Enumeração exaustiva e outras formas de aproximação

MC102-2018s1-Aula09-180327-takeaway

Arthur J. Catto, PhD

27 de março de 2018

1 Enumeração exaustiva e outras formas de aproximação

Computadores são ótimos para tarefas que exigem grande número de repetições.

Segundo a Harvard Database of Useful Biological Numbers, uma pessoa rápida consegue piscar umas 5 vezes por segundo. Nesse mesmo tempo um processador moderno é capaz de executar pelo menos algumas centenas de milhões de instruções...

Portanto, vamos botar essas máquinas pra trabalhar...

1.1 O método da enumeração exaustiva

Chama-se *enumeração exaustiva* um método de solução de problemas em que nos aproximamos progressivamente de uma solução desejada examinando todos os possíveis candidatos (ou pelo menos uma parte considerável deles).

1.1.1 Exemplo: *Calcular a raiz cúbica de um cubo inteiro perfeito*

Como o problema só fala de inteiros, este problema pode ser resolvido por força bruta, examinando exaustivamente todos os possíveis candidatos.

O conjunto de candidatos pode ser limitado aos inteiros não negativos, se lembrarmos que quando $x < 0$, $\sqrt[3]{x} = (-1) \cdot \sqrt[3]{|x|}$.

Então, vamos lá...

```
In [ ]: # Ler o número cuja raiz cúbica se deseja encontrar
        x = int(input('Digite um número inteiro: '))
```

```
In [ ]: # Encontrar a raiz cúbica de absx por enumeração exaustiva
        raiz =
        while :
            raiz
```

```
In [ ]: # Exibir o resultado
        if :
            print(x, 'não é um cubo perfeito!')
        else:

            print('A raiz cúbica de', x, 'é', raiz)
```

Solução

```
In [ ]: # Ler o número cuja raiz cúbica se deseja encontrar
        x = int(input('Digite um número inteiro: '))
        absx = abs(x)
```

```
In [ ]: # Encontrar a raiz cúbica de absx por enumeração exaustiva
        raiz = 0
        while raiz ** 3 < absx:
            raiz += 1
```

```
In [ ]: # Exibir o resultado
        if raiz ** 3 != absx:
            print(x, 'não é um cubo perfeito!')
        else:
            if x < 0:
                raiz = -raiz
            print('A raiz cúbica de', x, 'é', raiz)
```

```
In [ ]: 123456789**3
```

Seria possível resolver esse problema usando `for` em vez de `while`?
Poderíamos começar com...

```
In [ ]: # Encontrar a raiz cúbica de absx por enumeração exaustiva
        for raiz in range(...):
            ...
```

Qual seria o argumento de `range`?

- `range` precisa conter a resposta.
- Não temos um palpite para isso, mas sabemos que a raiz cúbica de um inteiro não negativo é sempre menor ou igual a ele.
- Portanto, usar $absx + 1$ como *stop* em `range` parece razoável.

Assim, ficamos com...

```
In [ ]: # Encontrar a raiz cúbica de absx por enumeração exaustiva
        for raiz in range(absx + 1):
            ...
```

Agora, o que seria uma *suite* aceitável?

- Do jeito em que está, o `for` sempre percorrerá a `range` toda e assim, quase sempre, passará reto por cima da resposta desejada...

Como evitar isso?

- É preciso dar um jeito de interromper o `for` caso a resposta seja encontrada ou tenhamos certeza de que o problema não tem solução.

Como fazer isso?

- Vamos ter que usar um comando condicional...

```
In [ ]: # Encontrar a raiz cúbica de absx por enumeração exaustiva
        for raiz in range(absx + 1):
            if raiz ** 3 >= x:
                # parar o `for`
```

Para interromper a execução de um `for`, antes de seu término normal, usamos um comando `break`.

```
In [ ]: # Encontrar a raiz cúbica de absx por enumeração exaustiva
        for raiz in range(absx + 1):
            if raiz ** 3 >= x:
                break
```

E o problema está resolvido...

