Nome: Lara Amélia Maia de Freitas **Matrícula:** 2024005912
Matemática Discreta - Turma TM1

Os Fractais desenvolvidos no TP são:

i) Fractal Onda Senoidal 1 de von Koch:

Axioma: F

θ = $\pi/3$

Regra: F →F+F--FF++F-F

ii) Fractal Preenchimento de Espaço de Hilbert:

Axioma: X

θ = $\pi/2$

Regras: X → -YF+XFX+FY

Y → +XF-YFY-FX+

iii) Fractal Hexagonal Gosper:

Axioma: XF

θ = $\pi/3$

Regras: X → X+YF++YF-FX--FXFX-YF+

Y → -FX+YFYF++YF+FX--FX-Y

1. Descrição das implementações e estratégias escolhidas

Em termos gerais, para todos os 3 fractais foi utilizada uma implementação iterativa para a geração das strings. Foi combinada com o uso de alocação dinâmica de memória para permitir que fosse possível gerar estágios mais avançados dos fractais.

i) Fractal Onda Senoidal 1 de von Koch

Para a geração do fractal Onda Senoidal 1 de von Koch foram utilizadas duas funções: uma que calcula o tamanho final do fractal e uma função que gera a string em si.

```
//Calcula o tamanho da string final do fractal i (Onda Senoidal 1 de von Koch)
float calcula_tamanho_koch(int iteracoes)
{
   float tamanho = 0;
   tamanho = (1.0/5.0)*((11*pow(6, iteracoes)) - 6);
   return tamanho;
}
```

A função acima, calcula_tamanho_koch, faz o cálculo de qual será o tamanho/comprimento final da string na iteração dada pelo usuário, que é um inteiro passado como parâmetro para a função. A expressão para o cálculo desse tamanho foi derivada da equação de recorrência que descreve a quantidade de

símbolos no fractal em uma dada iteração, e o processo para a sua obtenção está detalhado na parte da documentação que trata das equações de recorrência.

```
//Utiliza a função para o tamanho do fractal de Koch (i) na iteração desejada
//"Prepara" as strings que serão utilizadas na geração do fractal de Koch
int tamanho_koch = calcula_tamanho_koch(iteracoes);
char* resultado_koch = (char*)malloc(tamanho_koch + 1);
if(resultado_koch == NULL)
{
    printf("Erro na alocação de memória para resultado_koch\n");
    return EXIT_FAILURE;
}
resultado_koch[0] = '\0';
char* temp_koch = (char*)malloc(tamanho_koch + 1);
if(temp_koch == NULL)
{
    printf("Erro na alocação de memória para temp_koch\n");
    return EXIT_FAILURE;
}
temp_koch[0] = '\0';
```

No trecho de código acima (que está na main do programa), a função calcula_tamanho_koch é chamada e o tamanho obtido (tamanho_koch) é utilizado para alocar memória dinamicamente para as strings que serão utilizadas na geração do fractal. Ambas são inicializadas como strings vazias.

```
//Utiliza a função gerar_fractal_von_koch para gerar a string do fractal de Koch (i)
strcat(resultado_koch, "F");
gerar_fractal_von_koch(iteracoes, resultado_koch, temp_koch);
//Faz a impressão do resultado obtido no arquivo de saída correspondente, i.txt
FILE *saida_koch = fopen("i.txt", "w");
if(saida_koch == NULL)
{
    printf("Houve um erro para abrir o arquivo i.txt\n");
    return EXIT_FAILURE;
}
else
{
    fprintf(saida_koch, "%s", resultado_koch);
    fclose(saida_koch);
}
free(temp_koch);
free(resultado_koch);
```

No trecho de código acima (que está na main do programa), tem-se a inicialização de resultado_koch com o axioma ‘F’ e a chamada da função gerar_fractal_von_koch. Após essa chamada, estando a string produzida por ela armazenada na string resultado_koch, é criado o arquivo de saída i.txt, em que é impresso o resultado utilizando a função fprintf. Por fim, o arquivo é fechado e a memória alocada para as strings temp_koch e resultado_koch é liberada, finalizando-se o algoritmo para produção do fractal de von Koch.

```c
//Função utilizada para gerar o fractal i (Onda Senoidal 1 de von Koch)
void gerar_fractal_von_koch(int iteracoes, char *resultado_koch, char* temp_koch)
{
    for(int j = 0; j < iteracoes; j++)
    {
        temp_koch[0] = '\0';
        int tamanho_string = strlen(resultado_koch);
        for (int i = 0; i < tamanho_string; i++)
        {
            if (resultado_koch[i] == 'F')
            {
                strcat(temp_koch, "F+F--FF++F-F");
            }
            else
            {
                strncat(temp_koch, &resultado_koch[i], 1);
            }
        }
        strcpy(resultado_koch, temp_koch);
    }
}
```

A função gerar_fractal_von_koch é a função que, de fato, faz a geração da string correspondente ao estágio desejado do fractal. Para tanto, ela recebe como parâmetros o número de iterações (um inteiro) e as strings resultado_koch e temp_koch, que foram alocadas dinamicamente na main antes da chamada da função. Não é necessário que o tamanho de nenhuma das strings seja expandido no decorrer do processo (utilizando o comando realloc, por exemplo), uma vez que as strings são alocadas já com o tamanho final do fractal e, logo, são capazes de comportar tanto esse resultado final quanto quaisquer dos outros resultados intermediários obtidos na geração do fractal (que obviamente têm comprimento menor).

A string resultado_koch é recebida já contendo o axioma ‘F’. A string temp_koch é inicializada como uma string vazia a cada iteração do loop externo (índice j), uma vez que é utilizada para gerar os resultados de estágios seguintes do fractal. Um loop interno (índice i) analisa cada caractere de resultado_koch (que representa o estado atual do fractal). A string temp_koch armazenará as alterações que são feitas nesse loop. Caso o caractere atual seja ‘F’, ele é substituído por “F+F--FF++F-F” (o que consiste na regra de geração do fractal, F→ F+F--FF++F-F) utilizando a função strcat, que concatena essa regra de formação à string temp_koch. Caso o caractere atual de resultado_koch não seja um ‘F’, ele é apenas copiado sem quaisquer alterações para a string temp_koch, utilizando a função strncat. Ao fim do processo (ou seja, quando o loop interno acaba), resultado_koch é atualizada, recebendo a string construída em temp_koch. O loop externo continua e repete esse processo até que todas as iterações desejadas tenham sido calculadas. A função

opera diretamente sobre as strings, que são arrays passados por parâmetro e, logo, seu tipo de retorno é void.

