Juan Carlos Díaz Rodríguez
Subgrupo U1
juancdia@ucm.es

Geometría Computacional
Práctica Opcional. Problema 2

21/04/2023

1. Introducción

En esta práctica veremos cómo calcular una aproximación de la dimensión de Hausdorff de un fractal en $\mathbb{R}^2$ dado por un conjunto discreto de puntos. Además, como ejemplo ilustrativo, representaremos el famoso Triángulo de Sierpinksi y calcularemos su dimensión.

2. Material utilizado

Se ha empleado el lenguaje *Python* así como las librerías de `numpy` y `matplotlib` para la representación del fractal.

La dimensión de Hausdorff o dimensión de Hausdorff-Besicovitch es una generalización métrica del concepto de dimensión de un espacio topológico, que permite definir una dimensión fraccionaria (no entera) para un objeto fractal.

Sea I un conjunto de índices, la colección $\{U_i\}_{i\in I}$ es un δ-recubrimiento de $E \subset \mathbb{R}^n$ si $E \subset \bigcup_{i\in I} U_i$ y $0 < |U_i| \leq \delta \; \forall \, i \in I$.

Sea $F \subset \mathbb{R}^n$ y s un número no negativo. Para cualquier $\delta > 0$ se define:

$$\mathcal{H}^s_\delta(F) = \text{ínf}\{\sum_{i=1}^{\infty} |U_i|^s\},$$

donde el ínfimo se toma de entre todos los δ-recubrimientos numerales de F.

La medida exterior s-dimensional de Hausdorff del conjunto F se define como el valor

$$\mathcal{H}^s(F) = \lim_{\delta\to 0} \mathcal{H}^s_\delta(F).$$

Este límite existe, sin embargo, como $\mathcal{H}^s_\delta$ crece cuando δ decrece, puede ser infinito.

Finalmente, para todo conjunto $F \subset \mathbb{R}^n$ existe $s_0 \leq n$ tal que $\mathcal{H}^s(F) = 0$ si $s > s_0$ y $\mathcal{H}^s(F) = \infty$ si $s < s_0$. Es a este valor s_0 a lo que se denomina dimensión de Hausdorff del conjunto F.

En cuanto al cálculo de dicho valor, se ha decidido implementarlo en C++ por razones que comentaremos más tarde. La idea es computar la dimensión de Hausdorff adheriéndonos en lo posible a la definición. Para ello, tomaremos δ-recubrimientos cada vez más finos y trataremos de obtener el recubrimiento mínimo. Dado que sería increíblemente costoso buscar realmente el mínimo, lo que haremos será poner cuadrados de lado δ sobre puntos aleatorios que aún no hayan sido cubiertos y marcar los puntos que abarca dicho cuadrado. Para buscar esos puntos ordenaremos lexicográficamente los elementos del fractal para saber cuándo parar la búsqueda de puntos recubiertos. Para acercarnos al mínimo fijaremos un parámetro `reps`, que nos indicará el número de recubrimientos aleatorios para cada δ entre los que buscar el mínimo. Podría haberse hecho un mallado del cuadrado $[0,1]\times[0,1]$, pero de esta forma tendremos que considerar un menor número de cuadrados, ya que todos los que miremos cubrirán al menos un elemento, mientras que del otro modo estaríamos en muchas ocasiones comprobando cuadrados vacíos.

Una vez tengamos el tamaño mínimo N_δ encontrado para cada precisión, realizaremos una búsqueda binaria de s donde tendremos que tomar un s menor, respectivamente mayor, si la función $f_s(\delta) = N_\delta\left(\delta\sqrt{2}\right)^s$ es decreciente cuando $\delta \to 0$, respectivamente creciente. Nótese la similitud entre f_s y $\mathcal{H}^s_\delta$. Sin embargo, dado que N_δ no se toma del ínfimo de entre los δ-recubrimientos numerables, ya no podemos asegurar que f_s sea decreciente con δ, lo cual es explotado durante la búsqueda binaria. Puesto que solo vamos a tener un número finito de imágenes de f_s y la derivada por tanto será un mero cociente incremental, vamos a tomar una precisión máxima de δ próxima a 0 y calcularemos únicamente la imagen de ese δ y de uno un poco mayor. Así reducimos la cantidad de recubrimientos mínimos que buscar (aunque estos dos son los más costosos).

