MAC2010 - Laboratório de Métodos Numéricos

Relatório Exercício Programa I

Fellipe Souto Sampaio, 7990422
Renan Fichberg, 7991131
Prof: Ernesto Birgin

Universidade de São Paulo
Dezembro de 2016

Conteúdo

Capítulo 1

Aritmética de Ponto Flutuante

1.1 Questão 1

Seja:

$X = \pm S \times 2^E$, onde:

- $S = (0.b_2 b_3 b_4 \ldots b_2 4)$
- $\frac{1}{2} < S < 1$
- $-128 < E < 127$

1.1.1 A

Para que X seja o maior número de ponto flutuante do sistema precisamos que três condições sejam verdade:

1. S tem que ter a maior mantissa possível
2. E tem que ter o maior valor possível
3. X tem que ser positivo

Então $X = +(0.111\ldots 1) \times 2^{126}$ é o maior número positivo de ponto flutuante do sistema

1.1.2 B

Para que X seja o menor número positivo de ponto flutuante do sistema precisamos que três condições sejam verdade:

1. S tem que ter a menor mantissa possível, de forma que $\frac{1}{2} < S < 1$
2. E tem que ter o menor valor possível
3. X tem que ser positivo

Então $X = +(0.100\ldots 1) \times 2^{-127}$ é o menor número positivo de ponto flutuante do sistema

1.1.3 C

Para resolver essa questão podemos tentar escrever os primeiros números inteiros no dado sistema

- $X_1 = +(0.100\dots0) \times 2^1 = 1.000\dots0 = 1$
- $X_2 = +(0.100\dots0) \times 2^2 = 10.000\dots0 = 2$
- $X_3 = +(0.110\dots0) \times 2^2 = 11.000\dots0 = 3$
- $X_4 = +(0.100\dots0) \times 2^3 = 100.000\dots0 = 4$

Entretanto verificando a definição de S vemos que $\frac{1}{2} < S < 1$, portanto para ser um número válido no sistema $S \neq (0.100\dots0)$. Concluimos que o menor inteiro que não é exatamente representável no sistema é o número 1.

1.2 Questão 2

Na precisão single temos que $p = 23$.

1.2.1 A

Seja: $X = (\frac{1}{10})_{10} = (0.00011001100\dots)_2 \times 2^0 = (1.10011001100\dots)_2 \times 2^{-4}$

Round down

Para X_- truncarmos a dízima no b_{p-1}, ou seja no vigésimo segundo dígito da mantissa. Temos então:

$round(X) = X_- = (1.1001100110011001100110)_2 \times 2^{-4}$

Round up

Para X_+ adicionamos 1 no vigésimo segundo dígito da mantissa. Temos então:

$round(X) = X_+ = (1.1001100110011001100111)_2 \times 2^{-4}$

Round towards zero

Como $X = \frac{1}{10} > 0$ então $round(X) = X_-$

Round to nearest

Como $X_- < X < X_+$ e $X_- - X < X_+ - X$ verificamos que X está mais próximo de X_-, portanto $round(X) = X_-$

1.2.2 B

Seja $X = (1 + 2^{-25}) = (1.0000000000000000000000001)_2 \times 2^0$

Round down

Para X_- truncarmos a dízima no b_{p-1}, ou seja no vigésimo segundo dígito da mantissa. Temos então:

$round(X) = X_- = (1.0000000000000000000000)_2 \times 2^0$

Round up

Para X_+ adicionamos 1 no vigésimo segundo dígito da mantissa. Temos então:

$round(X) = X_+ = (1.0000000000000000000001)_2 \times 2^0$

Round towards zero

Como $X = (1 + 2^{25}) > 0$ então $round(X) = X_-$

Round to nearest

Como $X_- < X < X_+$ e $X_- - X < X_+ - X$ verificamos que X está mais próximo de X_-, portanto $round(X) = X_-$

1.2.3 C

Seja $X = (2^{130}) = (1.0000000000000000000000000)_2 \times 2^{130}$

Round down

No formato single temos que $N_{max} \approx 2^{128}$, como nosso E é maior que o limite temos então:

$round(X) = N_{max}$

Round up

Como X_- tem o maior valor representado consideramos que :

