

EXERCICIO PROGRAMA 3 - MAP2212 2024

Victor Rocha Cardoso Cruz 11223757

Larissa Aparecida Marques Pimenta Santos 12558620

1 Enunciado

Neste EP, você deve analisar experimentalmente a velocidade de convergência dos procedimentos de integração implementados no EP 2 ao substituir o gerador de números pseudo-aleatórios por um gerador quase-aleatório.

- Como um aprimoramento do EP 2, considere trocar o gerador de números pseudo-aleatórios por um gerador de números quase-aleatórios.
- Suas rotinas de integração por Monte Carlo funcionam melhor? Empiricamente, o quão mais rápidas são as novas rotinas?
- Explique cuidadosamente por que e como você fez suas análises empíricas e chegou à sua conclusão.

2 Desenvolvimento

A partir do código implementado no EP 2, para as diversas variantes do método de Monte Carlo, foi feita a substituição das funções de distribuição do `scipy.stats` (*uniform*, *beta*, *gamma* e *weibull min*), responsáveis pela geração de números pseudo-aleatórios, pelo módulo *Quasi-Monte Carlo* (`qmc`) da mesma biblioteca, capaz de gerar uma distribuição uniforme quase aleatória. Dentre os modelos de distribuições disponíveis, optou-se pela sequência de Halton, devida a sua baixa variância e elevada uniformidade.

Para simular as distribuições Beta, Gama e Weibull, através da geração quase aleatória, os valores passaram por uma transformação pela função P.P.F, que é a inversa da função P.P.F, característica de cada distribuição. Além disso, para Gama e Weibull, que geram valores maiores que um, foi feita a normalização para os limites de integração desejados, conforme já implementado no EP 2.

3 Conclusão

Os códigos do EP 2 e 3 foram submetidos a uma série de testes, promovendo os valores de variância para cada método, bem como a duração (em segundos) de toda a execução. Nota-se que ambos os geradores, pseudo e quasi, possuem uma variância baixa e são convergentes entre si, gerando uma quantidade de execuções n próximos. Portanto, os dois métodos são equivalentes, não sendo possível definir um como superior ao outro nesse quesito.

Entretanto, dentro da execução para dimensões maiores, como $n = 100000$, percebe-se uma clara vantagem dos números pseudo aleatórios, uma vez que a velocidade de execução é 3 vezes menor que o gerador quase aleatório. Em um dos testes, a implementação de números pseudo aleatórios levou 35,76 s para conclusão, enquanto o quase aleatórios demorou 233,03 s. Essa grande diferença de duração concentra-se no MMC Amostragem por Importância, especialmente para as distribuições Gamma e Weibull, elas chegam a levar 10x mais tempo para finalizar. Para os demais métodos de distribuição uniforme e Beta, há uma equivalência entre os dois, assim como se enxergou nas variâncias.

4 Código em Python

```
1  '''
2
3  C DIGO DO EP 3
4
5  '''
6  import numpy as np
7  from scipy.stats import uniform, beta, gamma, weibull_min, qmc
8  import time
9
10 # Defini o dos par metros a e b
11 a = 0.564263989 # RG
12 b = 0.51197762809 # CPF
13
14 # Fun o alvo f(x)
15 def f(x):
16     fx = np.exp(-a * x) * np.cos(b * x)
17     return fx
18
19 print("*** EP 3 INICIANDO ***")
20 print()
21 print("*** EXPERIMENTO PILOTO ***")
22 # EXPERIMENTO PILOTO #
23 # M todo Cru para defini o da integral esperada
24 # Defini o a largura da distribui o
25 n = 1000000
26 # Amostragem usando o m todo de Halton
27 halton_samples_x = qmc.Halton(d=1).random(n)
28 halton_samples_y = f(halton_samples_x)
29 # Aproxima o atrav s de Distribui o Halton
30 AreaPilot = np.mean(halton_samples_y)
31 print(f" rea Piloto {AreaPilot}")
32 print()
33
```

```

34 # Defini o da margem de erro e quantil da normal
35 epsilon = 0.05 / 100 * AreaPilot
36 z_alfa = 1.64
37
38 total_time = 0
39
40 # ***** M TODO DE MONTE CARLO CRU
    ***** #
41 print("*** M TODO DE MONTE CARLO CRU ***")
42 # Distribui o Uniforme
43 n = 100000
44 start_time = time.time()
45 halton_samples_x = qmc.Halton(d=1).random(n)
46 halton_samples_y = f(halton_samples_x)
47 # Aproxima o atravs de Distribui o Uniforme
48 area_cru = np.mean(halton_samples_y)
49 variancia = np.sum((halton_samples_y - area_cru)**2)/n
50 erro_padrao = np.sqrt(variancia)/np.sqrt(n)
51 end_time = time.time()
52 execution_time = end_time - start_time
53 total_time += execution_time
54 print(f" rea estimada: {area_cru}")
55 print(f"Vari ncia do estimador: {variancia}")
56 print(f"Erro padr o do estimador: {erro_padrao:.10f}")
57 print(f"Largura da Distr.: {n}")
58 print()
59 n = round(((z_alfa * np.sqrt(variancia))/epsilon)**2)
60 print(f"Valor de n calculado: {n}")
61 print("Tempo de execu o:", execution_time, "segundos \n")
62 print()
63
64 #
    *****
    #

