EP3 - Cálculo Numérico

Professor: Arnaldo Gammal
Monitor: Raphael

Aluno: Jhoão Gabriel Martins Campos de Almeida Arneiro
Nº USP: 10819721

Questão 1

Desenvolvemos nesta questão um programa capaz de resolver numericamente, em precisão simples, a integral $\int_0^1 (7 - 5x^4)dx$ utilizando o método de Trapézios. Tal programa está transcrito abaixo e nele utilizamos $N = 2^{10} = 1024$ passos de integração.

```
from numpy import float32

#Funcoes uteis
def f(x):
    ''' Funcao utilizada para calcular o valor do integrando em um ponto
        arbitrario x em precisao simples'''

    integ = float32(7 - 5*x**4)

    return integ

def Trapezio(a,b,N):
    ''' A partir de um numero de intervalos N e um intervalo de integracao
        [a,b], realiza uma integracao numerica da funcao f(x) pelo
        metodo de trapezios em precisao simples de a ate b em passos de
        tamanho h = 1/N'''

    #Iniciamos tomando o valor de cada passo h = 1/N e o valor inicial do
    #somatorio da integral = 0
    h = float32(1/N)
    T = float32(0)

    #Setamos o valor inicial de x_{i+1} como o inicio do nosso intervalo [a,b]
    x_i_1 = float32(a)

    #Eh iniciado entao um loop de N passos para realizar a integracao numerica
    for i in range(N):
```

```
        #Colocamos os valores de x_i e x_{i+1}, onde x_i toma o valor anterior
        #de x_{i+1}, e x_{i+1} toma o proximo valor de x = x_i + h (o valor
        #de x_i mais o tamanho do passo)
        x_i = float32(x_i_1)
        x_i_1 = float32(x_i + h)

        #Calculamos entao o novo termo da somatoria e entao o adicionamos ao
        #nosso valor total da integral
        termo = float32(h*((f(x_i) + f(x_i_1))/2))
        T = float32(T + termo)

    return T

#Valores extremos do intervalo [a,b] de integracao
a = 0
b = 1

#Definimos o nosso numero de passos N
N = 2**10

#Realizamos a integral para N passos, indo de a ate b
Integral = float32(Trapezio(a,b,N))

#Printamos por fim o valor obtido da integral
print("A integral de 0 a 1 de f(x) = 7 - 5x^4, calculada pelo metodo de
    Trapezios em precisao simples, com %i passos de integracao eh dada por I =
    %f" %(N, Integral))

>>> A integral de 0 a 1 de f(x) = 7 - 5x^4, calculada pelo metodo de Trapezios
     em precisao simples, com 1024 passos de integracao eh dada por I =
    5.999997
```

Item a)

Adaptando o programa acima com um laço para realizar a integração para diferentes intervalos de integração, dados por $N = 2^p$, onde $p \in \{1, 25\}$ é o número do laço, fizemos a tabela abaixo, onde $erro = |I_{num} - I|$, sendo I_{num} o valor calculado numericamente para a integral e sendo I o valor analítico da integral, dado por:

$$\int_0^1 (7 - 5x^4)dx = \left[7x - x^5\right]_0^1 = 7 - 1 = 6$$

Sendo assim, os resultados obtidos foram:

p	N	I_{num}	$erro$
1	2	5.593750	0.406250
2	4	5.896484	0.103516
3	8	5.973999	0.026001
4	16	5.993492	0.006508
5	32	5.998372	0.001628
6	64	5.999592	0.000408
7	128	5.999898	0.000102
8	256	5.999973	0.000027
9	512	5.999988	0.000012
10	1024	5.999997	0.000003
11	2048	5.999996	0.000004
12	4096	6.000007	0.000007
13	8192	6.000003	0.000003
14	16384	6.000004	0.000004
15	32768	5.999997	0.000003
16	65536	6.000094	0.000094
17	131072	6.000277	0.000277
18	262144	5.999516	0.000484
19	524288	6.002491	0.002491
20	1048576	6.007617	0.007617
21	2097152	5.995615	0.004385
22	4194304	6.044366	0.044366
23	8388608	5.722580	0.277420
24	16777216	6.564703	0.564703
25	33554432	4.000000	2.000000

Item b)

