

移動情報ネットワーク特論

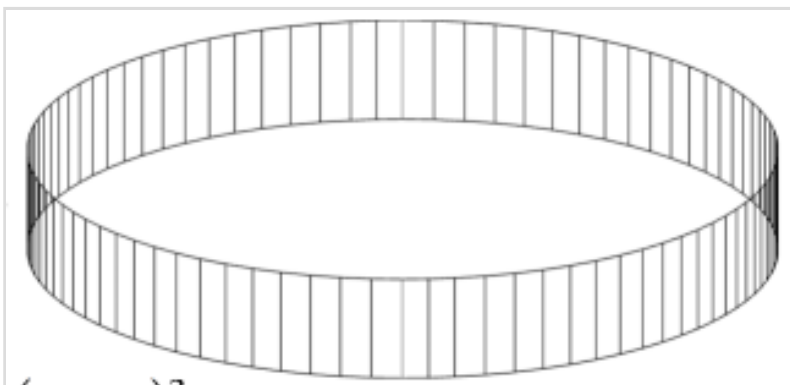
レポート課題2

電気情報工学専攻 情報工学コース
F20C004C 太田剛史

移動がある場合のM/M/S/Sのシミュレーションの作成

以下に「移動が無い場合のM/M/S/Sのシミュレーション」を実行するためのスクリプトを示す。

今回利用した移動のモデルは、1次元のセル構造を用いており、中でも始点と終端をつなげた以下の図に示すような、環状型のセル構造となっている。



使用した言語は Python 3.6.9 である。

プログラムの中身は `MobileToken` クラス, `ServiceAreaManager` クラス, `Simulator` クラス の3つのクラスと, それらにパラメータを与え実行した `main` 関数 がある。

シミュレーションの結果はJSONファイルに出力されるようになっている。

```

1  import numpy as np
2  from typing import List, Tuple
3  import pandas as pd
4  import random
5  import json
6
7
8  class MobileToken:
9      """
10     高速道路においてサービスを受ける移動体端末
11     """
12
13     def __init__(self, belonging_cell_idx: int, cell_length: float, service_time: float, v: float = 10.0):
14         """
15         Params
16         -----
17         belonging_cell_idx : int
18             端末が所属している初期のセル
19         cell_length : float
20             各セルの長さ
21         """
22         # 端末の移動速度
23         self._v: float = v
24
25         # 端末のサービス利用残り時間
26         self._remain_service_time: float = service_time
27
28         # 端末が所属しているセルの番号
29         self.belonging_cell_idx: int = belonging_cell_idx
30
31         # 1つのセルの長さ. 各セルにおいて固定長.
32         self._cell_length: float = cell_length
33
34         # ハンドオフまでの時間, 初期値は [0.0, 最大時間) の範囲で値をとる
35         self._time_to_handoff: float = random.random() * self._cell_length / self._v

```

```

36
37 def handoff(self, cell_num: int):
38     """
39     ハンドオフ時の処理を行います。
40     1. ハンドオフまでの時間のリセット
41     2. 次のセルへ移動
42
43     Params
44     -----
45     cell_num : int
46         サービスエリアにおける総セル数
47     """
48     self._time_to_handoff = self._cell_length / self._v
49     self.belonging_cell_idx = (self.belonging_cell_idx + 1) % cell_num
50
51 def get_next_event(self) -> Tuple[str, float]:
52     """
53     この端末における次に発生するイベントの情報を取得します。
54
55     Returns
56     -----
57     event_name : str
58         次に発生するイベント名
59     time_to_next_event : float
60         次に発生するイベントまでの時間
61     """
62     if self._time_to_handoff >= self._remain_service_time:
63         return "close", self._remain_service_time
64     else:
65         return "handoff", self._time_to_handoff
66
67 def passage(self, passage_time: float):
68     """
69     時間経過に対して、内部パラメータの適用を行います。
70
71     Params
72     -----
73     passage_time: float
74         経過時間
75     """
76     self._remain_service_time -= passage_time
77     self._time_to_handoff -= passage_time
78
79
80 class ServiceAreaManager:
81     """
82     リング状サービスエリアの管理を行うクラス
83     """
84
85     def __init__(self, cap_size: int, cell_num: int, cell_length: float):
86         """
87         Params
88         -----
89         cap_size : int
90             各セルにおける許容量
91         cell_num : int
92             サービスエリアの全体のセル総数
93         cell_length : float
94             サービスエリアの1つ1つのセルの長さ
95         """
96         # サービスエリアの空き具合
97         self._usage_conditions: List[int] = [cap_size] * cell_num
98
99         # サービスエリアの全体のセル総数
100        self._cell_num = cell_num
101
102        # サービスを受けている端末を格納する配列
103        self._in_service: List[MobileToken] = []
104
105        # 各セルの長さ(距離)
106        self._cell_length = cell_length
107
108    def _sort(self):
109        """
110        イベントが来る順に配列をソートする
111        """

