Cursul 11

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 "spaţial" ca un tablou dreptunghiular de elemente dispuse pe m linii şi n coloane, fiecare element fiind accesat prin precizarea a doi indici, x – indicele de coloană, abscisa, şi y – indicele de linie, ordonata. Se pune problema *serializării* acestei bazei de date, a stocării ei secvenţiale, într-un fisier pe disc, de exemplu.

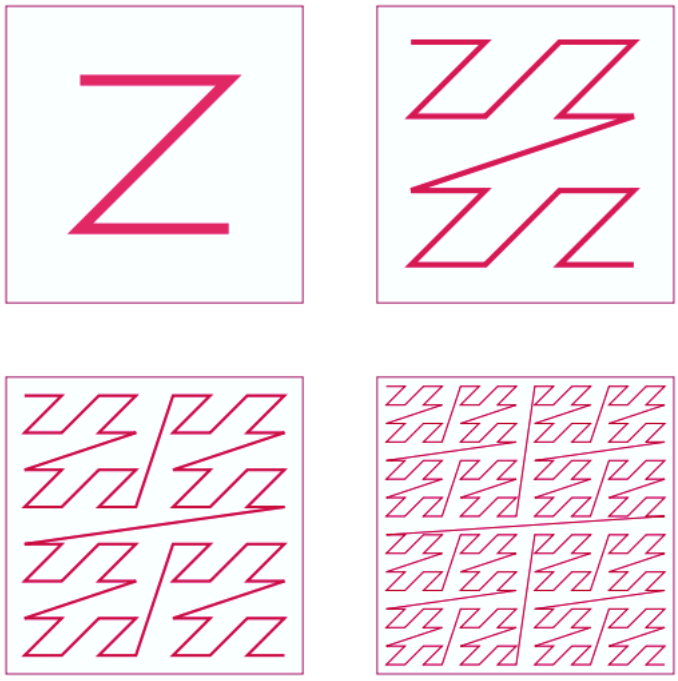

FIGURA 1. Parcurgerea în Z-ordine.

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

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

şi mulţimea $\{0, 1, \ldots, mn-1\}$.

Soluţia cea mai simplă şi cea mai des folosită este *parcurgerea pe linii*, de la stânga la dreapta şi de sus în jos. Dacă notăm cu N *rangul* lui (x, y) în înşiruirea

$$(0,0),(1,0),\ldots,(n-1,0),(0,1),(1,1),\ldots,(0,m-1),(1,m-1),\ldots,(n-1,m-1)$$

atunci avem relaţia

$$N = x + ny$$

şi reciproc,

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

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

Aceste două impedimente sunt înlăturate, măcar parţial, de *parcurgerea în Z-ordine*, care este, în esenţă, forma bidimensională a algoritmului de căutare binară. Impărţim matricea în două benzi orizontale şi două verticale, obţinem 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 totuşi în algoritmul de determinare a corespondenţei

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

sugerat de Figura 2, şi anume: utilizând scrierea numerelor în baza 2, biţii lui N se obţin prin intercalarea biţilor lui y cu ai lui x, aşezând fiecare bit din y în faţa bitului corespunzător lui în x. Justificarea este foarte simplă: la împărtirea iniţială în două benzi orizontale şi două benzi vericale, 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, aşa că numărarea pe doi biţi

