterwards. Since it is nearly impossible to predict which word spaces are put into equivalence classes we do the following: we go through all possible IFSs and determine their equivalent iterated function systems. After this, we only consider one representative of the equivalence classes and drop all other IFSs. This is also summed up in Listing 5.1 as a pseudo algorithm and implemented in `Python` in `preprocessing.py`, see Listing A.10.

Mathematically this means that we put all essential word spaces in one equivalence class, which are equivalent under renaming of the alphabet. Obviously, these renamings have to be bijective.

In fact, we do not have to put the IFSs into equivalence classes. But since the calculation of the free parameters takes a lot of computing time, this reduction is very

```

1 | set_sw := set of all possible single word spaces
2 | for single_word_space in set_sw:
3 |     for each possible mapping:
4 |         mapped_sw := apply the mapping onto single_word_space
5 |         new_sw := rename mapped_sw
6 |     save new_sw as part of the equivalence class
7 | determine all unique equivalence classes

```

Listing 5.1: Pseudo algorithm for determine all equivalence classes.

welcome, since we only have to consider between 8–12% of all cases, which depends on the fractal. Furthermore the calculation time to determine all equivalence classes is relatively short.

## 5.2 Calculate free weights

In the next big step we go through all equivalence classes of iterated function systems and determine the number of free weights. To this end, we fix a particular IFS and keep in mind that we will iterate over all IFSs. So, the next steps in this section will be done for every representative of equivalent IFSs.

We can set up a list of equations, which have to be fulfilled regarding to Theorem 4.1.4. Every equation is of the form

$$m(v_1)m(v_2) \cdots m(v_n) = m(w_1)m(w_2) \cdots m(w_n) \quad (5.1)$$

for equivalent words  $v \sim w$  with  $v = v_1 \dots v_n$  and  $w = w_1 \dots w_n$ ,  $v, w \in \mathfrak{W}$ .

This list of equations has to be replenished by

$$\sum_{a=1}^N m(a) = 1, \quad (5.2)$$

since we consider a mass function  $m$ . We summarize this in the next definition.

**Definition 5.2.1** (essential equations). *For shortness we introduce the abbreviations  $x_a := m(a)$  for  $a \in \mathcal{A}$  and  $x := (x_1, \dots, x_N)$ .*

*We define*

$$g_1 : (0, 1)^N \rightarrow \mathbb{R}, \quad x \mapsto g_1(x) := \sum_{i=1}^N x_i,$$

$$h_1 : (0, 1)^N \rightarrow \mathbb{R}, \quad x \mapsto h_1(x) := 1$$

and rewrite the Property (5.2) as

$$g_1(x) = h_1(x).$$

The same notation style should hold for the equations in (5.1). For  $v, w \in \mathfrak{W}$  with  $v \sim w$  we write

$$\begin{aligned} g_i &: (0, 1)^N \rightarrow \mathbb{R}, & x \mapsto g_i(x) &:= m(v) = m(v_1) \cdots m(v_n) \\ h_i &: (0, 1)^N \rightarrow \mathbb{R}, & x \mapsto h_i(x) &:= m(w) = m(w_1) \cdots m(w_n) \end{aligned}$$

where we introduce the index  $i$  to differ between different equations with  $i = 2, \dots, M$ .

We call the equation system

$$\begin{aligned} g_1(x) &= h_1(x) \\ &\vdots \\ g_M(x) &= h_M(x) \end{aligned} \tag{5.3}$$

the essential equations. These equations must hold in order to fulfill (B2) and are named according to the essential word space  $\mathfrak{W}$ . We note, that  $g_i$  and  $h_i$  are continuous and differentiable.

The number of free parameters is defined as the maximum number of different variables  $x_i \neq \frac{1}{N}$ , which can be chosen independently of each other and at the same time (5.3) holds.

We can solve the essential equations in (5.3) with a computer algebra system. For this we can use the Python–package `SymPy` (see [Meu+17]), which can solve also non–linear systems. Unfortunately `SymPy` returns on some IFSs an error and says, that the list of equations is not solvable by its build–in function `nonlinsolve`.

Consequently, we develop two different algorithms to solve this problem in the following subsections.

### 5.2.1 With a constructive algorithm

The first way to determine the number of free parameters is in a constructive way. We simplify the Essential Equations (5.3) until the equations can not be further simplified. These simplified equations show the connections of the variables and allows us to determine the variables, which can be chosen in  $(0, 1)$ . At the same time we get the dependencies for those variables, which have to be calculated by other variables. At the end we have determined, how many free parameters exists and which are those free parameters. In other words, we calculated the number of free parameters in a constructive way. The name of the algorithm comes from this fact.

First of all, we simplify the Essential Equations (5.3) in parts on our own. For this we have several possibilities, which we will discuss in the following methods.

**Method 5.2.2** (“delete double equations”). *If two equations are identical, we can reduce our equation list by one of those equations and consider only the reduced equation list. For example we can reduce*

$$\begin{aligned}x_a x_b &= x_c x_d \\x_c x_d &= x_a x_b\end{aligned}$$

to

$$x_a x_b = x_c x_d$$

with  $x_a, x_b, x_c, x_d > 0$ .

This method seems to be quite trivial, nevertheless we mention this method. This allows us to refer to it later and we may build up a more complex algorithm using this besides the following methods.

**Method 5.2.3** (“delete factorial variables”). *We can shorten any factor  $x_i, i \in \mathcal{A}$ , which occurs on both sides, since  $x_i > 0$ . For example we can replace*

$$x_a^k x_b = x_a^l x_c \text{ with } k, l \in \mathbb{R}$$

by

$$x_b = x_a^{l-k} x_c.$$

This allows us to shorten unnecessary factors without touching the statement of the equation. This fact is essential, since we only simplify our equations for SymPy. The next method will also preserve the statement.

**Method 5.2.4** (“replace polynomial terms”). *We can extract roots and exponents, if they occur at one side with the same exponent. This is possible, since  $x_a > 0$  for all  $a \in \mathcal{A}$  must hold. For example we can replace*

$$x_a^k = x_b x_c^2 \text{ with } k \in \mathbb{R} \setminus \{0\}$$

by

$$x_a = (x_b x_c^2)^{-k}$$

The purpose of this method is the observation that SymPy has several problems handling exponents, especially with rational exponents. This method seems to be counterproductive,

since we modify the equations in the way, that (almost surely) rational exponents occur. On the other hand it allows us to define and use the following method. We use the fact that some variable occur isolated at one side.

**Method 5.2.5** (“substitute variables”). *We substitute variables, if they are isolated at one side and the exponent equals 1 (which we can achieve by Method 5.2.4). For example we can replace*

$$\begin{aligned}x_a x_b &= x_c x_d \\x_a &= \sqrt{x_e x_f}\end{aligned}$$

by

$$\begin{aligned}\sqrt{x_e x_f} x_b &= x_c x_d \\x_a &= \sqrt{x_e x_f}.\end{aligned}$$

*Further, we know that the substituted variable can not be free and we reduced our problem by one dimension.*

All these introduced methods are interesting, but alone they are not really useful. Thus we combine them and make them a very powerful weapon to simplify our equations.

**Method 5.2.6** (“simplify equations”). *We combine the Methods 5.2.2 – 5.2.5 to the following recursive method, called simplify equations.*

*We start with all essential equations and call them free equations. Furthermore we start with so called free variables, which are all variables. For the recursion we implement the fixed equations and fixed variables as empty. Then we do the following:*

1. *replace in free equations all polynomial terms (by Method 5.2.4),*
2. *delete in free equations all factorial variables (by Method 5.2.3),*
3. *delete in free equations all double equations (by Method 5.2.2),*
4. *if we can substitute a free variable in free equations (by Method 5.2.5), substitute the variable, save it as a fixed variable and delete it from free variables. Further add the equation defining the variable as fixed equation and delete it from free equations. After this, start again at 1..*

*Otherwise we are done.*

With this method we receive a list of free equations (potentially empty), a list of fixed equations (defining the fixed variables), a list of free variables and a list of fixed variables.

After we apply Method 5.2.6 to our equations we receive in most cases an empty list of free equations. Further, the list of fixed equations and the list of fixed variables have the same length, since every equation in fixed equations provides a definition of a fixed variable. If the list of free equations is empty, it is clear that we can choose for every variable in the list of free variables a different value, where maybe some restrictions must hold. Nevertheless we can choose them independently of each other and those variables span up a sub-space of  $(0, 1)^N$ .

Since we are only interested in the dimension of this sub-space (where the dimension equals the number of free parameters) we have to consider a mathematical way to verify this.

For this, we apply the implicit function theorem. To do so, let us first fix some notations.

**Definition 5.2.7.** *Let us fix a specific IFS, its word space and the essential equations which have to be fulfilled regarding to Theorem 4.1.4:*

$$\begin{aligned} g_1(x) &= h_1(x) \\ &\vdots \\ g_M(x) &= h_M(x) \end{aligned} \tag{5.4}$$

We apply Method 5.2.5 on the Essential Equations (5.4).

For simplicity let us denote the free variables by  $\bar{x}_1, \dots, \bar{x}_k$  and the fixed variables by  $\bar{y}_1, \dots, \bar{y}_m$ , where  $k + m = N$  holds. Further we note by  $\bar{x} := (\bar{x}_1, \dots, \bar{x}_k)$  and  $\bar{y} := (\bar{y}_1, \dots, \bar{y}_m)$  those variables combined as a vector.

We write the list of all  $\bar{M}$  equations (the combination of free and fixed equations) as

$$\begin{aligned} \bar{g}_1(\bar{x}, \bar{y}) &= \bar{h}_1(\bar{x}, \bar{y}) \\ &\vdots \\ \bar{g}_{\bar{M}}(\bar{x}, \bar{y}) &= \bar{h}_{\bar{M}}(\bar{x}, \bar{y}) \end{aligned} \tag{5.5}$$

where  $\bar{g}_i(\bar{x}, \bar{y})$  and  $\bar{h}_i(\bar{x}, \bar{y})$  are polynomials in  $\bar{x}_1, \dots, \bar{x}_k, \bar{y}_1, \dots, \bar{y}_m$  and  $\bar{M} (\leq M)$  holds.

We define  $F_i(\bar{x}, \bar{y}) := \bar{g}_i(\bar{x}, \bar{y}) - \bar{h}_i(\bar{x}, \bar{y})$  for  $i = 1, \dots, \bar{M}$  and rewrite Equations (5.5)

as a single function by

$$F : (0, 1)^k \times (0, 1)^m \rightarrow \mathbb{R}^{\bar{M}}, \quad (x, y) \mapsto F(x, y) := \begin{pmatrix} F_1(x, y) \\ \vdots \\ F_{\bar{M}}(x, y) \end{pmatrix}.$$

Finally we are interested in solutions with

$$F(x, y) = 0$$

and moreover to maximize  $m$  and verify, that  $m$  can be chosen maximal.  $m$  equals then the number of free parameters to this IFS. Further are  $\bar{y}_1, \dots, \bar{y}_m$  the free parameters which can be chosen.

With this notation it is more clear, what we do. Further, this notation implies already an idea of the direction our considerations will take.

First of all, let us notice that always  $m \leq \bar{M}$  must hold. This comes from the fact that for every variable  $\bar{y}_i$  an equation exists, in particular in fixed variables. Therefore we can split up  $\bar{M}$  into  $\bar{M} = \bar{M}_1 + m$ , where  $\bar{M}_1$  equals the number of equations in free equations and  $m$  the number of equations in fixed equations.

If further  $\bar{M}_1 = 0$  and thus  $\bar{M} = m$  holds, we can apply the implicit function theorem. For completeness we repeat the implicit function theorem in a general formulation.

**Theorem 5.2.8** (Implicit Function Theorem, [For17]). *Let  $U_1 \subset \mathbb{R}^k$  and  $U_2 \subset \mathbb{R}^m$  open subsets and*

$$F : U_1 \times U_2 \rightarrow \mathbb{R}^m, F(x, y) := \begin{pmatrix} F_1(x, y) \\ \vdots \\ F_m(x, y) \end{pmatrix}$$

*a continuous and differentiable function. Let  $(a, b) \in U_1 \times U_2$  a point with  $F(a, b) = 0$ . If the partial Jacobian matrix*

$$J = \frac{\partial F}{\partial y} := \frac{\partial(F_1, \dots, F_m)}{\partial(y_1, \dots, y_m)} := \begin{pmatrix} \frac{\partial F_1}{\partial y_1} & \dots & \frac{\partial F_1}{\partial y_m} \\ \vdots & & \vdots \\ \frac{\partial F_m}{\partial y_1} & \dots & \frac{\partial F_m}{\partial y_m} \end{pmatrix}$$

*is invertible at  $(a, b)$ , then there exists an open neighborhood  $V_1 \subset U_1$  from  $a$ , an open neighborhood  $V_2 \subset U_2$  from  $b$  and a continuous and differentiable function  $G : V_1 \rightarrow V_2$*

with  $G(a) = b$  such that

$$F(x, G(x)) = 0 \quad \text{for all} \quad x \in V_1.$$

If further  $(x, y) \in V_1 \times V_2$  is a point with  $F(x, y) = 0$ , then it holds that  $y = G(x)$ .

The implicit function theorem is well known in multivariable calculus and there exist a lot of standard textbooks about it (see for example [For17; PM85; LS90]) and even a whole textbook specialized to the implicit function theorem itself [KP13]. For this reason we refer the reader to one of these textbooks and recommend [KP13] for some less common applications of the implicit function theorem.

For our purpose we formulate a modification of the implicit function theorem which fits exactly our requirements.

**Theorem 5.2.9** (Application of the Implicit Function Theorem). *We consider a specific IFS. We denote by  $F$  the simplified equations as in Definition 5.2.7. If  $\bar{M} = m$  holds, we are facing the following problem:*

$$F : (0, 1)^k \times (0, 1)^m \rightarrow \mathbb{R}^m, F(x, y) := \begin{pmatrix} F_1(x, y) \\ \vdots \\ F_m(x, y) \end{pmatrix}$$

where  $F$  is a continuous and differentiable function.

Let  $J$  be the partial Jacobian matrix defined by

$$J(x, y) := \frac{\partial F}{\partial y} := \frac{\partial(F_1, \dots, F_m)}{\partial(y_1, \dots, y_m)}(x, y) := \begin{pmatrix} \frac{\partial F_1}{\partial y_1}(x, y) & \cdots & \frac{\partial F_1}{\partial y_m}(x, y) \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial y_1}(x, y) & \cdots & \frac{\partial F_m}{\partial y_m}(x, y) \end{pmatrix}$$

and define

$$x_{hom} := \left( \frac{1}{N}, \dots, \frac{1}{N} \right) \text{ with } k \text{ entries,}$$

$$y_{hom} := \left( \frac{1}{N}, \dots, \frac{1}{N} \right) \text{ with } m \text{ entries.}$$

as the homogeneous solution points.

If  $\det J(x_{hom}, y_{hom}) \neq 0$  holds, we can choose  $k$  variables arbitrarily in a small neighborhood  $U_1 \subset (0, 1)^k$  of  $x_{hom}$  and thus the IFS has  $k$  free parameters.

*Proof.* The proof is in fact an application of the Implicit Function Theorem 5.2.8.



We recall that if the partial Jacobian  $J$  is invertible at the point  $(x_{\text{hom}}, y_{\text{hom}})$ , then there exists an open neighborhood  $U_1 \subset (0, 1)^k$  of  $x_{\text{hom}}$ , an open neighborhood  $U_2 \subset (0, 1)^m$  of  $y_{\text{hom}}$  and a continuous differentiable function  $G : U_1 \rightarrow U_2$  with  $G(x_{\text{hom}}) = y_{\text{hom}}$  such that

$$F(x, G(x)) = 0 \quad \text{holds for all } x \in U_1.$$

The exact form of  $G$  is irrelevant, but we can choose in  $U_1$  each coordinate of  $x$  independently of each other. In other words we can choose  $k$  weights independently and the IFS has  $k$  free parameters.  $\square$

The theorem allows us to determine the number of free parameters in a clear, mathematical way. The main benefit lies indeed in another property: we can implement Theorem 5.2.9 as a computer algorithm. To do so, we only have to set up the function  $F$ , choose  $k$  (and  $m$ ) wisely and verify that  $\det J(x_{\text{hom}}, y_{\text{hom}}) \neq 0$  holds. All those things can be done by a computer software and in particular by using the Python package `SymPy`. This will be done later in this subsection. For now, we consider an example, where we apply Theorem 5.2.9 by hand. Thereby we can identify the following example with a certain number (or counter) in our algorithm.

**Example 5.2.10** (Hexagasket, counter 29 393). *We consider the Hexagasket. The main structure of the Hexagasket is a hexagon and six copies are placed in each corner. Obviously we can rotate (and flip) the similarities. Our particular example consists of the IFS with the mappings*

$$\begin{aligned} S_1 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} \cos(0) & -\sin(0) \\ \sin(0) & \cos(0) \end{pmatrix} y + \left(\frac{1}{3}, \frac{2}{3}\right)^T = \\ &= \frac{1}{3}x + \left(\frac{1}{3}, \frac{2}{3}\right)^T \\ S_2 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} \cos\left(-\frac{\pi}{3}\right) & -\sin\left(-\frac{\pi}{3}\right) \\ \sin\left(-\frac{\pi}{3}\right) & \cos\left(-\frac{\pi}{3}\right) \end{pmatrix} y + \left(\frac{5-3\sqrt{3}}{12}, \frac{7+\sqrt{3}}{12}\right)^T = \\ &= \frac{1}{3} \begin{pmatrix} \frac{1}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{1}{2} \end{pmatrix} y + \left(\frac{5-3\sqrt{3}}{12}, \frac{7+\sqrt{3}}{12}\right)^T \\ S_3 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} \cos\left(\frac{\pi}{3}\right) & -\sin\left(\frac{\pi}{3}\right) \\ \sin\left(\frac{\pi}{3}\right) & \cos\left(\frac{\pi}{3}\right) \end{pmatrix} y + \left(\frac{5-\sqrt{3}}{12}, \frac{3-\sqrt{3}}{12}\right)^T = \\ &= \frac{1}{3} \begin{pmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{pmatrix} y + \left(\frac{5-\sqrt{3}}{12}, \frac{3-\sqrt{3}}{12}\right)^T \end{aligned}$$

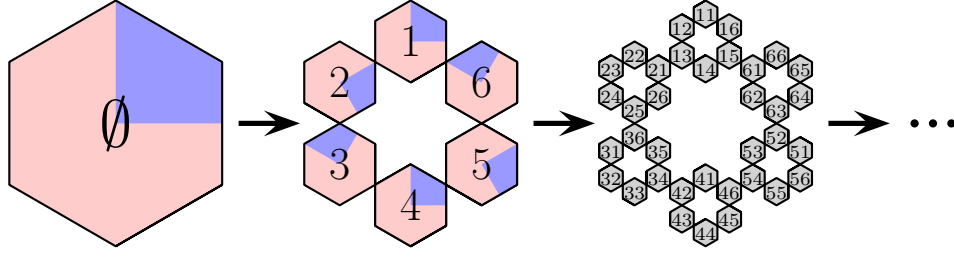


Figure 5.3: The word space of the Hexagasket corresponding to counter 29 393. As in previous figures, the coloring should help to understand the orientation and the reflection.

$$\begin{aligned}
 S_4 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} \cos(0) & -\sin(0) \\ \sin(0) & \cos(0) \end{pmatrix} y + \left(\frac{1}{3}, 0\right)^T = \\
 &= \frac{1}{3}y + \left(\frac{1}{3}, 0\right)^T \\
 S_5 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} \cos\left(-\frac{\pi}{3}\right) & -\sin\left(-\frac{\pi}{3}\right) \\ \sin\left(-\frac{\pi}{3}\right) & \cos\left(-\frac{\pi}{3}\right) \end{pmatrix} y + \left(\frac{5 + \sqrt{3}}{12}, \frac{3 + \sqrt{3}}{12}\right)^T = \\
 &= \frac{1}{3} \begin{pmatrix} \frac{1}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{1}{2} \end{pmatrix} y + \left(\frac{5 + \sqrt{3}}{12}, \frac{3 + \sqrt{3}}{12}\right)^T \\
 S_6 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} \cos\left(\frac{\pi}{3}\right) & -\sin\left(\frac{\pi}{3}\right) \\ \sin\left(\frac{\pi}{3}\right) & \cos\left(\frac{\pi}{3}\right) \end{pmatrix} y + \left(\frac{5 + 3\sqrt{3}}{12}, \frac{7 - \sqrt{3}}{12}\right)^T = \\
 &= \frac{1}{3} \begin{pmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{pmatrix} y + \left(\frac{5 + 3\sqrt{3}}{12}, \frac{7 - \sqrt{3}}{12}\right)^T,
 \end{aligned}$$

and equals the IFS with counter 29 393 in the computer algorithm, where we established a connection between a natural number and a particular IFS for an easier handling. For example, we can gain all information about this particular IFS of the Hexagasket, if we run the program `info_main.py` (see Listing A.6) to this counter.

The IFS and its word space is also plotted in Figure 5.3, which might be more useful for a better understanding of the IFS and the word space. The essential word space  $\mathfrak{W}$  consists then of the words

$$\begin{aligned}
 \mathfrak{W} &= \{13, 21, 25, 36, 34, 42, 46, 54, 52, 63, 61, 15 \in \mathcal{W} \text{ with} \\
 &13 \sim 21, 25 \sim 36, 34 \sim 42, 46 \sim 54, 52 \sim 63, 61 \sim 15\}.
 \end{aligned}$$

We follow from this that the essential equations

$$x_1x_3 = x_2x_1$$

$$x_2x_5 = x_3x_6$$

$$x_3x_4 = x_4x_2$$

$$x_4x_6 = x_5x_4$$

$$x_5x_2 = x_6x_3$$

$$x_6x_1 = x_1x_5$$

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 1$$

with  $x_a > 0, a = 1, \dots, 6$ , have to be fulfilled. Normally we would solve these equations in the usual manner as we did it in previous examples like in Example 4.3.3. We won't do this here, instead we apply Theorem 5.2.9.

Before we apply Method 5.2.6, we initialize the free variable, the fixed variables, the free equations and the fixed equations:

$$\text{free\_variables} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$$

$$\text{fixed\_variables} = \{ \}$$

$$\text{free\_equations} = \{x_1x_3 = x_2x_1, x_2x_5 = x_3x_6, x_3x_4 = x_4x_2,$$

$$x_4x_6 = x_5x_4, x_5x_2 = x_6x_3, x_6x_1 = x_1x_5$$

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 1\}$$

$$\text{fixed\_equations} = \{ \}.$$

We simplify this with the help of Method 5.2.5 and we also transfer some variables from free\_variables to fixed\_variables. After this we receive:

$$\text{free\_variables} = \{x_2, x_4, x_5\}$$

$$\text{fixed\_variables} = \{x_1, x_3, x_6\}$$

$$\text{free\_equations} = \{ \}$$

$$\text{fixed\_equations} = \{x_6 = x_5, x_3 = x_2, x_1 = 1 - x_2 - x_3 - x_4 - x_5 - x_6\}.$$

It holds that for every fixed variable there is one particular fixed equation which defines the dependencies of this variable. So, for example, the variable  $x_1$  has to be equal

$$1 - x_2 - x_3 - x_4 - x_5 - x_6.$$

In terms of Definition 5.2.7 we now consider the function  $F$  defined as

$$F : (0, 1)^3 \times (0, 1)^3 \rightarrow \mathbb{R}^3,$$

$$(x_2, x_4, x_5) \times (x_1, x_3, x_6) \mapsto \begin{pmatrix} x_6 - x_5 \\ x_3 - x_2 \\ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 - 1 \end{pmatrix}$$

It holds that the size of `fixed_variables` and the size of `fixed_equations` are equal. Therefore we can set up the partial Jacobian matrix

$$J(x_1, \dots, x_6) = \frac{\partial F}{\partial (x_1, x_3, x_6)}(x_1, \dots, x_6) = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

with  $\det(J(x_1, \dots, x_6)) = 1$  for all  $(x_1, \dots, x_6) \in \mathbb{R}^6$ . For the homogeneous solution point  $(\frac{1}{6}, \dots, \frac{1}{6})$  it follows in particular  $\det\left(J\left(\frac{1}{6}, \dots, \frac{1}{6}\right)\right) = 1$

We can apply Theorem 5.2.9 and we can choose 3 free variables, in particular  $x_2, x_4$  and  $x_5$ .

In the case of  $\bar{M} \neq m$  we can not apply Theorem 5.2.9 directly. Obviously it holds that the partial Jacobian matrix  $J$  of  $F$  is non-quadratic and hence not invertible, which is the key of the Implicit Function Theorem 5.2.8.

This gives at the same time a glue, how we could apply Theorem 5.2.9 even in the case  $\bar{M} \neq m$ . In fact only the case  $m < \bar{M}$  occurs. The main idea is the following: we modify our problem in such a way that the partial Jacobian matrix becomes quadratic. This is formulated in the next corollary.

**Corollary 5.2.11.** *We consider the same setting as in Theorem 5.2.9 and facing the problem with  $k$  free variables,  $m$  fixed variables and the function*

$$F : (0, 1)^k \times (0, 1)^m \rightarrow \mathbb{R}^{\bar{M}}$$

with  $m < \bar{M}$ .

We set  $m_0 := \bar{M} - m$ , which is the number of missing variables such that  $J$  is quadratic.

We choose  $m_0$  variables from the fixed variables and transfer them to the free variables. Therefore we have  $k_0 := k + m_0$  free variables and  $\bar{M}$  fixed variables. For the function  $F$  follows

$$F : (0, 1)^{k_0} \times (0, 1)^{\bar{M}} \rightarrow \mathbb{R}^{\bar{M}}$$

and we can apply Theorem 5.2.9 with  $F$  and  $k_0$  free variables and  $\bar{M}$  fixed variables.

If  $\det J(y_{hom}) \neq 0$  holds, we have  $k_0$  free parameters. Otherwise we can check another subset of  $m_0$  free variables.

Up to now the case  $m < \bar{M}$  did not occur in practice and we conclude from this, that the functionality of Method 5.2.6 is quite good. Nevertheless, we still hold on to Corollary 5.2.11 since we use it in the next remark. For this remark we make some prior observations. First of all, we can implement Method 5.2.6 and Theorem 5.2.9 in a computer and the corresponding algorithm determines the number of free parameters in a valid way. At the same time it holds that the algorithm and especially Method 5.2.6 need a lot of calculation time with the Python package SymPy. If we take a closer look, we see that the simplification of equation

$$x_1 + \cdots + x_N = 1 \tag{5.6}$$

consumes most of the time. This comes from the fact that we substitute some of the variables into this equation and SymPy needs extraordinary time (we speak here from hours to days for one simplification!) to simplify or resolve this equation.

For this reason we implement our algorithm in such a way, that we exclude the Equation (5.6) from the simplifications. We sum this up in the following remark.

**Remark 5.2.12.** *We consider a specific IFS representing a fractal. We set up the essential word space  $\mathfrak{W}$  of this IFS and the equations which have to be fulfilled regarding to Theorem 4.1.4.*

*We simplify the essential equations without*

$$x_1 + \cdots + x_N = 1$$

*using Method 5.2.6. After this, we have a list of free variables, fixed variables, free equations and fixed equations. We add the summation equation*

$$x_1 + \cdots + x_N = 1$$

*to the list of fixed equations.*

*We apply Corollary 5.2.11, since the size of fixed variables and fixed equations differ by at least one and we may have to select multiple times a subset of free variables which are transferred.*

With this remark we finally set up and developed a mathematical background which can be implemented at a computer and has a sufficient good calculation time. To understand

Remark 5.2.12 in a better way, we include the next example.

**Example 5.2.13** (Pentagasket, counter 519). *We consider the Pentagasket and apply Remark 5.2.12 to it. The Pentagasket consists of five small copies of a (regular) pentagon in the plane and the small copies are arranged in the corner such that the center is empty. Later, we also consider the case with a filled center.*

*The five copies are shrank in the way that they touch neighboring cells in exact one point. We can do a small calculation for the contraction ratio and get*

$$\frac{3 - \sqrt{5}}{2} = \frac{1}{\Phi^2} \approx 0.382,$$

where  $\Phi = \frac{1+\sqrt{5}}{2}$  is the golden ratio.