ii) Fractal Preenchimento de Espaço de Hilbert

A geração do Fractal Preenchimento de Espaço de Hilbert tem funcionamento similar ao processo utilizado na geração do Fractal de von Koch e utiliza 3 funções: calcula_tamanho_hilbert, gerar_fractais_ii_iii e remove_xy.

```
//Calcula o tamanho da string final do fractal ii (Preenchimento de Espaço de Hilbert)
float calcula_tamanho_hilbert(int iteracoes)
{
    float tamanho = 0;
    tamanho = ((10.0/3.0)*pow(4, iteracoes)) - 7.0/3.0;
    return tamanho;
}
```

A função acima, calcula_tamanho_hilbert, é utilizada para calcular o tamanho final da string (incluindo os símbolos X e Y) na iteração definida pelo usuário, que é passada como parâmetro. Similarmente ao caso do Fractal de von Koch, a expressão para o cálculo foi obtida através da equação de recorrência e a sua obtenção está detalhada na parte da documentação que trata das equações de recorrência.

```
//Utiliza a função para o tamanho do fractal de Hilbert (ii) na iteração desejada
//"Prepara" as strings que serão utilizadas na geração do fractal de Hilbert
int tamanho_hilbert = calcula_tamanho_hilbert(iteracoes);
char* resultado_hilbert = (char*)malloc(tamanho_hilbert + 1);
if(resultado_hilbert == NULL)
{
    printf("Erro na alocação de memória para resultado_hilbert\n");
    return EXIT_FAILURE;
}
resultado_hilbert[0] = '\0';
char* temp_hilbert = (char*)malloc(tamanho_hilbert + 1);
if(temp_hilbert == NULL)
{
    printf("Erro na alocação de memória para temp_hilbert\n");
    return EXIT_FAILURE;
}
temp_hilbert[0] = '\0';
```

No trecho de código acima, que está na main do programa, é chamada a função calcula_tamanho_hilbert e o seu resultado, armazenado na variável tamanho_hilbert, é utilizado para alocar memória dinamicamente para as strings resultado_hilbert e temp_hilbert, que serão utilizadas na geração do fractal. Ambas são inicializadas como strings vazias.

```c
//Utiliza as funções gerar_fractais_ii_iii e remove_xy para obter o fractal de Hilbert (ii)
char axioma_hilbert[5] = "X";
strcat(resultado_hilbert, axioma_hilbert);
char regrax_hilbert[12] = "-YF+XFX+FY-";
char regray_hilbert[12] = "+XF-YFY-FX+";
gerar_fractais_ii_iii(iteracoes, resultado_hilbert, regrax_hilbert, regray_hilbert, temp_hilbert);
remove_xy(resultado_hilbert);
//Faz a impressão do resultado obtido no arquivo de saída correspondente, ii.txt
FILE *saida_hilbert = fopen("ii.txt", "w");
if(saida_hilbert == NULL)
{
    printf("Houve um erro para abrir o arquivo ii.txt\n");
    return EXIT_FAILURE;
}
else
{
    fprintf(saida_hilbert, "%s", resultado_hilbert);
    fclose(saida_hilbert);
}
free(temp_hilbert);
free(resultado_hilbert);
```

No trecho de código acima, que também está na main do programa, são definidos o axioma e as regras de formação do fractal, armazenados, respectivamente, em axioma_hilbert, regrax_hilbert e regray_hilbert. A string resultado_hilbert recebe o axioma do fractal e são chamadas as funções gerar_fractais_ii_iii e remove_xy, que são as funções utilizadas para gerar o fractal em si. Posteriormente, é criado o arquivo de saída ii.txt e impresso o resultado obtido com as funções, armazenado em resultado_hilbert, utilizando-se a função fprintf. Por fim, o arquivo é fechado e a memória dinamicamente alocada para as strings temp_hilbert e resultado_hilbert é liberada, finalizando o algoritmo para a geração do fractal ii, Preenchimento de Espaço de Hilbert.

```c
//Função utilizada para gerar os fractais ii e iii (Espaço de Hilbert e Hexagonal Gosper), com os símbolos X e Y
void gerar_fractais_ii_iii(int iteracoes, char *resultado, char* regrax, char* regray, char* temp)
{
    for(int j = 0; j < iteracoes; j++)
    {
        temp[0] = '\0';
        for(int i = 0; i < strlen(resultado); i++)
        {
            if(resultado[i] == 'X')
            {
                strcat(temp, regrax);
            }
            else if(resultado[i] == 'Y')
            {
                strcat(temp, regray);
            }
            else
            {
                //concatena o caractere analisado à string temporária
                strncat(temp, &resultado[i], 1);
            }
        }
        strcpy(resultado, temp);
    }
}
```

A função acima, como indicado pelo nome e pela lista de parâmetros menos específica, é mais generalista e é utilizada para gerar tanto o fractal ii (Preenchimento de Espaço de Hilbert) quanto o fractal iii (Hexagonal Gosper de Mandelbrot). A função recebe como parâmetros um inteiro (correspondente ao número de iterações inserido pelo usuário), uma string resultado (que armazena o estágio atual do fractal e o resultado final da função), uma string regrax (que representa a regra de substituição para símbolos X que aparecem na string), uma string regray (que representa a regra de substituição para símbolos Y) e uma string temp (que é utilizada como auxiliar, sendo utilizada na geração do estágio seguinte do fractal).

O funcionamento da função também é similar ao da função gerar_fractal_koch, estando adaptada para a existência de mais de uma regra de formação e para a generalização para uso em mais de um fractal. Não é necessário redimensionar o tamanho das strings resultado e temp (com realloc, por exemplo), pois elas já são alocadas na main com o tamanho final da string (contendo os símbolos X e Y) e, logo, têm tamanho suficiente para armazenar a string final e quaisquer das strings de outros estágios, que têm tamanho menor.

Cada caractere da string resultado, que na primeira chamada contém o axioma do fractal que se deseja gerar, é analisado através do loop interno. As alterações feitas com base nessa análise que geram o fractal são armazenadas na string temp. Caso o caractere atual seja um símbolo X, ele é substituído pela string regrax (que corresponde à regra de substituição de símbolos X para o fractal que se deseja gerar), sendo ela concatenada à string temp (que armazena as mudanças e gera o estágio seguinte do fractal) utilizando-se a função strcat. O mesmo ocorre para quando o caractere atual é Y, mas, nesse caso, é substituído pela string regray. Caso o caractere não seja nenhum dos 2 (ou seja, é F, + ou -), ele apenas é copiado para a string temp sem sofrer nenhuma alteração, utilizando a função strncat. Quando o loop interno (índice i) acaba, resultado_hilbert é atualizada e recebe o conteúdo de temp_hilbert, ou seja, passa a armazenar a string recém-calculada e a "carrega" para a iteração seguinte do loop externo. Caso ainda não tenha atingido o limite superior, o loop mais externo (índice j) segue para a iteração seguinte, resetando temp e seguindo para a produção do próximo estágio do fractal. A função recebe as strings que são modificadas por parâmetro e, logo, opera diretamente sobre elas, de forma que o seu tipo de retorno pode ser void.

