Chapter 2

Two-dimensional structures from algorithms

The roles of order and chance: the example of 2d-fractals

In the first chapter we presented some introductory considerations on *information* in comparison with the Aristotelian-Thomistic notion of *form*. A special attention has been devoted to the notion of *algorithmic information* viewed as a suitable candidate to approach the logical/ontological notions of *definition/essence* regarding the *structure* of an entity and respectively its *nature* on the side of its *dynamics* according to which it can operate and especially the dynamics generating the entity itself.[1]

Two simple didactical examples of fractal generation were enough to show how a suitable assigned (mathematical or physical) *law* can hide the capability of *defining/constructing* an entire complex entity, which exhibits a precise ordered structure like a *Julia set* and a fractal *basin of attraction* of a *magnetic pendulum*.

In the present chapter we will examine in more deepness how *order* and *chance* may enter into the *definition/structure* and the *dynamics/nature* of some kinds of *organized entities*.

2.1 Ordered entity structures generated by ordered processes

2.1.1 Analytic geometry

As a first trivial class of ordered entity structures built by a very simple mathematical law we can consider the entire environment of Cartesian *analytic geometry*.

[1] We remember that, roughly speaking, by Aristoelian-Thomistic notion of *form* we mean a non-material principle organizing the structure of an entity, while by *nature* we mean the same principle as it is able to determine the *dynamics* of the same entity.

In fact *algebraic equations* involving two or respectively three variables (co-ordinates) as:

$$f(x,y)=0, \qquad g(x,y,z)=0, \tag{2.1}$$

or *algebraic inequalities* as:

$$f(x,y)\leq 0, \qquad g(x,y,z)\leq 0, \tag{2.2}$$

are enough to determine univocally a set of points in a Cartesian plane or, respectively, $3D$ space,[2] f,g being functions of the co-ordinates x,y or x,y,z of each point belonging to the set.[3] Equations identify *paths* on Cartesian plane or respectively surfaces[4] in space, while inequalities determine a region of points aside the paths (internal or external if the path and the surface are closed), of the same dimension as the plane or respectively the space.

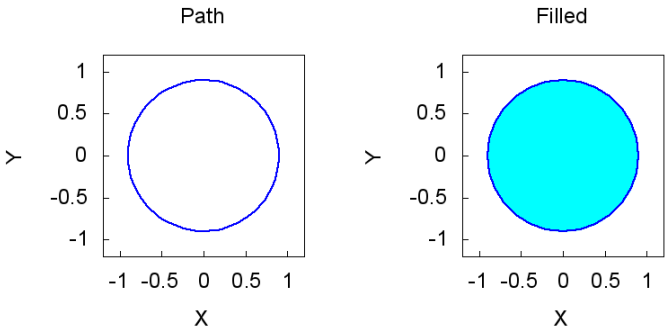

Fig.1 - a) *Path set* defined by the Cartesian equation $x^2+y^2=R^2, R=0.9$;
b) *Filled set* defined by the inequality $x^2+y^2\leq R^2, R=0.9$.

Such structures are essentially simple, *i.e.,* non-complex, since they do not exhibit self-similarity properties, being built by a non-iterative procedure. In this sense we have qualified them as a trivial class of sets. In effect they represent at most a sort of first level of idealized approximation to real bodies.

The interest of even apparently simple structures arises, as it has been pointed out by René Thom, if we consider their boundaries as *singularities* emerging within a continuous body represented by the whole plane, or respectively the whole space[5]. In fact each boundary manifold of Cartesian equation $f(x_i)=0, i=1,2,\cdots,n-1$ (where n is the space dimensionality), represents, for physical bodies, a front across which some physical quantity, as *e.g.,*

[2] More generally also in hyperspace of any dimension.
[3] Of course inequalities like $f(x,y)\geq 0, g(x,y,z)\geq 0$ can be reduced to the form (2.2) multiplying each member by -1.
[4] More generally manifolds in hyperspace of any dimension.
[5] Or hyperspace if we consider higher dimensional abstract spaces.

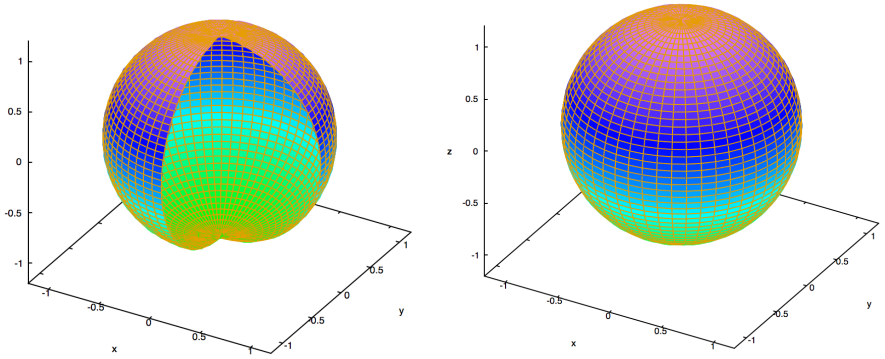

Fig.2 - a) Section of a *surface set* defined by the Cartesian equation $x^2 + y^2 + z^2 = R^2, R = 1.0$;
b) *Filled set (body)* defined by the inequality $x^2 + y^2 + z^2 \leq R^2, R = 1.0$.

mass density, becomes discontinuous so that some body may be retailed and distinguished by the other ones, determining its geometrical *form*.[6]

2.1.2 Two-dimensional fractals in a plane

A more relevant class of structures is provided by *fractal structures,* which can be obtained, generally, by an "adequate" number[7] of iterations of some mathematical law. A typical property of fractals is *self-similarity* (exact or at least statistical). In fact they exhibit repeated geometrical shapes at any scale they may be examined. So it results impossible to decompose them into simpler elementary parts characterized by a lower level of complexity, *i.e.,* to reduce their *fractal dimension*[8] analyzing them at any deeper (in principle even infinitesimal) scale level. A physical limit is imposed only by the computational power of our machines.

Here we are not interested in examining in detail fractals and their properties, but we are interested in emphasizing that generally many of them are generated by procedures that apply a mathematical algorithm (*law*) following an *ordered* sequence of operations, so that an *ordered* (*self-similar*) fractal structure of the resulting entity arises. In this sense we can say that "*order* generates *order*". It is not surprising to obtain *order* from *order*; more surprise

[6]Here the word *form* means mainly *shape* and the related *information* law from which that shape is obtained. The problem has been examined in mathematical detail, with an effort to establish comparison to the Aristotelian notion of *space* and *form* by R.Thom, in [17].

[7]Where "adequate" means, in principle "infinite", and in practice "sufficiently great" in order that self-similarity appears according to the desired detail level.

[8]We remember that fractal dimension is a measure of the fraction of straight line filled by a set of points, or the fraction of plane filled by a curve (increased by one), or the fraction of space filled by a shape, increased by two, and so on. Generally the formula $D = \log N / \log K$ is employed to evaluate *a-priori* or to estimate *a-posteriori* (with the *box counting* method) the fractal dimension of a fractal path. D represents the fractal dimension, N the number of segments with which, at recursion step $n+1$, one replaces the segment obtained at the step n, being divided into K parts.

will arise when we will observe "*order* arising from *chance*" thanks to a *law* hidden into the apparent disorder.

Examples

Typically, the programs generating *Mandelbrot set, Julia sets* and *Newton's method fractals* perform a *sequentially ordered* scanning of a square region of the Cartesian plane applying to the co-ordinates of each point a *recursion law* in order to determine if it belongs to the fractal set[9] or not, and plot to the computer display a pixel of a color corresponding to the numerical result obtained.[10]

In particular *Mandelbrot set, Julia sets* nad *Newton's method sets* are representations on the *complex plane* of the domain of convergence of a complex series, while other kinds of fractals may arise by sequential operations on real numbers.

The degree of complexity of those and similar fractals may depend:

i) on the combined effect of the level of non-linearity of the function (*law*) involved in the recursion procedure;

ii) on the number of iterations of the procedure itself;

iii) and on the number of the control parameters included into the *law*.

An intensively studied *recursion law* has the general form:

$$z_{n+1} = f(z_n) + c, \tag{2.3}$$

where $z = x + iy, c = a + ib$ are complex numbers and $f(z)$ is an assigned complex function. The search for the convergence domain of the series:

$$\sum_{n=1}^{+\infty} z_n \equiv z_1 + z_2 + \cdots + z_n + \cdots, \tag{2.4}$$

leads to fractal sets.[11] Gaston Julia (1893-1978) and Benoit Mandelbrot (1924-2010) studied in detail the simplest non-trivial case when:

$$f(z) = z^2. \tag{2.5}$$

In particular when it is assumed that $z_0 = 0$ and c swaps the entire complex plane, the *Mandelbrot set* is generated, while, on the contrary, when c is fixed, during calculations, at some chosen value and z_0 swaps the complex plane, the *Julia sets* are obtained.