*Of course, we can arrange the small copies in a lot of possible ways and we consider in this example the Pentagasket which equals in the computer algorithm the counter 519.*

*The mappings are then:*

$$\begin{aligned} S_1 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad x \mapsto \frac{3 - \sqrt{5}}{2}x + \begin{pmatrix} \frac{-1+\sqrt{5}}{4} \\ \frac{\sqrt{5}-\sqrt{5}}{2\sqrt{2}} \end{pmatrix} \\ S_2 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad x \mapsto \frac{3 - \sqrt{5}}{2}x + \begin{pmatrix} 0 \\ \frac{\sqrt{5}-2\sqrt{5}}{2} \end{pmatrix} \\ S_3 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad x \mapsto \frac{3 - \sqrt{5}}{2} \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix} \\ S_4 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad x \mapsto \frac{3 - \sqrt{5}}{2} \begin{pmatrix} \frac{-1+\sqrt{5}}{4} & \frac{\sqrt{5}+\sqrt{5}}{2\sqrt{2}} \\ -\frac{\sqrt{5}+\sqrt{5}}{2\sqrt{2}} & \frac{-1+\sqrt{5}}{4} \end{pmatrix} x + \begin{pmatrix} \frac{15-5\sqrt{5}}{8} \\ \frac{\sqrt{5}-\sqrt{5}}{4\sqrt{2}} \end{pmatrix} \\ S_5 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad x \mapsto \frac{3 - \sqrt{5}}{2} \begin{pmatrix} \frac{1-\sqrt{5}}{4} & -\frac{\sqrt{5}+\sqrt{5}}{2\sqrt{2}} \\ -\frac{\sqrt{5}+\sqrt{5}}{2\sqrt{2}} & \frac{-1+\sqrt{5}}{4} \end{pmatrix} x + \begin{pmatrix} \frac{-3+5\sqrt{5}}{8} \\ \frac{\sqrt{5}\sqrt{5}-\sqrt{5}}{4\sqrt{2}} \end{pmatrix}. \end{aligned}$$

*We can apply these mappings to a pentagon and denote at the same time the word space, which is plotted in Figure 5.4. This figure illustrates also the IFS in a graphical way. We can determine the essential word space as*

$$\begin{aligned} \mathfrak{W} &= \{13, 25, 24, 31, 32, 43, 42, 55, 51, 14 \in \mathcal{W} \text{ with} \\ &\quad 13 \sim 25, 24 \sim 31, 32 \sim 43, 42 \sim 55, 51 \sim 14\} \end{aligned}$$

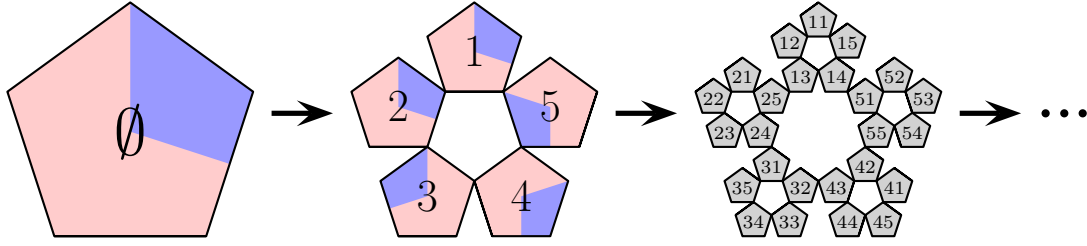


Figure 5.4: The word space of the Pentagasket corresponding to counter 519. As in previous figures, the coloring should help to understand the orientation and the reflection.

and we follow by Theorem 4.1.4 that the essential equations

$$\begin{aligned}
 x_1x_3 &= x_2x_5 \\
 x_2x_4 &= x_3x_1 \\
 x_3x_2 &= x_4x_3 \\
 x_4x_2 &= x_5x_5 \\
 x_5x_1 &= x_1x_4 \\
 x_1 + x_2 + x_3 + x_4 + x_5 &= 1
 \end{aligned} \tag{5.7}$$

with  $x_i > 0$  for  $i = 1, \dots, 5$  have to be fulfilled.

We apply Remark 5.2.12 to (5.7). From this we receive:

$$\begin{aligned}
 \text{free\_variables} &= \{x_1, x_2\} \\
 \text{fixed\_variables} &= \{x_3, x_4, x_5\} \\
 \text{free\_equations} &= \{ \} \\
 \text{fixed\_equations} &= \left\{ x_4 = x_5, x_2 = x_4, x_3 = \frac{x_2^2}{x_1}, \right. \\
 &\quad \left. x_1 + x_2 + x_3 + x_4 + x_5 = 1 \right\}
 \end{aligned} \tag{5.8}$$

and we have to transfer one variable from `free_variables` to `fixed_variables` in order to apply Theorem 5.2.9. This can be either  $x_1$  or  $x_2$ .

In a first try we transfer  $x_1$  and receive the function  $F$  as

$$F_1 : (x_2) \times (x_1, x_3, x_4, x_5) \mapsto \begin{pmatrix} x_4 - x_5 \\ x_2 - x_4 \\ x_3 - \frac{x_2^2}{x_1} \\ x_1 + x_2 + x_3 + x_4 + x_5 - 1 \end{pmatrix}$$

We set up the partial Jacobian matrix  $J_1$  of  $F_1$  and receive

$$J_1(x_1, \dots, x_5) = \frac{\partial F_1(x_1, \dots, x_5)}{\partial (x_1, x_3, x_4, x_5)} = \begin{pmatrix} 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 0 \\ \frac{x_2^2}{x_1} & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}.$$

We calculate the determinant and it follows that

$$\det(J_1(x_1, \dots, x_5)) = 1 - \frac{x_2^2}{x_1^2} \quad (5.9)$$

holds. We have to substitute the homogeneous solution in (5.9) and thus

$$\det\left(J_1\left(\frac{1}{5}, \dots, \frac{1}{5}\right)\right) = 0$$

follows. This implies that we can not apply the Implicit Function Theorem 5.2.9 if we transfer  $x_1$ .

For this reason we go back to the point, where we select the variable to transfer. This time we choose the variable  $x_2$  to transfer in (5.8).

We set up the function  $F_2$  as

$$F_2 : (x_1) \times (x_2, x_3, x_4, x_5) \mapsto \begin{pmatrix} x_4 - x_5 \\ x_2 - x_4 \\ x_3 - \frac{x_2^2}{x_1} \\ x_1 + x_2 + x_3 + x_4 + x_5 - 1 \end{pmatrix}$$



and the associated partial Jacobian

$$J_2(x_1, \dots, x_5) = \frac{\partial F_2(x_1, \dots, x_5)}{\partial(x_2, x_3, x_4, x_5)} = \begin{pmatrix} 0 & 0 & 1 & -1 \\ 1 & 0 & -1 & 0 \\ -\frac{2x_2}{x_1} & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}.$$

We calculate the determinant of  $J_2$  as

$$\det(J_2(x_1, \dots, x_5)) = 1 + \frac{2x_2}{x_1}.$$

At the homogeneous solution follows

$$\det\left(J_2\left(\frac{1}{5}, \dots, \frac{1}{5}\right)\right) = 3. \quad (5.10)$$

We can apply Theorem 5.2.9, since in Equation (5.10)  $\det J_2 \neq 0$  holds. It follows that this IFS has 1 free parameter and 4 fixed parameters. This information is sufficient for us in the general context, but we can take a closer look at the dependencies.

We can choose the free variable

$$x_1 \in (0, 1)$$

and resolve the fixed variables w.r.t.  $x_1$ . Therefore we have:

$$\begin{aligned} x_2 &= -\frac{3}{2}x_1 + \sqrt{\frac{5}{4}x_1^2 + x_1}, \\ x_3 &= \frac{7}{2}x_1 + 1 - 3\sqrt{\frac{5}{4}x_1^2 + x_1}, \\ x_4 &= x_2 = -\frac{3}{2}x_1 + \sqrt{\frac{5}{4}x_1^2 + x_1}, \\ x_5 &= x_2 = -\frac{3}{2}x_1 + \sqrt{\frac{5}{4}x_1^2 + x_1}. \end{aligned} \quad (5.11)$$

The implicit function  $G$  from the Implicit Function Theorem 5.2.8 can be created with

(5.11) and it follows

$$G : (0, 1) \rightarrow (0, 1)^4, x_1 \mapsto \begin{pmatrix} x - \frac{3}{2}x_1 + \sqrt{\frac{5}{4}x_1^2 + x_1} \\ \frac{7}{2}x_1 + 1 - 3\sqrt{\frac{5}{4}x_1^2 + x_1} \\ -\frac{3}{2}x_1 + \sqrt{\frac{5}{4}x_1^2 + x_1} \\ -\frac{3}{2}x_1 + \sqrt{\frac{5}{4}x_1^2 + x_1} \end{pmatrix}.$$

This shows us again, why we use the term “free variable”: the variable  $x_1$  can be chosen in a free way and the fixed variables are determined by the free variable(s).

The previous considerations allow us to determine the number of free parameters. One thing still remains, which is the relatively high computing time.

We can speed up our algorithm if we split up our problem into multiple parts. To be precise, we split this up in the number of kernels of the computer and each kernel receives a part of the calculation. Hence we can calculate parallel. A further acceleration would be possible, if we split up our algorithm to multiple computers, but we skip this here since it would make our algorithm harder to understand and the essential part of splitting up is already contained in the process of parallel computing.

All together we can set up a pseudo algorithm, which can be found in Listing 5.2. In Line 25 is a statement for raising a Warning. Up to now, this has not occurred at any considered fractal. If a warning appears we should extend Method 5.2.5 or determine why this problem occurs. For this reason the function `information_to_fractal` is provided (see Listing A.6).

The complete Python-Code can be found in Appendix A. The module `B2.py` (see Listing A.2) contains all needed functions used by the preprocessing in `preprocessing.py` (see Listing A.10), the main program in `main.py` (see Listing A.7) and the postprocessing in `postprocessing.py` (see Listing A.9). The preprocessing determines the equivalent IFSs, the main program calculates the number of free parameters to each representative. The postprocessing collects all results. All in all, this can be also combined in one program `main_complete.py` (see Listing A.8).

The provided algorithm allows us to determine the number of free variables in total. Since we are also able to gather the free variables this algorithm is in some sense constructive. In Section 5.3 we will discuss the results of the algorithm. But before we do so, we consider a similar algorithm in the next subsection.

```

1 initialize the fractal
2 all_IFS := list of all possible IFSs
3
4 all_eq_classes := determine all equivalence classes of all_IFS (compare to
   Listing 5.1)
5
6 for each kernel:
7     pick an element from all_eq_classes which was until now not
       considered:
8         set up the list of essential equations associated to the IFS
9         # we apply Remark 5.2.12, in particular:
10        apply Method 5.2.6 to the list of equations except the summation
           equation and receive free_equations, fixed_equations,
           free_variables, fixed_variables (with the notation of Definition
           5.2.7)
11
12        add the summation equation to the list of fixed_equation.
13
14        m_0 := length(all_equations) – length(fixed_variables)
15
16        # we use Corollary 5.2.11:
17        for all combinations of subsets of free_variables with size m_0:
18            new_fixed_variables := combine variables in combination and
               fixed_variables
19            new_free_variables := all free_variables except variables in
               combination
20            set up the partial Jacobian matrix J
21            if det(J(y_hom)) ≠ 0:
22                save length(new_free_variables) as the number of free
                   parameters
23                stop for–loop from Line 17
24            if no number of free parameters was determined:
25                raise Warning
26 merge results of the different kernels
27 return list with IFSs, number of free parameters and length of its
       equivalence class

```

Listing 5.2: Pseudo algorithm for calculating the number of free parameters in a constructive way for all IFSs generating a specific fractal.

## 5.2.2 With an existential algorithm

In the previous subsection we developed an algorithm, which determines the number of free parameters to a particular fractal. This algorithm looks at all possible realizations which create the same attractor. This is done in a constructive way and afterwards we can say, which weights can be chosen. This is indeed a nice information, but sometimes not needed. The price for this information is a relatively high computing time.

In this subsection we develop a computer algorithm, which does not provide this information. At the same time this algorithm is faster. Of course, both the algorithms determine the same number of free parameters, since this is an unique property of each IFS.

We call this algorithm the “existential algorithm”, since we determine the number of free parameters in an existential way, see Theorem 5.2.15 below.

The idea for the algorithm is in some sense based on Remark 5.2.12, where we choose a subset of variables in such a way that the Jacobian matrix is invertible. This is the crux of the whole problem. The idea is the following: we calculate the rank of the Jacobian. Then, we select a suitable set of equations and variables such that the Jacobian is invertible at the homogeneous solution. If this is the case, the selected variables are fixed and therefore we can choose the remaining variables in a free way. This can be done directly, if we consider the rank of the Jacobian in the homogeneous solution as the number of fixed variables.

First of all we start with some notations, which is quite similar to the notation in Subsection 5.2.1.

For the rest of this subsection we fix a certain fractal and a certain IFS.

**Definition 5.2.14.** *We consider the Essential Equations (5.3)*

$$\begin{aligned} g_1(x) &= h_1(x) \\ &\vdots \\ g_M(x) &= h_M(x) \end{aligned}$$

and define  $F_i(x) := g_i(x) - h_i(x)$  for  $i = 1, \dots, M$ . Further, we define

$$F : (0, 1)^N \rightarrow \mathbb{R}^M, \quad x \mapsto F(x) = \begin{pmatrix} F_1(x) \\ \vdots \\ F_M(x) \end{pmatrix}, \quad (5.12)$$

which is continuous and differentiable function and the Jacobian matrix  $J$  by

$$J(x) := \frac{\partial F}{\partial x}(x) = \begin{pmatrix} \frac{\partial F_1}{\partial x_1}(x) & \cdots & \frac{\partial F_1}{\partial x_N}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial F_M}{\partial x_1}(x) & \cdots & \frac{\partial F_M}{\partial x_N}(x) \end{pmatrix}.$$

We notice, that the function  $F$  in (5.12) is defined in a slightly different way than in Definition 5.2.7. The equations are not simplified before the definition, instead we consider  $F$  straight afterwards. This means in practice, that the codomain is relatively large. However, this is not problematic, as we are only interested at the rank of  $J$ .

**Theorem 5.2.15.** *We consider a certain IFS, the function  $F$  as in Definition 5.2.14 and the associated Jacobian matrix  $J$ . Further we define  $x_{\text{hom}} := \left(\frac{1}{N}, \dots, \frac{1}{N}\right) \in \mathbb{R}^N$  and  $m := \text{rank}(J(x_{\text{hom}}))$ .*

*Then, it holds that the number of free parameters equals  $N - m$ .*

*Proof.* The idea of the proof follows mainly Remark 5.2.12. In contrast to the remark, we now have to select multiple variables and equations.

We consider the function

$$F : (0, 1)^N \rightarrow \mathbb{R}^M, \quad x \mapsto F(x) = \begin{pmatrix} F_1(x) \\ \vdots \\ F_M(x) \end{pmatrix}$$

as in Definition 5.2.14 and notice that  $F$  is a continuous and differentiable function.

We denote by  $x_{\text{hom}} := \left(\frac{1}{N}, \dots, \frac{1}{N}\right) \in (0, 1)^N$ ,  $m := \text{rank}(J(x_{\text{hom}}))$  and  $k := N - m$ .

From  $\text{rank}(J(x_{\text{hom}})) = m$  follows that there are subsets (also called index sets)

$$\begin{aligned} A &\subseteq \mathcal{A}, |A| = k, A = \{\alpha_1, \dots, \alpha_k\} \\ B &:= \mathcal{A} \setminus A, |B| = m, B = \{\beta_1, \dots, \beta_m\} \\ C &\subseteq \{1, \dots, M\}, |C| = m, C = \{\gamma_1, \dots, \gamma_m\} \end{aligned}$$

such that

$$\hat{J}(x) := \begin{pmatrix} \frac{\partial F_{\gamma_1}}{\partial x_{\beta_1}}(x) & \cdots & \frac{\partial F_{\gamma_1}}{\partial x_{\beta_m}}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial F_{\gamma_m}}{\partial x_{\beta_1}}(x) & \cdots & \frac{\partial F_{\gamma_m}}{\partial x_{\beta_m}}(x) \end{pmatrix}$$

is invertible in  $x_{\text{hom}}$ , i.e.

$$\det(\hat{J}(x_{\text{hom}})) \neq 0. \tag{5.13}$$

The index set  $A$  describes which variables are free and  $B$  describes which variables are fixed. The set  $C$  selects those equations which determine a fixed variable. We introduce a projection  $\pi$ , which splits a vector in the components in the right way:

$$\pi : (0, 1)^N \rightarrow (0, 1)^k \times (0, 1)^m, \quad (x_1, \dots, x_N) \mapsto (x_{\alpha_1}, \dots, x_{\alpha_k}, x_{\beta_1}, \dots, x_{\beta_m}).$$

With this, we can build up a function  $\bar{F}$ , which partial Jacobian matrix equals  $\hat{J}$ . We define

$$\begin{aligned} \bar{F} : (0, 1)^k \times (0, 1)^m &\rightarrow \mathbb{R}^m, \\ (x_{\alpha_1}, \dots, x_{\alpha_k}, x_{\beta_1}, \dots, x_{\beta_m}) &\mapsto \bar{F}(x_{\alpha_1}, \dots, x_{\beta_m}) = \begin{pmatrix} \bar{F}_1(x_{\alpha_1}, \dots, x_{\beta_m}) \\ \vdots \\ \bar{F}_m(x_{\alpha_1}, \dots, x_{\beta_m}) \end{pmatrix} \end{aligned}$$

with

$$\bar{F}_i(x_{\alpha_1}, \dots, x_{\beta_m}) := F_{\gamma_i}(\pi^{-1}(x_{\alpha_1}, \dots, x_{\beta_m})) \quad \text{for } i = 1, \dots, m$$

We determine the partial Jacobian matrix by

$$\bar{J}(x_{\alpha_1}, \dots, x_{\beta_m}) = \begin{pmatrix} \frac{\partial \bar{F}_1}{\partial x_{\beta_1}}(x_{\alpha_1}, \dots, x_{\beta_m}) & \dots & \frac{\partial \bar{F}_1}{\partial x_{\beta_m}}(x_{\alpha_1}, \dots, x_{\beta_m}) \\ \vdots & \ddots & \vdots \\ \frac{\partial \bar{F}_m}{\partial x_{\beta_1}}(x_{\alpha_1}, \dots, x_{\beta_m}) & \dots & \frac{\partial \bar{F}_m}{\partial x_{\beta_m}}(x_{\alpha_1}, \dots, x_{\beta_m}) \end{pmatrix}$$

and observe that  $\hat{J}(x) = \bar{J}(\pi(x))$  and hence

$$\bar{J}(x_{\alpha_1}, \dots, x_{\beta_m}) = \hat{J}(\pi^{-1}(x_{\alpha_1}, \dots, x_{\beta_m})) \quad (5.14)$$

holds.

In the last step we consider the determinant of  $\bar{J}$  at the homogeneous point. It follows that

$$\det \bar{J}(x_{\text{hom}}) = \det \hat{J}(x_{\text{hom}}) \neq 0$$

holds, where we first used Equation (5.14) and then (5.13). This implies that  $\bar{J}$  is invertible at  $(\frac{1}{N}, \dots, \frac{1}{N})$ . We can apply the Implicit Function Theorem 5.2.9 to  $\bar{F}$  and the IFS has  $k = N - \text{rank } J(x_{\text{hom}})$  free parameters.  $\square$

This theorem allows us in a very fast and simple way to calculate the number of free parameters. At the same time we get no information, which parameters are free.

We can build with Theorem 5.2.15 a computer algorithm. Again, we only consider one

```

1 initialize the fractal
2 all_IFS := list of all possible IFSs
3
4 all_eq_classes := determine all equivalence classes of all_IFS (compare to
   Listing 5.1)
5
6 for each kernel:
7     pick an element from all_eq_classes which was until now not
       considered:
8         set up the list of essential equations associated to the IFS
9         set up the function F regarding to Definition 5.2.14
10        set up the Jacobian J of F
11        # We apply Theorem 5.2.15
12        m := rank(J(x_hom))
13        # The number of free parameters can then be determined:
14        fp := N - m
15        save fp as the number of free parameters
16 merge results of the different kernels
17 return list with IFSs, number of free parameters and length of its
       equivalence class

```

Listing 5.3: Pseudo algorithm for calculating the number of free parameters in an existential way for all IFSs generating a specific fractal.

representative of equivalent IFSs. This is done before the main task. After this, we set up the notations and essential equations for each IFS and calculate the rank of the partial Jacobian matrix. All in all, we can build up a pseudo algorithm which can be found in Listing 5.3. The source code for the calculation is contained in Appendix A, especially if we run in the first place the program `preprocessing.py` (see Listing A.10) and afterwards the main program `main.py` (see Listing A.7) with “mode=existential”. The benefit of this splitted calculation is the fact that we can consider the equivalence classes, before we start the calculation. Alternatively we can run the program `main_complete.py` (see Listing A.8) with option “mode=existential”, where both tasks are done successively. In the following we provide an example to Theorem 5.2.15 (resp. Listing 5.3), which should help to understand the way, we calculate free parameters.

**Example 5.2.16** (filled Hexagasket, counter 16 116). *We consider the filled Hexagasket, or sometimes called Hexagasket without hole. The filled Hexagasket is quite similar to the normal Hexagasket, which we already considered in Example 5.2.10, but the center is filled and we have in total seven similarities, respectively  $N = 7$ .*

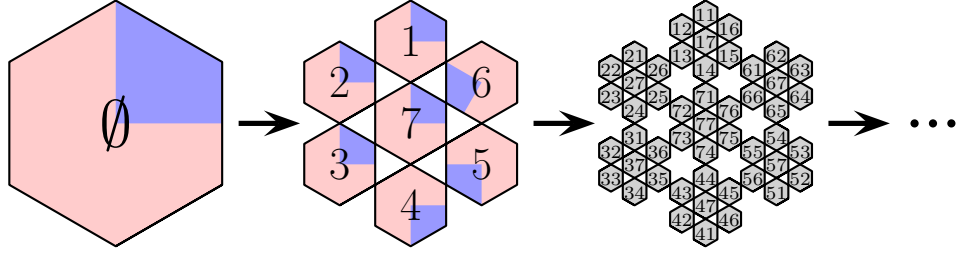


Figure 5.5: The word space of the filled Hexagasket, which equals counter 16 116. As in previous figures, the coloring should help to understand the orientation and the reflection.

As always, we can arrange the mappings in multiple ways and in this example we consider the filled Hexagasket which equals the counter 16 116 in the computer algorithm. The seven similarities are then:

$$\begin{aligned}
 S_1 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \begin{pmatrix} \frac{1}{3} \\ \frac{2}{3} \end{pmatrix} \\
 S_2 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \begin{pmatrix} \frac{1}{3} - \frac{\sqrt{3}}{6} \\ \frac{1}{2} \end{pmatrix} \\
 S_3 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \begin{pmatrix} \frac{1}{3} - \frac{\sqrt{3}}{6} \\ \frac{1}{6} \end{pmatrix} \\
 S_4 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} y + \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \end{pmatrix} \\
 S_5 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} y + \begin{pmatrix} \frac{\sqrt{3}}{6} + \frac{2}{3} \\ \frac{1}{2} \end{pmatrix} \\
 S_6 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3} \begin{pmatrix} -\frac{1}{2} & -\frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{1}{2} \end{pmatrix} y + \begin{pmatrix} \frac{\sqrt{3}}{4} + \frac{7}{12} \\ \frac{\sqrt{3}}{12} + \frac{7}{12} \end{pmatrix} \\
 S_7 : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \quad y \mapsto \frac{1}{3}y + \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \end{pmatrix}.
 \end{aligned}$$

For a better understanding we plot again the word space up to length 2. This can be found in Figure 5.5.



We determine the essential word space as

$$\begin{aligned} \mathfrak{W} = \{ & 13, 26, 14, 71, 15, 61, 24, 31, 25, 72, 35, 43, \\ & 36, 73, 44, 74, 45, 56, 54, 65, 55, 75, 66, 76 \in \mathcal{W} \text{ with} \\ & 13 \sim 26, 14 \sim 71, 15 \sim 61, 24 \sim 31, 25 \sim 72, 35 \sim 43, \\ & 36 \sim 73, 44 \sim 74, 45 \sim 56, 54 \sim 65, 55 \sim 75, 66 \sim 76\} \end{aligned}$$

and the essential equations as

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 &= 1 \\ x_1x_3 &= x_2x_6 \\ x_1x_4 &= x_7x_1 \\ x_1x_5 &= x_6x_1 \\ x_2x_4 &= x_3x_1 \\ x_2x_5 &= x_7x_2 \\ x_3x_5 &= x_4x_3 \\ x_3x_6 &= x_7x_3 \\ x_4x_4 &= x_7x_4 \\ x_4x_5 &= x_5x_6 \\ x_5x_4 &= x_6x_5 \\ x_5x_5 &= x_7x_5 \\ x_6x_6 &= x_7x_6 \end{aligned}$$

with  $x_i > 0$  for  $i = 1, \dots, 7$ . We define the function  $F$  regarding to (5.12), which is in this case

$$F : (0, 1)^7 \rightarrow \mathbb{R}^{13}, \quad x \mapsto F(x) = \begin{pmatrix} x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 - 1 \\ x_1x_3 - x_2x_6 \\ x_1x_4 - x_7x_1 \\ x_1x_5 - x_6x_1 \\ x_2x_4 - x_3x_1 \\ x_2x_5 - x_7x_2 \\ x_3x_5 - x_4x_3 \\ x_3x_6 - x_7x_3 \\ x_4x_4 - x_7x_4 \\ x_4x_5 - x_5x_6 \\ x_5x_4 - x_6x_5 \\ x_5x_5 - x_7x_5 \\ x_6x_6 - x_7x_6 \end{pmatrix}.$$

*F* is then continuous and differentiable, which is an important fact, since we can then apply the Implicit Function Theorem.

The Jacobian matrix is then

$$J(x) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ x_3 & -x_6 & x_1 & 0 & 0 & -x_2 & 0 \\ x_4 - x_7 & 0 & 0 & x_1 & 0 & 0 & -x_1 \\ x_5 - x_6 & 0 & 0 & 0 & x_1 & -x_1 & 0 \\ -x_3 & x_4 & -x_1 & x_2 & 0 & 0 & 0 \\ 0 & x_5 - x_7 & 0 & 0 & x_2 & 0 & -x_2 \\ 0 & 0 & -x_4 + x_5 & -x_3 & x_3 & 0 & 0 \\ 0 & 0 & x_6 - x_7 & 0 & 0 & x_3 & -x_3 \\ 0 & 0 & 0 & 2x_4 - x_7 & 0 & 0 & -x_4 \\ 0 & 0 & 0 & x_5 & x_4 - x_6 & -x_5 & 0 \\ 0 & 0 & 0 & x_5 & x_4 - x_6 & -x_5 & 0 \\ 0 & 0 & 0 & 0 & 2x_5 - x_7 & 0 & -x_5 \\ 0 & 0 & 0 & 0 & 0 & 2x_6 - x_7 & -x_6 \end{pmatrix}.$$

We can determine the rank of *J* at  $x_{\text{hom}}$  (for example with the help of a computer algebra system) and receive

$$\text{rank } J(x_{\text{hom}}) = 2.$$

We apply Theorem 5.2.15 and it follows that this IFS has 2 free parameters.

*If we are interested, which parameters we can choose, we must apply the constructive algorithm from Subsection 5.2.1.*

As the Example 5.2.16 shows, the algorithm determines the free parameters in a fast way. We only have to set up the notation and the matrix  $J$  may be relatively large to handle by hand. But this is no problem for the computer algorithm, since this is done automatically.

All in all we now have two algorithms to determine the number of free parameters.

### 5.3 Summarized results of both the algorithms

In this section we discuss the results of both the algorithms of Listings 5.2 and 5.3. Of course, both the algorithms determine the same number of free parameters. Because of this, we will omit the used algorithm if we take a closer look at the number of free parameters of some fractals. In the second part of this section we compare the computing time of both the algorithms.

Let us now apply our algorithm to several fractals. Of course, we investigate the already introduced fractals. Further, we add some other common fractals and in total we consider the following fractals:

- the Sierpiński gasket;
- the 3-level Sierpiński gasket, often shorten by  $SG_3$ ;
- the 4-level Sierpiński gasket, often shorten by  $SG_4$ ;
- the Vicsek fractal (see [Vic92]);
- the Pentagasket (see [Ada+03; Ima00]);
- the filled Pentagasket, also known as Pentaflake;
- the Hexagasket (see [Str06]);
- the filled Hexagasket, also known as Lindstrøm snowflake (see [Lin90]);
- the Sierpiński tetrahedron (see also Example 3.2.13) and
- the Sierpiński carpet (see also Example 3.1.10).

As a small reminder contains Figure 5.6 (resp. the sub-figures) the associated attractors of these fractals in the unweighted case. We use for the Sierpiński carpet the modified definition of  $\sim$  as introduced in Example 3.1.10.

In a first step we collect some information about each fractal. With these information we can compare the different fractals in a solid way. These information are summarized in Table 5.1.

Table 5.1 contains the length of the alphabet  $N = |\mathcal{A}|$ , the number of different mappings for a small copy and the number of essential equations which have to be fulfilled. The number of essential equations stays the same, if we consider different mappings. Further, the table contains the number of equivalent IFSs, if we apply the algorithm from Listing 5.1 and the total number of IFSs.

There are some fractals, which are in one (or more) property outstanding. The 4-level Sierpiński gasket  $SG_4$  has an alphabet of length 10 which leads to a high number of equivalence classes and a high number of total IFSs. We will discuss this relation at a later point, see Equations (5.15) and (5.16). On the other hand there is the Sierpiński tetrahedron, which has a high number of essential equations but only a medium number of IFSs.

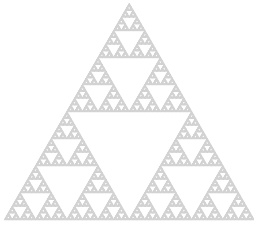
All this properties interferes with the number of free parameters, as we will see now.

We execute our algorithm to those fractals and receive for every fractal the amount of iterated function systems with a certain number of free parameters. In Table 5.2 the results are summarized.