$round(X) = X_+ = \infty$

Round towards zero

Como $X = (2^{130}) > 0$ então $round(X) = X_-$

Round to nearest

Não é possível comparar X com ∞ portanto $round(X) = X_-$

1.3 Questão 3

Temos que:

$(1 \oplus x) = round(1 + x) = (1 + x)(1 + \delta) = 1$ pelo teorema enunciado em Overton <2001>. Precisamos que $x \approx 0$ para não influenciar na soma, para isso na precisão single x deve assumir o menor valor positivo possível de um ponto flutuante, que é $x = +(1.000\ldots00) \times 2^{-126}$. Esse valor é muito pequeno, e sua soma com 1 será arredondado para 1 também.

Na precisão dupla temos a mesma situação, só que $x = +(1.000\ldots00) \times 2^{-1022}$, valor menor ainda e que será somado com 1 e resultará em 1 após o arredondamento.

1.4 Questão 4

Comutatividade

$(X \oplus Y) = round(X + Y)$[1] $= round(Y + X) = (Y \oplus X)$, portanto a operação $\oplus$ é comutativa

Associatividade

$(X \oplus (Y \oplus Z)) = (X \oplus round(Y + Z)) = round(X + (1 + Y + Z) \times (1 + \delta)) = (1 + X + (1 + Y + Z) \times (1 + \delta)) \times (1 + \delta) = \ldots$

A soma não é associativa, pois a cada arredondamento existe perda de precisão da conta original, isso faz com que se a ordem das somas variar, de uma operação para outra, o arredondamento final não seja o mesmo.

[1] Comutatividade aritmética

Capítulo 2

Método de Newton

A implementação do nosso Método de Newton se baseou na disponível em **?**. Para calcular a sequência de valores X_k aplicamos o seguinte pseudo-algoritmo:

Algorithm 1 Método de Newton

procedure NEWTON
    $X_K \leftarrow \infty + i$
    $X_{k+1} \leftarrow X_K$
    $mx\_it \leftarrow 15$
    $\delta \leftarrow 1.0 \times 10^{-8}$

    while $abs(X_{k+1} - X_k > \delta)||(it < mx\_it)$ **do**
        $it++$
        $X_K = X_{k+1}$
        if $f^{'}(X_k) == 0$ **then**
            O método não está definido para esse ponto, então consideramos que ele falhou
            return $\infty + \infty \times i$
        else
            $X_{k+1} \leftarrow X_K - \frac{f(X_k)}{f^{'}(X_k)}$

            if $it > mx\_it$ **then**
                O método não está convergindo, então consideramos que ele falhou
                return $\infty + \infty \times i$
    A raíz encontrada é $X_K = X + Y \times i$
    return $X + Y \times i$

Escolhemos um número fixo de iterações igual a 15, em testes práticos demonstrou-se que a função convergia em no máximo 12 iterações. Repetimos esse algoritmo para todos os pontos da nossa partição escolhida, e para cada resposta fazemos um mapeamento da raíz para um inteiro.

Um polinômio de grau N com raízes distintas mapeia N+1 cores diferentes, a cor adicional, neste caso preto, serve para identificar os pontos em que o método falhou.

Realizamos testes com diferentes funções, variando a quantidade de pixeis na imagem. Os gráficos podem ser encontrados no Apêndice (.1) desse relatório. O programa foi implementado no arquivo **new_method_basins.m**

Capítulo 3

Encontrando Todas as Raízes de Funções

No enunciado da terceira questão é mencionado sobre uma função f(x) de classe C^2 definida em um intervalo [a, b]. Foi, então, suposto por nós que a e b são números reais, e por ser de classe C^2, a função f(x) é garantidamente contínua.