[9] The set being defined as the convergence domain of a suitable series.

[10] When colors are chosen according to suitable color maps the beauty of the picture may result of great effect.

[11] With the exception of the trivial function $f(z) = z$.

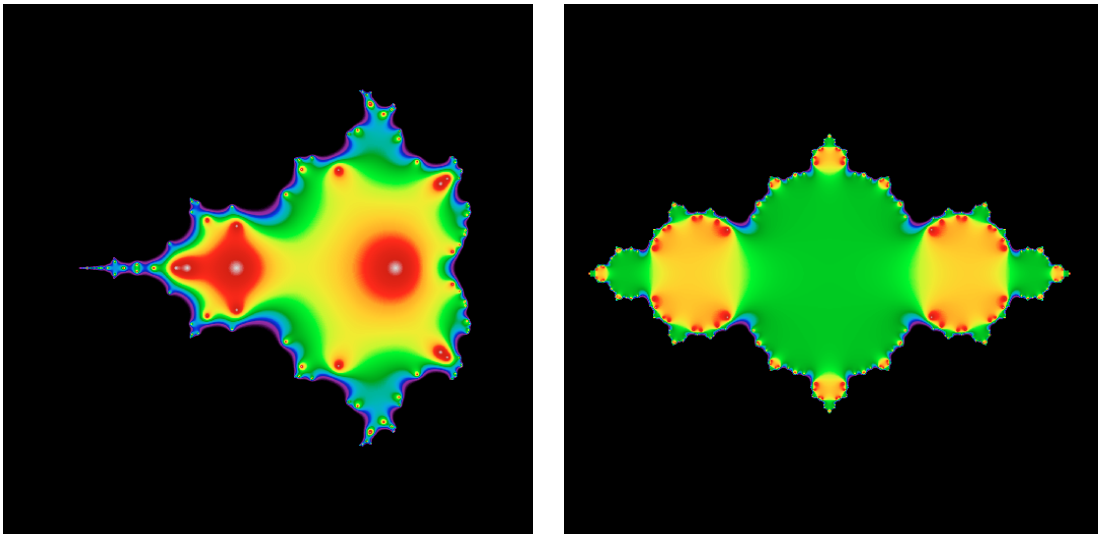

Fig.3 - a) Mandelbrot set; b) Julia set ($c = -0.7454294$)

The *Matplolib* module in *Python 3* provides a very efficient set of instructions to build $2D$ fractals as *wholes*.

Python 3 codes to generate pictures in fig. 3

```
##########################################
# 2D Mandelbrot set with complex arrays
# (matplotlib module)
##########################################

import numpy as np           # import numpy module
import matplotlib.pyplot as plt      # import matplotlib module

n = 8                # set number of cycles
Cx = -.8              # set initial x parameter shift
Cy = 0.0              # set initial y parameter shift
L = 1.7              # set square area side
M = 2024              # set side number of pixels

x = np.linspace(Cx-L,Cx+L,M)      # x variable array
y = np.linspace(Cy-L,Cy+L,M)      # y variable array
X,Y = np.meshgrid(x,y,sparse=True)    # square area grid
Z = np.zeros(M)          # complex starting points area
C = X + 1j*Y            # complex plane area

for k in range(1,n+1):          # recursion cycle
  Z1 = Z**2 + C
  Z = Z1
W = np.e**(-abs(Z))          # smoothed sum moduls

plt.imshow(W,interpolation='nearest', cmap=plt.cm.nipy_spectral)
plt.axis("off")

plt.show()              # plot image
```

```
#####################################################
# 2D Julia set with complex arrays (c=-0.7454294)
# (matplotlib module)
#####################################################

import numpy as np            # import numpy module
import matplotlib.pyplot as plt      # import matplotlib module

n = 9                # set number of cycles
Cx =  -0.7454294          # set c parameter real part value
Cy = 0              # set c parameter imaginary part value
C = Cx + 1j*Cy
L = 1.7              # set square area side
M = 2024              # set side number of pixels

x = np.linspace(-L,L,M)        # x variable array
y = np.linspace(-L,L,M)        # y variable array
X,Y = np.meshgrid(x,y,sparse=True)    # square area grid
Z = X + 1j*Y            # complex plane area

for k in range(1,n+1):          # recursion cycle
  Z1 = Z**2 + C
  Z = Z1
W = np.e**(-abs(Z))          # smoothed sum moduls

plt.imshow(W,interpolation='nearest', cmap=plt.cm.nipy_spectral)
plt.axis("off")

plt.show()              # plot image
```

We may observe that while the *Mandelbrot set* shape is simply stretched and scaled if z_0 is fixed at a value different from zero (see fig. 4), the *Julia sets* assume very different shapes depending on the choice of the parameter c (see figs 5-6).

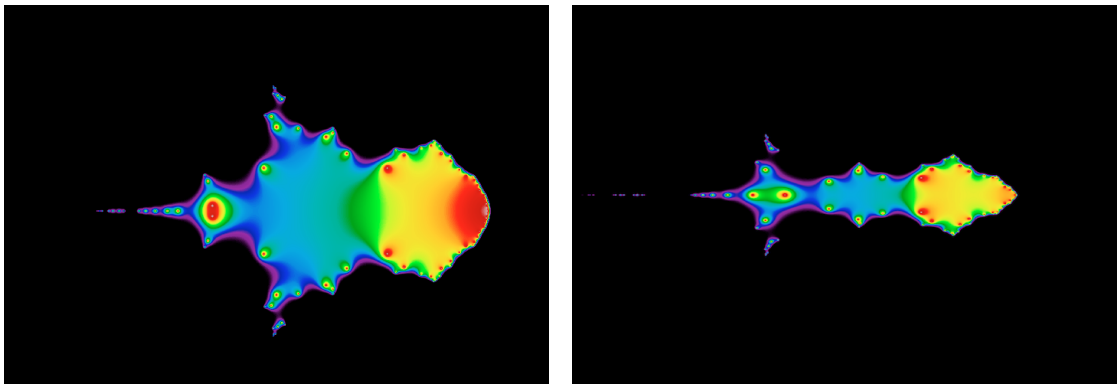

Fig.4 - a) Mandelbrot set ($z_0 = 1.0$); b) Mandelbrot set ($z_0 = 1.3$)

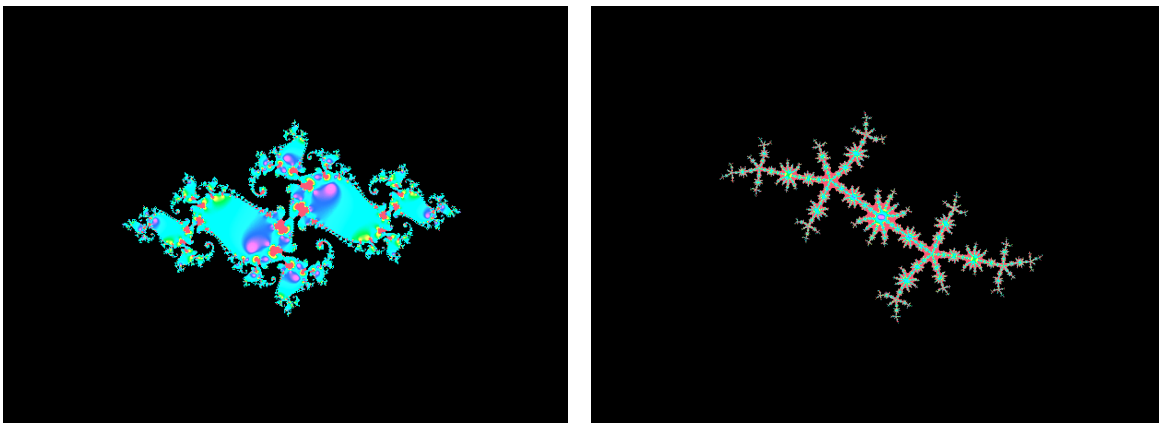

Fig.5 - a) Julia set for $c = -0.7454294 + i0.113089$; b) Julia set for $c = -0.561321 - i0.641$