$$00, 01, 10, 11$$

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

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

In clasa $C^{\#}$ următoare aplicăm algoritmul de mai sus ca să parcurgem în Z-ordine o reţea 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 $\tilde{x}$ şi $\tilde{y}$ din $\{0, 1, \ldots, 2^k - 1\}$.

```
public class Z_orderBin : ComplexForm
{
    Complex i = new Complex(0, 1);

    bool incrementare(byte[] regBin)
    {   //trecem pe 01000111
        //in        01001000
        //prin adunarea cu 1
        bool avemTransport = true;
        for (int k = 0; k < regBin.Length && avemTransport; k++)
        {
            if (regBin[k] == 1)
            {
                regBin[k] = 0;
            }
            else
            {
                regBin[k] = 1;
                avemTransport = false;
            }
        }
        return !avemTransport;
    }
```

```
    void traseaza(List<Complex> li)
    {
        for (int n = 1; n < li.Count; n++)
        {
            setLine(li[n - 1], li[n], getColor(n / 5));
        }
        resetScreen();
    }
    public override void makeImage()
    {
        setXminXmaxYminYmax(-0.1, 1.1, 1.1, -0.1);
        int k = 5;
        byte[] reg = new byte[2 * k];
        for (int h = 0; h < reg.Length; h++)
        {
            reg[h] = 0;
        }
        List<Complex> fig = new List<Complex>() { 0 };

        while (incrementare(reg))
        {
            //aflam x si y
            double y = 0;
            double p = 1;
            for (int h = 2 * k - 1; h > 0; h -= 2)
            {
                p *= 2;
                y += (double)reg[h] / p;
            }
            double x = 0;
            p = 1;
            for (int h = 2 * k - 2; h >= 0; h -= 2)
            {
                p *= 2;
                x += (double)reg[h] / p;
            }
            fig.Add(x + i * y);
        }
        traseaza(fig);
    }
}
```

In program generăm scrierea binară a rangului N direct într-un registru pe $2k$ biţi, în vectorul

```
byte[] reg = new byte[2 * k];
```

pe care îl iniţializăm cu zero şi apoi îl incrementăm cu metoda

```
bool incrementare(byte[] regBin)
```

până când obţinem depăşire de format. La fiecare pas obţinem coordonatele x şi y ale punctului curent z prin dez-intercalarea biţilor lui N şi memorăm punctul z în lista `fig`. In final desenăm traseul păstrat în listă.

O clasă care implementează algoritmul de intercalare binară de mai sus utilizând direct operaţii pe biţi este următoarea:

```
public class Z_orderPeBiti : ComplexForm
{
    public override void makeImage()
    {
        int k = 6; // atentie: 0 < k < 15
        uint nrPuncte = 1u << (2 * k);
        double lat = 1u << k;
        setXminXmaxYminYmax(-1, lat, lat, -1);
        Complex znou, zvechi = 0;
        for (uint n = 0; n < nrPuncte; n++)
        {
            uint x = 0, y = 0, nn = n;
            for (int h = 0; h < k; h++)
            {
                x |= (nn & 1u) << h;
                nn >>= 1;
                y |= (nn & 1u) << h;
                nn >>= 1;
            }
            znou = Complex.setReIm(x, y);
            setLine(znou, zvechi, getColor((int)n / 10));
            zvechi = znou;
            if (!resetScreen()) return;
        }
    }
}
```

Observaţie: pentru ca sensul de creştere a lui y pe ecran să fie în jos (ca la matrice) în apelul

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

am fixat `ymin` mai mare decât `ymax`!

§2. Metoda transformărilor iterate

Vom trasa acum curba dată de parcurgerea în Z ordine pe baza următoarei proprietăţi de autosimilaritate observate în Figura 3: pentru fiecare k, în etapa de ordin k avem în pătratul unitate o parcurgere din colţul din stânga – sus până în colţul din dreapta – jos. Dacă reducem totul la scara $1/2$ cu centru în `c0` – centrul pătratului iniţial –, translatăm de patru ori "pătrăţelul" obţinut asfel încât `c0` să ajungă pe rând în `c2`, `c3`, `c1` şi `c4` – în această ordine – şi racordăm în formă de Z cele patru parcurgeri, vom obţine curba de ordin $k+1$.