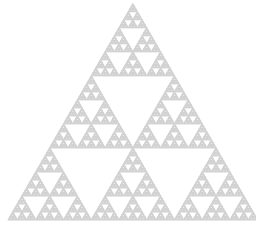
As we can see, there is only one realization of the Sierpiński gasket with two free parameters, which is exactly Example 4.2.3. Furthermore occur only 21 IFSs with one free parameter and the majority of cases have zero free parameters.

The 3-level Sierpiński gasket (abbreviated as  $SG_3$ ) is generated by six similarities and consequently five free parameters would be the maximum number of possible free parameters. Indeed this does not occur. Neither four, three or two free parameters do. As we can see, there are only 399 IFSs (which is about 0.9%) with one free parameters and hence this is very uncommon. The remaining IFSs have zero free parameters which implies, that those IFSs only fulfill (B2) in the homogeneous case. One of the reasons is the fact, that there are in total nine equations which have to be fulfilled, making it harder to find any solution besides the homogeneous case.

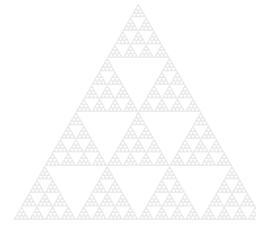
The 4-level Sierpiński gasket (abbreviated as  $SG_4$ ) is the logical extension in the series of the Sierpiński gasket and the 3-level Sierpiński gasket. In fact, the  $n$ -level Sierpiński gasket exists for  $n \in \mathbb{N}$ , where  $n$  represents the number of similarities at one edge of the



(a) The Sierpiński gasket.



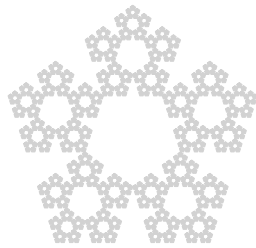
(b) The 3-level Sierpiński gasket.



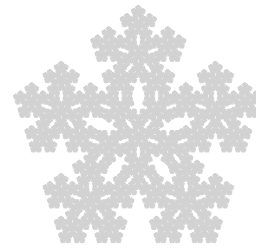
(c) The 4-level Sierpiński gasket.



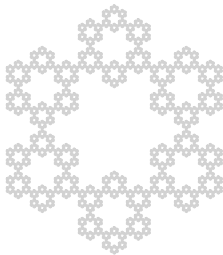
(d) The Vicsek fractal.



(e) The Pentagasket.



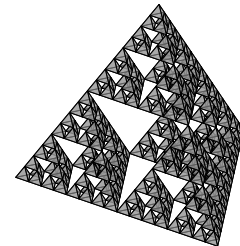
(f) The filled Pentagasket.



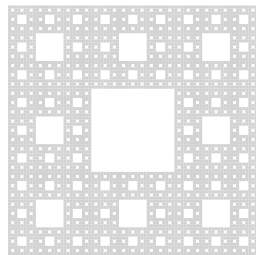
(g) The Hexagasket.



(h) The filled Hexagasket.



(i) The Sierpiński tetrahedron.



(j) The Sierpiński carpet.

Figure 5.6: The attractor of different fractals.

fractal	N	number of different mappings	number of essential equations	number of equiva- lence classes	total number of IFSs
Sierpiński gasket	3	6	4	44	216
SG <sub>3</sub>	6	6	10	7 860	46 656
SG <sub>4</sub>	10	6	19	10 080 504	60 466 176
Vicsek	5	8	5	4 360	32 768
Pentagasket	5	10	6	10 104	100 000
filled Pentagasket	6	10	26	100 220	1 000 000
Hexagasket	6	12	7	250 010	2 985 984
filled Hexagasket	7	12	13	2 991 900	35 831 808
Sierpiński tetrahe- dron	4	24	7	14 022	331 776
Sierpiński carpet	8	8	117	2 101 776	16 777 216

Table 5.1: Summarized information about different fractals.

fractal	number of free parameters						total number of IFSs
	0	1	2	3	4	5	
Sierpiński gasket	194	21	1	–	–	–	216
SG <sub>3</sub>	46 257	399	0	0	0	0	46 656
SG <sub>4</sub>	60 466 080	96	0	0	0	0	60 466 176
Vicsek	20 544	10 112	2 048	64	0	–	32 768
Pentagasket	88 025	11 875	100	0	0	–	100 000
filled Pentagasket	999 995	5	0	0	0	0	1 000 000
Hexagasket	2 599 398	361 007	24 075	1 452	51	1	2 985 984
filled Hexagasket	35 724 006	107 412	390	0	0	0	35 831 808
Sierpiński tetrahe- dron	326 010	5 675	90	1	–	–	331 776
Sierpiński carpet	16 775 164	2 048	4	0	0	0	16 777 216

Table 5.2: Summarized results about the number of free parameters on different fractals. A value of “–” indicates that this value can not occur, since the alphabet is too small.

triangle. The  $SG_4$  consists of ten similarities and there are 19 essential equations. This restricts the number of free parameters in a very sharp way and the maximal number of free parameters is one. In total there are only 96 IFSs with one free parameter.

On the Vicsek fractal the number of free parameters are wider distributed, for example there are 64 IFSs with three free parameters and over 12 160 IFSs with one or two free parameters. Nevertheless an IFS with the full potential of four free parameters is missing.

The Pentagasket behaves in a total other way than the Vicsek fractal, even if the alphabet has also a length of  $N = 5$ . Most IFSs have zero free parameters, about 11.8% have one free parameter and only 100 have two free parameters. Those 100 IFSs are contained in 14 different equivalence classes.

If we add a similarity in the middle, we receive the filled Pentagasket which is also known as Pentaflake. It is obvious, that this restricts the number of free parameters. Indeed this restricts the choice in a massive way. There is only one (!) equivalence class containing five IFSs, where we can choose one free parameter. At all other IFSs we are not able to choose any free parameter.

The Hexagasket has roughly the same geometrical structure as the Pentagasket, but with six similarities. The distribution of free parameters is relatively broad and we are indeed able to choose an IFS with the maximum of five free parameters. This stands in contrast to the Pentagasket where the number of essential equations is lower. It looks like the symmetry of the hexagon comes into account and allows more free parameter.

We observe another analogy at the filled Hexagasket, which was first introduced by Lindstrøm in [Lin90] and is sometimes called Lindstrøm snowflake. It is generated like the Hexagasket, but again (like the filled Pentagasket) with a filled center. The center restrict again the number of free parameters compared to the normal Hexagasket and we can choose in maximum two free parameters at 390 different IFSs.

The Sierpiński tetrahedron is the three-dimensional analogue of the Sierpiński gasket. The alphabet has one letter more and has length four. The maximal number of free parameters is three and this is achieved by exact one IFS. Therefore the Sierpiński tetrahedron is one of three examples in this thesis where the full amount of free parameters can be chosen.

The last considered fractal is the Sierpiński carpet. We have to adjust the definition of equivalent words as in Example 3.1.10 and we receive a high number of essential equations, in particular 117 equations. Surprisingly, 2 052 IFSs exist with one or more free parameter. However, the majority still has zero free parameters.

This structure occurs on all fractals and we can discover another structure: if there are more equations which have to be fulfilled, it is less common to choose more free parameters.

Nevertheless we can find for every fractal a realization where we can choose at least one free parameter. It is not known, if there is a fractal where all realizations have zero free parameters. We conjecture that the number of essential equations for this fractal should be very large.

Both the algorithms in Listings 5.2 and 5.3 have some nice properties, but at the same time also some disadvantages. One of the disadvantages of both algorithms is the time for the total computation. This comes from the fact that we check all equivalence classes of iterated function systems. For this we recall that the number of all IFSs grows very fast, in particular by the law

$$\#\text{IFSs} = |\text{different mappings for a single copy}|^N. \quad (5.15)$$

The number of equivalence classes can not be calculated in an explicit form, but is approximately

$$\#\text{equivalence classes} \approx |\text{different mappings for a single copy}|^{N-1}, \quad (5.16)$$

which comes from the fact that one single equivalence class contains often exactly the number of possible small mappings.

This explains why our algorithms need more calculating time if the number of equivalence classes increases. Since the amount grows exponentially, this gets worse, if either the alphabet gets bigger or if the number of different mappings for a single copy increases. The fact that we use parallel computing can only compensate this exponential growing rate in a minor way.

Furthermore, it should not be underestimated that the number of equations increases also the computing time, especially at the constructive algorithm.

We can also compare both calculation methods to each other on the same fractal. The constructive algorithm needs a lot more computing time, since we do a lot of algebraic transformations. At the existential algorithm we only have to set up the Jacobian matrix and determine the rank of the matrix, which is done in a relatively short time.

Hence the time to determine the number of free parameters depends in the first place at the mode, in the second place on the number of IFSs and in the third place on the numbers of equations/variables (and of course on the used hardware).

We collected the calculating time of both the algorithms for the previously considered fractals. We performed all computations on the same machine. We used 60 (of 64) kernels,



fractal	number of equivalence classes	constructive algorithm	existential algorithm
Sierpiński gasket	44	2.36 sec	0.774 sec
Vicsek	4 360	146 sec	3.01 sec
SG <sub>3</sub>	7 860	28 min 19 sec	8.37 sec
Pentagasket	10 104	563 sec	6.32 sec
Sierpiński tetrahe- dron	14 022	652 sec	6.48 sec
filled Pentagasket	100 220	2 d 5 h	254 sec
Hexagasket	250 010	7 h 20 min	156 sec
Sierpiński carpet	2 101 776	≈ 941 d 3 h <sup>a)</sup>	14 h 12 min
filled Hexagasket	2 991 900	41 d 34 min	1 h 13 min
SG <sub>4</sub>	10 080 504	≈ 181 d 12 h <sup>a)</sup>	11 h 11 min

<sup>a)</sup> Estimated time – based on a sufficient large number of calculations.

Table 5.3: The calculation time for different fractals in different mode sorted by the number of equivalence classes.

the processors are all Intel Xeon processors of type E7–4830. Thus, the required times are comparable and the detailed calculation times can be found in Table 5.3.

The calculation time differs from fractal to fractal, since we consider a different amount of IFSs. We treat as a first topic the computing time of the constructive algorithm in more detail. The Sierpiński gasket needed only a few seconds to compute all 44 equivalence classes. The Vicsek fractal only took 146 seconds, whereas the 3–level Sierpiński gasket SG<sub>3</sub> took about 1 700 seconds (or approx. 28 min 19 sec) and the Pentagasket took 563 seconds. The Sierpiński tetrahedron needs 652 seconds. Compared to the number of equivalence classes performs the SG<sub>3</sub> thus relatively slow, which comes from the fact that the SG<sub>3</sub> has more essential equations.

The computing time of the filled Pentagasket took extremely long and our computer needed about 191 150 seconds (or approx. 2 d 5 h). The Hexagasket was again calculated relatively fast and took only 26 400 seconds respectively 7 h 20 min. The filled version of the Hexagasket needs more time, in total approximately 41 d 34 min.

The calculation time explodes, if we examine the Sierpiński carpet. The constructive algorithm would need approximately 941 days. We did not run the computer for such a long time, but we run the algorithm for some days and extrapolated the total running time. We included the time in the table to show the limits of the constructive algorithms.

The SG<sub>4</sub> has about five times more equivalence classes and is at the same time five times faster than the Sierpiński carpet. The expected calculation time is “only” 181.5 days.

Again, this calculation time is extrapolated. This means that our constructive algorithm is 25 times faster on the 4-level Sierpiński gasket than on the Sierpiński carpet which comes from the fact that the Sierpiński carpet has more essential equations.

We can observe a similar behavior for the calculation time of the existential algorithm: fractals with less equivalent IFSs need less computing time. These calculation times are in fact much faster than in the constructive way. We discussed the reason for this earlier. The calculation of the Sierpiński gasket with the existential algorithm is about three times faster. This factor becomes larger, if we consider other fractals. For example the computation time of the Sierpiński tetrahedron is about 100 times faster. Again, the Sierpiński carpet plays on another level. The computation time is in this case 1 590 times faster and hence reduces the calculation time from over 2.5 years to 14 hours.

In future we could use both the algorithm to investigate also other fractals. For this we only need to implement the required equations and the definition of the small copies, which can be done relatively quickly. Also the determination of the equivalence classes can be done quickly. The most restricting part is indeed the calculation of the free parameters. The Sierpiński carpet indicates that the number of equations is the main factor for a long calculating time, which will be the limiting factor of further considerations.

All the considered samples show that there are a lot of fractals where we can apply successfully our algorithm and we can find for every fractal a realization with at least one free parameter. This means that we can apply the theory of Chapter 3 to a broad set of fractals and determine their Martin boundary in the inhomogeneous case.

# Chapter 6

## Outlook and further research

In this chapter, we very briefly present some open problems and questions for further research in this field.

### Existence of a geometric criterion for boundary cells

We gave in the introduction in Figure 1.1 some nice pictures of harmonic functions on fractals where we simulated the harmonic function in a numerical way. While we worked on these simulations, the question arose what the boundary (or the “domain” in terms of analysis, i.e. the set on which we impose boundary conditions) of a fractal looks like. In particular we asked ourselves, if there is a geometric criterion which characterizes cells or points as part of the boundary. Ideally, this should apply in general context.

The (analytic) boundary of a particular fractal is well known. For the Sierpiński gasket the boundary consists of the three vertices of the big triangle and for the Sierpiński carpet this is the boundary of the surrounding square.

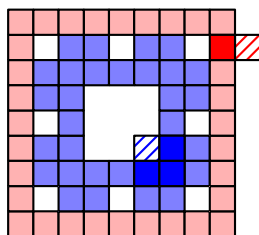


Figure 6.1: The idea behind a geometric description of boundary cells: the strong, red cell misses the hatched, red cell and is therefore a boundary cell. On the other hand the dark blue cells are no boundary cells, since the absence of the hatched, blue cell comes from the structure of the fractal.

We already have an idea, how this can be described in a geometric way. This idea is based on the observation that boundary cells “miss” a neighboring cell where a cell should appear, if we continue the self-similarity to the outer area. This is illustrated in Figure 6.1. In mathematical words this can be described if we consider a word  $w$ . If we map  $w$  with  $S_i$ , we receive  $iw$ . If  $iw$  has now a neighboring cell where previously was no neighbor, we would like to say that  $w$  is a boundary cell. This would be an interesting topic and may give some clearer insight to the fractal world.

## Dimension of intersecting cells

We mentioned in Remark 3.1.9 the dimension of intersecting cells. We considered the Sierpiński carpet, where the intersection of cells has either Hausdorff dimension 0 or 1. This can be described by  $\dim_H(S_v(K) \cap S_w(K))$ . We can redefine the transition probability to this fact. For this, we introduce a function

$$b : \mathcal{W} \times \mathcal{W} \rightarrow [0, \infty)$$

which represents the dimension of the intersection and should be increasing in  $\dim_H$ . Based on this we modify the definition of  $p$  in 3.2.1 in the following way:

$$p(v, w) := \begin{cases} \frac{m(w)}{\sum_{\hat{v} \sim v} m(\hat{v})} b(v, w^-), & \text{if } w = \hat{v}i \text{ with } \hat{v} \sim v \text{ and } i \in \mathcal{A}, \\ 0, & \text{else,} \end{cases}$$

which needs to be renormalized again.

Up to now we have not really any idea, how  $b$  should be defined in exact terms. This may require some detailed research, especially in the direction of dimension. This can also be seen in Figure 6.2, where we give another example. We call this fractal barbell due to its shape which reminds us on this sport equipment. Furthermore this is a slight modification of the Sierpiński carpet. In this case, the dimension of intersections can take values in  $\left\{ \frac{\ln 2}{\ln 3}, 1 \right\}$ . These values arise from the fact that some parts have fractal intersection. To be precise: the intersection is either the classical Cantor set or a straight line. This example can further be modified (in this case with 20 similarities) such that every intersection is of fractal dimension. Of course, all this should be done in the context of weighted fractals.

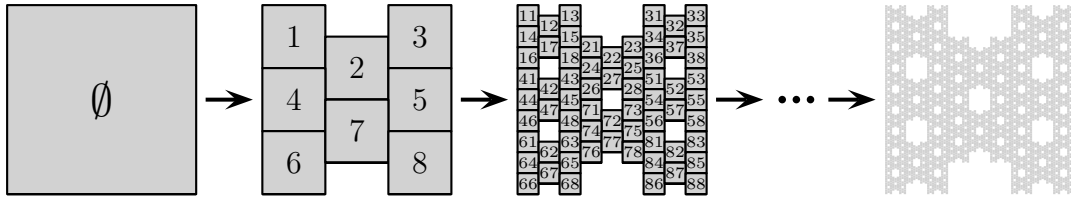


Figure 6.2: The iterated function system and the attractor for the fractal called barbell, where some parts have a fractal dimension of intersection.

## Analysis of $\mathfrak{W}$

We considered in Section 5.1 the essential word space  $\mathfrak{W}$  and studied different essential word spaces with the help of equivalence classes. This reduced the calculation effort a lot and is an important step to reduce the complexity. At the same time we asked ourselves, if those essential word spaces can be compared in another way. One ansatz may be to consider a generalized essential word space in form of

$$\begin{aligned} \mathfrak{W}_{\text{general}} := \{ & 1a_2, 2b_1, 1a_3, 3c_1, 2b_3, 3c_2 \in \mathcal{W} \text{ with} \\ & a_i, b_i, c_i \in \mathcal{A}, a_2 \neq a_3, b_1 \neq b_3, c_1 \neq c_2, \\ & 1a_2 \sim 2b_1, 1a_3 \sim 3c_1, 2b_3 \sim 3c_2; a_2 \neq a_3, b_1 \neq b_3, c_1 \neq c_2 \} \end{aligned}$$

for the Sierpiński gasket. This codes in some sense the whole topology of the fractal (here: the Sierpiński gasket) and it might be worth to take a closer look at this.

At the same time we can invert the question. Can we find a fractal, if we provide a certain essential word space? This could be interesting in topology, since this defines a fractal with a certain, predefined topology.

## Implementation of the algorithm in C++

The provided algorithm in Appendix A is written in Python 3. This has several reasons. In the first place, the algorithm was developed for some basic investigations of (B2). We extended the algorithm, after we recognized the importance of (B2). This led to the current version of the algorithms. The choice for Python as programming language is further based on the skills of the author. These programming skills were most comprehensively developed in Python and we were able to extend this knowledge throughout the implementation of the source code. In addition, the Python package SymPy can already perform many tasks.

A possible, further development of the algorithm could be the migration into C++ or another programming language, which is more compiler affine as Python. This could give

a significant runtime boost, if the implementation is done in the right way. This requires of course a good understanding of the mechanisms inside C++. Since we do not currently have this knowledge, this would require some preliminary work regarding C++. We are uncertain whether this preliminary work is in a suitable relation to the scientific gain. At the same time we would be pleased to work with people who have this required knowledge.

## Possible extension to fractals with weak separation condition

The entire calculus presented in this thesis is only valid under the open set condition (beside some other conditions). This is maybe the most common condition on fractals, but by far not the only one. There are plenty of other conditions on the separation of a fractal, which are often abbreviated with three or four letters. The most common one beside the OSC is the WSC, which stands for weak separation condition and was first introduced by Lau and Ngai in [LN99] especially in context of multifractal measures. The introduced measures in Chapter 2 are also multifractal measures. For a better understanding of the WSC we recall the definition:

**Definition 6.0.1** (cf. [LN99, Def. 6.2]). *The IFS  $\{S_1, \dots, S_N\}$  with contraction ratios  $c_i$  is said to satisfy the weak separation condition (shorten by WSC) if there exists a  $x_0 \in \mathbb{R}^n$  and  $N_0 \in \mathbb{N}$  such that for any  $x = S_w(x_0)$  with  $w \in \mathcal{W}$  every closed ball with radius  $c^k$  contains at most  $N_0$  distinct  $S_v(x)$ ,  $v \in \Lambda_k$ , where  $c := \min_{i=1, \dots, N} \{c_i\}$  and*

$$\Lambda_k := \left\{ w = w_1 \dots w_n \in \mathcal{W} : \prod_{i=1}^n c_i \leq c^k \leq \prod_{i=1}^{n-1} c_i, n \geq 1 \right\}.$$

It is well known that the OSC implies the WSC, but the reverse is wrong. The following example supports this fact and gives at the same time a brief impression of the WSC.

We consider the IFS consisting of

$$S_1 : \mathbb{R} \rightarrow \mathbb{R}, x \mapsto cx$$

and

$$S_2 : \mathbb{R} \rightarrow \mathbb{R}, x \mapsto cx + 1 - c,$$

where  $c = \frac{1}{\varphi} = \frac{2}{1+\sqrt{5}}$  is the inverse of the golden ratio  $\varphi = \frac{1+\sqrt{5}}{2}$  (in fact, it is sufficient if  $c$  is the inverse of a Pisot number). In Figure 6.3 are some iterations of this fractal and the

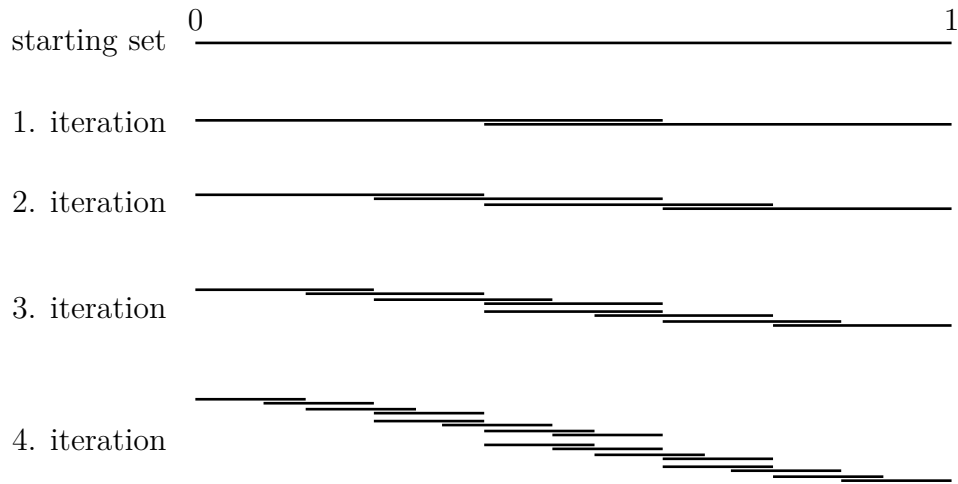


Figure 6.3: Some iterations of an example fulfilling the WSC, but not the OSC. The starting set is the unit interval  $I = [0, 1]$ .

attractor is  $I = [0, 1]$ . This fractal fulfills the WSC and obviously not the OSC.

Lau and Wang extended in a recent paper [LW17] the Martin boundary theory to fractals fulfilling the WSC and in particular to the given example. Because of this, it seems to be possible to replace the OSC in some cases by the WSC (or other separation conditions). This may be researched in more detail and it would be very interesting, if this could be done also in the context of inhomogeneous fractals.





# Appendix A

## Source code

For completeness we include here the total source code of the algorithms developed in Chapter 5. The source code can also be received directly from the author.

The source code should run on every machine, where `Python 3` is installed. In addition, some (common) packages may be required. The code is split into several files:

- `additional.py` (see Listing A.1 on Page 127): Here are all things defined, which are needed in addition like a function to generate a  $\text{\LaTeX}$  table of the results.
- `B2.py` (see Listing A.2 on Page 139): The file `B2.py` contains all functions which are required to calculate the number of free parameters. This includes the simplifications regarding to Method 5.2.5, the calculation by Theorem 5.2.9 and Theorem 5.2.15.
- `config.py` (see Listing A.3 on Page 159): In this file are all global variables defined. Further are two helpful functions defined: one function to create folders, one for the feedback to the user.
- `find_example_with_fp.py` (see Listing A.4 on Page 161): This is an executable file. It loads from `additional.py` a function to find examples to every number of free parameters of a given fractal and prints the results of this to the terminal.
- `fractals.py` (see Listing A.5 on Page 162): This file contains the function `init_fractal(fractal_name)` which is necessary to initialize a certain fractal. This function will generate the word space, the list of all possible mappings and the essential equations.
- `info_main.py` (see Listing A.6 on Page 175): This is an executable file. At the beginning the considered fractal must be initialized and a list of counters should

be defined, to whom the information should be generated. The program runs the function `information_to_fractal` with the considered parameters.

- `main.py` (see Listing A.7 on Page 176): This is an executable file. It runs the main task (in particular the calculation of free parameters) to a fractal, which is initialized in the first step. The main task can either be done with `mode="con"` (for constructive) or `mode="ex"` (for existential), which are based on Theorem 5.2.9 and Theorem 5.2.15.
- `main_complete.py` (see Listing A.8 on Page 177): This is an executable file. It does all needed calculations to one fractal, which is initialized at the beginning. This includes the preprocessing, where the equivalent counter are determined, the main task (finding the number of free parameter) and the postprocessing.
- `postprocessing.py` (see Listing A.9 on Page 178): This is an executable file. It runs the postprocessing for different fractals and generates afterwards a  $\text{\LaTeX}$ -file containing two tables. Table 5.1 and Table 5.2 are generated by this.
- `preprocessing.py` (see Listing A.10 on Page 179): This is an executable file. The fractal needs to be initialized. Then the iterated function systems are put into equivalent IFSs.

```

1  # -*- coding: utf-8 -*-
2
3  """
4  @author:      Stefan Kohl, University of Stuttgart
5  @file name:   additional.py
6  @description: Provides additional functions for the program.
7
8  """
9
10 # =====
11 # =                                     import modules                               =
12 # =====
13 # load own modules:
14 import config
15 import fractals
16 import B2
17
18 # load public modules:
19 from sympy import *
20 import numpy as np
21 import pandas as pd
22 import subprocess
23 import os
24 import time
25 import warnings
26 import multiprocessing as mp
27 import shutil
28 import re
29
30 # =====
31 # =                                     definition of functions                       =
32 # =====
33
34 def transform_sympy_to_latex(entry):
35     """
36     transforms a single line of an equation / matrix / vector to LaTeX code.
37
38     :param entry: the line which should be converted.
39     :return:     LaTeX code representing the line.
40     """
41
42     # convert the entry into latex:
43     result = latex(entry)
44
45     # shift the indices of the variable:
46     result = shift_name_of_latex_variables(result)
47
48     return result
49
50
51 def shift_name_of_latex_variables(entry):
52     """
53     replace the name of the variables shifted by 1, since Python works with 0,...,N-1
54     variables instead of 1,..., N variables (because of Python's way to handle list
55     indices):
56     """
57     result = entry
58
59     # go through all possible letters in a reverse order and replace in a suitable way:
60     for n in reversed(range(0, len(config.x))):
61         result = result.replace("x_{"+ str(n) + "}", "x_{"+ str(n + 1) + "}")
62
63     return result
64
65
66 def eq_to_latex(equations):
67     """
68     Transforms an equation in SymPy into LaTeX code.
69
70     :param equations: list of equations which should be transformed.
71     :return:         string of LaTeX code.
72     """

```

```

72 # start the equations with a new line and an align-environment:
73 result = "\n" + "□" * 8 + "\\begin{align*}\n" + "□" * 12 + "&□"
74
75 # differ between the case, that equations is empty or not.
76 if equations == []:
77     result += "\\emptyset□"
78 else:
79     # transform each single line into LaTeX:
80     result += ("\\\\\\n" + "□"*12 + "&□").join([transform_sympy_to_latex(eq) for eq
81         in equations])
82
83 # add the end of the environment:
84 result += "\n" + "□" * 8 + "\\end{align*}"
85
86 return result
87
88 def var_to_latex(variable):
89     """
90     transform variables into LaTeX code:
91
92     :param variable: list of variables
93     :return: LaTeX code
94     """
95     # put everything into an align* environment:
96     result = "\n" + "□" * 8 + "\\begin{align*}\n" + "□" * 12 + "&"
97
98     # differ between the case, that variable is empty or not.
99     if variable == []:
100         result += "\\emptyset□"
101     else:
102         result += ",□".join([transform_sympy_to_latex(var) for var in variable])
103
104     result += "\n" + "□" * 8 + "\\end{align*}"
105
106     return result
107
108
109 def matrix_to_latex(matrix, centering=True):
110     """
111     transform a matrix into LaTeX code.
112
113     :param matrix: the matrix which should be converted.
114     :param centering: if the matrix should be in an align*-environment.
115     :return: final LaTeX code
116     """
117
118     result = latex(matrix)
119
120     # put everything in an align*-environment, if requested:
121     if centering:
122         result = "\\begin{align*}\n" + result + "\n\\end{align*}\n"
123
124     # shift the indices of the variable:
125     result = shift_name_of_latex_variables(result)
126
127     # put everything into a pmatrix-environment:
128     result = result.replace("\\left[\\begin{matrix}",
129         "\\begin{pmatrix}").replace("\\end{matrix}\\right]", "\\end{pmatrix}")
130
131     return result
132
133 def information_to_counter_with_progress(counter, mode="constructive", progress=-1,
134     print_progress=False):
135     """
136     evaluates all information for a specific counter and prints possibly the overall
137     progress.
138
139     :param counter: the specific counter, which should be considered.
140     :param mode: the mode of the calculation. Either "existential" for the
141         existential way (using the rank of the Jacobian), "constructive" for the
142         constructive way (resolving the equations and apply the Implicit Function

