

FP2zuMandelbrot

June 16, 2020

1 Sensibilität von f_r gegenüber den Startwerten

Zuerst gleisen wir wiederum die [logistische Funktion](#) auf und rekapitulieren das Wichtigste.

```
[1]: from numba import jit # just in time compiler

@jit
def fr(x,r):
    return r*x*(1-x)
```

```
[2]: @jit
def frorbit(x0=0.5, r=1.4, n=120):
    orbit = []
    orbit.append(x0)
    for k in range(n+1):
        orbit.append(fr(orbit[k],r))
    return orbit
```

1.1 Attraktorwerte von f_r

```
[3]: %matplotlib inline
#%matplotlib notebook
%config InlineBackend.figure_format = 'pdf'

import numpy as np
import matplotlib
matplotlib.rcParams['text.usetex'] = True
import matplotlib.pyplot as plt
from numpy import linspace

plt.style.use('seaborn')

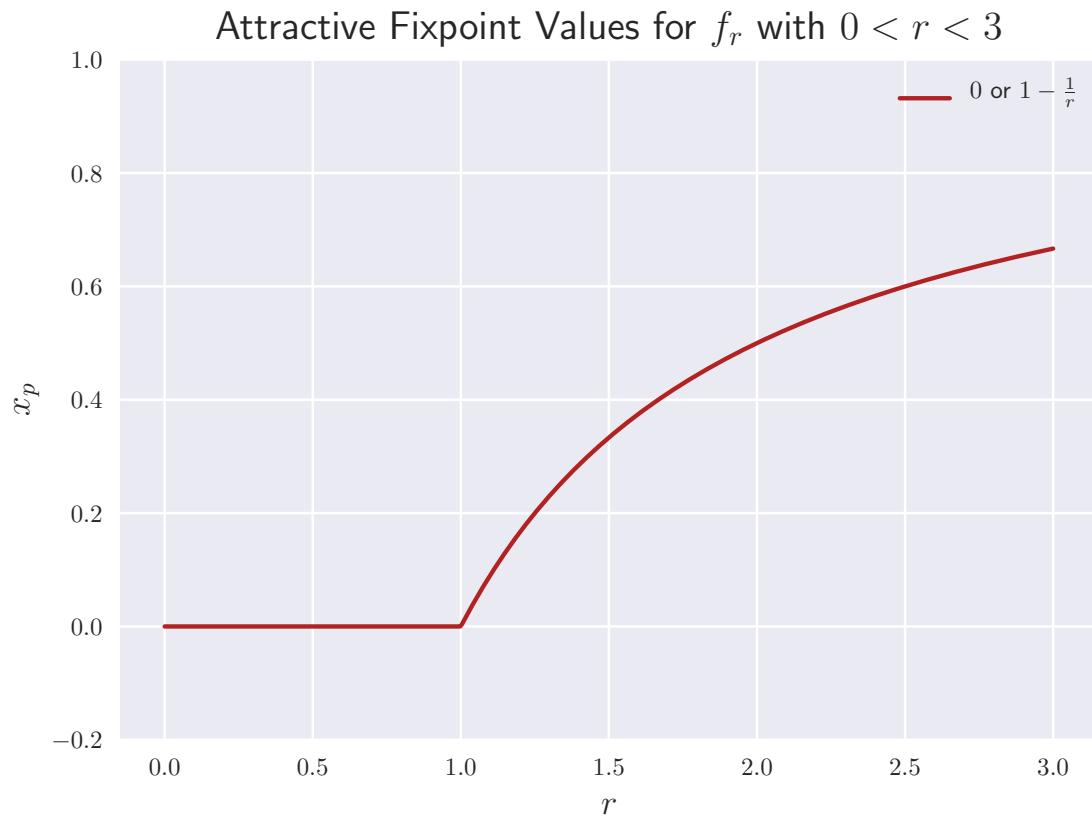
x1 = np.linspace(0, 3, 500)
y = np.piecewise(x1, [x1 < 1, x1 >= 1], [lambda x1: 0, lambda x1: 1-1/x1])

fig = plt.figure()
fig.set_size_inches(7, 5, forward=True)
```

```

plt.plot(xl,y, color='firebrick', label=r'$0$ or $1-\frac{1}{r}$')
plt.ylim(-0.2,1)
plt.title(r'Attractive Fixpoint Values for $f_r$ with $0 < r < 3$', fontsize=16)
plt.legend()
plt.xlabel(r'$r$', fontsize=14)
plt.ylabel(r'$x_p$', fontsize=14)
plt.show()

```



Wir haben also noch durch obigen Satz so was wie ein “sicheres” Attraktorintervall

$$I_r := \left(\frac{1}{2} - \frac{1}{2r}, \frac{1}{2} + \frac{1}{2r} \right)$$

erhalten. Den Attraktorbereich der Fixpunkte der Periode 2 kann man sich auf Wunsch noch mal in Youtube auf gym math zu Gemüte führen: [Attraktoren von \$f_r\$](#) .

[4] :

```

rvalue = 1.4
dummy = 1/(2*rvalue)
starting = 1/2 - dummy
ending = 1/2 + dummy

```

```
print(rf'I_n = ({starting},{ending}))')
fixpoints_ord2_range=1+6**0.5
```

```
I_n = ( 0.14285714285714285 , 0.8571428571428572 )
```

```
[5]: from sympy import *
from IPython.display import Math
display(Math("I_n = (" + latex(starting) + "," + latex(ending) + ")"))
display(Math(r"1+\sqrt{6} = " + latex(fixpoints_ord2_range)))
```

$I_n = (0.14285714285714285, 0.8571428571428572)$

$1 + \sqrt{6} = 3.449489742783178$

1.1.1 Was passiert für $r > 3$?

Klar ist, dass für $r > 3$ die beiden Fixpunkte nicht mehr attraktiv sind; sie sind aber natürlich nach wie vor “da”.

Wir kriegen also nebst $x_{p1} = 0$ und $x_{p2} = 1 - \frac{1}{r}$ nun zusätzlich

$$x_{p3,p4} = \frac{(r+1) \pm \sqrt{(r+1)(r-3)}}{2r}$$

und haben bereits gesehen, dass diese Fixpunkte für $3 < r < 3.45$ anziehend sind. Bemerkenswert ist nun, dass die weiteren Verzweigungen (**Bifurkationen**) in immer kürzer werdenden Abständen stattfinden. Und, dass diese verschiedenen oberen Grenzen der Attraktivität der Ordnung k , r_k , selbst einen Grenzwert haben. Es ist:

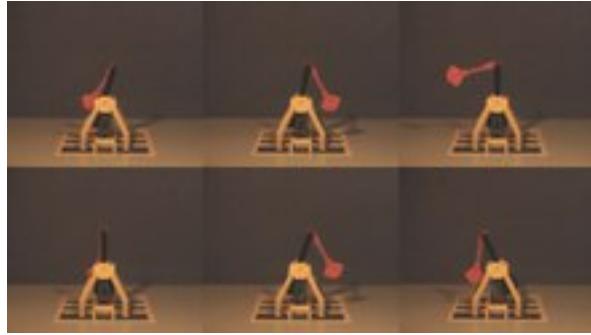
$$r_\infty := \lim_{k \rightarrow \infty} r_k = 3.569945672\dots$$

Für $r > r_\infty$ hat man chaotisches Verhalten. Aber selbst in diesen Bereichen gibt es immer wieder “Fenster” der Ordnung. Zudem kann der Verlauf eines Orbits für fixes r äußerst empfindlich gegenüber Anfangsbedingungen sein.

