Física Computacional

FCUP 2025/26

Exercícios selecionados adaptados e traduzidos do livro adotado: Mark Newman, 1ª edição.

Série 1: revisão de programação em python

■ Exercício 1.1 | Números de Catalan

Os números de Catalan C_n são uma sequência de inteiros $1, 1, 2, 5, 14, 42, 132, \ldots$ que surge em problemas de Mecânica Quântica e na Teoria dos Sistemas Desordenados. Estes números podem ser definidos pela regra de recorrência seguinte:

$$C_0 = 1, \qquad C_{n+1} = \frac{4n+2}{n+2} C_n.$$

Escreva um programa que calcule e mostre todos os números de Catalan tais que $C_n < 10^9$.

■ Exercício 1.2 | Cálculo recursivo

Uma característica muito útil de funções definidas pelo utilizador em `python` é a capacidade de uma função se chamar recursivamente a si mesma. Por exemplo, considere a seguinte definição do fatorial $n!$ de um inteiro positivo n:

$$n! = \begin{cases} 1 & \text{se } n = 1, \\ n \times (n-1)! & \text{se } n > 1. \end{cases}$$

Esta constitui uma definição completa do fatorial que nos permite calcular o valor de $n!$ para qualquer inteiro positivo. Podemos aplicar essa definição diretamente para criar uma função `factorial(n)` para o cálculo de fatoriais, por exemplo:

```
def factorial(n):
    if n==1:
        return 1
    else:
        return n*factorial(n-1)
```

Note como, neste exemplo, enquanto $n \neq 1$, a função chama-se a si própria para calcular o fatorial de $n-1$, e assim sucessivamente. A este procedimento chamamos *recursão*. Se executarmos `factorial(5)` o computador irá imprimir corretamente a resposta: 120.

Os números de Catalan C_n foram definidos no exercício anterior. Com apenas um pequeno rearranjo, a sua definição pode ser reescrita na forma

$$C_n = \begin{cases} 1 & \text{se } n = 0, \\ \dfrac{4n-2}{n+1} C_{n-1} & \text{se } n > 0. \end{cases}$$

Escreva uma função `python`, que use explicitamente recursão para calcular C_n. Use essa função para calcular e imprimir C_{100}.

Comentário: Comparando o cálculo dos números de Catalan feito aqui com o do exercício anterior, vemos que é possível fazer o cálculo de duas maneiras: diretamente ou usando recursão. Na maioria dos casos, se uma quantidade puder ser calculada sem recursão, o cálculo será mais rápido e normalmente recomendamos que proceda dessa forma, se possível. (No entanto, algumas aplicações são essencialmente impossíveis, ou pelo menos muito mais difíceis de implementar, sem recursividade.)

■ Exercício 1.3 | Erros de truncagem: fatorial revisitado

Revisitemos o cálculo do fatorial $n!$ de um inteiro n.

1. Escreva uma função `factorial(n)` que calcule $n!$ para n inteiro. Ao construir essa função *assuma implicitamente* que o argumento n será sempre passado à função como um inteiro. Use essa função para calcular o fatorial de 200, invocando `factorial(200)`.

2. Experimente passar o valor do argumento n como um número em representação de ponto flutuante (*float*), invocando por exemplo `factorial(200.)` em vez de `factorial(200)`. Compare e discuta o que observa em cada caso.

3. Adapte a sua função `factorial(n)` para que se comporte de forma robusta, devolvendo o valor correto do fatorial, mesmo se o argumento n for erradamente passado como *float*.

■ Exercício 1.4 | Constante de Madelung

Em Física da Matéria Condensada, a constante de Madelung representa a energia potencial elétrica total de um ião na rede cristalina, a qual depende das cargas e localizações de todos os outros iões vizinhos. Considere, por exemplo, o caso do cloreto de sódio (NaCl, o sal de cozinha) onde os iões estão dispostos alternadamente numa rede cúbica simples: o de sódio com uma carga positiva $+e$ e o de cloro com uma carga negativa $-e$, onde e representa a carga elementar. Se etiquetarmos cada nodo da rede por três números inteiros (i, j, k), então os iões sódio ocupam as posições onde $i + j + k$ é par, e os iões cloro as posições onde $i + j + k$ é ímpar.

