

Recursão Revisitada

usando a `PLAYCB`

JOSÉ CARLOS LOUREIRO RALHA

1 Recursão Revisitada

Em documento anterior apresentamos alguns exemplos de problemas cuja solução foi obtida utilizando procedimentos recursivos. Esse foi o caso para problemas como

- *quicksort*
- *mergesort*
- *torres de hanói*

Neste módulo, vamos mostrar recursão através de exemplos cuja saída será na forma gráfica. Para isso vamos empregar a `PLAYCB`.



Nosso primeiro problema será traçar uma curva famosa denominada *curva de Koch* em homenagem ao seu propositor. A curva de Koch básica é construída tomando como base um segmento de reta. Vamos fazer o traçado usando uma caneta sem tirá-la do papel. Do ponto localizado na extremidade esquerda traçamos um segmento de reta de comprimento n paralelo ao eixo x . Nesse ponto mudamos a direção do traçado; vamos agora traçar um segmento de reta de mesmo comprimento mas em um ângulo de 60° . Na extremidade direita do segmento traçado voltamos a mudar a direção em 120° . A partir daí traçamos um segmento de mesmo comprimento. Por fim, mudamos de novo o ângulo, desta vez em 60° , e traçamos um novo segmento. A Fig. 1 mostra o padrão descrito.¹ Um ponto importante diz respeito aos ângulos: *a medida é tomada sobre a reta que acabamos de traçar*.

Se repetirmos o mesmo padrão em cada segmento de reta da Fig. 1 obtemos a curva de Koch de ordem 2, exibida na Fig. 2. Esse padrão pode ser repetido indefinidamente. No limite, *quando a ordem tende a infinito*, obtemos uma *curva fractal*. A Fig. 3 mostra o caso para a ordem 5.

¹Este tipo de traçado tem uma representação gráfica especial denominada *Turtle Graphics* e é a base da linguagem `LOGO`. A linguagem `LOGO` foi projetada como parte de uma experiência cognitiva relacionada a capacidade de aprendizagem de crianças de tenra idade.

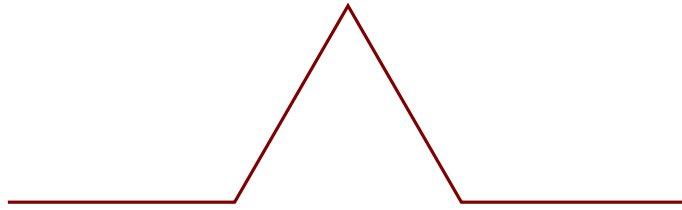


Figura 1: Curva de Koch de ordem 1.

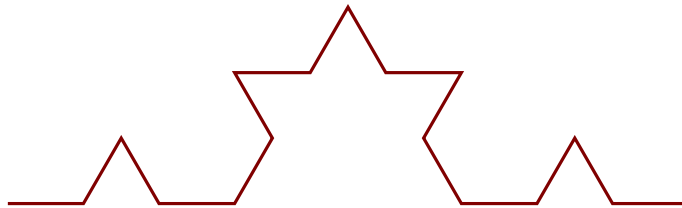


Figura 2: Curva de Koch de ordem 2.