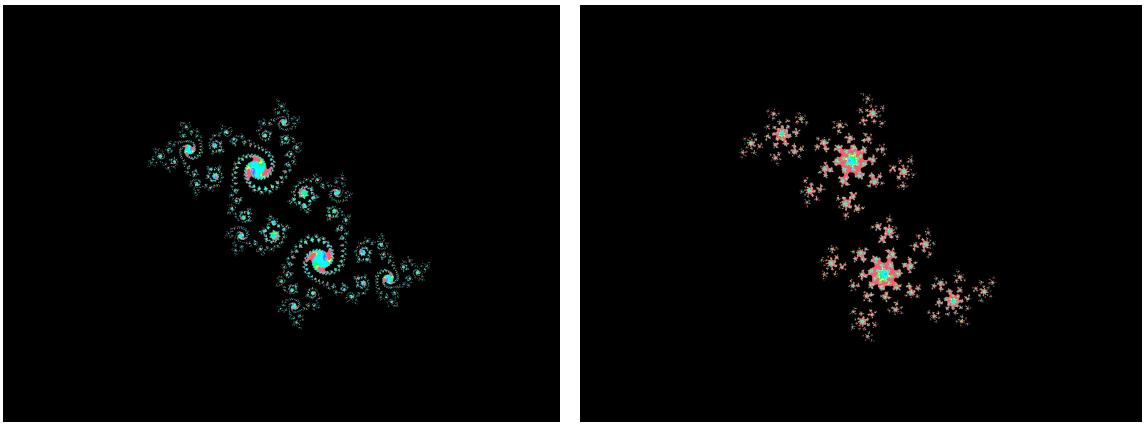

Fig.6 - a) Julia set for $c = -0.2009 - 0.67037$ b) Julia set for $c = 0.11031 - i0.67037$

Generalized Mandelbrot sets have been obtained starting from a different choice of the function $f(z)$, as it is shown in figs 7-10.

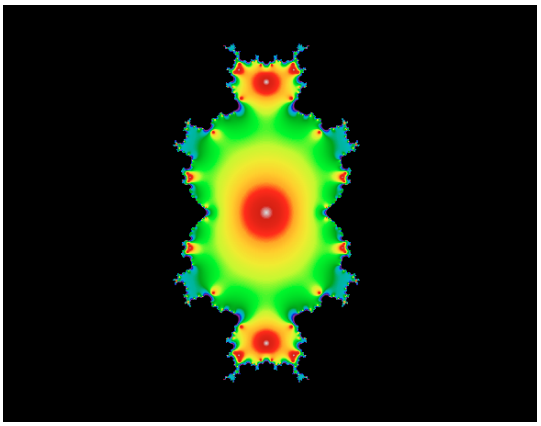

Fig.7 - $f(z) = z^3 + c$

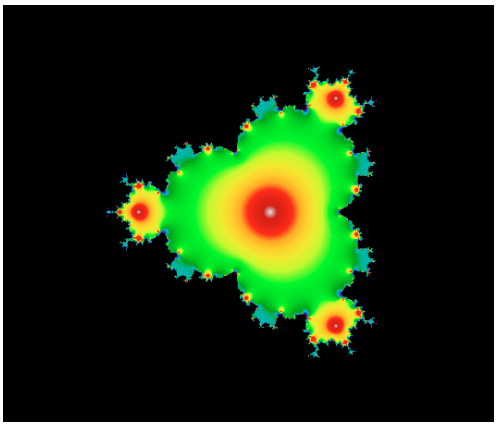

Fig.8 - $f(z) = z^4 + c$

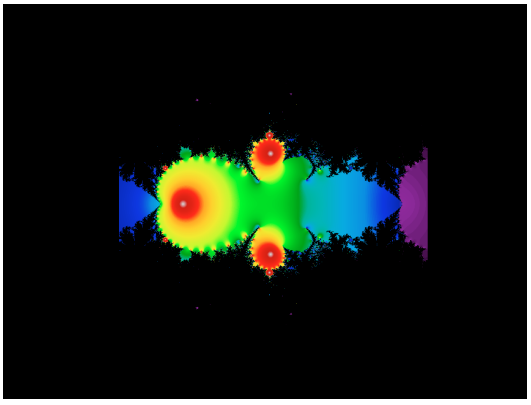

Fig.9 - $f(z) = \cos^2 z + c$

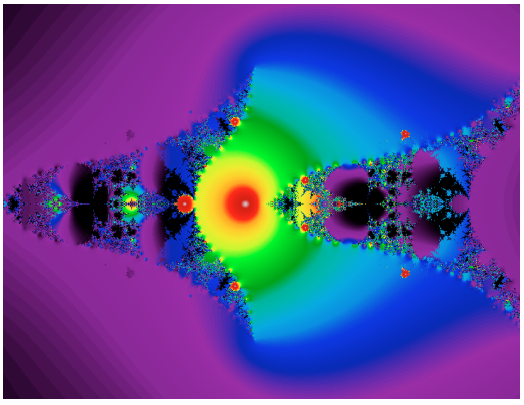

Fig.10 - $f(z) = \tan^2 z + c$

Similarly generalized Julia sets can be obtained starting from a different choice of the function $f(z)$ (see figs 11-12).

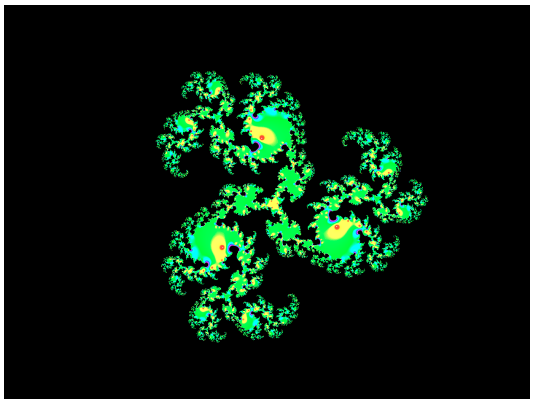

Fig.11 - $f(z) = z^3 + c$

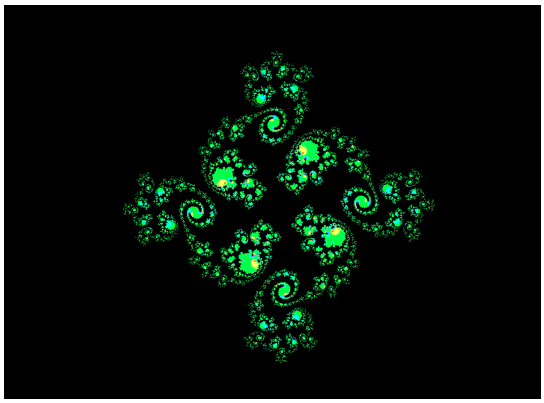

Fig.12 - $f(z) = z^4 + c$

We conclude this section with two examples based on *Newton's method*.

Newton's method is used to check approximated zero solutions to polynomials of any degree, being based on the recursion law:[12]

$$z_{k+1} = z_k - \frac{f(z_k)}{f'(z_k)}. \tag{2.6}$$

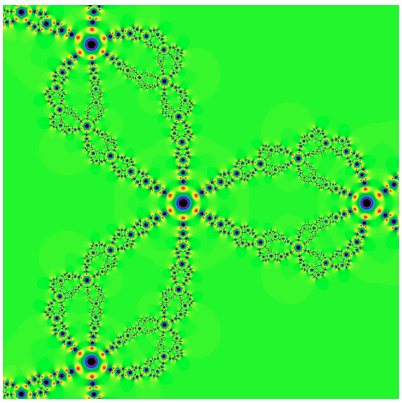

Fig.13 - $f(z) = z^3 + 1$

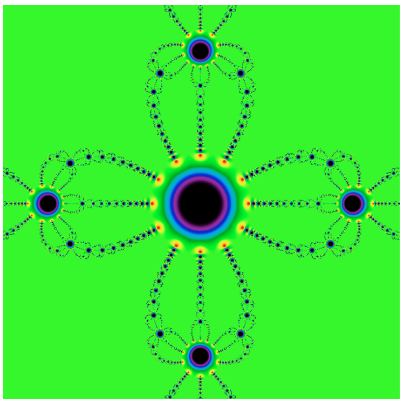

Fig.14 - $f(z) = z^4 + 1$

[12] This law can be obtained by the first order Taylor expansion of $f(z)$ in the neighborhood of z_0, given by: $f(z) = f(z_0) + f'(z_0)(z - z_0)$. Requiring that $f(z_{k+1}) = 0$ and setting $z_0 = z_k$ it results, solving by z_k that $z_{k+1} = z_k - \frac{f(z_k)}{f'(z_k)}$, provided that it is assumed that $f'(z_k \neq 0$.