3. Resultados

Apartado i). La representación del Triángulo se ha hecho dibujando segmentos de manera recursiva según se avanza en la aplicación de las homotecias. También se da otra forma de aproximar el fractal muy interesante, que además será la aproximación empleada para el cálculo de la dimensión. Consiste en tomar un punto inicial

aleatorio en el cuadrado $[0,1] \times [0,1]$. A continuación, se elige un vértice del triángulo original aleatoriamente y se añade el punto entre el punto anterior y el vértice. Actualizamos el punto actual al calculado e iteramos tantas veces queramos. Ambas representaciones se pueden ver en las Figuras 1, 2, 3 y 4.

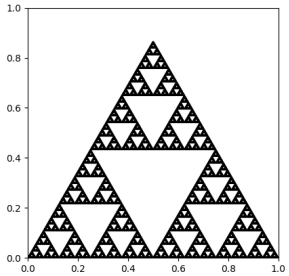

Figura 1: Segmentos sin zoom

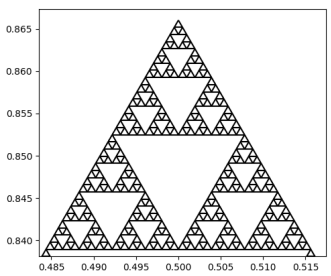

Figura 2: Segmentos con zoom

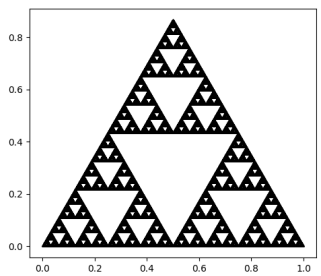

Figura 3: Puntos aleatorios sin zoom

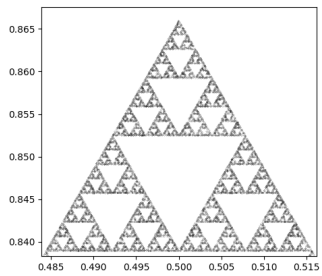

Figura 4: Puntos aleatorios con zoom

Apartado ii). Para el cálculo de la dimensión se sigue el algoritmo descrito anteriormente. Sin embargo, ese algoritmo tiene una problema. La dimensión de Hausdorff de un conjunto discreto siempre es 0, por lo que si aumentamos arbitrariamente la precisión de los recubrimientos todas las funciones f_s terminará convergiendo a 0 como se puede intuir en la Figura 5, junto a los cocientes incrementales. Para obtener una buena aproximación tenemos que aumentar el número de puntos del conjunto discreto a la par que reducimos δ. No he conseguido determinar una forma de hacer esto de forma rigurosa, así que a la vista de la Figura 5, obtenida con $4 \cdot 10^6$ puntos, tomaremos como máxima precisión $0{,}0016$, antes del colapso de la monotonía de f_s.

En la Figura 6 podemos ver los valores explorados por la búsqueda binaria. En la gráfica inferior tenemos los cocientes incrementales, donde se aprecia cómo se busca el 0. Hay que remarcar que el algoritmo no computará todos estos valores, sino solo los dos más a la derecha para cara s, pero se han incluido a modo ilustrativo.