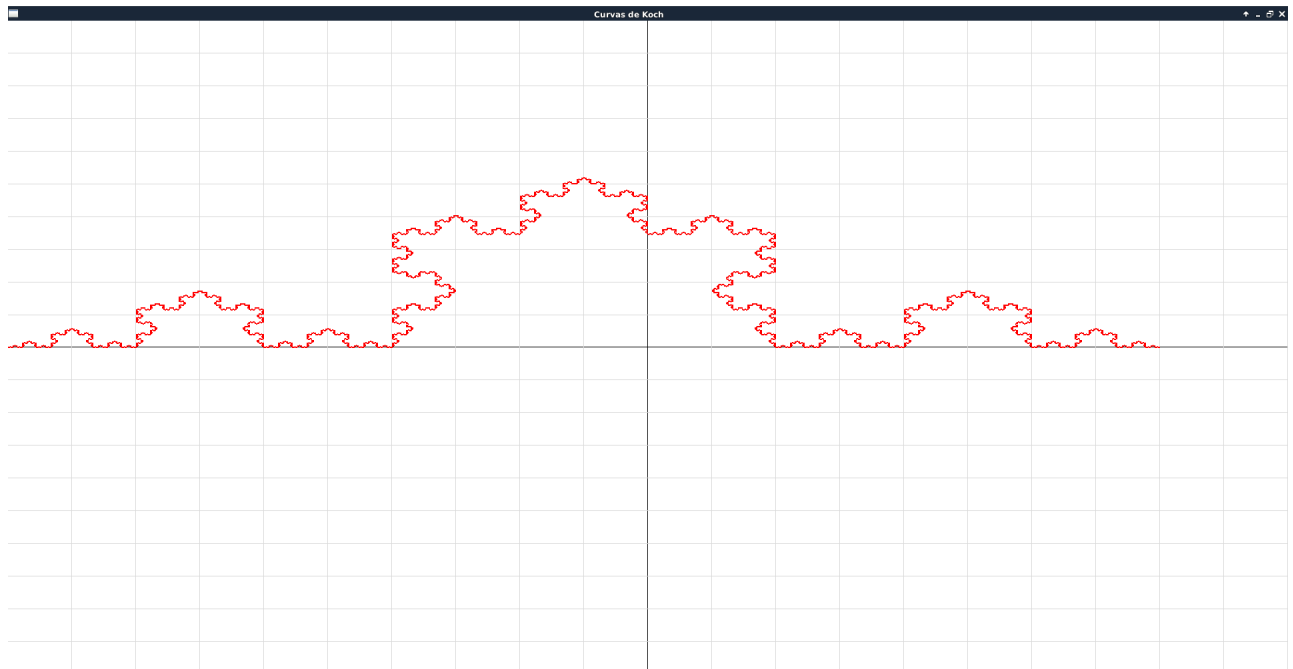


Figura 3: Curva de Koch de ordem 5. Tela produzida pelo programa `koch.c` usando a `PLAYCB`.

A repetição da aplicação do padrão de subdivisão é naturalmente realizado usando recursão. A Listagem 1 mostra o programa recursivo completo usando a `PLAYCB`.

Listagem 1: Programa `koch.c`

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <playcb.h>

#define LARGURA 1962//800
#define ALTURA 1280//600
#define SIZE 180

Ponto movaCaneta(Ponto from, double theta, double len) {
    Ponto to;
    double rad = theta*PI/180;
    to.x = from.x + len*cos(rad);
    to.y = from.y + len*sin(rad);
    return to;
}

void koch (Ponto from, double len, double theta, int order) {
    Ponto to;
    double rad = theta*PI/180;

    if (order==0) {
        to.x = from.x + len*cos(rad);
        to.y = from.y + len*sin(rad);
        CriaReta(from, to); Grafite(2); Pintar(255,0,0);
    }
    else {
        koch(from, len/3, theta, order-1);

        to = movaCaneta(from, theta, len/3);
        from = to;//dispensável.
        // seu uso é para tornar intuitiva a chamada abaixo
        koch(from, len/3, theta+60, order-1);

        to = movaCaneta(from, theta+60, len/3);
        from = to;
        koch(from, len/3, theta-60, order-1);

        to = movaCaneta(to, theta-60, len/3);
        from = to;
        koch(from, len/3, theta, order-1);
    }
}
```

```

int main(void) {
    int ordem=5;
    Ponto p0;

    MostraPlanoCartesiano(10);
    AbreJanela(LARGURA, ALTURA, "Curvas de Koch");
    PintarFundo(255, 255, 255);

    p0.x = -100; p0.y = 0.0;

    koch(p0, SIZE, 0, ordem);

    Desenha();