A string gerada pela função gerar_fractais_ii_iii e armazenada em resultado_hilbert ainda contém os símbolos X e Y, de forma que utilizamos a função remove_xy para remover esses caracteres e obter a string final contendo apenas os símbolos F, + e -.

```c
//Função utilizada para remover os símbolos X e Y na geração dos fractais ii e iii
void remove_xy(char *resultado)
{
    int i = 0, j = 0;
    while(resultado[i] != '\0')
    {
        if((resultado[i] != 'X') && (resultado[i]) != 'Y')
        {
            resultado[j] = resultado[i];
            j++;
        }
        i++;
    }
    resultado[j] = '\0';
    return;
}
```

Como mencionado, a função remove_xy é utilizada para remover os caracteres X e Y da string gerada pela função gerar_fractais_ii_iii. Ela também é relativamente genérica e é utilizada tanto na construção do fractal ii (Preenchimento de Espaço de Hilbert) quanto do fractal iii (Hexagonal Gosper), recebendo apenas uma string resultado como parâmetro. Mais uma vez, como se trata de uma string que é recebida por parâmetro, a função opera diretamente sobre ela e seu tipo de retorno é void.

É utilizado um loop do tipo while com 2 índices: o índice i representa a posição atual a ser analisada na string, enquanto o índice j indica qual é a posição na string em que o próximo caractere válido (que não seja X ou Y) deve ser inserido. Enquanto não se atinge o final da string resultado (quando resultado[i] == '\0'), checa-se se o caractere na posição i de resultado é diferente de X e de Y e, se for o caso, o caractere é considerado válido e é copiado para a string resultado na posição j. Sempre que isso ocorre o valor de j é incrementado em um, indicando que avançou-se uma posição e garantindo que o próximo caractere válido será adicionado em uma posição também válida. Se o caractere na posição i de resultado for inválido (ou seja, é X ou Y), é ignorado e incrementa-se somente i, passando a analisar o próximo caractere. Quando o loop termina (ou seja, quando atinje-se o final da string resultado, indicado pelo caractere '\0'), adicionamos '\0' na última posição j válida, sinalizando o fim da string. O comando return é utilizado para garantir que a chamada da função é concluída.

iii) Hexagonal Gosper de Mandelbrot

O fractal selecionado para o item (iii) é chamado de Hexagonal Gosper, foi escrito por Paul Borke e é atribuído a Mandelbrot. Disponível em: https://paulbourke.net/fractals/lsys/

Como se trata de um fractal que também conta com duas regras de formação baseadas em substituições de símbolos X e Y, o seu processo de geração é

praticamente o mesmo do fractal (ii), sendo utilizadas 3 funções para tanto: calcula_tamanho_gosper, gerar_fractais_ii_iii e remove_xy.

```c
//Calcula o tamanho da string final do fractal iii (Hexagonal Gosper de Mandelbrot)
float calcula_tamanho_gosper(int iteracoes)
{
    float tamanho = 0;
    tamanho = (pow(7, iteracoes)*10.0/3.0) - 4.0/3.0;
    return tamanho;
}
```

A função acima, calcula_tamanho_gosper, recebe como parâmetro um inteiro iterações que representa o número de iterações selecionado pelo usuário. A partir disso, a função calcula qual é o tamanho final da string (incluindo símbolos X e Y) gerada para esse número de iterações. Assim como nos outros 2 casos, a expressão para o cálculo foi derivada da equação de recorrência e o seu processo de obtenção está detalhado na parte da documentação que trata das equações de recorrência. A função retorna esse tamanho.

```c
//Utiliza a função para o tamanho do fractal Gosper (iii) na iteração desejada
//"Prepara" as strings que serão utilizada sna geração do fractal Gosper
int tamanho_gosper = calcula_tamanho_gosper(iteracoes);
char* resultado_gosper = (char*)malloc(tamanho_gosper + 1);
if(resultado_gosper == NULL)
{
    printf("Erro na alocação de memória para resultado_gosper\n");
    return EXIT_FAILURE;
}
resultado_gosper[0] = '\0';
char* temp_gosper = (char*)malloc(tamanho_gosper + 1);
if(temp_gosper == NULL)
{
    printf("Erro na alocação de memória para temp_gosper\n");
    return EXIT_FAILURE;
}
temp_gosper[0] = '\0';
```

A função calcula_tamanho_gosper é chamada e o seu retorno é armazenado na variável tamanho_gosper. Esse valor é utilizado para alocar memória dinamicamente para as string resultado_gosper e temp_gosper, que serão utilizadas na geração dos fractais. Ambas são inicializadas como strings vazias.

```c
//Utiliza as funções gerar_fractais_ii_iii para gerar o fractal Gosper (iii)
char axioma_gosper[5] = "XF";
strcat(resultado_gosper, axioma_gosper);
char regrax_gosper[50] = "X+YF++YF-FX--FXFX-YF+";
char regray_gosper[50] = "-FX+YFYF++YF+FX--FX-Y";
gerar_fractais_ii_iii(iteracoes, resultado_gosper, regrax_gosper, regray_gosper, temp_gosper);
remove_xy(resultado_gosper);
//Faz a impressão do resultado obtido no arquivo de saída correspondente, iii.txt
FILE *saida_gosper = fopen("iii.txt", "w");
if(saida_gosper == NULL)
{
    printf("Houve um erro para abrir o arquivo iii.txt\n");
    return EXIT_FAILURE;
}
else
{
    fprintf(saida_gosper, "%s", resultado_gosper);
    fclose(saida_gosper);
}
free(temp_gosper);
free(resultado_gosper);
```

Nesse trecho de código, que está na main do programa, são definidos o axioma do fractal e as regras de substituição para X e Y, armazenados, respectivamente, nas strings axioma_gosper, regrax_gosper e regray_gosper. A string resultado_gosper recebe o axioma pela função de concatenação strcat antes de serem chamadas as funções gerar_fractais_ii_iii e remove_xy para a geração do fractal. Após o processo de geração ser finalizado, é criado o arquivo de saída “iii.txt” e impresso nele o resultado obtido pela aplicação das funções (e armazenado em resultado_gosper), através da função fprintf. O arquivo é fechado e a memória alocada para as strings resultado_gosper e temp_gosper é liberada, sinalizando o fim do algoritmo de geração do fractal (iii) Hexagonal Gosper.

A função gerar_fractais_ii_iii é utilizada na geração do fractal (iii) também mas, nesse caso, recebe como parâmetros um inteiro iteracoes, a string resultado_gosper, a string regrax_gosper, a string regray_gosper e a string temp_gosper. O seu funcionamento é exatamente o mesmo daquele descrito para o fractal (ii) Preenchimento de Espaço de Hilbert. A função remove_xy também é utilizada para a construção do fractal (iii). Assim como no caso do fractal (ii), ela é chamada após a chamada para gerar_fractais_ii_iii, quando a string resultado_gosper contem o resultado final do fractal, incluindo os símbolos X e Y. Assim, remove_xy recebe resultado_gosper como parâmetro, sendo o seu funcionamento, nesse caso, também idêntico ao descrito para o fractal (ii).