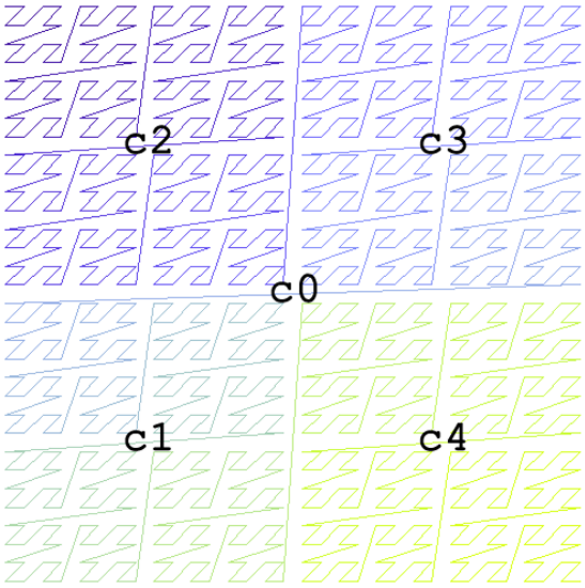

FIGURA 3. `class Z_orderBin`

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 `s1`, `s2`, `s3` şi `s4`.

```
public class Z_order : ComplexForm
{
    static Complex i = new Complex(0, 1);
    static Complex c0 = (1 + i) / 2;
    static Complex c1 = (1 + i) / 4;
    static Complex c2 = (1 + 3 * i) / 4;
    static Complex c3 = (3 + 3 * i) / 4;
    static Complex c4 = (3 + i) / 4;
    //sk(z)=c0+0.5*(z-c0) +(ck-c0)=ck+0.5*(z-c0)
    Complex s1(Complex z) { return c1 + 0.5 * (z - c0); }
    Complex s2(Complex z) { return c2 + 0.5 * (z - c0); }
    Complex s3(Complex z) { return c3 + 0.5 * (z - c0); }
    Complex s4(Complex z) { return c4 + 0.5 * (z - c0); }
    void transforma(ref List<Complex> li)
    {
        List<Complex> rez = new List<Complex>();
        foreach (Complex z in li)
            rez.Add(s2(z));
        foreach (Complex z in li)
            rez.Add(s3(z));
        foreach (Complex z in li)
            rez.Add(s1(z));
        foreach (Complex z in li)
```

```
            rez.Add(s4(z));
        li = rez;
    }
    void traseaza(List<Complex> li)
    {
        initScreen();
        //trasam chenarul
        Color col = getColor(0);
        setLine(0.0, 1.0, col);
        setLine(1.0, 1 + i, col);
        setLine(1 + i, i, col);
        setLine(i, 0.0, col);
        //desenam curba curenta
        for (int n = 1; n < li.Count; n++)
            setLine(li[n - 1], li[n], getColor(n / 5));

    }
    public override void makeImage()
    {
        setXminXmaxYminYmax(-0.1, 1.1, -0.1, 1.1);
        List<Complex> fig = new List<Complex>();
        fig.Add(c0);
        for (int k = 0; k < 5; k++)
        {
            transforma(ref fig);
            traseaza(fig);
            if (!resetScreen()) return;
            delaySec(1);
        }
    }
}
```

Dacă în figura iniţială avem numai punctul z_0, vom obţine la fiecare etapă exact curbele generate de programul precedent

Observăm că, pentru k din ce în ce mai mare, curbele noastre trec printr-o reţea din ce în ce mai deasă de puncte din pătratul unitate. Ne întrebăm dacă nu cumva acest şir de curbe converge la o curbă care trece prin toate punctele pătratului. Răspunsul este pozitiv şi a fost dat în anul 1904 de matematicianul francez Henri Lebesgue (1875 - 1941), utilizând o parametrizare analitică a curbei limită. Demonstraţia fiind laborioasă şi greu de urmărit, în secţiunea următoare vom prezenta pe larg un alt exemplu de astfel de curbă, care, deşi este mai dificil de desenat, are o demonstraţie a convergenţei şirului de curbe iterate mai uşor de înţeles!

