EP3: COMPORTAMENTO DINÂMICO DO MÉTODO DE NEWTON

1. Método de Newton

O método de Newton é um método numérico, muito simples, para calcular aproximações de raízes de uma função real da qual podemos calcular a primeira derivada.

Suponha que temos uma função diferenciável $f\colon \mathbb{R} \to \mathbb{R}$. Dado um valor inicial x_0 (idealmente, um chute para uma raiz de f), podemos calcular aproximações sucessivas para uma raiz de f gerando uma seqüência de valores

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad n \geq 0.$$

Sob certas condições muitas vezes satisfeitas, o método *converge*, isto é, temos que o limite

$$\lim_{n\to\infty} x_n$$

existe e é uma raiz de f.

Não é difícil entender a idéia por trás do método. Começamos uma iteração com um chute x para uma raiz, então calculamos a tangente da função no ponto x (usando a derivada) e seguimos pela tangente na direção do zero (veja Figura 1).

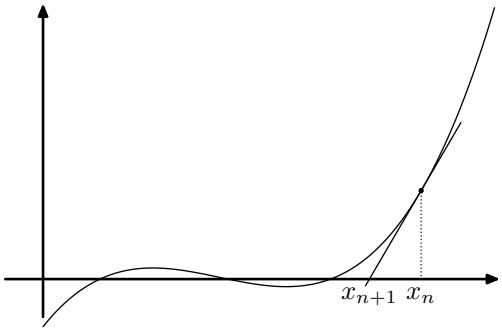

Figura 1. Uma iteração do método de Newton.

2. Bacias de atração

Funções de uma variável complexa também têm derivadas e as regras de derivação que aprendemos em cálculo também se aplicam a elas. Por exemplo, se $f(x) = (2+i)x^2 - ix + 4i$, então $f'(x) = (4+2i)x - i$; se $f(x) = \sin((1-2i)x)$, então $f'(x) = (1-2i)\cos((1-2i)x)$. O método de Newton, exatamente como descrito acima, também pode ser usado para calcular raízes de uma função complexa, bastando para tanto que possamos calcular a primeira derivada da função.

Um fenômeno interessante ocorre quando usamos o método de Newton para encontrar uma raiz de uma função complexa $f\colon \mathbb{C} \to \mathbb{C}$. Para cada ponto inicial x para o qual o método converge, obtemos uma raiz de f. O conjunto de pontos iniciais que são levados pelo método de Newton a uma dada raiz é chamado de *bacia de atração*. Podemos ter uma idéia das bacias de atração do método de

Newton fazendo um desenho de um pedaço do plano complexo no qual pintamos cada bacia de atração com uma cor diferente.

Por exemplo, o polinômio $x^5 - 1$ tem cinco raízes complexas. Pintando os pontos do quadrado $[-5, 5]^2 \subseteq \mathbb{C}$ de acordo com as bacias de atração, obtemos a Figura 2.

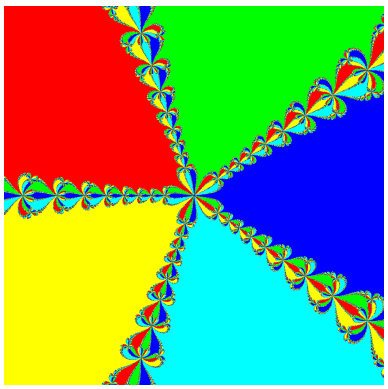

FIGURA 2. Bacias de atração para $f(x) = x^5 - 1$ dentro do quadrado $[-5, 5]^2$.

Nosso objetivo neste EP é gerar imagens como a da Figura 2. Sua tarefa será dividida em partes, cada uma correspondendo à criação de uma função. Como de costume, você deve usar o esqueleto de programa disponível na Graúna.