Parte 2 - Estratégias para implementação dos fractais

1. **Versão iterativa com uso de arquivos intermediários**

Um dos principais pontos positivos dessa estratégia se relaciona com a memória. Nesse caso, não é necessário lidar com a memória RAM estritamente, o que reduz a tendência a erros relacionados a esse fator (sejam eles erros na alocação, erros de acesso a áreas indevidas ou uso de espaço não alocado, segmentation faults, etc). O algoritmo também se torna mais escalável, uma vez que a string do fractal não depende totalmente da memória RAM para o seu armazenamento. Ele se torna relativamente menos limitado pelo fator de espaço e, com isso, pode ser utilizado para computar resultados de iterações maiores. A geração de arquivos também ocupa memória, mas não é necessariamente a memória RAM, o que faz com que esse continue a ser um fator importante, mas menos limitante.

Outro ponto importante é que, ao registrar cada estágio em um arquivo distinto, o acesso ao processo de geração do fractal é facilitado. Utilizando esse método, é possível analisar os resultados de cada etapa de forma mais explícita, o que favorece a detecção de erros e a realização de análises para atividades como construção de equações de recorrência. Além disso, caso ocorram erros durante a execução e o algoritmo seja adaptado para tanto, a geração do fractal poderá ser retomada a partir do último arquivo que foi produzido, ou seja, é possível recuperar as informações já existentes. Nesse sentido, os arquivos podem ser armazenados de forma que não seja necessário reexecutar o algoritmo para produzir um estágio que já foi computado e os arquivos já existentes (sejam eles gerados pela execução atual ou por outras) podem ser reutilizados também.

Um ponto negativo é a necessidade de se gerenciar múltiplos arquivos. Podem ocorrer erros de acesso, leitura ou escrita, sendo importante adicionar ao algoritmo tratamentos para esses casos, o que pode aumentar a sua complexidade. Nesse mesmo sentido, esse método também pode levar à ocupação de grande quantidade de espaço em disco, o que é “melhor” que ocupar muito espaço na memória RAM (que costuma ser mais limitada e também pode influenciar na qualidade/velocidade da execução do algoritmo) mas também pode gerar problemas. Outra desvantagem é que os processos de leitura e gravação em arquivos podem ser demorados, em especial nos casos em que são utilizadas strings muito grandes.

2. Versão recursiva

Devido à natureza auto-similar dos fractais, o funcionamento de uma implementação recursiva parece ser natural para a geração de seus estágios (o processo de formação do fractal é inerentemente recursivo). No entanto, ainda existem pontos positivos e negativos associados a essa estratégia.

A implementação recursiva pode ser mais recomendável em termos de legibilidade do código, uma vez que não requer, por exemplo, que existam diversos loops aninhados para a geração de seus estágios. Também não é necessário lidar com arquivos e com possíveis problemas de entrada, saída ou escrita, o que torna o código menos complexo e, em geral, mais simples.

Alguns dos principais pontos negativos quando se utiliza a estratégia recursiva estão relacionados à memória. A cada chamada da função, possíveis variáveis locais e outros dados são realocados, o que pode gerar um consumo excessivo de memória caso esteja se tratando de um número grande de chamadas da função. Outro problema que pode surgir com quantidades expressivas de chamadas da função (o que ocorre quando deseja-se calcular o fractal em um nível muito avançado/profundo) é o estouro de pilha, ou seja, pode ser ultrapassado o limite de espaço destinado à pilha e, com isso, surgem erros na execução do programa. A sobrecarga associada a essa situação também pode prejudicar o desempenho na execução do algoritmo, tornando-o mais lento. Mais uma das questões que envolvem a memória diz respeito à finitude da memória RAM, sobre a qual o programa é executado e são feitas as chamadas da função. Como a memória RAM tende a ser limitada, o algoritmo fica mais restringido pelo fator de espaço, correndo o risco de não ter memória suficiente para realizar as ações necessárias (em especial no caso de cálculo de estágios mais profundos) e, com isso, gerando uma parada.

Como as chamadas recursivas de funções não podem ser acompanhadas explicitamente (exceto com o uso de alguns métodos de depuração que permitem tem algum controle sobre a execução), a detecção e correção de erros pode ser dificultada quando se opta por essa estratégia. O acompanhamento da execução, em geral, tende a ser mais difícil do que quando se utiliza a iteratividade.

3. Iteratividade com alocação dinâmica de memória

Uma possibilidade é utilizar a iteratividade combinada a alocação dinâmica de memória, o que elimina a necessidade de se lidar com múltiplos arquivos e com os possíveis erros que podem surgir a partir deles.

Nesse caso, a string que representará o fractal pode ser alocada dinamicamente e, à medida que o estágio do fractal e o comprimento da string aumentam, ela pode ser redimensionada (utilizando realloc, por exemplo), acompanhando o crescimento da string. Caso o algoritmo seja adaptado para tanto, os estágios intermediários do fractal também podem ser armazenados, seja em outras strings ou também em arquivos.

Um ponto negativo que surge nessa estratégia é a susceptibilidade à ocorrência de erros relacionados à memória. Ao manipular as strings e alocar espaço dinamicamente para elas, podem ocorrer erros como acesso a áreas indevidas de memória, erros na alocação ou desalocação, segmentation faults, memory leaks (quando a desalocação de regiões de memória alocadas não é feita), dentre outros. Também pode ocorrer o esgotamento da memória, já que, como já foi discutido, a memória RAM (onde o programa roda e onde é alocação dinâmica de memória é

feita) é limitada, e o seu consumo pode gerar problemas de desempenho ou tornar o algoritmo limitado a estágios de geração do fractal mais baixos. Outro possível problema é a existência de loops aninhados para a geração da string do fractal, sendo a sua legibilidade e manutenção ambas difíceis.

4. Equações de Recorrência

i) Fractal Onda Senoidal 1 de von Koch

Para obter a equação de recorrência, o algoritmo desenvolvido foi utilizado para fornecer informações acerca da quantidade de símbolos total e de cada símbolo específico, gerando as informações da tabela:

n	**#F**	**#símbolos**
0	1	1
1	6	12
2	36	78
3	216	474
4	1296	2850

Na tabela, **n** representa o número da iteração, **#F** o número de caracteres F e **#símbolos** o número total de símbolos.

