MAC0210: Relatório EP 1

18 de Setembro de 2016

Nathan Benedetto Proença - 8941276
Victor Sena Molero - 8941317

Sumário

Parte 1: Aritmética de Ponto Flutuante

Questão 1 (3.11)

Suponha que temos um sistema de representação de ponto flutuante com base 2 e,

$$x = \pm S \times 2^E,$$

$$\text{com } S = (0.1b_2b_3b_4 \dots b_{24}),$$

$$\text{i.e, } \frac{1}{2} \leq S < 1$$

onde o expoente $-128 < E < 127$.

a) Qual é o maior número de ponto flutuante desse sistema?

Resposta. $2^{126} - 2^{101}$, basta preencher todos os bits (de b_2 até b_{24}) e escolher o maior expoente possível. □

b) Qual é o menor número de ponto flutuante positivo desse sistema?

Resposta. 2^{-128}, basta escolher a menor mantissa possível (0.1) e o menor expoente possível (-127). □

c) Qual é o menor inteiro positivo que não é exatamente representável nesse sistema?

Resposta. $2^{24} + 1$, basta escolher a menor mantissa não representável $(0.10\dots01)$ e o menor expoente para o qual ela representa um inteiro (25). □

Questão 2 (5.1)

Qual é a representação do número 1/10 no formato IEEE single para cada um dos quatro modos de arredondamento?

Resposta. A resposta curta é:

$$x = 0.0\overline{0011},$$

$$\text{round\_down}(x) = \text{round\_towards\_zero}(x) = 1.10011001100110011001100 \times 2^{-4} \text{ e}$$

$$\text{round\_up}(x) = \text{round\_to\_nearest}(x) = 1.10011001100110011001101 \times 2^{-4}.$$

Se x é uma representação binária exata de $1/10$, $x = 0.0\overline{0011} = 1.\overline{1001} \times 2^{-4}$ (os números abaixo da barra representam uma dízima periódica, se repetem infinitamente).
Calculamos então $x_- = 1.10011001100110011001100 \times 2^{-4}$ e $x_+ = 1.10011001100110011001101 \times 2^{-4}$. Se o modo é *Round Down*, o número será representado por x_- e se for *Round Up*, será x_+, ambos pela definição dos modos.
Já que $x > 0$, o modo *Round towards zero* também usará x_-, porém x é mais próximo de x_+ do que de x_-, basta perceber que o erro relativo entre x e x_- é maior que 1/2, portanto, o modo *Round to nearest* usará a representação x_+. □

E para os números $1 + 2^{-25}$

Questão 2 (5.1) continued on next page. . .

Resposta.

$$x = 1.0000000000000000000000001 \times 2^0,$$
$$\text{round\_down}(x) = \text{round\_towards\_zero}(x) = \text{round\_to\_nearest}(x) = 1.00000000000000000000000 \times 2^0,$$
$$\text{round\_up}(x) = 1.00000000000000000000001 \times 2^0.$$

Se x é uma representação binária exata de $1 + 2^{-25}$, $x = 1.0000000000000000000000001 \times 2^0$, $x_- = 1.00000000000000000000000 \times 2^0$ e $x_+ = 1.00000000000000000000001 \times 2^0$. O modo *Round down* vai levar para x_- e o modo *Round up* vai levar para x_+, como sempre.
Já que $x > 0$, o modo *Round towards zero* também usará x_-, além disso, o erro relativo entre x e x_- é $1/4$, logo, o modo *Round to nearest* também levará para x_-. □

e 2^{130}?

Resposta.

$$x = 1 \times 2^{130},$$
$$\text{round\_down}(x) = \text{round\_towards\_zero}(x) = \text{round\_to\_nearest}(x) =$$
$$= \text{round\_up}(x) = 1.00000000000000000000000 \times 2^{130}.$$

Se x é uma representação binária exata de 2^{130}, $x = 1 \times 2^{130}$, porém, x é exatamente representável no sistema IEEE single, logo, em todos os modos de arredondamento, será representado como $1.00000000000000000000000 \times 2^{130}$. □

Questão 3 (6.4)

Qual é o maior número de ponto flutuante x tal que $1 \oplus x$ é exatamente 1, assumindo que o formato usado é IEEE single e modo de arredondamento para o mais próximo?

Resposta. $x = 2^{-24}$. $1 + x$ é igualmente próximo de $1 + 2^{-23}$ e 1, porém, por causa do critério de arredondamento em empate (0 menos significativo), ele é arredondado para 1, qualquer x maior do que esse causará um arredondamento para um número maior do que 1. □

E se o formato for IEEE double?