    return 0;
}

```



Logo no início da listagem encontramos as diretivas de inclusão e entre estas a da `playcb.h` e a `math`. A biblioteca `math` disponibiliza as funções seno—`sin`—e cosseno—`cos`—além da constante `PI`. Os `defines` caracterizam a largura, altura da janela a ser aberta. Ajuste esses valores a capacidade de sua placa gráfica. A constante `SIZE` especifica a extensão da reta base, se esta fosse desenhada em sua totalidade.

A função `novaCaneta` é a alma *Turtle Graphics* do programa. Ela movimenta a caneta de desenho sem tocar o papel. O que isso quer dizer é que a partir do Ponto `from`, do ângulo `theta` e da extensão `len` passadas como entrada, a rotina calcula o ponto de destino usando transformações lineares simples. O ponto de destino `to` é retornado como valor da função. Notem que funções podem devolver `structs`.

Vamos deixar a função `koch` para o final dado que ela é a alma do traçado.

A função `main` é bem simples. Ela define uma variável `ordem` a qual especifica a ordem da curva de Koch. Experimentem mudar os valores: quanto mais baixo mais simples a curva será—menos detalhes serão apresentados. O papel da variável `p0` é definir o ponto referencial da curva, ou seja, onde a curva começa. Usando o grid de fundo produzido pela função `PLAYCB MostraPlanoCartesiano(10)`—a qual quadricula o plano em quadrados de 10 unidades de medida—determinamos os valores iniciais para as componentes `x` e `y` do Ponto `p0`.

Chegamos agora a chamada inicial da função `koch`. Para essa primeira chamada passamos os valores

- do ponto referencial `p0`,
- a extensão `SIZE` da linha reta *virtual de suporte*² da curva,
- o ângulo—`0`—em que devemos virar a cabeça da tartaruga,

²Parece evidente que a extensão de uma curva fractal tende a infinito enquanto a da reta virtual suporte se mantém fixo.

- e a **ordem** da curva.

Agora é a vez de discutirmos a função recursiva **koch**.

A função recebe como valores de entrada

- um **Ponto from** que especifica o ponto inicial do (sub)traçado,
- a extensão **double len** do segmento de reta a traçar,
- o ângulo **theta** em que devemos virar a cabeça da tartaruga.

A cabeça aponta para onde o segmento de reta deve ser traçado. E por analogia os ângulos podem virar a *esquerda* ou a *direita* da cabeça da tartaruga,

- a **ordem** da curva. A ordem controla o número de chamadas recursivas e consequentemente o nível de “rendilhado” da curva.

Fora as declarações das variáveis locais **Ponto to**; e **double rad = theta*PI/180**; a função tem apenas uma única instrução, a saber: **if else**.

O **if else** controla as chamadas recursivas. Os traçados de retas só são feitos quando a **ordem** for igual a zero. Caso contrário—**ordem > 0**—o ramo **else** se encarrega de realizar 4 chamadas recursivas.

O ramo “**then**” da instrução **if** quando alcançado se responsabiliza pelo desenho dos segmentos de reta. Para isso é necessário calcular o ponto de destino já que a **PLAYCB** trabalha com a função **CriaReta** entre dois pontos. Por essa razão, declaramos localmente a variável **to** a qual vai armazenar a posição do outro ponto do segmento de reta. O cálculo das coordenadas **to.x** e **to.y** são feitos usando trigonometria básica e não requerem maiores explicações. Agora que estamos de posse dos dois pontos extremos do segmento de reta podemos usar a função **PLAYCB CriaReta(from, to)**; mudar a espessura do traço **Grafite(2)**; e designar a cor do traçado **Pintar(255, 0, 0)**; . É importante lembrar que a **PLAYCB** não adere ao sistema *Turtle*. Isso significa que o ponto de referência não é automaticamente mudado ao final do traçado. Essa é a razão para a existência da função **movaCaneta** no ramo **else**.

No ramo **else** encontramos 4 chamadas recursivas para a função **koch**. A ideia básica é que cada chamada se encarrega de substituir cada um dos quatro segmentos da curva por réplicas de toda a curva em uma escala de 1/3. Como devemos respeitar a direção em que a tartaruga anda, temos que atualizar **theta** de acordo. Esse é o motivo para a soma e subtração de 60° ao ângulo **theta**. Como já havia dito, a **PLAYCB**³ não adere ao formato *Turtle* e por isso temos que calcular onde a caneta deveria estar ao término da execução de cada chamada recursiva. Esse é o papel das duas instruções: **to = movaCaneta...** e **from = to**;

Agora que sabemos desenhar as curvas de Koch, fica fácil desenhar um *floco de neve*.

2 Floco de Neve

Para desenhar um floco de neve junte três curvas de Koch usando um triângulo como inspiração. A Listagem 2 mostra como realizar essa proeza.

³A **PLAYCB** usa a API **OPENGL** que por adotar uma visão mais genérica e 3D não adere ao sistema *Turtle Graphics*.

Listagem 2: Programa `snowflake.c`

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <playcb.h>

#define LARGURA 800 //1962
#define ALTURA 600 //1280
#define SIZE 180

Ponto movaCaneta(Ponto from, double theta, double len) {
    Ponto to;
    double rad = theta*PI/180;
    to.x = from.x + len*cos(rad);
    to.y = from.y + len*sin(rad);
    return to;
}

Ponto koch (Ponto from, double len, double theta, int order) {
    Ponto to;
    double rad = theta*PI/180;

    if (order==0) {
        to.x = from.x + len*cos(rad);
        to.y = from.y + len*sin(rad);

        CriaReta(from, to); Grafite(2); Pintar(255,0,0);
    }
    else {
        koch(from, len/3, theta, order-1);

        to = movaCaneta(from, theta, len/3);
        from = to; //dispensável.
        // seu uso é para tornar intuitiva a chamada abaixo
        koch(from, len/3, theta+60, order-1);

        to = movaCaneta(from, theta+60, len/3);
        from = to;
        koch(from, len/3, theta-60, order-1);

        to = movaCaneta(to, theta-60, len/3);
        from = to;
        koch(from, len/3, theta, order-1);
    }
    return movaCaneta(from, theta, len/3);
}
```

```

void floco_de_neve(int ordem) {
    Ponto p0;
    p0.x = -45.0; p0.y = 26.0; //centro do floco de neve

    p0 = koch(p0, SIZE/2, 0, ordem);
    p0 = koch(p0, SIZE/2, -120, ordem);
    p0 = koch(p0, SIZE/2, 120, ordem);

}

int main(void) {
    int ordem=6;

    MostraPlanoCartesiano(10);
    AbreJanela(LARGURA, ALTURA, "Flocos de Neve");
    PintarFundo(255, 255, 255);

    floco_de_neve(ordem);

    Desenha();

    return 0;
}

```



COMENTÁRIOS O código é basicamente uma cópia melhorada de **koch.c**. Note que no presente código a função **koch** tem 2 instruções e se encarrega de calcular e devolver o ponto extremo direito do traçado realizado. Isso simplifica o projeto da função **floco_de_neve**.