Repetimos o mesmo processo empregado no item anterior, desta vez adaptando nosso código para realizar a integração em precisão dupla. Este código nos permitiu gerar a seguinte tabela:

p	N	I_{num}	$erro$
1	2	5.593750000000000	0.406250000000000
2	4	5.896484375000000	0.103515625000000
3	8	5.973999023437500	0.026000976562500
4	16	5.993492126464844	0.006507873535156
5	32	5.998372554779053	0.001627445220947
6	64	5.999593108892441	0.000406891107559
7	128	5.999898275360465	0.000101724639535
8	256	5.999974568723701	0.000025431276299
9	512	5.999993642173649	0.000006357826351
10	1024	5.999998410543148	0.000001589456852
11	2048	5.999999602635665	0.000000397364335
12	4096	5.999999900658928	0.000000099341072
13	8192	5.999999975164746	0.000000024835254
14	16384	5.999999993791160	0.000000006208840
15	32768	5.999999998447826	0.000000001552174
16	65536	5.999999999611941	0.000000000388059
17	131072	5.999999999903161	0.000000000096839
18	262144	5.999999999975759	0.000000000024241
19	524288	5.999999999994185	0.000000000005815
20	1048576	5.999999999998548	0.000000000001452
21	2097152	5.999999999999328	0.000000000000672
22	4194304	6.000000000000703	0.000000000000703
23	8388608	6.000000000000163	0.000000000000163
24	16777216	5.999999999997827	0.000000000002173
25	33554432	5.999999999999432	0.000000000000568

Utilizando os resultados das duas tabelas acima, confeccionamos um gráfico de $log_{10}(erro)$ em função do número de iteração p.

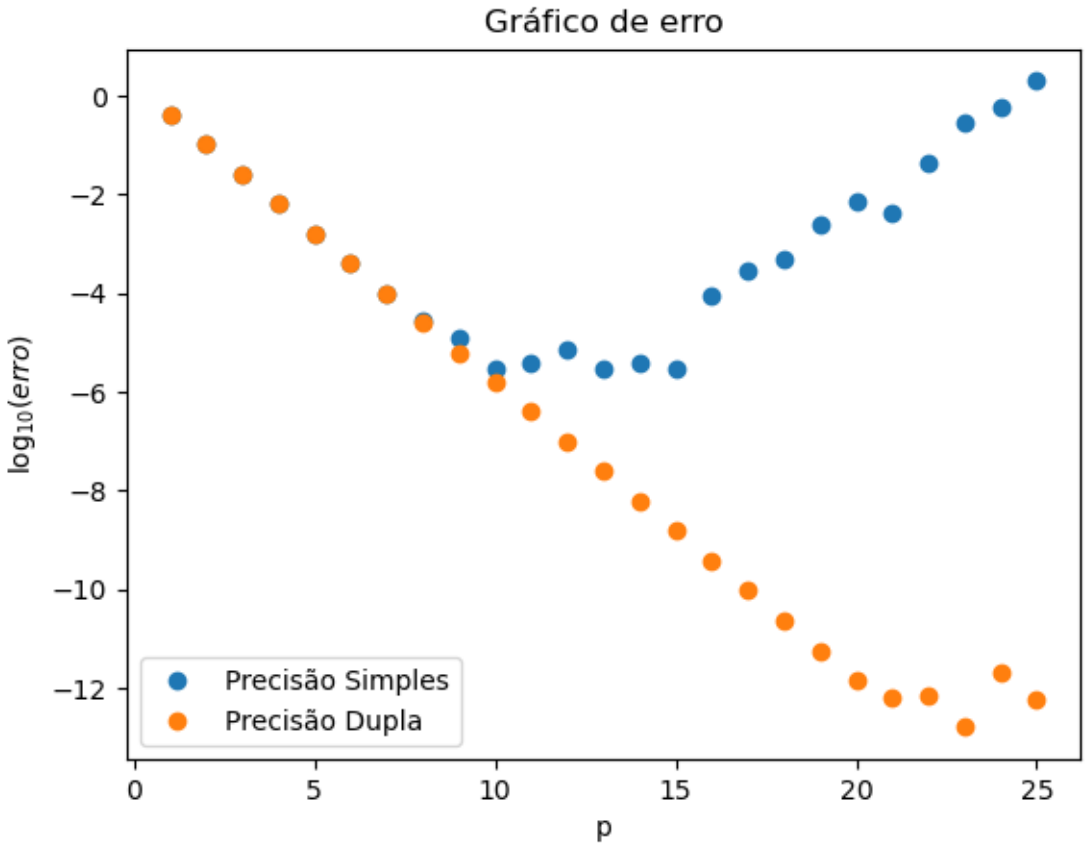

Figura 1: Relação entre os erros

Em aula estudamos como o comportamento do erro deveria variar tanto pelo truncamento do método de trapézios quanto pelo round-off. Segundo nossas previsões, a ordem de grandeza do erro do método de trapézios deveria variar com $\mathcal{O}(h^2)$, isto é, o erro deveria ter a forma $erro = Ah^2$ (onde A é uma constante associada ao método utilizado), o que implicaria que, ao tomar o gráfico do logaritmo do erro em função de p, e recordando que $h = \frac{1}{N} = \frac{1}{2^p}$, então