Para usar o programa, é necessário fazer as devidas modificações:

1. Mudar o valor da variável *choice* para um dos valores: 1, 2 ou 3.
   (a) Para choice = 1, o programa considerará a função $f(x) = 2cosh(\frac{x}{4}) - x$
   (b) Para choice = 2, o programa considerará a função $f(x) = \frac{sin(x)}{x}$ se $x \neq 0$ ou $f(x) = 1$ se $x = 0$
   (c) Para choice = 3, o programa considerará um polinomio, que deve ser escrito de acordo com a sintaxe do Octave, na função *function3*, no código do programa. Convém que a função que representa o polinomia seja, no mínimo, de classe C^2, conforme exigência do enunciado (isso fica a cargo do usuário e não do programa).
2. Mudar os valores das variáveis *lower_ edge* e *higher_ edge* (estas fazem os papeis de a e b do enunciado, respectivamente).
3. Mudar o valor da variável n_inter para o número de partições (sub-intervalos) em que o intervalo [a, b] deve ser dividido.
4. Mudar a tolerância *tol* do erro para aceitar o valor calculado pelo método de Newton como significativamente próximo ao valor real da raíz.

Finalmente, atribuídos os valores iniciais, o programa vai realizar a tarefa que foi designada a nós no enunciado da seguinte maneira:

- Será invocado o método *intervals*, que é onde tudo acontecerá. Deve-se escolher o tamanho dos passos (= steps, caso não goste do que está escrito no código)
- A variável *range* conterá todos os pontos x de f(x) que deverão ser testados para encontrar a raíz
- A variável *sub_ interval* serve para monitorar qual é o número do intervalo em que o ponto x que está sendo testado está
- A variável *sub_ inter_ length* guarda o tamanho das partições, enquanto as variáveis *sub_ lower_ edge* e *sub_ higher_ edge* guardam os limites da participação atual (em que o programa se encontra em tempo de execução)

Note que para um intervalo [a, e] dividido em 4 partes, as partições de [a, e] são [a, b];]b, c];]c, d],]d, e] respectivamente, além disso apenas a primeira partição possui o inteiro limitante inferior.

Em seguida, o primeiro ponto do nosso intervalo é avaliado em f(x) separadamente para obtermos o sinal inicial de f(x). Se dermos sorte e o primeiro valor já for zero, significa que primeiro ponto é uma raiz. Neste caso, haverá uma impressão do intervalo, o número do intervalo e o valor da raíz no terminal. Note que valor 1 representa sinal não negativo e o valor 2 representa sinal negativo.

Obtido o sinal, o programa entra em um loop que vai do segundo ao penúltimo valor de x. Calculamos o novo valor $f(x_1)$, considerando que x_1 é o ponto seguinte da sequência após x. Com isso temos os possíveis resultados:

- Se $f(x_1) = 0$ então uma raíz foi encontrada em x_1, e nesse caso o sinal é considerado positivo, as informações são impressar e o laço já recomeça a próxima iteração, sem rodar o método de Newton
- Se houver uma mudança de sinal de f(x) para $f(x_1)$ significa que existe uma raíz entre esses dois pontos, e nesse caso aplicamos o método de Newton para buscar uma aproximação do valor real da raiz considerando a tolerância tol

Se a cada iteração o decrescimento não for significativo (isto é, nosso ponto de partida x não é tão bom e, portanto, não estamos conseguindo aproximações expressivas do valor real da raíz), o método de Newton será interrompido e um novo valor do x atual será buscado pelo método da biseção (que fará 3 iterações apenas, conforme o enunciado pede).

Encontrado o novo valor de x pelo método da biseção, usamos este valor como entrada do método de Newton e assim obteremos, eventualmente, a nova raíz até então desconhecida. Ao encontrar a raíz, assim como nas vezes anteriores, será impresso no terminal as suas informações.

Isso é repetido até que chegamos, finalmente, ao final da iteração do penúltimo valor de x a ser testado, finalizando assim o loop principal do programa. Após isso é checado separadamente o último valor de x, considerando o penúltimo valor de x.

O código pertinente a esta parte está bastante comentado, então entendendo esta explicação e acompanhando o código deve ser mais que o suficiente para entender a implementação sem maiores dificuldades.

