

## Curbe care trec prin toate punctele unui plan

### §1. Parcurgerea planului în $Z$ -ordine

Să considerăm o bază de date bidimensională – cu două chei de intrare – pe care o putem reprezenta “spațial” ca un tablou dreptunghiular de elemente dispuse pe  $m$  linii și  $n$  coloane, fiecare element fiind accesat prin precizarea a doi indici,  $x$  – indicele de coloană, abscisa, și  $y$  – indicele de linie, ordonata. Se pune problema *serializării* acestei bazei de date, a stocării ei secvențiale, într-un fisier pe disc, de exemplu.

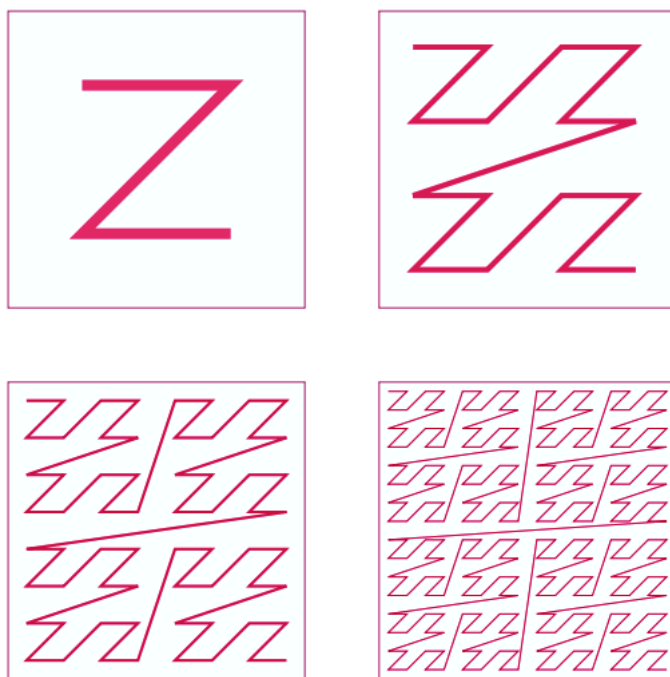


FIGURA 1. Parcurgerea în  $Z$ -ordine.

Altfel spus, trebuie să precizăm o *parcursere* a tabloului într-o anumită ordine, prin care să accesăm fiecare element o singură dată, mai precis, să stabilim o corespondență bijectivă între produsul cartezian

$$\{0, 1, \dots, n-1\} \times \{0, 1, \dots, m-1\}$$

și mulțimea  $\{0, 1, \dots, mn-1\}$ .

Soluția cea mai simplă și cea mai des folosită este *parcurgerea pe linii*, de la stânga la dreapta și de sus în jos. Dacă notăm cu  $N$  rangul lui  $(x, y)$  în înșiruirea  $(0, 0), (1, 0), \dots, (n-1, 0), (0, 1), (1, 1), \dots, (0, m-1), (1, m-1), \dots, (n-1, m-1)$  atunci avem relația

$$N = x + ny$$

și reciproc,

$$x = N \bmod n, \quad y = (N - x)/n.$$

Aceast mod de parcurgere are două puncte slabe: operațiile aritmetice implicate în găsirea indicilor  $x$  și  $y$  nu sunt adecvate calculului pe biți și prin urmare sunt consumatoare de timp, iar parcurgerea nu este *localizată*, două elemente vecine în matrice se pot găsi oricât de departe în înșiruire – vezi cazul a două elemente aflate unul sub altul pe o coloană.

Aceste două impedimente sunt înlăturate, măcar parțial, de *parcurgerea în Z-ordine*, care este, în esență, forma bidimensională a algoritmului de căutare binară. Împărțim matricea în două benzi orizontale și două verticale, obținem patru regiuni pe care le parcurgem după un drum în forma literei Z: sus stânga – dreapta, apoi jos stânga – dreapta. Dacă este necesar, fiecare regiune va fi parcursă tot în această ordine (vezi Figura 1).

	x: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
y: 0 000	000000 000001 000100 000101 010000 010001 010100 010101							
1 001	000010 000011 000110 000111 010010 010011 010110 010111							
2 010	001000 001001 001100 001101 011000 011001 011100 011101							
3 011	001010 001011 001110 001111 011010 011011 011110 011111							
4 100	100000 100001 100100 100101 110000 110001 110100 110101							
5 101	100010 100011 100110 100111 110010 110011 110110 110111							
6 110	101000 101001 101100 101101 111000 111001 111100 111101							
7 111	101010 101011 101110 101111 111010 111011 111110 111111							

FIGURA 2. Algoritmul de intercalare binară

Este clar că parcurgerea în  $Z$ -ordine are o localizare mai bună decât parcurgerea pe linii, avantajul principal constă totuși în algoritmul de determinare a corespondenței

$$N \leftrightarrow (x, y)$$

sugerat de Figura 2, și anume: utilizând scrierea numerelor în baza 2, biții lui  $N$  se obțin prin intercalarea biților lui  $y$  cu ai lui  $x$ , așezând fiecare bit din  $y$  în fața bitului corespunzător lui în  $x$ . Justificarea este foarte simplă: la împărțirea inițială în două benzi orizontale și două benzi verticale, primul bit al lui  $y$  precizează dacă elementul  $(x, y)$  se află în banda de sus sau în cea de jos, iar primul bit al lui  $x$  precizează dacă el se află în stânga sau în dreapta, așa că numărarea pe doi biți

$$00, 01, 10, 11$$

dă exact parcurgerea celor patru regiuni în  $Z$ -ordine. Aplicând recursiv procedeul în fiecare regiune, obținem exact algoritmul precizat.