$$\log_{10}(erro) = \log_{10}(Ah^2)$$

$$= \log_{10}(A) + \log_{10}(h^2)$$

$$= \log_{10}((\frac{1}{N})^2) + \log_{10}(A)$$

$$= \log_{10}((\frac{1}{2^p})^2) + \log_{10}(A)$$

$$= \log_{10}((2)^{-2p}) + \log_{10}(A)$$

$$= -2p\log_{10}(2) + \log_{10}(A)$$

$$= [-2\log_{10}(2)]p + [\log_{10}(A)]$$

Isto é, nós deveríamos observar uma reta com coeficiente angular $m_t = -2\log_{10}(2) \simeq -0.60206$ que representa o erro devido ao truncamento do método de trapézios decaindo a medida que nós aumentamos o nosso número de intervalos N, ou melhor, mostra o erro diminuindo a medida que p aumenta. Se tivéssemos um computador de infinita precisão, este seria o único comportamento observado em nosso gráfico, porém como bem sabemos, nossos computadores possuem precisão limitada, o que acaba por gerar erros devido ao round-off realizado pelo computador quando a precisão exigida pelas nossas contas ultrapassa a precisão do computador.

Vimos em aula que o erro associado ao round-off é de ordem $\mathcal{O}(\sqrt{N})$, e que ele pode ser escrito como $erro = lh^{-\frac{1}{2}} = l\sqrt{N}$, onde l é um valor associado a precisão do nosso programa (para referência, $l \simeq 10^{-7}$ para precisão simples e $l \simeq 10^{-15}$ para precisão dupla), sendo assim, nós obtemos a partir deste erro que

$$\log_{10}(erro) = \log_{10}(l\sqrt{N})$$

$$= \log_{10}(l) + \log_{10}(N^{\frac{1}{2}})$$

$$= \log_{10}((2^p)^{\frac{1}{2}}) + \log_{10}(l)$$

$$= \log_{10}((2)^{\frac{p}{2}}) + \log_{10}(l)$$

$$= \frac{p}{2}\log_{10}(2) + \log_{10}(l)$$

$$= [\frac{1}{2}\log_{10}(2)]p + [\log_{10}(l)]$$

Isto é, o erro devido ao round-off deve se apresentar em nosso gráfico como uma reta de coeficiente angular $m_{ro} = \frac{1}{2}\log_{10}(2) \simeq 0.15051$.

Se observarmos novamente o nosso gráfico, podemos ver três regimes de comportamento para precisão simples e dois para precisão dupla. Para a precisão simples, vemos um decaimento linear no erro, de $p = 1$ até $p \simeq 8$, e então um crescimento mais desordenado, seguindo

dois comportamentos aproximadamente lineares entre $p \simeq 8$ e $p \simeq 16$ e entre $p \simeq 16$ até o fim do gráfico. Já para a precisão dupla, nós observamos o mesmo decaimento linear observado na precisão simples, contudo desta vez ele se estende até $p \simeq 20$, para depois também iniciar um comportamento desordenado crescente, que grosseiramente assumiremos ser linear. Se nós realizarmos ajustes lineares para os comportamentos relativos aos erros de truncamento (decaimentos) e relativos aos erros de round-off em primeira aproximação (crescimentos menos acentuados), obtemos o seguinte gráfico