“Leicht” verschiedene Startwerte Im Folgenden soll die Empfindlichkeit des zeitlichen Verlaufs der logistischen Funktion f_r gegenüber nur minim unterschiedlichen Startwerten illustriert werden. Für gewisse r wird nämlich das Verhalten der Iteration so zu sagen “chaotisch”. Man spricht in diesem Kontext auch etwa vom **Schmetterlingseffekt**. Natürlich findet man diesen Effekt nicht für alle $0 < r < 4$, wie wir bereits wissen.

Übung : Google “Schmetterlingseffekt”

Dabei stösst man auch auf dynamische Probleme, welche analytisch anspruchsvoll sind: zum Beispiel das **Doppelpendel**



Betrachte Orbits für $r = 4$:

```
[6]: orbit1 = frorbit(0.1, 4.0, 80)
orbit2 = frorbit(0.1-10**(-15), 4.0, 80)
orbit3 = frorbit(0.1, 4.0, 80)
orbit4 = frorbit(0.1+10**(-17), 4.0, 80)

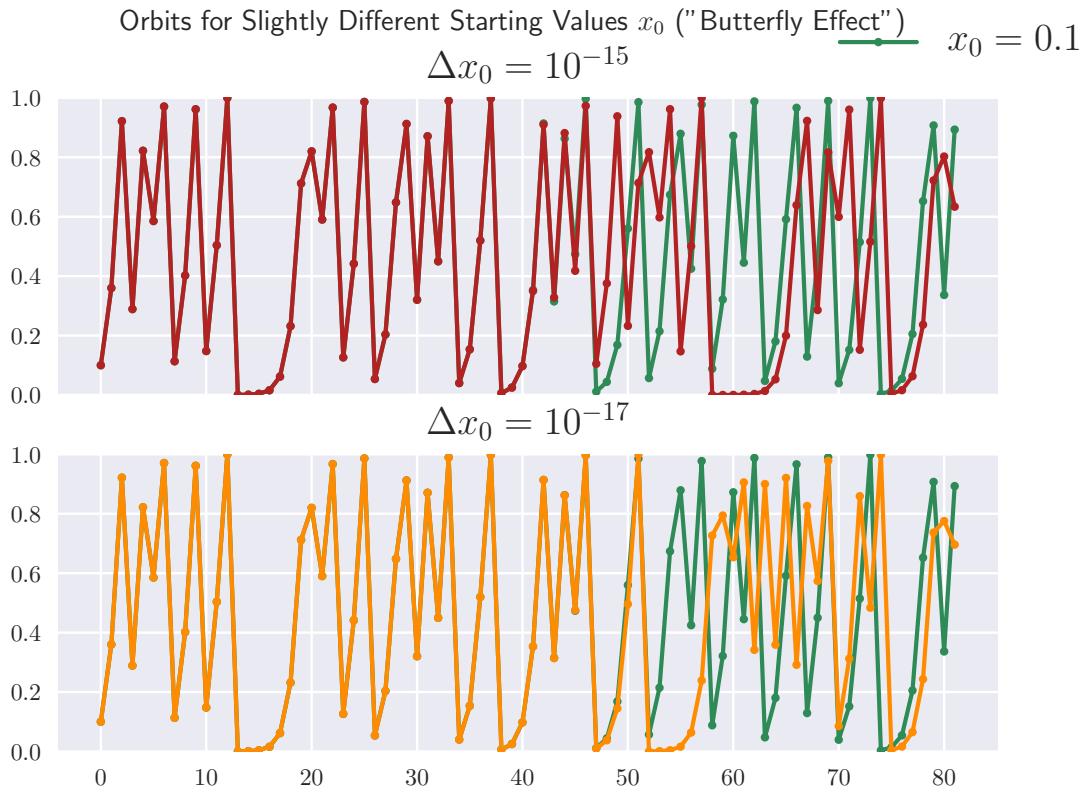
fig, axs = plt.subplots(2)
fig.set_size_inches(7, 5, forward=True)

axs[0].plot(orbit1, marker='.', color='seagreen', label=r'$x_0=0.1$')
axs[0].plot(orbit2, marker='.', color='firebrick')
axs[0].set_title(r'$\Delta x_0=10^{-15}$', fontsize=16)
axs[1].plot(orbit3, marker='.', color='seagreen')
axs[1].plot(orbit4, marker='.', color='darkorange')
axs[1].set_title(r'$\Delta x_0=10^{-17}$', fontsize=16)

fig.suptitle(r'Orbits for Slightly Different Starting Values $x_0$ ("Butterfly Effect")', fontsize=12)

axs[0].set_ylim([0, 1])
axs[1].set_ylim([0, 1])
fig.legend(fontsize=16)

for ax in axs.flat:
    ax.label_outer()
```



```
[7]: from matplotlib import rc
from numba import jit # just in time compiler

#@jit
def plot_cobweb(f, r, x0, nmax=40):

    # Plotte  $y = f_r(x)$  und  $y = x$  für  $0 \leq x \leq 1$ , illustriere für den Startwert  $x_0$ 
    # die Iteration  $x_{k+1} = f_r(x_k)$ .

    x = np.linspace(0, 1, 500)
    fig = plt.figure()
    fig.set_size_inches(7, 7, forward=True)
    fig.suptitle('Graphische Iteration der Logistischen Funktion', fontsize=20)
    ax = fig.add_subplot(111)

    # Plot  $y = f(x)$  and  $y = x$ 
    ax.plot(x, f(x, r), color="#444444", linewidth=2)
    ax.plot(x, x, color="#444444", linestyle='--', linewidth=1.5)

    # Iterate  $x = f_r(x)$  für  $nmax$  steps mit Startwert  $x_0$ .
    px, py = np.empty((2,nmax+1,2))
```

```

px[0], py[0] = x0, 0
for n in range(1, nmax, 2):
    px[n] = px[n-1]
    py[n] = f(px[n-1], r)
    px[n+1] = py[n]
    py[n+1] = py[n]

# Pfad des Orbits plotten
ax.plot(px, py, color='seagreen', linewidth=0.9, alpha=0.7)
ax.plot(x0,x0, color='lime', marker='o', zorder=5)
ax.plot(px[nmax],px[nmax], 'ro', zorder=5)

# Beschriften
ax.minorticks_on()
ax.grid(which='minor', alpha=0.5)
ax.grid(which='major', alpha=0.8)
#ax.set_aspect('equal')
ax.set_xlabel(r'$x$', fontsize=16)
ax.set_ylabel(r'$f_r(x)$', fontsize=16)
ax.set_title(r'$x_0 = {:.1}, r = {:.2}$'.format(x0, r), fontsize=20)

# plt.savefig('cobweb_{:.1}_{:.2}.png'.format(x0, r))

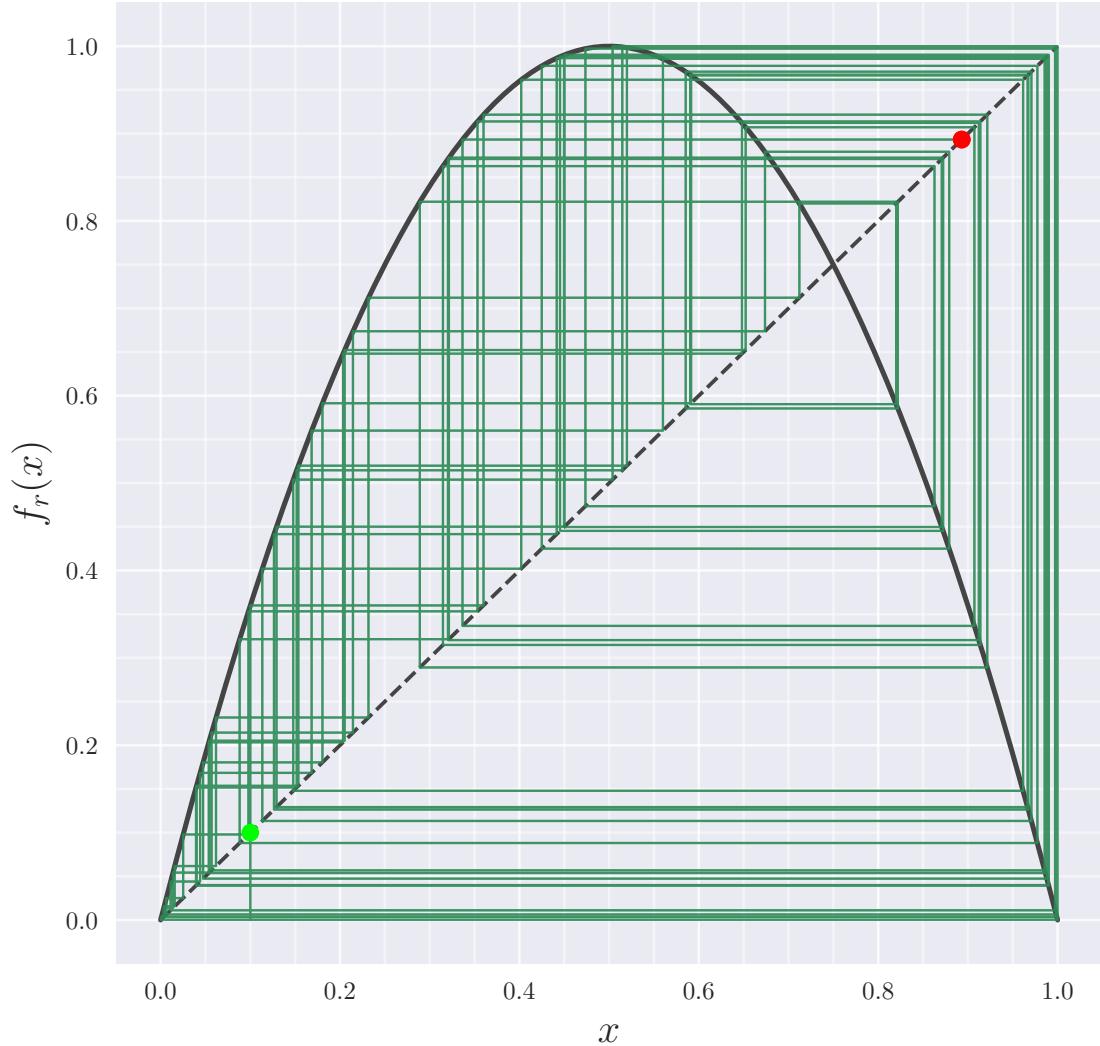
# Logistische Funktion  $f_r(x) = rx(1-x)$ .
func = (lambda x,r: r*x*(1-x))

plot_cobweb(func, 4.0, 0.1, 162)