3 Curvas de Sierpiński

Sim, o nome tem um acento agudo na letra n.

Essa é uma curva mais sofisticada, do tipo *space-filling curve* que como o nome diz tendem a preencher todo o espaço disponível (e sem nunca se cortar). As figuras Fig.4 – Fig. 7 mostram as curvas de ordem 1, 2, 3 e 4 resp..

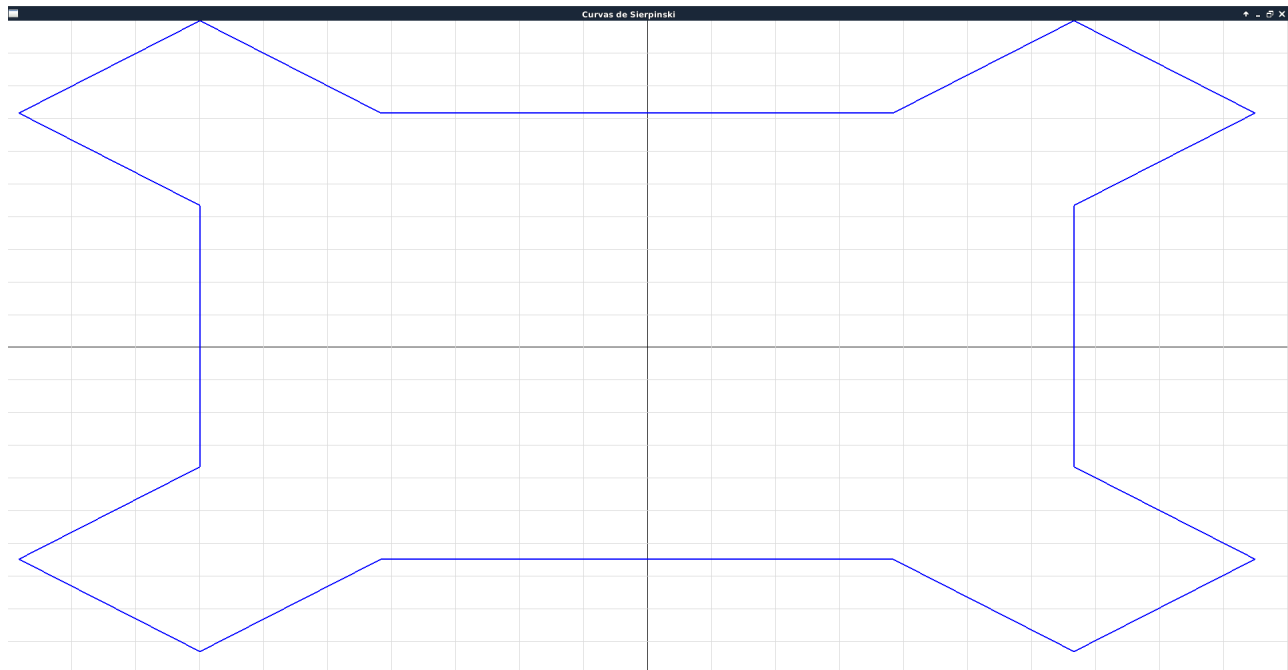


Figura 4: Curva de Sierpiński de ordem 1

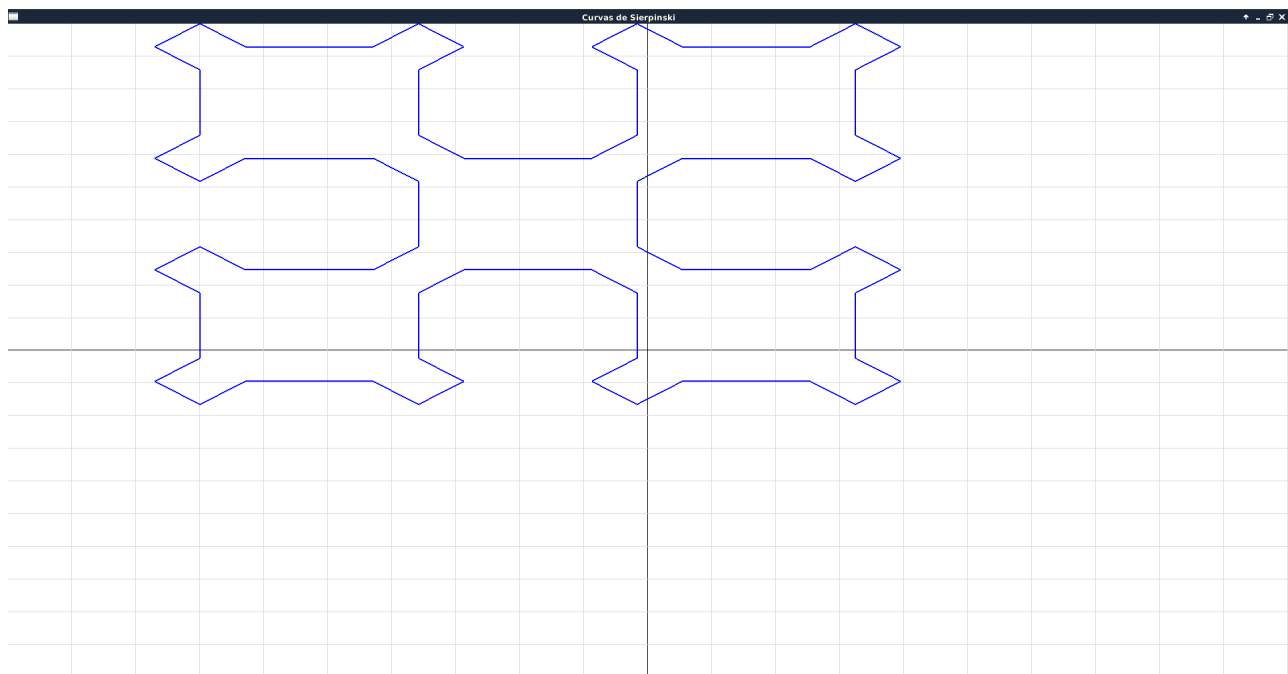


Figura 5: Curva de Sierpiński de ordem 2

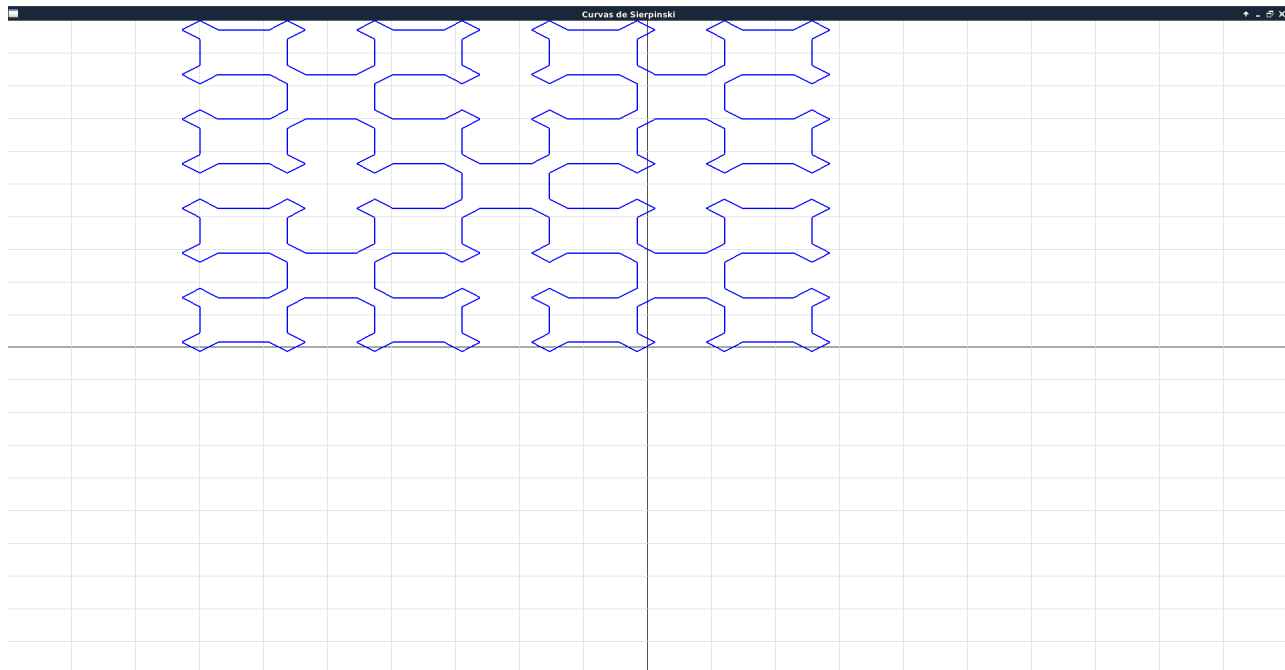


Figura 6: Curva de Sierpiński de ordem 3

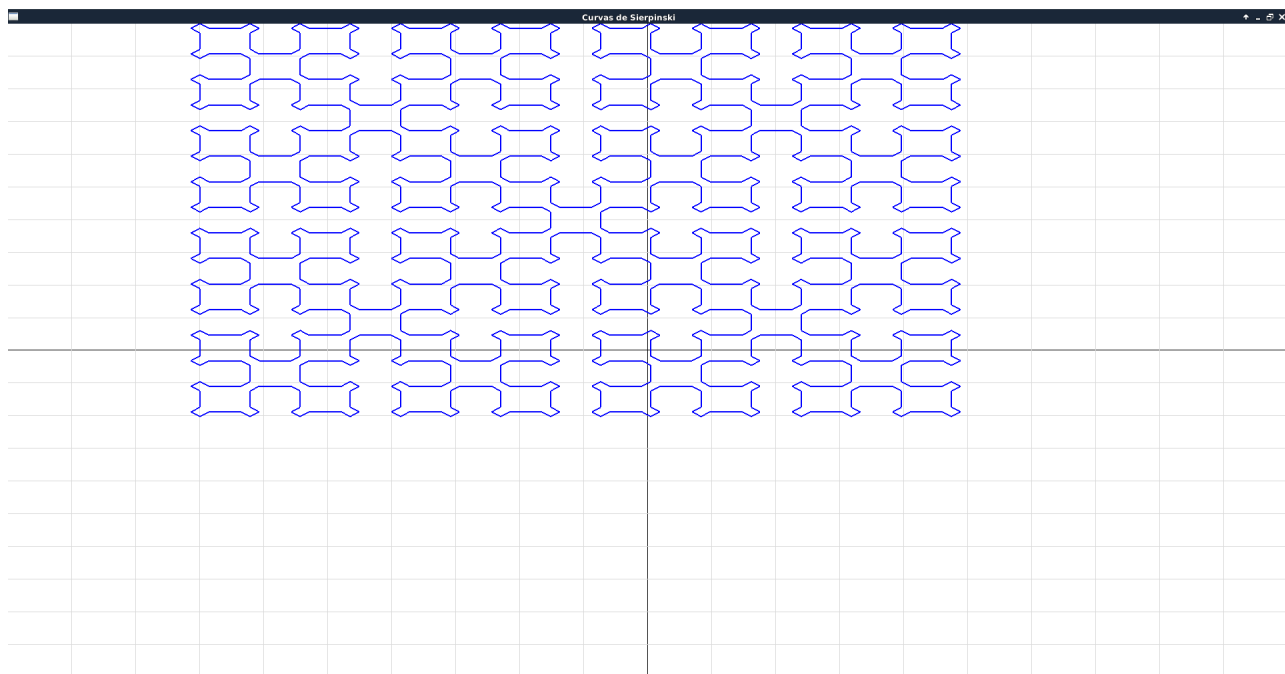





Figura 7: Curva de Sierpiński de ordem 4

Essa é uma curva que assusta a primeira vista pois ela é mutuamente recursiva entre 4 funções! Não se assuste, a coisa não é tão complicada.

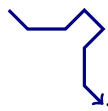
Para entender o padrão recursivo vamos usar a seguinte descrição:⁴


- A curva básica **S** é dada pelo padrão **A↘B↙C↗D↗**