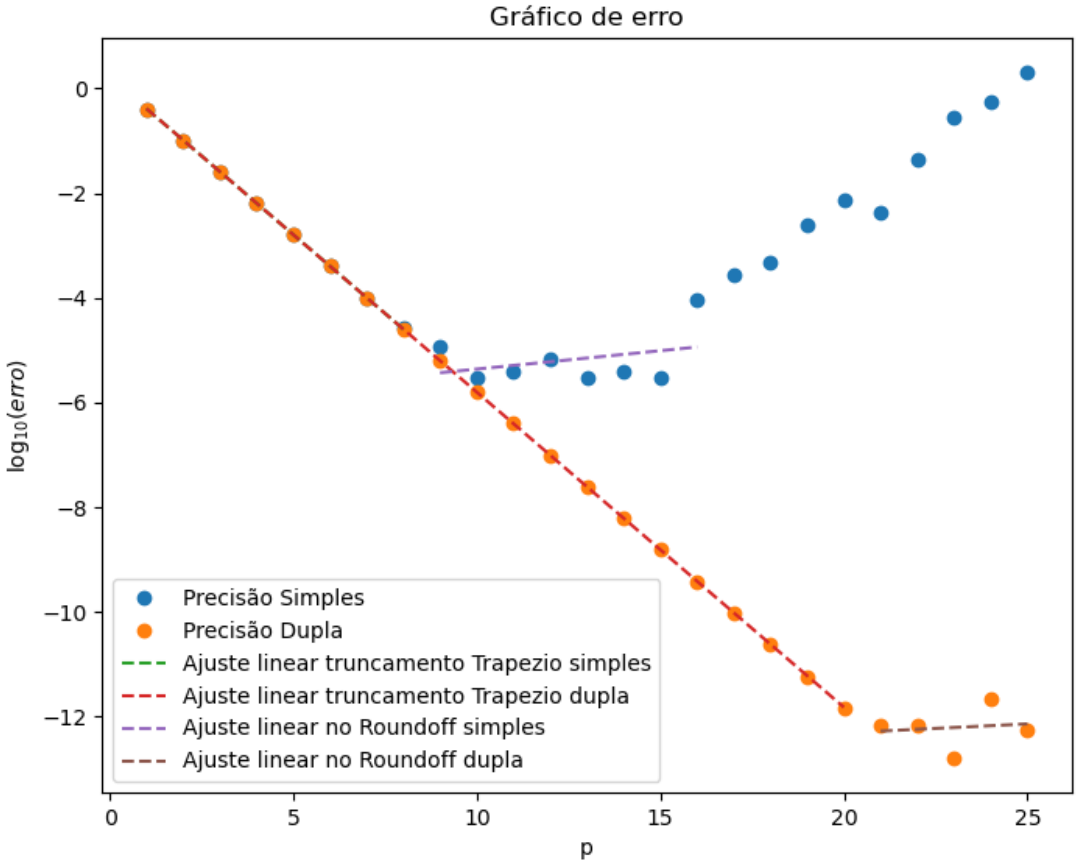

Figura 2: Ajuste para cada regime de erro

Analisando nossos ajustes, vemos que, em ambas as precisões, o ajuste relativo ao erro por truncamento do método de trapézios possuem coeficiente angular $m_t = -0.60183(11)$, o que está próximo da nossa previsão teórica de que $m_t \simeq -0.60206$, mostrando que de fato, o erro do método aumenta com $\mathcal{O}(h^2)$. Já os ajustes relativos aos erros por round-off não se adéquam tão bem a previsão teórica de $m_{ro} \simeq 0.15051$, embora cheguem perto: Para a precisão simples nós temos um coeficiente angular de $m_{ro} = 0.070 \pm 0.081$, enquanto que para a precisão dupla $m_{ro} = 0.03441 \pm 0.144$. Ou seja, nós temos um erro que aumenta próximo

a $\mathcal{O}(h^{-1/2})$, mas não exatamente com esta ordem, isto se dá pois os erros de round-off são originados de forma pseudo-aleatória, o que causa uma distribuição mais desorganizada para os pontos ao redor de nosso ajuste, podendo, por vezes, fugir completamente da nossa previsão inicial, o que pode ser observado na região com $p > 16$, para os erros de precisão simples.

Questão 2

Nós queremos calcular o período de um pêndulo simples para ângulos apreciáveis e desprezando a resistência do ar. Para isso, nós devemos calcular

$$T = 4\sqrt{\frac{l}{g}}\int_0^{\pi/2} \frac{1}{\sqrt{1 - k^2 \sin^2 \xi}} d\xi$$

onde $k \equiv \sin(\theta_0/2)$ e θ_0 é o ângulo inicial do pêndulo, em radianos. Podemos ver que o nosso problema se resume, essencialmente a calcular a integral