Resposta. $x = 2^{-53}$, seguindo a mesma lógica usada para concluir a resposta do item anterior. □

Questão 4 (6.8)

Em aritmética exata, a soma é um operador comutativo e associativo. O operador de soma de ponto flutuante é comutativo?

Resposta. Sim, pois para calcular o resultado em soma de ponto flutuante o padrão exige que seja calculado o valor exato e, então, arredondado para o sistema escolhido. Formalmente, denotaremos por $fl(x)$ a representação em ponto flutuante de um real x e por $\oplus$ a operação de soma em ponto flutuante. Temos $fl(x) \oplus fl(y) = fl(fl(x) + fl(y)) = fl(fl(y) + fl(x)) = fl(y) \oplus fl(x)$. □

E associativo?

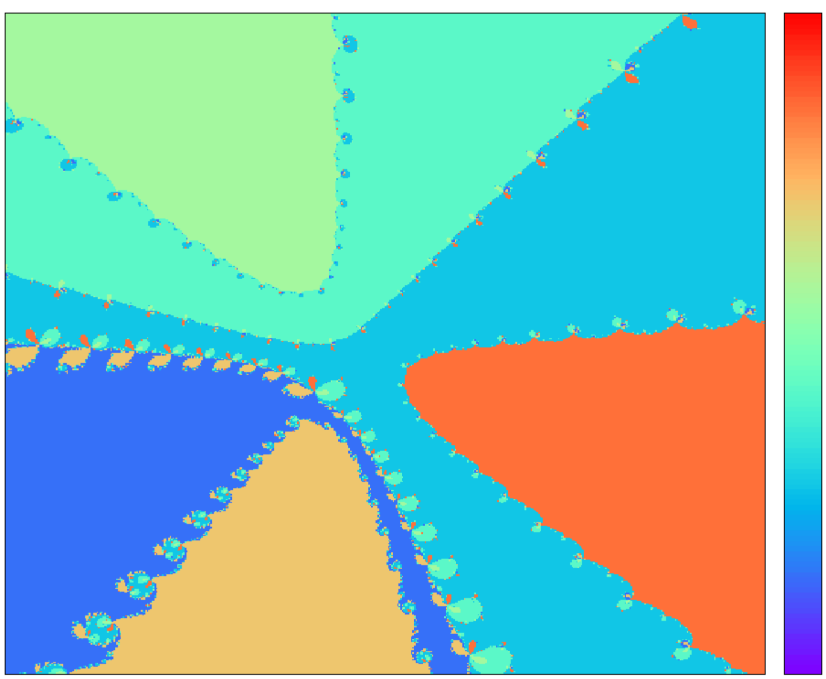
Resposta. Não, os erros de arredondamento podem fazer com que a ordem das somas faça diferença. Por exemplo, considere um sistema com um dígito binário de precisão $(1.b_1)$ e expoentes entre -4 e 4, por exemplo. Escolha os números $x = 1 = 2^0$ e $y = z = 1/4 = 2^{-2}$. Teremos, na notação do sistema (base binária) que $(x \oplus y) \oplus z = (1.0 \times 2^0 \oplus 1.0 \times 2^{-2}) \oplus 1.0 \times 2^{-2} = 1.0 \times 2^0 \oplus 1.0 \times 2^{-2} = 1.0 \times 2^0$, por outro lado, $x \oplus (y \oplus z) = 1.0 \times 2^0 \oplus 1.0 \times 2^{-1} = 1.1 \times 2^0$. □

Parte 2: Bacias de Newton

Para se obter as bacias é necessário aplicar o método de Newton nos n^2 pontos do grid criado no plano complexo. Isso ocorre devido à restrição das funções que deveriam ser implementadas descritas no enunciado. Tivesse as iteraçoes do método acesso aos pontos já calculados, poderia se aproveitar desta informação e ter uma implementação muito mais eficiente.
Assim, o que buscamos foi otimizar as iterações do método de Newton. Dado que os pontos iniciais e a função são quem ditam a taxa de convergência do método, e, além disto, estão fixados pela instância do problema, nossos esforços para melhorar a performance se tornaram mais restritos ainda e consistem em usufruir das vantagens da vetorização em MATLAB.
Felizmente, tanto a avaliação do polinõmio em um ponto quanto a derivação dele são operadores lineares. Isso nos permitiu expressar ambos de forma sucienta. A avaliação se reduz a um produto interno com um vetor cuja i-ésima coordenada é x^{n-1-i} e a derivação é o resultado de aplicar uma matriz diagonal no vetor de coeficientes, que em MATLAB pode facilmente ser implementado através de um produto componente a componente entre os vetores.
Outro fator que impactou bstante a eficiência do código foi o tratamento de entrada e saída. O comportamento padrão da função FPRINTF é de limpar o buffer de saída a cada chamada da função. Isso causou um grande *overhead* no programa, pois eram adicionados poucos caracteres a cada chamada desta função, o que faz a chamada acontecer várias vezes para que toda a saída seja impressa. Assim, houve notória melhoria ao mudar a flag passada à função FOPEN para que o buffer de saída fosse descarregado apenas ao final do programa.