As setas servem para indicar a virada de ângulo das retas que “fecham” o desenho das funções **A**, **B**, **C** e **D**.
- A curva **A** é dada pelo padrão **A↘B→D↗A**
- A curva **B** é dada pelo padrão **B↙C↓A↘B**
- A curva **C** é dada pelo padrão **C↗D←B↙C**
- A curva **D** é dada pelo padrão **D↗A↑C↗D**

Os padrões podem ser acompanhados na Fig. 4. Vamos iniciar a observação da figura começando na parte de cima, aquela junto a identificação da janela, e seguindo um trajeto em sentido horário. A curva **A** desenha a parte de cima da figura. A curva **B** desenha as três retas na parte lateral direita. A curva **C** desenha as três retas na parte horizontal inferior da figura. A curva **D** desenha as três retas na parte vertical esquerda da figura. O padrão **S** desenha as retas que ligam essas quatro curvas abertas. O resultado disso é uma curva fechada, a curva de Sierpiński de ordem 1.



Se ainda assim você não visualizou a sequência de traçado, vamos ao mundo das tartarugas. Em um *gráfico de tartaruga* não podemos tirar o lápis do papel enquanto desenhamos. Comece com **A**; o lápis desenha uma reta que emenda na outra que emenda na outra. Qual é o resultado disso? Só pode ser . Quando **A** termina a caneta está posicionada onde desenhamos a ponta da seta. Agora olhe no padrão **S** e veja que após o término de **A** vem o traçado de uma reta . De onde paramos acrescentamos essa reta e teremos .

A qual traça a curva começando de onde paramos. E o seu traçado é . O resultado dessa junção só pode ser



. Como acabou a execução de **B** entra em ação a ligação no padrão **S** antes da chamada de **C**; essa ligação



desenha . Some essa reta a figura produzida até agora e obteremos .

acrescenta  ao traçado produzindo . O resto fica como exercício mental.

O programa completo é apresentado na Listagem 3.