$$\int_0^{\pi/2} \frac{1}{\sqrt{1 - k^2 \sin^2 \xi}} d\xi$$

e para avaliá-la, nós faremos uso do método de Simpson. Para isso foi construído um programa que a resolvesse para 10 valores igualmente espaçados para $\theta_0 \in [0, \pi)$. Escolhemos de forma arbitrária $l = 1$ e usamos $g = 9.8$. Para o número de divisões trapezoidais N, foram realizados alguns testes, chegando a um valor de $N = 128$. O programa está transcrito abaixo, e abaixo dele, uma tabela com os 10 valores de θ_0, e os 10 valores de T respectivos calculados.

```
from math import sin, sqrt, pi

def k(theta_0):
    ''' Toma um valor de angulo inicial theta_0, em radianos, e determina a
        constante k = sin(theta_0/2) '''

    K = sin(theta_0/2)

    return K

def f(csi,k):
    ''' A partir de um ponto csi e uma constante k, determina o valor da
        funcao f(csi) = 1/sqrt(1 - k^2*sin^2(csi)) neste ponto '''

    func = 1/sqrt(1 - k**2 * sin(csi)**2)

    return func

def Simpson(K,a,b):
    ''' Dado uma constante de valor inicial K, um intervalo de integracao
```

```
        [a,b], realiza uma integracao numerica da funcao f(x) pelo metodo
        de Simpson, por meio da formula simplificada
        S = h/3[f_1 + f_n + 4*sum_{i pares} f_i + 2*sum_{i impares} f_i] '''

    #Iniciamos determinando o numero de intervalos N para realizar a
        integracao
    #e conjuntamente o tamanho do passo de integracao h
    N = 128
    h = (b - a)/N

    #Definimos o somatorio parcial ja calculando seus dois primeiros termos
    #nos extremos (f_1 e f_n)
    S = f(a,K) + f(b,K)

    #Iniciamos um loop para calcular os outros termos do somatorio indo de 1 a
    #N-1
    for i in range(1,N):

        #Calculamos a funcao f_{i+1} - Note que isso se faz necessario pois
        #i parte de 1, mas a nossa formula parte de i = 2
        termo = f(a + i*h, K)

        #Quando o valor de (i+1) for par, devemos multiplicar a funcao
            calculada
        #por 4, e quando for impar, devemos multiplica-lo por 2
        if i%2 != 0:

            S = S + 4*termo

        else:

            S = S + 2*termo

    #Apos somar todos os termos presentes nas chaves [] que determinam a
    #integral pelo metodo de Simpson, multiplicamos pela fator de h/3
    S = S*h/3

    return S

#Valores extremos do intervalo de integracao [a,b]
a = 0
b = pi/2

#Para calcular o periodo do pendulo necessitamos de valores para l e para g.
#l foi arbitrariamente tomado como l = 1 metro
l = 1
g = 9.8

#Definimos aqui uma variavel com o valor maximo tomado por theta_0 por
#conveniencia
theta = pi
```

```
#Devemos tomar 10 valores de theta_0 e portanto iniciamos um loop que se
    repete
#10 vezes
for i in range(10):

    #Escolhemos o valor do angulo inicial dividindo o intervalo [0,pi) em 10
    #intervalos igualmente espacados e tomamos cada um destes valores
    #individualmente
    theta_0 = (theta/10)*i

    #A partir de theta_0 determinamos a constante k
    K = k(theta_0)

    #Realizamos entao a integracao para o valor de theta_0 atual, e
        multiplicamos
    #a integral por 4*sqrt(l/g)
    T = 4*sqrt(l/g)*Simpson(K,a,b)

    #Tendo o valor do periodo, imprimimos os valores junto com o valor tomado
    #para theta_0
    print('theta_0 = %f, T = %f' %(theta_0,T))
```

```
>>> theta_0 = 0.000000, T = 2.007090
>>> theta_0 = 0.314159, T = 2.019541
>>> theta_0 = 0.628319, T = 2.057763
>>> theta_0 = 0.942478, T = 2.124541
>>> theta_0 = 1.256637, T = 2.225206
>>> theta_0 = 1.570796, T = 2.369050
>>> theta_0 = 1.884956, T = 2.572458
>>> theta_0 = 2.199115, T = 2.866702
>>> theta_0 = 2.513274, T = 3.321928
>>> theta_0 = 2.827433, T = 4.159474
```

θ_0	T
0	2.007090
0.314159	2.019541
0.628319	2.057763
0.942478	2.124541
1.256637	2.225206
1.570796	2.369050
1.884956	2.572458
2.199115	2.866702
2.513274	3.321928
2.827433	4.159474

Aumentamos então o número de valores para θ_0, utilizando agora 100 valores igualmente espaçados de $[0, \pi)$, e além disso aumentamos o número de divisões trapezoidais para $N =$

256. A partir destes novos valores, fizemos um gráfico de $T/T_{Galileu}$ em função de θ_0, colocado abaixo. Note antes, que como $T_{Galileu} = 2\pi\sqrt{l/g}$, nós simplificamos o cálculo da razão tomando

$$\frac{T}{T_{Galileu}} = \frac{4\sqrt{l/g}}{2\pi\sqrt{l/g}} \int_0^{\pi/2} \frac{1}{\sqrt{1 - k^2 \sin^2 \xi}} d\xi = \frac{2}{\pi} \int_0^{\pi/2} \frac{1}{\sqrt{1 - k^2 \sin^2 \xi}} d\xi$$