Vale lembrar que um `break` interrompe apenas a execução do `for` dentro do qual ele "realmente" está.

```
In [ ]: for i in range(2):
            print('começando i =', i)
            for j in range(2):
                print('   começando j =', j)
                for k in range(2):
                    print('      começando k =', k)
                    print('     break')
                    break
                    print('      terminando k =', k)
                print('   terminando j =', j)
            print('terminando i =', i)
        print('fim do programa')
```

1.1.2 Exemplo: *Calcular a raiz quadrada de um número real não-negativo*

Neste caso, saímos do campo dos inteiros...

Podemos adaptar o exemplo anterior usando um `for`? Por que?

- Não, porque `range` não trabalha com *floats*.

Mas é possível usar o algoritmo anterior com `while`.

Podemos continuar incrementando *raiz* de 1 em 1?

- Provavelmente não... vamos querer uma aproximação melhor que essa...
- Vamos supor que 10^{-6} seja um passo aceitável e vamos atribuir o nome *step* a ele.

As células abaixo contêm nosso exemplo da raiz cúbica. Com algumas alterações rápidas, nosso programa começará a tomar forma.

Mãos à obra...

In [ ]:
```
# Ler o número cuja raiz cúbica se deseja encontrar
x = int(input('Digite um número inteiro: '))
absx = abs(x)
```

In [ ]:
```
# Encontrar a raiz cúbica de absx por enumeração exaustiva
raiz = 0
while raiz ** 3 < absx:
    raiz += 1
```

In [ ]:
```
# Exibir o resultado
if raiz ** 3 != absx:
    print(x, 'não é um cubo perfeito!')
else:
    if x < 0:
        raiz = -raiz
    print('A raiz cúbica de', x, 'é', raiz)
```

Solução

In [ ]:
```
# Ler o número cuja raiz quadrada se deseja encontrar
x = float(input('Digite um número não-negativo: '))
step = 1e-6
```

In [ ]:
```
# Encontrar uma aproximação para a raiz quadrada de x por enumeração exaustiva
raiz = 0
while raiz ** 2 <= x:
    raiz += step
print(raiz)
```

In [ ]:
```
# Exibir o resultado
if raiz ** 2 != x:
    print('não consegui calcular uma aproximação para a raiz quadrada de', x)
else:
    print(raiz, 'é aproximadamente igual à raiz quadrada de', x)
```

Vamos testá-lo calculando a raiz quadrada de 25.

Ops! O que será que aconteceu? Adicionando um `print(raiz)` no final do cálculo da `raiz` vemos que o valor calculado é `5.000000000344985`, que não está mal como aproximação...

Por que ele não deu essa resposta?

- Porque a condição do `if` na exibição do resultado é `raiz ** 2 != x` e o resultado dessa comparação é `False`...

Vamos examinar o que está acontecendo reunindo os trechos essenciais de nossa solução...

In [ ]:
```
x = 25
step = 1e-6
raiz = 0
while raiz ** 2 <= x:
    raiz += step
print(x, raiz, raiz ** 2, (raiz ** 2 != x))
```

Isso acontece sempre que se usam *floats* para representar *reais* e decorre de como *floats* são representados num computador (ou mesmo numa folha de papel).

Uma ligeira digressão para falar de *floats*. Numa folha de papel ou num computador, somente conseguimos representar números com uma quantidade limitada de algarismos significativos. Por maior que seja a quantidade de algarismos usados, isso representará quase nada do espaço infinito dos *reais*.

Para conseguir expandir o intervalo que seria coberto pelos *floats*, sacrificamos a precisão e adotamos a *notação exponencial*. Vamos ver como.

Na *notação exponencial*, um número é representado como um inteiro com um certo número de algarismos significativos multiplicado por 10 elevado a um certo expoente. Quando estamos fazendo cálculos à mão, o número de algarismos significativos usados dependerá de nossa escolha. Para facilitar a leitura, suponha que sejam 6. Por exemplo,...

$$\begin{aligned} 1.25 &= 125000 \cdot 10^{-5} \\ 3.1416 &= 314160 \cdot 10^{-5} \\ 123.456 &= 123456 \cdot 10^{-3} \end{aligned}$$

Com 6 algarismos significativos e em notação decimal, o maior número que se consegue representar é 999999. Em notação exponencial, graças ao fator 10^e, podemos ir muito além. O único limite será imposto pela faixa aceitável para o expoente e. Por exemplo,...

$$\begin{aligned} 999999 &= 999999 \cdot 10^{0} \\ 999999000 &= 999999 \cdot 10^{3} \\ 999999000000 &= 999999 \cdot 10^{6} \end{aligned}$$

Suponha agora que $a = 999999000000$ e $b = 999999999999$.