3. Implementação do método de Newton

Primeiramente você deve implementar a função de protótipo

```
newton(f, fp, x, eps, maxiter),
```

em que

(1) `f` e `fp` são uma função e sua derivada, respectivamente;
(2) `x` é o ponto inicial, um número complexo;
(3) `eps` é um real positivo, utilizado como critério de parada;
(4) `maxiter` é o número máximo de iterações do método de Newton que devemos executar.

Você deve ter algumas dúvidas no momento. Como podemos passar uma função como argumento para outra função? Como usar números complexos? Essas perguntas serão respondidas; antes porém convém descrever em detalhes o funcionamento da função `newton`.

Cada iteração do método de Newton deve transcorrer da seguinte forma:

(1) No início da iteração, temos um valor x que esperamos estar próximo de uma raiz. Se $|f(x)| <$ `eps`, então x está próximo o bastante de uma raiz: a função `newton` devolve o par (k, x), em que k é o número de iterações que foram concluídas até o momento.
(2) Caso x não esteja próximo de uma raiz, então calculamos a derivada $f'(x)$. Se $|f'(x)| <$ `eps`, então não podemos prosseguir, ou faríamos uma divisão por um número muito próximo de zero, arruinando a precisão dos cálculos; o método de Newton falha e a função `newton` devolve $(-1, 0)$. Caso contrário, substituímos x por $x - f(x)/f'(x)$ e passamos para a próxima iteração.

Se `maxiter` iterações forem executadas sem que uma boa aproximação seja encontrada, então o método de Newton também falha e a função `newton` devolve $(-1, 0)$.

Você deve implementar o método de Newton exatamente como descrito acima. A única complicação que pode ocorrer é que pode não ser possível calcular $f(x)$

ou $f'(x)$ durante alguma iteração; por exemplo, $\log 0$ não está definido. Nesses casos, as funções `f` e `fp` devolvem o valor especial `None`. Caso em alguma iteração uma dessas funções devolver `None`, a função `newton` deve devolver $(-1, 0)$ para indicar que falhou. Para testar se um valor é `None`, você deve usar o operador `is`, como no exemplo:

```
val = f(x)
if val is None: return (-1, 0)
# val não é None, deve ser o valor de f em x.
```

3.1. **Números complexos.** Python disponibiliza o tipo `complex`, usado para representar números complexos:

```
>> a = complex(1, 2)
>> b = complex(0, 1)
>> print(a)
(1+2j)
>> print(b)
1j
>> b ** 2
(-1+0j)
>> abs(a + b)
3.1622776601683795
```

Note que você pode operar livremente com números complexos, da mesma forma que opera com números reais. A função `abs` devolve o módulo de um número complexo.

3.2. **Funções como argumentos.** Em Python, funções são objetos assim como listas, números ou strings. Em particular, elas podem ser passadas como argumentos para outras funções. Considere o seguinte exemplo simples:

```
def soma2(x):
    return x + 2

def aplica(f, L):
    """Aplica f a cada elemento de L e devolve lista resultante."""

    ret = []
    for x in L:
        ret.append(f(x))
    return ret

def main():
    L = aplica(soma2, [ 1, 2, 3, 4 ])
    print(L)
```

Python inclusive fornece uma implementação própria da função `aplica` definida acima, chamada `map`.

A função `newton` recebe duas funções como argumentos. Você pode testar sua implementação da seguinte forma:

```
def p(x): return x ** 5 - 1
def q(x): return 5 * x ** 4

def main():
    print(newton(p, q, complex(-1, 2), 1e-8, 100))
```

Acima, `1e-8` é uma forma de representar o número 10^{-8}. A saída do seu programa deve ser algo como

```
(12, (-0.8090169943832873+0.5877852522889782j)).
```

4. Geração dos dados para a imagem

Vejamos agora como usar a função `newton` de modo a gerar os dados para fazer a imagem das bacias de atração. Dado um retângulo no plano complexo, consideramos uma grade de pontos uniformemente espaçados dentro do retângulo e os usamos como pontos iniciais para o método de Newton. Cada ponto será levado a uma raiz diferente da função; mais adiante veremos como identificar quais pontos foram levados a uma mesma raiz para pintá-los com uma mesma cor.