Exemplos

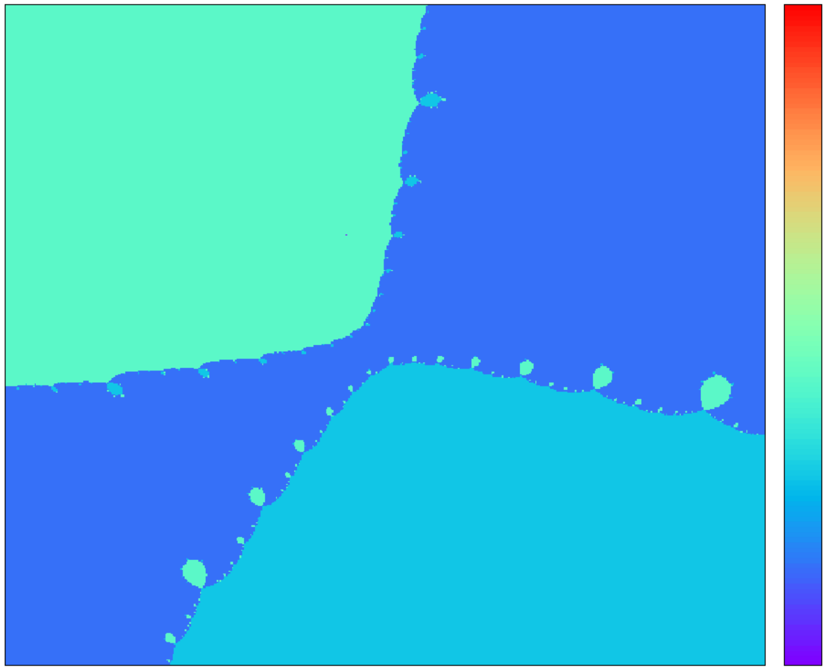
Para ilustrar o uso do método desenvolvido na parte 3 do EP. Escolhemos gerar as *Newton Basins* para um polinômio e as expansões de Taylor deste polinômio em torno do ponto 0. Para isso, escolhemos a função $f(x) = x^6 + (3-7i)x^5 + (-20-15i)x^4 + (-40+20i)x^3 + (-26+40i)x^2 + (52+52i)x$ e rodamos nosso EP nela.



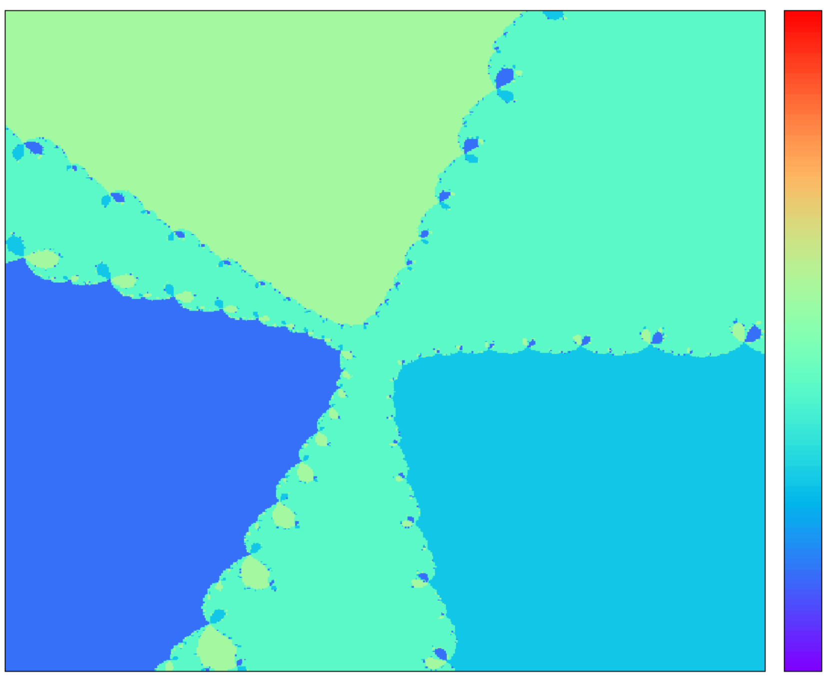
E, como planejado, repetimos o processo para as expansões de Taylor de diversos graus deste polinômio. Denotaremos a i-ésima expansão por $f_i(x)$. Primeiro, expandimos em grau 1, ou seja $f_1(x) = (52 + 52i)x$.