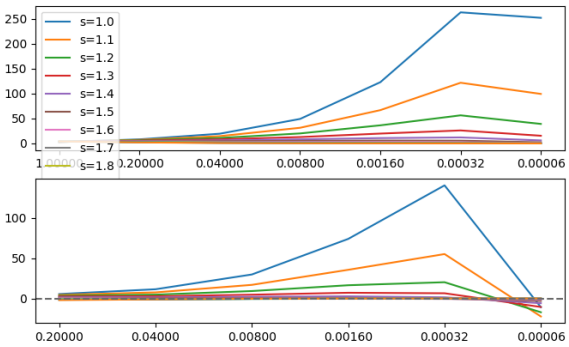

Figura 5: Exploración de la precisión

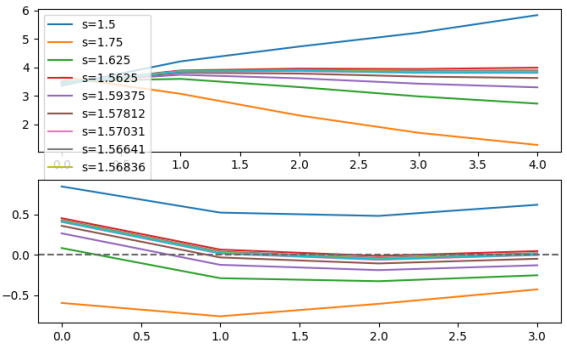

Figura 6: Búsqueda binaria de la dimensión

La dimensión calculada para el Triángulo de Sierpinski es de **1.56934**. El error comentido por la búsqueda binaria es de $\pm 2^{-10}$. Podemos ver cómo este error deja fuera al valor de la dimensión de Hausdorff exacta ($\frac{\log 3}{\log 2} \approx 1{,}58$). Esto es debido a que la mayor parte del error proviene de la aproximación discreta del fractal y la precisión tomada en consecuencia para los recubrimientos. El cálculo de este error requiere de un estudio más profundo y en general no parece ser un problema nada fácil.

4. Conclusión

De esta práctica podemos aprender lo siguiente:

- Para que un conjunto discreto de puntos se asemeje a un continuo, puede ser útil limitar el tamaño de los abiertos con que se trabaja.
- La dimensión de Hausdorff de un fractal arbitrario es difícil de calcular si no se tienen métodos más sofisticados que seguir la definición. Existen métodos más elaborados y rigurosos en caso de tener una función que nos ayude a determinar la pertenencia de un punto a un fractal, pero para aproximaciones discretas no hay un método general cuya precisión sea fácil de determinar.
- En la computación científica muchos algoritmos son fácilmente paralelizables y suponen una gran mejora de rendimiento. Es por ello que aquí se decició usar C++, por su eficiencia y facilidad para crear *threads* dada la independencia de la memoria.

5. Código

Se ha excluido el código empleado para la generación de las Figuras 5 y 6, pues siguen el mismo algoritmo que el aquí expuesto, pero el código es más sucio debido a la entrada salida necesaria para pasar los datos a Python y graficar los resultados. Se excluye también el código de Python que crea las gráficas por carecer de interés alguno.