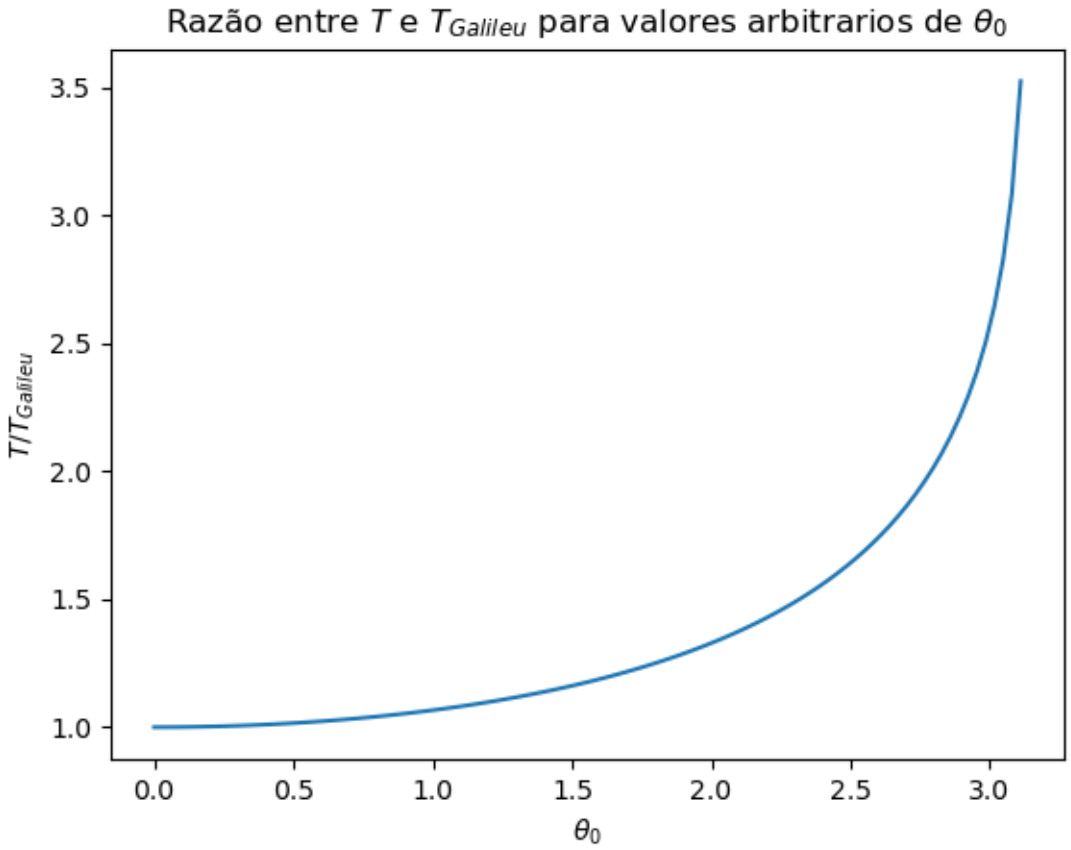

Figura 3: $T/T_{Galileu}$ em função de θ_0

Veja que para valores de $\theta_0 < 10°$, nós temos valores muito próximos de 1, como seria de se esperar, e a medida que aumentamos θ_0, o período T começa a aumentar radicalmente em relação a $T_{Galileu}$. Por fim, perceba que, quando $\theta_0 = 0$, $k = 0$ e $T = T_{Galileu}$, e de fato esta conta está certa, especialmente levando em conta de $T_{Galileu}$ independe de θ_0, embora fisicamente o que aconteceria seria que o pêndulo se manteria parado, e portanto $T = 0$.