Você deve escrever uma função de protótipo

```
faz_matriz(f, fp, x1, y1, x2, y2, N, eps, maxiter),
```

em que

(1) `f` e `fp` são uma função e sua derivada, respectivamente;
(2) `x1`, `y1` e `x2`, `y2` são as coordenadas do cantos esquerdo inferior e direito superior de um retângulo no plano complexo;
(3) `N` é o número de pontos da grade no lado menor do retângulo;
(4) `eps` e `maxiter` são parâmetros para a função `newton`.

Seja $\delta = \min\{\texttt{y2} - \texttt{y1}, \texttt{x2} - \texttt{x1}\}/\texttt{N}$. Escreva[1] $m = 1 + \lceil(\texttt{y2} - \texttt{y1})/\delta\rceil$ e $n = 1 + \lceil(\texttt{x2} - \texttt{x1})/\delta\rceil$. Sua função deve devolver matrizes I e R, ambas $m \times n$, tais que para $0 \le i < m$ e $0 \le j < n$, se

$$(k, x) = \texttt{newton(f, fp, complex(x1} + j * \delta\texttt{, y2} - i * \delta\texttt{), eps, maxiter)},$$

então

$$I[i][j] = k \qquad \text{e} \qquad R[i][j] = x$$

(veja Figura 3). Assim, a matriz I contém o número de iterações executadas pelo método de Newton e a matriz R as aproximações de raízes para cada ponto da grade. A assimetria acima entre a parte real e a parte complexa do ponto inicial deve-se à forma como imagens são representadas no computador. Quando desenhamos o gráfico de uma função no papel, os valores do eixo vertical aumentam de baixo para cima. Numa imagem de computador, o eixo vertical aumenta de cima para baixo (o pixel no topo esquerdo superior tem coordenadas $(0, 0)$, o pixel logo abaixo tem coordenadas $(0, 1)$ e assim por diante).

5. Atribuição de cores

Uma vez que temos a matriz calculada pela função `faz_matriz`, precisamos atribuir a cada raiz uma cor diferente. O problema é que o método de Newton é um método numérico. Nós executamos algumas iterações, até ficar contentes com a aproximação obtida. Além disso, usamos aritmética de ponto flutuante, que é em si fonte de imprecisões. Por isso, se `x` e `y` são duas aproximações de raízes devolvidas pela função `newton`, faz pouco sentido fazer a comparação `x == y`; o que devemos verificar é se as duas aproximações estão próximas o bastante uma da outra. Em outras palavras, nosso problema é: para cada entrada (i, j) tal que $I[i][j] \neq -1$ temos em $R[i][j]$ uma raiz aproximada; queremos decidir quais raízes devemos considerar como sendo iguais e quais não.

[1] Aqui, $\lceil x\rceil$ é *teto* de x, o menor inteiro maior ou igual ao número real x. Assim, $\lceil 4\rceil = 4$ e $\lceil 4.5\rceil = 5$. Para calcular o teto de um número em Python, use a função `ceil`.

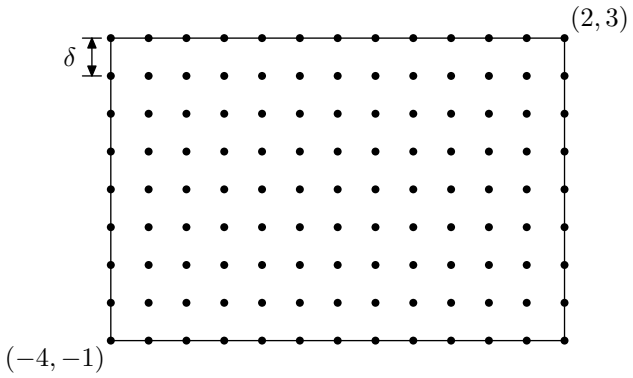

FIGURA 3. Escolha de grade para um dado retângulo no plano complexo, com `x1` $= -4$, `y1` $= -1$, `x2` $= 2$ e `y2` $= 3$. Se `N` $= 8$, então temos $\delta = 0.5$, $m = 9$ e $n = 13$.