Python 3 codes to generate Figs 13 and 14

```
################################################
# Newton's method set (Z**3+1=0 or Z**4+1=0)
# with complex arrays (matplotlib module)
################################################

import numpy as np           # import numpy module
import matplotlib.pyplot as plt      # import matplotlib module

n = 8  # alternative n = 12        # set number of cycles
Cx = 0.0              # set initial x parameter shift
Cy = 0.0              # set initial y parameter shift
L = 1.0              # set square area side
M = 2024              # set side number of pixels

x = np.linspace(-L-Cx,L-Cx,M)      # x variable array
y = np.linspace(-L-Cy,L-Cy,M)      # y variable array
X,Y = np.meshgrid(x,y,sparse=True)    # square area grid

Z = X + 1j*Y            # complex plane area

for k in range(1,n+1):          # recursion cycle
  Z1 = Z - (Z**3 + 1)/(3*Z**2)
  # alternative Z1 = Z - (Z**4 + 1)/(4*Z**3)
  Z = Z1

W = np.e**(-.5*abs(Z))        # smoothed sum moduls

plt.imshow(W,interpolation='nearest', cmap=plt.cm.nipy_spectral)
plt.axis("off")
plt.show()              # plot image
```

In the examples we have just presented the *recursion law* (*information*) – according to a philosophical perspective – plays a role which appears to be similar to that of a *form* or *essence* respect to the resulting *entity* (the fractal object), since it defines exactly and univocally its structure. While the individual paper sheet on which the image is printed or the individual screen on which it is displayed plays the role of *matter* determining each *singular* actualization of the *form*. We emphasize that, in the previous examples, the action of the *form* is revealed only as final result of its operation. So the fractal object is considered as a *whole,* while the process of the emergence of its *ordered* structure by the action *form* is not revealed.

In order to reveal how an *algorithm* operates in generating the *whole,* starting from unrelated *parts,* we need a different programming strategy which allows to show the fractal emergence *point by point* and not only as a final *whole*.

In $2D$ we have can achieve easily our goal thanks, *e.g.,* to *Graphics* module in *Python 3*.

Showing the ordered sequential process generating $2D$ fractals *point by point*

Therefore, beside showing the pictures of fractals *as wholes,* it is relevant[13] for us to show also the possibile dynamics capable to generate each image, examining the evolutionary process of image generation at each stage, so revealing the role of *form/information* as *operating nature.*

An elementary process is provided by scanning a region of the complex (or xy) plane *sequentially* (raw after raw, column after column), so that it appears clearly as *order generates order.*

Steps of the generation process of *Mandelbrot set, Julia set* ($c = 0.7454294$) and *Newton's method set* ($f(z) = z^6 + 1$) are shown in figs 15-17. Here are the related programming codes.

Python 3 codes to generate Figs 15, 16 and 17

```
################################################################
# Sequential ordered steps of 2D Mandelbrot set generation
# (graphics module)
################################################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*   # import graphics module

Radius = 10   # set escape rate threshold
x0 = .5   # set initial x co-ordinate shift
y0 = 0.0   # set initial y co-ordinate shift
Side = 1.2   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N   # set number of cycles
sT=5   #  set step jump

win = GraphWin("Mandelbrot set", int(5*M/3),int(5*M/3))  # set window title
win.setBackground("white")     # set background color

def rectCol(p,q,w):   # define elementary cell
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
    Point(int(p+sT/2),int(q+sT/2)))
    Rect.draw(win).setFill(color_rgb(int(w),int(128-w/2),
    int(128+w/2)))

for p in range(1,M,sT):   # column scanning cycle
  Incy = y0 - Side + 2*Side/M*p   # define column scanning function
  for q in range(1,M,sT):   # raw scanning cycle
    Incx = x0 - Side + 2*Side/M*q   # define raw scanning function
    x = 0.0   # set starting x co-ordinate
    y = 0.0   # set starting y co-ordinate
    w = 0   # set starting escape modulus value
    for n in range(1,Num):   # recursion cycle
      xx = x*x - y*y - Incx
      yy = 2*x*y - Incy
      x = xx
      y = yy
```

[13] In particular for future applications to biology, as it will be shown in chapters 8 and 9.

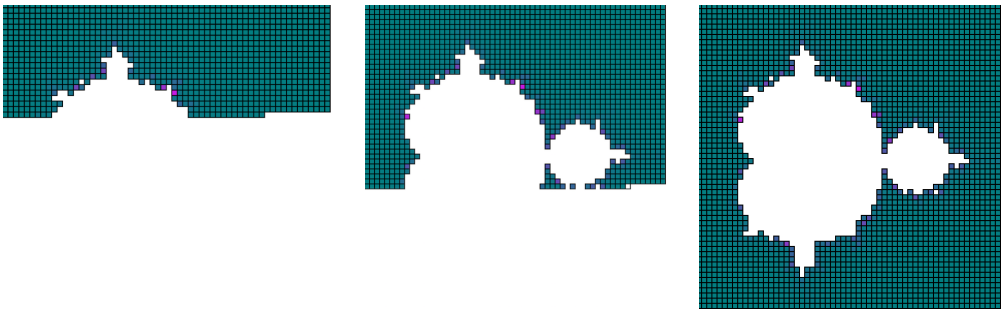

Fig.15 - Rough scheme of ordered sequential generation of Mandelbrot set

VIEW ANIMATION (requires internet connection)

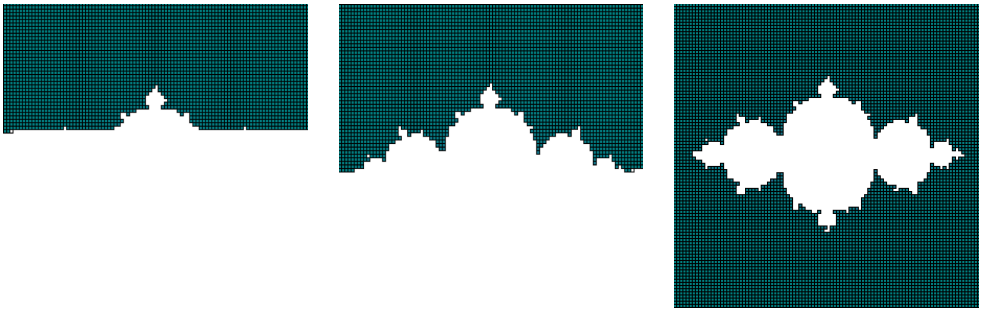

Fig.16 - Rough scheme of ordered sequential generation of a Julia set ($c = 0.7454294$)

VIEW ANIMATION (requires internet connection)

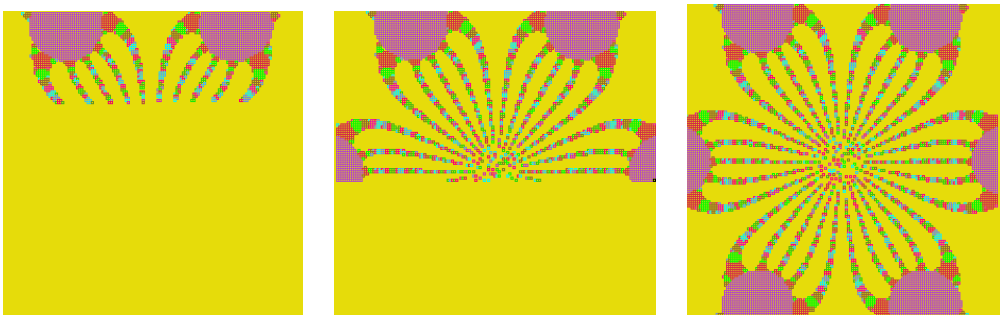

Fig.17 - Rough scheme of ordered sequential generation of a Newton's method set ($f(z) = z^6 + 1$)