Como eles seriam representados?

$$\begin{aligned} a &= 999999000000 = 999999 \cdot 10^{6} \\ b &= 999999999999 = 999999 \cdot 10^{6} \end{aligned}$$

E se agora calcularmos $b - a$?

- O resultado será $000000 \cdot 10^0$, quando deveria ser $999999 \cdot 10^0$.

Você vê a perda de precisão?

Num computador acontece exatamente a mesma coisa. Apenas, como o computador trabalha na base 2 e não na base 10, os valores críticos serão outros. Por exemplo,...

```
In [ ]: a = 2**72
        b = a + 524288
        fa = float(a)
        fb = float(b)
        print(format(a, "24d"), format(b, "28d"), format(b - a, "32d"))
        print(format(fa, "28.21e"), format(fb, "28.21e"), format(fb - fa, "28.21e"))
```

Não são apenas grandes números que sofrem com a perda de precisão.

Na notação exponencial decimal com 6 algarismos significativos $\frac{1}{3}$ é representado como $333333 \cdot 10^{-6}$.

Assim, quando somamos $\frac{1}{3} + \frac{1}{3} + \frac{1}{3}$, obtemos $999999 \cdot 10^{-6}$ que é diferente de 1 $(100000 \cdot 10^{-5})$

Como no caso dos grandes números, esse problema também ocorre num computador digital, embora com outros valores por causa da mudança da base. Por exemplo, $\frac{1}{10}$ não tem representação exata na base 2, assim como $\frac{1}{3}$ não tem na base 10...

Assim, se somarmos 0.1 dez vezes, o resultado obtido não será rigorosamente igual a 1.0...

```
In [ ]: tot = 0.0
        for i in range(10):
            tot += 0.1
        print(1.0, tot, tot == 1.0, 1 - tot)
```

A perda de precisão em cálculos relativamente simples pode ser muito significativa. Suponha que queiramos calcular

$$\frac{(x+a)^2 - 2xa - a^2}{x^2}$$

O resultado esperado dessa expressão é 1, quaisquer que sejam x e a, desde que $x \neq 0$. Vejamos o que acontece quando criamos um *script* Python para fazer esse cálculo usando vários valores de a e x...

```
In [ ]: print(7 * ' ' + 'x' + 15 * ' ' + 'a' + 16 * ' ' + 'f(x, a)' + 16 * ' ' + 'e(x, a)')
        print(14 * '-' + '  ' + 14 * '-' + '  ' + 21 * '-' + '  ' + 21 * '-')
        for p in range (6):
            a = 10 ** p
            x = 10 ** -(p + 1)
            f = ((x + a) ** 2 - 2 * x * a - a ** 2) / (x ** 2)
            e = ((a + x) ** 2 - a ** 2 - 2 * a * x) / (x ** 2)
            print(f'{x:14.8e}  {a:14.8e}  {f:21.14e}  {e:21.14e}')
```

Todos os valores nas colunas `f(x, a)` e `e(x, a)` deveriam ser "iguais" a 1.0, mas vários estão longe disso.

Esse não é um privilégio de Python, mas sim uma consequência da forma como *floats* estão implementados nos processadores usados nos nossos computadores. Toda vez que você operar com números muito díspares (neste caso a^2 é muito maior que x^2) haverá o risco de perda grave de precisão. Se você fizer o mesmo cálculo em outras plataformas (inclusive Excel) vai encontrar o mesmo resultado.

1.1.3 De volta ao nosso exemplo...

A discussão sobre *floats* deixou claro o perigo de fazermos testes de igualdade com eles. A saída é usar *testes de proximidade*. Um *teste de proximidade* compara dois valores e vê se a distância entre eles está dentro de uma certa tolerância ε (*epsilon*) pré-definida.