Listagem 3: Programa `sierpinski.c`

```
#include <stdlib.h>
#include <stdio.h>
```

⁴Essa descrição está em [1].

```

#include <math.h>
#include <playcb.h>

// desenho realizado no estilo conhecido como Turtle Graphics
#define LARGURA 1962 //800
#define ALTURA 1280 //600
#define SIZE 180

typedef unsigned int Cardinal;

Ponto A (Cardinal k, float h, Ponto p);
Ponto B (Cardinal k, float h, Ponto p);
Ponto C (Cardinal k, float h, Ponto p);
Ponto D (Cardinal k, float h, Ponto p);

Ponto reta(Cardinal fator, Cardinal ext, Ponto p) {
    // fator é para multiplicar por 45 graus
    Ponto p1;
    double ar = fator * 45.0 * PI/180; //ângulo em radianos

    p1.x = p.x + ext*cos(ar); //coordenada x do ponto de destino
    p1.y = p.y + ext*sin(ar); //coordenada y do ponto de destino
    CriaReta(p,p1); Grafite(2); Pintar(0,0,255);
    return p1;
}

Ponto A (Cardinal k, float h, Ponto p) {
    if ( k > 0 ) {
        p = A(k-1,h, p); p = reta(7,h, p);
        p = B(k-1,h, p); p = reta(0,2*h, p);
        p = D(k-1,h, p); p = reta(1,h, p);
        p = A(k-1,h, p);
    }
    return p;
}

Ponto B (Cardinal k, float h, Ponto p) {
    if ( k > 0 ) {
        p = B(k-1,h, p); p = reta(5,h, p);
        p = C(k-1,h, p); p = reta(6,2*h, p);
        p = A(k-1,h, p); p = reta(7,h, p);
        p = B(k-1,h, p);
    }
    return p;
}

```

```

Ponto C (Cardinal k, float h, Ponto p) {
    if ( k > 0 ) {
        p = C(k-1,h, p); p = reta(3,h, p);
        p = D(k-1,h, p); p = reta(4,2*h, p);
        p = B(k-1,h, p); p = reta(5,h, p);
        p = C(k-1,h, p);
    }
    return p;
}

Ponto D (Cardinal k, float h, Ponto p) {
    if ( k > 0 ) {
        p = D(k-1,h, p); p = reta(1,h, p);
        p = A(k-1,h, p); p = reta(2,2*h, p);
        p = C(k-1,h, p); p = reta(3,h, p);
        p = D(k-1,h, p);
    }
    return p;
}

int main(void) {

    Ponto p;
    int i = 5; //dai pra cima a coisa fica grande e mais lenta
    float h = 40;

    p.x = -70; p.y = 100; //ordem=1 cabe todo na tela

    MostraPlanoCartesiano(10);
    AbreJanela(LARGURA,ALTURA, "Curvas de Sierpinski");
    PintarFundo(255, 255, 255);

    if (i>0) h /= i*i;
    p = A(i,h, p); p = reta(7,h, p);
    p = B(i,h, p); p = reta(5,h, p);
    p = C(i,h, p); p = reta(3,h, p);
    p = D(i,h, p); p = reta(1,h, p);

    Desenha();

    return 0;
}

```

No código usamos as inclinações das retas designadas por valores entre 0 e 7, inclusive. Esses são fatores multiplicativos para um ângulo base de 45° e são passados como o 1º argumento das funções. A extensão das retas é passada como 2º parâmetro. Quando as retas são horizontais ou verticais suas extensões são maiores que as demais.

O terceiro parâmetro corresponde ao ponto de referência inicial do traçado. Note que as funções se encarregam de devolver esse valor quando terminam suas execuções.

Hora de descansar. Que tal um joguinho de tabuleiro?

4 Problema das oito rainhas

Este problema pede para colocarmos de forma segura oito rainhas num tabuleiro de xadrez. Em outras palavras, nenhuma rainha pode estar sob ataque de outra.

Para os curiosos: antes de iniciar a discussão sobre a solução apresentada, este problema tem 92 soluções das quais 12 são independentes entre si. As demais soluções podem ser obtidas por operações de reflexão, simetria, etc. a partir de outra solução.

A Listagem 4 mostra um programa que apresenta *apenas uma* das muitas soluções desse problema.

Notem que o tabuleiro é representado de forma implícita através de um único vetor de tamanho 8. Observando o programa podemos ver que a estratégia usada para “percorrer” o tabuleiro em busca de uma casa segura consiste em usar uma variável que representa a coluna corrente, um vetor que representa a linha onde poderíamos colocar uma rainha. So far, so good. Mas o problema são as diagonais. Como descobrir se podemos ou não colocar uma rainha na linha—vetor `tabul`—aparentemente disponível? *A resposta é dada por uma relação que envolve linha e coluna.* Esses fatos são usados pela função `ta_livre`, linhas 68–74.

O program foi originalmente escrito objetivando apresentar **TODAS** as soluções do problema. Isso se consegue com o uso de *retrocesso*—*backtracking*. A implementação apresentada na Listagem 4 mostra como podemos interromper o processo de retrocesso, linhas 90 e 91.