Să notăm următorul dezavantaj al metodei:  $n$  și  $m$  trebuie să fie egale și de forma  $n = m = 2^k$ .

În clasa  $C^\#$  următoare aplicăm algoritmul de mai sus ca să parcurgem în  $Z$ -ordine o rețea de puncte din planul complex, mai precis, cele  $2^{2k}$  puncte din pătratul unitate  $[0, 1] \times [0, 1]$  de forma

$$z = (\tilde{x} + i\tilde{y})/2^k$$

cu  $\tilde{x}$  și  $\tilde{y}$  din  $\{0, 1, \dots, 2^k - 1\}$ .

```
import ComplexPygame as C
import Color
def Z_orderBin():
    def incrementare(regBin):
        # trecem pe 01000111
        # in          01001000
        # prin adunarea cu 1
        avemTransport = True
        k = 0
        while k < len(regBin) and avemTransport:
            if regBin[k] == 1:
                regBin[k] = 0
            else:
                regBin[k] = 1
                avemTransport = False
            k += 1
        return not avemTransport

    def traseaza(li):
        for n in range(1, len(li)):
            C.drawLine(li[n - 1], li[n], Color.Index(600 + n // 5))
        C.refreshScreen()
```

```

C.setXminXmaxYminYmax(-0.1, 1.1, 1.1, -0.1)
k = 6
k2 = 2 * k
reg = [0] * k2
fig = [0]
while incrementare(reg):
    # aflam x si y
    y = 0
    p = 1
    for h in range(k2 - 1, 0, -2):
        p *= 2
        y += reg[h] / p
    x = 0
    p = 1
    for h in range(k2 - 2, -1, -2):
        p *= 2
        x += reg[h] / p
    fig.append(complex(x, y))
traseaza(fig)

if __name__ == '__main__':
    C.initPygame()
    C.run(Z_orderBin)

```

În program generăm scrierea binară a rangului  $N$  direct într-un registru pe  $2k$  întregi, pe care îl inițializăm cu zero și apoi îl incrementăm cu funcția `incrementare()` până când obținem depășire de format. La fiecare pas obținem coordonatele  $x$  și  $y$  ale punctului curent  $z$  prin dez-intercalarea biților lui  $N$  și memorăm punctul  $z$  în lista `fig`. În final desenăm traseul păstrat în listă.

O funcție care implementează algoritmul de intercalare binară de mai sus utilizând direct operații pe biți este următoarea:

```

def Z_orderPeBiti():
    k = 6 # atentie: 0 < k < 15
    nrPuncte = 1 << (2 * k)
    lat = 1 << k
    C.setXminXmaxYminYmax(-1, lat, lat, -1)
    zvechi = 0
    for n in range(nrPuncte):
        x = 0
        y = 0
        nn = n
        for h in range(k):
            x |= (nn & 1) << h
            nn >>= 1
            y |= (nn & 1) << h
            nn >>= 1

```

```

znou = complex(x, y)
C.drawLine(znou, zvechi, Color.Index(n // 10))
zvechi = znou
if C.mustClose(): return

```

Observație: pentru ca sensul de creștere a lui  $y$  pe ecran să fie în jos (ca la matrice) în apelul

```
setXminXmaxYminYmax(-1, lat, lat, -1);
```

am fixat  $y_{\min}$  mai mare decât  $y_{\max}$ !