Este é um tipo de problema comum em computação, chamado de problema de *aglomeração* (em inglês, *clustering*): temos diversos pontos no espaço (que representam dados, como obras literárias, escolhas de clientes numa loja virtual, seqüências de DNA etc.) e queremos aglomerar esses pontos em conjuntos de acordo com alguma medida de similaridade.

Há diversas abordagens para resolver problemas de aglomeração. Em nosso programa vamos usar uma abordagem baseada em centros. Fixe um número real $\rho \geq 0$ e seja S um conjunto finito de números complexos (no nosso caso, as aproximações de raízes). Nosso trabalho será encontrar números $x_1, \ldots, x_n \in S$, chamados de *centros*, tais que

(1) $|x_i - x_j| > \rho$ para todo $i \neq j$ e
(2) para todo $y \in S$ existe i tal que $|x_i - y| \leq \rho$.

Assim, podemos associar cada ponto de S a um dos centros, quebrando empates de forma arbitrária; os pontos associados a um mesmo centro recebem uma mesma cor.

Você deve escrever uma função

```
determina_cores(I, R, ro)
```

que recebe as matrizes I e R devolvidas pela função `faz_matriz` e o parâmetro `ro` $= \rho$ usado na descrição acima. Sua função deve encontrar centros como acabamos de descrever e associar cada entrada (i, j) para a qual $I[i][j] \neq -1$ a um dos centros. Entradas associadas a um mesmo centro devem receber uma mesma cor.

Uma cor é uma tripla (r, g, b) de valores inteiros no intervalo $[0, 255]$. Os valores correspondem às intensidades das cores azul, verde e vermelha, respectivamente; quanto maior o valor, maior a intensidade. Por exemplo, a tripla $(255, 0, 0)$ representa a cor azul; as triplas $(0, 0, 0)$ e $(255, 255, 255)$ representam preto e branco, respectivamente.

A função `determina_cores` deve devolver uma matriz C do mesmo tamanho de R. Se (i, j) é tal que $I[i][j] \neq -1$, então $C[i][j]$ deve ser a cor associada à raiz $R[i][j]$. Para obter cores, você deve usar a função `nova_cor`, que se encontra no esqueleto do EP. Cada chamada à função devolve uma tripla que representa uma cor.

Note que há diversas maneiras de encontrar os centros, você deve pensar na sua própria maneira de fazê-lo.

6. Sombreamento

Na imagem da Figura 2, cada bacia de atração foi pintada com uma cor uniforme. Podemos fazer uma figura mais interessante pintando cada ponto com uma cor tanto mais escura quanto maior for o número de iterações do método de Newton quando executado com aquele ponto como ponto inicial (veja Figura 4).

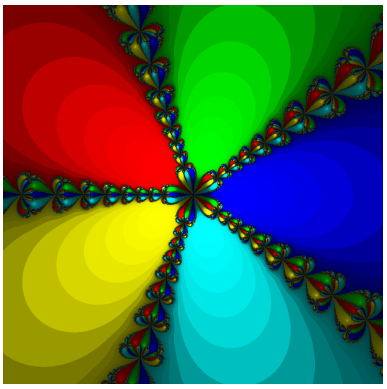

Figura 4. Bacias de atração para $f(x) = x^5 - 1$ dentro do quadrado $[-5, 5]^2$; quanto maior o número de iterações executadas pelo método de Newton num dado ponto, mais escura a cor a ele associada.