Aquí se encuentra el código en C++ encargado de computar la dimensión de Hausdorff aprovechando la librería `thread` soportada a partir de C++11.

```
// Compile with:
// g++ problema2.cpp -lpthread -std=c++11 -o testThread -Wall

#include <fstream>
#include <iostream>
#include <cmath>
#include <algorithm>
#include <vector>
#include <thread>

#define PI 3.14159265

using namespace std;
struct dPoint{
    double x;
    double y;
};

int get_random_uncovered(const vector<bool>& covered, size_t N){
    int i = rand() % N;
    while (covered[i%N]){++i;}
    return i%N;
}

/// @brief  Cálculo de un recubrimiento por cuadrados de lado side
/// @param points Puntos a recubrir
/// @param side Lado de los cuadrados con que se recubre
/// @param results Array donde poner el resultado
/// @param index Posición del array donde poner el resultado
void cover(
    const vector<dPoint> *points,
    double side,
    vector<size_t> *results,
    size_t index
    ){
    double side2 = side / 2;
    int N = int(points->size());
    vector<bool> covered(N, false);
    int n_covered = 0;
    size_t balls_used = 0;
    while (n_covered < N){
        int index = get_random_uncovered(covered, N);
        ++balls_used;
        covered[index] = true;
        ++n_covered;
        int j = index + 1;
        while (j < N && (*points)[index].x + side2 > (*points)[j].x){
            if (abs((*points)[index].y - (*points)[j].y) < side2 &&
                !covered[j])
            {
                ++n_covered;
                covered[j] = true;
```

```
            }
            ++j;
        }
        j = index - 1;
        while (j >= 0 && (*points)[index].x - side2 < (*points)[j].x){
            if (abs((*points)[index].y - (*points)[j].y) < side2 &&
                !covered[j]){
                ++n_covered;
                covered[j] = true;
            }
            --j;
        }
    }
    (*results)[index] = balls_used;
}

/// @brief Cálculo de la dimensión de Hausdorff de un fractal dado como un conjunto
/// discreto de puntos
/// @param points Fractal representado como un conjunto discreto de puntos
/// @param delta Precisión máxima con la que tomar los recubrimientos.
/// Nótese que si la precisión es muy alta el resultado convergerá a lower bound,
/// ya que la dimensión de Hasudorff de cualquier conjunto finito de puntos es siempre 0.
/// @param reps Número de veces que tomar un recubrimiento para buscar el ínfimo
/// @param lower_bound Cota inferior de la dimensión de Hausdorff del fractal
/// @param upper_bound Cota superior de la dimensión de Hausdorff del fractal
/// @param max_search Número de veces que hay que iterar la búsqueda binaria
/// @return Aproximnación de la dimensión de Hausdorff del fractal representado por points
double hausdorff_dimension(
    const vector<dPoint> points,
    double delta,
    int reps,
    double lower_bound,
    double upper_bound,
    int max_search)
{
    // Para paralelizar los cálculos
    vector<size_t> results1(reps);
    vector<size_t> results2(reps);
    vector<std::thread> threads;
    // Primero se calcula el número de bolas para el recubrimiento
    // mínimo con dos diámetros próximos a la precisión pedida
    size_t min_cover1;
    size_t min_cover2;
    double side1 = delta*5; // Precisión algo menor
    double side2 = delta;   // Precisión pedida
    for (int i = 0; i < reps; ++i){
        // Lanzar recubrimientos concurrentes