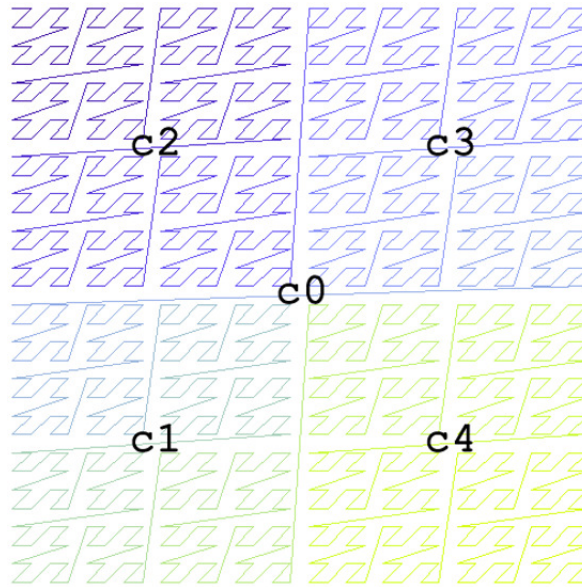


FIGURA 3.  $Z\_orderBin()$

## §2. Metoda transformărilor iterate

Vom trasa acum curba dată de parcurgere în  $Z$  ordine pe baza următoarei proprietăți de autosimilaritate observate în Figura 3: pentru fiecare  $k$ , în etapa de ordin  $k$  avem în pătratul unitate o parcurgere din colțul din stânga – sus până în colțul din dreapta – jos. Dacă reducem totul la scara  $1/2$  cu centru în  $c_0$  – centrul pătratului inițial –, translatăm de patru ori “pătrățelul” obținut astfel încât  $c_0$  să ajungă pe rând în  $c_2$ ,  $c_3$ ,  $c_1$  și  $c_4$  – în această ordine – și racordăm în formă de  $Z$  cele patru parcurgeri, vom obține curba de ordin  $k + 1$ .

Următoarea clasă implementează această metodă, numită *metoda transformărilor iterate*, pentru trasarea parcurgerii în  $Z$ -ordine, utilizând cele patru transformări geometrice descrise mai sus, notate cu  $s_1$ ,  $s_2$ ,  $s_3$  și  $s_4$ .

```

def Z_order():
    c0 = (1 + 1j) / 2
    c1 = (1 + 1j) / 4
    c2 = (1 + 3j) / 4

```

```

c3 = (3 + 3j) / 4
c4 = (3 + 1j) / 4

# sk(z)=c0+0.5*(z-c0) +(ck-c0)=ck+0.5*(z-c0)
def s1(z):
    return c1 + 0.5 * (z - c0)

def s2(z):
    return c2 + 0.5 * (z - c0)

def s3(z):
    return c3 + 0.5 * (z - c0)

def s4(z):
    return c4 + 0.5 * (z - c0)

def transforma(li):

    rez = []
    for z in li:
        rez.append(s2(z))
    for z in li:
        rez.append(s3(z))
    for z in li:
        rez.append(s1(z))
    for z in li:
        rez.append(s4(z))
    return rez

def traseaza(li):
    C.fillScreen()
    # trasam chenarul
    C.drawNgon([0, 1, 1 + 1j, 1j], Color.Index(500))
    # desenam curba curenta
    for n in range(1, len(li)):
        C.drawLine(li[n - 1], li[n], Color.Index(n // 5))

C.setXminXmaxYminYmax(-0.1, 1.1, -0.1, 1.1)
fig = [c0]
for k in range(5):
    fig = transforma(fig)
    traseaza(fig)
    if C.mustClose(): return
    C.wait(100)

```

Dacă în figura inițială avem numai punctul  $c_0$ , vom obține la fiecare etapă exact curbele generate de programul precedent

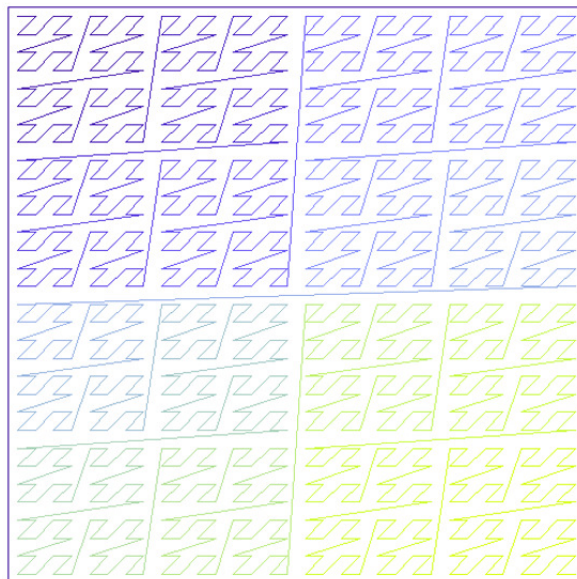


FIGURA 4. `Z_order()`

Observăm că, pentru  $k$  din ce în ce mai mare, curbele noastre trec printr-o rețea din ce în ce mai deasă de puncte din pătratul unitate. Ne întrebăm dacă nu cumva acest șir de curbe converge la o curbă care trece prin toate punctele pătratului. Răspunsul este pozitiv și a fost dat în anul 1904 de matematicianul francez Henri Lebesgue (1875 - 1941), utilizând o parametrizare analitică a curbei limită. Demonstrația fiind laborioasă și greu de urmărit, în secțiunea următoare vom prezenta pe larg un alt exemplu de astfel de curbă, care, deși este mai dificil de desenat, are o demonstrație a convergenței șirului de curbe iterate mai ușor de înțeles!

## 2. Curba lui Hilbert. Definiție

Primul exemplu de *curbă care umple un pătrat* (mai precis, o aplicație continuă și surjectivă  $z : [0, 1] \rightarrow [0, 1] \times [0, 1]$ ), a fost dat în 1890 de Giuseppe Peano, printr-o construcție analitică (pe baza scrierii numerelor în baza trei), fără nici un caracter intuitiv geometric. Acest exemplu, care contrariază intuiția comună, a pus problema fundamentării riguroase a noțiunii de *dimensiune* a unei figuri geometrice.