```

Graphische Iteration der Logistischen Funktion

$$x_0 = 0.1, r = 4.0$$



1.2 Das Feigenbaum-Diagramm

Jetzt möchte ich noch einen Vergleich zwischen den Werten von r und den Fixpunktwerten illustrieren. Wers schon kennt findet das passende Youtube-Video [Feigenbaumdiagramm auf gym math](#). Setzen wir den Plot von oben fort und suchen Fixpunkte höherer Ordnung, so ergibt sich folgendes Bild, das **Feigenbaum-Diagramm** genannt wird:

```
[8]: %load_ext cython
```

```
[9]: %%cython

import numpy as np
cimport numpy as np
cimport cython

@cython.boundscheck(False) # turn off bounds-checking for entire function
@cython.wraparound(False) # turn off negative index wrapping for entire function
def bifurcation(np.int64_t precision=1000, np.int64_t keep=500, np.int64_t num_compute=10000,
                np.float64_t xmin=0, np.float64_t xmax=4, np.float64_t ymin=0, np.float64_t ymax=1):
    """ Acquire bifurcation points for varying mu for logistic map """
    cdef np.ndarray[np.float64_t, ndim=1] mu = np.linspace(xmin, xmax, precision, dtype=np.float64)
    cdef np.float64_t x = 0.5
    cdef np.int64_t i, j, k
    cdef np.ndarray[np.float64_t, ndim=2] points = np.zeros((len(mu) * keep, 2), dtype=np.float64)
    k = 0
    for i in range(len(mu)):
        for j in range(num_compute):
            x = mu[i] * x * (1 - x)
            if j > (num_compute - keep): # we throw away the transient
                points[k, 0] = mu[i]
                points[k, 1] = x
                k += 1
    return points
```

```
[10]: %%cython

import numpy as np
cimport numpy as np

cdef f(np.float64_t mu, np.float64_t x, int n):
    cdef int i
    cdef np.float64_t x0 = x
    for i in range(n):
        x0 = mu * x0 * (1 - x0)
    return x0

def cobweb(np.float64_t mu, int n=1, int num=100, int keep=100, np.float64_t initial = 0.5):
    """ Generate the path for a cobweb diagram """
    cdef np.ndarray[np.float64_t, ndim=2] web = np.zeros((keep, 2))
    cdef np.float64_t x = initial
    cdef np.float64_t y = initial
```

```

cdef int offset = num - keep
cdef int state = 1
if num == keep:
    offset = num - keep + 1
for i in range(1, num):
    if state:
        y = f(mu, x, n)
    else:
        x = y
    state ^= 1
    if i >= offset:
        web[i - offset, 0] = x
        web[i - offset, 1] = y
return web