Seja I a matriz de iterações devolvida pela função `faz_matriz` e seja M o maior número que ocorre em I. Para $k = 0, \ldots, M$, denote por $a[k]$ o número de entradas (i, j) tais que $I[i][j] = k$. Escreva

$$S = \sum_{k=0}^{M} a[k]$$

e para $k = 0, \ldots, M + 1$ escreva $s[k] = 1 - (a[0] + \cdots + a[k-1])/S$.

Fixe um número real $\alpha \in [0, 1]$. O peso da cor associada a uma entrada (i, j) com $k = I[i][j] \neq -1$ deve ser

$$1 - \alpha + \alpha s[k].$$

Desta forma, quanto maior k, menor será o peso da cor. Se a cor $c = (r, g, b)$ tem peso p, então a cor usada na imagem final deve ser

$$(\lfloor pr \rfloor, \lfloor pg \rfloor, \lfloor pb \rfloor).$$

Assim, quanto maior o peso, mais clara será a cor; quanto menor o peso, mais escura ela será.

O parâmetro α pode ser usado para controlar o quanto os pesos dependem do número de iterações. Se $\alpha = 0$, então o peso será sempre 1, independente do número de iterações: obtemos assim uma figura na qual cada bacia de atração recebe uma cor uniforme, como a Figura 2. Se $\alpha = 1$, os pesos dependem totalmente do número de iterações. Para gerar a Figura 4 usou-se $\alpha = 0.8$.

Você deve escrever uma função de protótipo

`calcula_sombras(I)`

que recebe a matriz de iterações I devolvida pela função `faz_matriz` e devolve a lista s definida acima. Sua função não deve alterar a matriz.

7. Gravação da imagem

Para gravar a imagem de saída usaremos o formato PPM[2], que permite que imagens sejam armazenadas em arquivos de texto. Eis um exemplo de arquivo PPM:

```
P3
3 2
255
255 0 0 0 255 0 0 0 255
38 20 255 0 0 0 255 255 255
```

A primeira linha do arquivo é, por convenção do formato, sempre igual a '`P3`'. A segunda linha traz o número de colunas (largura) e o número de linhas (altura) da imagem, em pixels; a imagem acima, por exemplo, tem largura 3 e altura 2. A terceira linha traz o valor que representa a maior intensidade de cor; no nosso caso, este valor será sempre 255.

Cada uma das linhas seguintes representa uma linha da imagem. Cada pixel é representado por três valores consecutivos, um para cada cor fundamental (vermelho, verde e azul). Por exemplo, a primeira linha da imagem tem 3 pixels, o primeiro vermelho, o segundo verde e o terceiro azul; o último pixel da segunda linha é branco.

Você deve escrever uma função de protótipo

`grava_imagem(I, C, s, alfa, nome_arquivo)`

que recebe a matriz de iterações I devolvida pela função `faz_matriz`, a matriz de cores C devolvida pela função `determina_cores`, a lista s devolvida pela função `calcula_sombras`, um número real `alfa` em $[0, 1]$ e o nome de um arquivo de saída. Sua função deve gravar a imagem no arquivo de saída, usando as cores em C e os pesos em s, levando também em consideração o parâmetro $\alpha =$ `alfa` como explicado na seção anterior. Se uma entrada (i, j) é tal que $I[i][j] = -1$, então o pixel correspondente na imagem de saída deve ser preto.

8. Programa principal

A função `main` já se encontra escrita no esqueleto do EP; você não deve alterá-la. Ela chama as demais funções que você deve implementar para gerar uma imagem.

Para usar o programa final, você deve criar um arquivo de texto contendo a função a ser usada e sua primeira derivada, cada uma em uma linha. A variável da função deve ser `x`. Você pode definir a função escrevendo sua expressão em Python normalmente; você também pode usar as seguintes funções do Python em sua definição:

`sin`, `cos`, `sinh`, `cosh`, `tan`, `exp`, `log`.