Vamos fazer isso no nosso algoritmo...

```
In [ ]: # Encontrar uma aproximação para a raiz quadrada de x por enumeração exaustiva
        raiz = 0
        while abs(raiz ** 2 - x) > epsilon:
            raiz += step
```

Vamos reunir os pedaços e testar o resultado... mas antes disso precisamos escolher um valor razoável para *epsilon*. O que você acha?

```
In [ ]: # Ler o número cuja raiz quadrada se deseja encontrar
        x = float(input('Digite um número não-negativo: '))
        step = 1e-6
        epsilon = 1e-4
```

```
In [ ]: # Encontrar uma aproximação para a raiz quadrada de x por enumeração exaustiva
        raiz = 0
        while abs(raiz ** 2 - x) > epsilon:
            raiz += step
```

```
In [ ]: # Exibir o resultado
        print(raiz, 'é aproximadamente igual à raiz quadrada de', x)
```

O que acontece com nosso programa se o número dado for muito grande? Por exemplo, 1524157875019052100?

Cansamos de esperar? Esse inteiro é o quadrado de 1234567890. Como partimos de zero e estamos usando um *step* de 10^{-6}, serão necessários mais de 10^{15} ciclos do `while` para chegar à resposta. Isso é muita coisa, mesmo para um computador muito rápido.

Por outro lado, se usarmos um *step* maior, nosso programa poderá deixar de funcionar para números pequenos. O que fazer?

Uma saída seria usar um *step* variável, que pudesse ir sendo refinado à medida que nos aproximássemos da solução.

1.2 O método da bisseção

O *método da bisseção* é um método de aproximação com *step* variável, muito rápido e muito usado na solução de diversos problemas reais.

Suponha que sejamos capazes de definir um intervalo $[esq..dir]$ no qual, com certeza, a resposta está contida. Vamos escolher como candidato o valor médio desse intervalo e associá-lo à variável *raiz*.

Agora, duas coisas podem acontecer:

- $|\,raiz^2 - x\,| \leq \varepsilon \rightarrow$ nosso problema está resolvido, ou,
- $|\,raiz^2 - x\,| > \varepsilon \rightarrow$ é preciso encontrar um novo candidato. Nesse caso...
  - Se $raiz^2 < x$, o candidato era pequeno demais. Portanto, a resposta deverá estar no intervalo $[raiz..dir]$ e um novo candidato deverá ser procurado dentro dele.
  - Se $raiz^2 > x$, o candidato era grande demais. Portanto, a resposta deverá estar no intervalo $[esq..raiz]$ e um novo candidato deverá ser procurado dentro dele.

Note que, a cada passo, a largura do intervalo onde a resposta certamente se encontra é reduzida pela metade, o que explica o nome do método e a velocidade com que o algoritmo se aproxima da solução.

Vamos adaptar nosso algoritmo que emprega o método de enumeração exaustiva para que ele adote o método da bisseção.

```
In [ ]: # Ler o número cuja raiz quadrada se deseja encontrar
        x = float(input('Digite um número não-negativo: '))
        epsilon = 1e-8
```

```
In [ ]: # Encontrar uma aproximação para a raiz quadrada de x por bisseção

        # Definição do intervalo de busca inicial e do primeiro candidato
        if x > 1:
            lim_esq, lim_dir = 1.0, x
        else:
            lim_esq, lim_dir = x, 1.0

        raiz = (lim_esq + lim_dir) / 2

        # refinamento progressivo da aproximação
        while abs(raiz ** 2 - x) > epsilon:
            if raiz ** 2 < x:
                lim_esq = raiz
            else:
                lim_dir = raiz
            raiz = (lim_esq + lim_dir) / 2
```

```
In [ ]: # Exibir o resultado
        print(raiz, 'é a raiz quadrada de', x, 'a menos de', epsilon)
```

Vamos testar nosso programa com os valores usados anteriormente. Será que ele consegue calcular um aproximação da raiz quadrada de 15241578750190521000?