```

```
[11]: import matplotlib.patches as mpatches

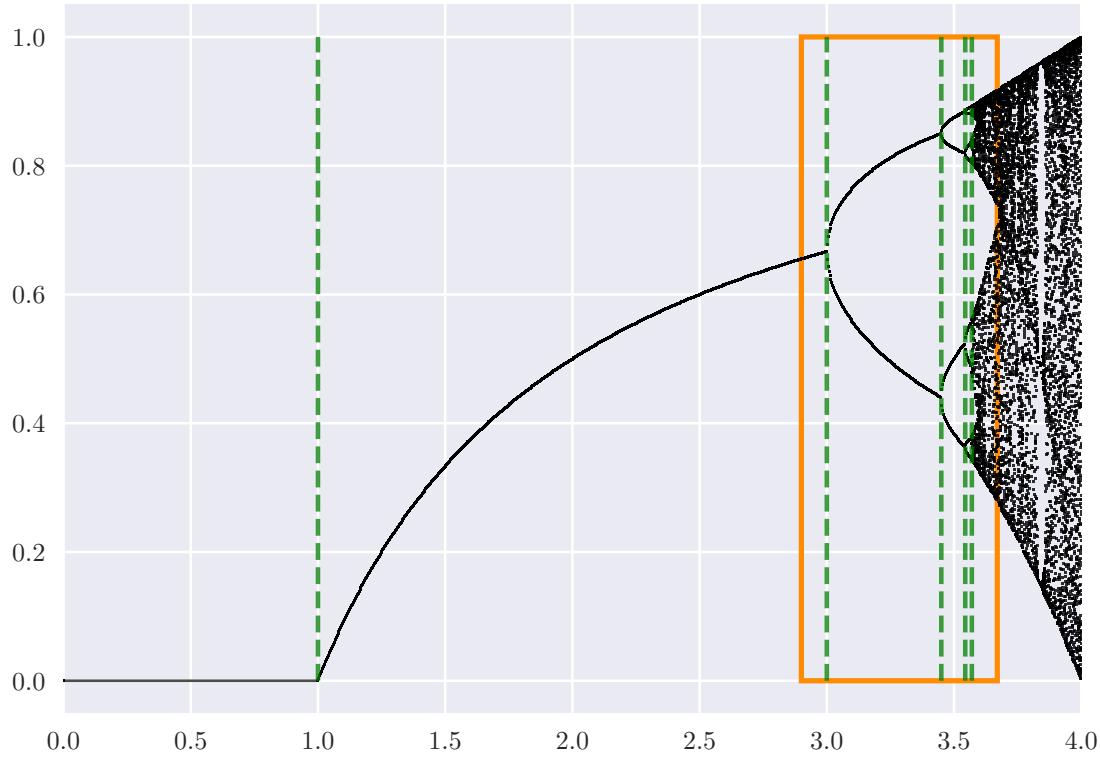
points = bifurcation(xmin=1, xmax=4, precision=800, num_compute=20000, keep=80)
plt.figure(figsize=(7, 5))
plt.plot(points[:, 0], points[:, 1], ',', color='k', alpha=0.8)
plt.plot([0,1],[0,0], color='k', alpha=0.7, linewidth=1)

rectangle = plt.Rectangle((2.9,0.0),3.67-2.9,1.
                           ↪0,linewidth=2,edgecolor='darkorange',facecolor='none')
plt.gca().add_patch(rectangle)

mu_vals = np.array([1, 3, 3.45, 3.544, 3.569945672,])

for mu in mu_vals:
    plt.plot(np.ones(5) * mu, np.linspace(0, 1, 5), color='green', ↪
             ↪linestyle='dashed', alpha=0.75)
plt.xlim(0, 4)
plt.show()

```



Ihr seht hier den kompletten Bereich $0 < r < 4$. Grün habe ich jeweils markante Änderungen im Fixpunktverhalten hervorgehoben. Oben sind das der Übergang vom Attraktor 0 zum Attraktor $1 - \frac{1}{r}$ bei $r = 1$, dann zum 2-Zyklus bei $r = 3$, zum 4-Zyklus bei $r = 1 + \sqrt{6}$ und etwas später zum 8-Zyklus.

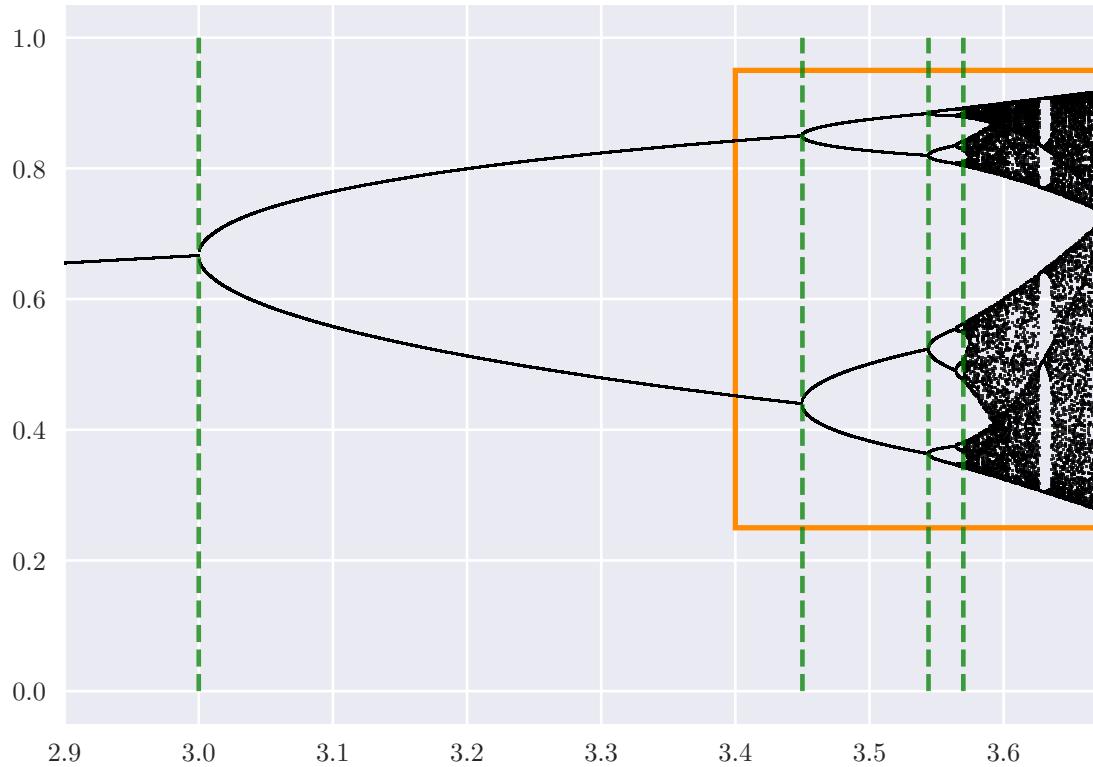
Ferner, falls vom einem zum nächsten Plot ein “Zoom” erfolgt, so hab ich das Fenster in Orange angedeutet. Natürlich kann man sich auch an der Skala orientieren.

```
[12]: points = bifurcation(xmin=1, xmax=4, precision=8000, num_compute=10000, keep=50)
plt.figure(figsize=(7, 5))
plt.plot(points[:, 0], points[:, 1], '.', color='k', alpha=0.8)

mu_vals = np.array([1, 3, 3.45, 3.544, 3.569945672,])

rectangle = plt.Rectangle((3.4, 0.25), 3.67-3.4, 0.95-0.
                           ↵25, linewidth=2, edgecolor='darkorange', facecolor='none')
plt.gca().add_patch(rectangle)

for mu in mu_vals:
    plt.plot(np.ones(5) * mu, np.linspace(0, 1, 5), color='green', ↵
                           ↵linestyle='dashed', alpha=0.75)
plt.xlim(2.9, 3.67)
plt.show()
```