```

```

139     Theorem) or "both" for both methods.
140 :param progress: overall progress
141 :return: TeX-String, which contains all information in TeX-format
142     for the specific counter.
143
144 # print the progress:
145 if print_progress:
146     print_text = "overall-progress:"
147     if config.file_extension != "":
148         print_text = "overall-progress_" + config.file_extension + ":"
149     config.user_feedback(print_text + str(round(progress * 100, 1)).rjust(5) + "%_
150     (Counter=_ + str(counter) + ")._")
151
152 latex_content = "\n\\newpage\n\\LARGE\\bfseries_Counter=_ + str(counter) +
153     "\\\\2ex\\n\\setcounter{Counter}{ + str(counter) + }"
154
155 # determine the equivalent counter and add them to the content:
156 latex_content += "equivalent_counter:\n\\begin{center}" + ",_".join([str(content)
157     for content in B2.equivalent_counter(counter)]) + "\\end{center}\n"
158
159 # get the ifs:
160 ifs = fractals.iso_counter_to_ifs(counter)
161
162 # modify the ifs for printing:
163 ifs_string = []
164 for letter in range(len(ifs)):
165     ifs_string += ["letter_" + str(letter) + ":",_].join([str(subletter) for
166         subletter in ifs[letter]])
167 ifs_string = "\\\\".join(ifs_string)
168 for n in reversed(range(0, len(config.x))):
169     ifs_string = re.sub("(?!\\d)" + str(n) + "(?!\\d)", str(n+1), ifs_string)
170
171 # add the ifs to the body:
172 latex_content += "IFS:\n\\begin{center}\\begin{tabular}{ll}" +
173     ifs_string + "\n\\end{tabular}\\end{center}\n"
174
175 # generate the list of equations which need to be fulfilled:
176 equation_list = config.update_equations(ifs)
177
178 # parse them to LaTeX and append:
179 latex_content += "all_equations:" + eq_to_latex(equation_list) + "\n"
180
181 # delete double equations (this may be the equations the user wants to work with)
182 and add them to the body:
183 shorten_eq = B2.delete_double_equations(equation_list)
184 latex_content += "equations_(duplicates_free):" + eq_to_latex(shorten_eq) +
185     "\\\\"
186
187 # calculate the number of free parameters based on mode:
188 if mode.lower().startswith("e") or mode.lower().startswith("b"):
189     # mode starts with "e" for "existential" or "b" for "both"
190     latex_content += "\\vspace*{1cm}\\textbf{\\Large results from the
191     calculations in mode \"existential\":}\n"
192     fp, sol_type, jacobian_matrix_at_xhom, jacobian_matrix =
193     B2.fp_per_rank_jacobian(counter, skip_general_jacobian_matrix=False)
194     latex_content += "Jacobian_matrix:\n\\begin{align*}J(x) = " +
195     matrix_to_latex(jacobian_matrix, centering=False) + "\\end{align*}\n"
196     latex_content += "Jacobian_matrix_substituted_at_x_{\\text{hom}} =
197     \\left(\\frac{ + str(config.N) + },...,\\frac{ + str(config.N) +
198     "\\right)^T:\n\\begin{align*}J(x_{\\text{hom}}) = " +
199     matrix_to_latex(jacobian_matrix_at_xhom, centering=False) +
200     "\\end{align*}\n"
201     latex_content += "$\\rightarrow \\text{rank}(J(x_{\\text{hom}})) = $ +
202     str(jacobian_matrix_at_xhom.rank()) + "\n"
203     latex_content += "\\begin{center}\\large\\bfseries$\\rightarrow$ free
204     parameters = " + str(fp) + "\\end{center}\n"
205
206 if mode.lower().startswith("c") or mode.lower().startswith("b"):
207     # mode starts with "c" for "constructive" or "b" for "both"
208     latex_content += "\\vspace*{1cm}\\textbf{\\Large results from the
209     calculations in mode \"constructive\":}\n"

```

```

193     fp, sol_type, complete_equation_list, free_equation_list, fixed_equation_list,
        free_var, fixed_var, m_0, set_u1, set_u2, jacobian_matrix,
        jacobian_matrix_without_substitution = B2.fp_per_ift(counter)
194     latex_content += "solution type:" + sol_type + "\\\n"
195     latex_content += "free equation list:" + eq_to_latex(free_equation_list) +
        "\n"
196     latex_content += "fixed equation list:" + eq_to_latex(fixed_equation_list) +
        "\n"
197     latex_content += "complete equation list:" +
        eq_to_latex(complete_equation_list) + "\n"
198     latex_content += "free variables:" + var_to_latex(free_var) + "\n"
199     latex_content += "fixed variables:" + var_to_latex(fixed_var) + "\n"
200     latex_content += "SetU1:" + var_to_latex(set_u1) + "\n"
201     latex_content += "SetU2:" + var_to_latex(set_u2) + "\n"
202     latex_content += "Jacobian matrix:" + "\begin{align*}J(x)=" +
        matrix_to_latex(jacobian_matrix_without_substitution, centering=False) +
        "\end{align*}\n"
203     latex_content += "Jacobian matrix substituted at $x_{\text{hom}}$ =
        \\\left(\frac{1}{" + str(config.N) + "}, \dots, \frac{1}{" + str(config.N) +
        "}\right)^T$:" + "\begin{align*}J(x_{\text{hom}})=" +
        matrix_to_latex(jacobian_matrix, centering=False) + "\end{align*}\n"
204     latex_content += "\\\begin{center}\\\large\\bseries$\\\rightarrow$ free
        parameters = " + str(fp) + "\\\end{center}\n"
205
206     return latex_content
207
208
209 def information_to_fractal(start=0, end=None, percent_increment=0.1,
        list_of_counter=None, mode="constructive", amount_kernel=mp.cpu_count(),
        timestamp=None, latex_mode="silent"):
210     """
211     prints information for a range (or list) of counters to a fractal
212
213     :param start:          start value of IFS
214     :param end:            end number of IFS
215     :param percent_increment: For user feedback: in which step size should a feedback
        appear.
216     :param list_of_counter: alternative input of IFSs to consider (committing start
        and end it is not possible
217                             to skip some IFSs)
218     :param mode            the mode of the calculation. Either "existential" for
        the existential way (using the rank of the Jacobian), "constructive" for the
        constructive way (resolving the equations and apply the Implicit Function
        Theorem) or "both" for both methods.
219     :param amount_kernel  how many kernels should be used
220     :param timestamp:     string, which will be added at the beginning of the
        tex-file.
221                             Standard value: [Year]-[Month]-[Date]_[Hour]-[Minute]
222     :param latex_mode:    if and how the TeX-file should be compiled directly
        afterwards. possible values:
223                             * silent = compiling, but without any output to the user
224                             * normal = compiling and the log of pdflatex is shown
        in console
225                             * None, "no", "none" = no compilation
226     :return:              (nothing)
227     """
228
229     # set the mode to a common label:
230     if mode.lower().startswith("e"):
231         mode = "existential"
232     if mode.lower().startswith("c"):
233         mode = "constructive"
234     if mode.lower().startswith("b"):
235         mode = "both"
236
237     config.user_feedback("Start information to fractal!", print_date=False)
238     config.user_feedback("fractal: " + config.fractal_name, indentation=1,
        print_date=False)
239     config.user_feedback("# kernels: " + str(amount_kernel), indentation=1,
        print_date=False)
240     config.user_feedback("mode: " + mode, indentation=1, print_date=False)
241     config.user_feedback("LaTeX-mode: " + latex_mode, indentation=1, print_date=False)
242     config.user_feedback("It is now: " + time.strftime('%Y-%m-%d_%H:%M:%S'),

```

```

243         indentation=1, print_date=False)
244 # save the starting time for later.
245 start_time = time.time()
246
247 # standard value for time stamp:
248 if timestamp is None:
249     timestamp = time.strftime('%Y-%m-%d_%H-%M')
250
251 # generate/open the file. First, create the folder:
252 config.create_folder("tex")
253
254 # set the file name:
255 latex_file_name = "tex/" + timestamp + "__Info_for_" + config.fractal_name +
256     ".tex"
257
258 # open the file:
259 latex_file = open(latex_file_name, "w+")
260
261 # extract version number:
262 version = extract_version()
263
264 # initialize the string of the LaTeX file body (the stuff that stand between
265     \begin{document} .. \end{document}) as empty.
266 body_string = ""
267
268 # differ between input per list and input per range:
269 if list_of_counter is None:
270     # cancel side-effect: if the variable "end" is defined at the beginning to the
271     # value len(config.single_wordspace) ** config.N, then this would be zero,
272     # since at the import of this module this was zero. Thus:
273     if end is None:
274         end = len(config.single_wordspace) ** config.N
275
276     config.user_feedback("Data:░░░░░░░░range░(start░=░" + str(start) + " ,░end░=░" +
277         str(end) + " )\n", indentation=1, print_date=False)
278     list_of_counter = range(start, end)
279 else:
280     config.user_feedback("Data:░░░░░░░░list░(length░=░" + str(len(list_of_counter))
281         + " )\n", indentation=1, print_date=False)
282
283 # write the header of the LaTeX file:
284 latex_file.write("\documentclass[10pt, a4paper]{article}\n\usepackage{amssymb,
285     amsmath, amsfonts, fancyhdr}\n\usepackage[a4paper, left=25mm, right=25mm,
286     top=2.5cm,
287     bottom=2cm]{geometry}\n\setlength{\parindent}{0cm}\n\pagestyle{fancy}
288     \fancyhf{}\n\fancyhead[R]{generated:░" + time.strftime('%d.%m.%Y, %H:%M:%S')
289     + " }\n\fancyhead[C]{fractal:░" + config.fractal_name +
290     " }\n\fancyhead[L]{Info}\n\fancyfoot[L]{version:░" + str(version) +
291     " }\n\fancyfoot[C]{ }\n\fancyfoot[R]{page:░
292     \thepage}\n\renewcommand{\footrulewidth}{0.4pt}\n\newcounter{Counter}
293     \n\begin{document}\n\n")
294 latex_file.write(" This file contains all information about some single counters to
295     the fractal\n\begin{center}\n\huge\bfseries\n\texttt{" +
296     config.fractal_name + "\n}\end{center}\n")
297
298 # save the starting time for later:
299 start_string = time.strftime('%Y-%m-%d_%H:%M:%S')
300
301 # flush the content of the LaTeX file.
302 latex_file.flush()
303
304 # List for printing:
305 printing_list = np.round(np.arange(0, 1, percent_increment)*len(list_of_counter))
306
307 # Init multiprocessing.Pool()
308 pool = mp.Pool(amount_kernel)
309
310 # go through all elements of the list:
311 result_objects = [pool.apply_async(information_to_counter_with_progress,
312     args=(counter, mode, counter /
313         len(list_of_counter), counter in
314         printing_list))

```

```

296         for counter in list_of_counter]
297 results = [r.get() for r in result_objects]
298
299 # Don't forget to close:
300 pool.close()
301
302 # wait, until all processes are finished:
303 pool.join()
304
305 # write at the beginning of the file some text about the needed time etc.:
306 latex_file.write("\section*{Remark:}\n\\begin{center}\n\\begin{tabular}{l}\n
        \nStarted at:&' + start_string + "\\\\n\\end{tabular}\n\\end{center}\n\\newpage\n\\fancyhead[L]{Counter=
        \nStarted at:&' + start_string + "\\\\n\\end{tabular}\n\\end{center}\n\\newpage\n\\fancyhead[L]{Counter=
        time.strftime('%Y-%m-%d_%H:%M:%S') +
        "\n\\end{tabular}\n\\end{center}\n\\newpage\n\\fancyhead[L]{Counter=
        \\\theCounter}\n")
307
308 # lower a little bit the font size and write the results of each counter:
309 latex_file.write("\small\n\n" + "\n".join(results) + "\n\\end{document}")
310
311 # close the file.
312 latex_file.close()
313
314 # if the user wants a compilation of the LaTeX-file, do it now:
315 compile_latex_file(latex_file_name=latex_file_name[4:], path="tex",
        latex_mode=latex_mode, clean_up=True)
316
317 # save the time where the calculation is finished:
318 end_time = time.time()
319
320 config.user_feedback("\ninformation to fractal finished!", print_date=False)
321 config.user_feedback("fractal: " + str(config.fractal_name), indentation=1,
        print_date=False)
322 config.user_feedback("# kernels " + str(amount_kernel), indentation=1,
        print_date=False)
323 config.user_feedback("duration: %.3f sec" % (end_time - start_time),
        indentation=1, print_date=False)
324 config.user_feedback("# problems: " + str(len(list_of_counter)), indentation=1,
        print_date=False)
325 config.user_feedback("mode: " + mode, indentation=1, print_date=False)
326 config.user_feedback("LaTeX-mode: " + latex_mode, indentation=1, print_date=False)
327 config.user_feedback("It is: " + time.strftime('%Y-%m-%d_%H:%M:%S'),
        indentation=1, print_date=False)
328
329
330 def db_row_to_tex(row, line_break=""):
331     """
332     transform a row of a data frame into tex code.
333
334     :param row:         one particular line of the data frame;
335     :param line_break: additional space of the line break;
336     :return:           string with tex code.
337     """
338
339     # initialize the result as empty:
340     result = ""
341
342     # go through all columns of this row:
343     for column in range(len(row)):
344         # get the content of this cell of the data frame:
345         content_of_db = row[column]
346
347         if type(content_of_db) == str:
348             # if the content is a string:
349             result += "&" + content_of_db + "\n"
350         else:
351             if np.isnan(content_of_db):
352                 # replace NaNs by "--"
353                 result += "&--\n"
354             else:
355                 # otherwise its a number. Put it into "\numprint{.}" for a nice
356                 # representation:
357                 result += "&\numprint{" + str(int(content_of_db)) + "\n"

```



```

358     # add a newline to the result and a possible extension of the line break:
359     result += "\\n" + line_break + "\n"
360
361     # return all:
362     return result
363
364
365 def collect_results_to_df(max_fp=5):
366     """
367     adds all results of the different fractals into one single DataFrame, which will
368     be returned.
369
370     :param max_fp: maximal number of free parameters in the statistic.
371     :return df: a DataFrame, containing all information
372     """
373
374     config.user_feedback("Collect results into DataFrame.")
375
376     # initialize result as empty:
377     df = pd.DataFrame()
378
379     # go through all fractals:
380     for fractal in extract_considered_fractals():
381
382         # report to user, which fractal is actually considered.
383         config.user_feedback("* Consider fractal " + fractal + "\", indentation=1)
384
385         # load the fractal:
386         fractals.init_fractal(fractal)
387
388         # load the files, if they exist:
389         if os.path.isfile("sol/" + config.fractal_name + ".sol"):
390             db = pd.read_csv("sol/" + config.fractal_name + ".sol")
391
392             # convert column "fp":
393             db["fp"] = db["fp"].astype('float64')
394
395             # The number of small copies equals the length of the alphabet:
396             df.at["smallcopies", config.fractal_name] = config.N
397
398             # the number of mappings equals the length of one single word space:
399             df.at["mappings", config.fractal_name] = len(config.single_wordspace)
400
401             # set the number of equations:
402             df.at["equations", config.fractal_name] =
403                 len(config.update_equations(fractals.iso_counter_to_ifs(0)))
404
405             # set the number of equivalence classes:
406             df.at["numbereq", config.fractal_name] = db.shape[0]
407
408             # set the number of counters with unknown number of free parameters as
409             # zero. This value will be updated later.
410             df.at["unknownfp", config.fractal_name] = 0
411
412             # extract all values, where the number of free parameters is valid:
413             valid_db = db[~np.isnan(db.fp)]
414
415             # count, how many valid solutions exist:
416             number_of_solutions = 0
417
418             # consider now for all values in 0,..,max_fp, how many values of free
419             # parameters exist:
420             for n in range(0, max_fp + 1):
421                 # if it would be more free variables then the length of the alphabet,
422                 # set it as NaN. This will be displayed in the tex statistic as "-".
423                 if n >= config.N:
424                     df.at[str(n) + "fp", config.fractal_name] = np.NaN
425                 else:
426                     # otherwise sum over the length of the equivalence classes with n
427                     # free parameters:
428                     df.at[str(n) + "fp", config.fractal_name] =
429                         sum(valid_db.length[valid_db.fp == n])

```

```

424         # update number_of_solutions:
425         number_of_solutions += sum(valid_db.length[valid_db.fp == n])
426
427     # raise a warning, if max_fp is too small:
428     if sum(valid_db.length[valid_db.fp > max_fp]) > 0:
429         warnings.warn("some free parameters where not considered, since max_fp
430             is too small!")
431
432     # set the amount of total IFSs:
433     df.at["totalifs", config.fractal_name] =
434         len(config.single_wordspace)**config.N
435
436     # calculate the number of fractals with unknown number of free parameters
437     (hopefully zero):
438     df.at["unknownfp", config.fractal_name] = df.loc["totalifs",
439         config.fractal_name] - number_of_solutions
440
441     config.user_feedback("Results collected!")
442     return df
443
444 def generate_tables(generate_standalone=True, latex_mode="silent", max_fp=5,
445     skip_unknown_fp=True):
446     """
447     generates two tables with the results to each fractal.
448
449     :param generate_standalone: boolean, if in addition a standalone-file should be
450         created.
451     :param latex_mode: if and how the standalone-file should be compiled:
452         * silent = compiling, but without any output to the
453             user
454         * normal = compiling and the log of pdflatex is shown
455             in console
456         * None, "no", "none" = no compilation
457     :param max_fp: maximal number of free parameters in the table.
458     :param skip_unknown_fp: if the line with "unknown number of free parameters"
459         should be skipped.
460     :return: result, the data of both table as as DataFrame
461     """
462
463     # generate the statistic as DataFrame:
464     df = collect_results_to_df(max_fp=max_fp)
465
466     # Set the file names of the two tables:
467     table1_name = "table-info"
468     table2_name = "table-fp"
469
470     # Set up a dictionary with the full names of the columns:
471     dict_column_names = {"smallcopies": "N", "mappings": "number_of_different_mappings",
472         "equations": "number_of_essential_equations", "numbereq": "number_of_equivalence
473         classes", "unknownfp": "unknown_number_of_free_parameters", "totalifs": "total
474         number_of_IFSs"}
475
476     # Set the correct name of fractals by a dictionary:
477     dict_row_names = {"HexagasketWITHOUTHOLE": "filled_Hexagasket",
478         "PentagasketWITHOUTHOLE": "filled_Pentagasket", "SG2": "Sierpi\\'nski_gasket",
479         "ST": "Sierpi\\'nski_tetrahedron", "SC": "Sierpi\\'nski_carpet",
480         "SG3": "SG_3", "SG4": "SG_4"}
481
482     # add remaining names into dict_row_names:
483     for row in df.columns:
484         if row not in dict_row_names:
485             dict_row_names[row] = row
486
487     # set the columns for each table:
488     columns_table1 = ["smallcopies", "mappings", "equations", "numbereq", "totalifs"]
489     columns_table2 = [str(n) + "fp" for n in range(max_fp+1)] + ["totalifs"]
490     if not skip_unknown_fp:
491         columns_table2 = columns_table2[:-1] + ["unknownfp"] + [columns_table2[-1]]
492
493     #####
494     # FIRST TABLE:
495     # generate the file with the content of the first table, containing infos to the
496     fractals:

```

```

481 latex_file = open(table1_name + ".tex", "w+")
482 latex_file.write("\begin{tabular}{@{m}{3.6cm}R{1cm} +
      "R{2cm}*(len(columns_table1)-1) + "@{}}\n%\n\toprule\nfractal")
483
484 # write the header of the table:
485 for column in columns_table1:
486     latex_file.write("_&_" + dict_column_names[column])
487 latex_file.write("\n\midrule\n")
488
489 # write the different single lines of the table:
490 for row in df.columns:
491     latex_file.write(dict_row_names[row] +
      db_row_to_tex(df.loc[columns_table1,row]))
492
493 # ... and we are done. Close the table and the file:
494 latex_file.write("\bottomrule\n\end{tabular}")
495 latex_file.close()
496
497 #####
498 # SECOND TABLE:
499 # generate the file with the content of the second table, containing infos to the
      number of free parameters:
500 latex_file = open(table2_name + ".tex", "w+")
501 latex_file.write("\begin{tabular}{@{m}{3.6cm} + "r"*len(columns_table2) +
      "@{}}\n%\n\toprule\n")
502
503 # write the header of the table:
504 latex_file.write("fractal_&\multicolumn{" + str(max_fp+1) + "}c{number_of_free_
      parameters}_&_" + "_&_".join(["\multirow{2}{2.5cm}{ " +
      dict_column_names[entry] + "} for entry in columns_table2[max_fp+1:]] +
      "\n\n")
505
506 # write the second line of the heading:
507 for number_fp in range(max_fp + 1):
508     latex_file.write("_&_" + str(number_fp))
509 latex_file.write("_&_"*(len(columns_table2) - max_fp - 1) + "\n\midrule\n")
510
511 # write the different single lines of the table:
512 for row in df.columns:
513     latex_file.write(dict_row_names[row] +
      db_row_to_tex(df.loc[columns_table2,row]))
514
515 # ... and we are done. Close the table and the file:
516 latex_file.write("\bottomrule\n\end{tabular}")
517 latex_file.close()
518
519 # report to user, that we are done:
520 config.user_feedback("Two tables successfully generated!")
521
522 # if requested, generate the standalone file:
523 if generate_standalone:
524     # set the name and open the file:
525     latex_file_name = "standalone_tables.tex"
526     latex_file = open(latex_file_name, "w+")
527
528     # write the header and content of the file:
529     latex_file.write("\documentclass[a4paper]{article}\usepackage{numprint,
      tabularx, multirow, booktabs, geometry, datetime}
      \n\setlength{\parindent}{0cm}
      \n\newcolumntype{R}[1]{>\raggedleft\arraybackslashp{#1}}
      \n\pagestyle{empty}\n\begin{document}\nThese tables were generated on
      " + time.strftime("%d.%B.%Y at %H:%M:%S") + "~and compiled on \today~at
      \hhmmsstime!\n\n[3ex]\n\ninput{" + table1_name +
      "}\n\n\n\vspace*{1cm}\n\ninput{" + table2_name + "}\n\n\end{document}")
530
531     # close the file:
532     latex_file.close()
533
534     # if the user wants a compilation of the LaTeX-file, do it now:
535     compile_latex_file(latex_file_name, path="", latex_mode=latex_mode,
      clean_up=True)
536
537 return df

```

```

538
539
540 def extract_version():
541     """
542     give the version back to the user
543
544     :return: version
545     """
546     return "1.0.74"
547
548
549 def extract_considered_fractals():
550     """
551     determines, which fractals have been calculated and returns them back as a list.
552
553     :return: list of fractals, which have been considered.
554     """
555     result = []
556
557     # get the content of the folder "sol":
558     for file_or_folder_name in os.listdir(os.path.join(os.path.abspath("."), "sol")):
559         # check, if it is a folder. If true, append it to results:
560         if os.path.isdir(os.path.join(os.path.abspath("."), "sol",
561                                     file_or_folder_name)):
562             result.append(file_or_folder_name)
563
564     # sort the list by the alphabet:
565     result.sort()
566
567     return result
568
569 def find_example_to_all_fp(fractal):
570     """
571     determines examples to a certain fractal with all appearances of number of free
572     parameters.
573
574     :param fractal: which fractal should be considered
575     :return: (nothing)
576     """
577
578     # load the fractal
579     fractals.init_fractal(fractal)
580
581     config.user_feedback("Considering fractal \u" + config.fractal_name + "\!",
582                         print_date=False)
583
584     # load the data frame, if existent:
585     if os.path.isfile("sol/" + config.fractal_name + ".sol"):
586         db = pd.read_csv("sol/" + config.fractal_name + ".sol")
587
588         # convert column "fp":
589         db["fp"] = db["fp"].astype('float64')
590
591         # extract representatives, where no free parameter has been calculated:
592         missing_fp = db[pd.isnull(db.fp)]
593
594         # raise a Warning, if there are some representatives where fp is not
595         # calculated:
596         if not missing_fp.empty:
597             config.user_feedback("WARNING! There are \u" + str(missing_fp.shape[0]) + "\u
598                                 (from \u" + str(db.shape[0]) + "\u) representatives where the number of
599                                 free parameters have not been calculated! This may cause a falsified
600                                 result!", print_date=False)
601
602         # go through all possible number of free parameters:
603         for fp in range(config.N):
604             # restrict to the data frame, where free parameters equals the given
605             # number and reset the index.
606             part_db = db[db.fp == fp].reset_index()
607
608             # print the results, if there are some:
609             if not part_db.empty:

```

```

603         user_text = ""
604         # differ between singular and plural.
605         if part_db.shape[0] == 1:
606             user_text += "*There is" + str(part_db.shape[0]) + "
                representative with"
607
608         else:
609             user_text += "*There are" + str(part_db.shape[0]) + "
                representatives with"
610
611         if fp == 1:
612             user_text += str(fp) + " free parameter."
613         else:
614             user_text += str(fp) + " free parameters."
615
616         config.user_feedback(user_text + "One representative has the counter
                \" + str(part_db.at[0, "representative"]) + "\".", indentation=1,
                line_length=110, print_date=False)
617
618     else:
619         config.user_feedback("The requested fractal\" + str(fractal) + "\" does not
                exist and must be calculated first!")
620
621 def compile_latex_file(latex_file_name, path="", latex_mode="silent", clean_up=True):
622     """
623     compile a LaTeX file.
624
625     :param latex_file_name: the name of the file
626     :param path: the path to the file
627     :param latex_mode: different LaTeX-mode:
628         * silent = compiling, but without any output to the user
629         * normal = compiling and the log of pdflatex is shown in
630           console
631         * None, "no", "none" = no compilation
632     :param clean_up: if the additional files of pdflatex should be cleaned up.
633     :return: (nothing)
634     """
635     # check, if the path is correct:
636     if path == "":
637         path = "."
638
639     if path[-1] == "/":
640         path = path[:-1]
641
642     # check, if the file should be compiled:
643     if (latex_mode is not None) and (latex_mode.lower() != "no") and
644         (latex_mode.lower() != "none"):
645
646         # check, if pdflatex is installed:
647         if shutil.which("pdflatex") is not None:
648             # pdflatex is installed.
649             config.user_feedback("start with compiling of a LaTeX-file.",
650                                 print_date=True)
651             config.user_feedback("*file name:" + latex_file_name, indentation=1,
652                                 print_date=False)
653
654             command = "pdflatex" + latex_file_name
655
656             # differ between silent / normal mode:
657             if latex_mode.lower().startswith("norm"):
658                 config.user_feedback("*Normal mode log:", indentation=1,
659                                     print_date=False)
660                 subprocess.call(command, shell=True, cwd=path)
661             else:
662                 config.user_feedback("*Silence mode (log is suppressed). Please
663                                     wait ...", indentation=1, print_date=False)
664                 subprocess.call(command, shell=True, cwd=path,
665                                 stderr=subprocess.DEVNULL, stdout=subprocess.DEVNULL)
666
667         # delete the additional files, if requested:
668         if clean_up:
669             # go through different files:
670             for file_extension in ["aux", "log"]:
671                 # check if file exist.

