Lista 1

F 625 - 2s/2019 - Grupo 8
Pedro V. Pinho - 185886
Thiago Virgilio - 093112

22 de agosto de 2019

Exercício 2.2: Altitude de um satélite

2.2a)

A força que atua em um satélite que orbita a Terra em uma trajetória circular - desprezando arrasto com a fina atmosfera presente e efeitos gravitacionais de outros corpos celestes - é, simplesmente, a força gravitacional. A "força"centrifuga não é uma força real, apenas uma força aparente causada pela inercia de um corpo em um movimento circular. O movimento do corpo é então descrito por:

$$\vec{F} = m \cdot \frac{d^2\vec{r}}{dt^2}$$

Escrevendo $\vec{r} = r(cos(\theta), sin(\theta))$ e $\vec{F} = -\frac{GMm}{r^2}$ - onde **G** é a constante de gravitação universal, **M** é a massa da Terra, **m** é a massa do satélite e **r** é a distância até o centro do corpo central - temos que:

$$m \cdot r \left(\frac{d\vec{\theta}}{dt}\right)^2 = \frac{GMm}{r^2}$$

Para movimentos circulares podemos escrever $\frac{d\theta}{dt} = \omega = \frac{2\pi}{T}$, onde T é o período da órbita, portanto:

$$T^2 = \frac{4\pi^2 r^3}{GM}$$

Para encontrar a altura do satélite, em relação ao solo, em função do período, basta inverter a equação e notar que se o raio da Terra é **R**, então a altura do satélite, **h**, é dada por $\mathbf{h = r - R}$:

$$h = \left(\frac{GMT^2}{4\pi^2}\right)^{1/3} - R$$

2.2b)

Um programa simples que pode ser empregado para o calculo da altura é apresentado abaixo:

```
# -*- coding: utf-8-sig -*-
import numpy as np

##############################################
#--Um programa que toma como input o período--#
#--de órbita do satélite e calcula sua--------#
#--altura em relação ao solo------------------#
##############################################
def alt_sat(T):
    G = 6.67e-11      #[m^3 kg^-1 s^-2]
    M = 5.97e24       #[kg]
    R = 6.371e6       #[m]

    r = ((G*M*T**2)/(4*np.pi**2))**(1/3)
    return r - R

#Recebendo dados do usuário e convertendo para segundos
T_horas = float(input("Período do satélite (em horas): "))
T_sec   = T_horas * 3600

h = alt_sat(T_sec)

print('A orbita é %.4E km acima do solo.' %(h/1000))
```

2.2c)

Analisando a tabela 1 fica obvio que orbitas mais distantes possuem períodos maiores, porém a informação mais importante que pode ser tirada disso é que não existem orbitas com período de 45 minutos.

Período (horas)	Altura (km)
24	35856
1.5	279.3
0.75	-2181
23.93	35774

Tabela 1: Tabela da relação entre altura em relação ao solo e o período de orbita do satélite

2.2d)

Um dia sideral representa o tempo necessário para a Terra dar uma volta, em seu eixo, em relação um ponto distante no cosmo - estrelas muito distantes que parecem fixas em um curto

período de tempo. O dia sideral é 4 minutos menor que o dia solar devido ao movimento de translação da Terra. A diferença entre orbitas de 24 horas e 23.96 horas é de apenas 82 km.

Exercício 2.5: Degrau de potencial

```
# -*- coding: utf-8-sig -*-
import numpy as np

#--Sequência de programas que calculam---#
#--respectivamente a probabilidade de----#
#--transmissão e reflexão de um----------#
#--elétron com energia "E_electron" ao---#
#--encontrar uma barreira de potencial---#
#--de valor "E_step"--------------------#

def transmission(E_step, E_electron):
    if E_electron <= E_step:
        print("ERRO: ENERGIA DO ELÉTRON DEVE SER MAIOR QUE POTENCIAL")
    m    = 9.11e-31                                  #[kg]
    hbar = 6.58e-16                                  #[eV*s]
    k1   = np.sqrt(2*m*E_electron)/hbar
    k2   = np.sqrt(2*m*(E_electron - E_step))/hbar
    T    = (4*k1*k2)/((k1 + k2)**2)
    return T

def reflection(E_step, E_electron):
    if E_electron <= E_step:
        print("ERRO: ENERGIA DO ELÉTRON DEVE SER MAIOR QUE POTENCIAL")
    m    = 9.11e-31                                  #[kg]
    hbar = 6.58e-16                                  #[eV*s]
    k1   = np.sqrt(2*m*E_electron)/hbar
    k2   = np.sqrt(2*m*(E_electron - E_step))/hbar
    R    = ((k1 - k2)/(k1 + k2))**2
    return R


#Recebendo dados do usuário
E_electron = float(input("Energia do elétron (em eV): "))
E_step     = float(input("Energia do potencial (em eV): "))

#Calculando as probabilidades
T, R = transmission(E_step, E_electron), reflection(E_step, E_electron)

print("As probabilidades são T = %.2f e R = %.2f" %(T,R))
```