Realizamos a execução do programa para as três possíveis choises do programa, os resultados dos testes podem ser encontrado no Apêncide (.2) desse relatório. O programa foi implementado no arquivo **n_roots_function.m**

Bibliografia

[Overton 2001] Overton, Michael L.: **Numerical Computing with IEEE Floating Point Arithmetic**. University City Science Center, Philadelphia : SIAM - Society for Industrial and Applied Mathematics, 2001

Apêndice

.1 Bacias de convergência

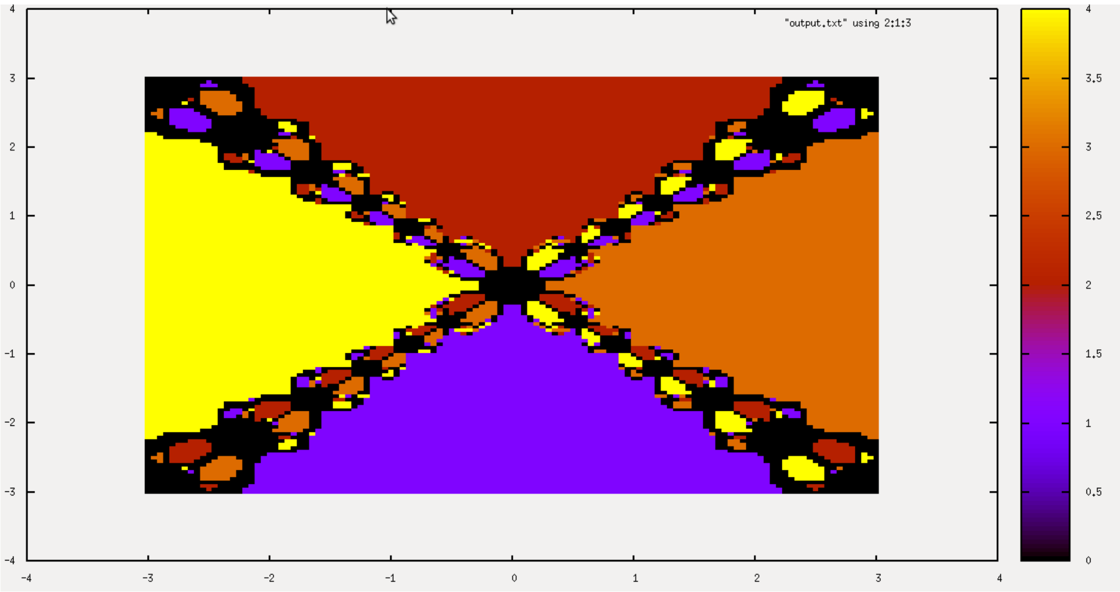

Figura 1: $f(x) = x^4 - 1$ *com intervalo* $= [-3 : 0.05 : 3]$, *total de 14641 pontos*