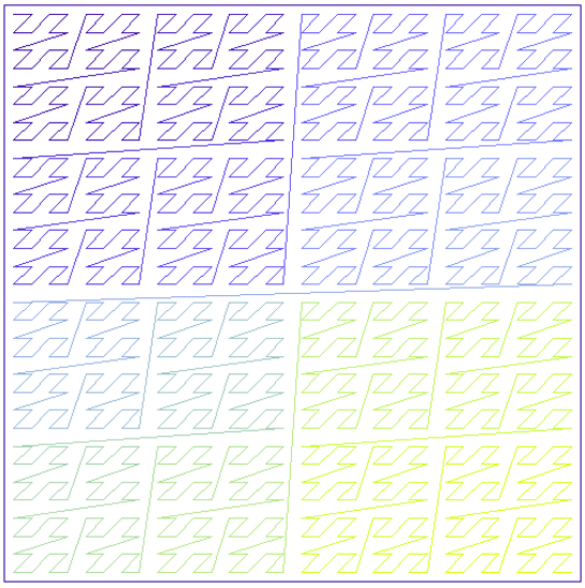

FIGURA 4. `class Z_order`

2. Curba lui Hilbert. Definiţie

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

Mai târziu mulţi alţi 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 alţii, de E. H. Moore în 1900, Henri Lebesgue în 1904 şi Waclaw Sierpinski în 1912. Astăzi multe dintre aceste curbe poartă denumirea de *curbă a lui Peano*, un termen generic pentru *curbă care umple planul*.

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

Să presupunem că la un moment dat avem trasată o curbă oarecare în pătratul unitate, care pleacă din vecinătatea colţului din stânga – jos şi ajunge în preajma colţului din dreapta – jos (vezi Figura 5). Micşorăm întreaga figură la scara 1/2, o rotim, o oglindim dacă e nevoie şi o translatăm de patru ori, poziţionând ca în Figura 6 pătrăţelele obţinute, şi racordăm cele patru arce astfel încât să obţinem o nouă curbă care pleacă din vecinătatea colţului din stânga – jos şi ajunge în preajma colţului din dreapta – jos. Repetăm apoi la nesfârşit aceste operaţii, obţinând astfel un şir de curbe despre care David Hilbert a arătat că, indiferent

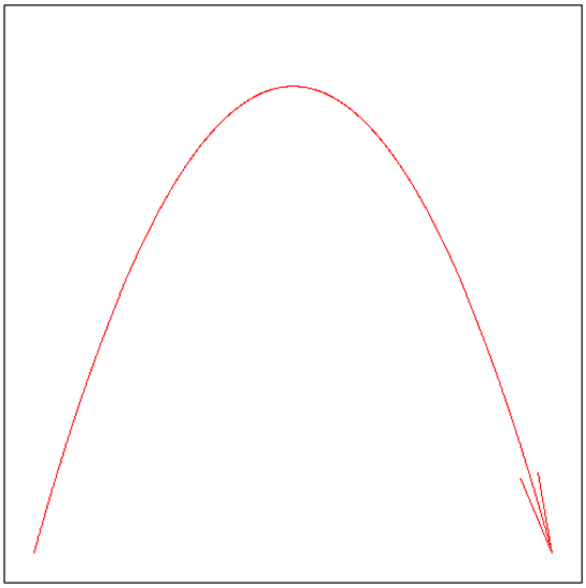

FIGURA 5. `Curba lui Hilbert. Figura iniţială`

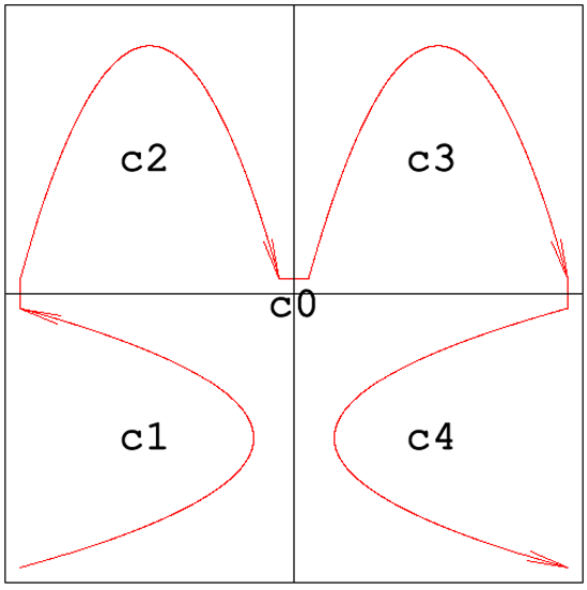

FIGURA 6. `Curba lui Hilbert. Figura transformată`

de curba iniţială, şirul converge la o curbă continuă unic determinată care trece prin toate punctele pătratului unitate, curbă care astăzi îi poartă numele.