Para um elétron de energia igual a 10 eV encontrando um barreira de potencial igual a 9 eV temos que as probabilidades de transmissão e reflexão são igual a 73% e 27%, respectivamente. Um resultado interessante que mostra a natureza ondulatória da matéria.

Exercício 2.9: Constante de Madelung

```
# -*- coding: utf-8-sig -*-
import numpy as np

################################################
#--Programa que recebe o número de atomos (N)----#
#--na aresta de um cubo com N^3 átomos de NaCL---#
#--e calcula a constante de Madelung-------------#
#--aproximada para esse número de átomos.--------#
################################################

# Recebendo o número de átomos
L  = int(input("Número de átomos em uma aresta do cubo: "))
M = 0

# Varrendo para cada átomo nas posições (i,j,k)
for i in range(-L, L+1):
    for j in range(-L, L+1):
        for k in range(-L, L+1):
            # Cálculo da distância até o átomo central
            d = np.sqrt(i**2 + j**2 + k**2)
            # Ignorando o átomo central
            if (i == j == k == 0):
                continue

            M_parc = 1/d
            # Alternando entre íons negativos e positivos
            if (i + j + k)%2 == 1:
                M_parc *= -1
            # Adicionando contribuição individual ao valor final
            M += M_parc

print('Constante de Madelung para o cristal de NaCl: %.3f' %abs(M))
```

Este é um código simples que calcula o inverso da distância à origem de cada átomo em uma rede cristalina de NaCl e as somam. A menos de constantes, a Constante de Madelung é simplesmente o potencial elétrico sentido por um átomo devido todos os outros átomos da rede cristalina. Assim sendo, se o átomo contribuinte é um íon de Sódio, sua contribuição é positiva devido seu excesso de prótons, o oposto ocorre caso o contribuinte seja um íon de Cloro (Cl). A verdadeira Constante de Madelung aparece ao considerar todos os átomos do matérial, entretanto isso é computacionalmente impossível e utilizando valores altos de **L** conseguimos

uma resposta suficientemente próxima.

O valor real para a Constante de Madelung para o cristal de NaCl é de 1.748. Podemos ver, na tabela 2, que com 10000 átomos considerados, temos um valor aproximado de 1.742, um erro de 0.3%.

L^3	|M|
100	1.693
400	1.719
2500	1.736
10000	1.742

Tabela 2: Convergência da Constante de Madelung

Exercício 3.2: Gráficando Curvas Polares

Se temos uma função polar em função de θ e r, podemos utilizar as transformações para coordenadas cartesianas:

$$x = r \cdot cos(\theta)$$
$$y = r \cdot sin(\theta)$$

E assim, poderemos gráficar tais funções utilizando os métodos comuns. Abaixo podemos ver 3 códigos que fazem uso dessa técnica e os gráfico gerados por eles:

```
# -*- coding: utf-8-sig -*-
import numpy as np
import matplotlib.pyplot as plt

#Criação do array com valores Teta
a = np.linspace(0, 2*np.pi, 100)

#Conversão para cartesianas
x = 2*np.cos(a) + np.cos(2*a)
y = 2*np.sin(a) - np.sin(2*a)

#Gráfico
plt.plot(x, y, '.k-')
plt.ylim(-3,3)
plt.grid(alpha = 0.5)
#plt.savefig('ex_3.2a.jpg', dpi = 300)
plt.show()
```

```
# -*- coding: utf-8-sig -*-
import numpy as np
```

```
import matplotlib.pyplot as plt

#Criação do array com valores Teta e r
a = np.linspace(0, 10*np.pi, 300)
r = a**2

#Conversão para cartesianas
x = r*np.cos(a)
y = r*np.sin(a)

#Gráfico
plt.plot(x, y, '.k-')
plt.grid(alpha = 0.5)
#plt.savefig('ex_3.2b.jpg', dpi = 300)
plt.show()
```

```
# -*- coding: utf-8-sig -*-
import numpy as np
import matplotlib.pyplot as plt

#Criação do array com valores Teta e r
a = np.linspace(0, 12*np.pi, 1000)
r = np.exp(np.cos(a)) - 2*np.cos(4*a) + (np.sin(a/12))**5

#Conversão para cartesianas
x = r*np.cos(a)
y = r*np.sin(a)

#Gráfico
plt.plot(x, y, '.k-', markersize = 2)
plt.grid(alpha = 0.5)
#plt.savefig('ex_3.2c.jpg', dpi = 300)
plt.show()
```

Exercício 3.4:

Um código capaz de reproduzir a distribuição de átomos de uma estrutura cristalina do tipo CS, *Cubic Simple* é composto apenas de um entrelaçamento de loops **for** com intuito de varrer todas as posições (i,j,k) discretamente e posicionar um átomo em cada ponto. Uma ressalva deve ser feita: como é de interesse reproduzir um cristal de NaCl, é necessário que os átomos de cada elemento sejam diferentes, assim, quando a posição de um átomo for ímpar - entende-se como posição ímpar aquela cuja soma das coordenadas é ímpar - a esfera na figura será amarela, representando o átomo de Cloro, caso contrário, será branca para o átomo de Sódio. Um exemplo de código é apresentado abaixo:

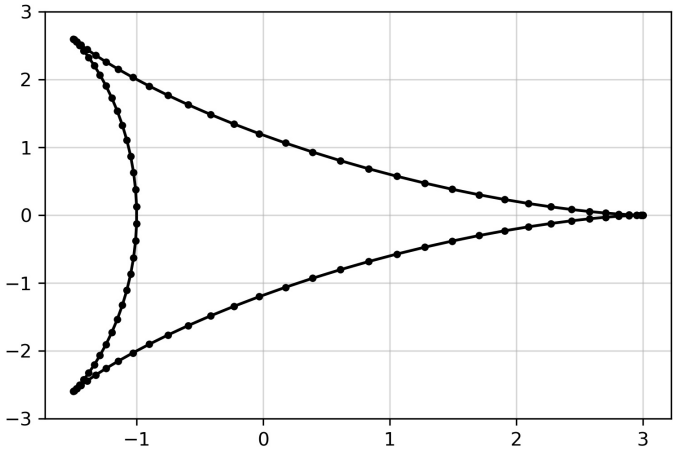

Figura 1: Gráfico para $x = 2cos(\theta) + cos(2\theta)$ e $y = 2sin(\theta) - sin(2\theta)$