Mai târziu mulți alți matematicieni au găsit astfel exemple, dintre ei David Hilbert fiind primul care a dat, în 1891, o modalitate geometrică de generare a unei astfel de curbe. Alte exemple au fost date, printre alții, de E. H. Moore în 1900, Henri Lebesgue în 1904 și Waclaw Sierpinski în 1912. Astăzi multe dintre

aceste curbe poartă denumirea de *curbă a lui Peano*, un termen generic pentru *curbă care umple planul*.

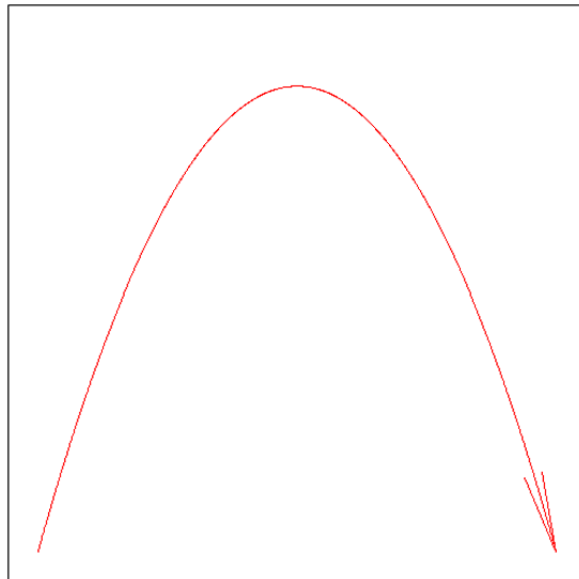


FIGURA 5. Curba lui Hilbert. Figura inițială

În continuare vom prezenta *curba lui Hilbert*, pe care o vom construi, pentru început, prin metoda transformărilor geometrice iterate.

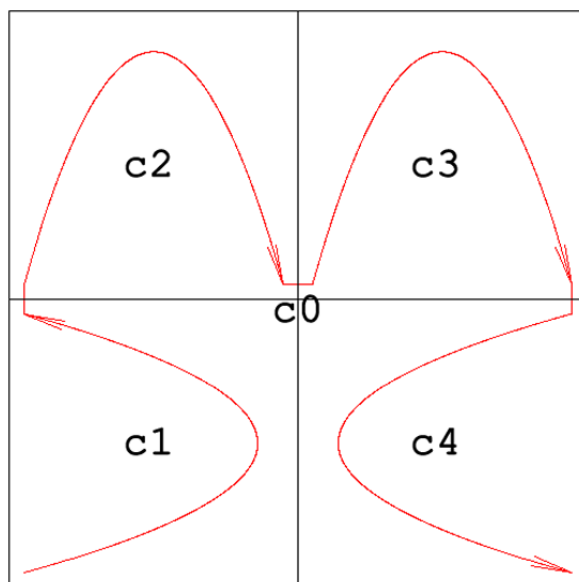


FIGURA 6. Curba lui Hilbert. Figura transformată



Să presupunem că la un moment dat avem trasată o curbă oarecare în pătratul unitate, care pleacă din vecinătatea colțului din stânga – jos și ajunge în preajma colțului din dreapta – jos (vezi Figura 5). Micșorăm întreaga figură la scara  $1/2$ , o rotim, o oglindim dacă e nevoie și o translatăm de patru ori, poziționând ca în Figura 6 pătrățelele obținute, și racordăm cele patru arce astfel încât să obținem o nouă curbă care pleacă din vecinătatea colțului din stânga – jos și ajunge în preajma colțului din dreapta – jos. Repetăm apoi la nesfârșit aceste operații,

obținând astfel un șir de curbe despre care David Hilbert a arătat că, indiferent de curba inițială, șirul converge la o curbă continuă unic determinată care trece prin toate punctele pătratului unitate, curbă care astăzi îi poartă numele.

Un program Python care să implementeze metoda descrisă poate fi obținută imediat din funcția `Z_order()` prezentată mai sus, actualizând numai definițiile și ordinea de aplicare a transformărilor geometrice `s1`, `s2`, `s3` și `s4`:

```
def Hilbert():
    c0 = (1 + 1j) / 2
    c1 = (1 + 1j) / 4
    c2 = (1 + 3j) / 4
    c3 = (3 + 3j) / 4
    c4 = (3 + 1j) / 4

    def s1(z):
        return c1 + 0.5j * (z - c0).conjugate()

    def s2(z):
        return c2 + 0.5 * (z - c0)

    def s3(z):
        return c3 + 0.5 * (z - c0)

    def s4(z):
        return c4 - 0.5j * (z - c0).conjugate()

    def transforma(li):
        rez = []
        for z in li:
            rez.append(s1(z))
        for z in li:
            rez.append(s2(z))
        for z in li:
            rez.append(s3(z))
        for z in li:
            rez.append(s4(z))
        return rez
```

```

def traseaza(li):
    C.fillScreen()
    # trasam chenarul
    C.drawNgon([0, 1, 1 + 1j, 1j], Color.Index(500))
    # desenam curba curenta
    for n in range(1, len(li)):
        C.drawLine(li[n - 1], li[n], Color.Index(n // 5))

C.setXminXmaxYminYmax(-0.1, 1.1, -0.1, 1.1)
fig = [c0]
for k in range(5):
    fig = transforma(fig)
    traseaza(fig)
    if C.mustClose(): return
    C.wait(100)