```

```

65 # ***** M TODO DE MONTE CARLO 'HIT OR MISS'
    ***** #
66 print("*** M TODO DE MONTE CARLO 'HIT OR MISS' ***")
67 # Aproxima o atraves de Distribuio Uniforme
68 n = 100000
69 PontosDentro = 0
70 start_time = time.time()
71 hitmiss_samples_x = qmc.Halton(d=1).random(n)
72 hitmiss_samples_y = qmc.Halton(d=1).random(n)
73 for i in range(n):
74     xi = hitmiss_samples_x[i]
75     yi = hitmiss_samples_y[i]
76     # Checa se o ponto sorteado est abaixo da curva
77     if yi <= f(xi):
78         PontosDentro += 1
79 area_hitmiss = PontosDentro/n
80 # Variancia do estimador
81 variancia = area_hitmiss*(1 - area_hitmiss)/n
82 erro_padrao = np.sqrt(variancia)/np.sqrt(n)
83 end_time = time.time()
84 execution_time = end_time - start_time
85 total_time += execution_time
86 print(f" rea estimada: {area_hitmiss}")
87 print(f"Vari ncia do estimador: {variancia:.10f}")
88 print(f"Erro padr o do estimador: {erro_padrao:.10f}")
89 print(f"Largura da Distr.: {n}")
90 print()
91 n = round(((z_alfa * np.sqrt(variancia))/epsilon)**2)
92 print(f"Valor de n calculado: {n}")
93 print("Tempo de execu o:", execution_time, "segundos")
94 print()
95
96 #
    *****

```

```

#
97 # ***** M TODO DE MONTE CARLO AMOSTRAGEM POR
    IMPORT NCIA ***** #
98 print("*** M TODO DE MONTE CARLO AMOSTRAGEM POR IMPORT NCIA ***
    ")
99 # Distribui o Uniforme
100 n = 100000
101 pesos = []
102 pdfx = []
103 start_time = time.time()
104 uniform_samples_x = qmc.Halton(d=1).random(n)
105 xi = uniform_samples_x
106 # Fun o Densidade de probabilidade da Distribui o Uniforme
107 pdfx = uniform.pdf(xi)
108 yi = f(xi)
109 # Calculo dos pesos, como o quociente entre os valores f(xi) da
    fun o pela FDP da distribui o uniforme
110 pesos = yi/pdfx
111 area_uniform = np.mean(np.array(pesos))
112 # Variancia do estimador
113 variancia = np.sum(np.array(pdfx)*(np.array(pesos) - area_uniform
    )**2)/n
114 erro_padrao = np.sqrt(variancia)/np.sqrt(n)
115 end_time = time.time()
116 execution_time = end_time - start_time
117 total_time += execution_time
118 print(f"Uniforme: rea estimada: {area_uniform}")
119 print(f"Uniforme: Vari ncia do estimador: {variancia:.10f}")
120 print(f"Uniforme: Erro padr o do estimador: {erro_padrao:.10f}")
121 print(f"Uniforme: Quantidade de itera es: {n}")
122 print()
123 n = round(((z_alfa * np.sqrt(variancia))/epsilon)**2)
124 print(f"Valor de n calculado: {n}")
125 print("Tempo de execu o:", execution_time, "segundos")