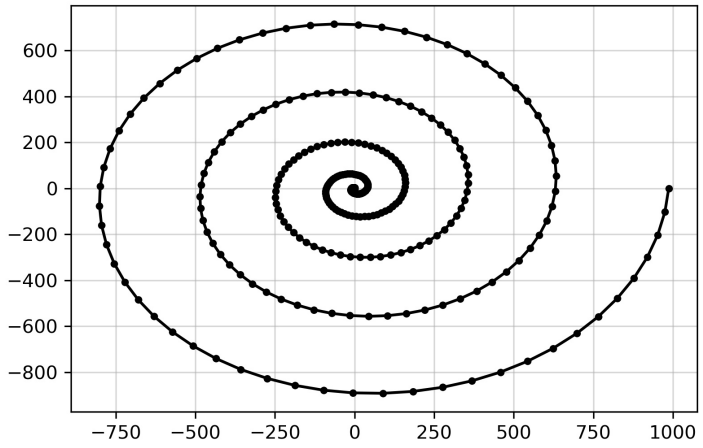

Figura 2: Gráfico para $r = \theta^2$

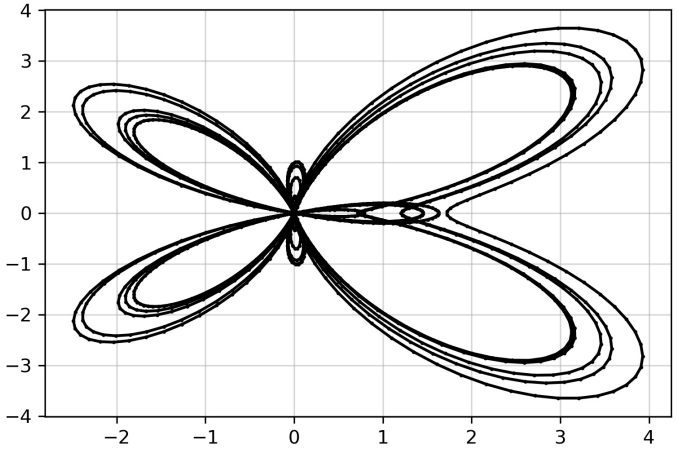

Figura 3: Gráfico para $r = e^{cos(\theta)} - 2cos(4\theta) + sin^5(\frac{\theta}{12})$

```
# -*- coding: utf-8 -*-
from vpython import sphere, vector, canvas

# O número de átomos totais é (2N+1)^3 cujos raios são iguais a R
L = 2
R = 0.5

# Inicializando a figura
scene = canvas(title='Sodium Cloride Lattice')
# Varrendo cada posição (i,j,k) e colocando um átomo
for i in range(-L, L + 1):
    for j in range(-L, L + 1):
        for k in range(-L, L + 1):
            # Alternando entre átomos de sódio e de cloro
            if (i+j+k)%2 == 0:
                sphere(pos = vector(i,j,k), radius = R, color = vector(1,1,0))
            else:
                sphere(pos = vector(i,j,k), radius = R)
```

A construção de um código capaz de desenhar uma estrutura do tipo FCC, *Face Centered Cubic*, é imediata se for facilmente notado que uma FCC é uma CS onde os átomos em posições ímpares estão faltando. Assim, restam apenas os átomos nos vértices e no centro de cada face. Abaixo apresenta-se um exemplo de código:

```
# -*- coding: utf-8 -*-
from vpython import sphere, vector, canvas

#####################################################
#--Programa que desenha uma estrutura cristalina--#
#--de tipo FCC - face-centered cubic. A lógica----#
#--se basea no fato que a FCC pode ser vista------#
#--como uma CS onde os átomos de posição impar----#
#--estão em falta. Posição ímpar sendo aquela-----#
#--onde (i+j+k) é ímpar.-------------------------#
#####################################################

# Número de átomos na aresta é (L+1)/2 cujos raios são R
L = 7
R = 0.7

#Inicializando a figura
scene = canvas(title='Fcc Lattice')
#Varrendo para cada posição (i,j,k)
for i in range(L):
    for j in range(L):
        for k in range(L):
            # Se a posição for par adiciona um átomo, c.c pula a posição
            if (i+j+k)%2 == 0:
```

```
            sphere(pos = vector(i,j,k), radius = R)
        else:
            continue
```

As figuras 4 e 5 apresentam os resultados de cada um dos programas:

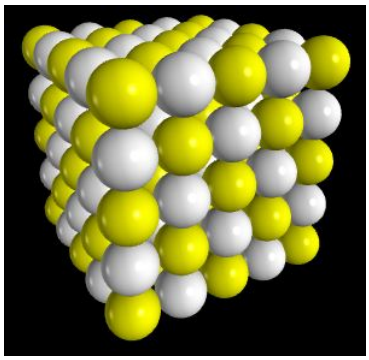

Figura 4: Cúbica Simples com 125 átomos em seu volume

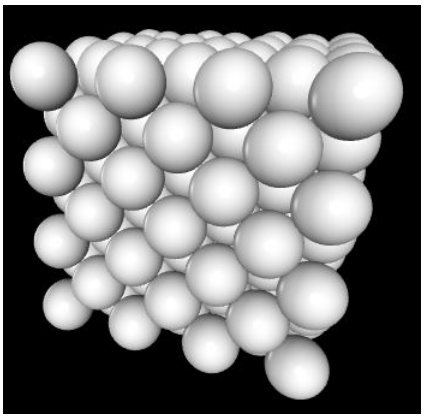

Figura 5: Cúbica de face centrada com 104 átomos em seu volume

Exercício 3.7: Conjunto de Mandelbrot

Seja a equação complexa $z' = z^2 + c$, onde **z** e **c** são números complexos. Para um valor inicial de **z**, seja $z = 0$, temos um novo valor de **z'**, dependendo do valor de **c**. Se após infinitas interações o módulo de **z'** não for maior que 2 esse ponto é considerado dentro do conjunto de Mandelbrot. O valor de **c** pode ser qualquer ponto do plano complexo ($c = x + iy$).

Para construir um programa que reproduza uma aproximação do conjunto de Mandelbrot basta dividir o plano complexo em elementos (x,y) e calcular o valor de **c**, a partir disso e de um valor inicial de **z** basta iterar com um número suficiente de vezes - digamos 100 - e sempre verificar o módulo de **z'**.

```
# -*- coding: utf-8-sig -*-
import numpy as np
import matplotlib.pyplot as plt
```

```
#####################################
#--Função que calcula o módulo de----#
#--z = z' + c por método iterativo---#
#--apartir de um ponto inicial ------#
#--z = 0. Caso no final o módulo-----#
#--seja maior que 2 então o z é------#
#--considerado fora do conjunto------#
#--de Mandelbrot e a função retorna--#
#--o número de iterações necessárias-#
#--para que z tomasse módulo maior---#
#--2.--------------------------------#
#####################################
def mandelbrot(x,y):
    c = complex(x,y)
    z = 0
    for i in range(1,100):
        if abs(z) > 2:
            return np.log(i)
        z = z**2 + c
    return 0

#####################################
#--A número de subdivisões do grid---#
#--é de N*N. Cada ponto é separado---#
#--por d. Assim, a extensão do plano-#
#--considerado é de |x|<=2.5 e-------#
#--|y|<=2.5--------------------------#
#####################################
N = 1000
d = 0.004

# Inicialização do grid
grid = np.empty([N,N], np.float)

#####################################
#--Varredura por todos os pontos do--#
#--grid calculando os valores de-----#
#--x e y da variavel z = x + iy.-----#
#--O valores de N/2 e 2N/3 servem----#
#--para fazer a translação do--------#
#--conjunto para que o mesmo apareça-#
#--por completo na figura------------#
#####################################
for i in range(N):
    y = d*(i - N/2)
    for j in range(N):
        x = d*(j - N/2)
        grid[i,j] = mandelbrot(x,y)
```

```
# Inicializando a figura e salvando-a
plt.figure(figsize = (20,20))
plt.imshow(grid,cmap='hot')
#plt.savefig('mandelbrot.jpg', dpi = 300)
plt.show()
```

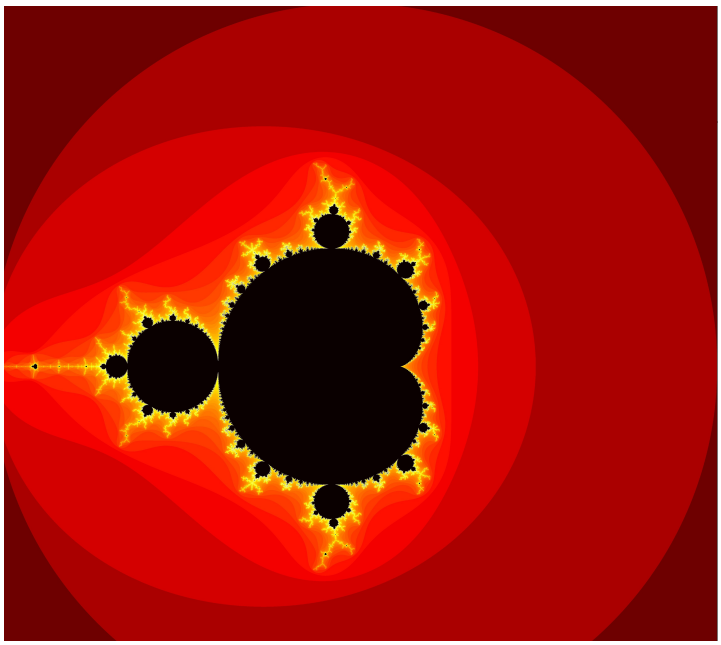

Figura 6: Conjunto de Mandelbrot em um grid de N x N

A figura 6 apresenta o resultado do código utilizando $N = 10.000$, com 100 interações para cada ponto. O código demorou 59 minutos para executar. Podemos ver bastantes detalhes do Conjunto de Mandelbrot.