Consideremos o ião sódio situado na origem, $i = j = k = 0$, e calculemos a constante de Madelung. Se o espaçamento entre os iões for a (constante da rede), então a distância da origem ao ião na posição (i, j, k) é

$$\sqrt{(ia)^2 + (ja)^2 + (ka)^2} = a\sqrt{i^2 + j^2 + k^2},$$

e o potencial na origem criado por esse ião é

$$V(i, j, k) = \pm \frac{e}{4\pi\epsilon_0 a\sqrt{i^2 + j^2 + k^2}},$$

onde ϵ_0 é a permitividade no vácuo e o sinal da expressão depende de $i + j + k$ ser par ou ímpar. O potencial total na origem, V_{total} é então a soma sobre todas as restantes posições. Supondo um cristal de formato cúbico centrado na origem, com L iões em todas as direções, temos então

$$V_{\text{total}} = \sum_{\substack{i,j,k=-L \\ (\text{exceto } i=j=k=0)}}^{L} V(i,j,k) = \frac{e}{4\pi\epsilon_0 a} M,$$

onde M é a constante de Madelung. Tecnicamente, a constante de Madelung corresponde ao limite $L \to \infty$, mas podemos obter uma boa aproximação usando L suficientemente grande.

Desenvolva um programa que calcule a constante de Madelung do cloreto de sódio. Use um valor suficientemente grande para L, mas que lhe permita correr o programa num minuto ou menos. Compare com o valor exato para o NaCl: -1.748.

Nota: Conseguirá a abordagem mais eficiente recorrendo a vetorização com arrays e funções da biblioteca `numpy` para avaliar somas de elementos de arrays.

□ Exercício 1.5 | Coeficientes Binomiais (complementar)

O coeficiente binomial $\binom{n}{k}$ é um número inteiro igual a

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)}{1 \times 2 \times \ldots \times k}$$

se $k \geq 1$, ou $\binom{n}{0} = 1$ quando $k = 0$.

1. Usando esta definição, crie uma função `binomial(n,k)` que calcula o coeficiente binomial para dados n e k. Certifique-se de que a sua função retorna a resposta na forma de um inteiro (não um float!), e que retorna o valor correto $\binom{n}{0} = 1$ sempre que $k = 0$.

2. Usando a sua função, escreva um programa para imprimir as primeiras 20 linhas do "triângulo de Pascal". A linha n do triângulo de Pascal contém $n + 1$ números, que correspondem aos coeficientes $\binom{n}{0}$, $\binom{n}{1}$, até $\binom{n}{n}$. As primeiras linhas são,

   1 1
   1 2 1
   1 3 3 1
   1 4 6 4 1

3. A probabilidade de uma moeda justa apresentar k resultados "caras" quando lançada n vezes é dada por $\frac{1}{2^n}\binom{n}{k}$. Escreva um programa que calcula: (a) A probabilidade de obter 60 caras quando lança a moeda 100 vezes; (b) a probabilidade de obter *pelo menos* 60 caras.

□ Exercício 1.6 | Números primos (complementar)

O programa no Exemplo 2.8 do livro adotado não corresponde ao modo mais eficiente de calcular números primos: verifica cada número para ver se é divisível por qualquer número menor que ele.

Podemos desenvolver um programa muito mais rápido para determinar números primos tendo em consideração as seguintes observações:

1. Um número n é primo se não tiver nenhum fator primo menor que n. Por isso, só precisamos verificar se é divisível por outros primos.