VIEW ANIMATION (requires internet connection)

```
      if x*x + y*y > Radius:   # escape rate condition
        w = n/N   # escape modulus normalization
        rectCol(int(M/3+q),int(M/3+p),int(w))   # plot elementary cell
        break  # interrupt cycle

win.getMouse()   # wait for mouse click
win.close()   # close window



##########################################################
# Sequential ordered steps of a 2D Julia set generation
# (c=0.7454294)
# (graphics module)
##########################################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*   # import graphics module

Radius = 10   # set escape rate threshold
Cx = 0.7454294   # set c  parameter real part value
Cy = 0.0   # set c  parameter imaginary part value
Side = 1.7   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N   # set number of cycles
sT=3  #  set step jump

win = GraphWin("Julia set", 5*M/3,5*M/3)  # set window title
win.setBackground("white")     # set background color

def rectCol(p,q,w):   # define elementary cell
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
    Point(int(p+sT/2),int(q+sT/2)))
    Rect.draw(win).setFill(color_rgb(int(w),int(128-w/2),
    int(128+w/2)))

for p in range(1,M,sT):   # column scanning cycle
  Incy = - Side + 2*Side/M*p   # define column scanning function
  for q in range(1,M,sT):   # raw scanning cycle
    Incx = - Side + 2*Side/M*q   # define raw scanning function
    x = Incx   # set starting increment of x co-ordinate
    y = Incy   # set starting increment of y co-ordinate
    w = 0   # set starting escape modulus value
    for n in range(1,Num):  # recursion cycle
      xx = x*x - y*y - Cx
      yy = 2*x*y - Cy
      x = xx
      y = yy
      if x*x + y*y > Radius:   # escape rate condition
        w = n/N   # escape modulus normalization
        rectCol(int(M/3+q),int(M/3+p),int(w))   # plot elementary cell
        break  # interrupt cycle

win.getMouse()   # wait for mouse click
win.close()   # close window
```

```
############################################################
# Sequential ordered steps of a 2D Newton's method set
# generation - Polynomial f(z) = z**6+1
# (graphics module)
############################################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*   # import graphics module
import numpy as np   # import numpy module

Radius = .5   # set escape rate threshold
Cx = 0.0   # set initial x parameter shift
Cy = 0.0   # set initial y parameter shift
Side = .8   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N  # set number of cycles
sT=2   #  set step jump

win = GraphWin("Newton's method set", 5*M/3,5*M/3)  # set window title
win.setBackground(color_rgb(230,220,10))    # set background color

def rectCol(p,q,w):   # define elementary cell
  Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
  Point(int(p+sT/2),int(q+sT/2)))
  Rect.draw(win).setOutline(color_rgb(np.int(255*np.sin(w)**2),
  np.int(255*np.cos(w)**2),np.int(255*np.cos(w/2)**2)))

# Alternative values Cx 0.1747, 0.1747    Cy -.072,-1.072
# Side 0.0015, 0.00015 Num 1024

for p in range(1,M,sT):   # column scanning cycle
  Incy = - Side + 2*Side/M*p   # define column scanning function
  for q in range(1,M,sT):   # raw scanning cycle
    Incx = - Side + 2*Side/M*q   # define raw scanning function
    x = Incx   # set starting increment of x co-ordinate
    y = Incy   # set starting increment of y co-ordinate
    w = 0   # set starting escape modulus value
    for n in range(1,Num):   # recursion cycle
      xx = 5*x/6.0 - x*(x*x*x*x - 10*x*x*y*y +5*y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
      yy = 5*y/6.0 + y*(5*x*x*x*x - 10*x*x*y*y + y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
      x = xx
      y = yy
      if (x-Cx)*(x-Cx) + (y-Cy)*(y-Cy) < Radius:  # escape rate condition
        w = n/N   # escape modulus normalization
        rectCol(int(M/3+q),int(M/3+p),int(w))   # plot elementary cell
        break  # interrupt cycle

win.getMouse()   # wait for mouse click
win.close()   # close window
```

2.2 Ordered entity structures generated by random processes

Now it is relevant, especially regarding biological applications, to observe that the *ordered sequential process,* we have just tested in the previous §2.1 does not provide the only possibile

dynamics capable to generate *ordered* structures. In alternative to assign the *initial conditions* – starting from which one applies the mathematical *algorithm* (*information*) generating the corresponding point of the plot (pixel or cell on the screen) – according to a *sequential order,* we may always choose them *at random.*

With that choice each point or cell will appear on the computer display here and there, *randomly.* But at the end of the process, the same *ordered structure* of the structure will result. In other words, *chance* seems to *generate order,* but only thanks to the *information* hidden within the mathematical law encoded in the algorithm. The ordinating principle is *information* and not *chance* as such.

2.2.1 Showing the random process generating $2D$ fractals

Mandelbrot, Julia and Newton's method sets

A typical example is offered by $2D$ *fractals.*

We present, for a comparison with the sequential process, pictures and *Python 3* related codes generating *Mandelbrot, Julia* and *Newton's method* fractals arising starting from random initial conditions. Of course the smaller are the elementary squares building the plot the more refined image will result.[14]

We show in figs 18, 19 and 20 the same *Mandelbrot, Julia* and *Newton's method sets* considered before, now generated by random assignment of initial conditions. One may recognize how order, initially lacking appears slowly step by step,

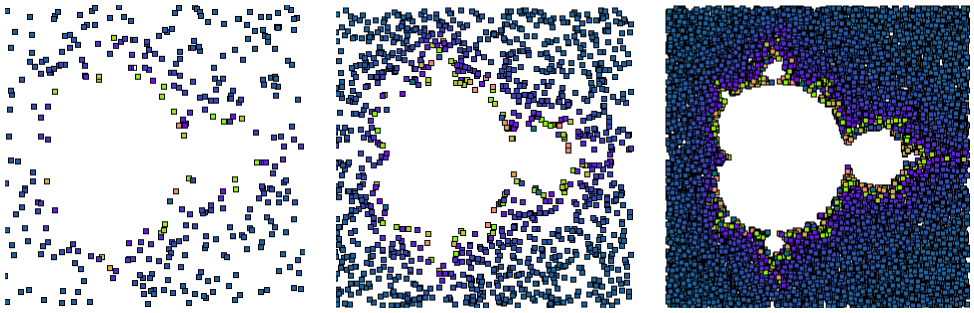

Fig.18 - Rough scheme of random generation of Mandelbrot set

VIEW ANIMATION (requires internet connection)

[14] Naturally a more refined image requires a longer computing time.

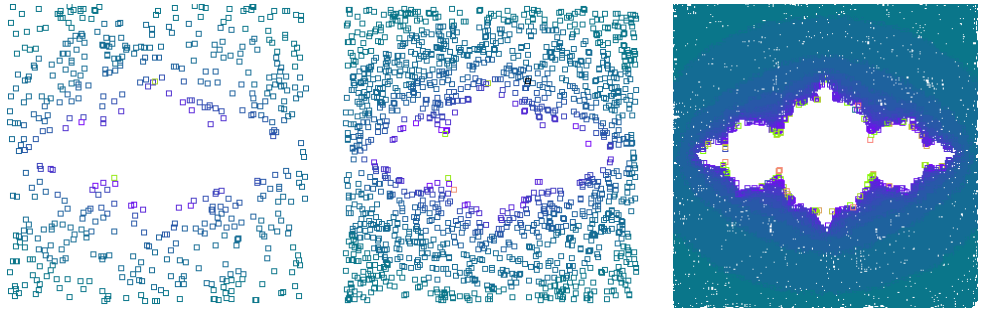

Fig.19 - Rough scheme of random generation of a Julia set ($c = 0.7454294$)

VIEW ANIMATION (requires internet connection)

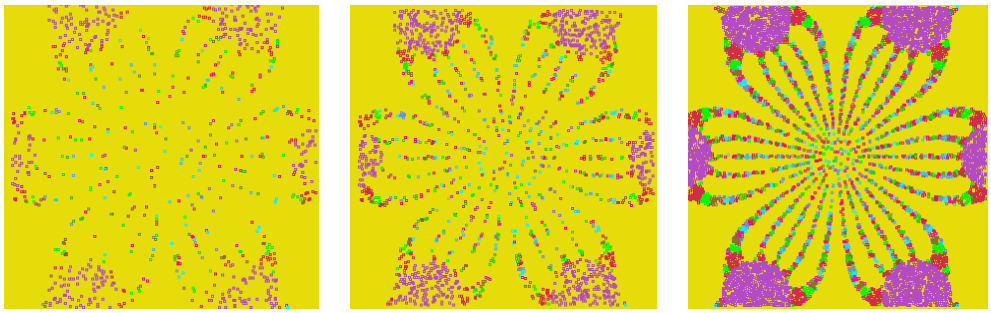

Fig.20 - Rough scheme of random generation of a Newton's method set ($f(z) = z^6 + 1$)

VIEW ANIMATION (requires internet connection)