O clasă $C^{\#}$ care să implementeaze metoda descrisă poate fi obţinută imediat din clasa `Z_order` prezentată mai sus, actualizând numai definiţiile şi ordinea de aplicare a transformărilor geometrice `s1`, `s2`, `s3` şi `s4`:

```
static Complex i = new Complex(0.0, 1.0);
Complex c0 = (1 + i) / 2;
Complex c1 = (1 + i) / 4;
Complex c2 = (1 + 3 * i) / 4;
Complex c3 = (3 + 3 * i) / 4;
Complex c4 = (3 + i) / 4;

Complex s1(Complex z)//
{
    return c1 + 0.5 * i * (z - c0).conj;
}
//rez=zz0+omega*(z-zz0) +(zzk-zz0)=zzk+omega*(z-zz0)
Complex s2(Complex z)
{
    return c2 + 0.5 * (z - c0);
}
Complex s3(Complex z)
{
    return c3 + 0.5 * (z - c0);
}
Complex s4(Complex z)
{
    return c4 - 0.5 * i * (z - c0).conj;
}
```

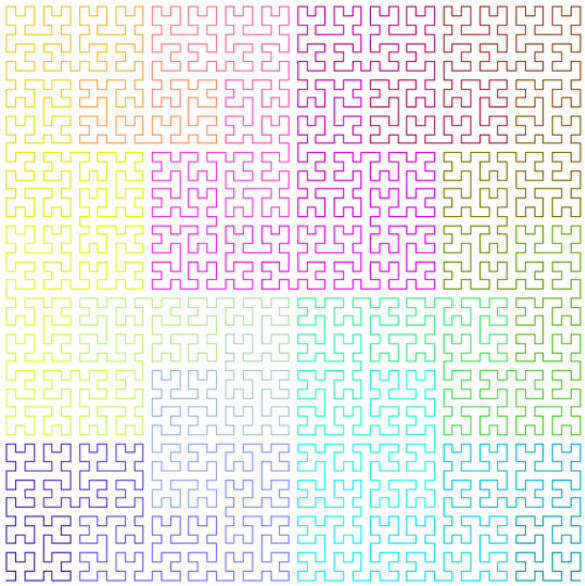

FIGURA 7. Curba lui Hilbert. Forma clasică

In cazul în care curba iniţială se reduce la un singur punct, centrul pătratului, se obţin iteratele clasice ale curbei lui Hilbert reprezentate în Figura 7.

§3. Curba lui Hilbert. Proprietăţi

Pentru a da o demonstraţie riguroasă a existenţei curbei lui Hilbert, avem nevoie de un alt mod de construcţie, şi anume prin metoda motivelor iterate. După cum ştim metoda constă în înlocuirea, la fiecare iteraţie, 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.