```

```

126 print()
127
128 # Distribui o Beta
129 # Implementa o método de Halton para gerar amostras quase
    aleat rias
130 pesos = []
131 pdfx = []
132 n = 100000
133 start_time = time.time()
134 beta_samples_x = qmc.Halton(d=1).random(n)
135 xi = beta_samples_x
136 # Transforma o da vari vel quase-aleat ria uniforme na
    distribui o Beta
137 xi = beta.ppf(xi, a = 1, b = 1)
138 # Obten o da fun o densidade de probabilidade, para a
    distribui o Beta
139 pdfx = beta.pdf(xi, a = 1, b = 1)
140 yi = f(xi)
141 pesos = yi/pdfx
142 area_beta = np.mean(pesos)
143 # Variancia do estimador
144 variancia = np.sum(np.array(pdfx)*(np.array(pesos) - area_beta)
    **2)/n
145 erro_padrao = np.sqrt(variancia)/np.sqrt(n)
146 end_time = time.time()
147 execution_time = end_time - start_time
148 total_time += execution_time
149 print(f"Beta: rea estimada: {area_beta}")
150 print(f"Beta: Vari ncia do estimador: {variancia:.10f}")
151 print(f"Beta: Erro padr o do estimador: {erro_padrao:.10f}")
152 print(f"Beta: Quantidade de itera es: {n}")
153 print()
154 n = round(((z_alfa * np.sqrt(variancia))/epsilon)**2)
155 print(f"Valor de n calculado: {n}")

```

```

156 print("Tempo de execu    o:", execution_time, "segundos")
157 print()
158
159
160 # Distribui    o Gamma
161 # Normaliza    o da distribui    o Gamma a partir da fun    o
    cumulativa nos pontos de interesse
162 cdf_lower = gamma.cdf(0, a=1, scale=2)
163 cdf_upper = gamma.cdf(1, a=1, scale=2)
164 const_normalizacao = 1 / (cdf_upper - cdf_lower)
165 # Fun    o que gera valores quase aleat rios para a
    distribui    o Gama
166 def generate_n_gamma_samples(num_samples):
167     samples_no_intervalo = []
168     while len(samples_no_intervalo) < num_samples:
169         # Gerar amostras aleat rias pela distribui    o Gama
170         gamma_samples_x = qmc.Halton(d=1).random(1)
171         gamma_samples_x = gamma.ppf(gamma_samples_x, a= 1, scale
            = 2)
172         # Checa se a amostra est    dentro do intervalo
173         if 0 <= gamma_samples_x <= 1:
174             samples_no_intervalo.append(gamma_samples_x)
175     return np.array(samples_no_intervalo)
176
177 pesos = []
178 pdfx = []
179 n = 100000
180 start_time = time.time()
181 gamma_samples_x = generate_n_gamma_samples(n)
182 xi = gamma_samples_x
183 # Obten    o da fun    o densidade de probabilidade, para a
    distribui    o Gamma, e normalizada    0 e 1
184 pdf_value = gamma.pdf(xi, a = 1, scale = 2)
185 pdfx = pdf_value * const_normalizacao

```



```

186 yi = f(xi)
187 pesos = yi/pdfx
188 area_gama = np.mean(pesos)
189 # Variância do estimador
190 variancia = np.sum(np.array(pdfx)*(np.array(pesos) - area_gama)
    **2)/n
191 erro_padrao = np.sqrt(variancia)/np.sqrt(n)
192 end_time = time.time()
193 execution_time = end_time - start_time
194 total_time += execution_time
195 print(f"Gama: rea estimada: {area_gama}")
196 print(f"Gama: Variância do estimador: {variancia}")
197 print(f"Gama: Erro padrão do estimador: {erro_padrao:.10f}")
198 print(f"Gama: Quantidade de iterações: {n}")
199 print()
200 n = round(((z_alfa * np.sqrt(variancia))/epsilon)**2)
201 print(f"Valor de n calculado: {n}")
202 print("Tempo de execução:", execution_time, "segundos")
203 print()
204
205 # Distribuição Weibull
206 # Normaliza a distribuição Weibull a partir da função
    cumulativa nos pontos de interesse
207 cdf_lower = weibull_min.cdf(0, c=1)
208 cdf_upper = weibull_min.cdf(1, c=1)
209 const_normalizacao = 1 / (cdf_upper - cdf_lower)
210 # Função que gera valores quase aleatórios para a
    distribuição Weibull
211 def generate_n_weibull_samples(num_samples):
212     samples_no_intervalo = []
213     while len(samples_no_intervalo) < num_samples:
214         # Gerar amostras aleatórias pela distribuição Gama
215         weibull_samples_x = qmc.Halton(d=1).random(1)
216         weibull_samples_x = weibull_min.ppf(weibull_samples_x, c