Python 3 codes to generate Figs 18, 19 and 20

```
##########################################
# 2D Mandelbrot set random generation
# (graphics module)
##########################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import random  # import random module

Radius = 10  # set escape rate threshold
Cx = .5  # set initial x co-ordinate shift
Cy = 0.0  # set initial y co-ordinate shift
Side = 1.3   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
```

```
Num = 256*N   # set number of cycles
sT=5   #  set step jump

win = GraphWin("Mandelbrot set", int(5*M/3),int(5*M/3))   # set window title

def rectCol(p,q,w):      # define elementary cell
  Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
  Point(int(p+sT/2),int(q+sT/2)))
  Rect.draw(win).setFill(color_rgb(int(10*w%255),
  int((128-10*w)%255),int((128+10*w)%255)))

# Alternative values Cx 0.1747, 0.1747 Cy -.072,-1.072
# Side 0.0015, 0.00015 Num 1024

i = 1  # set non-zero index value
while i > 0:   # set random co-ordinates choice cycles
  p = random.randrange(1,M)
  q = random.randrange(1,M)
  Incx = Cx - Side + 2*Side/M*q
  Incy = Cy - Side + 2*Side/M*p
  x = 0.0  # set starting x co-ordinate
  y = 0.0  # set starting y co-ordinate
  w = 0  # set starting escape modulus value
  for n in range(1,Num):
    xx = x*x - y*y - Incx
    yy = 2*x*y - Incy
    x = xx
    y = yy
    if x*x + y*y > Radius:   # escape rate condition
      w = n/N   # escape modulus normalization
      rectCol(int(M/3+q),int(M/3+p),int(w))  # plot elementary cell
      break    # interrupt cycle



###################################
# 2D Julia set random generation
# (graphics module)
###################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import random  # import random module

Radius = 10  # set escape rate threshold
Cx = 0.7454294   # set c  parameter real part value
Cy = 0.0  # set c  parameter imaginary part value
Side = 1.7   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N   # set number of cycles
sT=2   #  set step jump

win = GraphWin("Julia set", 5*M/3,5*M/3)  # set window title

def rectCol(p,q,w):   # define elementary cell
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
    Point(int(p+sT/2),int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(int(10*w%255),
```

```
    int((128-10*w)%255),int((128+10*w)%255)))

# Alternative values Cx 0.1747, 0.1747 Cy -.072,-1.072
# Side 0.0015, 0.00015 Num 1024

i = 1  # set non-zero index value
while i > 0:   # set random co-ordinates choice cycles
  p = random.randrange(1,M)
  q = random.randrange(1,M)
  Incx = - Side + 2*Side/M*q   # set x increment
  Incy = - Side + 2*Side/M*p   # set y increment
  x = Incx
  y = Incy
  w = 0   # set starting escape modulus value
  for n in range(1,Num):
    xx = x*x - y*y - Cx
    yy = 2*x*y - Cy
    x = xx
    y = yy
    if x*x + y*y > Radius:    # escape rate condition
      w = n/N     # escape modulus normalization
      rectCol(int(M/3+q),int(M/3+p),int(w))   # plot elementary cell
      break    # interrupt cycle



###############################################
# 2D Newton's method set random generation
# Polynomial f(z)=z**6+1
# (graphics module)
###############################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import numpy as np   # import numpy module
import random    # import random module

Radius = .5   # set escape rate threshold
Cx = 0.0   # set initial x parameter shift
Cy = 0.0   # set initial y parameter shift
Side = .8    # set square area side
M = 300     # set side number of elementary squares
N = 1    # set color map scale factor
Num = 256*N   # set number of cycles
sT=2   #  set step jump

win = GraphWin("Newton's method set", 5*M/3,5*M/3)  # set window title
win.setBackground(color_rgb(230,220,10))      # set background color

def rectCol(p,q,w):   # define elementary cell
  Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
  Point(int(p+sT/2),int(q+sT/2)))
  Rect.draw(win).setOutline(color_rgb(np.int(255*np.sin(w)**2),
  np.int(255*np.cos(w)**2),np.int(255*np.cos(w/2)**2)))

# Alternative values Cx 0.1747, 0.1747 Cy -.072,-1.072
# Side 0.0015, 0.00015 Num 1024

i = 1  # set non-zero index value
```

```
while i > 0:    # set random co-ordinates choice cycles
  p = random.randrange(1,M)
  q = random.randrange(1,M)
  Incx = - Side + 2*Side/M*q    # set starting increment of x co-ordinate
  Incy = - Side + 2*Side/M*p    # set starting increment of y co-ordinate
  x = Incx
  y = Incy
  w = 0  # set starting escape modulus value
  for n in range(1,Num):
    xx = 5*x/6.0 - x*(x*x*x*x - 10*x*x*y*y +5*y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
    yy = 5*y/6.0 + y*(5*x*x*x*x - 10*x*x*y*y + y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
    x = xx
    y = yy
    if (x-Cx)*(x-Cx) + (y-Cy)*(y-Cy) < Radius:    # escape rate condition
      w = n/N   # escape modulus normalization
      rectCol(int(M/3+q),int(M/3+p),int(w))    # plot elementary cell
      break   # interrupt cycle
```

The iterating function system (IFS) generating natural ordered fractal structures

Another example of *ordered structures* generated by a *random dynamics* governed by a mathematical *algorithmic information* is offered by the fractals obtained applying the *Iterated Function System* (*IFS*). This method is employed, generally, to model shapes existing in nature, like coast or mountain profiles, leaves, ferns, trees, clouds and so on. The recursion law of the algorithm is characterized by *affine transformations* of type:

$$x_{p\,n+1} = a_p\, x_n + b_p\, y_n + c_p, \qquad y_{p\,n+1} = d_p\, x_n + e_p\, y_n + f_p, \tag{2.7}$$

where $a_p, b_p, c_p, d_p, e_p, f_p$ are constant coefficients the value of which is chosen in a suitable way in order to obtain the desired shape. This method introduces chance at the level of the *random probability* p according to which each coefficient value may occur. So, if *e.g.,* a randomly chosen p' is greater than p_1 and less than p_2, the coefficients will assume the values $a_{p_1}, b_{p_1}, c_{p_1}, d_{p_1}, e_{p_1}, f_{p_1}$. While if a different random value p'' of the probability occurs, say, between p_2 and p_3 the coefficients will be assigned to the different values $a_{p_2}, b_{p_2}, c_{p_2}, d_{p_2}, e_{p_2}, f_{p_2}$. Then the law (2.7) is changed according to some chance criterion. Typical examples are the *fractal fern* (fig. 21), the *fractal tree* (fig. 22), or the *Sierpinski triangle* (fig. 23).

Fig.21 - Fern generation steps (IFS method)

VIEW ANIMATION (requires internet connection)

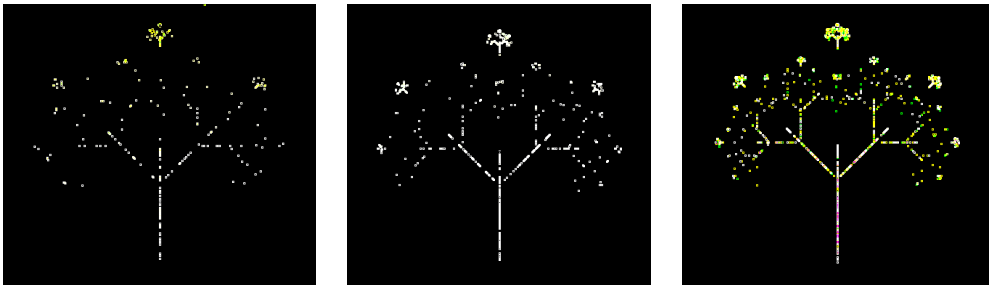

Fig.22 - Tree generation steps (IFS method)

VIEW ANIMATION (requires internet connection)

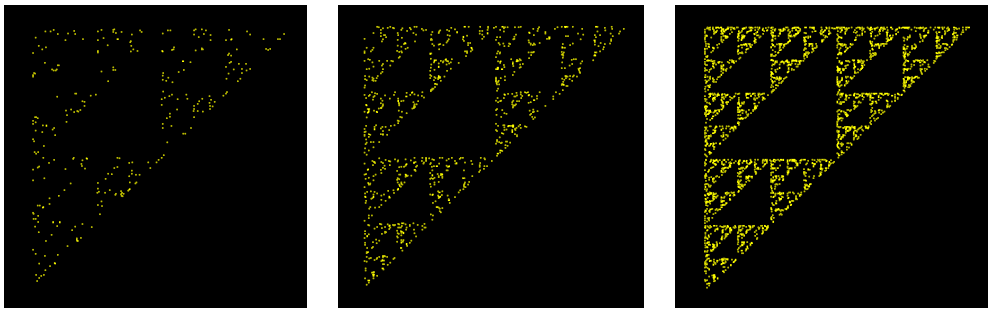

Fig.23 - Sierpinski triangle generation steps (IFS method)

VIEW ANIMATION (requires internet connection)

The related computer codes of programs to generate such images are given below.