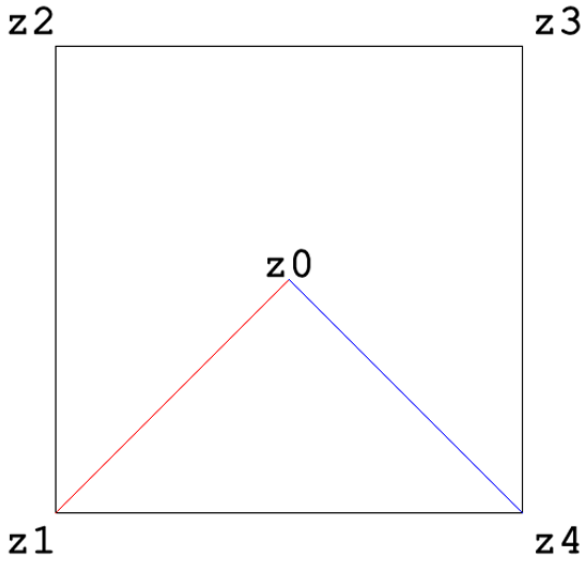

FIGURA 8. `HilbertMotiveIterate. Baza (k=0).`

In 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 înjumătăţită, dispuse şi parcurse aşa cu 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 şi z_4, în această ordine, unde z_0 este centrul pătratului. Evident că z_2 şi z_3 pot fi regăsite prin calcul. Această metodă de stocare, pe lângă economia de memorie, are avantajul că pune în evidenţă şi parcurgerea generată de segmentele z_1z_0 şi z_0z_4 (roşu şi albastru) din pătratul de bază, segmente care se racordează în mod automat la construirea motivului, formând parcurgerea de ordin următor.

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

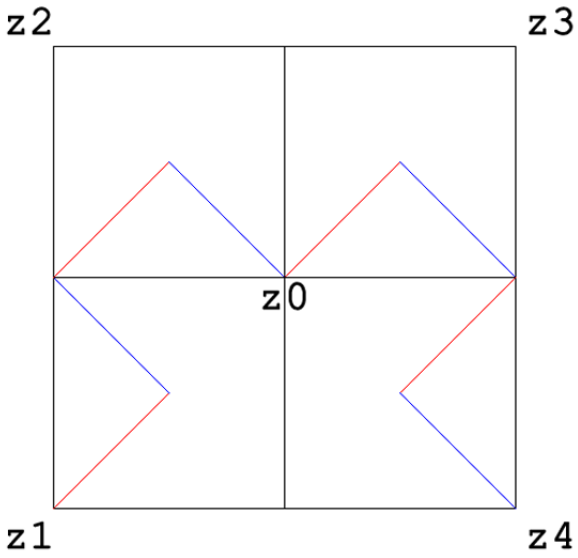

FIGURA 9. HilbertMotiveIterate. Motivul (k=1).

```
public class HilbertMotiveIterate : ComplexForm
{
    static Complex i = new Complex(0.0, 1.0);
    void transforma(ref List<Complex> li)
    {
        List<Complex> rez = new List<Complex>();
        Complex z0, z1, z2, z3, z4;
        z1 = li[0];
        rez.Add(z1);
        for (int k = 1; k < li.Count; k += 2)
        {
            z0 = li[k];
            z4 = li[k + 1];
            z3 = 2 * z0 - z1;
            z2 = 2 * z0 - z4;
            rez.Add((z1 + z0) / 2);
            rez.Add((z1 + z2) / 2);
            rez.Add((z0 + z2) / 2);
            rez.Add(z0);
            rez.Add((z0 + z3) / 2);
            rez.Add((z3 + z4) / 2);
            rez.Add((z0 + z4) / 2);
            rez.Add(z4);
            z1 = z4;
        }
        li = rez;
```

```
        }
        void traseazaPatrate(List<Complex> li)
        {
            Complex z0, z1, z2, z3, z4;
            initScreen();
            Color col = Color.Black;
            z1 = li[0];
            for (int k = 1; k < li.Count; k += 2)
            {
                z0 = li[k];
                z4 = li[k + 1];
                z3 = 2 * z0 - z1;
                z2 = 2 * z0 - z4;
                setLine(z1, z4, col);
                setLine(z1, z2, col);
                setLine(z2, z3, col);
                setLine(z3, z4, col);
                setLine(z1, z0, Color.Red);
                setLine(z0, z4, Color.Blue);
                z1 = z4;
            }
            resetScreen();
        }
        public override void makeImage()
        {
            setXminXmaxYminYmax(-0.25, 1.25, -0.25, 1.25);
            ScreenColor = Color.White;
            List<Complex> fig = new List<Complex>() { 0, 0.5 + 0.5 * i, 1 };
            traseazaPatrate(fig);
            for (int k = 0; k < 3; k++) {
                transforma(ref fig);
                traseazaPatrate(fig);
                delaySec(0.5);
            }
        }
    }
```