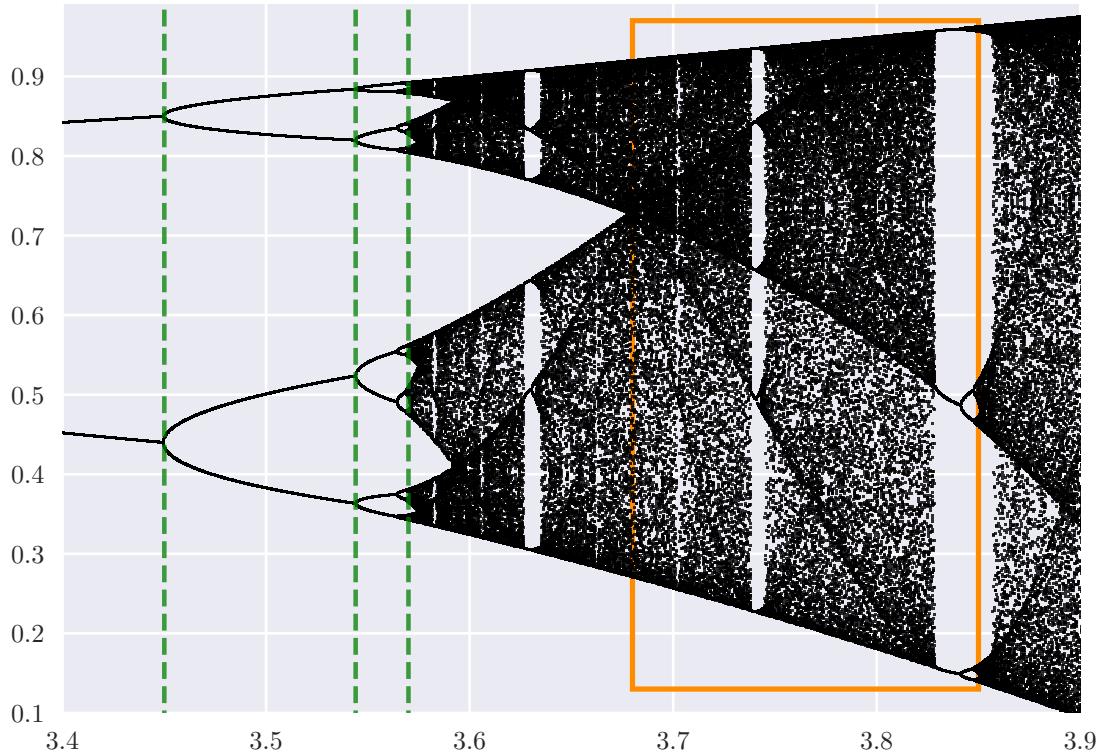


```
[13]: points = bifurcation(xmin=1, xmax=4, precision=10000, num_compute=10000, ↴
    ↪keep=100)
plt.figure(figsize=(7, 5))
plt.plot(points[:, 0], points[:, 1], '.', color='k', alpha=0.8)

mu_vals = np.array([1, 3, 3.45, 3.544, 3.569945672,])

rectangle = plt.Rectangle((3.68, 0.13), 3.85 - 3.68, 0.97 - 0. ↴
    ↪13, linewidth=2, edgecolor='darkorange', facecolor='none')
plt.gca().add_patch(rectangle)

for mu in mu_vals:
    plt.plot(np.ones(5) * mu, np.linspace(0, 1, 5), color='green', ↴
        ↪linestyle='dashed', alpha=0.75)
plt.xlim(3.4, 3.9)
plt.ylim(0.1, 0.99)
plt.show()
```