Python 3 codes to generate figs 21, 22 and 23

```
################################
# IFS fern random generation
# (graphics module)
################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import numpy as np    # import numpy module
import random    # import random module

Mxy=[[0.0,0.0,0.0,0.6,0.0,0.0,0.01],          # assign probability matrix
    [0.85,0.04,-0.04,0.85,0.0,1.6,0.85],
    [0.2,-0.26,0.23,0.22,0.0,1.6,0.07],
    [-0.15,0.28,0.26,0.24,0.0,0.44,0.07]]

a = [0,.85, .2, -.15]       # assign affine transformations coefficients
```

```
b = [0, .04, -.26, .28]
c = [0, -.004, .23, .26]
d = [.16, .85, .22, .24]
e = [0, 0, 0, 0]
f = [0, 1.6, 1.6, .44]

M = 300    # set side number of elementary squares
Num = 30000   # set number of cycles
sT = 1    #  set step jump

win = GraphWin("Fern", 2*M,2*M)   # set window title
win.setBackground('black')       # set background color

def rectCol(p,q,w):     # define elementary cell
    Rect = Rectangle(Point(np.int(p-sT/2),np.int(q-sT/2)),
    Point(np.int(p+sT/2),np.int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(w,255-np.int(.5*w),
    np.int(.5*w)))

x = 1    # set x initial value
y = 1    # set x initial value

for n in range(0,Num):    # set random probabilities choice cycles
  P = random.random()
  if P <= Mxy[0][6]:
    k = 0
  elif P <= Mxy[0][6] + Mxy[1][6]:
    k = 1
  elif P <= Mxy[0][6] + Mxy[1][6] + Mxy[2][6]:
    k = 2
  else:
    k = 3

  xx = a[k]*x+b[k]*y+e[k]     # affine transformation recursion cycle
  yy = c[k]*x+d[k]*y+f[k]
  x = xx
  y = yy

  rectCol(np.int(M+50*x),np.int(2*M-30-50*y),np.int(np.abs(20*y)))    # plot elementary cell

win.getMouse()    # wait for mouse click
win.close()    # close window


################################
# IFS tree random generation
# (graphics module)
################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import numpy as np    # import numpy module
import random    # import random module

Mxy=[[0.195 , -0.488, 0.344 , 0.443 , 0.4431, 0.2452],          # assign probability matrix
  [0.462 , 0.414 , -0.252, 0.361 , 0.2511, 0.5692],
  [-0.058, -0.07 , 0.453 , -0.111, 0.5976, 0.0969],
  [-0.035, 0.07  , -0.469, -0.022, 0.4884, 0.5069],
  [-0.637, 0     , 0     , 0.501 , 0.8662, 0.2513]]
```

```
# Mxy=[[0.0,0.0,0.0,0.6,0.0,0.0,0.01],
#     [0.85,0.04,-0.04,0.85,0.0,1.6,0.85],
#     [0.2,-0.26,0.23,0.22,0.0,1.6,0.07],
#     [-0.15,0.28,0.26,0.24,0.0,0.44,0.07]]

a = [0,.42, .42, .1]     # assign affine transformations coefficients
b = [0, -.42, .42, 0]
c = [0, .42, -.42, 0]
d = [.5, .42, .42, .1]
e = [0, 0, 0, 0]
f = [0, .2, .2, .5]

M = 300    # set side number of elementary squares
Num = 3000  # set number of cycles
sT = 2    #  set step jump

win = GraphWin("Tree", 2*M,2*M)   # set window title
win.setBackground('black')       # set background color

def rectCol(p,q,w):       # define elementary cell
    Rect = Rectangle(Point(np.int(p-sT/2),np.int(q-sT/2)),
    Point(np.int(p+sT/2), np.int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(w,255-w,np.int(.5*w)))

x = 1    # set x initial value
y = 1    # set x initial value

for n in range(0,Num):     # set random probabilities choice cycles
  k = random.randrange(0,4)

  xx = a[k]*x+b[k]*y+e[k]          # affine transformation recursion cycle
  yy = c[k]*x+d[k]*y+f[k]
  x = xx
  y = yy

  # plot elementary cell
  rectCol(np.int(M+600*x),np.int(2*M-100-600*y), np.int(n*128/Num+np.abs(100*(.8-y))))

win.getMouse()    # wait for mouse click
win.close()   # close window



##############################################
# IFS Sierpinski triangle random generation
# (graphics module)
##############################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*   # import graphics module
import numpy as np     # import numpy module
import random    # import random module

a = [0.50,  0.50,  0.50]       # assign affine transformations coefficients
b = [0.00,  0.00,  0.00]
c = [0.00,  0.00,  0.00]
d = [0.50,  0.50,  0.50]
e = [0.00,  0.00,  0.50]
```

```
f = [0.00,  0.50,  0.50]

M = 300    # set side number of elementary squares
Num = 10000   # set number of cycles
sT = 1   #  set step jump

win = GraphWin("Sierpinski triangle", 2*M,2*M)   # set window title
win.setBackground('black')   # set background color

def rectCol(p,q,w):        # define elementary cell
    Rect = Rectangle(Point(np.int(p-sT/2),
    np.int(q-sT/2)),Point(np.int(p+sT/2),np.int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(w,w,0))

x = 1    # set x initial value
y = 1    # set x initial value

for n in range(0,Num):       # set random probabilities choice cycles
  k = random.randrange(0,3)

  xx = a[k]*x+b[k]*y+e[k]           # affine transformation recursion cycle
  yy = c[k]*x+d[k]*y+f[k]
  x = xx
  y = yy

  rectCol(np.int(np.int(.2*M)+500*x),np.int(2*M-50-500*y),255)      # plot elementary cell

win.getMouse()   # wait for mouse click
win.close()      # close window
```

2.2.2 Remark

The generation of an ordered structure starting from random initial conditions seems especially interesting in order to model biological entities (cells, organs, etc.). In fact a living ordered structure seems to appear, quite magically, by random process and arise by chance. Actually chance involves only the *initial conditions* and perhaps some of the subsequent *bifurcations* of the generation process dynamics, while some information hidden into, *e.g.,* the *DNA* and other supports, governs the entire generation dynamics. Such information may probably be hidden into some very complex string, or a nested structure of strings which partly is able to write its code step by step, as an unfolding strip.

2.3 Non-computable $2D$ structures

Only for some special structures[15] one is able to find a *mathematical formula* (*law*) which allows to define a sort of *essence* of some entity (*body* or *system*), which may be coded into a string shorter than the mere list of the co-ordinates of each point of the body or system itself. We have seen, in the previous sections, the examples of $2D$ fractals as significant structures.

[15] Some of those structures are well known and have been deeply studied.

Very many situations are known such that a compact formula cannot be found

– either because of some *technical difficulties*

– or for *principle reasons.*

In the former case one may always hope that in future a skillful and lucky researcher will be able to grasp such hidden law. In the latter case this lucky circumstance will be impossible, since the *Gödel's number* representing this formula is non-computable and the associated proof of the law results *undecidable* within the axiomatic system. According to computer science language one says that the *string* of the list of all the co-ordinates of the system points results to be incompressible and no regular order appears examining the sequence of the digits of the string, or the map of the points representing them geometrically. In some situation the compression of the string may be made *locally,* thanks to some technical trick,[16] but it does not exist a *global* unique formula (*shorter string*) compacting the whole structure of the system, defining it as a sort of *essence.*

2.3.1 Sequential process generating a map of prime numbers

An interesting example of non-compressible string seems to be offered (at least until now) by the sequence of the *prime numbers.*[17]

In fact, at least at present, we do not know any *law* to generate a number formed by the sequence of the first n prime numbers, shorter than the full list of those number themselves; like *e.g.,* the number built by the sequence of the first 5 numbers greater than 2. The first 5 prime numbers greater than 2 are $3, 5, 7, 11, 13$ and the number resulting is 3571113. In similar situations an ordered dynamics as the sequential scanning of a region of the Cartesian plane does not seem to produce any order. In the following figures we have plotted a portion of the Cartesian plane in such a way that

– red pixels are associated to points the absolute values of the co-ordinates of which are both prime numbers;

– green pixels are related to points of prime abscissa and non-prime ordinate;

– blue pixels are related to points of non-prime abscissa and prime ordinate;

– white pixels are related to points the co-ordinates of which are both non-prime numbers.

In particular, in fig. 24 the dynamics generated the plot is *sequentially ordered,* while in fig. 25 the dynamics generating the plot is *random.* In both cases the co-ordinates of each point are to be evaluated individually since there is no recursion formula allowing to generate the subsequent prime number starting from a known one. We point out that notwithstanding

[16] Generally the compression methods of image or text files are based on such local expedients which allow to shorten, *e.g.* a sequence of identical digits.