Pentru fiecare $k = 0, 1, 2 \ldots$ obţinem la etapa k o curbă $\mathcal{C}_k$ sub formă de linie poligonală roşu – albastră, care pleacă din vârful $z_1 = 0$ al pătratului unitate şi ajunge în $z_4 = 1$. Pentru a arăta că acest şir de curbe defineşte o curbă limită, trebuie mai întâi să precizăm ce *reprezentări parametrice* avem în vedere pentru curbele $\mathcal{C}_k$, adică să precizăm un şir de aplicaţii continue

$$z_k : [0, T] \to \mathbb{C}, \ \ k = 0, 1, 2, \ldots,$$

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.

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

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

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

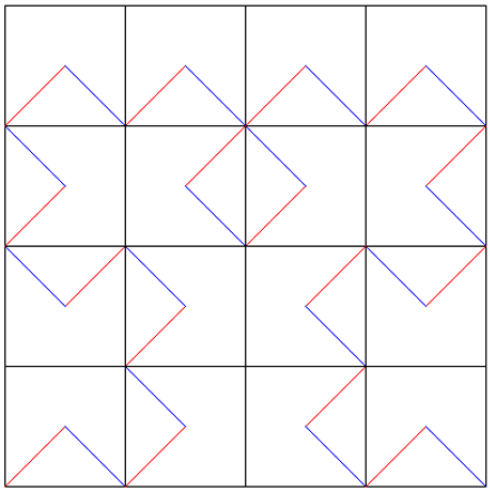

FIGURA 10. `HilbertMotiveIterate. Parcurgerea de ordin k=2`

Pentru arăta ca funcţia limită defineaşte o curbă $\mathcal{C}$, va trebui să demonstrăm că este funcţie continuă, şi din acest motiv vom căuta ca şirul de funcţii $(z_k)_k$ să fie *uniform converget* la z, adică să avem

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

Se ştie că dacă toţi termenii unui şir uniform convergent sunt funcţii continue, limita sa este continuă.

Pentru a fixa modul de parcurgere al curbelor $\mathcal{C}_k$, reanalizăm acum metoda lor de construcţie. Observăm că fiecare aplicare a motivului măreşte de 4 ori numărul pătratelor implicate şi le înjumătăţeşte lungimea laturii, iar o dată cu aceasta numărul de segmente roşii şi albastre se măreşte de patru ori dar cu lungimile micşorate de două ori, dublându-şi astfel lungimea totală. Deducem imediat că la fiecare etapă k avem, înainte de aplicarea motivului, o reţea 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 $L_k = 2^k L_0 = 2^k\sqrt{2}$.

Deoarece fiecare curbă $\mathcal{C}_k$ este o linie poligonală cu un număr finit de laturi ℓ_k^h, $h = 1, 2, \ldots, N_k$, având fiecare lungime finită, putem considera parcurgerile

cu viteză v_k constantă în modul, fiecare latură ℓ_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 $h = 1, 2, \ldots, N_k$. Constantele complexe ω_k^h şi ζ_k^h se determina în mod unic din condiţiile de racordare, din aproape în aproape, astfel încât funcţia $z_k : [0, T] \to \mathbb{C}$ să fie continuă şi să parcurgă linia poligonală $\mathcal{C}_k$.