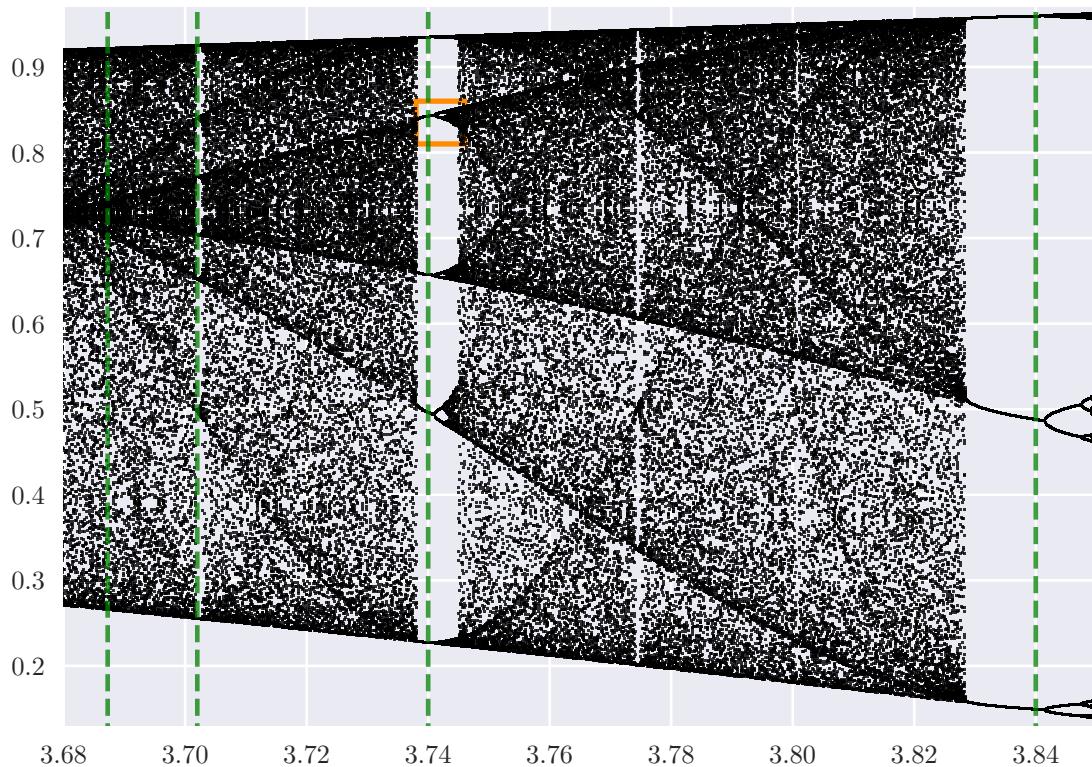
Jetzt setze ich die grünen Marker auf Fenster, in denen man 3-, 5-, 7- und 9-Zyklen beobachten kann.

```
[14]: points = bifurcation(xmin=1, xmax=4, precision=20000, num_compute=20000, keep=100)
plt.figure(figsize=(7, 5))
plt.plot(points[:, 0], points[:, 1], ',', color='k', alpha=0.8)

rectangle = plt.Rectangle((3.738,0.81),3.746-3.738,0.86-0.81,linewidth=2,edgecolor='darkorange',facecolor='none')
plt.gca().add_patch(rectangle)

mu_vals = np.array([3.84, 3.74, 3.702, 3.68725])

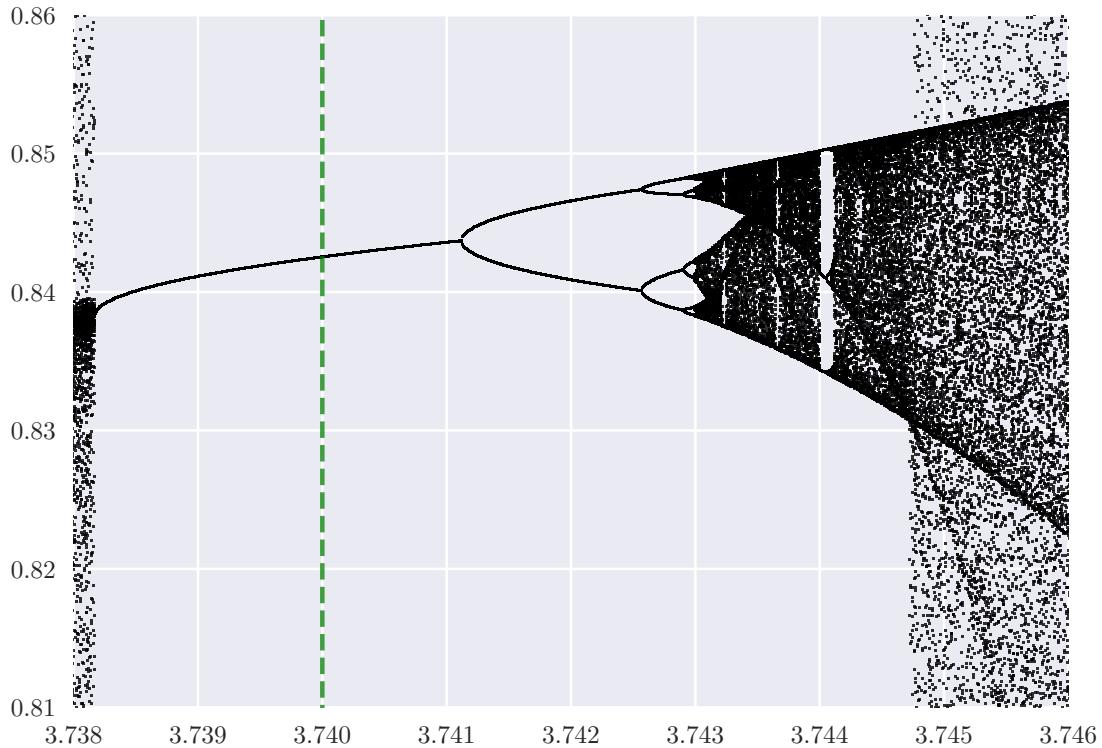
for mu in mu_vals:
    plt.plot(np.ones(5) * mu, np.linspace(0, 1, 5), color='green', linestyle='dashed', alpha=0.75)
plt.xlim(3.68, 3.85)
plt.ylim(0.13, 0.97)
plt.show()
```



```
[15]: points = bifurcation(xmin=1, xmax=4, precision=400000, num_compute=20000,
                           ↪keep=400)
plt.figure(figsize=(7, 5))
plt.plot(points[:, 0], points[:, 1], '.', color='k', alpha=0.8)

mu_vals = np.array([3.84, 3.74, 3.702, 3.68725])

for mu in mu_vals:
    plt.plot(np.ones(5) * mu, np.linspace(0, 1, 5), color='green',
             ↪linestyle='dashed', alpha=0.75)
plt.xlim(3.738, 3.746)
plt.ylim(0.81, 0.86)
plt.show()
```



Phänomenal! Da haben wir eine kleine Kopie — zumindest siehts so aus — des “grossen” Feigenbaum-Diagramms! Man spricht in diesem Zusammenhang etwa auch von **Selbstähnlichkeit**. Dazu mehr im nächsten Kapitel Section 2.