Questão 3

Item a)

Para construir a rotina random(Z_i), nós fizemos uso de alteração de variáveis globais. Sendo assim, nosso programa se inicia com a semente (seed) dada pelo meu número USP como uma variável global $Z = 10819721$, e então a altera (globalmente) sempre que a rotina for chamada, pelo método do "linear congruential generator"(LCG), isto é, sempre que a rotina é chamada ela altera Z fazendo com que

$$Z \leftarrow (aZ + c) \bmod m = (1103515245\ Z + 12345) \bmod (2147483647)$$

Feita a alteração, a rotina gera o número "aleatório" $U = Z/m$. Observe que, do jeito como construímos o programa, não foi necessário que a rotina recebesse qualquer parâmetro como argumento. A rotina está transcrita abaixo e para os itens subsequentes desta questão, utilizaremos ela como nosso gerador de números aleatórios.

```
#Seed inicial Z_0 do nosso LCG, dada pelo meu numero USP
Z = 10819721

def random():
    ''' A partir de uma semente (seed) Z_0 = 10819721, definida globalmente,
        a funcao calcula o proximo numero Z segundo um LCG, alterando a
            variavel
        global, e por fim calcula o nosso numero 'aleatorio' no intervalo
            [0,1]
        pela divisao Z/m '''

    #Parametros do LCG
    a = 1103515245
    c = 12345
    m = 2147483647

    #Chamamos a nossa variavel global para podermos altera-la apos a chamada
    #da funcao
    global Z

    #Calculamos o valor Z_{i+1}
    Z = (a*Z+c)%m

    #Definimos o numero entre 0 e 1 dividindo Z por m
    U = Z/m

    return U
```

Item b)

Usando a rotina acima, utilizamos o método de Monte Carlo para gerar 100 valores para $0 < x < 1$ e 100 valores para $0 < y < 1$ (isto é, 100 *pontos* (x, y) dentro de um quadrado

1×1) de forma aleatória. Como resultado obtemos 17 pontos sob a curva $y = x^4$, de modo que o método de Monte Carlo nos dá um valor aproximado para a integral de

$$I \simeq \frac{\text{número de pontos abaixo}}{\text{número total de pontos}} = \frac{17}{100} = 0.17$$

Podemos também checar analiticamente que o valor esperado para ela é de

$$I = \int_0^1 x^4 dx = \left[\frac{x^5}{5}\right]_0^1 = \frac{1}{5} = 0.2$$

O que nos mostra que por mais que o resultado não seja muito preciso, já chega relativamente próximo do resultado esperado. O código que calculou esta integral pelo método de Monte Carlo está transcrito abaixo, e seguido dele um gráfico com a visualização do método

```
#Seed inicial Z_0 do nosso LCG, dada pelo meu numero USP
Z = 10819721

def random():
    ''' A partir de uma semente (seed) Z_0 = 10819721, definida globalmente,
        a funcao calcula o proximo numero Z segundo um LCG, alterando a
            variavel
        global, e por fim calcula o nosso numero 'aleatorio' no intervalo
            [0,1]
        pela divisao Z/m '''

    #Parametros do LCG
    a = 1103515245
    c = 12345
    m = 2147483647

    #Chamamos a nossa variavel global para podermos altera-la apos a chamada
    #da funcao
    global Z

    #Calculamos o valor Z_{i+1}
    Z = (a*Z+c)%m

    #Definimos o numero entre 0 e 1 dividindo Z por m
    U = Z/m

    return U

def f(x):
    ''' Dado um ponto x, calculamos o valor de y = x^4 '''

    y = x**4

    return y

def Monte_Carlo(N):
```

```
    ''' Dado um numero de pontos N, geramos N valores de x e N valores de y
        aleatoriamente, e entao avaliamos, pelo metodo de Monte Carlo, o valor
        da integral de 0 a 1 de f(x) '''

    #Comecamos com o numero de pontos embaixo da curva f(x) igual a zero
    MC = 0

    #Realizamos um loop do tamanho do numero de passos desejados
    for i in range(N):

        #Geramos dois numeros aleatorios x e y
        x = random()
        y = random()

        #Caso o valor de y esteja abaixo da funcao f(x), entao consideramos
        #este ponto para o nosso calculo, atualizando MC
        if y < f(x):

            MC = MC + 1

    #Apos obter todos os pontos abaixo de f(x), dividimos este numero pelo
    #numero total de pontos para obter o valor aproximado da integral
    I = MC/N

    return I

#Definimos quantos pontos desejamos
N = 100

#Calculamos a integral pelo metodo de Monte Carlo para os N pontos
I = Monte_Carlo(N)

#Printamos o valor obtido
print("Utilizando %i pontos, calculamos pelo metodo de Monte Carlo a integral
    de f(x) = x^4 de 0 a 1 como sendo, aproximadamente %f" %(N, I))

>>> Utilizando 100 pontos, calculamos pelo metodo de Monte Carlo a integral de
    f(x) = x^4 de 0 a 1 como sendo, aproximadamente 0.170000
```