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

Demonstraţie. Din modul de construcţie rezultă că fiecare pătrat $\mathcal{P}_k$ are un colţ de intrare şi unul de ieşire pe unde intră în pătrat, respectiv ies, toate parcurgerile z_n cu $n \geq k$. Punctul $z_{k+1}(t)$ se mişcă de două ori mai repede decât $z_k(t)$, iar în interiorul fiecărui pătrat $\mathcal{P}_k$ parcurge o lungime dublă faţă de $z_k(t)$, rezultă că cele două puncte traversează pătratul $\mathcal{P}_k$ în tot atâtea unităţi de timp. Deoarece toate parcurgerile pornesc simultan din acelaşi punct $z = 0$ şi ajung simultan în acelaşi punct $z = 1$, urmează că ele sunt *sincronizate* astfel încât în fiecare pătrat $\mathcal{P}_k$ punctele curente $z_k(t)$ şi $z_{k+1}(t)$ intră în acelaşi moment τ_k prin colţul său de intrare, parcurg apoi separat două, respectiv opt segmente din liniile lor poligonale, şi părăsesc pătratul de bază $\mathcal{P}_k$ tot simultan, prin colţul său de ieşire. In tot acest timp punctele $z_k(t)$ şi $z_{k+1}(t)$ se găsesc în interiorul pătratului $\mathcal{P}_k$ şi prin urmare distanţa 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 funcţii

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

este uniform convergentă pe $[0, T]$, şi o dată cu ea este uniform convergent şirul sumelor sale parţiale

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

□

Până acum am arătat astfel că şirul de funcţii continue $(z_k)_k$ este uniform convergent pe $[0, T]$ la o funcţie continuă z care defineşte astfel o curbă $\mathcal{C}$. Mai rămâne de arătat că imaginea funcţiei z este tot pătratului unitate.

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

Demonstraţie. Din proprietatea de sincronizare descrisă mai sus ştim că fiecare pătrat $\mathcal{P}_k$ are un colţ prin care intră în pătrat, la acelaşi moment τ_k, toate parcurgerile de ordin $n \geq k$. Deducem că şirul $(z_n(\tau_k))_n$ este constant de la k încolo, şi deci

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

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 $z^* \in [0,1] \times [0,1]$ şi ne propunem să dovedim existenţa unui $t^* \in [0,T]$ astfel încât $z(t^*) = z^*$. Este evident că, pentru orice ordin $k \geq 0$, punctul z^* se află cel puţin în interiorul sau pe laturile unui pătrat de ordin k, notat în continuare $\mathcal{P}_k^*$. Din cele arătate mai sus, ştim 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 distanţa dintre punctele z^* şi $z(\tau_k^*)$ este mai mică decât diagonala pătratului $\mathcal{P}_k^*$, care le conţine. Aşadar

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

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

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

Şirul $(\tau_k^*)_k$ fiind mărginit, are cel puţin un subşir $(\tau_{k_n}^*)_n$ convergent la un t^* din intervalul $[0,T]$. Folosim acum continuitatea funcţiei limită z şi demonstrăm egalitatea dorită:

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

□

In concluzie, am arătat că şirul 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 uşor de văzut că, dacă în etapa iniţială, în locul curbei $\mathcal{C}_0$ formată din două segmente, considerăm orice altă curbă *de lungime finită*, $0 < L_0 < +\infty$, care pleacă din $z = 0$ şi ajuge în $z = 1$, şirul iteratelor $\mathcal{C}_k$ va tinde la aceeaşi limită $\mathcal{C}$, deoarece aceasta este unic determinată de ordinea de trecere prin punctele de intrare/ieşire ale pătratelor $\mathcal{P}_k$, ordine care nu depinde de curba iniţială.

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

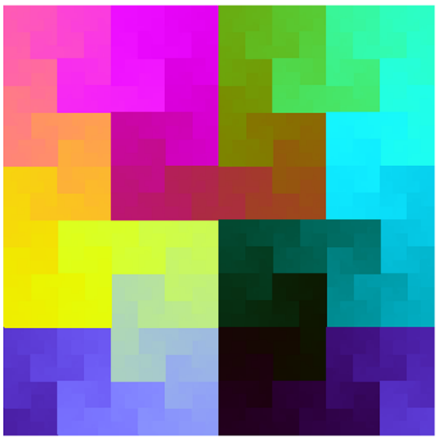

FIGURA 11. `HilbertMotiveIterate. Parcurgerea de ordin k=10`