A partir dela, fica claro que a equação que descreve o número de segmentos F gerados é dada por $T_F(n) = 6^n$, que pode ser expressa também como

$$T_F(n) = \begin{cases} T_F(0) = 1 \\ T_F(n) = 6T_F(n-1) \quad n \geq 1. \end{cases}$$

Já para a quantidade de símbolos, observa-se que, a cada aumento de uma unidade em n, há um aumento em #símbolos tal que:

$$T_s(n) = \begin{cases} T_S(0) = 1, \\ T_S(1) = 12, \\ T_S(n) = (T_S(n-1) - T_S(n-2)) \cdot 6 + T_S(n-1) \quad , n \geq 2 \end{cases}$$

Essa expressão também pode ser dada como:

$$T_s(n) = \begin{cases} T_S(0) = 1, \\ T_S(1) = 12, \\ T_S(n) = 7 \cdot T_S(n-1) - 6 \cdot T_S(n-2) \quad , n \geq 2 \end{cases}$$

A partir dela, é realizada a expansão que gera a expressão utilizada na função calcula_tamanho_koch. A equação característica associada a ela é $x^2 - 7x + 6 = 0$, gerando os resultados x = 6 e x = 1. Logo, a equação geral será da forma $T_S(n) = A.6^n + B$, com as constantes A e B obtidas a partir das condições iniciais da geração. Resolve-se o sistema de equações:

(1) $A + B = 1$

(2) $6A + B = 12$, obtemos que $A = 11/5,\ B = -6/5$

Substituindo esses valores, chegamos a $T_S(n) = 1/5.(11.6^n - 6)$.

ii) Fractal Preenchimento de Espaço de Hilbert

Similarmente ao caso do fractal (i), o próprio programa foi utilizado para fornecer informações acerca da quantidade de símbolos (total e específica) em cada estágio do fractal. Assim, foi possível gerar a seguinte tabela:

n	**#F**	**#(X, Y)**	**#(+, -)**	**#símbolos (com X e Y)**	**#símbolos (sem X e Y)**
0	0	1	0	1	0
1	3	4	4	11	7
2	15	16	20	51	35
3	63	64	84	211	147
4	255	256	340	851	595

Na tabela, **n** representa o número do estágio do fractal, **#(X, Y)** o número de símbolos X e Y, **#(+, -)** o número de símbolos + e -, **#símbolos (com X e Y)** a quantidade de símbolos na string considerando X e Y, e **#símbolos (sem X e Y)** a

quantidade de símbolos na string desconsiderando X e Y (contando somente F, + e -).

A expressão para a quantidade de segmentos F gerados é dada por:

$$T_F(n) = \begin{cases} T_F(0) = 0, \\ T_F(n) = 4 \cdot T_F(n-1) + 3, \quad n \geq 1 \end{cases}$$

Essa mesma expressão também pode ser dada por $T_F(n) = 4^n - 1$.

Já para a quantidade total de símbolos em um certo estágio (com X e Y), tem-se:

$$T_S(n) = \begin{cases} T_S(0) = 1, \\ T_S(1) = 11, \\ T_S(n) = 4 \cdot (T_S(n-1) - T_S(n-2)) + T(n-1), \quad n \geq 2. \end{cases}$$

Essa expressão também pode ser vista como:

$$T_S(n) = \begin{cases} T_S(0) = 1, \\ T_S(1) = 11, \\ T_S(n) = 5T_S(n-1) - 4T_S(n-2), \quad n \geq 2. \end{cases}$$

Para a quantidade total de símbolos, sem contar X e Y, obtemos:

$$T_SF(n) = \begin{cases} T_SF(0) = 0, \\ T_SF(n) = T_SF(n-1) + 7 \cdot 4^{n-1}, \quad n \geq 1 \end{cases}$$

A partir da equação de recorrência para o número total de símbolos (com X e Y), é possível obter a expressão utilizada na função calcula_tamanho_hilbert. A equação característica associada é $x^2 - 5x + 4 = 0$, com resultados x = 4 e x = 1. Logo, a solução geral da equação de recorrência tem a forma $T_S(n) = A.4^n + B$. Os valores das constantes A e B são obtidos a partir das condições iniciais e resolve-se o sistema:

(1) $A + B = 1$

(2) $4A + B = 11$, obtendo os resultados $A = 10/3$ *e* $B = -7/3$.

Substituindo os valores, chega-se à equação final $T_S(n) = 10/3.4^n - 7/3$.

iii) Fractal Hexagonal Gosper de Mandelbrot

Para obter a equação de recorrência do fractal (iii), também foi utilizado o programa para fornecer informações acerca da quantidade de símbolos em cada estágio. A partir delas, foi gerada a tabela:

n	#F	#(X, Y)	#(+, -)	#símbolos (com X e Y)	#símbolos (sem X e Y)
0	1	1	0	2	1
1	7	7	8	22	15
2	49	49	64	162	113
3	343	343	456	1142	799
4	2401	2401	3200	8002	5601

A expressão para o cálculo do número de segmentos F é visivelmente dada por $T_F(n) = 7^n$, que pode ser escrita também como:

$$T_F(n) = \begin{cases} T_F(0) = 1, \\ T_F(n) = 7 \cdot T_F(n-1), \quad n \geq 1 \end{cases}$$

Já para a quantidade total de símbolos em um estágio, incluindo os símbolos X e Y, temos:

$$T_S(n) = \begin{cases} T_S(0) = 2, \\ T_S(1) = 22, \\ T_S(n) = 7 \cdot (T_S(n-1) - T_S(n-2)) + T_S(n-1), \quad n \geq 2 \end{cases}$$

Essa expressão também pode se reescrita para:

$$T_S(n) = \begin{cases} T_S(0) = 2, \\ T_S(1) = 22, \\ T_S(n) = 8 \cdot T_S(n-1) - 7 \cdot T_S(n-2), \quad n \geq 2 \end{cases}$$

A partir dela, é possível obter a expressão utilizada função calcula_tamanho_gosper. A equação característica associada é $x^2 - 8x + 7 = 0$, com os resultados x=7 e x=1. Logo, a solução geral é da forma $T_S(n) = A.7^n + B$, os valores das constantes A e B são obtidos pelas condições iniciais com a resolução do sistema:

(1) $A + B = 2$

(2) $7A + B = 22$, com os resultados $A = 10/3$ *e* $B = -4/3$.

Logo, a equação final tem a forma $T_S(n) = 10/3.7^n - 4/3$.

5. Complexidade dos algoritmos

A complexidade dos algoritmos pode ser considerada com base nas funções que geram os fractais, que contém as operações mais relevantes na execução do código.

i) Fractal Onda Senoidal 1 de von Koch: $\theta(12^n)$.

Na produção do Fractal Onda Senoidal 1 de von Koch, conforme a sua regra de produção, cada caractere 'F' é substituído por uma sequência de 12 caracteres, sendo 6 'F' (considera-se que o F substituído é "preservado", adicionando-se 11 caracteres) e outros 6 símbolos + e -. O loop externo na função gerar_fractal_von_koch é executado n vezes (sendo n o número de iterações) e o loop interno, a cada iteração, percorre toda a string correspondente ao estágio anterior. Assim, o custo está relacionado com essas duas operações e é dado por $\boldsymbol{\theta}(12^n)$.

ii) Fractal Preenchimento de Espaço de Hilbert: $\theta(4^n)$.