```

```

112     self._in_service.sort(key=lambda x: x.get_next_event()[1])
113
114     def get_first_token(self) -> MobileToken:
115         """
116         サービス享受中のトークンの内、もっともイベント発生が速いトークンを取得します
117         (提供中のサービスが無い場合はValueErrorが発生)
118
119         Return
120         -----
121         first_token : MobileToken
122             サービス内でもっとも直前にイベントが発生するMobileTokenインスタンス
123         """
124         if len(self._in_service) == 0:
125             raise ValueError("提供中のサービスが存在しません")
126         return self._in_service[0]
127
128     def get_next_event(self) -> Tuple[str, float]:
129         """
130         サービス中のトークン群において、一番先に発生するイベントの情報を返します。
131
132         Returns
133         -----
134         event_name : str
135             発生するイベント名
136         time_to_event_occurrence : float
137             そのイベントが発生するまでの時間
138         """
139         if len(self._in_service) == 0:
140             return "NO_TOKEN", np.inf
141         return self._in_service[0].get_next_event()
142
143     def call(self, service_time: float) -> bool:
144         """
145         サービスエリアにおける呼の生起に対する処理を行います。
146
147         Param
148         -----
149         service_time : float
150             その呼のサービス享受時間
151
152         Return
153         -----
154         is_successfull : bool
155             呼損ならFalse
156         """
157         target_cell_idx = self._get_random_cell_idx()
158         if self._allocation(target_cell_idx):
159             self._in_service.append(MobileToken(target_cell_idx, self._cell_length, service_time))
160             self._sort()
161             return True
162         else:
163             return False
164
165     def close(self):
166         """
167         サービスエリアにおける呼の終了に対する処理を行います。
168         """
169         self._release(self._in_service[0].belonging_cell_idx)
170         self._in_service.pop(0)
171
172     def handoff(self) -> bool:
173         """
174         サービスエリアにおけるハンドオフに対する処理を行います。
175
176         Return
177         -----
178         is_successfull : bool
179             呼損ならFalse
180         """
181         target_cell_idx = self._in_service[0].belonging_cell_idx
182         self._release(target_cell_idx)
183
184         target_cell_idx = (target_cell_idx + 1) % self._cell_num
185         if self._allocation(target_cell_idx): # handoff成功
186             self._in_service[0].handoff(self._cell_num)
187             self._sort()

```

```

188         return True
189     else: # handoff失敗
190         self._in_service.pop(0)
191         return False
192
193     def advance_time(self, time: float) -> None:
194         """
195         サービスエリア全体の時間を進めます。
196
197         Param
198         -----
199         time : float
200             経過する時間
201         """
202         for idx in range(len(self._in_service)):
203             self._in_service[idx].passage(time)
204
205     def _get_random_cell_idx(self) -> int:
206         """
207         一様にランダムで、セル番号を返します。
208
209         Return
210         -----
211         cell_idx : int
212             ランダムなセル番号
213         """
214         return random.randint(0, self._cell_num-1)
215
216     def _allocation(self, target_cell_idx: int) -> bool:
217         """
218         Param
219         -----
220         target_cell_idx : int
221             割り当て対象セルのインデックス
222
223         Return
224         -----
225         succeed : bool
226             対象セルにおけるサービスの割当てが成功した場合->True
227         """
228         if self._usage_conditions[target_cell_idx] > 0:
229             self._usage_conditions[target_cell_idx] -= 1
230             return True
231         else:
232             return False
233
234     def _release(self, target_cell_idx: int):
235         """
236         Param
237         -----
238         target_cell_idx : int
239             解放対象セルのインデックス
240         """
241         self._usage_conditions[target_cell_idx] += 1
242
243
244     class Simulator:
245         """
246         シミュレーションを行うためのクラス
247         """
248
249     def __init__(self, prob_of_reach: float, ave_service_time: float, capacity: int = 5, cell_num: int = 5, cell_length: int = 10):
250         """
251         Params
252         -----
253         prob_of_reach: float
254             シミュレーションにおける到着率
255         ave_service_time: float
256             シミュレーションにおける平均サービス時間
257         capacity: int (default=5)
258             システムの容量
259         cell_num: int (default=5)
260             サービスエリアにおけるセル数
261         """
262         self._lam: float = prob_of_reach
263         self._mu: float = 1.0 / ave_service_time