O tabuleiro 8×8 é desenhado pela função `tabuleiro`, linhas 22–40. Essa função pinta cada casa do tabuleiro alternando as cores, linhas 30–35. A função `casa`, linhas 14–20, é quem pinta um quadrado na cor especificada.

O programa representa uma rainha através do desenho de um círculo azul. Isso é feito pela função `rainha`, linhas 9–12.

Listagem 4: Programa `rainhas.c`

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <playcb.h>
4
5 int tabul[8]; //armazena o índice da rainha na linha i
6 //Se tabul[5] for 3, então na linha 5 a rainha está na coluna 3
7
8 /* Início das rotinas gráficas do tabuleiro */
9 void rainha (Ponto p) {
10     CriaCirculo(4,p); //raio, ponto
11     Pintar(0,0,255);
12 }
13
```

```

14 void casa (Ponto p, int tipo) {
15     CriaRetangulo(10,10,p);
16     if (tipo == 0)
17         Pintar(0,255,0);
18     else
19         Pintar(255,255,0);
20 }
21
22 void tabuleiro (void) {
23     int i, j, cor = 1;
24
25     Ponto p;
26
27     p.x = -40;
28     p.y = -40;
29
30     for (i=0; i<8; i++) {
31         for (j=0; j<8; j++) { //pinta as 8 colunas de uma linha
32             casa(p, cor%2);
33             cor++;
34             p.x += 10;;
35         }
36         p.x = -40;
37         p.y += 10;
38         cor++; //próxima linha tem que alternar a cor inicial
39     }
40 }
41 /* Fim das Rotinas Gráficas do tabuleiro */
42
43
44 void print () { //rotina de desenho que está associada ao problema 8 rainhas
45     Ponto p;
46
47     int i,j;
48
49     p.x = -35;
50     p.y = -35;
51
52     for (i=0; i<8; i++) {
53         for (j=0; j<8; j++)
54             if (j==tabul[i]) {
55                 if (j==0) p.x = p.x + j*10;
56                 else      p.x = p.x + j*10 + 5;
57                 rainha (p);
58             }
59         p.x = -40;
60         p.y = p.y + 10;
61     }

```

```

62 }
63 }
64
65 //Verifica se a posição (indx,indy) está sob ataque de uma das
66 // rainhas 0...(indy-1)
67
68 int ta_livre (int indx, int indy) {
69     int i;
70
71     for (i=0; i<indy; i++)
72         if ((tabul[i]==indx) || (abs(tabul[i]-indx)==abs(i-indy))) return 0;
73     return 1;
74 }
75
76 //Tenta por a rainha n na linha i
77 void tenta (int n) {
78     static int pare = 0; //variável responsável pelo corte no retrocesso
79
80     int i;
81
82     if (n==8) {
83         pare = 1; //sinaliza que encontrou a 1ª solução
84         print(); //imprime o tabuleiro
85     }
86     else
87         for (i=0; i<8; i++)
88             if (ta_livre(i,n)) {
89                 tabul[n]=i;
90                 if (!pare) tenta (n+1);
91                 else break; //corta o backtracking. Só uma solução é impressa!
92             }
93 }
94
95
96 int main (void) {
97     Ponto p;
98
99     MostraPlanoCartesiano(10);
100    AbreJanela(800,600, "Tabuleiro de Xadrez");
101    PintarFundo(255, 255, 255);
102
103    tabuleiro();
104    tenta(0);
105
106    Desenha();
107
108    return 0;
109 }

```

A função `tenta` é responsável pela colocação das rainhas no tabuleiro. Para isso, ela se vale da função `ta_livre`. `ta_livre` verifica se a casa corrente está ou não livre. Se estiver, `ta_livre` retorna 1. Com isso, na linha 88, entramos no ramo “then” e armazenamos em `tabul` a coordenada da rainha. A próxima instrução, linha 90, verifica se já encontramos uma solução. Se esse for o caso, linha 91, quebramos o `for` da linha 87 e com isso encerramos a função `tenta`.

Um aspecto novo para vocês é o uso da variável `pare`.

Essa variável é `local` a função `tenta` mas é declarada como `static`. *Variáveis `static` tem a capacidade de guardar o valor entre chamadas da função em que foram declaradas!* Por essa razão, quando uma das chamadas recursivas de `tenta` consegue colocar a oitava rainha no tabuleiro—linha 82—atribuímos a essa variável o valor 1—linha 83. Dessa forma, as demais instâncias de `tenta` ficam sabendo que é hora de parar—linhas 90 e 91.

5 Torres de Hanói

Trabalho em progresso. Neste caso eu vou implementar em 3D usando a OpenGL. O resultado final será bonito mas o código será bem menos legível para iniciantes.

Referências

- [1] Niklaus Wirth. *Algoritmos e Estruturas de Dados*. Prentice Hall do Brasil, 1989.