```

```

        =1)
217     # Checa se a amostra est  dentro do intervalo
218     if 0 <= weibull_samples_x <= 1:
219         samples_no_intervalo.append(weibull_samples_x)
220     return np.array(samples_no_intervalo)
221
222 pesos = []
223 pdfx = []
224 n = 100000
225 start_time = time.time()
226 weibull_samples_x = generate_n_weibull_samples(n)
227 xi = weibull_samples_x
228 # Obten  o da fun  o densidade de probabilidade, para a
    distribui  o Weibull, e normalizada  0 e 1
229 pdf_value = weibull_min.pdf(xi, c=1)
230 pdfx = pdf_value * const_normalizacao
231 yi = f(xi)
232 pesos = yi/pdfx
233 area_weibull = np.mean(pesos)
234 # Variancia do estimador
235 variancia = np.sum(np.array(pdfx)*(np.array(pesos) - area_weibull
    )**2)/n
236 erro_padrao = np.sqrt(variancia)/np.sqrt(n)
237 end_time = time.time()
238 execution_time = end_time - start_time
239 total_time += execution_time
240 print(f"Weibull: rea  estimada: {area_weibull}")
241 print(f"Weibull: Vari ncia do estimador: {variancia:.10f}")
242 print(f"Weibull: Erro padr o do estimador: {erro_padrao:.10f}")
243 print(f"Weibull: Quantidade de itera  es: {n}")
244 print()
245 n = round(((z_alfa * np.sqrt(variancia))/epsilon)**2)
246 print(f"Valor de n calculado: {n}")
247 print("Tempo de execu  o:", execution_time, "segundos")

```

```

248 print()
249
250 #
    *****
    #
251 # ***** M TODO DE MONTE CARLO VARI VEIS DE
    CONTROLE ***** #
252 print("*** M TODO DE MONTE CARLO VARI VEIS DE CONTROLE ***")
253 # Defini o dos limites de integra o para a fun o f e
254 limInfer, limSuper= 0, 1
255 # Fun o polinomial que aproxima-se curva da fun o f(
    x), encontrada a partir dos pontos (1, f(1)) e (0, f(0))
256 def phi(x):
257     phix = ((f(1)-f(0))/(1 - 0))*x + 1
258     #phix = 1 - 0.564263989*x + 0.02814*x**(2) + 0.04400*x**(3)-
        0.01377*x**(4)
259     return phix
260 # Fun o primitiva de
261 def phiPrim(x):
262     phiPrim = x + ((f(1)-f(0))/(1 - 0))*x**(2)/2
263     #phiPrim = x - 0.564263989*x**(2)/2 + 0.02814*x**(3)/3 +
        0.04400*x**(4)/4 - 0.01377*x**(5)/5
264     return phiPrim
265
266 termo = 0
267 fxn, phixn = [], []
268 n = 100000
269 start_time = time.time()
270 controlv_samples_x = qmc.Halton(d=1).random(n)
271 #xi = controlv_samples_x[i]
272 # Armezando os valores de f(xi) e g(xi) para o c lculo final da
    vari ncia
273 fxn = f(controlv_samples_x)
274 phixn = phi(controlv_samples_x)

```

```

275 # Termo do somatório para cálculo de gama chapéu
276 termo = np.sum(f(controlv_samples_x) - phi(controlv_samples_x) +
    (phiPrim(limSuper)-phiPrim(limInfer)))
277 area_control = termo/n
278 var_f = np.var(np.array(fxn))
279 var_phi = np.var(np.array(phixn))
280 correlacao = np.cov(fxn, phixn, rowvar=False)[0,1]
281 variancia = (1/n)*(var_f + var_phi - 2*correlacao*np.sqrt(var_f)*
    np.sqrt(var_phi))
282 erro_padrao = np.sqrt(variancia)/np.sqrt(n)
283 end_time = time.time()
284 execution_time = end_time - start_time
285 total_time += execution_time
286 print(f"Área estimada: {area_control}")
287 print(f"Variancia do estimador: {variancia:.10f}")
288 print(f"Erro padrão do estimador: {erro_padrao:.10f}")
289 print(f"Quantidade de iterações: {n}")
290 print()
291 n = round(((z_alfa * np.sqrt(variancia))/epsilon)**2)
292 print(f"Valor de n calculado: {n}")
293 print("Tempo de execução:", execution_time, "segundos")
294 print()
295 print("Tempo total de execução:", total_time, "segundos")

```