```

```

264     self._traffic_intensity: float = self._lam / self._mu
265
266     self.capacity: int = capacity
267
268     self._in_service: List[MobileToken] = []
269
270     self.sa_manager: ServiceAreaManager = ServiceAreaManager(
271         cap_size=capacity,
272         cell_num=cell_num,
273         cell_length=cell_length
274     )
275
276     def _get_service_time(self) -> float:
277         """
278         サービス適用時間を計算し取得します
279         """
280         return float(np.random.exponential(1.0 / self._mu, size=1))
281
282     def _get_start_time_remaining(self) -> float:
283         """
284         サービス開始までの時間を算出し取得します
285         """
286         return float(np.random.exponential(1.0 / self._lam, size=1))
287
288     def run(self, stop_all_call: int) -> Tuple[int, int, int, int]:
289         call_block_num: int = 0
290         call_num: int = 0
291
292         handoff_block_num: int = 0
293         handoff_num: int = 0
294
295         time_to_call: float = self._get_start_time_remaining()
296
297         while True:
298             if call_num >= stop_all_call:
299                 break
300
301             event_name, time_to_next_event = self.sa_manager.get_next_event()
302
303             if time_to_call < time_to_next_event:
304                 event_name = "call"
305                 time_to_next_event = time_to_call
306             elif time_to_next_event < time_to_call:
307                 pass
308             else:
309                 raise NotImplementedError("生起と終了が同時に発生しました")
310
311             self.sa_manager.advance_time(time_to_next_event)
312             time_to_call -= time_to_next_event
313
314             if event_name == "call":
315                 call_num += 1
316                 if not self.sa_manager.call(self._get_service_time()):
317                     call_block_num += 1
318                 time_to_call = self._get_start_time_remaining()
319             elif event_name == "close":
320                 self.sa_manager.close()
321             elif event_name == "handoff":
322                 call_num += 1 # ハンドオフの流入時に呼の生起をカウントするかどうか
323                 handoff_num += 1
324                 if not self.sa_manager.handoff():
325                     call_block_num += 1
326                 handoff_block_num += 1
327             else:
328                 raise ValueError(f"予期せぬイベント'{event_name}'が発生しました")
329
330             return call_num, call_block_num, handoff_num, handoff_block_num
331
332     def get_traffic_intensity(self):
333         return self._traffic_intensity
334
335
336 def main(cell_length):
337     prob_of_reach_list = np.logspace(0.1, 9, 100, base=2)
338     ave_service_time = 1
339     capacity = 3