[17] We remember that a natural number n is said to be *prime* if it allows as exact divisors only the unit (1) and itself (n).

that the figure is plotted according to some symmetry criterion, since for each pair of prime numbers (x, y) we plot four symmetric points of co-ordinates $(x, y), (-x, y), (x, -y), (-x, -y)$, the visual perception of such symmetries is gradually lost being overridden by the randomness of the prime number sequence.

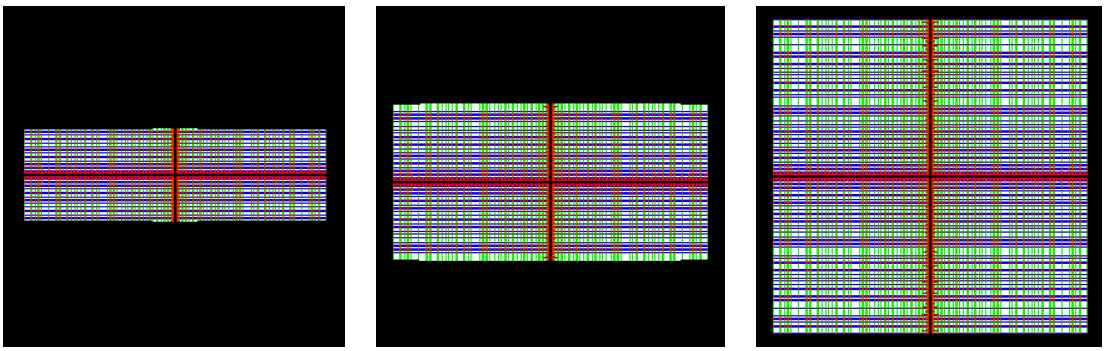

Fig.24 - Prime numbers sequential generation steps

VIEW ANIMATION (requires internet connection)

Python 3 code to generate images in fig. 24

```
############################
# Prime numbers > 2
# sequential generation
# (graphics module)
############################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*     # import graphics module
import numpy as np       # import numpy module

M = 253    # set side number of elementary square (max number to be checked)
sT=1    #  set step jump

P = np.zeros(M)    #  x co-ordinate array
Q = np.zeros(M)   #  y co-ordinate array
r = np.zeros([M,M])    # zero red color matrix
g = np.zeros([M,M])   # zero green color matrix
b = np.zeros([M,M])   # zero blue color matrix

win = GraphWin("Prime set (sequential)", int(10*M/3),int(10*M/3))    # set window title
win.setBackground("black")     # set background color

def rectCol(p,q,R,G,B):         # define elementary cell
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),Point(int(p+sT/2),
    int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(R,G,B))

for p in range(3,M,2):    # x sequential cycle
```

```
  for q in range(3,M,2):     # y sequential cycle
    j = 3     # set initial prime number
    Wp = 1    # set default x control index
    Wq = 1    # set default y control index

    while j < p and j < q:     # prime number checking cycle
      if p%j != 0:
        WWp = 1
        Wp = Wp*WWp  # ensures that no quotient is exact
      else:
        WWp = 0
        Wp = Wp*WWp  # ensures that no quotient is exact
      if q%j != 0:
        WWq = 1
        Wq = Wq*WWq  # ensures that no quotient is exact
      else:
        WWq = 0
        Wq = Wq*WWq  # ensures that no quotient is exact
      j = j + 2

    if (Wp == 1 and Wq == 1):     # red color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 255
      g[p,q] = 0
      b[p,q] = 0
    elif (Wp == 1 and Wq == 0):      # green color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 0
      g[p,q] = 255
      b[p,q] = 0
    elif (Wp == 0 and Wq == 1):      # blue color select  conditions
      P[p] = p
      Q[q]
      r[p,q] = 0
      g[p,q] = 0
      b[p,q] = 255
    elif (Wp == 0 and Wq == 0):     # white color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 255
      g[p,q] = 255
      b[p,q] = 255

for q in range(3,M,2):       # sequential  plot cycles
  for p in range (3,M,2):
    rectCol(int(5*M/3-P[p]),int(5*M/3+Q[q]),
    np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
    rectCol(int(5*M/3+P[p]),int(5*M/
    3+Q[q]),np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
    rectCol(int(5*M/3+P[p]),int(5*M/3-Q[q]),
    np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
    rectCol(int(5*M/3-P[p]),int(5*M/3-Q[q]),
    np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))

win.getMouse()     # wait for mouse click
win.close()      # close window
```

2.3.2 Random process generating a map of prime numbers

Here are the images and code related to ordered pairs of prime number generation according to a *random* choice of initial conditions. Neither *sequentially ordered* nor *random* dynamics seems to generate order.

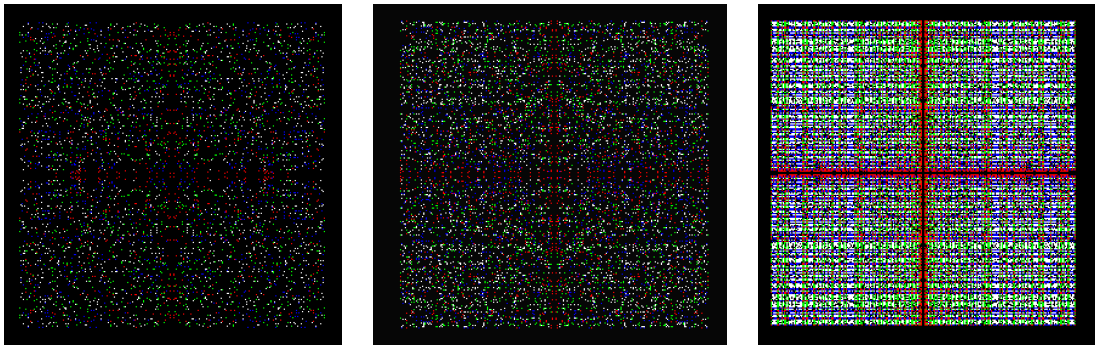

Fig.25 - Prime numbers random generation steps

VIEW ANIMATION (requires internet connection)

Python 3 code to generate images in fig. 25

```
########################
# Prime numbers > 2
# random generation
# (graphics module)
########################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*      # import graphics module
import numpy as np       # import numpy module
import random as rd      # import random module

M = 253    # set side number of elementary square (max number to be checked)
sT=1    #  set step jump

P = np.zeros(M)     #  x co-ordinate array
Q = np.zeros(M)    #  y co-ordinate array

r = np.zeros([M,M])     # zero red color matrix
g = np.zeros([M,M])    # zero green color matrix
b = np.zeros([M,M])    # zero blue color matrix

win = GraphWin("Prime set (random)", int(10*M/3),int(10*M/3))    # set window title
win.setBackground("black")      # set background color

def rectCol(p,q,R,G,B):          # define elementary cell
```

```
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),Point(int(p+sT/2),
    int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(R,G,B))

for p in range(3,M,2):      # x sequential cycle
  for q in range(3,M,2):      # y sequential cycle
    j = 3     # set initial prime number
    Wp = 1      # set default x control index
    Wq = 1    # set default y control index

    while j < p and j < q:       # prime number checking cycle
      if p%j != 0:
        WWp = 1
        Wp = Wp*WWp  # ensures that no quotient is exact
      else:
        WWp = 0
        Wp = Wp*WWp  # ensures that no quotient is exact
      if q%j != 0:
        WWq = 1
        Wq = Wq*WWq  # ensures that no quotient is exact
      else:
        WWq = 0
        Wq = Wq*WWq  # ensures that no quotient is exact
      j = j + 2

    if (Wp == 1 and Wq == 1):      # red color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 255
      g[p,q] = 0
      b[p,q] = 0
    elif (Wp == 1 and Wq == 0):    # green color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 0
      g[p,q] = 255
      b[p,q] = 0
    elif (Wp == 0 and Wq == 1):    # blue color select  conditions
      P[p] = p
      Q[q]
      r[p,q] = 0
      g[p,q] = 0
      b[p,q] = 255
    elif (Wp == 0 and Wq == 0):    # white color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 255
      g[p,q] = 255
      b[p,q] = 255


i = 1      # set non-zero control paramter value
while i > 0:    # random point selection cycles
  p = rd.randrange(1,M,2)
  q = rd.randrange(1,M,2)

  # plot cell
  rectCol(int(5*M/3-P[p]),int(5*M/3+Q[q]),
  np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
  rectCol(int(5*M/3+P[p]),int(5*M/
  3+Q[q]),np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
  rectCol(int(5*M/3+P[p]),int(5*M/3-Q[q]),
```

```
np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
rectCol(int(5*M/3-P[p]),int(5*M/3-Q[q]),
np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
```