Por exemplo, para gerar a Figura 4, você pode criar um arquivo chamado, digamos, '`p5.txt`', com o conteúdo:

```
x ** 5 - 1
5 * x ** 4
```

Basta então executar o programa, informando os parâmetros desejados:

[2]Procure na Wikipedia por "netpbm format". Vamos colocar na Graúna alguns links para visualizadores de imagens PPM. Note entretanto que o visualizador padrão de uma distribuição Linux é capaz de abrir imagens no formato PPM.

```
> python ep3.py
EP3 - Comportamento dinâmico do Método de Newton

Arquivo com as funções: p5.txt
Arquivo de saída:       p5.ppm

Parâmetros da grade
x1: -5
y1: -5
x2: 5
y2: 5
N : 1000

Parâmetros para o Método de Newton
eps:     1e-8
maxiter: 100

Parâmetros para coloração
ro:   1e-5
alfa: 0.8

Calculando imagem... pronto.
```

Considere também o arquivo de entrada '`trig.txt`':

```
sin(complex(1, 1) * x) + cos(complex(0, 2) * x)
complex(1, 1) * cos(complex(1, 1) * x) - complex(0, 2) * sin(complex(0, 2) * x)
```

Usando esse arquivo e os parâmetros abaixo, obtemos a Figura 5.

```
> python ep3.py
EP3 - Comportamento dinâmico do Método de Newton

Arquivo com as funções: trig.txt
Arquivo de saída:       trig.ppm

Parâmetros da grade
x1: -3
y1: -2
x2: 3
y2: 2
N : 400

Parâmetros para o Método de Newton
eps:     1e-8
maxiter: 100

Parâmetros para coloração
ro:   1e-5
alfa: 0.8

Calculando imagem... pronto.
```

Para obter figuras interessantes, muitas vezes é preciso realçar uma pequena região do domínio, como no exemplo do arquivo '`pretty.txt`':

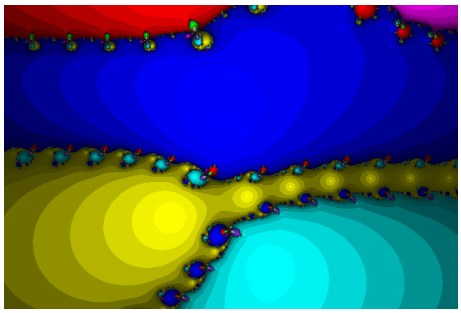

FIGURA 5. Imagem obtida do arquivo '`trig.txt`'.

```
x ** 8 + 3 * x ** 4 - 4
8 * x ** 7 + 12 * x ** 2
```

Usando esse arquivo com os parâmetros abaixo, obtemos a Figura 6.

```
> python ep3.py
EP3 - Comportamento dinâmico do Método de Newton
Arquivo com as funções: pretty.txt
Arquivo de saída:       pretty.ppm

Parâmetros da grade
x1: 0.09
y1: 0.08
x2: 0.1
y2: 0.091
N : 600

Parâmetros para o Método de Newton
eps:     1e-5
maxiter: 100

Parâmetros para coloração
ro:   1e-5
alfa: 0.8
```

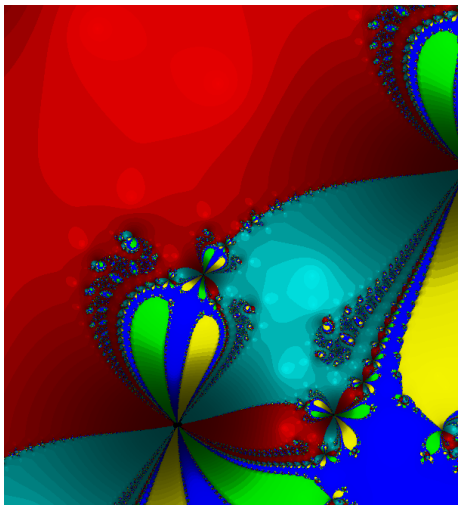

FIGURA 6. Imagem obtida do arquivo '`pretty.txt`'.