Seguido de $f_2(x) = (-26 + 40i)x^2 + (52 + 52i)x$.

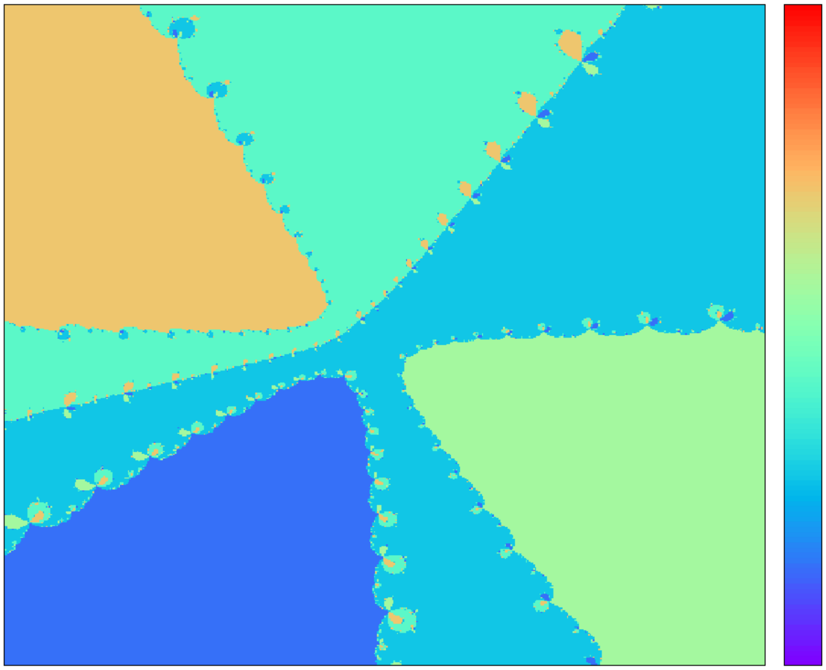
Depois $f_3(x) = (-40 + 20i)x^3 + (-26 + 40i)x^2 + (52 + 52i)x$.



Depois $f_4(x) = (-20 - 15i)x^4 + (-40 + 20i)x^3 + (-26 + 40i)x^2 + (52 + 52i)x$.



E, finalmente, $f_5(x) = (3 - 7i)x^5 + (-20 - 15i)x^4 + (-40 + 20i)x^3 + (-26 + 40i)x^2 + (52 + 52i)x$.



Parte 3: Encontrando todas as raízes de funções

Novamente aqui há poucas decisões de projeto a serem feitas. O enunciado já descreve o algoritmo e os critérios para as decisões, restando pouco a ser feito que não implementar.
Assumimos que as funções anônimas eram vetorizadas, para que pudessem ser facilmente aplicadas em diversos pontos. Além disso, utilizamos o método descrito apenas nos intervalos nos quais nenhuma das bordas era uma raíz da função. Isso descarta possíveis pontos, mas interpretamos como uma limitação do método em si, por ser sensível à escolha do *ninter*.
O que é uma solução simples para tratar o caso em que raízes da função estão entre os pontos da primeira amostragem é retornar a união das raízes encontradas ao se chamar com *ninter* e *ninter* + 1.
Isto resolve pois é impossível que um valor de x seja borda de intervalos em ambos os casos. Se fosse, existiriam inteirom $0 \leq k \leq n$ e $0 \leq t \leq n+1$ tais que

$$a + k * (b-a)/n = a + t(b-a)/(n+1), \text{ ou seja,}$$

$$(n+1)k = nt$$

Mas este inteiro $(n+1)k$, por ser múltiplo de n, é um múltiplo de $\text{mmc}(n+1, n) = n(n+1)$. Mas isso implica que $k \geq n$, o que nos permite afirmar que este ponto em comum é o próprio b. Assim, tratamos todos os possíveis pontos que são raíz no interior do intervalo, sem piora na complexidade.
O método se mostrou robusto, encontrando todas as raízes de ambas verificações dadas no enunciado. Para testar, pode-se abrir o *octave* na pasta 03 e rodar

```
find_all_roots(@(t)(sinc(t./pi),
@(t)(t.*cos(t)-sin(t))./(t.*t)),
-10, 10, 10**(-5), 100)
```

.