Modelação e Simulação de Sistemas Naturais

Trabalho Prático 3

52553 - Artur Assis
52864 - Tiago Teixeira

Licenciatura em Engenharia Informática e Multimédia
Semestre de Inverno 2025/2026
14/12/2025

Índice

Introdução

Este relatório apresenta o desenvolvimento e a análise de um conjunto de simulações no âmbito da disciplina de Modelação e Simulação de Sistemas Naturais, realizadas com a biblioteca do Processing. O trabalho está organizado em quatro blocos principais, cada um explorando conceitos fundamentais da modelação de sistemas físicos e comportamentos inteligentes.

No primeiro bloco, “Função Logística”, analisa-se a dinâmica do mapa logístico, com destaque na dependência sensível às condições iniciais, conhecido como o conceito de “Efeito Borboleta”, os resultados obtidos são produzidos e ilustrados pelo Jupyter Notebook, que permite observar a evolução temporal das trajetórias para diferentes valores do parâmetro de controlo.

No segundo bloco, é implementada uma versão simples do Jogo do Caos, evidenciando a emergência de estruturas fractais a partir de um processo iterativo e aleatório.

O terceiro bloco, “Gramáticas de Lindenmayer”, focou-se na implementação de dois objetos fractais, usando a técnica designada por L-Systems” e a criação de uma árvore de frutos usando como base variáveis, constantes, axiomas e regras que vão ser explicadas e comentadas neste relatório.

Por fim, no quarto bloco, “Conjuntos de Julia e Mandelbrot”, foram coloridos pontos que não pertenciam a um conjunto de Mandelbrot, de modo a ilustrar-se melhor o conceito.

Cada secção inclui uma discussão crítica dos resultados, realçando as opções de implementação e o comportamento observado nas simulações.

A. Função Logística (Efeito Borboleta)

Neste exercício analisamos uma função logística, f(x) = rx(1-x). Com esta função podemos testar o chamado “efeito borboleta”, que consiste na dependência sensível às condições iniciais em sistemas dinâmicos não lineares. Ou seja, duas trajetórias são semelhantes no início e evoluem de forma completamente diferente ao longo do prazo do tempo.

De modo a ilustrar este fenómeno criámos duas variáveis de estado com valores muito próximos um do outro:

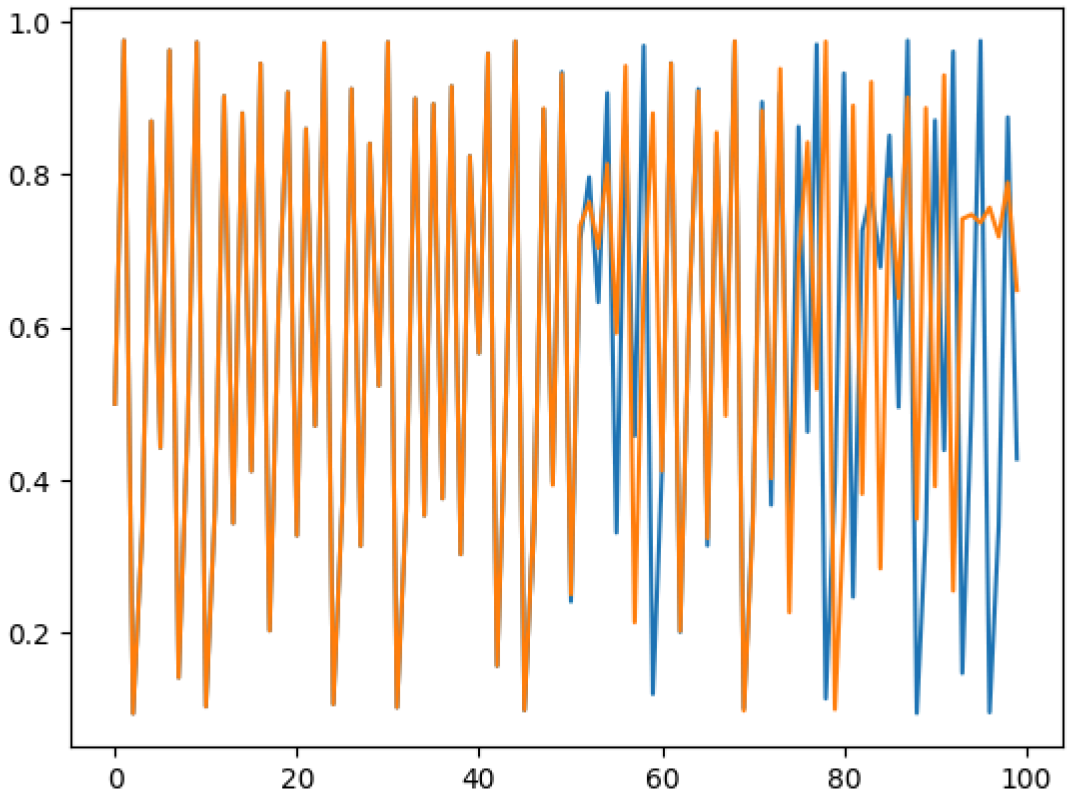
x0 = 0.5000000; x0 = 0.5000001; r = 3.9 (atrator aperiódico/caótico)