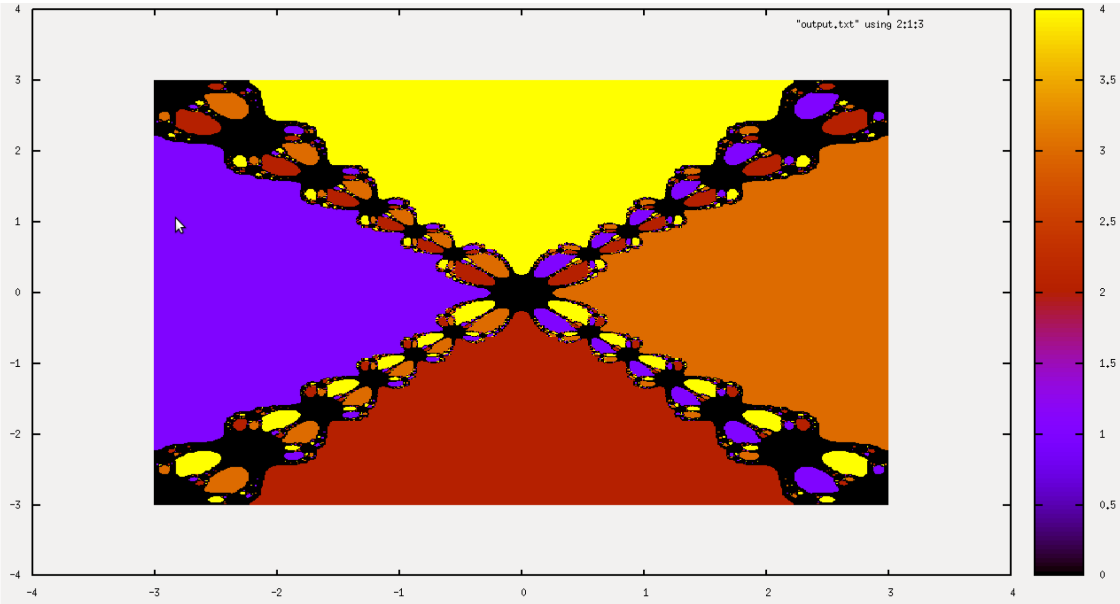

Figura 2: $f(x) = x^4 - 1$ *com intervalo* $= [-3 : 0.01 : 3]$, *total de 361201 pontos*

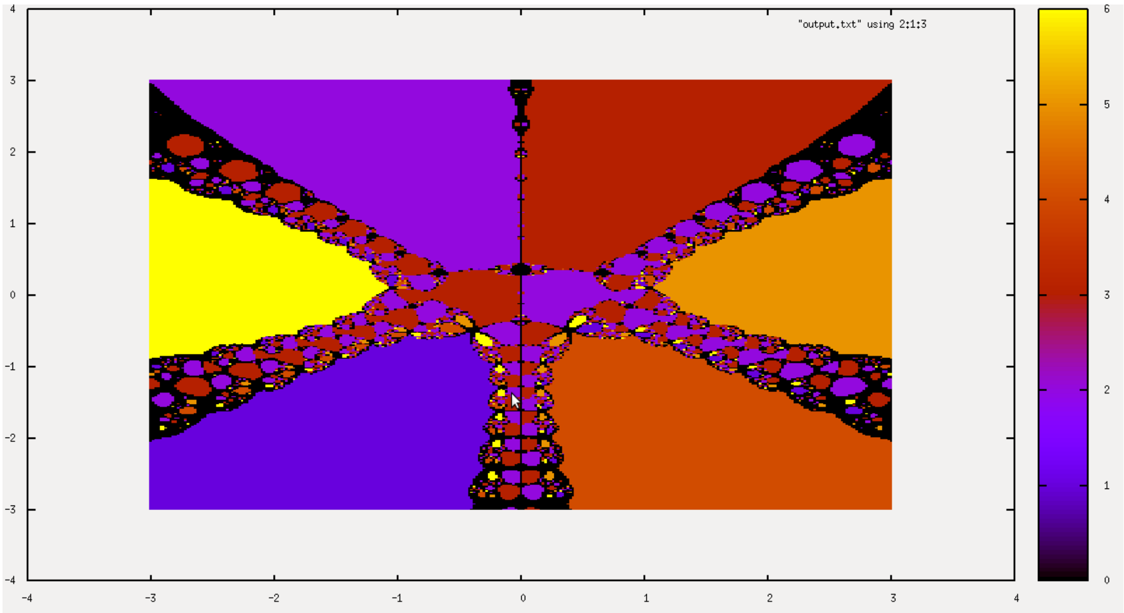

Figura 3: *$f(x) = x^6 + 1 * x^4 - 1 * x^2 - 2 * x^1 + 2$ com intervalo $= [-3 : 0.02 : 3]$, total de 90601 pontos*