        threads.push_back(thread(cover, &points, side1, &results1, i));
        threads.push_back(thread(cover, &points, side2, &results2, i));
    }
    // Esperar a todos los threads
    for (auto& th : threads) th.join();

    // Calculamos las bolas necesarias para el recubrimiento mínimo
    min_cover1 = *std::min_element(results1.begin(), results1.end());
    min_cover2 = *std::min_element(results2.begin(), results2.end());

    // Búsqueda binaria
    // Si es creciente se busca hacia arriba
    // Si es decreciente se busca hacia abajo
    // Se deja un número de iteraciones máxima
```

```
    double s;
    for(int i = 0; i < max_search; ++i){
        s = (lower_bound + upper_bound) / 2;

        double difference =
            min_cover2 * pow(side2 * sqrt(2), s) -
            min_cover1 * pow(side1 * sqrt(2), s);
        if (difference == 0) {
            break;
        }
        if (difference > 0) {
            lower_bound = s;
        }
        else{
            upper_bound = s;
        }
    }
    return s;
}

// -----------------------------------------
// |   Creación del Triángulo de Sierpinski   |
// -----------------------------------------

float sign (dPoint p1, dPoint p2, dPoint p3)
{
    return (p1.x-p3.x)*(p2.y-p3.y) - (p2.x-p3.x)*(p1.y-p3.y);
}

bool PointInTriangle (dPoint pt, dPoint v1, dPoint v2, dPoint v3)
{
    float d1, d2, d3;
    bool has_neg, has_pos;

    d1 = sign(pt, v1, v2);
    d2 = sign(pt, v2, v3);
    d3 = sign(pt, v3, v1);

    has_neg = (d1 < 0) || (d2 < 0) || (d3 < 0);
    has_pos = (d1 > 0) || (d2 > 0) || (d3 > 0);

    return !(has_neg && has_pos);
}

vector<dPoint> Sierpinski_points(
    size_t n_points,
    vector<dPoint> v)
{
    vector<dPoint> triangle;
    float x0 = 1;
    float y0 = 1;
    // Se toma un punto inicial aleatorio dentro del triángulo
    // Realmente para obtener una buena aproximación ni siquiera
    // es necesario que el punto inicial esté en el triángulo, pero
    // bueno, no nos cuesta nada y es preferible
    while(!PointInTriangle({x0, y0}, v[0], v[1], v[2])){
        x0 = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
        y0 = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
    }
    dPoint curr = {x0, y0};
    for (size_t i = 0; i < n_points; ++i){
```

```
        // Elegimos un vértice aleatorio y tomamos el siguiente punto
        // como el punto medio entre dicho vértice y el punto actual
        int index = rand() % 3;
        double x_next = (curr.x + v[index].x) / 2;
        double y_next = (curr.y + v[index].y) / 2;
        triangle.push_back({x_next, y_next});
        curr = {x_next, y_next};
    }
    return triangle;
}
bool compare_dPoint(const dPoint& p1, const dPoint& p2){
    if (p1.x == p2.x)
        return p1.y < p2.y;
    return p1.x < p2.x;
}
int main() {
    srand(time(NULL));
    vector<dPoint> vertices;
    // Vértices iniciales del triángulo
    vertices.push_back({0.5, sin(PI/3)});
    vertices.push_back({0,0});
    vertices.push_back({1,0});

    vector<dPoint> triangle = Sierpinski_points(4000000, vertices);
    // Se ordenan según el orden lexicográfico de las coordenadas (x,y)
    sort(triangle.begin(), triangle.end(), compare_dPoint);
    cout << "haursdorff dimension = " <<
        hausdorff_dimension(
            triangle,
            0.0016,
            10,
            1.0,
            2.0,
            10
        ) << '\n';
    return 0;
}
```