```

```

664         if os.path.exists(path + "/" + latex_file_name[: -3] +
665             file_extension):
666             # delete the file, if it exists.
667             os.remove(path + "/" + latex_file_name[: -3] + file_extension)
668             config.user_feedback("LaTeX file successfully compiled!")
669     else:
670         # raise a warning that pdflatex is not installed!
671         config.user_feedback("WARNING: pdflatex is not installed, the file " +
672             path + "/" + latex_file_name + " cannot be compiled!", indentation=0,
673             print_date=False, line_length=100)

```

Listing A.1: The module `additional.py` containing all additional things needed for the main program and/or pre- and postprocessing.

```

1  # -*- coding: utf-8 -*-
2
3  """
4  @author:      Stefan Kohl, University of Stuttgart
5  @file name:   B2.py
6  @description: contains all essential functions for determining the free parameters of
7                a fractal.
8
9  """
10 # =====
11 # =                               import modules                               =
12 # =====
13 # load own modules:
14 import config
15 import fractals
16 import additional
17
18 # load public modules:
19 from sympy import diff, Matrix, det, expand, add, exp, log, cancel, simplify, root,
20     solve, FiniteSet, Rational
21 import numpy as np
22 import pandas as pd
23 import itertools
24 import time
25 import multiprocessing as mp
26 import os
27 import socket
28
29 # =====
30 # =                               definition of functions                               =
31 # =====
32
33
34 def collect_result(result):
35     """
36     function for writing result into the handler from config.file
37
38     :param result: string, which should be written.
39     :return:      (nothing)
40     """
41     config.file.write(result)
42
43
44 def write_runtime_statistic(length_list=None, file_name=None, amount_kernel=None,
45     mode="constructive", start_time=None, end_time=None):
46     """
47     write infos for the actual calculation into a runtime statistic
48
49     :param length_list: how many problems have been considered
50     :param file_name:  in which file the solutions have been written
51     :param amount_kernel: how many kernels have been used
52     :param mode:       calculation mode, either "constructive" or "existential"
53     :param start_time: when the calculation started
54     :param end_time:   when the calculation ended
55     :return:           (nothing)
56     """
57
58     # fix the file name:
59     statistic_file_name = "runtime_statistic.csv"
60     statistic_file_header = ""
61
62     # if the file does not exist, we have to write the header too.
63     if not os.path.isfile(statistic_file_name):
64         statistic_file_header = "fractal,Amount_of_problems,file ,computer," +
65             "#kernels , version ,mode,start ,end ,duration [sec]"
66
67     statistic_file = open(statistic_file_name, "a+")
68
69     # differ between the start of the calculation and the end of the calculation.
70     if end_time is None:
71         # we are at the start of the calculation:

```

```

70         statistic_file.write(statistic_file_header + "\n" + str(config.fractal_name) +
71                               ", " + str(length_list) + ', ' + file_name + ', ' + str(socket.gethostname())
72                               + ", " + str(amount_kernel) + ", " + str(additional.extract_version()) + ", "
73                               + mode + ", " + time.strftime('%d.%m.%Y_%H:%M:%S',
74                               time.localtime(start_time)) + ",")
75
76     else:
77         # we are at the end of the calculation:
78         statistic_file.write(time.strftime('%d.%m.%Y_%H:%M:%S',
79         time.localtime(end_time)) + ",%.3f" % (end_time - start_time))
80
81     # close the file (and we are done):
82     statistic_file.close()
83
84 def ifs_mp(use_file=None, amount_kernel=None, percent_increment=0.01,
85           list_of_counters=None, timestamp=None, mode="constructive"):
86     """
87     consider multiple IFSs, splits the task up to (multiple) kernels.
88     The result of the computation is written into a file.
89
90     :param use_file:          file-path which contains a list which should be
91                             considered.
92     :param amount_kernel:    how many kernels should be used
93     :param percent_increment: For user feedback: in which step size should a feedback
94                             appear.
95     :param list_of_counters: alternative input of IFSs to consider
96     :param timestamp:        string, which will be added at the beginning of the
97                             csv-file.
98
99                             Standard value: [Year]-[Month]-[Date]--[Hour]-[Minute]
100
101     :param mode:             the mode of the calculation. Either "existential" for
102                             the existential way (using the rank of the Jacobian) or anything else for the
103                             constructive way (resolving the equations and apply the Implicit Function
104                             Theorem)
105
106     :return:                 (nothing)
107     """
108
109     # if no time stamp was given, set it now (this has to be defined here, since
110     # otherwise it would be fix to the time the program was compiled).
111     if timestamp is None:
112         timestamp = time.strftime('%Y-%m-%d_%H-%M-%S')
113
114     # save starting time:
115     start_time = time.time()
116
117     # fix amount of kernels:
118     if amount_kernel is None:
119         # use all kernels:
120         amount_kernel = mp.cpu_count()
121
122         # except the case the machine has 64.
123         # In this case we only take 60 and other users can use the remaining 4
124         # kernels...
125         if mp.cpu_count() == 64:
126             amount_kernel = 60
127
128     # don't use to much kernels, especially if the user sets the amount of kernels.
129     amount_kernel = min(amount_kernel, mp.cpu_count())
130
131     # set the mode to a common label:
132     if mode.lower().startswith("ex"):
133         mode = "existential"
134     else:
135         mode = "constructive"
136
137     # fix the file name:
138     file_name = config.fractal_file_folder + config.fractal_name + "/" + timestamp +
139                "_-" + config.fractal_name + config.file_extension + ".sol"
140
141     # print some infos for the user:
142     config.user_feedback("Start with calculating free parameters!", print_date=False)
143     config.user_feedback("fractal: " + str(config.fractal_name), indentation=1,
144                          print_date=False)
145     if config.file_extension != "":

```



```

127         config.user_feedback("file_extension: " + str(config.file_extension),
128                               indentation=1, print_date=False)
129
130     config.user_feedback("file_name: " + str(file_name), indentation=1,
131                          print_date=False)
132     config.user_feedback("#kernels: " + str(amount_kernel), indentation=1,
133                          print_date=False)
134     config.user_feedback("mode: " + mode, indentation=1, print_date=False)
135     config.user_feedback("starting_time: " + time.strftime('%Y-%m-%d.%H:%M:%S'),
136                          indentation=1, print_date=False)
137
138     # generate the folders and files, where the results are written:
139     config.create_folder(config.fractal_file_folder)
140     config.create_folder(config.fractal_file_folder + config.fractal_name + "/")
141     config.file = open(file_name, "w+")
142     config.file.write("begin, counter, fp, sol_type, end\n")
143     config.file.flush()
144
145     # generate index-file / index-entry for postprocessing:
146     index_file = open(config.fractal_file_folder + config.fractal_name +
147                      "/index.index", "a+")
148     index_file.write(file_name + "\n")
149     index_file.close()
150
151     # get the counters to consider. Depending on the fact if list_of_counters was
152     # given, differ between two cases.
153     if list_of_counters is None:
154         # no list was given.
155         if use_file is None:
156             # no specific file was given. Use the file which is referenced in the
157             # index file in fandf:
158             with open("fandf/" + config.fractal_name + "_missing.index") as f:
159                 temp_file_name = f.read().splitlines()
160
161             if len(temp_file_name) == 0:
162                 raise ValueError("file with reference to counters (" +
163                                  "fandf/" +
164                                  config.fractal_name + "_missing.index") is empty!")
165
166             # we have potentially multiple entries in the "..._missing.index"-file, so
167             # we content of these files together:
168             list_of_counters = []
169             for use_file in temp_file_name:
170                 list_of_counters += csv_to_list_of_counter(use_file)
171
172             # empty the content of the "..._missing.index"-file, such that the content
173             # of the imported files won't be read / calculated multiple times.
174             temp_file = open("fandf/" + config.fractal_name + "_missing.index", "w+")
175             temp_file.close()
176
177             config.user_feedback("data: " + list_of_counters + "\n",
178                                 indentation=1, print_date=False)
179         else:
180             config.user_feedback("data: " + list_of_counters + "\n",
181                                 indentation=1, print_date=False)
182
183     # write the values of the runtime statistics:
184     write_runtime_statistic(length_list=len(list_of_counters), file_name=file_name,
185                             amount_kernel=amount_kernel, mode=mode, start_time=start_time)
186
187     # list of counters for printing:
188     printing_list = np.round(np.arange(0, 1, percent_increment) *
189                              len(list_of_counters))
190
191     # initialize the pools for multiprocessing:
192     pool = mp.Pool(amount_kernel)
193
194     # do the calculus asynchronous:
195     for counter in range(0, len(list_of_counters)):
196         # handle the problem to fp_per_ift_with_progress over using the arguments of
197         # args and the return of "fp_per_ift_with_progress" should be passed to the
198         # callback-function "collect_result"
199         pool.apply_async(fp_to_counter, args=(list_of_counters[counter], counter /
200                                             len(list_of_counters), counter in printing_list, mode),

```

```

        callback=collect_result)
183
184 # close the pools:
185 pool.close()
186
187 # wait until all pools are finished:
188 pool.join()
189
190 # save the time where the calculation is finished:
191 end_time = time.time()
192
193 # close the file handler:
194 config.file.close()
195
196 # write the second part of the runtime statistic:
197 write_runtime_statistic(start_time=start_time, end_time=end_time)
198
199 # Report to user the end of the calculation:
200 config.user_feedback("\nCalculating to free parameters has finished!",
201                     indentation=1, print_date=False)
202 config.user_feedback("fractal: {}" + str(config.fractal_name), indentation=1,
203                     print_date=False)
204 config.user_feedback("# kernels {}" + str(amount_kernel), indentation=1,
205                     print_date=False)
206 config.user_feedback("duration: {}%3f sec" % (end_time - start_time),
207                     indentation=1, print_date=False)
208 config.user_feedback("# problems: {}" + str(len(list_of_counters)), indentation=1,
209                     print_date=False)
210 config.user_feedback("mode: {}" + mode, indentation=1, print_date=False)
211 config.user_feedback("end time: {}" + time.strftime('%Y-%m-%d %H:%M:%S'),
212                     indentation=1, print_date=False)
213
214 #
215 # all functions for the simplification of the equations:
216 # =====
217 #
218
219 def delete_factorial_variables(equation_list):
220     """
221     Apply Method 5.2.3 to the equation list:
222     factor outs variables, for example:
223         c1 * x[i]**a - c2 * x[i]**b
224     gets
225         c1 - c2 * x[i]**(b-a)
226
227     :param equation_list: list of equations
228     :return: list of simplified equations
229     """
230
231     # initialize the resulting equation list as empty:
232     result = []
233
234     # go through all equation in equation list
235     for equation in range(len(equation_list)):
236         actual_equation = expand(equation_list[equation])
237
238         # go through all variables:
239         for variable in range(config.N):
240
241             # check, if x[variable] is contained in every term of equation:
242             xn_contained_everywhere = True
243
244             # differ between a summation and every other thing (at the moment not
245             # implemented) and check, if the actual variable is contained in every
246             # part of the term.
247             if actual_equation.func == add.Add:
248
249                 # go through all monomials which generates the whole terms:
250                 for sub in range(len(actual_equation.args)):
251
252                     # test, if the variable is contained in this sub-term:
253                     if config.x[variable] not in

```

```

247         actual_equation.args[sub].free_symbols:
248         xn_contained_everywhere = False
249     else:
250         # in this case the term is no summation, but for example of the form
251         #  $x[1]*x[2]**b$ . This means that a variable has to be zero and a
252         # simplification is not possible.
253         xn_contained_everywhere = False
254
255     if xn_contained_everywhere:
256         # in this case  $x[variable]$  is contained everywhere and we can shorten.
257
258         # factor_aq can be an arbitrary monomial in  $x[variable]$ . It gets
259         # automatically replaced by the first monomial in  $x[variable]$ 
260         factor_aq = config.x[variable]
261
262         # go through all sub-terms and determine the minimal power of
263         #  $x[variable]$ 
264         for n in range(len(actual_equation.args)):
265
266             # shorten all other variables by substitute them by 1
267             actual_equation_temp =
268                 actual_equation.args[n].subs(list(zip(config.x[:variable] +
269                 config.x[(variable + 1):], [1] * (config.N - 1))))
270
271             # shorten the scalar of the actual term. With this,
272             # "actual_equation_temp" is of the shape  $x[variable]**power$ 
273             actual_equation_temp = actual_equation_temp /
274                 (actual_equation_temp.subs(config.x[variable], 1))
275
276             # determine the monomial, which can be later shorten. If the
277             # actual monomial is smaller or  $n==0$  holds, overwrite factor_aq.
278             if (log(actual_equation_temp.subs(config.x[variable], exp(1))) <
279                 log(factor_aq.subs(config.x[variable], exp(1)))) or (n == 0):
280
281                 # choose new minimal power of  $x[variable]$ :
282                 factor_aq = actual_equation_temp
283
284             # divide the equation by the factor:
285             actual_equation = actual_equation / factor_aq
286
287             # shorten:
288             actual_equation = cancel(actual_equation)
289
290             # and expand afterwards for reuse:
291             actual_equation = expand(actual_equation)
292
293             # if the new equation is not zero, append it (should be always the case).
294             if simplify(actual_equation) != 0:
295                 result += [actual_equation]
296
297     # return the new equation list.
298     return result
299
300 def delete_double_equations(equation_list):
301     """
302     Apply Method 5.2.2 to the equation list:
303     removes equations, which already occur.
304
305     :param equation_list: list of equations
306     :return: list of unique equations
307     """
308
309     # initialize the result as empty:
310     result = []
311
312     # go through all equations:
313     for n in range(len(equation_list)):
314
315         # remember, if an equation occurs multiple times:
316         equation_found = False
317
318         # go through all equations from the actual equation till the end:

```

```

309     for m in range(n + 1, len(equation_list)):
310
311         # check, if the equation "n" and "m" are identical.
312         if (simplify(equation_list[n] - equation_list[m]) == 0) or
313             (simplify(equation_list[m] - equation_list[n]) == 0):
314             # in this case we found a multiple equation and we keep this in mind.
315             equation_found = True
316
317         # if no identical equation has been found and thus "equation_found" equals
318         # False and further the equation is non-zero, we append this equation to
319         # result.
320         if not equation_found and equation_list[n] != 0:
321             result += [equation_list[n]]
322
323     # after all, return the results:
324     return result
325
326 def replace_polynomial_terms(equation_list):
327     """
328     Apply Method 5.2.4 to the equation list:
329     replace in equation_list single equations of the shape
330     a*x[i]**c - d
331     by
332     x[i] - root(d / a, c)
333
334     Replaces also terms of the shape
335     a*x[i]**c + b*x[i]**c - d
336     by
337     x[i] - root(d/(a + b), c)
338
339     :param equation_list: list of equations
340     :return: list of equations, where polynomial terms have been
341             replaced.
342     """
343
344     # initialize the result as empty:
345     result = []
346
347     # go through all equations:
348     for eq in range(len(equation_list)):
349         actual_equation = expand(equation_list[eq])
350
351         # go through all variables:
352         for xn in config.x:
353
354             # save in "remaining_term" all stuff, that does not contain xn:
355             remaining_term = 0
356
357             # save in "term_with_xn" all stuff, that does contain xn:
358             term_with_xn = 0
359
360             if actual_equation.func == add.Add:
361                 # the actual equation is a summation of sub-terms.
362
363                 # go through all sub-terms "sub":
364                 for sub in range(len(actual_equation.args)):
365
366                     # check, if xn is part of the actual sub-term "sub":
367                     if xn in actual_equation.args[sub].free_symbols:
368                         # if true, add sub-term "sub" to "term_with_xn":
369                         term_with_xn += actual_equation.args[sub]
370                     else:
371                         # otherwise add sub-term "sub" to remaining_term:
372                         remaining_term += actual_equation.args[sub]
373
374             else:
375                 # the "actual_equation" is not of the type summation.
376
377                 # differ between the case that xn is contained in "actual_equation" or
378                 # not:
379                 if xn in actual_equation.free_symbols:
380                     # add the equation to "term_with_xn", while "remaining_term" stays
381                     # zero.

```

```

376         term_with_xn = actual_equation
377     else:
378         # otherwise set "remaining_term" as "actual_equation", while
           # "term_with_xn" stays zero:
379         remaining_term = actual_equation
380
381     # in the case that "term_with_xn" is not zero there could be a
           # substitution. Further has to be "remaining_term" non-zero, otherwise
           # could "xn" be factored out (this can be done by the function
           # "delete_factorial_variables")
382     if term_with_xn != 0 and remaining_term != 0:
383
384         # check, if the power of "xn" is constant. This is essential for the
           # case, where "term_with_xn" consists of multiple summands:
385         # first of all, the power is constant and zero.
386         power_is_constant = True
387         power = 0
388
389         # differ between the case, that "term_with_xn" consists of multiple
           # summands or not. This is necessary, since we cannot apply
           # "term_with_xn_args[...]" in the case, that "term_with_xn" is only
           # one summand.
390     if term_with_xn.func == add.Add:
391         # in this case "term_with_xn" consists of multiple summands, for
           # example a*xn**b + c*xn**b
392
393         # go through all sub-terms "sub" of "term_with_xn":
394         for sub in range(len(term_with_xn.args)):
395             actual_term = term_with_xn.args[sub]
396
397             # shorten all factors of actual_term:
398             actual_term = actual_term / (actual_term.subs(xn, 1))
399
400             # actual_term is now of the shape xn**actual_power. Determine
           # actual_power:
401             actual_power = log(actual_term.subs(xn, exp(1)))
402
403             # if we consider the first sub-term "sub", the power gets
           # fixed:
404             if sub == 0:
405                 power = actual_power
406
407             # if the actual power differs from all other powers, we can
           # later not factor out xn**power:
408             if power != actual_power:
409                 # so, we save that the power is not constant at all terms:
410                 power_is_constant = False
411     else:
412         # the term is only a monomial and of the shape "a*xn**b
413
414         # shorten all factors of actual_term:
415         actual_term = term_with_xn / (term_with_xn.subs(xn, 1))
416
417         # actual_term is now of the shape xn**power. Determine power:
418         power = log(actual_term.subs(xn, exp(1)))
419
420     # if the power is constant and equals not one, we can do something
           # (otherwise we are done with this equation).
421     if power_is_constant and power != 1:
422         # we can apply the simplification. In total we have until now that
           # the equation is of the form:
423         #      term_with_xn - remaining_term
424         # where "term_with_xn" can be represented as
425         #      xn**power * term_shorten_by_xn
426         # and we determine "term_shorten_by_xn" now:
427         term_shorten_by_xn = simplify(term_with_xn / (xn ** power))
428
429         # we then divide the "remaining_term" by "term_shorten_by_xn":
430         remaining_term = simplify(remaining_term / term_shorten_by_xn)
431
432         # so now our equation is of the form
433         #      xn**power - remaining_term
434         # which we want to replace by

```

```

435         #         xn = root(-remaining_term, power)
436         # This is our new "actual_equation":
437         actual_equation = xn - root(-remaining_term, power)
438
439         # we add "actual_equation" to the results, regardless we were able to modify
440         # it or not:
441         result += [actual_equation]
442
443     # finally, return the results:
444     return result
445
446 def fp_per_ift(counter):
447     """
448     Apply the Implicit function theorem from Theorem 5.2.9 (resp. Corollary 5.2.11) to a
449     certain IFS, which is determined by "counter".
450
451     :param counter: the counter, which stands for a particular IFS.
452     :returns: the following 12 variables:
453         * fp: the number of determined free parameters.
454         * sol_type: a string, representing the solution type.
455         * complete_equation_list: The list of all equations
456         * free_equation_list: a list of free equations
457         * fixed_equation_list: a list of equations which fix a certain
458           variable
459         * free_var: a list of variables which are free
460         * fixed_var: a list of fixed variables
461         * m_0: the number of variables which are missed for a quadratic
462           shape of the Jacobi Matrix (hopefully one which corresponds
463           to a quadratic form)
464         * set_u1: The set U1, representing the space of free variables
465         * set_u2: The set U1, representing the space of fixed variables
466         * jacobian_matrix: the Jacobi Matrix at the homogeneous point
467         * jacobian_matrix_without_substitution: the (general) Jacobi
468           Matrix
469
470     """
471
472     # initialize the free parameter "fp" with -1 and "sol_type" with "NaN" for the
473     # case that the calculation fails:
474     fp = -1
475     sol_type = "NaN"
476
477     # call simplify_equations_complete, which sets up the equations for this
478     # particular IFS, simplifies them and returns a list of free equations, fixed
479     # equations, free variables and fixed variables (see also Definition 5.2.7)
480     free_equation_list, fixed_equation_list, free_var, fixed_var =
481     simplify_equations_complete(counter)
482
483     # the list of all equations:
484     complete_equation_list = free_equation_list + fixed_equation_list
485
486     # we apply Corollary 5.2.11 already here.
487     # Please note, that we have always m_0 >= 1 from the equation m(1) + ... + m(N) = 1
488     m_0 = len(complete_equation_list) - len(fixed_var)
489
490     # for completeness, define set_u1, set_u2, jacobian_matrix and
491     # jacobian_matrix_without_substitution:
492     set_u1 = free_var
493     set_u2 = fixed_var
494     jacobian_matrix = []
495     jacobian_matrix_without_substitution = []
496
497     if m_0 < 1:
498         # there are more fixed variables as fixed equations (which is a contradiction)
499         # (There is no known case of this)
500         sol_type = "MoreVariablesThenEquations(length_free_eq=" +
501         str(len(free_equation_list)) + ",length_fix_eq=" +
502         str(len(fixed_equation_list)) + ",length_free_var=" + str(len(free_var)) +
503         ",length_fixed_var=" + str(len(fixed_var)) + ")"
504     else:
505         # we determine the sets U1 and U2. For this, we consider all possible
506         # transfers until we found a valid solution.

```

```

493
494 # remember, if we found a solution:
495 sol_found = False
496
497 # count the amount of cases, where we did not find a solution.
498 try_counter = 0
499
500 # we consider all possible subsets of cardinality m_0 from free_var:
501 for subset in list(itertools.permutations(free_var, m_0)):
502
503     # until now we do not have any solution and we have to check, if we have a
504     # solution with this subset:
505     if not sol_found:
506
507         # we raise the counter of tries:
508         try_counter += 1
509
510         # we set up the variables for the set U2, which is the union of fixed
511         # variables and the variables contained in the subset:
512         set_u2 = fixed_var + list(subset)
513
514         # the set U1 consists of all free variables which are not part of
515         # "subset":
516         set_u1 = [x_i for x_i in free_var if x_i not in subset]
517
518         # initialize the content of the Jacobian Matrix as empty. Additionally
519         # we keep also the matrix without a substitution at the homogeneous
520         # point, which will be used for information to the user:
521         matrix_content = []
522         matrix_content_without_substitution = []
523
524         # we go through all components of the function F:
525         for f_i in complete_equation_list:
526
527             # we initialize the row as empty:
528             row = []
529             row_without_substitution = []
530
531             # we differentiate f_i after all fixed variables in ascending
532             # order:
533             for x_i in [x_temp for x_temp in config.x if x_temp in set_u2]:
534
535                 # we only differentiate
536                 row_without_substitution += [diff(f_i, x_i)]
537
538                 # we differentiate and substitute the homogeneous case:
539                 row += [diff(f_i, x_i).subs(list(zip(config.x,
540                 tuple(FiniteSet(Rational(1, config.N)) ** config.N)[0])))]
541
542             # we add the rows to the content:
543             matrix_content_without_substitution += [row_without_substitution]
544             matrix_content += [row]
545
546         # with the previously calculated content we set up the Jacobian matrix:
547         jacobian_matrix = Matrix(matrix_content)
548         jacobian_matrix_without_substitution =
549             Matrix(matrix_content_without_substitution)
550
551         # now we can consider the determinant:
552         if det(jacobian_matrix) != 0:
553             # we found a feasible solution, since the determinant is non-zero.
554             # The number of free parameters equals the length of set_u1:
555             fp = len(set_u1)
556             sol_type = "ift"
557             sol_found = True
558         else:
559             # we have no solution, since the determinant is zero. In this case
560             # we have to choose another subset and hope for the best...
561             sol_type = "determinantZero(length_free_eq=" +
562                 str(len(free_equation_list)) + ";length_fix_eq=" +
563                 str(len(fixed_equation_list)) + ";length_free_var=" +
564                 str(len(free_var)) + ";length_fixed_var=" +
565                 str(len(fixed_var)) + ")"

```

```

552
553     # if we needed more than one solution we save this at the solution type:
554     if try_counter > 1:
555         sol_type += "-try=" + str(try_counter)
556
557     # if we had to transfer more than one variable we also save this at the
558     # solution type:
559     if m_0 != 1:
560         sol_type += "-miss_var=" + str(m_0)
561
562     return fp, sol_type, complete_equation_list, free_equation_list,
563           fixed_equation_list, free_var, fixed_var, m_0, set_u1, set_u2,
564           jacobian_matrix, jacobian_matrix_without_substitution
565
566 def simplify_equations_complete(counter):
567     """
568     simplify equations in that way, that the first equation (sum of all variables) is
569     skipped for the replacements by
570     simplify_equations.
571     After the simplifications this equation is added.
572
573     :param counter: which counter should be considered.
574     :return: 4 variables, in detail:
575         * free_equation_list: list of free equations (ideally empty)
576         * fixed_equation_list: list of fixed equations
577         * free_var: list of free variables
578         * fixed_var: list of fixed variables
579     """
580
581     # load the IFS to this counter:
582     ifs = fractals.iso_counter_to_ifs(counter)
583
584     # load the equations related to this IFS:
585     equation_list = config.update_equations(ifs)
586
587     # simplify the equations except the summation equation ( $m(1) + \dots + m(N) = 1$ ).
588     # NOTE: We skip the summation equation, since this leads to some problems and to a
589     # massive increase in calculation time (with no real benefit). So we will add
590     # the summation equation at a later point and keep in mind, that we have one
591     # fixed variable more. See also Remark 5.2.12
592     free_equation_list, fixed_equation_list, free_var, fixed_var =
593     simplify_equations(free_equation_list=equation_list[1:], free_var=config.x)
594     sum_equation = [equation_list[0]]
595
596     # The summation equation  $m(1) + \dots + m(N) = 1$  determine the last free variable.
597     # Thus we definitely need at least one free variable.
598     if len(free_var) == 0:
599         raise ValueError("simplify_equations_complete: no free variables!")
600
601     # add the summation equation to the list of fixed equations:
602     fixed_equation_list += sum_equation
603
604     # return all:
605     return free_equation_list, fixed_equation_list, free_var, fixed_var
606
607 def simplify_equations(free_equation_list, free_var, fixed_equation_list=None,
608                       fixed_var=None):
609     """
610     We apply Method 5.2.6 and simplify equations by delete factorial variable, replacing
611     polynomial terms, deleting double equations and, if possible, substitute
612     variables.
613
614     :param free_equation_list: list of equations where things can be replaced.
615     :param free_var: list of free variables.
616     :param fixed_equation_list: list of equations determining solutions for fixed
617     variables.
618     :param fixed_var: list of fixed variables.
619     :return: 4 variables, in detail:
620         * free_equation_list: updated list of free equations
621         * fixed_equation_list: updated list of fixed
622         equations

```



```

611         * free_var:          updated list of free variables
612         * fixed_var:       updated list of fixed
                               variables
613     """
614     # set standard value for the variables "fixed_equation_list" and "fixed_var":
615     if fixed_equation_list is None:
616         fixed_equation_list = []
617     if fixed_var is None:
618         fixed_var = []
619
620     # first, we delete all factorial variables:
621     free_equation_list = delete_factorial_variables(free_equation_list)
622
623     # in the next step we replace polynomial terms:
624     free_equation_list = replace_polynomial_terms(free_equation_list)
625
626     # now we delete all multiple equations:
627     free_equation_list = delete_double_equations(free_equation_list)
628
629     # finally, we check if there is any equation which contains an isolated variable
        which can be substituted into the other equations. If so, we can add this
        variable to the list of fixed variables.
630     # We remember with the boolean "simple_equation_found", if we already found an
        equation with an isolated variable:
631     simple_equation_found = False
632
633     # with "variable_simple_equation" we remember the variable, which can be replaced:
634     variable_simple_equation = []
635
636     # and with "simple_equation" we note the equation:
637     simple_equation = []
638
639     # we go through all free variables. We do this in a reversed order such that for
        example the variable x[3] gets replaced before the variable x[1] (this is only
        for aesthetical reasons)
640     for xm in reversed(free_var):
641
642         # check, if we found already a variable which can be replaced.
643         if not simple_equation_found:
644             # otherwise we have to check, if "xm" can be replaced.
645
646             # for this, we go through all free equations:
647             for actual_equation in free_equation_list:
648
649                 # we check, if we can solve the actual equation in an unique way with
                    respect to "xm":
650                 if len(solve(actual_equation, xm)) == 1:
651
652                     # if so, we found an equation which determines the variable "xm":
653                     simple_equation = actual_equation
654                     simple_equation_found = True
655                     variable_simple_equation = xm
656
657     # check, if we found an equation which contains an isolated variable.
658     if simple_equation_found:
659
660         # we initialize the updated free equation list as empty:
661         new_free_equation_list = []
662
663         # we determine the term for the substitution. It holds then, that "xm" equals
            "substitution"
664         substitution = solve(simple_equation, variable_simple_equation)[0]
665
666         # we go through all equations and substitute "xm" by "substitution"
667         for equation in free_equation_list:
668             new_free_equation_list += [equation.subs(variable_simple_equation,
                substitution)]
669
670         # we set up the setting for the recursion.
671         # For this, we have to extend the fixed equation by the definition of "xm":
672         new_fixed_equation_list = fixed_equation_list + [substitution -
            variable_simple_equation]
673

```

```

674         # the "variable_simple_equation" has to be canceled from free variables:
675         new_free_var = [xn for xn in free_var if xn != variable_simple_equation]
676
677         # but added to the fixed variables:
678         new_fixed_var = fixed_var + [variable_simple_equation]
679
680         # and finally, the recursion:
681         free_equation_list, fixed_equation_list, free_var, fixed_var =
            simplify_equations(new_free_equation_list, new_free_var,
                               new_fixed_equation_list, new_fixed_var)
682
683     # return the values:
684     return free_equation_list, fixed_equation_list, free_var, fixed_var
685
686
687 def csv_to_list_of_counter(file, only_unique=True):
688     """
689     Read a csv file and return a list of counter. The list contains only unique
690     counter, if "only_unique" is true.
691
692     :param file: file path, which should be read.
693     :param only_unique: specification, if only unique counter should be considered.
694     :return: list with counter.
695     """
696     # check, if the file exists:
697     if os.path.isfile(file):
698
699         # read the file
700         db = pd.read_csv(file, names=["counter"])
701
702         # convert it to a list:
703         list_of_counter = list(db.counter)
704
705         # check, if only unique counters should be considered:
706         if only_unique:
707             list_of_counter = np.unique(list_of_counter)
708         else:
709             # the file cannot be found. Raise a Error:
710             raise ValueError("csv_to_list_of_counter: file \"+ file + "\" doesn't
711                             exist!")
712
713     # return the list:
714     return list(list_of_counter)
715
716
717 def apply_mapping_to_ifs(ifs, mapping_index):
718     """
719     We apply a mapping to a specific IFS and rename it afterwards such that the first
720     list corresponds again to the letter "0", the second list to the letter "1"
721     and so on.
722
723     :param ifs: the IFS, which should be mapped
724     :param mapping_index: index of the mapping
725     :return: mapped and renamed IFS.
726     """
727
728     # get the specific mapping:
729     single_mapping = config.mapping[mapping_index]
730
731     # get the way we have to rename:
732     rename = config.single_wordspace[mapping_index]
733
734     # initialize the mapped IFS as empty:
735     ifs_mapped = []
736
737     # go through all small mappings of the particular IFS:
738     for first_letter in single_mapping:
739         # first letter before renaming
740
741         # initialize the second cell / letter as empty.
742         single_cell = []
743
744         # As the second letter we have to consider all letters of the alphabet:

```

```

741     for second_letter in range(config.N):
742
743         # map this letter and rename it afterwards.
744         single_cell += [rename[ifs[first_letter]][single_mapping[second_letter]]]
745
746         # now we can add this renamed mapping to the total IFS:
747         ifs_mapped += [list(single_cell)]
748
749     # return everything:
750     return ifs_mapped
751
752
753 def test_iso():
754     """
755     This function helps to test, if the function "iso_counter_to_ifs" and
756     "iso_ifs_to_counter" are inverse maps to each other.
757
758     :return: (nothing), but prints text if there is a wrong counter.
759     """
760
761     for n in range(config.N):
762         ifs = fractals.iso_counter_to_ifs(n)
763         counter = fractals.iso_ifs_to_counter(ifs)
764         if n != counter:
765             config.user_feedback("Isomorphism fails at counter" + str(n) + "!" )
766
767 def equivalent_counter(counter):
768     """
769     This function calculates to a specific counter all equivalent counter.
770
771     :param counter: counter, which should be considered.
772     :return: a sorted and unique list of equivalent counter.
773     """
774
775     # initialize the result as empty:
776     result = []
777
778     # get the IFS to the counter:
779     ifs = fractals.iso_counter_to_ifs(counter)
780
781     # go through all possible mappings:
782     for single_mapping in range(len(config.mapping)):
783
784         # map the IFS by "single_mapping"
785         ifs_mapped = apply_mapping_to_ifs(ifs, single_mapping)
786
787         # determine the counter of the mapped IFS:
788         mapped_counter = fractals.iso_ifs_to_counter(ifs_mapped)
789
790         # add the counter to the result:
791         result += [mapped_counter]
792
793     # sort the results:
794     result.sort()
795
796     # delete duplicates in result:
797     result = list(np.unique(result))
798
799     # return the sorted and unique list:
800     return result
801
802
803 def equivalent_counter_with_progress(counter, progress=-1, print_progress=False):
804     """
805     determines the equivalent counter and prints possibly the progress for the user.
806
807     :param counter: the counter, to whom all equivalent counter should be
808                     determined.
809     :param progress: actual progress
810     :param print_progress: if the progress should be printed
811     :return: list of sorted and unique counter equivalent to the given
812             counter.