Exercício 4.2: Equação Quadráticas

```
# -*- coding: utf-8 -*-
########################################
#--Função que recebe os coeficientes da---#
#--equação ax^2 + bx + c = 0 e calcula----#
#--as raizes do polinómio por meio da-----#
#--equação x = (-b +- sqrt(b^2 - 4ac))/2a-#
########################################

def quad(a, b, c):
    sol = [(-b + (b**2 - 4*a*c)**0.5)/(2*a), (-b - (b**2 - 4*a*c)**0.5)/(2*a)]
    return sol

# Recebendo os coeficientes e atribuindo as variaveis
a,b,c = map(float, input("Coeficientes (a,b,c): ").split())
```

```
# Calculando soluções
sol = quad(a, b, c)

print('\nAs soluções sÃo: {:.12f} e {:.12f}'.format(sol[0], sol[1]))

# -*- coding: utf-8 -*-
##########################################
#--Função que recebe os coeficientes da---#
#--equação ax^2 + bx + c = 0 e calcula----#
#--as raízes do polinômio por meio da-----#
#--equação x = 2c/(-b -+ sqrt(b^2 - 4ac))-#
##########################################

def quad(a, b, c):
    sol = [(2*c)/(-b - (b**2 - 4*a*c)**0.5), (2*c)/(-b + (b**2 - 4*a*c)**0.5)]
    return sol

# Recebendo os coeficientes e atribuindo às variaveis
a,b,c = map(float, input("Coeficientes (a,b,c): ").split())
# Calculando soluções
sol = quad(a, b, c)

print('\nAs soluções são: {:.12f} e {:.12f}'.format(sol[0], sol[1]))
```

A equação quadrática utilizada foi $0.001x^2 + 1000 + 0.001$. É fácil ver que teremos problemas de cálculo de ponto flutuante dependendo de como resolvermos essa equação. Nos códigos acima são apresentados duas formas de achar as raízes de uma equação de segundo grau. De acordo com o primeiro código as raízes são: $x_1 = -0.999989 \cdot 10^{-6}$ e $x_2 = -1 \cdot 10^6$; de acordo com o segundo código as soluções são $x_1 = -1 \cdot 10^{-6}$ e $x_2 = -1.00001058 \cdot 10^6$.

Note que em ambos os códigos uma das soluções tem em sua expressão o termo: $(-b + \sqrt{(b^2 - 4ac)})$. Entretanto, o valor da raiz, neste termo, é muito próximo de b, ao subtrair ambos valores é perdida informação no arredondamento e temos um valor errado para a raiz. Portanto, um dos métodos é útil para calcular a menor raiz e o outro é útil para calcular a raiz maior. Um bom código seria aquele que utilizasse os dois métodos, uma para cada raiz. Abaixo temos um exemplo de tal código:

```
# -*- coding: utf-8-sig -*-
##########################################################
#--Função que corrige o problema de cálculo-------------#
#--de ponto flutuante ao achar a raiz-------------------#
#--maior de ax^2+bx+c por meio da equação---------------#
#--(-b - (b**2 - 4*a*c)**0.5)/(2*a) e a raiz------------#
#--menor por meio de (2*c)/(-b - (b**2 - 4*a*c)**0.5).--#
##########################################################
def quad_opt(a, b, c):
    sol = [(2*c)/(-b - (b**2 - 4*a*c)**0.5), (-b - (b**2 - 4*a*c)**0.5)/(2*a)]
    return sol
```

```
# Recebendo dados do usuário
a,b,c = map(float, input('Coeficientes: ').split())

# Calculando as raizes
sol_opt = quad_opt(a, b, c)

print('\nAs soluções são: {:.12E} e {:.12E}'.format(sol_opt[0], sol_opt[1]))
```

As raízes calculadas por esse programa são as esperadas: $x_1 = -1 \cdot 10^{-6}$ e $x_2 = -1 \cdot 10^{6}$.