As regras de geração para X e para Y ambas contém 11 caracteres, ou seja, a cada vez que um caractere X ou Y aparece na string, é substituído por um conjunto de 11 caracteres. O loop externo na função gerar_fractais_ii_iii é executado n vezes (sendo n o número de iterações escolhido pelo usuário) e o loop interno percorre toda a string correspondente ao estágio anterior de geração do fractal. Pela tabela presente na seção de equações de recorrência foi possível perceber que, a cada estágio do fractal, o número de elementos X e Y presentes na string é 4^n, o que indica que essa é a quantidade de operações de substituição/concatenação realizadas na string. Para remover os caracteres X, Y, também são necessárias 4^n operações. Logo, o custo assintótico é **$\theta(4^n)$.**

iii) Fractal Hexagonal Gosper de Mandelbrot: $\theta(7^n)$

Para o Fractal Hexagonal Gosper, as regras de substituição para X e Y ambas contém 21 caracteres, ou seja, cada caractere X ou Y é substituído por um conjunto de 21 caracteres para gerar o fractal. Pela tabela exibida na seção de equações de recorrência, tem-se que, em cada estágio, o número de elementos X ou Y é 7^n, sendo n o estágio atual de geração do fractal. Assim, em cada estágio, são realizadas 7^n operações de substituição/concatenação de strings. O valor de 7^n também está diretamente relacionado com a quantidade de caracteres a serem retirados pela função remove_xy. Logo, o custo assintótico é **$\theta(7^n)$.**

6. Softwares para desenho dos fractais

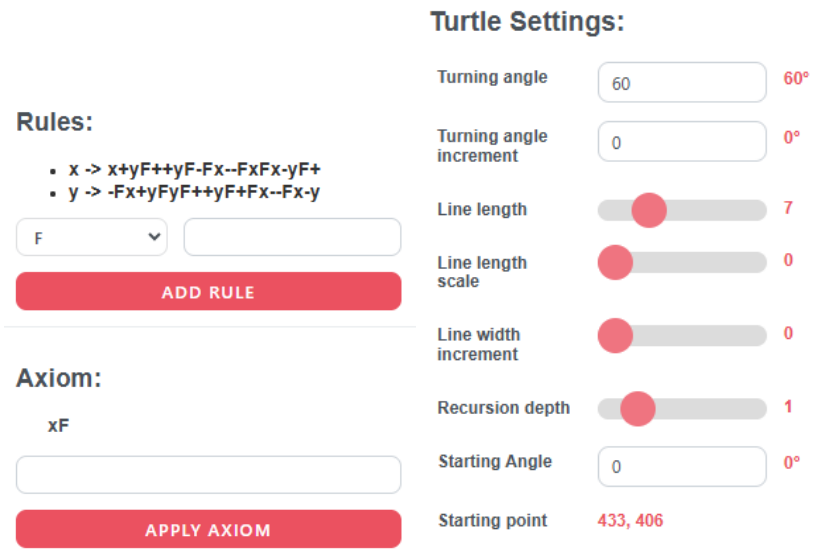
1) L-System Studio

O L-System Studio é um software gratuito desenvolvido por Paul Borke e disponível em: https://raw.githack.com/hunorg/L-System-Studio/main/index.html. Um manual sobre o uso do site, que inclui também alguns dos fractais criados pelo seu desenvolvedor (incluindo o fractal Hexagonal Gosper utilizado neste trabalho) está disponível em: https://paulbourke.net/fractals/lsys/.

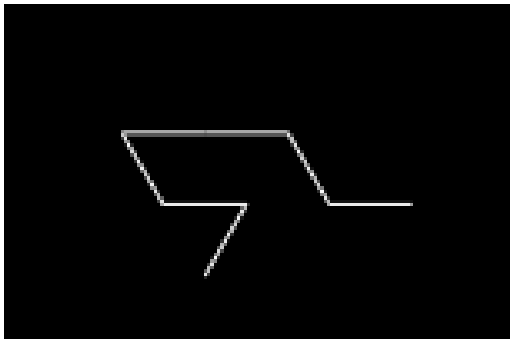
O aspecto mais interessante sobre o L-System Studio é que ele permite gerar os fractais através do próprio L-Sistema que os define, exatamente no formato em que foi especificado para este trabalho, por exemplo. Assim, ao utilizar o software, tem-se um campo para a definição do axioma do fractal, campos para a definição das regras de substituição (em que, em geral, são empregados símbolos similares aos utilizados no TP, como F, + e -, além de também estarem disponíveis diversos outros símbolos e comandos que podem ser usados na geração dos fractais) e campos chamados de "Turtle settings", que correspondem às configurações relacionadas ao desenho dos fractais em si. Nessa área, incluem-se configurações como "turning angle" (que indica o ângulo de "virada" da caneta), "line length" (comprimento das linhas desenhadas) e "recursion depth" (que indica o estágio que se deseja desenhar do fractal). Ainda existem outras configurações mais avançadas que permitem desenhar fractais mais complexos, possibilitando ações como incrementos no comprimento das linhas e no ângulo de virada. Ao lado direito da área de definição das configurações, tem-se uma tela em que é gerado o desenho do fractal de acordo com as especificações dadas. Dessa forma, é possível acompanhar o processo de criação à medida que o fractal é gerado. Também é possível visualizar alguns fractais que já estão "prontos" e são apresentados randomicamente ao clicar no ícone de dado, na parte superior da região de configurações.

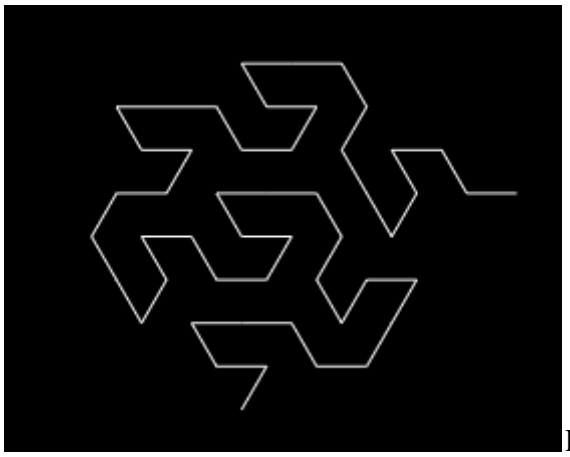
Como mencionado, o principal ponto positivo do software é o uso explícito dos L-Sistemas para desenhar os fractais. A partir dessa tática, ele oferece uma interface relativamente simples e intuitiva, sem que seja necessário, por exemplo, escrever código para gerar as imagens. Um ponto negativo é que o seu alcance é um pouco limitado, uma vez que só fornece resultados até o nível 6 de profundidade de recursão, ou seja, até a sexta iteração para a geração de qualquer fractal. É válido pontuar que alguns dos símbolos utilizados no TP também têm significado diferente do empregado no softwares: no L-System Studio, o símbolo "+" indica que se deve virar à esquerda, enquanto o símbolo "-" indica que se deve virar à direita.