```

```

811     """
812
813     # if true, print the progress.
814     if print_progress:
815         config.user_feedback("overall-progress" + str(round(progress*100, 1)).rjust(5)
816                               + "%_(Counter=_" + str(counter) + ")._")
817
818     # determine the equivalent counter:
819     result = equivalent_counter(counter)
820
821     # return everything:
822     return result
823
824 def equiv_counter_for_file(counter, progress=-1, print_progress=False):
825     """
826     determines, if the counter is the smallest counter in its equivalence class.
827     If this is true, a string for the ".sol" file is given back and will be written.
828
829     :param counter:          the counter, to whom all equivalent counter should be
830                             determined.
831     :param progress:        actual progress
832     :param print_progress:  if the progress should be printed
833     :return:                either an empty string or a string with all the
834                             information of the equivalence class.
835     """
836
837     # determine the equivalent counter:
838     eq_class = equivalent_counter_with_progress(counter, progress, print_progress)
839
840     # initialize the result as empty.
841     result = ""
842
843     if eq_class[0] == counter:
844         # the considered counter is the smallest counter in this equivalence class
845         result = str(counter) + "," + str(len(eq_class)) + ",\n" + str(eq_class) +
846                "\",,\n"
847
848     return result
849
850 def extract_missing_counter(df=None, fandfsize=1500000, missing_in_index=True):
851     """
852     extracts from the ".sol"-file all representatives where no free parameters are
853     calculated.
854     Those missing counters are saved under the directory "fandf" into files containing
855     at most "fandfsize" many counters.
856
857     :param df:                the data frame, where the counters should be extracted
858                             from.
859     :param fandfsize:        the number of counters in each missing file
860     :param missing_in_index: if the missing files should be added to the index file.
861     :return:                (nothing)
862     """
863
864     # set a timestamp, such that all files have the same timestamp but a different
865     numbering:
866     timestamp = time.strftime('%Y-%m-%d_%H-%M')
867
868     if df is None:
869         # load the data frame:
870         df = pd.read_csv("sol/" + config.fractal_name + ".sol")
871
872     # determine missing entries:
873     missing_db_entries = df[pd.isnull(df.fp)]
874
875     # save missing entries, if not empty:
876     if not missing_db_entries.empty:
877         config.user_feedback("_export_missing_counter_(length:_)" +
878                               str(missing_db_entries.shape[0]) + ")")
879         config.create_folder("fandf/" + config.fractal_name + "/")
880
881     # save the missing into files, which contain at most "fandfsize" many entries.

```

```

        Number them with an additional counter "split_part_number". Initialize
        this counter:
875     split_part_number = 1
876
877     # go through all splits:
878     for split in range(0, missing_db_entries.shape[0], fandfsize):
879         # fix the file name:
880         missing_counter_file = "fandf/" + config.fractal_name + "/" + timestamp +
            "-part%02i_-_missing.list" % split_part_number
881
882         # raise "split_part_number" for the next file:
883         split_part_number += 1
884
885         # export the missing counter without index or header:
886         missing_db_entries.iloc[split:(split + fandfsize),
            :].to_csv(missing_counter_file, index=False, header=False,
            columns=["representative"])
887
888         # register the file in the index file:
889         if missing_in_index:
890             # open the index file of missing counters:
891             index_file = open("fandf/" + config.fractal_name + "_missing.index",
                "a+")
892             # write the file name and path of the file, containing the missing
                counter:
893             index_file.write(missing_counter_file + "\n")
894             # close the file.
895             index_file.close()
896             config.user_feedback(str(split_part_number - 1) + " file(s) with missing
                counters were created!", indentation=1)
897
898
899 def preprocessing_mp_direct(amount_kernel=mp.cpu_count(), percent_increment=0.1,
    list_of_counter=None, fandfsize=1500000, file_extension=""):
900     """
901     calculates to a list of counters (or otherwise all counters) the equivalence
        classes and extracts the smallest counter as representative for this
        equivalence class. The information are then written into the dataframe
        containing the information to the fractal.
902     """
903
904     # save the starting time for later.
905     start_time = time.time()
906
907     config.user_feedback("Start of direct Preprocessing!")
908     config.user_feedback("fractal: " + str(config.fractal_name), indentation=1,
        print_date=False)
909     config.user_feedback("#kernels: " + str(amount_kernel), indentation=1,
        print_date=False)
910     config.user_feedback("It is: " + time.strftime('%Y-%m-%d_%H:%M:%S'),
        indentation=1, print_date=False)
911
912     if list_of_counter is None:
913         list_of_counter = range(len(config.single_wordspace)**config.N)
914
915     # generate the folders and files, where the results are written:
916     config.create_folder("sol/")
917     file_name = "sol/" + config.fractal_name + file_extension + ".sol"
918     config.file = open(file_name, "wt")
919     config.file.write("representative, length, eq_class, fp, sol_type, source\n")
920     config.file.flush()
921
922     # List for printing:
923     printing_list = np.round(np.arange(0, 1, percent_increment)*len(list_of_counter))
924
925     # initialize the pools for multiprocessing:
926     pool = mp.Pool(amount_kernel)
927
928     for counter in range(0, len(list_of_counter)):
929         # do the calculus asynchronous:
930         pool.apply_async(equiv_counter_for_file, args=(list_of_counter[counter],
            counter / len(list_of_counter), counter in printing_list),
            callback=collect_result)

```

```

931
932 # close the pools:
933 pool.close()
934
935 # wait until all pools are finished:
936 pool.join()
937
938 # save the time where the calculation is finished:
939 end_time = time.time()
940
941 # close the file handler:
942 config.file.close()
943
944 # extract the "missing" counter. In this case, this are all representatives.
945 config.user_feedback("Start with extracting missing counters!")
946 extract_missing_counter(fandfsize=fandfsize)
947
948 # Report to user the end of the calculation:
949 config.user_feedback("\nDirect preprocessing has finished!", print_date=False)
950 config.user_feedback("fractal: {}" + str(config.fractal_name), indentation=1,
951                      print_date=False)
952 config.user_feedback("# kernels {}" + str(amount_kernel), indentation=1,
953                      print_date=False)
954 config.user_feedback("duration: {}%.3f sec" % (end_time - start_time),
955                      indentation=1, print_date=False)
956 config.user_feedback("It is: {}" + time.strftime('%Y-%m-%d %H:%M:%S'),
957                      indentation=1, print_date=False)
958
959
960 def postprocessing(fractal=None, empty_index=True, file_extension_export="",
961                  missing_in_index=True, fandfsize=1500000):
962     """
963     do the postprocessing for a specific fractal.
964
965     :param fractal:          which fractal should be considered;
966     :param empty_index:     if the index file with the partial solutions should
967                             by cleaned;
968     :param file_extension_export: file extension for the .sol-file (for example
969                             "temp");
970     :param missing_in_index: if the missing counters should be added to the
971                             index file;
972     :param fandfsize:       number of counters in each file in fandf;
973     :return:                (nothing)
974     """
975
976     # initialize the fractal:
977     if fractal is not None:
978         fractals.init_fractal(fractal)
979
980     # print some infos for the user:
981     config.user_feedback("Start with postprocessing for fractal " +
982                         config.fractal_name + ".")
983
984     # if the data frame with the solution does not exist, we have to stop. Otherwise
985     # we can proceed.
986     if not os.path.isfile("sol/" + config.fractal_name + ".sol"):
987         config.user_feedback("Missing data base! No postprocessing to fractal " +
988                             config.fractal_name + " performed!!")
989         return
990
991     # load the data frame:
992     db = pd.read_csv("sol/" + config.fractal_name + ".sol")
993
994     # report to the user, that loading was successful.
995     config.user_feedback("{} data frame was successfully loaded!")
996
997     # convert column "fp", such that entries of type "NaN" are displayed correctly.
998     db["fp"] = db["fp"].astype('float64')
999
1000    # read the names of the single solution files, which should be added to the data
1001    # frame:
1002    index_file_name = "sol/" + config.fractal_name + "/index.index"
1003    with open(index_file_name) as f:

```

```

992     partial_db_files = f.read().splitlines()
993
994 # empty index file, if requested:
995 if empty_index:
996     index_file = open(index_file_name, "w+")
997     index_file.close()
998
999 # initialize data frame with new entries:
1000 new_db_entries = pd.DataFrame()
1001
1002 # read the single files and update the data frame:
1003 for file_name in partial_db_files:
1004     config.user_feedback("-open file" + file_name + "\", indentation=1)
1005
1006     # read the .sol-file:
1007     partial_new_db_entries = pd.read_csv(file_name)
1008
1009     # extract entries, where the line does not start with "BEGIN" and does not end
1010     # with "END". Those lines are corrupt!
1011     temp_partial_new_db_entries =
1012         partial_new_db_entries[(partial_new_db_entries.begin != "BEGIN") |
1013                                (partial_new_db_entries.end != "END")]
1014
1015     # the corrupt lines should be empty. Otherwise we raise an error:
1016     if not temp_partial_new_db_entries.empty:
1017         incomplete_rows = str((temp_partial_new_db_entries.index + 2).tolist())
1018         raise ValueError("An error occurred while reading the file. One (or more)
1019             lines are corrupt. Affected row(s):\n" + str(incomplete_rows))
1020
1021     # save the source of the solution (may be relevant for tracking)
1022     partial_new_db_entries["source"] = file_name
1023
1024     # delete columns "begin" and "end", since they are now obsolete:
1025     partial_new_db_entries = partial_new_db_entries.drop(["begin", "end"], axis=1)
1026
1027     # the partial_new_db_entries are fine and can be merged with the
1028     # new_db_entries:
1029     new_db_entries = pd.concat([new_db_entries, partial_new_db_entries],
1030                               sort=False)
1031
1032 if partial_db_files:
1033     # there are some new solution files which can be included.
1034
1035     # generate a data frame with from calculated entries (should be empty):
1036     wrong_calculated_entries = new_db_entries[new_db_entries.fp < 0]
1037
1038     # all other entries are correct:
1039     new_db_entries = new_db_entries[new_db_entries.fp >= 0]
1040
1041     # delete entries, which agree in "counter" and "fp":
1042     new_db_entries = new_db_entries[~new_db_entries.duplicated(["counter", "fp"],
1043                       keep="first")]
1044
1045     # extract all entries, which contradict each other (should be empty):
1046     contradicting_db_entries =
1047         new_db_entries[new_db_entries.duplicated(["counter"], keep=False)]
1048
1049     # keep all other non-contradicting entries:
1050     new_db_entries = new_db_entries[~new_db_entries.duplicated(["counter"],
1051                       keep=False)]
1052
1053     # merge the entries of new_db_entries into db :
1054     full_db = db.merge(new_db_entries, left_on="representative",
1055                       right_on="counter", how="outer")
1056
1057     # add rows from full_db with representative == NaN to contradicting_db_entries.
1058     # For this rows holds the following: they have in new_db_entries a counter,
1059     # but in full_db no associated representative.
1060     contradicting_db_entries = pd.concat([contradicting_db_entries,
1061                                         full_db[pd.isnull(full_db.representative)]], sort=False)
1062
1063     # delete from full_db all entries with representative == NaN:
1064     full_db = full_db[~pd.isnull(full_db.representative)]

```

```

1053
1054     # determine entries, where the already calculated fp (not NaN) contradicts the
1055     # new calculated fp (also not NaN):
1056     # These are also contradicting entries:
1057     contradicting_db_entries = pd.concat([contradicting_db_entries,
1058     full_db[~pd.isnull(full_db.fp_x) & (full_db.fp_x != full_db.fp_y) &
1059     ~pd.isnull(full_db.fp_y)]], sort=False)
1060
1061     # Transfer the values, if fp_x == NaN:
1062     full_db.at[pd.isnull(full_db.fp_x), "sol_type_x"] =
1063     full_db.loc[pd.isnull(full_db.fp_x), "sol_type_y"]
1064     full_db.at[pd.isnull(full_db.fp_x), "source_x"] =
1065     full_db.loc[pd.isnull(full_db.fp_x), "source_y"]
1066     full_db.at[pd.isnull(full_db.fp_x), "fp_x"] =
1067     full_db.loc[pd.isnull(full_db.fp_x), "fp_y"]
1068
1069     # delete all unnecessary columns (they were introduced at line 1045):
1070     full_db.drop(["fp_y", "sol_type_y", "source_y", "counter"], axis=1,
1071     inplace=True)
1072
1073     # rename the columns:
1074     full_db.rename(columns={"fp_x": "fp", "sol_type_x": "sol_type", "source_x":
1075     "source"}, inplace=True)
1076
1077     # feedback for user what will happen:
1078     config.user_feedback("Save total solution file.")
1079
1080     # save full_db:
1081     full_db.to_csv("sol/" + config.fractal_name + file_extension_export + ".sol",
1082     index=False)
1083
1084     # report the success of saving:
1085     config.user_feedback("Successful saved!", indention=1)
1086
1087     # generate uniform time stamp:
1088     timestamp = time.strftime('%Y-%m-%d_%H-%M')
1089
1090     # save all contradicting counter:
1091     if not contradicting_db_entries.empty:
1092         config.user_feedback("Export contradicting counter (length: " +
1093         str(contradicting_db_entries.shape[0]) + ")")
1094         config.create_folder("contradicting/")
1095         contradicting_db_entries.to_csv("contradicting/" + timestamp + "_-" +
1096         config.fractal_name + ".csv", index=False)
1097
1098     # save now all wrong calculated counter:
1099     if not wrong_calculated_entries.empty:
1100         config.user_feedback("Export wrong calculated counter (length: " +
1101         str(wrong_calculated_entries.shape[0]) + ")")
1102         config.create_folder("fandf/" + config.fractal_name + "/")
1103         wrong_calculated_entries.to_csv("fandf/" + config.fractal_name + "/" +
1104         timestamp + "_-wrong.csv", index=False)
1105
1106     # save missing entries:
1107     extract_missing_counter(df=full_db, fandfsize=fandfsize,
1108     missing_in_index=missing_in_index)
1109
1110     config.user_feedback("Postprocessing finished!")
1111
1112 def fp_per_rank_jacobian(counter, skip_general_jacobian_matrix=True):
1113     """
1114     calculates the number of free parameters to a counter by determining the rank of
1115     the Jacobian matrix.
1116
1117     :param counter: which counter should be considered
1118     :param skip_general_jacobian_matrix: if the general Jacobian matrix should be
1119     generated (for example for the information)
1120     :return: * fp: number of free parameters
1121     * sol_type: the type of solution
1122     * jacobian_matrix_at_x_hom: the Jacobian
1123     matrix evaluated at x_hom
1124     * jacobian_matrix: the Jacobian matrix
1125

```



```

1109     """
1110
1111     # get the IFS:
1112     ifs = fractals.iso_counter_to_ifs(counter)
1113
1114     # get the equations:
1115     equation_list = config.update_equations(ifs)
1116
1117     # set up the Jacobian matrix:
1118     jacobian_matrix_at_xhom_content = []
1119     jacobian_matrix_content = []
1120     for f_i in equation_list:
1121         row_at_xhom = []
1122         row = []
1123         for x_i in config.x:
1124
1125             # we differentiate and substitute the homogeneous case:
1126             row_at_xhom += [diff(f_i, x_i).subs(list(zip(config.x,
1127                 tuple(FiniteSet(Rational(1, config.N)) ** config.N)[0])))]
1128             if not skip_general_jacobian_matrix:
1129                 row += [diff(f_i, x_i)]
1130         # we add the row to the content:
1131         jacobian_matrix_at_xhom_content += [row_at_xhom]
1132         jacobian_matrix_content += [row]
1133
1134     # with the previously calculated content we set up the Jacobian matrix:
1135     jacobian_matrix_at_xhom = Matrix(jacobian_matrix_at_xhom_content)
1136     jacobian_matrix = Matrix(jacobian_matrix_content)
1137
1138     # the number of free parameters equals the size of the alphabet substituted by the
1139     # rank of the Jacobian matrix:
1140     fp = config.N - jacobian_matrix_at_xhom.rank()
1141
1142     # set the solution type as "rank_J", to indicate that the solution has been found
1143     # using the rank of the Jacobian matrix:
1144     sol_type = "rank_J"
1145
1146     return fp, sol_type, jacobian_matrix_at_xhom, jacobian_matrix
1147
1148 def fp_to_counter(counter, progress=-1, print_progress=False, mode="constructive"):
1149     """
1150     Determines the number of free parameters to a specific counter, using one of the
1151     specific modi. Prints further possibly the progress of the calculation.
1152
1153     :param counter: the counter, which stands for a particular IFS.
1154     :param progress: actual progress.
1155     :param print_progress: boolean, if the progress should be printed.
1156     :param mode: the mode of the calculation. Either "existential" for the
1157                   existential way (using the rank of the Jacobian) or anything else for the
1158                   constructive way (resolving the equations and apply the Implicit Function
1159                   Theorem)
1160     :return: string, which contains the solution and can be written
1161             directly into the solution file.
1162     """
1163
1164     # print the progress:
1165     if print_progress:
1166         print_text = "overall-progress:"
1167         if config.file_extension != "":
1168             print_text = "overall-progress" + config.file_extension + ":"
1169         config.user_feedback(print_text + str(round(progress * 100, 1)).rjust(5) + "%\n"
1170             (Counter=" " + str(counter) + ").\n")
1171
1172     # calculate the number of free parameters based on mode:
1173     if mode.lower().startswith("ex"):
1174         fp, sol_type, jacobian_matrix_at_xhom, jacobian_matrix =
1175             fp_per_rank_jacobian(counter)
1176     else:
1177         fp, sol_type, complete_equation_list, free_equation_list, fixed_equation_list,
1178         free_var, fixed_var, m_0, set_u1, set_u2, jacobian_matrix,
1179         jacobian_matrix_without_substitution = fp_per_ift(counter)

```

```
1170 |  
1171 | # format the string for the solution file:  
1172 | result = "BEGIN," + str(counter) + "," + str(fp) + "," + sol_type + ",END\n"  
1173 | return result
```

Listing A.2: The module `B2.py` containing all needed functions to determine the number of free parameters.

```

1  # -*- coding: utf-8 -*-
2
3  """
4  @author:      Stefan Kohl, University of Stuttgart
5  @file name:   config.py
6  @description: This module provides all global variables and small, helpful functions.
7
8  """
9
10 # =====
11 # =                                     import modules                               =
12 # =====
13 from sympy import symbols
14 import os
15 import time
16
17
18 # =====
19 # =                                     definition of global variables             =
20 # =====
21
22 # the file handler where all the main results are written.
23 file = 0
24
25 # for the fractal and the path:
26 fractal_name = "unknownFractal"
27 file_extension = ""
28 fractal_file_folder = "sol/"
29 fractal_file_path = ""
30 fractal_file_name = "unknown"
31
32 # for the fractal the function for generating the equations:
33
34
35 def update_equations(ifs):
36     """
37     function for generating the equations to a ifs.
38
39     :param ifs:   ifs for which the equations should be generated.
40     :return:     list of equations
41     """
42     return ["update_equations not initialized:" + str(ifs)]
43
44
45 # the single word space:
46 single_wordspace = []
47
48 # the reverse mapping
49 mapping = []
50
51 # the length of the alphabet:
52 N = 0
53
54 # maximal number of letters of the Alphabet actually implemented:
55 N_max = 8
56
57 # the variables:
58 x = [symbols('x%d' % i, positive=True) for i in range(N_max)]
59
60
61 # =====
62 # =                                     definition of functions                               =
63 # =====
64
65 def create_folder(directory):
66     """
67     create a folder structure for a directory.
68
69     :param directory: path, which should exist.
70     :return:         (nothing)
71     """
72
73     try:

```

```

74         if not os.path.exists(directory):
75             # create the path, since it does not exist.
76             os.makedirs(directory)
77     except OSError:
78         # print Error, if there was an error:
79         print("Error: Creating directory " + directory + "\n")
80
81
82 def user_feedback(text, indentation=0, line_length=60, print_date=True):
83     """
84     prints some text for the user like progress of the program etc.
85
86     :param text:          feedback text.
87     :param indentation:   The size of the indentation of the text.
88     :param line_length:   maximal number of characters per line.
89     :param print_date:    boolean, if the date should be printed.
90     :return:              (nothing)
91     """
92     full_text = " "*indentation*4 + text
93     # the text goes over more than one line:
94     if len(full_text) > line_length:
95         print(full_text[:line_length], flush=True)
96         user_feedback(full_text[line_length:], indentation=indentation,
97                       line_length=line_length, print_date=print_date)
98     else:
99         if print_date:
100             if len(full_text) > line_length - 1:
101                 full_text = full_text + "\n" + " "*line_length
102             else:
103                 full_text = full_text.ljust(line_length)
104             print(full_text + "It is: " + time.strftime('%Y-%m-%d%H:%M:%S'),
105                   flush=True)
106     else:
107         print(full_text, flush=True)

```

Listing A.3: The module `config.py` contains all definitions of global variables (and functions).

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  '''
5  @author:      Stefan Kohl, University of Stuttgart
6  @file name:   find_example_with_fp.py
7  @description: main program to find for every appearance of free parameter an example
   to a certain fractal.
8
9  '''
10
11 # =====
12 # =                               import modules                               =
13 # =====
14 import fractals
15 import additional
16
17
18 # =====
19 # =                               run the program                               =
20 # =====
21
22
23 if __name__ == "__main__":
24     additional.find_example_to_all_fp(fractal="SG2")
25     additional.find_example_to_all_fp(fractal="ST")
26     additional.find_example_to_all_fp(fractal="SG3")
27     additional.find_example_to_all_fp(fractal="SG4")
28     additional.find_example_to_all_fp(fractal="SC")
29     additional.find_example_to_all_fp(fractal="Vicsek")
30     additional.find_example_to_all_fp(fractal="Pentagasket")
31     additional.find_example_to_all_fp(fractal="PentagasketWITHOUTHOLE")
32     additional.find_example_to_all_fp(fractal="Hexagasket")
33     additional.find_example_to_all_fp(fractal="HexagasketW")

```

Listing A.4: The executable `find_example_with_fp.py` determines afterwards examples to a fractal with all free parameters which appear.