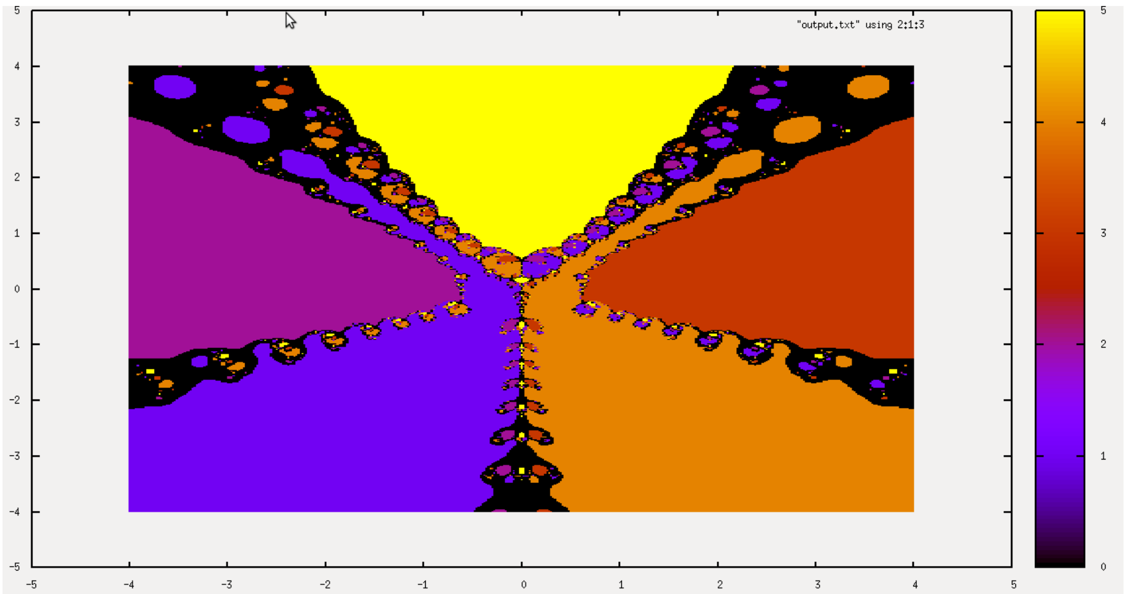

Figura 4: *$f(x) = 3 * x^5 + 2 * x^4 + 1 * x^3 - 1 * x^2 - 2 * x^1 - 3$ com intervalo $= [-4 : 0.02 : 4]$, total de 160801 pontos*

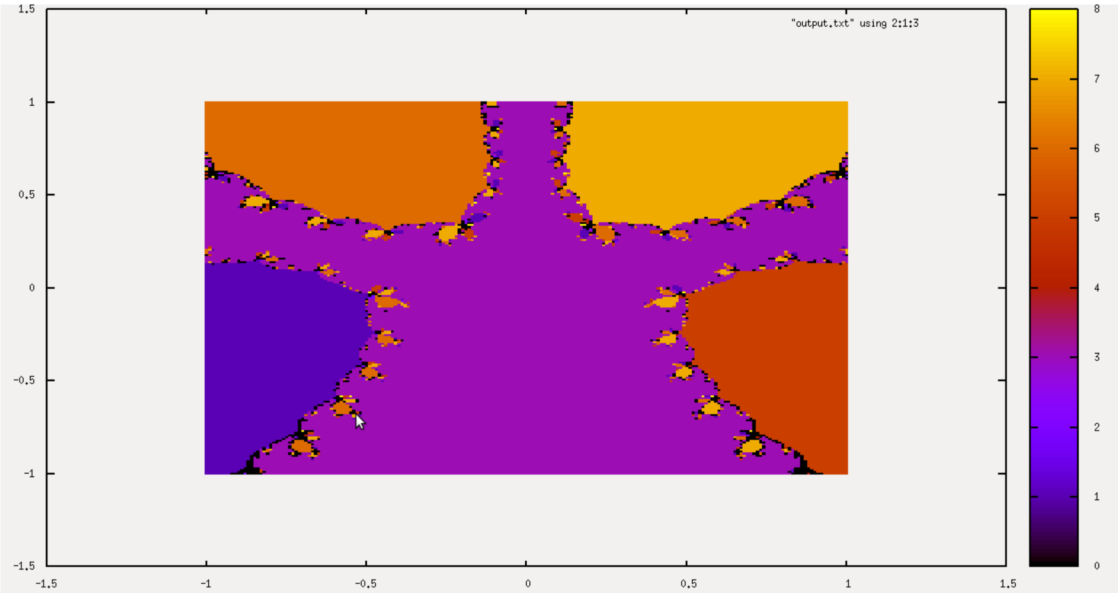

Figura 5: $f(x) = 1 * x^8 + 1 * x^6 + 5 * x^5 - 4 * x^4 + 3 * x^3 - 2 * x^2 + 1 * x^1$ *com intervalo* $= [-1 : 0.01 : 1]$, *total de 40401 pontos*