Lembra-se de que nesse caso o método de busca exaustiva necessitaria de mais do que 10^{15} ciclos do `while` para chegar à resposta? De quantos ciclos será que o método da bisseção necessitou?

Vamos incluir um contador, só por curiosidade...

```
In [ ]: # Encontrar uma aproximação para a raiz quadrada de x por bisseção
        # Definição do intervalo de busca inicial
        if x > 1:
            lim_esq, lim_dir = 1.0, x
        else:
            lim_esq, lim_dir = x, 1.0
        # O candidato é o ponto médio do intervalo
        raiz = (lim_esq + lim_dir) / 2

        ciclos = 0
        while abs(raiz ** 2 - x) > epsilon:
            if raiz ** 2 < x:
```

```
            lim_esq = raiz
        else:
            lim_dir = raiz
        raiz = (lim_esq + lim_dir) / 2
        ciclos += 1
    print('número de ciclos necessários =', ciclos)
```

1.3 O método de Newton (ou Newton-Raphson)

Newton criou um método que é usualmente adotado para encontrar as raízes reais de muitas funções e que se aplica também ao nosso exemplo. Raphson propôs uma ideia semelhante mais ou menos ao mesmo tempo e, por isso, o método é citado com os dois nomes.

No nosso caso, dado um número real não-negativo x, queremos encontrar r tal que $r^2 = x$. Passando x para o lado esquerdo, vemos que esse problema é o mesmo que encontrar a raiz do polinômio $p(r) = r^2 - x$, isto é, um valor *raiz* tal que $p(raiz) = 0$.

Newton demonstrou que se *raiz* é uma aproximação da resposta desejada, $raiz - \dfrac{p(raiz)}{\dot{p}(raiz)}$ é uma aproximação melhor ainda.

Vamos fazer alguns pequenos ajustes no nosso algoritmo para que ele implemente esse modelo.

```
In [ ]: # Ler o número cuja raiz quadrada se deseja encontrar
        x = float(input('Digite um número não-negativo: '))
        epsilon = 1e-8
```

```
In [ ]: # Encontrar uma aproximação para a raiz quadrada de x por Newton-Raphson
        # Definição do candidato inicial
        raiz = x / 2.0

        ciclos = 0
        while abs(raiz ** 2 - x) > epsilon:
            p = raiz ** 2 - x   # este é o polinômio calculado em "raiz"
            dp = 2 * raiz       # esta é a derivada calculada em "raiz"
            raiz -= p / dp
            ciclos += 1
        print('número de ciclos necessários =', ciclos)
```

```
In [ ]: # Exibir o resultado
        print(raiz, 'é a raiz quadrada de', x, 'a menos de', epsilon)
```

Newton-Raphson precisou de menos do que a metade dos ciclos necessários para o método da *bisseção*, que já era muito mais rápido e geral do que a *enumeração exaustiva*.

1.3.1 Exercício: *Encontrar uma potência que seja igual a um inteiro dado.*

Ler um inteiro n e encontrar dois inteiros b e e, $(0 < e < 6)$, tais que $b^e = n$.

1.3.2 Exercício: *Gerar todos os números primos menores que um valor dado*

Ler um inteiro n e criar uma lista com todos os números primos menores do que n.

1.3.3 Exercício: *Ler uma sequência de números e verificar se eles estão em ordem crescente*

Ler uma sequência de números e ***depois*** verificar se eles estão em ordem crescente.

1.3.4 Exercício: *Dada uma sequência de números quaisquer, encontrar o par mais próximo*

Ler uma sequência de números S e ***depois*** encontrar $a \in S$ e $b \in S$, $a \neq b$, tais que a distância entre eles, isto é, $|a - b|$, seja mínima.

1.3.5 Exercício: *Imprimir um triângulo numérico*

Dado um inteiro positivo n, imprimir um triângulo com n linhas e o seguinte formato: `1 1 2 1 2 3 1 2 3 4 ...`

1.3.6 Exercício: *Imprimir um tapete quadrado*

Dado um inteiro positivo n, imprimir um quadrado com n linhas e colunas e o seguinte formato (supondo $n = 5$):

```
+ * * * *
* + * * *
* * + * *
* * * + *
* * * * +
```