```

1  # -*- coding: utf-8 -*-
2
3  """
4  @author:      Stefan Kohl, University of Stuttgart
5  @file name:   fractals.py
6  @description: contains functions for the definition of the fractal.
7
8  """
9
10 # =====
11 # =                                     import modules                               =
12 # =====
13 # load the config-file with global variables:
14 import config
15
16 from sympy import *
17 import itertools
18 import warnings
19
20
21 # =====
22 # =                                     definition of functions                       =
23 # =====
24
25 def init_fractal(fractal_name, file_extension=""):
26     """
27     initialize all parameters for a specific fractal
28
29     :param fractal_name:    fractal which should be considered
30     :param file_extension:  the extension of the written file (e.g.: "-test", "-temp")
31     :return:                (nothing)
32     """
33
34     # We set the length of the alphabet (global variable N) back to zero. If after all
35     # possible fractals the length of the alphabet is still zero, we raise an error,
36     # since we did not initialize a fractal.
37     config.N = 0
38
39     # set file extension:
40     config.file_extension = file_extension
41
42     # initialize update_equations_local as empty (and overwrite it later)
43     def update_equations_local(ifs):
44         return ["in_{}_init_fractal_not_initialized!".format(ifs)]
45
46     # depending of "fractal_name" consider the different fractals.
47
48     # Parameters for the Sierpinski Gasket:
49     if fractal_name.lower() == "sg2":
50
51         # set fractal name:
52         config.fractal_name = "SG2"
53
54         # set the length of the alphabet:
55         config.N = 3
56
57         def update_equations_local(ifs):
58             """
59             define the equations for SG2
60
61             :param ifs:    which specific ifs should be considered.
62             :return:       equation_list
63             """
64             equation_list = [
65                 sum(config.x) - 1,
66                 config.x[0] * config.x[ifs[0][1]] - config.x[1] * config.x[ifs[1][0]],
67                 config.x[0] * config.x[ifs[0][2]] - config.x[2] * config.x[ifs[2][0]],
68                 config.x[1] * config.x[ifs[1][2]] - config.x[2] * config.x[ifs[2][1]]
69             ]
70             return equation_list
71
72         # define the single word space:
73         config.single_wordspace = [list(entry) for entry in

```

```

72         itertools.permutations(range(3))]
73
74 # Parameters for the Sierpinski tetrahedron:
75 if fractal_name.lower() == "st" or fractal_name.lower() == "sierpinski tetrahedron":
76     # set fractal name:
77     config.fractal_name = "ST"
78
79     # set the length of the alphabet:
80     config.N = 4
81
82     def update_equations_local(ifs):
83         """
84         define the equations for the Sierpinski tetrahedron
85
86         :param ifs: which specific ifs should be considered.
87         :return: equation_list
88         """
89         equation_list = [
90             sum(config.x) - 1,
91             config.x[0] * config.x[ifs[0][1]] - config.x[1] * config.x[ifs[1][0]],
92             config.x[0] * config.x[ifs[0][2]] - config.x[2] * config.x[ifs[2][0]],
93             config.x[0] * config.x[ifs[0][3]] - config.x[3] * config.x[ifs[3][0]],
94             config.x[1] * config.x[ifs[1][2]] - config.x[2] * config.x[ifs[2][1]],
95             config.x[1] * config.x[ifs[1][3]] - config.x[3] * config.x[ifs[3][1]],
96             config.x[2] * config.x[ifs[2][3]] - config.x[3] * config.x[ifs[3][2]]
97         ]
98         return equation_list
99
100     # define the single word space:
101     config.single_wordspace = [list(entry) for entry in
102                                itertools.permutations(range(4))]
103
104 # Parameters for the SG(3)
105 if fractal_name.lower() == "sg3":
106     # set fractal name:
107     config.fractal_name = "SG3"
108
109     # set the length of the alphabet:
110     config.N = 6
111
112     def update_equations_local(ifs):
113         """
114         define the equations for SG3
115
116         :param ifs: which specific ifs should be considered.
117         :return: equation_list
118         """
119         equation_list = [
120             sum(config.x) - 1,
121             config.x[0] * config.x[ifs[0][1]] - config.x[3] * config.x[ifs[3][0]],
122             config.x[0] * config.x[ifs[0][2]] - config.x[4] * config.x[ifs[4][0]],
123             config.x[1] * config.x[ifs[1][0]] - config.x[3] * config.x[ifs[3][1]],
124             config.x[1] * config.x[ifs[1][2]] - config.x[5] * config.x[ifs[5][1]],
125             config.x[2] * config.x[ifs[2][0]] - config.x[4] * config.x[ifs[4][2]],
126             config.x[2] * config.x[ifs[2][1]] - config.x[5] * config.x[ifs[5][2]],
127             config.x[3] * config.x[ifs[3][2]] - config.x[4] * config.x[ifs[4][1]],
128             config.x[3] * config.x[ifs[3][2]] - config.x[5] * config.x[ifs[5][0]],
129             config.x[4] * config.x[ifs[4][1]] - config.x[5] * config.x[ifs[5][0]]
130         ]
131         return equation_list
132
133     # define the single word space:
134     config.single_wordspace = [
135         [0, 1, 2, 3, 4, 5],
136         [0, 2, 1, 4, 3, 5],
137         [1, 0, 2, 3, 5, 4],
138         [1, 2, 0, 5, 3, 4],
139         [2, 0, 1, 4, 5, 3],
140         [2, 1, 0, 5, 4, 3]]
141
142 # Parameters for the SG(4)

```

```

143 if fractal_name.lower() == "sg4":
144
145     # set fractal name:
146     config.fractal_name = "SG4"
147
148     # set the length of the alphabet:
149     config.N = 10
150
151     def update_equations_local(ifs):
152         """
153         define the equations for SG4
154
155         :param ifs: which specific ifs should be considered.
156         :return: equation_list
157         """
158         equation_list = [
159             sum(config.x) - 1,
160             config.x[0]*config.x[ifs[0][1]] - config.x[3]*config.x[ifs[3][0]],
161             config.x[0]*config.x[ifs[0][2]] - config.x[4]*config.x[ifs[4][0]],
162             config.x[1]*config.x[ifs[1][0]] - config.x[5]*config.x[ifs[5][1]],
163             config.x[1]*config.x[ifs[1][2]] - config.x[8]*config.x[ifs[8][1]],
164             config.x[2]*config.x[ifs[2][0]] - config.x[7]*config.x[ifs[7][2]],
165             config.x[2]*config.x[ifs[2][1]] - config.x[9]*config.x[ifs[9][2]],
166             config.x[3]*config.x[ifs[3][1]] - config.x[5]*config.x[ifs[5][0]],
167             config.x[4]*config.x[ifs[4][2]] - config.x[7]*config.x[ifs[7][0]],
168             config.x[8]*config.x[ifs[8][2]] - config.x[9]*config.x[ifs[9][1]],
169             config.x[3]*config.x[ifs[3][2]] - config.x[4]*config.x[ifs[4][1]],
170             config.x[3]*config.x[ifs[3][2]] - config.x[6]*config.x[ifs[6][0]],
171             config.x[4]*config.x[ifs[4][1]] - config.x[6]*config.x[ifs[6][0]],
172             config.x[5]*config.x[ifs[5][2]] - config.x[6]*config.x[ifs[6][1]],
173             config.x[5]*config.x[ifs[5][2]] - config.x[8]*config.x[ifs[8][0]],
174             config.x[6]*config.x[ifs[6][1]] - config.x[8]*config.x[ifs[8][0]],
175             config.x[6]*config.x[ifs[6][2]] - config.x[7]*config.x[ifs[7][1]],
176             config.x[6]*config.x[ifs[6][2]] - config.x[9]*config.x[ifs[9][0]],
177             config.x[7]*config.x[ifs[7][1]] - config.x[9]*config.x[ifs[9][0]]
178         ]
179         return equation_list
180
181     # define the single word space:
182     config.single_wordspace = [
183         [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
184         [0, 2, 1, 4, 3, 7, 6, 5, 9, 8],
185         [1, 0, 2, 5, 8, 3, 6, 9, 4, 7],
186         [1, 2, 0, 8, 5, 9, 6, 3, 7, 4],
187         [2, 0, 1, 7, 9, 4, 6, 8, 3, 5],
188         [2, 1, 0, 9, 7, 8, 6, 4, 5, 3]]
189
190     # Parameters for the Vicsek fractal:
191     if fractal_name[0:6].lower() == "vicsek":
192
193         # set fractal name:
194         config.fractal_name = "Vicsek"
195
196         # set the length of the alphabet:
197         config.N = 5
198
199         # define the equations for Vicsek:
200         def update_equations_local(ifs):
201             """
202             define the equations for the Vicsek fractal
203
204             :param ifs: which specific ifs should be considered.
205             :return: equation_list
206             """
207             equation_list = [
208                 sum(config.x) - 1,
209                 config.x[0] * config.x[ifs[0][2]] - config.x[4] * config.x[ifs[4][0]],
210                 config.x[1] * config.x[ifs[1][3]] - config.x[4] * config.x[ifs[4][1]],
211                 config.x[2] * config.x[ifs[2][0]] - config.x[4] * config.x[ifs[4][2]],
212                 config.x[3] * config.x[ifs[3][1]] - config.x[4] * config.x[ifs[4][3]]
213             ]
214             return equation_list
215

```



```

216 # define the single word space:
217 config.single_wordspace = []
218 for n in range(4):
219     temp = []
220     for m in range(4):
221         temp += [(n + m) % 4]
222     temp += [4]
223     config.single_wordspace += [temp]
224 for n in range(4):
225     temp = []
226     for m in range(4):
227         temp += [-(n + m) % 4]
228     temp += [4]
229     config.single_wordspace += [temp]
230
231 # Parameters for the Pentagasket (with hole):
232 if (fractal_name.lower() == "pentagasket") or (fractal_name[0:12].lower() ==
    "pentagasket-"):
233
234     # set fractal name:
235     config.fractal_name = "Pentagasket"
236
237     # set the length of the alphabet:
238     config.N = 5
239
240     def update_equations_local(ifs):
241         """
242         define the equations for the Pentagasket with hole
243
244         :param ifs: which specific ifs should be considered.
245         :return: equation_list
246         """
247         equation_list = [
248             sum(config.x) - 1,
249             config.x[0] * config.x[ifs[0][2]] - config.x[1] * config.x[ifs[1][4]],
250             config.x[1] * config.x[ifs[1][3]] - config.x[2] * config.x[ifs[2][0]],
251             config.x[2] * config.x[ifs[2][4]] - config.x[3] * config.x[ifs[3][1]],
252             config.x[3] * config.x[ifs[3][0]] - config.x[4] * config.x[ifs[4][2]],
253             config.x[4] * config.x[ifs[4][1]] - config.x[0] * config.x[ifs[0][3]]
254         ]
255         return equation_list
256
257     # define the single word space:
258     config.single_wordspace = []
259     for n in range(5):
260         temp = []
261         for m in range(5):
262             temp += [(n + m) % 5]
263         config.single_wordspace += [temp]
264     for n in range(5):
265         temp = []
266         for m in range(5):
267             temp += [-(n + m) % 5]
268         config.single_wordspace += [temp]
269
270 # Parameters for the Pentagasket (WITHOUT HOLE):
271 if fractal_name[0:12].lower() == "pentagasketw":
272
273     # set fractal name:
274     config.fractal_name = "PentagasketWITHOUTHOLE"
275
276     # set the length of the alphabet:
277     config.N = 6
278
279     # define the equations for Pentagasket (WITHOUT HOLE):
280     def update_equations_local(ifs):
281         """
282         define the equations for the Pentagasket without hole
283
284         :param ifs: which specific ifs should be considered.
285         :return: equation_list
286         """
287         equation_list = [

```

```

288     sum(config.x) - 1,
289     config.x[0] * config.x[ifs[0][2]] - config.x[1] * config.x[ifs[1][4]],
290     config.x[0] * config.x[ifs[0][2]] - config.x[5] * config.x[ifs[5][3]],
291     config.x[1] * config.x[ifs[1][4]] - config.x[5] * config.x[ifs[5][3]],
292     config.x[1] * config.x[ifs[1][3]] - config.x[2] * config.x[ifs[2][0]],
293     config.x[1] * config.x[ifs[1][3]] - config.x[5] * config.x[ifs[5][4]],
294     config.x[2] * config.x[ifs[2][0]] - config.x[5] * config.x[ifs[5][4]],
295     config.x[2] * config.x[ifs[2][4]] - config.x[3] * config.x[ifs[3][1]],
296     config.x[2] * config.x[ifs[2][4]] - config.x[5] * config.x[ifs[5][0]],
297     config.x[3] * config.x[ifs[3][1]] - config.x[5] * config.x[ifs[5][0]],
298     config.x[3] * config.x[ifs[3][0]] - config.x[4] * config.x[ifs[4][2]],
299     config.x[3] * config.x[ifs[3][0]] - config.x[5] * config.x[ifs[5][1]],
300     config.x[4] * config.x[ifs[4][2]] - config.x[5] * config.x[ifs[5][1]],
301     config.x[4] * config.x[ifs[4][1]] - config.x[0] * config.x[ifs[0][3]],
302     config.x[4] * config.x[ifs[4][1]] - config.x[5] * config.x[ifs[5][2]],
303     config.x[0] * config.x[ifs[0][3]] - config.x[5] * config.x[ifs[5][2]],
304     config.x[0] * config.x[ifs[0][2]] *
305         config.x[ifs[ifs[0][2]][ifs[0][3]]] - config.x[5] *
306         config.x[ifs[5][3]] * config.x[ifs[ifs[5][3]][ifs[5][2]]],
307     config.x[1] * config.x[ifs[1][4]] *
308         config.x[ifs[ifs[1][4]][ifs[1][3]]] - config.x[5] *
309         config.x[ifs[5][3]] * config.x[ifs[ifs[5][3]][ifs[5][4]]],
310     config.x[1] * config.x[ifs[1][3]] *
311         config.x[ifs[ifs[1][3]][ifs[1][4]]] - config.x[5] *
312         config.x[ifs[5][4]] * config.x[ifs[ifs[5][4]][ifs[5][3]]],
313     config.x[2] * config.x[ifs[2][0]] *
314         config.x[ifs[ifs[2][0]][ifs[2][4]]] - config.x[5] *
315         config.x[ifs[5][4]] * config.x[ifs[ifs[5][4]][ifs[5][0]]],
316     config.x[2] * config.x[ifs[2][4]] *
317         config.x[ifs[ifs[2][4]][ifs[2][0]]] - config.x[5] *
318         config.x[ifs[5][0]] * config.x[ifs[ifs[5][0]][ifs[5][4]]],
319     config.x[3] * config.x[ifs[3][1]] *
320         config.x[ifs[ifs[3][1]][ifs[3][0]]] - config.x[5] *
321         config.x[ifs[5][0]] * config.x[ifs[ifs[5][0]][ifs[5][1]]],
322     config.x[3] * config.x[ifs[3][0]] *
323         config.x[ifs[ifs[3][0]][ifs[3][1]]] - config.x[5] *
324         config.x[ifs[5][1]] * config.x[ifs[ifs[5][1]][ifs[5][0]]],
325     config.x[4] * config.x[ifs[4][2]] *
326         config.x[ifs[ifs[4][2]][ifs[4][1]]] - config.x[5] *
327         config.x[ifs[5][1]] * config.x[ifs[ifs[5][1]][ifs[5][2]]],
328     config.x[4] * config.x[ifs[4][1]] *
329         config.x[ifs[ifs[4][1]][ifs[4][2]]] - config.x[5] *
330         config.x[ifs[5][2]] * config.x[ifs[ifs[5][2]][ifs[5][1]]],
331     config.x[0] * config.x[ifs[0][3]] *
332         config.x[ifs[ifs[0][3]][ifs[0][2]]] - config.x[5] *
333         config.x[ifs[5][2]] * config.x[ifs[ifs[5][2]][ifs[5][3]]]
334     ]
335     return equation_list
336
337     # define the single word space:
338     config.single_wordspace = []
339     for n in range(5):
340         temp = []
341         for m in range(5):
342             temp += [(n + m) % 5]
343         temp += [5]
344         config.single_wordspace += [temp]
345     for n in range(5):
346         temp = []
347         for m in range(5):
348             temp += [-(n + m) % 5]
349         temp += [5]
350         config.single_wordspace += [temp]
351
352     # Parameters for the Hexagasket:
353     if fractal_name.lower() == "hexagasket":
354
355         # set fractal name:
356         config.fractal_name = "Hexagasket"
357
358         # set the length of the alphabet:
359         config.N = 6
360

```

```

341 # define the equations for Hexagasket:
342 def update_equations_local(ifs):
343     """
344     define the equations for the Hexagasket
345
346     :param ifs: which specific ifs should be considered.
347     :return: equation_list
348     """
349     equation_list = [
350         sum(config.x) - 1,
351         config.x[0] * config.x[ifs[0][2]] - config.x[1] * config.x[ifs[1][5]],
352         config.x[1] * config.x[ifs[1][3]] - config.x[2] * config.x[ifs[2][0]],
353         config.x[2] * config.x[ifs[2][4]] - config.x[3] * config.x[ifs[3][1]],
354         config.x[3] * config.x[ifs[3][5]] - config.x[4] * config.x[ifs[4][2]],
355         config.x[4] * config.x[ifs[4][0]] - config.x[5] * config.x[ifs[5][3]],
356         config.x[5] * config.x[ifs[5][1]] - config.x[0] * config.x[ifs[0][4]]
357     ]
358     return equation_list
359
360 config.single_wordspace = []
361 for n in range(6):
362     temp = []
363     for m in range(6):
364         temp += [(n + m) % 6]
365     config.single_wordspace += [temp]
366 for n in range(6):
367     temp = []
368     for m in range(6):
369         temp += [-(n + m) % 6]
370     config.single_wordspace += [temp]
371
372 # Parameters for the Hexagasket without Hole:
373 if fractal_name[0:11].lower() == "hexagasketw":
374
375     # set fractal name:
376     config.fractal_name = "HexagasketWITHOUTHOLE"
377
378     # set the length of the alphabet:
379     config.N = 7
380
381     # define the equations for HexagasketWITHOUTHOLE:
382     def update_equations_local(ifs):
383         """
384         define the equations for HexagasketWITHOUTHOLE
385
386         :param ifs: which specific ifs should be considered.
387         :return: equation_list
388         """
389         equation_list = [
390             sum(config.x) - 1,
391             config.x[0] * config.x[ifs[0][2]] - config.x[1] * config.x[ifs[1][5]],
392             config.x[0] * config.x[ifs[0][3]] - config.x[6] * config.x[ifs[6][0]],
393             config.x[0] * config.x[ifs[0][4]] - config.x[5] * config.x[ifs[5][1]],
394             config.x[1] * config.x[ifs[1][3]] - config.x[2] * config.x[ifs[2][0]],
395             config.x[1] * config.x[ifs[1][4]] - config.x[6] * config.x[ifs[6][1]],
396             config.x[2] * config.x[ifs[2][4]] - config.x[3] * config.x[ifs[3][1]],
397             config.x[2] * config.x[ifs[2][5]] - config.x[6] * config.x[ifs[6][2]],
398             config.x[3] * config.x[ifs[3][0]] - config.x[6] * config.x[ifs[6][3]],
399             config.x[3] * config.x[ifs[3][5]] - config.x[4] * config.x[ifs[4][2]],
400             config.x[4] * config.x[ifs[4][0]] - config.x[5] * config.x[ifs[5][3]],
401             config.x[4] * config.x[ifs[4][1]] - config.x[6] * config.x[ifs[6][4]],
402             config.x[5] * config.x[ifs[5][2]] - config.x[6] * config.x[ifs[6][5]]
403         ]
404         return equation_list
405
406 config.single_wordspace = []
407 for n in range(6):
408     temp = []
409     for m in range(6):
410         temp += [(n + m) % 6]
411     temp += [6]
412     config.single_wordspace += [temp]
413 for n in range(6):

```

```

414     temp = []
415     for m in range(6):
416         temp += [-(n + m) % 6]
417     temp += [6]
418     config.single_wordspace += [temp]
419
420 # Parameters for the Sierpinski carpet:
421 if fractal_name[0:2].lower() == "sc":
422
423     # set fractal name:
424     config.fractal_name = "SC"
425
426     # set the length of the alphabet:
427     config.N = 8
428
429     # define the equations for SC:
430     def update_equations_local(ifs):
431         """
432         define the equations for Sierpinski carpet
433
434         :param ifs: which specific ifs should be considered.
435         :return: equation_list
436         """
437         equation_list = [
438             sum(config.x) - 1,
439             config.x[0] * config.x[ifs[0][2]] - config.x[1] * config.x[ifs[1][0]],
440             config.x[1] * config.x[ifs[1][2]] - config.x[2] * config.x[ifs[2][0]],
441             config.x[0] * config.x[ifs[0][6]] - config.x[7] * config.x[ifs[7][0]],
442             config.x[2] * config.x[ifs[2][4]] - config.x[3] * config.x[ifs[3][2]],
443             config.x[7] * config.x[ifs[7][6]] - config.x[6] * config.x[ifs[6][0]],
444             config.x[3] * config.x[ifs[3][4]] - config.x[4] * config.x[ifs[4][2]],
445             config.x[6] * config.x[ifs[6][4]] - config.x[5] * config.x[ifs[5][6]],
446             config.x[5] * config.x[ifs[5][4]] - config.x[4] * config.x[ifs[4][6]],
447             config.x[0] * config.x[ifs[0][4]] - config.x[1] * config.x[ifs[1][6]],
448             config.x[1] * config.x[ifs[1][4]] - config.x[7] * config.x[ifs[7][2]],
449             config.x[1] * config.x[ifs[1][6]] - config.x[7] * config.x[ifs[7][2]],
450             config.x[1] * config.x[ifs[1][4]] - config.x[2] * config.x[ifs[2][6]],
451             config.x[1] * config.x[ifs[1][4]] - config.x[3] * config.x[ifs[3][0]],
452             config.x[2] * config.x[ifs[2][6]] - config.x[3] * config.x[ifs[3][0]],
453             config.x[7] * config.x[ifs[7][4]] - config.x[6] * config.x[ifs[6][2]],
454             config.x[7] * config.x[ifs[7][4]] - config.x[5] * config.x[ifs[5][0]],
455             config.x[6] * config.x[ifs[6][2]] - config.x[5] * config.x[ifs[5][0]],
456             config.x[5] * config.x[ifs[5][2]] - config.x[3] * config.x[ifs[3][6]],
457             config.x[5] * config.x[ifs[5][2]] - config.x[4] * config.x[ifs[4][0]],
458             config.x[3] * config.x[ifs[3][6]] - config.x[4] * config.x[ifs[4][0]],
459             config.x[0] * config.x[ifs[0][2]] *
460                 config.x[ifs[ifs[0][2]][ifs[0][4]]] - config.x[0] *
461                 config.x[ifs[0][3]] * config.x[ifs[ifs[0][3]][ifs[0][2]]],
462             config.x[0] * config.x[ifs[0][2]] *
463                 config.x[ifs[ifs[0][2]][ifs[0][4]]] - config.x[1] *
464                 config.x[ifs[1][0]] * config.x[ifs[ifs[1][0]][ifs[1][6]]],
465             config.x[0] * config.x[ifs[0][2]] *
466                 config.x[ifs[ifs[0][2]][ifs[0][4]]] - config.x[1] *
467                 config.x[ifs[1][7]] * config.x[ifs[ifs[1][7]][ifs[1][0]]],
468             config.x[0] * config.x[ifs[0][3]] *
469                 config.x[ifs[ifs[0][3]][ifs[0][2]]] - config.x[1] *
470                 config.x[ifs[1][0]] * config.x[ifs[ifs[1][0]][ifs[1][6]]],
471             config.x[0] * config.x[ifs[0][3]] *
472                 config.x[ifs[ifs[0][3]][ifs[0][2]]] - config.x[1] *
473                 config.x[ifs[1][7]] * config.x[ifs[ifs[1][7]][ifs[1][0]]],
474             config.x[1] * config.x[ifs[1][0]] *
475                 config.x[ifs[ifs[1][0]][ifs[1][6]]] - config.x[1] *
476                 config.x[ifs[1][7]] * config.x[ifs[ifs[1][7]][ifs[1][0]]],
477             config.x[0] * config.x[ifs[0][3]] *
478                 config.x[ifs[ifs[0][3]][ifs[0][4]]] - config.x[0] *
479                 config.x[ifs[0][4]] * config.x[ifs[ifs[0][4]][ifs[0][2]]],
480             config.x[0] * config.x[ifs[0][3]] *
481                 config.x[ifs[ifs[0][3]][ifs[0][4]]] - config.x[1] *
482                 config.x[ifs[1][7]] * config.x[ifs[ifs[1][7]][ifs[1][6]]],
483             config.x[0] * config.x[ifs[0][3]] *
484                 config.x[ifs[ifs[0][3]][ifs[0][4]]] - config.x[1] *
485                 config.x[ifs[1][6]] * config.x[ifs[ifs[1][6]][ifs[1][0]]],
486             config.x[0] * config.x[ifs[0][4]] *

```



```

493     config.x[ifs [7][1]] * config.x[ifs [ifs [7][1]][ifs [7][2]]],
config.x[0] * config.x[ifs [0][4]] *
494     config.x[ifs [ifs [0][4]][ifs [0][6]]] - config.x[7] *
config.x[ifs [7][2]] * config.x[ifs [ifs [7][2]][ifs [7][0]]],
495     config.x[7] * config.x[ifs [7][1]] *
config.x[ifs [ifs [7][1]][ifs [7][2]]] - config.x[7] *
496     config.x[ifs [7][2]] * config.x[ifs [ifs [7][2]][ifs [7][0]]],
config.x[7] * config.x[ifs [7][6]] *
497     config.x[ifs [ifs [7][6]][ifs [7][4]]] - config.x[7] *
config.x[ifs [7][5]] * config.x[ifs [ifs [7][5]][ifs [7][6]]],
498     config.x[7] * config.x[ifs [7][6]] *
config.x[ifs [ifs [7][6]][ifs [7][4]]] - config.x[6] *
499     config.x[ifs [6][0]] * config.x[ifs [ifs [6][0]][ifs [6][2]]],
config.x[7] * config.x[ifs [7][6]] *
500     config.x[ifs [ifs [7][6]][ifs [7][4]]] - config.x[6] *
config.x[ifs [6][1]] * config.x[ifs [ifs [6][1]][ifs [6][0]]],
501     config.x[6] * config.x[ifs [6][0]] *
config.x[ifs [ifs [6][0]][ifs [6][2]]] - config.x[6] *
502     config.x[ifs [6][1]] * config.x[ifs [ifs [6][1]][ifs [6][0]]],
config.x[7] * config.x[ifs [7][5]] *
503     config.x[ifs [ifs [7][5]][ifs [7][4]]] - config.x[7] *
config.x[ifs [7][4]] * config.x[ifs [ifs [7][4]][ifs [7][6]]],
504     config.x[7] * config.x[ifs [7][5]] *
config.x[ifs [ifs [7][5]][ifs [7][4]]] - config.x[6] *
505     config.x[ifs [6][1]] * config.x[ifs [ifs [6][1]][ifs [6][2]]],
config.x[7] * config.x[ifs [7][5]] *
506     config.x[ifs [ifs [7][5]][ifs [7][4]]] - config.x[6] *
config.x[ifs [6][2]] * config.x[ifs [ifs [6][2]][ifs [6][0]]],
507     config.x[7] * config.x[ifs [7][4]] *
config.x[ifs [ifs [7][4]][ifs [7][6]]] - config.x[6] *
508     config.x[ifs [6][2]] * config.x[ifs [ifs [6][2]][ifs [6][0]]],
config.x[6] * config.x[ifs [6][1]] *
509     config.x[ifs [ifs [6][1]][ifs [6][2]]] - config.x[6] *
config.x[ifs [6][2]] * config.x[ifs [ifs [6][2]][ifs [6][0]]],
510     config.x[2] * config.x[ifs [2][6]] *
config.x[ifs [ifs [2][6]][ifs [2][4]]] - config.x[2] *
511     config.x[ifs [2][5]] * config.x[ifs [ifs [2][5]][ifs [2][6]]],
config.x[2] * config.x[ifs [2][6]] *
512     config.x[ifs [ifs [2][6]][ifs [2][4]]] - config.x[3] *
config.x[ifs [3][0]] * config.x[ifs [ifs [3][0]][ifs [3][2]]],
513     config.x[2] * config.x[ifs [2][6]] *
config.x[ifs [ifs [2][6]][ifs [2][4]]] - config.x[3] *
514     config.x[ifs [3][1]] * config.x[ifs [ifs [3][1]][ifs [3][0]]],
config.x[2] * config.x[ifs [2][5]] *
515     config.x[ifs [ifs [2][5]][ifs [2][6]]] - config.x[3] *
config.x[ifs [3][0]] * config.x[ifs [ifs [3][0]][ifs [3][2]]],
516     config.x[2] * config.x[ifs [2][5]] *
config.x[ifs [ifs [2][5]][ifs [2][4]]] - config.x[3] *
config.x[ifs [3][1]] * config.x[ifs [ifs [3][1]][ifs [3][0]]],
config.x[2] * config.x[ifs [2][4]] * config.x[ifs [ifs [2][4]][ifs [2][6]]],
517     config.x[2] * config.x[ifs [2][5]] *
config.x[ifs [ifs [2][5]][ifs [2][4]]] - config.x[3] *
config.x[ifs [3][1]] * config.x[ifs [ifs [3][1]][ifs [3][2]]],
518     config.x[2] * config.x[ifs [2][5]] *
config.x[ifs [ifs [2][5]][ifs [2][4]]] - config.x[3] *
config.x[ifs [3][2]] * config.x[ifs [ifs [3][2]][ifs [3][0]]],
519     config.x[2] * config.x[ifs [2][4]] *
config.x[ifs [ifs [2][4]][ifs [2][6]]] - config.x[3] *
config.x[ifs [3][1]] * config.x[ifs [ifs [3][1]][ifs [3][2]]],

```



```

542         config.x[ifs[ifs[6][4]][ifs[6][2]]] - config.x[5] *
        config.x[ifs[5][6]] * config.x[ifs[ifs[5][6]][ifs[5][0]]],
543     config.x[5] * config.x[ifs[5][7]] *
        config.x[ifs[ifs[5][7]][ifs[5][6]]] - config.x[5] *
        config.x[ifs[5][6]] * config.x[ifs[ifs[5][6]][ifs[5][0]]],
544     config.x[5] * config.x[ifs[5][2]] *
        config.x[ifs[ifs[5][2]][ifs[5][4]]] - config.x[5] *
        config.x[ifs[5][3]] * config.x[ifs[ifs[5][3]][ifs[5][2]]],
545     config.x[5] * config.x[ifs[5][2]] *
        config.x[ifs[ifs[5][2]][ifs[5][4]]] - config.x[4] *
        config.x[ifs[4][0]] * config.x[ifs[ifs[4][0]][ifs[4][6]]],
546     config.x[5] * config.x[ifs[5][2]] *
        config.x[ifs[ifs[5][2]][ifs[5][4]]] - config.x[4] *
        config.x[ifs[4][7]] * config.x[ifs[ifs[4][7]][ifs[4][0]]],
547     config.x[5] * config.x[ifs[5][3]] *
        config.x[ifs[ifs[5][3]][ifs[5][2]]] - config.x[4] *
        config.x[ifs[4][0]] * config.x[ifs[ifs[4][0]][ifs[4][6]]],
548     config.x[5] * config.x[ifs[5][3]] *
        config.x[ifs[ifs[5][3]][ifs[5][2]]] - config.x[4] *
        config.x[ifs[4][7]] * config.x[ifs[ifs[4][7]][ifs[4][0]]],
549     config.x[4] * config.x[ifs[4][0]] *
        config.x[ifs[ifs[4][0]][ifs[4][6]]] - config.x[4] *
        config.x[ifs[4][7]] * config.x[ifs[ifs[4][7]][ifs[4][0]]],
550     config.x[5] * config.x[ifs[5][3]] *
        config.x[ifs[ifs[5][3]][ifs[5][4]]] - config.x[5] *
        config.x[ifs[5][4]] * config.x[ifs[ifs[5][4]][ifs[5][2]]],
551     config.x[5] * config.x[ifs[5][3]] *
        config.x[ifs[ifs[5][3]][ifs[5][4]]] - config.x[4] *
        config.x[ifs[4][7]] * config.x[ifs[ifs[4][7]][ifs[4][6]]],
552     config.x[5] * config.x[ifs[5][3]] *
        config.x[ifs[ifs[5][3]][ifs[5][4]]] - config.x[4] *
        config.x[ifs[4][6]] * config.x[ifs[ifs[4][6]][ifs[4][0]]],
553     config.x[5] * config.x[ifs[5][4]] *
        config.x[ifs[ifs[5][4]][ifs[5][2]]] - config.x[4] *
        config.x[ifs[4][7]] * config.x[ifs[ifs[4][7]][ifs[4][6]]],
554     config.x[5] * config.x[ifs[5][4]] *
        config.x[ifs[ifs[5][4]][ifs[5][2]]] - config.x[4] *
        config.x[ifs[4][6]] * config.x[ifs[ifs[4][6]][ifs[4][0]]],
555     config.x[4] * config.x[ifs[4][7]] *
        config.x[ifs[ifs[4][7]][ifs[4][6]]] - config.x[4] *
        config.x[ifs[4][6]] * config.x[ifs[ifs[4][6]][ifs[4][0]]]
556     ]
557     return equation_list
558
559     # define the single word space:
560     config.single_wordspace = []
561     for n in range(4):
562         temp = []
563         for m in range(8):
564             temp += [(2 * n + m) % 8]
565         config.single_wordspace += [temp]
566     for n in range(4):
567         temp = []
568         for m in range(8):
569             temp += [-(2 * n + m) % 8]
570         config.single_wordspace += [temp]
571
572     # raise an error, if no fractal was initialized:
573     if config.N == 0:
574         raise ValueError("Unable to initialize the requested fractal" +
575             fractal_name + "\!")")
576
577     # generate symbols for word 1, ..., N, accessing them via x[0], ... x[N-1]
578     config.x = [symbols('x%d' % i, positive=True) for i in range(config.N)]
579
580     # set the name of the file:
581     config.fractal_file_name = config.fractal_name
582     if file_extension != "":
583         config.fractal_file_name = config.fractal_name + "._" + file_extension
584
585     # copy function update_equations_local to config.update_equations
586     config.update_equations = update_equations_local

```