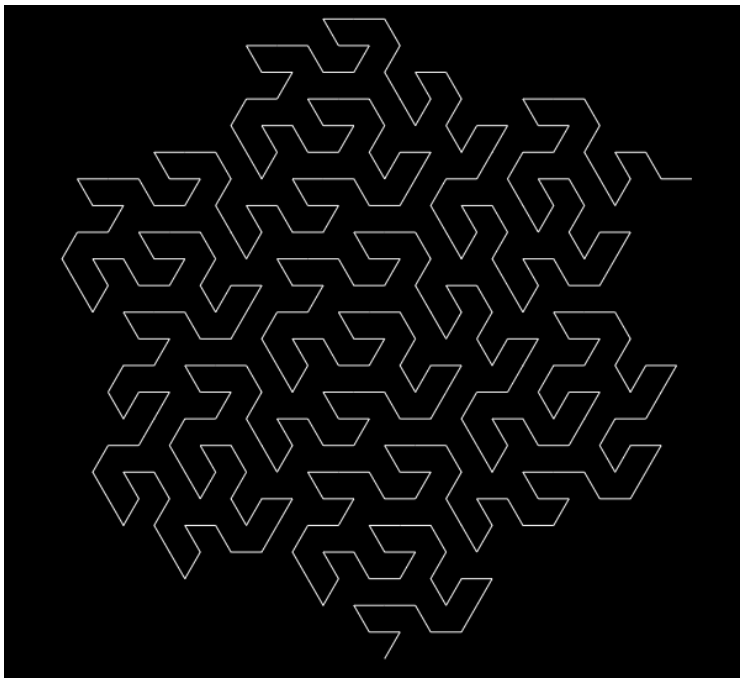
O L-System Studio é o software selecionado para apresentar os desenhos dos 4 primeiros estágios do fractal (iii), que são apresentados a seguir juntamente das configurações utilizadas para a sua geração:



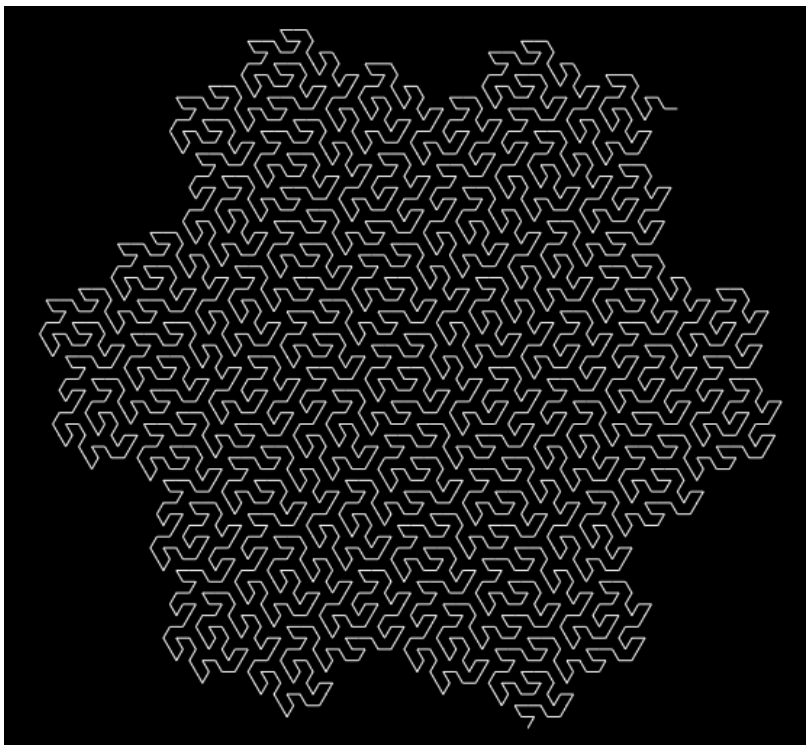
Nota-se que o parâmetro “recursion depth” é alterado para indicar o estágio do fractal que se deseja gerar.


Estágio 1 do fractal Hexagonal Gosper.


Estágio 2 do fractal Hexagonal Gosper.



Estágio 3 do fractal Hexagonal Gosper.



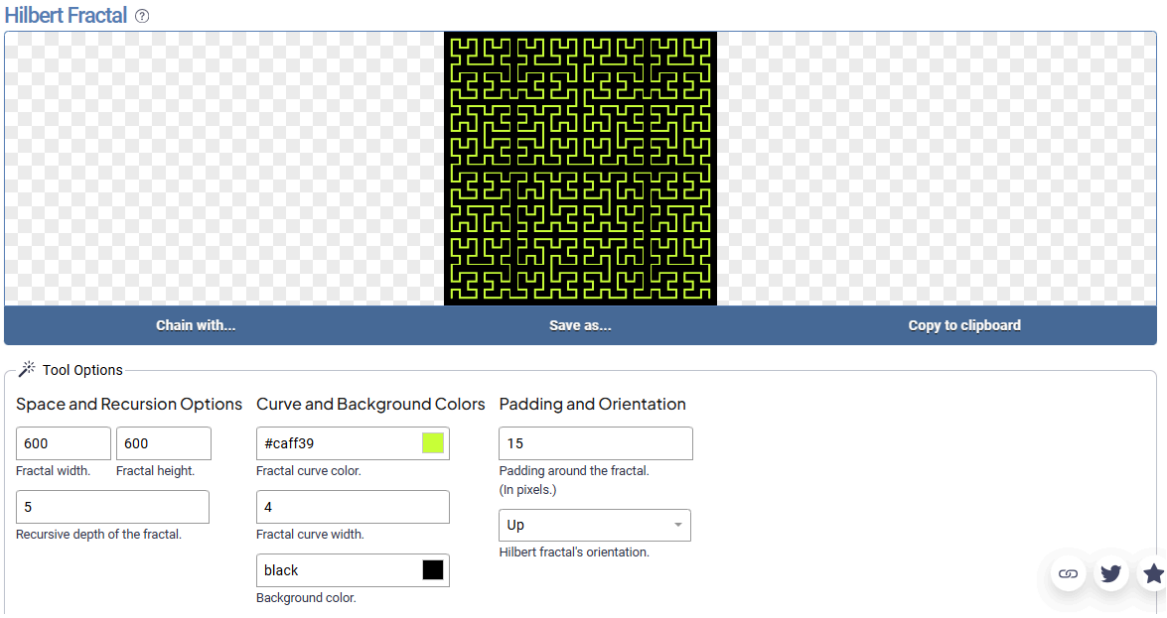
Estágio 4 do fractal Hexagonal Gosper.