```

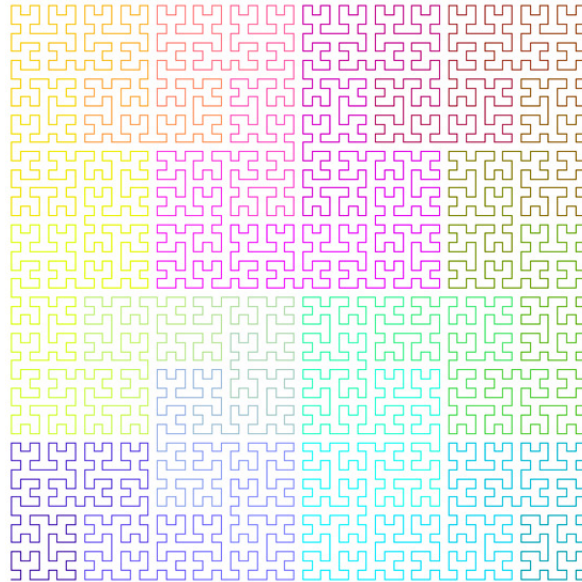


FIGURA 7. Curba lui Hilbert. Forma clasică

În cazul în care curba inițială se reduce la un singur punct, centrul pătratului, se obțin iteratele clasice ale curbei lui Hilbert reprezentate în Figura 7.

### §3. Curba lui Hilbert. Proprietăți

Pentru a da o demonstrație riguroasă a existenței curbei lui Hilbert, avem nevoie de un alt mod de construcție, și anume prin metoda motivelor iterate. După cum știm metoda constă în înlocuirea, la fiecare iterație, a unei figuri, *baza*, cu o altă figură, *motivul*, în care să se regăsească într-o formă sau alta, eventual de mai multe ori, baza – pentru reluarea iterării.

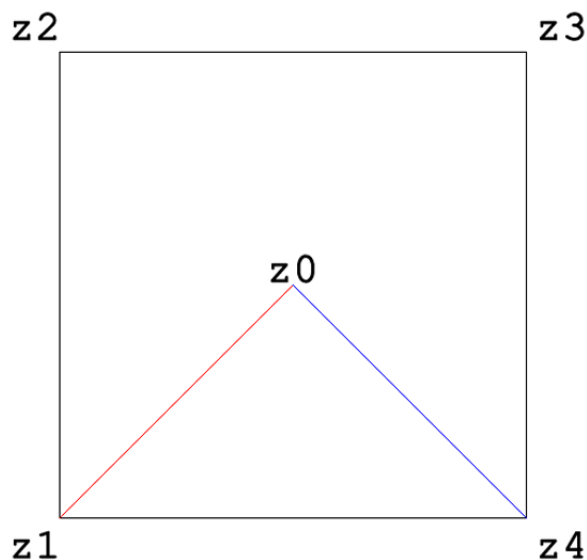


FIGURA 8. HilbertMotiveIterate. Baza ( $k=0$ ).

În cazul nostru, considerăm baza formată dintr-un *pătrat orientat*  $z_1z_2z_3z_4$  (vezi Figura 8) care la fiecare etapă este înlocuit cu motivul format din 4 *pătrate* de latură înjumătățită, dispuse și parcurse așa cum reiese din Figura 9.

La fiecare etapă vom forma o listă de pătrate, orice pătrat  $z_1z_2z_3z_4$  fiind păstrat în listă numai prin trei puncte,  $z_1$ ,  $z_0$  și  $z_4$ , în această ordine, unde  $z_0$  este centrul pătratului. Evident că  $z_2$  și  $z_3$  pot fi regăsite prin calcul. Această metodă de stocare, pe lângă economia de memorie, are avantajul că pune în evidență și parcurgerea generată de segmentele  $z_1z_0$  și  $z_0z_4$  (roșu și albastru) din pătratul de bază, segmente care se racordează în mod automat la construirea motivului, formând parcurgerea de ordin următor.

Această înlănțuire a tripletelor succesive  $z_1z_0z_4$  și  $z'_1z'_0z'_4$ , dată de egalitatea  $z_4 = z'_1$ , permite ca în listă fiecare triplet nou să fie memorat numai prin două valori,  $z'_0$  și  $z'_4$ , noul punct inițial  $z'_1$  fiind precedentul punct final  $z_4$ .