```

586     # generate the mapping:
587     config.mapping = single_wordspace_to_mapping(config.single_wordspace)
588
589
590 def single_wordspace_to_mapping(single_wordspace):
591     """
592     turns the single word space into a representation for mappings.
593
594     :param single_wordspace: the single word space to whom the mappings should be
595                             created.
596     :return:                  mapping
597     """
598
599     result = []
600     for entry in single_wordspace:
601
602         new_entry = []
603         for counter in range(config.N):
604             new_entry += [entry.index(counter)]
605
606         result += [list(new_entry)]
607     return result
608
609 def iso_counter_to_ifs(counter):
610     """
611     maps a specific counter to its associated ifs
612
613     :param counter: a number in [0, single_wordspace**N] which represents a specific
614                    ifs
615     :return:        an ifs.
616     """
617
618     # get the length of the single word space.
619     len_sw = len(config.single_wordspace)
620
621     # initialize the result as empty.
622     result = []
623
624     # x gets at the beginning the value of counter and gets then reduced.
625     x = counter
626
627     for _ in range(config.N):
628         # the remainder determines which mapping was chosen.
629         remainder = int(x % len_sw)
630         # add this mapping from single_wordspace to results:
631         result += [config.single_wordspace[remainder]]
632         # subtract the remainder and divide by len_sw, to get the new x.
633         x = (x - remainder) / len_sw
634
635     # since the order of result is exactly the other way round, apply reverse():
636     result.reverse()
637
638     # return the results.
639     return result
640
641 def iso_ifs_to_counter(ifs):
642     """
643     maps a specific ifs to its associated counter (= inverse function of
644     iso_counter_to_ifs)
645
646     :param ifs: a specific ifs
647     :return:    counter, a number in [0, single_wordspace**N] which represents the ifs
648     """
649
650     # fix the length of the single word space:
651     len_sw = len(config.single_wordspace)
652
653     # the actual number is initialized with 0.
654     counter = 0
655
656     # go through all mappings and determine their number in single_wordspace.

```

```

656     for n in range(config.N):
657
658         if ifs[n] not in config.single_wordspace:
659             # generate a warning, if the mapping cannot be found:
660             warnings.warn("ifs " + str(ifs[n]) + " is not in single_wordspace" +
661                           str(config.single_wordspace))
661             counter += config.single_wordspace.index(ifs[n]) * (len_sw**(config.N - n - 1))
662
663     return counter

```

Listing A.5: The module `fractals.py` contains the definitions for each fractal.

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  """
5  @author:      Stefan Kohl, University of Stuttgart
6  @file name:   info_main.py
7  @description: main program to determine information about a fractal, which are
8                written in a compiled LaTeX-file.
9  """
10
11 # =====
12 # =                                     import modules                               =
13 # =====
14 from additional import information_to_fractal
15 from fractals import init_fractal
16
17
18 # =====
19 # =                                     run the program                               =
20 # =====
21
22 # initialize the fractal:
23 init_fractal("SG2")
24 # init_fractal("ST")
25 # init_fractal("SG3")
26 # init_fractal("SG4")
27 # init_fractal("SC")
28 # init_fractal("Vicsek")
29 # init_fractal("Pentagasket")
30 # init_fractal("PentagasketWITHOUTHOLE")
31 # init_fractal("Hexagasket")
32 # init_fractal("HexagasketW")
33
34 # some counter, which should be considered:
35 list_of_counter = [0,1,9]
36
37 if __name__ == "__main__":
38     #information_to_fractal(list_of_counter=list_of_counter, latex_mode="normal",
39                             mode="b")
40     information_to_fractal(list_of_counter=list_of_counter, latex_mode="silent",
41                             mode="b")

```

Listing A.6: The executable `info_main.py` determines information to a certain counter of a fractal and displays them for the user as a pdf.

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  """
5  @author:      Stefan Kohl, University of Stuttgart
6  @file name:   main.py
7  @description: main program.
8                Solves for a particular fractal the main task to a list or a list in a
                  file per multiprocessing.
9
10 """
11
12 # =====
13 # =                                     import modules                               =
14 # =====
15 from fractals import init_fractal
16 from B2 import ifs_mp
17
18
19 # =====
20 # =                                     run the program                               =
21 # =====
22
23 # initialize the fractal:
24 init_fractal("SG2")
25 # init_fractal("ST")
26 # init_fractal("SG3")
27 # init_fractal("SG4")
28 # init_fractal("SC")
29 # init_fractal("Vicsek")
30 # init_fractal("Pentagasket")
31 # init_fractal("PentagasketWITHOUTHOLE")
32 # init_fractal("Hexagasket")
33 # init_fractal("HexagasketW")
34
35 if __name__ == "__main__":
36     # either constructive mode:
37     # ifs_mp(mode="con")
38     # or existential mode:
39     ifs_mp(mode="ex")

```

Listing A.7: The executable `main.py` determines the number of free parameters, if preprocessing has been already executed.

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  ***
5  @author:      Stefan Kohl, University of Stuttgart
6  @file name:   main_complete.py
7  @description: main program.
8  Runs for a particular fractal the preprocessing, the main task and the
9  postprocessing.
10 ***
11
12 # =====
13 # =                                import modules                                =
14 # =====
15 from fractals import init_fractal
16 from B2 import preprocessing_mp_direct, ifs_mp, postprocessing
17
18
19 # =====
20 # =                                run the program                                =
21 # =====
22
23 # initialize the fractal:
24 init_fractal("SG2")
25 # init_fractal("ST")
26 # init_fractal("SG3")
27 # init_fractal("SG4")
28 # init_fractal("SC")
29 # init_fractal("Vicsek")
30 # init_fractal("Pentagasket")
31 # init_fractal("PentagasketWITHOUTHOLE")
32 # init_fractal("Hexagasket")
33 # init_fractal("HexagasketW")
34
35 if __name__ == "__main__":
36     # Preprocessing:
37     # -----
38     preprocessing_mp_direct()
39
40     # Main Task:
41     # -----
42     # either constructive mode:
43     # ifs_mp(mode="con")
44     # or existential mode:
45     ifs_mp(mode="ex")
46
47     # Postprocessing:
48     # -----
49     postprocessing()

```

Listing A.8: The executable `main_complete.py` does all computations (including preprocessing) for a certain fractal.

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  """
5  @author:      Stefan Kohl, University of Stuttgart
6  @file name:   postprocessing.py
7  @description: This is the main file for postprocessing!
8
9  """
10
11 # =====
12 # =                                     import modules                               =
13 # =====
14 from B2 import postprocessing
15 from additional import generate_tables
16
17
18 # =====
19 # =                                     run the program                               =
20 # =====
21
22 if __name__ == "__main__":
23     # run postprocessing for different fractals:
24     postprocessing("SG2")
25     postprocessing("SG3")
26     postprocessing("SG4")
27     postprocessing("Vicsek")
28     postprocessing("Pentagasket")
29     postprocessing("PentagasketWITHOUTHOLE")
30     postprocessing("Hexagasket")
31     postprocessing("HexagasketWITHOUTHOLE")
32     postprocessing("ST")
33     postprocessing("SC")
34
35     # generate two tables containing all information:
36     generate_tables()

```

Listing A.9: The executable `postprocessing.py` has to be run after the main program and generates the statistic with free parameters.

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  '''
5  @author:      Stefan Kohl, University of Stuttgart
6  @file name:   preprocessing.py
7  @description: main file for preprocessing to a certain fractal.
8
9  '''
10
11 # =====
12 # =                               import modules                               =
13 # =====
14 from B2 import preprocessing_mp_direct
15 from fractals import init_fractal
16
17
18 # =====
19 # =                               run the program                               =
20 # =====
21
22 # initialize the fractal:
23 init_fractal("SG2")
24 # init_fractal("ST")
25 # init_fractal("SG3")
26 # init_fractal("SG4")
27 # init_fractal("Vicsek")
28 # init_fractal("Pentagasket")
29 # init_fractal("PentagasketWITHOUTHOLE")
30 # init_fractal("Hexagasket")
31 # init_fractal("HexagasketW")
32
33 if __name__ == "__main__":
34     preprocessing_mp_direct()

```

Listing A.10: The executable `preprocessing.py` classifies the IFSs into multiple equivalence classes.





# Bibliography

- [Ada+03] B. Adams, S. A. Smith, R. S. Strichartz, and A. Teplyaev. The spectrum of the Laplacian on the pentagasket. In: *Fractals in Graz 2001*. Trends Math. Birkhäuser, Basel, 2003, pp. 1–24. MR: 2091699. Zbl: 1037.31010.
- [BK91] C. Bandt and K. Keller. Self-similar sets. II. A simple approach to the topological structure of fractals. In: *Math. Nachr.* 154 (1991), pp. 27–39. ISSN: 0025-584X. DOI: 10.1002/mana.19911540104. MR: 1138368. Zbl: 0824.28007. URL: <https://doi.org/10.1002/mana.19911540104>.
- [BB89] M. T. Barlow and R. F. Bass. The construction of Brownian motion on the Sierpiński carpet. In: *Ann. Inst. H. Poincaré Probab. Statist.* 25.3 (1989), pp. 225–257. ISSN: 0246-0203. MR: 1023950. Zbl: 0691.60070. URL: [http://www.numdam.org/item?id=AIHPB\\_1989\\_\\_25\\_3\\_225\\_0](http://www.numdam.org/item?id=AIHPB_1989__25_3_225_0).
- [BB99] M. T. Barlow and R. F. Bass. Brownian motion and harmonic analysis on Sierpinski carpets. In: *Canad. J. Math.* 51.4 (1999), pp. 673–744. ISSN: 0008-414X. DOI: 10.4153/CJM-1999-031-4. MR: 1701339. Zbl: 0945.60071. URL: <https://doi.org/10.4153/CJM-1999-031-4>.
- [Bar+10] M. T. Barlow, R. F. Bass, T. Kumagai, and A. Teplyaev. Uniqueness of Brownian motion on Sierpiński carpets. In: *J. Eur. Math. Soc. (JEMS)* 12.3 (2010), pp. 655–701. ISSN: 1435-9855. DOI: 10.4171/jems/211. MR: 2639315. Zbl: 1200.60070. URL: <https://doi.org/10.4171/jems/211>.
- [BP88] M. T. Barlow and E. A. Perkins. Brownian motion on the Sierpiński gasket. In: *Probab. Theory Related Fields* 79.4 (1988), pp. 543–623. ISSN: 0178-8051. DOI: 10.1007/BF00318785. MR: 966175. Zbl: 0635.60090. URL: <https://doi.org/10.1007/BF00318785>.

- [Bar93] M. F. Barnsley. Fractals everywhere. Second edition. Revised with the assistance of and with a foreword by Hawley Rising, III. Academic Press Professional, Boston, MA, 1993, pp. xiv+534. ISBN: 0-12-079061-0. MR: 1231795. Zbl: 0784.58002.
- [Doo59] J. L. Doob. Discrete potential theory and boundaries. In: *J. Math. Mech.* 8 (1959), 433–458, erratum 993. MR: 0107098. Zbl: 0101.11503.
- [Doo01] J. L. Doob. Classical potential theory and its probabilistic counterpart. Classics in Mathematics. Reprint of the 1984 edition. Springer-Verlag, Berlin, 2001, pp. xxvi+846. ISBN: 3-540-41206-9. DOI: 10.1007/978-3-642-56573-1. MR: 1814344. Zbl: 0990.31001. URL: <https://doi.org/10.1007/978-3-642-56573-1>.
- [DS99] M. Denker and H. Sato. Sierpiński gasket as a Martin boundary. II. The intrinsic metric. In: *Publ. Res. Inst. Math. Sci.* 35.5 (1999), pp. 769–794. ISSN: 0034-5318. DOI: 10.2977/prims/1195143423. MR: 1739300. Zbl: 0980.60095. URL: <https://doi.org/10.2977/prims/1195143423>.
- [DS01] M. Denker and H. Sato. Sierpiński gasket as a Martin boundary. I. Martin kernels. In: *Potential Anal.* 14.3 (2001), pp. 211–232. ISSN: 0926-2601. DOI: 10.1023/A:1011232724842. MR: 1822915. Zbl: 0980.60094. URL: <https://doi.org/10.1023/A:1011232724842>.
- [DS02] M. Denker and H. Sato. Reflections on harmonic analysis of the Sierpiński gasket. In: *Math. Nachr.* 241 (2002), pp. 32–55. ISSN: 0025-584X. DOI: 10.1002/1522-2616(200207)241:1<32::AID-MANA32>3.0.CO;2-5. MR: 1912376. Zbl: 1020.60066. URL: [https://doi.org/10.1002/1522-2616\(200207\)241:1%3C32::AID-MANA32%3E3.0.CO;2-5](https://doi.org/10.1002/1522-2616(200207)241:1%3C32::AID-MANA32%3E3.0.CO;2-5).
- [Dyn69] E. B. Dynkin. The boundary theory of Markov processes (discrete case). In: *Uspehi Mat. Nauk* 24.2 (146) (1969), pp. 3–42. ISSN: 0042-1316. MR: 0245096. Zbl: 0222.60048.
- [Fal90] K. Falconer. Fractal geometry. Mathematical foundations and applications. John Wiley & Sons, Ltd., Chichester, 1990, pp. xxii+288. ISBN: 0-471-92287-0. MR: 1102677. Zbl: 0689.28003.

- [Fal97] K. Falconer. Techniques in fractal geometry. John Wiley & Sons, Ltd., Chichester, 1997, pp. xviii+256. ISBN: 0-471-95724-0. MR: 1449135. Zbl: 0869.28003.
- [For17] O. Forster. Analysis. 2. Grundkurs Mathematik. Differentialrechnung im  $\mathbb{R}^n$ , gewöhnliche Differentialgleichungen., Eleventh edition. Vieweg + Teubner, Wiesbaden; Springer Spektrum, Wiesbaden, 2017, pp. viii+244. ISBN: 978-3-658-19411-6; 978-3-658-19410-9. DOI: 10.1007/978-3-658-19411-6. MR: 3980407. Zbl: 1372.26001. URL: <https://doi.org/10.1007/978-3-658-19411-6>.
- [FK19] U. Freiberg and S. Kohl. Martin boundary theory on inhomogenous fractals. In: arXiv e-prints (July 2019). arXiv: 1907.07499 [math.PR]. URL: <https://arxiv.org/abs/1907.07499>.
- [FS92] M. Fukushima and T. Shima. On a spectral analysis for the Sierpiński gasket. In: Potential Anal. 1.1 (1992), pp. 1–35. ISSN: 0926-2601. DOI: 10.1007/BF00249784. MR: 1245223. Zbl: 1081.31501. URL: <https://doi.org/10.1007/BF00249784>.
- [Gol87] S. Goldstein. Random walks and diffusions on fractals. In: Percolation theory and ergodic theory of infinite particle systems (Minneapolis, Minn., 1984–1985). Vol. 8. IMA Vol. Math. Appl. Springer, New York, 1987, pp. 121–129. DOI: 10.1007/978-1-4613-8734-3\_8. MR: 894545. Zbl: 0621.60073. URL: [https://doi.org/10.1007/978-1-4613-8734-3\\_8](https://doi.org/10.1007/978-1-4613-8734-3_8).
- [Ham00] B. M. Hambly. Heat kernels and spectral asymptotics for some random Sierpinski gaskets. In: Fractal geometry and stochastics, II (Greifswald/Koserow, 1998). Vol. 46. Progr. Probab. Birkhäuser, Basel, 2000, pp. 239–267. DOI: 10.1007/978-3-0348-8380-1\_12. MR: 1786351. Zbl: 0947.60086. URL: [https://doi.org/10.1007/978-3-0348-8380-1\\_12](https://doi.org/10.1007/978-3-0348-8380-1_12).
- [Hun60] G. A. Hunt. Markoff chains and Martin boundaries. In: Illinois J. Math. 4 (1960), pp. 313–340. ISSN: 0019-2082. MR: 123364. Zbl: 0094.32103. URL: <http://projecteuclid.org/euclid.ijm/1255456049>.

- [Hut81] J. E. Hutchinson. Fractals and self-similarity. In: *Indiana Univ. Math. J.* 30.5 (1981), pp. 713–747. ISSN: 0022-2518. DOI: 10.1512/iumj.1981.30.30055. MR: 625600. Zbl: 0598.28011. URL: <https://doi.org/10.1512/iumj.1981.30.30055>.
- [Ima00] A. Imai. Pentakun, the mod 5 Markov chain and a Martin boundary. English. Göttingen: Univ. Göttingen, Fakultät Mathematik, 2000, p. 53. Zbl: 0958.60068.
- [Ima02] A. Imai. The difference between letters and a Martin kernel of a modulo 5 Markov chain. In: *Adv. in Appl. Math.* 28.1 (2002), pp. 82–106. ISSN: 0196-8858. DOI: 10.1006/aama.2001.0768. MR: 1884389. Zbl: 1001.60077. URL: <https://doi.org/10.1006/aama.2001.0768>.
- [JLW12] H. Ju, K.-S. Lau, and X.-Y. Wang. Post-critically finite fractal and Martin boundary. In: *Trans. Amer. Math. Soc.* 364.1 (2012), pp. 103–118. ISSN: 0002-9947. DOI: 10.1090/S0002-9947-2011-05270-0. MR: 2833578. Zbl: 1244.28004. URL: <https://doi.org/10.1090/S0002-9947-2011-05270-0>.
- [KSK76] J. G. Kemeny, J. L. Snell, and A. W. Knapp. Denumerable Markov chains. Second edition. With a chapter on Markov random fields, by David Griffeath, Graduate Texts in Mathematics, No. 40. Springer-Verlag, New York-Heidelberg-Berlin, 1976, pp. xii+484. MR: 0407981. Zbl: 0348.60090.
- [KSS20] M. Kesseböhmer, T. Samuel, and K. Sender. The Sierpiński gasket as the Martin boundary of a non-isotropic Markov chain. In: *J. Fractal Geom.* 7.2 (2020), pp. 113–136. ISSN: 2308-1309. DOI: 10.4171/jfg/86. MR: 4101689. Zbl: 1452.31018. URL: <https://doi.org/10.4171/jfg/86>.
- [Kig89] J. Kigami. A harmonic calculus on the Sierpiński spaces. In: *Japan J. Appl. Math.* 6.2 (1989), pp. 259–290. ISSN: 0910-2043. DOI: 10.1007/BF03167882. MR: 1001286. Zbl: 0686.31003. URL: <https://doi.org/10.1007/BF03167882>.
- [Kig93] J. Kigami. Harmonic calculus on p.c.f. self-similar sets. In: *Trans. Amer. Math. Soc.* 335.2 (1993), pp. 721–755. ISSN: 0002-9947. DOI: 10.2307/2154402. MR: 1076617. Zbl: 0773.31009. URL: <https://doi.org/10.2307/2154402>.

- [Kig01] J. Kigami. Analysis on fractals. Vol. 143. Cambridge Tracts in Mathematics. Cambridge University Press, Cambridge, 2001, pp. viii+226. ISBN: 0-521-79321-1. DOI: 10.1017/CB09780511470943. MR: 1840042. Zbl: 0998.28004. URL: <https://doi.org/10.1017/CB09780511470943>.
- [KL93] J. Kigami and M. L. Lapidus. Weyl's problem for the spectral distribution of Laplacians on p.c.f. self-similar fractals. In: *Comm. Math. Phys.* 158.1 (1993), pp. 93–125. ISSN: 0010-3616. MR: 1243717. Zbl: 0806.35130. URL: <http://projecteuclid.org/euclid.cmp/1104254132>.
- [Koc04] H. von Koch. Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire. French. In: *Ark. Mat. Astron. Fys.* 1 (1904), pp. 681–702. ISSN: 0365-4133. Zbl: 35.0387.02.
- [Koh20] S. Kohl. Topological conditions on inhomogeneous fractals in Martin boundary theory and their algorithmic testing. In: *arXiv e-prints* (May 2020). arXiv: 2005.10534 [math.DS].
- [KP13] S. G. Krantz and H. R. Parks. The implicit function theorem. Modern Birkhäuser Classics. History, theory, and applications, Reprint of the 2003 edition. Birkhäuser/Springer, New York, 2013, pp. xiv+163. ISBN: 978-1-4614-5980-4; 978-1-4614-5981-1. DOI: 10.1007/978-1-4614-5981-1. MR: 2977424. Zbl: 1269.58003. URL: <https://doi.org/10.1007/978-1-4614-5981-1>.
- [Kus87] S. Kusuoka. A diffusion process on a fractal. In: *Probabilistic methods in mathematical physics* (Katata/Kyoto, 1985). Academic Press, Boston, MA, 1987, pp. 251–274. MR: 933827. Zbl: 0645.60081.
- [LN99] K.-S. Lau and S.-M. Ngai. Multifractal measures and a weak separation condition. In: *Adv. Math.* 141.1 (1999), pp. 45–96. ISSN: 0001-8708. DOI: 10.1006/aima.1998.1773. MR: 1667146. Zbl: 0929.28007. URL: <https://doi.org/10.1006/aima.1998.1773>.
- [LN12] K.-S. Lau and S.-M. Ngai. Martin boundary and exit space on the Sierpinski gasket. In: *Sci. China Math.* 55.3 (2012), pp. 475–494. ISSN: 1674-7283. DOI: 10.1007/s11425-011-4339-x. MR: 2891314. Zbl: 1241.31014. URL: <https://doi.org/10.1007/s11425-011-4339-x>.

- [LN14] K.-S. Lau and S.-M. Ngai. Boundary theory on the Hata tree. In: *Nonlinear Anal.* 95 (2014), pp. 292–307. ISSN: 0362-546X. DOI: 10.1016/j.na.2013.08.013. MR: 3130523. Zbl: 1282.31004. URL: <https://doi.org/10.1016/j.na.2013.08.013>.
- [LW15] K.-S. Lau and X.-Y. Wang. Denker-Sato type Markov chains on self-similar sets. In: *Math. Z.* 280.1-2 (2015), pp. 401–420. ISSN: 0025-5874. DOI: 10.1007/s00209-015-1430-y. MR: 3343913. Zbl: 1320.28008. URL: <https://doi.org/10.1007/s00209-015-1430-y>.
- [LW17] K.-S. Lau and X.-Y. Wang. On hyperbolic graphs induced by iterated function systems. In: *Adv. Math.* 313 (2017), pp. 357–378. ISSN: 0001-8708. DOI: 10.1016/j.aim.2017.04.012. MR: 3649228. Zbl: 1366.28008. URL: <https://doi.org/10.1016/j.aim.2017.04.012>.
- [Lin90] T. Lindstrøm. Brownian motion on nested fractals. In: *Mem. Amer. Math. Soc.* 83.420 (1990), pp. iv+128. ISSN: 0065-9266. DOI: 10.1090/memo/0420. MR: 988082. Zbl: 0688.60065. URL: <https://doi.org/10.1090/memo/0420>.
- [LS90] L. H. Loomis and S. Sternberg. Advanced calculus. Jones and Bartlett Publishers, Boston, MA, 1990, pp. xii+580. ISBN: 0-86720-122-3. MR: 1140004. Zbl: 0782.46041.
- [Man82] B. B. Mandelbrot. The fractal geometry of nature. Schriftenreihe für den Referenten. [Series for the Referee]. W. H. Freeman and Co., San Francisco, Calif., 1982, pp. v+460. ISBN: 0-7167-1186-9. MR: 665254. Zbl: 0504.28001.
- [Mar41] R. S. Martin. Minimal positive harmonic functions. In: *Trans. Amer. Math. Soc.* 49 (1941), pp. 137–172. ISSN: 0002-9947. DOI: 10.2307/1990054. MR: 3919. Zbl: 0025.33302. URL: <https://doi.org/10.2307/1990054>.
- [Meu+17] A. Meurer et al. SymPy: symbolic computing in Python. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: <https://doi.org/10.7717/peerj-cs.103>.
- [PM85] M. H. Protter and C. B. jun. Morrey. Intermediate calculus. 2nd ed. English. Springer New York, 1985, p. 665. ISBN: 978-1-4612-1086-3. DOI: 10.1007/978-

1-4612-1086-3. Zbl: 0555.26002. URL: <https://doi.org/10.1007/978-1-4612-1086-3>.

- [Pyt] Python Language Reference, version 3.6. Python Software Foundation. Available at <https://www.python.org/>; documentation at <https://docs.python.org/3.6/>. URL: <https://docs.python.org/3.6/>.
- [Saw97] S. A. Sawyer. Martin boundaries and random walks. In: Harmonic functions on trees and buildings (New York, 1995). Vol. 206. Contemp. Math. Amer. Math. Soc., Providence, RI, 1997, pp. 17–44. DOI: 10.1090/conm/206/02685. MR: 1463727. Zbl: 0891.60073. URL: <https://doi.org/10.1090/conm/206/02685>.
- [Str06] R. S. Strichartz. Differential equations on fractals. A tutorial. Princeton University Press, Princeton, NJ, 2006, pp. xvi+169. ISBN: 978-0-691-12731-6; 0-691-12731-X. MR: 2246975. Zbl: 1190.35001.
- [Tri97] H. Triebel. Fractals and spectra related to Fourier analysis and function spaces. Vol. 91. Monographs in Mathematics. Birkhäuser Verlag, Basel, 1997, pp. viii+271. ISBN: 3-7643-5776-2. DOI: 10.1007/978-3-0348-0034-1. MR: 1484417. Zbl: 0898.46030. URL: <https://doi.org/10.1007/978-3-0348-0034-1>.
- [Vic92] T. Vicsek. Fractal growth phenomena. Second edition. With a foreword by Benoit B. Mandelbrot. World Scientific Publishing Co., Inc., River Edge, NJ, 1992, pp. xx+488. ISBN: 981-02-0668-2; 981-02-0669-0. DOI: 10.1142/1407. MR: 1182314. Zbl: 0861.28006. URL: <https://doi.org/10.1142/1407>.
- [Woe00] W. Woess. Random walks on infinite graphs and groups. Vol. 138. Cambridge Tracts in Mathematics. Cambridge University Press, Cambridge, 2000, pp. xii+334. ISBN: 0-521-55292-3. DOI: 10.1017/CB09780511470967. MR: 1743100. Zbl: 0951.60002. URL: <https://doi.org/10.1017/CB09780511470967>.
- [Woe09] W. Woess. Denumerable Markov chains. EMS Textbooks in Mathematics. Generating functions, boundary theory, random walks on trees. European Mathematical Society (EMS), Zürich, 2009, pp. xviii+351. ISBN: 978-3-

03719-071-5. DOI: 10.4171/071. MR: 2548569. Zbl: 1219.60001. URL:  
<https://doi.org/10.4171/071>.





## Abstract

In this thesis we consider inhomogeneous fractals and represent them as Martin boundaries. The representation of fractals by Martin boundary theory is one of the important tools for analysis on fractals, but has been so far only considered for homogeneous fractals. In particular, we examine self-similar fractals generated by iterated function systems (IFSs), where we can introduce a mass function which generates a self-similar measure on the fractal.

In the first part we introduce a Markov chain on the word space, where the word space can be associated to the fractal. We adapt the transition probabilities of the Markov chain according to the mass function. As a consequence of this, the transition probabilities can get arbitrarily small, hence the results on homogeneous fractals can not be applied, and the Martin boundary theory on fractals has to be revised. We set up three conditions, whereas (B2) is the central condition. We are able to prove that the Martin boundary of the Markov chain is homeomorphic to the weighted fractal under these conditions.

In the second part we take a closer look at Condition (B2). We reverse the condition in some sense and examine, in which cases (B2) can be fulfilled. We elaborate that the fulfillment of (B2) mainly depends on the chosen iterated function system for a fixed fractal. Further we disprove that (B2) can be induced by a well-known property of fractals like “nested” or “p.c.f.”. We give a general example which shows that (B2) is satisfied with inhomogeneous mass function for IFSs with any cardinality.

The last part of the thesis is a more practical work: we develop a computer algorithm which determines for which inhomogeneous mass functions a fractal fulfills (B2). The algorithm computes the maximum number of weights that can be chosen. To this end we give two modes of the algorithm: a constructive way and existential way. The constructive way calculates exactly which weights can be freely chosen, whereas the existential algorithm calculates the exact number only. The benefit of the existential algorithm is a shorter calculating time. We apply the algorithm to some common fractals and observe that the number of free weights varies a lot. For the sake of completeness, the complete source code is appended.