De acordo com este conceito, no início as trajetórias são semelhantes (quase idênticas) e vão assumir formas completamente diferentes ao longo do tempo. Para simularmos os resultados criámos em Python um programa que desenha o gráfico respectivo a cada variável:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import TextBox, Button

def log_fun(r, x):
   return r * x * (1 - x)

x = 0.5
x0_1 = 0.5000001
niter = 100
r = 3.9

case_1 = np.zeros(niter)
case_2 = np.zeros(niter)
case_1[0] = x
case_2[0] = x0_1
for i in range(len(case_1)):
   if (i != 0):
       case_1[i] = log_fun(r, case_1[i-1])
       case_2[i] = log_fun(r, case_2[i-1])
```

```
plt.figure()
plt.plot(case_1)
plt.plot(case_2)
```

Com este código obtivemos o seguinte resultado:



B. Jogo do Caos

1. Estrutura Principal e Abordagem

O algoritmo do Jogo do Caos foi implementado na classe JogoDoCaosApp, seguindo a estrutura do Processing (através da interface IProcessing).

Componentes:

- SubPlot plt: Responsável por gerenciar a transformação entre as coordenadas do Mundo (definidas pela janela window = {2, 8, 2, 8}) e as coordenadas do *viewport* da tela, conforme a especificação do *framework*.
- PVector[ ] vertices: Armazena os três vértices fixos do triângulo (A, B, C): (2, 2), (8, 2) e (5, 8).
- PVector pontoAtual: Representa o ponto móvel X, que é atualizado a cada iteração.
- Controle de Estado: Variáveis como jogoComecou e iteracoes controlam o fluxo da aplicação, permitindo que o usuário escolha o ponto inicial antes de iniciar o processo de geração do fractal.

2. Implementação do Algoritmo

2.1. Configuração (setup)

A função setup inicializa os vértices, o objeto random para o sorteio, e define as cores para os pontos:

- Vértice A: Vermelho (cores[0])
- Vértice B: Verde (cores[1])
- Vértice C: Azul (cores[2])

2.2. A Lógica da Iteração (executarIteracao)

O cerne do algoritmo está no método executarIteracao(PApplet p), que é chamado repetidamente no *loop* principal (draw).

```java
private void executarIteracao(PApplet p) {  1 usage
    // sorteia aletoriamente um ponto
    int verticeIndex = random.nextInt( bound: 3);
    PVector verticeEscolhido = vertices[verticeIndex];

    // calcula o ponto intermedio entre x e t
    // X = X + 0.5(T - X)
    pontoAtual.x = pontoAtual.x + 0.5f * (verticeEscolhido.x - pontoAtual.x);
    pontoAtual.y = pontoAtual.y + 0.5f * (verticeEscolhido.y - pontoAtual.y);