Pentru fiecare  $k = 0, 1, 2, \dots$  obținem la etapa  $k$  o curbă  $\mathcal{C}_k$  sub formă de linie poligonală roșu – albastră, care pleacă din vârful  $z_1 = 0$  al pătratului unitate și ajunge în  $z_4 = 1$ . Pentru a arăta că acest șir de curbe definește o curbă limită, trebuie mai întâi să precizăm ce *reprezentări parametrice* avem în vedere pentru

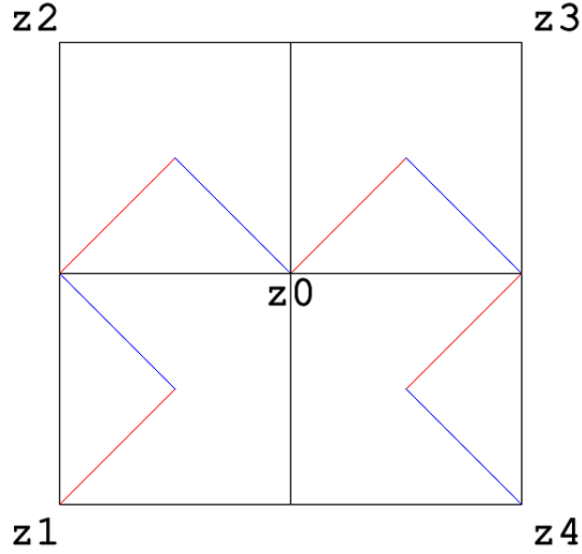


FIGURA 9. HilbertMotiveIterate. Motivul (k=1).

curbele  $\mathcal{C}_k$ , adică să precizăm un șir de aplicații continue

$$z_k : [0, T] \rightarrow \mathbb{C}, \quad k = 0, 1, 2, \dots,$$

astfel încât, pentru fiecare  $k$ , *punctul curent*  $z_k(t)$  să parcurgă complet curba  $\mathcal{C}_k$  pentru  $t \in [0, T]$ , unde  $T > 0$  este fixat arbitrar. Cum orice curbă  $\mathcal{C}_k$  poate fi parcursă în mai multe feluri, va trebui să alegem pentru fiecare câte o anumită parcurgere  $z_k$ .

```
def HilbertMotiveIterate():
    def transforma(li):
        z1 = li[0]
        rez = [z1]
        for k in range(1, len(li), 2):
            z0 = li[k]
            z4 = li[k + 1]
            z3 = 2 * z0 - z1
            z2 = 2 * z0 - z4
            rez.append((z1 + z0) / 2)
            rez.append((z1 + z2) / 2)
            rez.append((z0 + z2) / 2)
            rez.append(z0)
            rez.append((z0 + z3) / 2)
            rez.append((z3 + z4) / 2)
            rez.append((z0 + z4) / 2)
            rez.append(z4)
            z1 = z4
        return rez
```

```

def traseazaPatrate(li):
    C.fillScreen()
    z1 = li[0]
    for k in range(1, len(li), 2):
        z0 = li[k]
        z4 = li[k + 1]
        z3 = 2 * z0 - z1
        z2 = 2 * z0 - z4
        C.drawNgon([z1, z2, z3, z4], Color.Black)
        C.drawLine(z1, z0, Color.Red)
        C.drawLine(z0, z4, Color.Blue)
        z1 = z4
    C.refreshScreen()

C.setXminXmaxYminYmax(-0.25, 1.25, -0.25, 1.25)
C.fillScreen()
fig = [0, 0.5 + 0.5j, 1]
traseazaPatrate(fig)
for k in range(3):
    fig = transforma(fig)
    traseazaPatrate(fig)
    # traseazaLinii(fig)
C.wait(500)

```

Curba limită va fi dată atunci de *funcția limită punctuală*, dacă există, a șirului de funcții  $(z_k)_k$ , adică de funcția  $z : [0, T] \rightarrow \mathbb{C}$  definită de egalitatea

$$z(t) = \lim_{k \rightarrow \infty} z_k(t),$$

pentru fiecare  $t \in [0, T]$ .

Pentru a arăta că funcția limită definește o curbă  $\mathcal{C}$ , va trebui să demonstrăm că este funcție continuă, și din acest motiv vom căuta ca șirul de funcții  $(z_k)_k$  să fie *uniform convergent* la  $z$ , adică să avem

$$\lim_{k \rightarrow \infty} \sup_{t \in [0, T]} |z_n(t) - z(t)| = 0.$$

Se știe că dacă toți termenii unui șir uniform convergent sunt funcții continue, limita sa este continuă.

Pentru a fixa modul de parcurgere al curbelor  $\mathcal{C}_k$ , reanalizăm acum metoda lor de construcție. Observăm că fiecare aplicare a motivului mărește de 4 ori numărul pătratelor implicate și le înjumătățește lungimea laturii, iar o dată cu aceasta numărul de segmente roșii și albastre se mărește de patru ori dar cu lungimile micșorate de două ori, dublându-și astfel lungimea totală. Deducem imediat că la fiecare etapă  $k$  avem, înainte de aplicarea motivului, o rețea formată din  $4^k$  pătrate  $\mathcal{P}_k$  de latură  $l_k = 1/2^k$ , în fiecare pătrat curba  $\mathcal{C}_k$  având exact două segmente, ea fiind prin urmare formată din  $N_k = 2 \cdot 4^k$  segmente cu lungimea totală  $L_k = 2^k L_0 = 2^k \sqrt{2}$ .