Exercício 4.3: Calculando derivadas

A função $f(x) = x(x-1)$, analiticamente, tem derivada igual a: $f'(x) = 2x - 1$. Portanto, no ponto $x = 1$ assume valor igual a $f'(1) = 1$. Podemos calcular a derivada de uma função computacionalmente fazendo uso de uma de suas definições. Seja $f(x)$ uma função contínua e diferenciável:

$$f'(a) = \lim_{\delta \to 0} \frac{f(a+\delta) - f(a)}{\delta} \tag{1}$$

Fazendo δ muito pequeno podemos encontrar aproximações razoáveis para o real valor de uma derivada. Abaixo podemos ver um exemplo de código que tem esse papel:

```
# -*- coding: utf-8-sig -*-
# Definindo a função que calcula f(x) = x(x-1)
def func(x):
    return x*(x - 1)

################################################
#--Definfindo a função que calcula a derivada--#
#--por meio do metodo de divisão infinitesimal-#
#--ela recebe como argumentos uma função-------#
#--pré-definida, o ponto onde se calcular a----#
#--derivada e o intervalo delta----------------#
################################################

def derivative(f, a, d):
    deriv = (func(a + d) - func(a))/(d)
    return deriv

print("A derivada de f(x) = x(x-1) em x = 2 é: %.5f" %derivative(func, 1, 1e-2))
```

Para um valor de $\delta = 10^{-2}$ temos que a derivada no ponto x = 1, pelo programa, é de $f'(1) = 1.01$. A equação 1 calcula a derivada da função por meio de uma linearização da mesma, a medida que δ se torna cada vez menor, temos que a reta tangente no ponto $x = a$ se aproxima cada vez mais da função. Assim, como o valor de δ não é infinitesimal, temos uma discrepância entre o valor real da derivada e o valor computacional.

δ	$f'(1)$
10^{-2}	1.01
10^{-4}	1.0001
10^{-6}	1.000001
10^{-8}	1.000000004
10^{-10}	1.00000008
10^{-12}	1.00009

Tabela 3: Erro no calculo de $f'(1)$ com relação ao valor de δ

Por meio da tabela 3 podemos ver que com a diminuição do valor de δ temos uma melhora na precisão de $f'(1)$, entretanto, se δ é muito pequeno a precisão diminui. Isso é causado por erros no cálculo de ponto flutuante. Com δ muito pequeno ocorre que $f(a+\delta)$ e $f(a)$ se tornam valores muito próximos e informação é perdida no arredondamento dos cálculos por falta de memória.

Exercício 4.4: Calculando integrais

A definição de Riemann para a integral de uma função $f(x)$ consiste em dividir a região, envolta pela curva definida pela função, em pequeno retângulos. Para isso, o intervalo de integração ([a,b]) é dividido em partições de tamanhos iguais h, onde $h = (b-a)/N$, sendo N o número de partições e cada retângulo tem área igual a $y \cdot h$. Assim, a integral, I, é dada por:

$$I = \lim_{N \to \infty} \sum_{k=1}^{N} h \cdot y_k$$

Abaixo temos um exemplo de código que utiliza a Integração de Riemann para calcular a integral de $f(x) = \sqrt{1-x^2}$, no intervalo [-1,1], com N = 100:

```
# -*- coding: utf-8-sig -*-
import numpy as np

# Número de partições
N   = 100
# Tamanho das partições
h   = 2/N

####################################
#--Array com valores de -1 até 1---#
#--separados por h. Total de N ----#
#--partições. A integral é a soma--#
#--dos retângulos de base h e------#
#--altura y------------------------#
####################################

x   = np.arange(-1, 1, h)
```

```
y    = np.sqrt(1 - x**2)

integral  = sum(h*y)
print('A integral é: %.4f'%integral)
```

Com $N = 100$ encontramos um valor de $I = 1.569$, uma divergência de 0.11% do valor real de $\pi/2$. A maior precisão que foi atingida, no computador do autor deste relatório, requerendo que o código rodasse em menos de 1 segundo, foi de 0.00000001%, utilizando $N = 10^6$.