2) onlinetools.com

O site onlinetools.com é uma página de acesso gratuito que agrega diversas funções para tarefas matemáticas em geral. Disponível em: https://onlinetools.com/

Para acessar o campo de geração de fractais, basta buscar “fractal” no campo de pesquisa “search all tools”. Aparecerão várias opções de fractais com configurações já prontas que, ao serem selecionados, já fornecem uma imagem para o fractal escolhido em um dado estágio de sua geração. Alguns dos muitos fractais prontos já disponíveis são o Preenchimento de Espaço de Peano, o próprio Hexagonal Gosper, a curva de Hilbert, flocos de neve de Koch e muitos outros. Ao acessar algum dos fractais, é possível selecionar algumas das configurações para sua geração, como as cores utilizadas, o nível de recursão/estágio de geração, tamanho das linhas desenhadas, etc. Em seguida, é possível copiar ou salvar a imagem.

Um ponto positivo é que o software oferece diversos fractais que já vem “prontos”, permitindo o usuário modificar configurações relacionadas à apresentação da sua imagem. A interface também é acessível e intuitiva, e, ao acessar um fractal, são oferecidos exemplos de outros fractais similares. No entanto, não é possível criar fractais e nem está acessível como funciona o processo de geração dos fractais fornecidos (como os L-Sistemas que os definem, por exemplo), de forma que o usuário tem menos controle sobre o processo de construção do fractal e não pode desenvolver os seus próprios fractais “do 0”. O software também tende a falhar para a geração de fractais em níveis de recursão maiores, produzindo resultados pouco satisfatórios. Ainda assim, é uma boa ferramenta para visualização de resultados de construção.

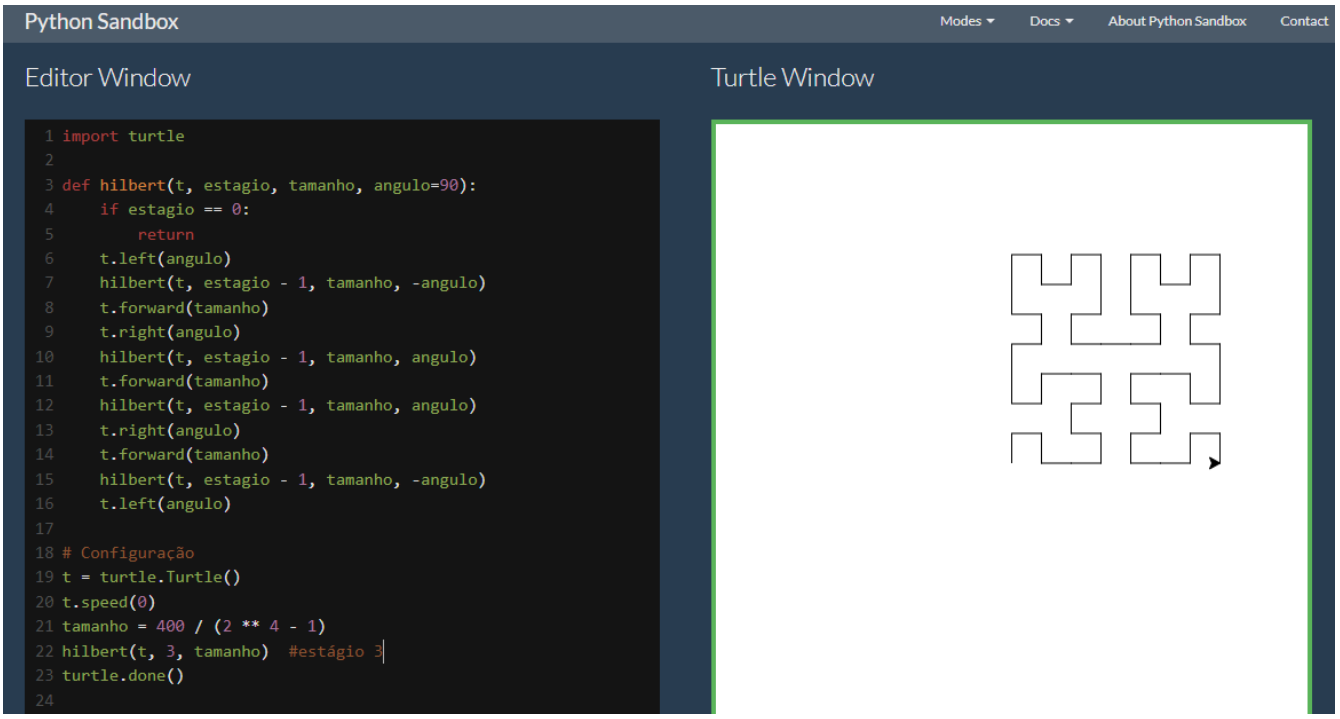


Exemplo de uso do site para gerar imagens de fractais. Nesse caso, é desenhado o fractal de Preenchimento de Espaço de Hilbert, sendo possível alterar configurações como orientação do fractal, tamanho das linhas, nível de recursão, e cores utilizadas.

3)Python - Biblioteca Turtle

A linguagem de programação Python oferece bibliotecas como Turtle e matplotlib que permitem desenhar fractais. A biblioteca turtle permite gerar um "turtle graphic" para desenhar o fractal, sendo o seu funcionamento baseado na produção de código que "guia" um cursor sobre um plano cartesiano, realizando o desenho desejado pelo programador. A interface é relativamente simples e intuitiva, mas requer que o usuário tenha algum conhecimento prévio sobre a linguagem Python e sobre programação em geral, uma vez que o "caminho" do cursor é estabelecido no código.

É possível utilizar a ferramenta online Python Sandbox tanto para a escrita do código quanto para a sua execução, em que o desenho é produzido em uma área à direita do código fornecido. É possível acompanhar o caminho realizado pelo cursor à medida que o fractal é desenhado, viabilizando o acompanhamento explícito desse processo. Um exemplo de uso da biblioteca turtle para gerar o estágio 3 do fractal de Preenchimento de Espaço de Hilbert segue abaixo:



No campo "Editor Window" deve ser inserido o código que gerará o fractal. No campo ao lado, "Turtle Window", o desenho correspondente é gerado ao pressionar o botão de "play" que fica logo abaixo da janela de editor. Como já mencionado, a interface é relativamente intuitiva, especialmente caso já se tenha algum

conhecimento prévio acerca da linguagem Python e de programação em geral. Ainda assim, se não for caso, como se trata de um método e linguagem populares, estão disponíveis diversos tutoriais sobre o seu uso, bem como uma ampla documentação que acompanha a biblioteca Turtle. O software também é gratuito e, como demonstrado pelo uso do Python Sandbox, pode ser utilizado online.

Python Sandbox (no “turtle mode” que foi utilizado para a geração do desenho acima) está disponível em: https://pythonsandbox.com/turtle

Documentação para a biblioteca Turtle (que fornece informações sobre o funcionamento da biblioteca, como utilizá-la e outras informações úteis) está disponível em: https://docs.python.org/3/library/turtle.html