```

```

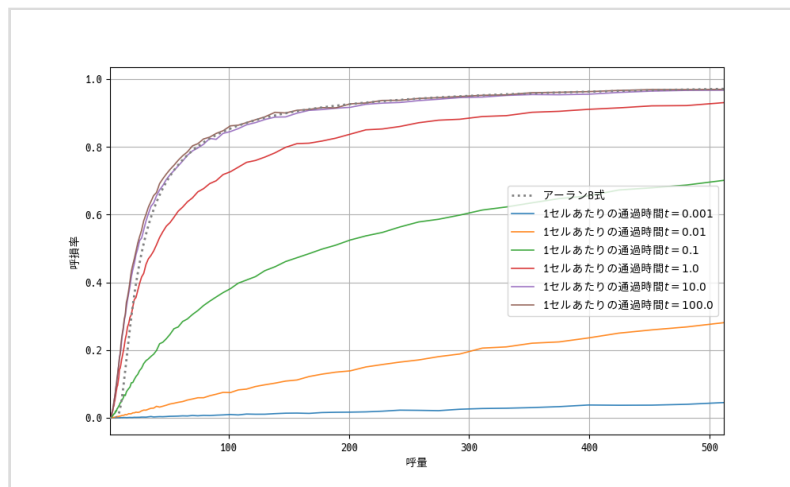
340 cell_num = 5
341 stop_all_call = 20000
342
343 # cell_length = 9
344
345 output_list = [{}]* len(prob_of_reach_list)
346
347 for i, prob_of_reach in enumerate(prob_of_reach_list):
348     print(f"@ {i:04d} : {prob_of_reach}")
349
350     sim = Simulator(
351         prob_of_reach=prob_of_reach,
352         ave_service_time=ave_service_time,
353         capacity=capacity,
354         cell_num=cell_num,
355         cell_length=cell_length
356     )
357
358     traffic_intensity = sim.get_traffic_intensity()
359     print("---- "*3, "params", "---- "*3)
360     print(f"到着率(λ) = {prob_of_reach}")
361     print(f"保留時間(1/μ) = {ave_service_time}")
362     print(f"呼量(a) = {traffic_intensity}")
363     print()
364
365     call_num, call_block_num, handoff_num, handoff_block_num = sim.run(
366         stop_all_call=stop_all_call)
367     print("---- "*3, "result", "---- "*3)
368     print(f"全呼 = {call_num}")
369     print(f"呼損 = {call_block_num}")
370     print(call_block_num)
371     print(call_num)
372     print(f"呼損率 = {call_block_num / call_num}")
373     print("\n\n")
374
375     output_list[i] = {
376         "prob_of_reach": prob_of_reach,
377         "ave_service_time": ave_service_time,
378         "traffic_intensity": traffic_intensity,
379         "capacity": capacity,
380         "call_num": call_num,
381         "call_block_num": call_block_num,
382         "handoff_num": handoff_num,
383         "handoff_block_num": handoff_block_num,
384         "block_rate": call_block_num / call_num
385     }
386
387     with open(f"./output/sim_ave={ave_service_time}_cap={capacity}_cell-len={cell_length}.json",
388             mode="w") as f:
389         json.dump({
390             "segment_time": cell_length / 10.0,
391             "output": output_list
392         }, f, indent=2)
393
394
395 if __name__ == "__main__":
396     # main()
397
398     # for i in range(10, 100, 10):
399     #     main(i)
400
401     prob_of_reach_list = np.logspace(0.1, 9, 100, base=2)
402     for i in [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000]:
403         main(i)
404

```

結果

上記のスクリプトを用いて、横軸(対数軸)を呼量、縦軸(線形軸)を呼損率としたグラフを作成した。

各変数について、どれが変数で、どれが定数か



考察

1セルあたりの通過時刻における比較

なぜこの比較か：「平均サービス提供時間」と「ハンドオフ発生までの時間」の関係によって呼損率が変化するかをみる→「平均サービス提供時間」を固定し、「ハンドオフ発生までの時間」を変えていく。「ハンドオフ発生までの時間」とは、大きく見れば「1セルを通過するのにかかる時間」と言い換えることができる。

前提として、移動体の速度がすべて一定であるため、ハンドオフしても基本的には呼損は発生しない。(具体的な数値やグラフ、説明の図があれば提示)

「1セルを通過するのにかかる時間」が小さくなれば、全呼量に対してハンドオフによる呼の割合が大きくなる。つまり先の前提と合わせて、呼損の発生が少なくなる。その逆もまた然りである。

アールンB式との比較

今回、アールンB式(キャパ=15)で示した、→ 大まかに見ればシステム全体でキャパが15だとみなせるため。(詳細には違うが、これもなにかあれば示す)

「1セルを通過するのにかかる時間」が大きくなると、アールンB式に漸近するが、呼料が低いと漸近していないことが見られる。これは…？
また漸近する理由は先の節でも説明したとおり、「1セルを通過するのにかかる時間」が大きくなれば、全呼量に対してハンドオフによる呼の割合がなくなっていく。つまり同じキャパシティを持つ移動を伴わないシステムと同じ状況になるためである。

作成した図を下記に示す。図中において、実線がシミュレーションによる結果で、破線がアールンB式による理論計算による結果である。また、今回はチャンネル数 $S \in \{2, 4, 6, 8, 10\}$ の場合でそれぞれ計算し、色別で示した。

図1 呼量と呼損率の関係

各チャンネル数において、シミュレーションにおける結果とアールンB式の値がほぼ一致しているのがわかる。
またいずれの場合においても、呼量が増えるに従って呼損率が増えていき、呼損率が1に漸近していった様子が見られる。