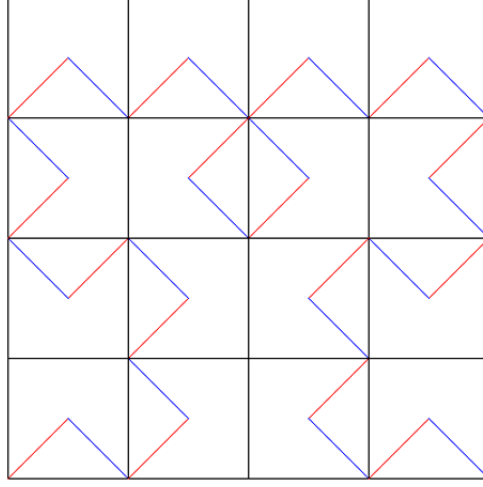


FIGURA 10. HilbertMotiveIterate. Parcurgerea de ordin  $k=2$

Deoarece fiecare curbă  $\mathcal{C}_k$  este o linie poligonală cu un număr finit de laturi  $\ell_k^h$ ,  $h = 1, 2, \dots, N_k$ , având fiecare lungime finită, putem considera parcurgerile cu viteză  $v_k$  constantă în modul, fiecare latură  $\ell_k^h$  fiind parcursă de punctul curent  $z_k(t)$  printr-o *mişcare rectilinie uniformă*, dată de o lege orară de forma

$$z_k(t) = \omega_k^h t + \zeta_k^h,$$

cu  $|\omega_k^h| = v_k = 2^k \sqrt{2}/T$ , pentru orice  $h = 1, 2, \dots, N_k$ . Constantele complexe  $\omega_k^h$  și  $\zeta_k^h$  se determina în mod unic din condițiile de racordare, din aproape în aproape, astfel încât funcția  $z_k : [0, T] \rightarrow \mathbb{C}$  să fie continuă și să parcurgă linia poligonală  $\mathcal{C}_k$ .

**Teoremă.** *Șirul de funcții  $z_n : [0, T] \rightarrow \mathbb{C}$ , definit mai sus, este uniform convergent pe  $[0, T]$  la o funcție continuă  $z : [0, T] \rightarrow \mathbb{C}$ , numită curba lui Hilbert.*

**Demonstrație.** Din modul de construcție rezultă că fiecare pătrat  $\mathcal{P}_k$  are un colț de intrare și unul de ieșire pe unde intră în pătrat, respectiv ies, toate parcurgerile  $z_n$  cu  $n \geq k$ . Punctul  $z_{k+1}(t)$  se mișcă de două ori mai repede decât  $z_k(t)$ , iar în interiorul fiecărui pătrat  $\mathcal{P}_k$  parcurge o lungime dublă față de  $z_k(t)$ , rezultă că cele două puncte traversează pătratul  $\mathcal{P}_k$  în tot atâtea unități de timp. Deoarece toate parcurgerile pornesc simultan din același punct  $z = 0$  și ajung simultan în același punct  $z = 1$ , urmează că ele sunt *sincronizate* astfel încât în fiecare pătrat  $\mathcal{P}_k$  punctele curente  $z_k(t)$  și  $z_{k+1}(t)$  intră în același moment  $\tau_k$  prin colțul său de intrare, parcurg apoi separat două, respectiv opt segmente din liniile lor poligonale, și părăsesc pătratul de bază  $\mathcal{P}_k$  tot simultan, prin colțul său de ieșire. În tot acest timp punctele  $z_k(t)$  și  $z_{k+1}(t)$  se găsesc în interiorul

pătratului  $\mathcal{P}_k$  și prin urmare distanța dintre ele este mai mică decât diagonala acestuia. Am arătat astfel că

$$|z_{k+1}(t) - z_k(t)| \leq l_k \sqrt{2} = \frac{\sqrt{2}}{2^k},$$

pentru orice  $t \in [0, T]$ .

Deoarece seria numerică

$$\sum_{k=0}^{\infty} \frac{\sqrt{2}}{2^k}$$

este convergentă, din *criteriul lui Weierstrass* urmează că seria de funcții

$$z_0(t) + \sum_{k=0}^{\infty} (z_{k+1}(t) - z_k(t))$$

este uniform convergentă pe  $[0, T]$ , și o dată cu ea este uniform convergent șirul sumelor sale parțiale

$$z_0(t) + (z_1(t) - z_0(t)) + \cdots + (z_n(t) - z_{n-1}(t)) = z_n(t).$$

□

Până acum am arătat astfel că șirul de funcții continue  $(z_k)_k$  este uniform convergent pe  $[0, T]$  la o funcție continuă  $z$  care definește astfel o curbă  $\mathcal{C}$ . Mai rămâne de arătat că imaginea funcției  $z$  este tot pătratul unitate.