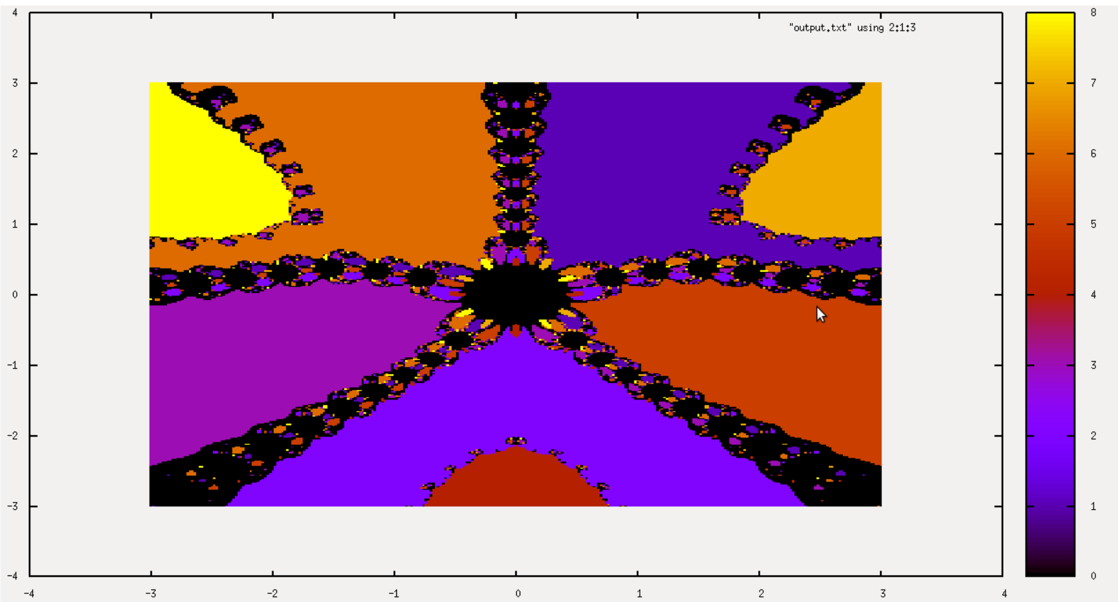

Figura 6: $f(x) = 1 * x^8 + 15 * x^5 + 16$ *com intervalo* $= [-3 : 0.02 : 3]$, *total de 90601 pontos*

.2 Todas as raízes de uma função

```
$ octave n_roots_function.m
Choise: 1
Interval: [a:step:b] -- [ -10 : 0.050000 : 10 ]
Points: 401
Interval #7: ]2, 4] has root 2.35755105
Interval #10: ]8, 10] has root 8.50719957

$ octave n_roots_function.m
Choise: 2
Interval: [a:step:b] -- [ -10 : 0.050000 : 10 ]
Points: 401
Interval #1: ]-10, -8] has root -9.42477796
Interval #2: ]-8, -6] has root -6.28318530
Interval #4: ]-4, -2] has root -3.14159265
Interval #7: ]2, 4] has root 3.14159265
Interval #9: ]6, 8] has root 6.28318531
Interval #10: ]8, 10] has root 9.42477796

$ octave n_roots_function.m
Choise: 3
Interval: [a:step:b] -- [ -10 : 0.050000 : 10 ]
Points: 401
Interval #4: ]-4, -2] has root -3.00000000
Interval #6: ]0, 2] has root 1.00000000
f(x) = 1*x^2 + 2*x^1 - 3

$ octave n_roots_function.m
Choise: 3
Interval: [a:step:b] -- [ -10 : 0.050000 : 10 ]
Points: 401
Interval #4: ]-4, -2] has root -2.30277564
Interval #5: ]-2, 0] has root -1.61803399
Interval #6: ]0, 2] has root 0.61803399
Interval #6: ]0, 2] has root 1.30277564
f(x) = 1*x^4 + 2*x^3 - 3*x^2 - 4*x^1 + 3

$ octave n_roots_function.m
Choise: 3
Interval: [a:step:b] -- [ -10 : 0.050000 : 10 ]
Points: 401
Interval #6: ]0, 2] has root 0.67078673
Interval #6: ]0, 2] has root 0.84877121
f(x) = 1*x^6 + 1*x^5 + 1*x^4 + 2*x^3 - 3*x^2 - 4*x^1 + 3
```