    // pinta com a cor correnpondente o x
    float[] pixelCoord = plt.getPixelCoord(pontoAtual.x, pontoAtual.y);
    p.strokeWeight(3);
    p.stroke(cores[verticeIndex]);
    p.point(pixelCoord[0], pixelCoord[1]);
}
```

(executarIteracao())

3. Resultados e Análise

3.1. Padrão Gerado

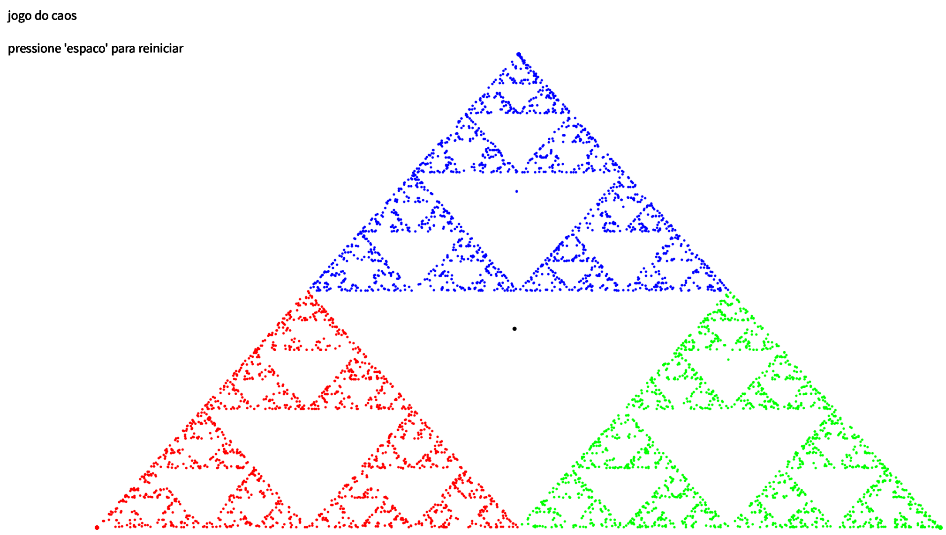
Após um número suficiente de iterações (o limite é maxIteracoes = 10000), a acumulação de pontos gerados converge para a forma do Triângulo de Sierpinski.

- Características do Padrão: O Triângulo de Sierpinski é um fractal autossemelhante. Isso significa que, se ampliarmos qualquer parte do padrão, veremos a repetição da mesma estrutura triangular.
- Densidade de Pontos: O padrão surge porque, ao se calcular repetidamente o ponto médio entre o ponto atual e um vértice aleatório, o ponto X é "puxado" para as regiões que compõem o fractal e, notavelmente, nunca entra nas regiões vazias (buracos) do Triângulo de Sierpinski, independentemente do ponto de partida.

3.2. Significado das Cores

A escolha de colorir o ponto X com a cor do vértice T sorteado revela uma propriedade interessante do fractal:

- Regiões Coloridas: As três grandes regiões (os triângulos menores que formam o Triângulo de Sierpinski) são predominantemente pintadas com uma única cor. Por exemplo, o triângulo no canto superior será majoritariamente azul (a cor associada ao vértice C), o triângulo inferior esquerdo será predominantemente vermelho (vértice A), e o inferior direito será verde (vértice B).
- Razão: Isso acontece porque, para que um ponto caia em uma das sub-regiões (sub-triângulos), ele deve ter sido atraído repetidamente para o vértice correspondente àquela sub-região.



(jogo do caos em ação após alguns segundos de iterações)

C. Gramáticas de Lindenmayer

1. Implementação de Objetos Fractais

Foram implementados dois fractais, cada um demonstrando um padrão de crescimento distinto:

Curva de Koch (KochCurveApp)

- **Axioma:** F
- **Regra:** F -> F + F - - F + F

Análise: Esta regra substitui cada segmento de linha por quatro segmentos de comprimento um terço do original, criando o contorno autossemelhante da Curva de Koch. A escala é reduzida para 1/3 a cada geração.

```
@Override
public void setup(PApplet parent) {
    plt = new SubPlot(window, viewport, parent.width, parent.height);
    Rule[] ruleset = new Rule[1];
    ruleset[0] = new Rule('F', "F+F--F+F");

    Ls = new LSystem("F", ruleset);
    turtle = new Turtle(8, PApplet.radians(60f));
}

@Override
public void draw(PApplet parent, float dt) {
    float[] bb = plt.getBoundingBox();
    parent.rect(bb[0], bb[1], bb[2], bb[3]);

    // A Curva de Koch começa horizontalmente (0 graus)
    turtle.setPose(startPos, PApplet.radians(0), parent, plt);
    turtle.render(Ls, parent, plt);
}
```

Planta (PlantApp)

- **Axioma:** X
- **Regras:** X -> F + [[X] - X] - F[- FX] + X e F -> FF

Análise: Os caracteres [e] instruem o mecanismo de desenho a salvar e restaurar o estado (posição e orientação do vetor de desenho), permitindo a criação de ramificações complexas. A regra F -> FF garante o alongamento do crescimento.