2. Se um número n é não primo, tendo um fator r, então $n = rs$, onde s também é um fator. Se $r \geq \sqrt{n}$ então $n = rs \geq s\sqrt{n}$, o que implica que $s \leq \sqrt{n}$. Por outras palavras, qualquer não-primo deve ter fatores e, portanto, também fatores primos, menores ou iguais a $\sqrt{n}$. Assim, para determinar se um número é primo, basta-nos verificar os seus fatores primos apenas até (e incluindo) $\sqrt{n}$; se não houver nenhum, o número é primo.

3. Se encontrarmos um único fator primo menor que $\sqrt{n}$, então sabemos que o número não é primo e, portanto, não há necessidade de verificar mais nada; podemos abandonar esse número e avançar.

Escreva um programa que encontre todos os números primos até dez mil. Crie uma lista para armazenar os primos que for encontrando, a qual contém inicialmente apenas um elemento: o número primo 2. Então, para cada número n de 3 a 10 000, verifique se o número é divisível por qualquer um dos primos da lista, até e incluindo $\sqrt{n}$. Assim que encontrar um único fator primo, poderá parar de verificar os restantes — sabe que n não é um primo. Se não encontrar fatores primos $\sqrt{n}$ ou menos, então n é primo e deve adicioná-lo à lista. Pode imprimir a lista de uma só vez no final do programa, ou pode imprimir cada um à medida que os for encontrando.

□ Exercício 1.7 | Erros de truncagem: equação quadrática (complementar)

Considere uma equação quadrática $ax^2 + bx + c = 0$ com soluções reais.

1. Escreva um programa que tome como dados três números, a, b e c, e imprima as duas soluções da equação quadrática $ax^2 + bx + c = 0$ usando a fórmula resolvente,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

   Use seu programa para calcular as soluções de $0.001x^2 + 1000x + 0.001 = 0$.

2. Existe outra maneira de escrever as soluções para uma equação quadrática. Multiplicando numerador e denominador da expressão acima por $-b \mp \sqrt{b^2 - 4ac}$, mostre que as soluções também podem ser escritas como

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}.$$

   Adicione código ao seu programa para imprimir as soluções obtidas de acordo com ambas as fórmulas resolventes. Use novamente o programa para resolver $0.001x^2 + 1000x + 0.001 = 0$. O que observa? Como o explica?

3. Usando o que aprendeu, crie uma função `quadratic_safe(a,b,c)` que calcule ambas as raízes de uma equação quadrática com precisão em todos os casos.

Comentário: Este é um bom exemplo de como os computadores nem sempre funcionam como se espera. Se aplicar simplesmente a fórmula padrão para a equação quadrática, pode receber uma resposta errada. Na prática, o método que elaborou aqui é a maneira correta de resolver uma equação quadrática num computador, mesmo que seja mais complicado do que a fórmula padrão.

□ Exercício 1.8 | Erros de truncagem: derivada simplista (complementar)

Suponha que temos função $f(x)$ e queremos calcular sua derivada no ponto x. Se soubermos a forma analítica da função, podemos fazê-lo com lápis e papel. Mas, em qualquer caso, podemos sempre fazê-lo no computador, aplicando a definição:

$$\frac{df}{dx} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}.$$

No computador, não podemos estritamente tomar o limite $\delta \to 0$, mas podemos obter uma aproximação razoável fazendo δ suficientemente pequeno.

1. Escreva um programa que defina uma função `f(x)` que devolve o valor $x(x-1)$, e que calcula a derivada dessa função no ponto $x = 1$ usando a fórmula acima com $\delta = 10^{-2}$. Calcule o valor real da mesma derivada analiticamente e compare com a resposta do seu programa. Os dois não concordarão perfeitamente. Por que não?

2. Repita o cálculo para para $\delta = 10^{-4}$, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}, e 10^{-14}. Verá que a precisão do cálculo inicialmente melhora à medida que o delta diminui, mas piora novamente. Porquê?