Aquí está el código en Python que representa los fractales de las dos formas descritas y reimplementa el cálculo de la dimensión de Hausdorff, pero sin características concurrentes y con un menor rendimiento.

```
import os
import numpy as np
import matplotlib as mpl
mpl.rcParams['animation.ffmpeg_path'] = '/usr/bin/ffmpeg'
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from matplotlib.collections import LineCollection


def recursive_plot_Sierpinski(ax, depth, p1, p2, p3):
    if depth == 0:
        return []
    middle1 = np.array([(p1[0]+p2[0])/2,(p1[1]+p2[1])/2])
    middle2 = np.array([(p2[0]+p3[0])/2,(p2[1]+p3[1])/2])
    middle3 = np.array([(p3[0]+p1[0])/2,(p3[1]+p1[1])/2])
    segs = [[middle1, middle2], [middle2, middle3], [middle3, middle1]]
    segs = segs + recursive_plot_Sierpinski(ax, depth-1, p1, middle1, middle3)
    segs = segs + recursive_plot_Sierpinski(ax, depth-1, middle1, p2, middle2)
    segs = segs + recursive_plot_Sierpinski(ax, depth-1, middle3, middle2, p3)
    if len(segs) > 10000:
        print('plotting')
```

```
            line_segments = LineCollection(np.array(segs), linestyle='solid', color='black')
            ax.add_collection(line_segments)
            segs = []
    return segs


def random_plot_Sierpinski(ax, vertices, n_points):
    x0 = np.random.uniform(vertices[1][0], vertices[2][0])
    y0 = np.random.uniform(vertices[1][1], vertices[0][1])
    curr = np.array([x0, y0])
    points = []
    for _ in range(n_points):
        i = np.random.choice([0,1,2])
        next = np.array([curr[0]+vertices[i][0], curr[1]+vertices[i][1]])/2
        points.append(next)
        curr = next
        if (len(points) > 10000):
            np_points = np.array(points)
            ax.scatter(np_points[:,0], np_points[:,1], s=.01, color='black')
            points = []
    np_points = np.array(points)
    ax.scatter(np_points[:,0], np_points[:,1], s=.01, color='black')

def Sierpinski_points(vertices, n_points):
    x0 = np.random.uniform(vertices[1][0], vertices[2][0])
    y0 = np.random.uniform(vertices[1][1], vertices[0][1])
    curr = np.array([x0, y0])
    points = []
    for _ in range(n_points):
        i = np.random.choice([0,1,2])
        next = np.array([curr[0]+vertices[i][0], curr[1]+vertices[i][1]])/2
        points.append(next)
        curr = next
    return np.array(points)

p1 = np.array([0.5, np.sin(np.pi/3)])
p2 = np.array([0, 0])
p3 = np.array([1,0])
vertices = np.array([p1, p2, p3])

fig, ax = plt.subplots()
a = np.array([[p1, p2], [p2,p3], [p3,p1]])
line_segments = LineCollection(a, linestyle='solid', color='black')
ax.add_collection(line_segments)
segs = recursive_plot_Sierpinski(ax, 10, p1, p2, p3)
line_segments = LineCollection(np.array(segs), linestyle='solid', color='black')
ax.add_collection(line_segments)
ax.set_aspect('equal', adjustable='box')
# plt.show()

fig, ax = plt.subplots()
random_plot_Sierpinski(ax, vertices, 4000000)
ax.set_aspect('equal', adjustable='box')
plt.show()

# Apartado ii)

def get_random_uncovered(covered, N):
    i = np.random.randint(0, N)
    while covered[i%N]:
        i += 1
```

```
    return i%N

def cover(points, side):
    side2 = side / 2
    N = points.shape[0]
    covered = [False]*N
    n_covered = 0
    balls_used = 0
    while n_covered < N:
        index = get_random_uncovered(covered, N)
        balls_used += 1
        covered[index] = True
        n_covered += 1
        j = index + 1
        while j < N and points[index][0] + side2 > points[j][0]:
            if  abs(points[index][1] - points[j][1]) < side2 and not covered[j]:
                n_covered += 1
                covered[j] = True
            j += 1
        j = index - 1
        while j >= 0 and points[index][0] - side2 < points[j][0]:
            if  abs(points[index][1] - points[j][1]) < side2 and not covered[j]:
                n_covered += 1
                covered[j] = True
            j -= 1
    return balls_used



def hausdorff_dimension(
        points,
        epsilon,
        reps,
        lower_bound,
        upper_bound,
        max_search):
    side1 = epsilon * 5
    side2 = epsilon
    min_cover1 = np.min([cover(points,  side1) for _ in range(reps)])
    min_cover2 = np.min([cover(points,  side2) for _ in range(reps)])

    for _ in range(max_search):
        s = (lower_bound + upper_bound) / 2

        difference = min_cover2 * pow(side2 * np.sqrt(2), s) -\
                     min_cover1 * pow(side1 * np.sqrt(2), s)
        if difference == 0:
            break
        if difference > 0:
            lower_bound = s
        else:
            upper_bound = s
    return s


points = Sierpinski_points(vertices, 100000)
sorted = np.lexsort((points[:,1],points[:,0]))
triangle = points[sorted]

print(f'Dimensión de Hausdorff = {hausdorff_dimension(triangle, 0.01, 10, 1.0, 2.0, 10)}')
```