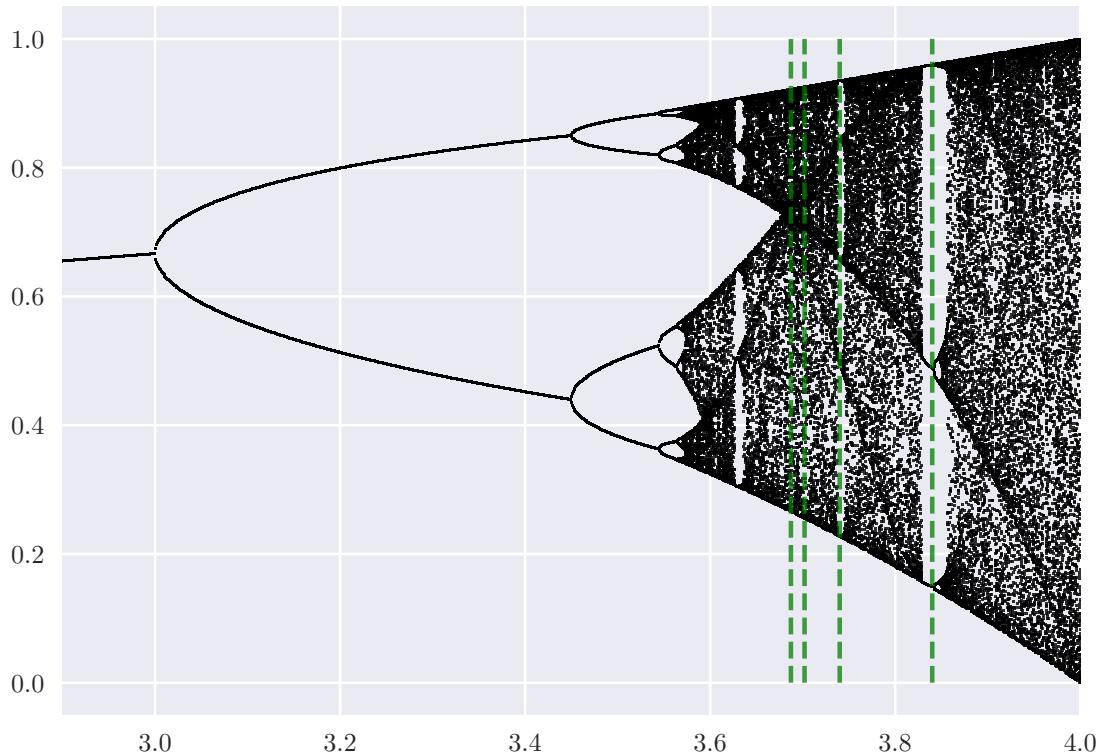
1.2.1 Ungerade Zyklen

Zu Beginn hatte man den Eindruck, es würden nur Zyklen der Ordnung 2^k existieren. Es gibt aber wie oben gesehen auch ungerade Zyklen, zum Beispiel der Ordnung 3,5,7 oder 9. Die sind im Folgenden im Überblick mit einem Cobweb Diagramm illustriert.

```
[16]: points = bifurcation(xmin=1, xmax=4, precision=3000, num_compute=20000, ↴
    ↴keep=100)
plt.figure(figsize=(7, 5))
plt.plot(points[:, 0], points[:, 1], 'r.', color='red', alpha=0.8)

mu_vals = np.array([3.84, 3.74, 3.702, 3.68725])

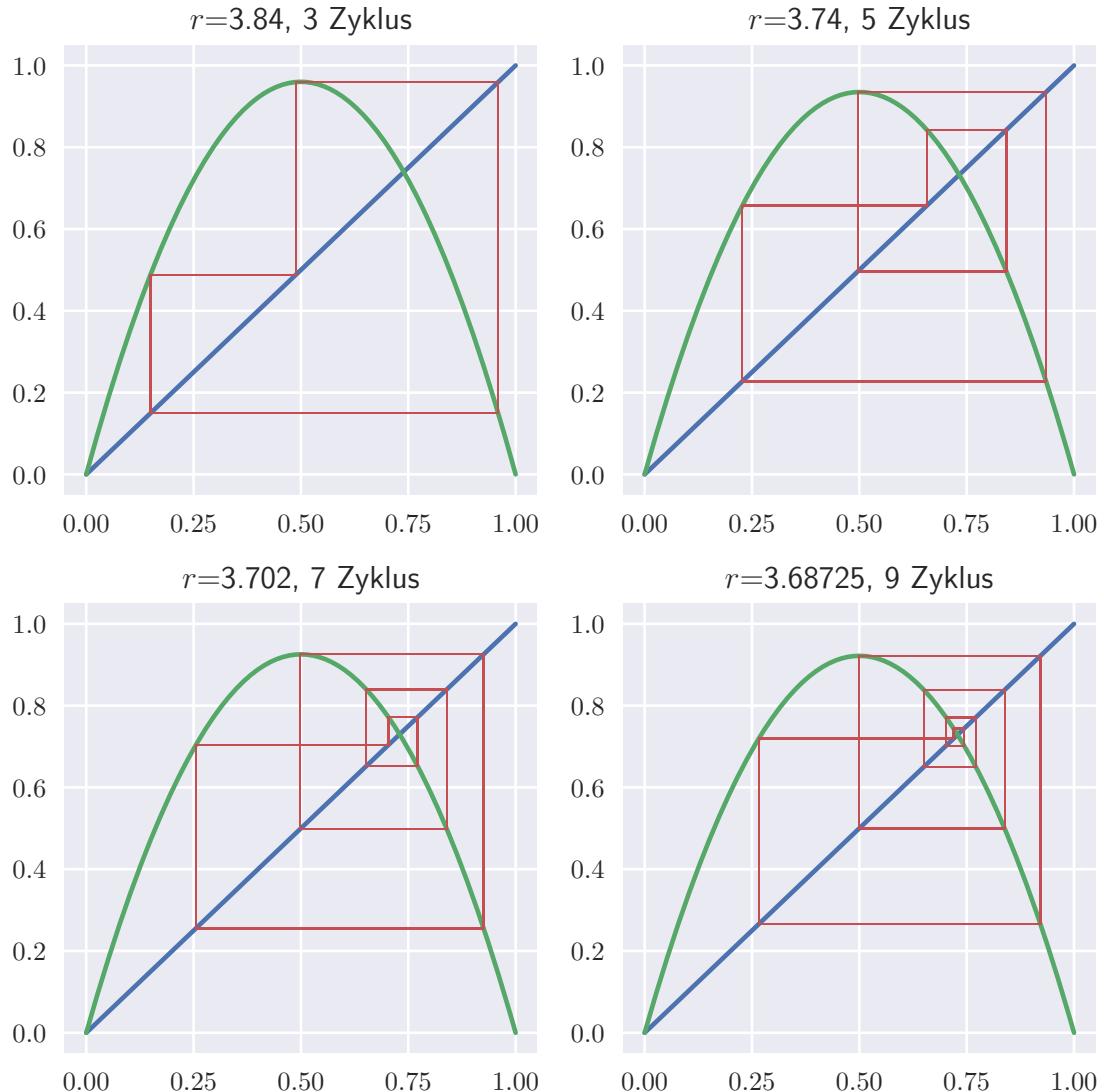
for mu in mu_vals:
    plt.plot(np.ones(4) * mu, np.linspace(0, 1, 4), color='green', ↴
    ↴linestyle='dashed', alpha=0.75)
plt.xlim(2.9, 4)
plt.show()
```



```
[17]: x = np.linspace(0, 1, 5000)
n = 1

def f(x, mu, n):
    x1 = x
    for i in range(n):
        x1 = mu * x1 * (1 - x1)
    return x1

fig, axarr = plt.subplots(2, 2, figsize=(6, 6))
for index, i, j in [(i, int(i / 2), i % 2) for i in range(4)]:
    axarr[i, j].plot(x, x)
    axarr[i, j].plot(x, f(x, mu_vals[index], n))
    web = cobweb(mu_vals[index], n=n, num=5000, keep=1000, initial=0.8)
    axarr[i, j].plot(web[:, 0], web[:, 1], linewidth=0.5)
    axarr[i, j].set_title(r'$r=${}, {} Zyklus'.format(mu_vals[index], (index + 1) * 2 + 1))
plt.tight_layout()
plt.show()
```



2 Fraktale und die Mandelbrotmenge

Ein **Fraktal** ist ...