**Teoremă.** *Curba lui Hilbert trece prin toate punctele pătratului  $[0, 1] \times [0, 1]$ .*

**Demonstrație.** Din proprietatea de sincronizare descrisă mai sus știm că fiecare pătrat  $\mathcal{P}_k$  are un colț prin care intră în pătrat, la același moment  $\tau_k$ , toate parcurgerile de ordin  $n \geq k$ . Deducem că șirul  $(z_n(\tau_k))_n$  este constant de la  $k$  încolo, și deci

$$z(\tau_k) = \lim_{n \rightarrow \infty} z_n(\tau_k) = z_k(\tau_k) \in \mathcal{P}_k.$$

Am arătat astfel că  $\mathcal{C}$  trece prin orice pătrat  $\mathcal{P}_k$ , pentru orice  $k \geq 0$ .

Fixăm acum în mod arbitrar un punct  $z^* \in [0, 1] \times [0, 1]$  și ne propunem să dovedim existența unui  $t^* \in [0, T]$  astfel încât  $z(t^*) = z^*$ . Este evident că, pentru orice ordin  $k \geq 0$ , punctul  $z^*$  se află cel puțin în interiorul sau pe laturile unui pătrat de ordin  $k$ , notat în continuare  $\mathcal{P}_k^*$ . Din cele arătate mai sus, știm că pentru  $\mathcal{P}_k^*$  există un moment  $\tau_k^* \in [0, T]$  astfel încât  $z(\tau_k^*) \in \mathcal{P}_k^*$ , de aici urmează ca distanța dintre punctele  $z^*$  și  $z(\tau_k^*)$  este mai mică decât diagonala pătratului  $\mathcal{P}_k^*$ , care le conține. Așadar

$$|z(\tau_k^*) - z^*| \leq \frac{\sqrt{2}}{2^k},$$

pentru orice  $k \geq 0$ , de unde rezultă

$$\lim_{k \rightarrow \infty} z(\tau_k^*) = z^*.$$

Șirul  $(\tau_k^*)_k$  fiind mărginit, are cel puțin un subșir  $(\tau_{k_n}^*)_n$  convergent la un  $t^*$  din intervalul  $[0, T]$ . Folosim acum continuitatea funcției limită  $z$  și demonstrăm egalitatea dorită:

$$z(\tau^*) = z\left(\lim_{n \rightarrow \infty} \tau_{k_n}^*\right) = \lim_{n \rightarrow \infty} z(\tau_{k_n}^*) = z^*.$$

□

În concluzie, am arătat că șirul iteratelor  $\mathcal{C}_k$  converge într-un sens bine precizat la o curbă continuă  $\mathcal{C}$  care trece prin toate punctele pătratului  $[0, 1] \times [0, 1]$ .

Este ușor de văzut că, dacă în etapa inițială, în locul curbei  $\mathcal{C}_0$  formată din două segmente, considerăm orice altă curbă *de lungime finită*,  $0 < L_0 < +\infty$ , care pleacă din  $z = 0$  și ajunge în  $z = 1$ , șirul iteratelor  $\mathcal{C}_k$  va tinde la aceeași limită  $\mathcal{C}$ , deoarece aceasta este unic determinată de ordinea de trecere prin punctele de intrare/ieșire ale pătratelor  $\mathcal{P}_k$ , ordine care nu depinde de curba inițială.

Desenarea prin degradeuri de culoare a curbei  $\mathcal{C}$  – *curba lui Hilbert* – prezentată în Figura 11, a fost obținută printr-o ușoară modificare a codului funcției `HilbertMotiveIterate()`.

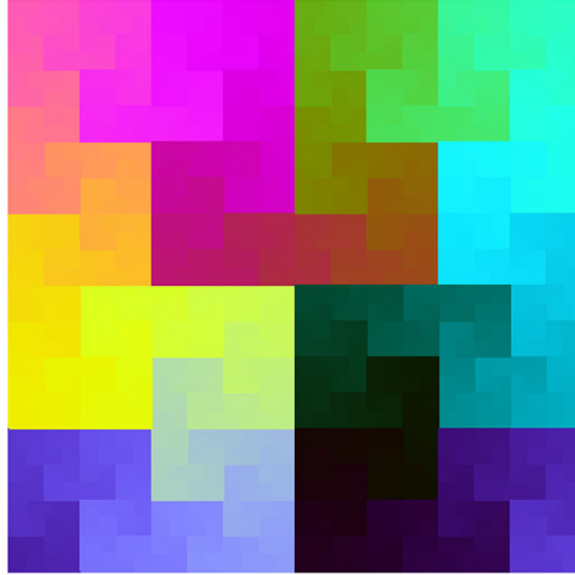


FIGURA 11. HilbertMotiveIterate. Parcurgerea de ordin  $k=10$