```java
@Override
public void setup(PApplet parent) {
    plt = new SubPlot(window, viewport, parent.width, parent.height);
    Rule[] ruleset = new Rule[2];
    ruleset[0] = new Rule( symbol: 'X',  string: "F+[[X]-X]-F[-FX]+X");
    ruleset[1] = new Rule( symbol: 'F',  string: "FF");

    Ls = new LSystem( axiom: "X", ruleset);
    turtle = new Turtle( len: 3, PApplet.radians( degrees: 22.5f));
}

@Override
public void draw(PApplet parent, float dt) {
    float[] bb = plt.getBoundingBox();
    parent.rect(bb[0], bb[1], bb[2], bb[3]);

    turtle.setPose(startPos, PApplet.radians( degrees: 90), parent, plt);
    turtle.render(Ls, parent, plt);
}
```

2. Árvore de Frutos (TreeFruitApp)

O sistema de Lindenmayer proposto foi implementado na classe TreeFruitApp para simular uma árvore ramificada.

- **Axioma:** F
- **Regras:** F -> G[+F] - F e G -> GG
- **Ângulo:** 22.5

Análise: A regra de F gera a ramificação: [+F] e [-F] criam novos galhos com rotação do vetor de direção. A regra de G (-> GG) garante o alongamento contínuo dos segmentos. O mecanismo de desenho aplica uma redução de escala de 0.5 a cada nova geração para controlar o tamanho dos ramos.

```
@Override
public void setup(PApplet parent) {
    plt = new SubPlot(window, viewport, parent.width, parent.height);
    Rule[] ruleset = new Rule[2];
    ruleset[0] = new Rule( symbol: 'F',  string: "G[+F]-F");
    ruleset[1] = new Rule( symbol: 'G',  string: "GG");

    Ls = new LSystem( axiom: "F", ruleset);
    turtle = new Turtle( len: 5, PApplet.radians( degrees: 22.5f));
}

@Override
public void draw(PApplet parent, float dt) {
    float[] bb = plt.getBoundingBox();
    parent.rect(bb[0], bb[1], bb[2], bb[3]);

    turtle.setPose(startPos, PApplet.radians( degrees: 90), parent, plt);
    turtle.render(Ls, parent, plt);
}
```

D. Conjuntos de Julia e Mandelbrot

Já com o código do Mandelbrot fornecido, a tonalidade (Hue) de cada ponto do plano foi determinada pelo número de iterações necessárias para que o módulo da sequência ultrapasse um determinado limite. A saturação e o brilho foram mantidos constantes, permitindo que apenas a cor representasse a rapidez com que cada ponto “escapa” do conjunto.

Em termos práticos:

- Pontos que permanecem próximos à origem por muitas iterações receberam tonalidades correspondentes a valores altos de iteração.
- Pontos que divergem rapidamente receberam tonalidades correspondentes a valores baixos de iteração.
- Essa escolha de mapeamento Hue garante que as regiões do conjunto que divergem mais rapidamente sejam visualmente distintas das regiões estáveis, produzindo padrões coloridos e detalhados que evidenciam a estrutura fractal.

Assim, a cor funciona como uma representação visual da dinâmica iterativa, destacando tanto a geometria do conjunto quanto o comportamento dos pontos próximos à fronteiras do conjunto.

```
float hue = p.map(i, 0, niter, 0, 255);
```

Conclusões

A. Funções Logística

Como se pode observar, os resultados comprovam a veracidade do conceito do efeito borboleta, no início assumem valores idênticos, e vão se alterando radicalmente a cada iteração.

B. Jogo do Caos

Através da escolha aleatória de vértices e do cálculo iterativo da posição de um ponto inicial, foi possível observar a formação do Triângulo de Sierpinski, evidenciando padrões auto-similares que emergem de regras simples. Além disso, a utilização de cores distintas para cada vértice permitiu visualizar a influência de cada um na trajetória do ponto, tornando o comportamento do sistema mais intuitivo e visualmente interessante.

C. Gramáticas de Lindenmayer

A implementação de sistemas de Lindenmayer permitiu explorar a geração de estruturas fractais através de regras formais simples aplicadas de forma iterativa. Nos exemplos desenvolvidos (Curva de Koch, árvore com ramificações e planta) foi possível observar o que era esperado.

D. Conjuntos de Julia e Mandelbrot

Esta simples implementação facilitou a compreensão da teoria e tornou o aspecto do programa mais apelativo com uma maior riqueza visual e detalhes finos da fronteira mais realçados.