Aus Wikipedia:

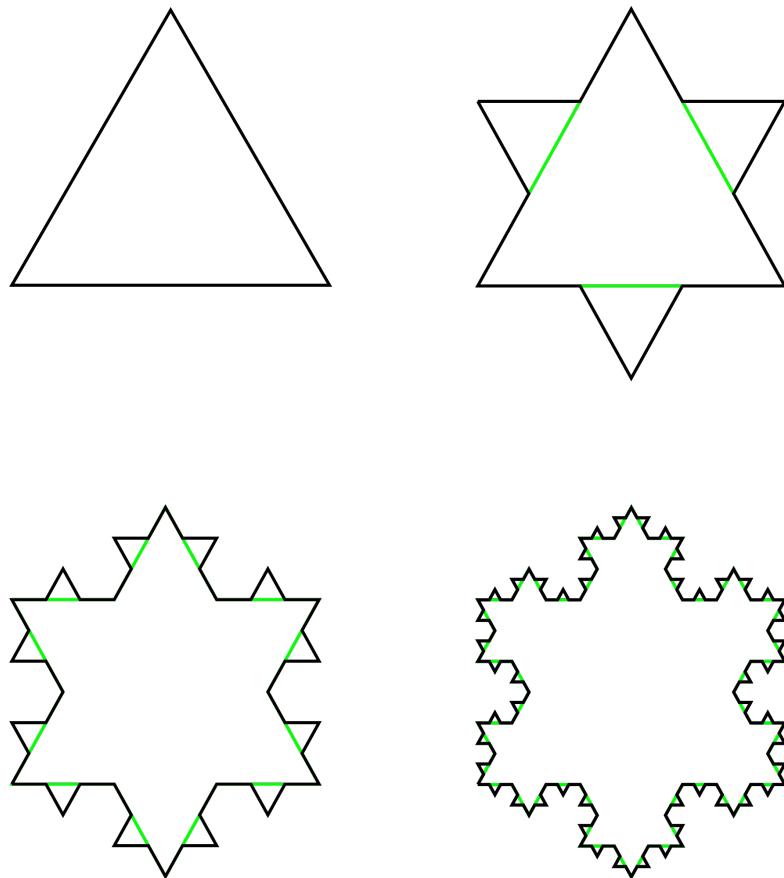
Fraktal ist ein vom Mathematiker Benoît Mandelbrot 1975 geprägter Begriff (lateinisch *fractus* “gebrochen”, von lateinisch *frangere*, (in Stücke zer-)brechen), der bestimmte natürliche oder künstliche Gebilde oder geometrische Muster bezeichnet.

Diese Gebilde oder Muster besitzen im Allgemeinen keine ganzzahlige Hausdorff-Dimension, sondern eine gebrochene – daher der Name – und weisen zudem einen hohen Grad von Skaleninvarianz bzw. Selbstähnlichkeit auf. Das ist beispielsweise der Fall, wenn ein Objekt aus mehreren verkleinerten Kopien seiner selbst besteht. Geometrische

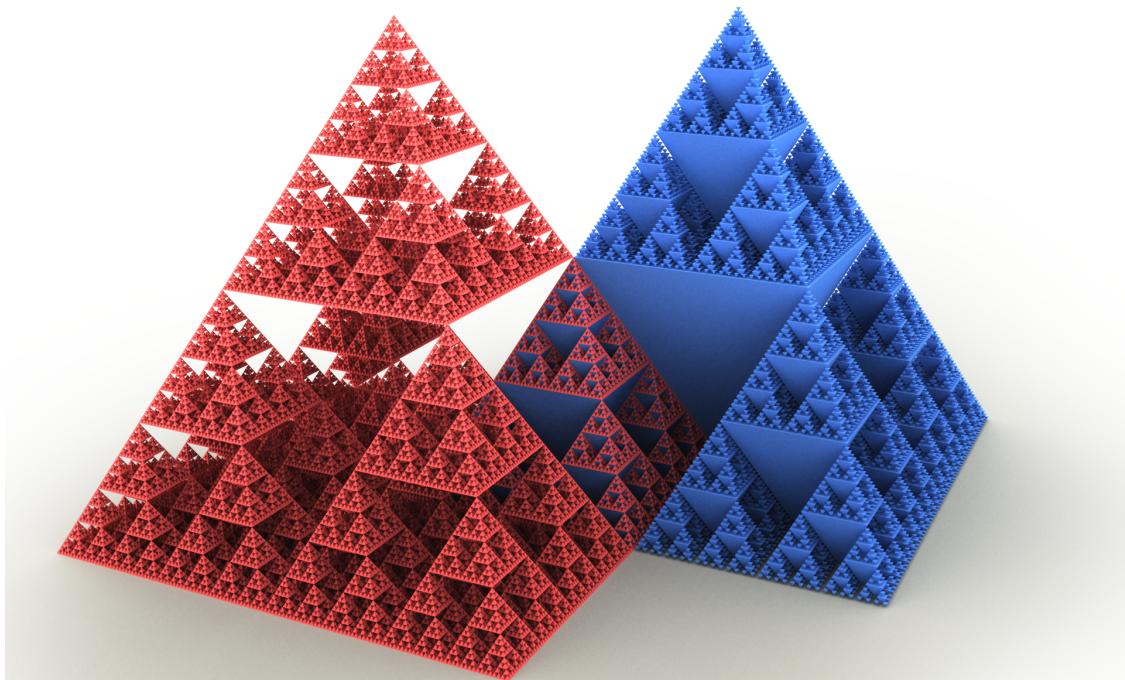
Objekte dieser Art unterscheiden sich in wesentlichen Aspekten von gewöhnlichen glatten Figuren.

Zuerst möchte ich einige Bilder zeigen, bevor wir uns dann sehnlichst auf die Mandelbrotmenge stürzen.

2.1 Die Koch'sche Schneeflocke



2.2 Die Sierpinski Pyramide



2.3 Romanesco



2.4 Farn



2.5 Mandelbrot Menge