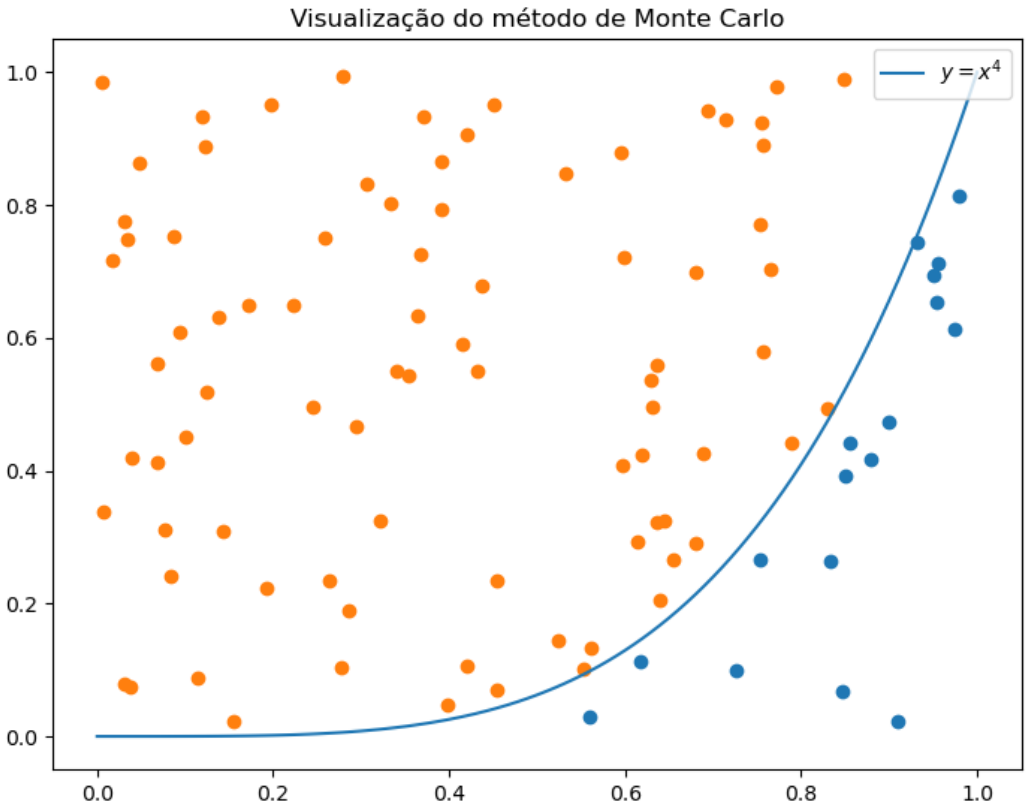

Figura 4: Visualização da integração por método de Monte Carlo

Item c)

Por fim, repetimos a integral calculada no item (b) com o método de Monte Carlo para $N_t = 2^p$ (com $p \in [1, 17]$) tentativas diferentes e com estes N_t valores de I_i nós calculamos o valor médio das integrais, I_m, seu desvio padrão, σ, e o desvio padrão da média, σ_m, dados respectivamente por

$$I_m = \frac{1}{N_t} \sum_{i=1}^{N_t} I_i$$

$$\sigma^2 = \frac{1}{N_t - 1} \sum_{i=1}^{N_t} (I_i - I_m)^2$$

$$\sigma_m = \frac{\sigma}{\sqrt{N_t}}$$

e com estes valores construímos a tabela abaixo. Note que o valor da integral será dado por $I = I_m \pm \sigma_m$.

N_t	I_m	σ	σ_m
2	0.245	0.106066017177982	0.075
4	0.1675	0.053774219349672	0.026887109674836
8	0.20625	0.041035698744247	0.014508310426393
16	0.214375	0.030103986446981	0.007525996611745
32	0.1953125	0.036542893488558	0.006459931947484
64	0.2109375	0.039869579046352	0.004983697380794
128	0.2025	0.039325013578045	0.003475872971411
256	0.1994921875	0.037793763588408	0.002362110224275
512	0.20048828125	0.038478344983335	0.001700518666659
1024	0.200048828124999	0.040162153714669	0.001255067303583
2048	0.200200195312500	0.039557999566620	0.000874116554492
4096	0.199797363281252	0.039974142471178	0.000624595976112
8192	0.199155273437517	0.039919272848983	0.000441049820805
16384	0.200081176757810	0.039966255707107	0.000312236372712
32768	0.200152893066381	0.040362721243802	0.000222974639833
65536	0.199979705810511	0.040084726208093	0.000156580961750
131072	0.200010681152286	0.039990727714121	0.000110459823247