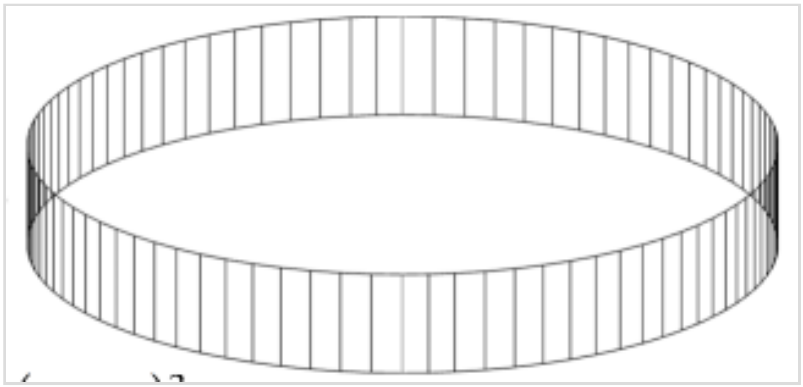
移動情報ネットワーク特論
レポート課題2

電気情報工学専攻 情報工学コース
F20C004C 太田剛史

目次

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.

また以下にシミュレーションのスクリプトを記すが, Github上にシミュレーションのスクリプトと描画のためのスクリプト, レポート作成に利用したmarkdownなどを載せてあるため, ネットワーク環境がある場合は下記のURLを参照していただきたい.

> https://github.com/haru1843/mobile_info_network_rep02

```python
import numpy as np
from typing import List, Tuple
import pandas as pd
import random
import json


class MobileToken:
    """
    高速道路においてサービスを受ける移動体端末
    """

```

```
    def __init__(self, belonging_cell_idx: int, cell_length: float, service_time: float, v: float = 10.0):
        """
        Params
        ------------------------------
        belonging_cell_idx : int
            端末が所属している初期のセル
        cell_length : float
            各セルの長さ
        """
        # 端末の移動速度
        self._v: float = v

        # 端末のサービス利用残り時間
        self._remain_service_time: float = service_time

        # 端末が所属しているセルの番号
        self.belonging_cell_idx: int = belonging_cell_idx

        # 1つのセルの長さ．各セルにおいて固定長．
        self._cell_length: float = cell_length

        # 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
        self._time_to_handoff: float = random.random() * self._cell_length / self._v

    def handoff(self, cell_num: int):
        """
        ハンドオフ時の処理を行います．
            1. ハンドオフまでの時間のリセット
            2. 次のセルへ移動

        Params
        ------------------------------
        cell_num : int
            サービスエリアにおける総セル数
        """
        self._time_to_handoff = self._cell_length / self._v
        self.belonging_cell_idx = (self.belonging_cell_idx + 1) % cell_num

    def get_next_event(self) -> Tuple[str, float]:
        """
        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．

        Returns
        ------------------------------
        event_name : str
            次に発生するイベント名
        time_to_next_event : float
            次に発生するイベントまでの時間
        """
        if self._time_to_handoff >= self._remain_service_time:
            return "close", self._remain_service_time
        else:
            return "handoff", self._time_to_handoff

    def passage(self, passage_time: float):
        """
        時間経過に対して，内部パラメータの適用を行います．

        Params
        ------------------------------
        passage_time: float
            経過時間
        """
        self._remain_service_time -= passage_time
        self._time_to_handoff -= passage_time


class ServiceAreaManager:
    """
    リング状サービスエリアの管理を行うクラス
    """

    def __init__(self, cap_size: int, cell_num: int, cell_length: float):
        """
        Params
        ------------------------------
```

```
        cap_size : int
            各セルにおける許容量
        cell_num : int
            サービスエリアの全体のセル総数
        cell_length : float
            サービスエリアの1つ1つのセルの長さ
        """
        # サービスエリアの空き具合
        self._usage_conditions: List[int] = [cap_size] * cell_num

        # サービスエリアの全体のセル総数
        self._cell_num = cell_num

        # サービスを受けている端末を格納する配列
        self._in_service: List[MobileToken] = []

        # 各セルの長さ(距離)
        self._cell_length = cell_length

    def _sort(self):
        """
        イベントが来る順に配列をソートする
        """
        self._in_service.sort(key=lambda x: x.get_next_event()[1])

    def get_first_token(self) -> MobileToken:
        """
        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
        (提供中のサービスが無い場合はValueErrorが発生)

        Return
        ----------------------------------
        first_token : MobileToken
            サービス内でもっとも直近にイベントが発生するMobileTokenインスタンス
        """
        if len(self._in_service) == 0:
            raise ValueError("提供中のサービスが存在しません")
        return self._in_service[0]

    def get_next_event(self) -> Tuple[str, float]:
        """
        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．

        Returns
        --------------------------------
        event_name : str
            発生するイベント名
        time_to_event_occurrence : float
            そのイベントが発生するまでの時間
        """
        if len(self._in_service) == 0:
            return "NO_TOKEN", np.inf
        return self._in_service[0].get_next_event()

    def call(self, service_time: float) -> bool:
        """
        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．

        Param
        ------------------------------------
        service_time : float
            その呼のサービス享受時間

        Return
        --------------------------------
        is_successfull : bool
            呼損ならFalse
        """
        target_cell_idx = self._get_random_cell_idx()
        if self._allocation(target_cell_idx):
            self._in_service.append(MobileToken(target_cell_idx, self._cell_length, service_time))
            self._sort()
            return True
        else:
            return False

```

```
    def close(self):
        """
        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．
        """
        self._release(self._in_service[0].belonging_cell_idx)
        self._in_service.pop(0)

    def handoff(self) -> bool:
        """
        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．

        Return
        ---------------------------------
        is_successfull : bool
            呼損ならFalse
        """
        target_cell_idx = self._in_service[0].belonging_cell_idx
        self._release(target_cell_idx)

        target_cell_idx = (target_cell_idx + 1) % self._cell_num
        if self._allocation(target_cell_idx):  # handoff成功
            self._in_service[0].handoff(self._cell_num)
            self._sort()
            return True
        else:  # handoff失敗
            self._in_service.pop(0)
            return False

    def advance_time(self, time: float) -> None:
        """
        サービスエリア全体の時間を進めます．

        Param
        ---------------------------------
        time : float
            経過する時間
        """
        for idx in range(len(self._in_service)):
            self._in_service[idx].passage(time)

    def _get_random_cell_idx(self) -> int:
        """
        一様にランダムで，セル番号を返します．

        Return
        ----------------------------------
        cell_idx : int
            ランダムなセル番号
        """
        return random.randint(0, self._cell_num-1)

    def _allocation(self, target_cell_idx: int) -> bool:
        """
        Param
        -----------------------------
        target_cell_idx : int
            割り当て対象セルのインデックス

        Return
        -----------------------------
        succeed : bool
            対象セルにおけるサービスの割当が成功した場合->True
        """
        if self._usage_conditions[target_cell_idx] > 0:
            self._usage_conditions[target_cell_idx] -= 1
            return True
        else:
            return False

    def _release(self, target_cell_idx: int):
        """
        Param
        -----------------------------
        target_cell_idx : int
            解放対象セルのインデックス
        """
```

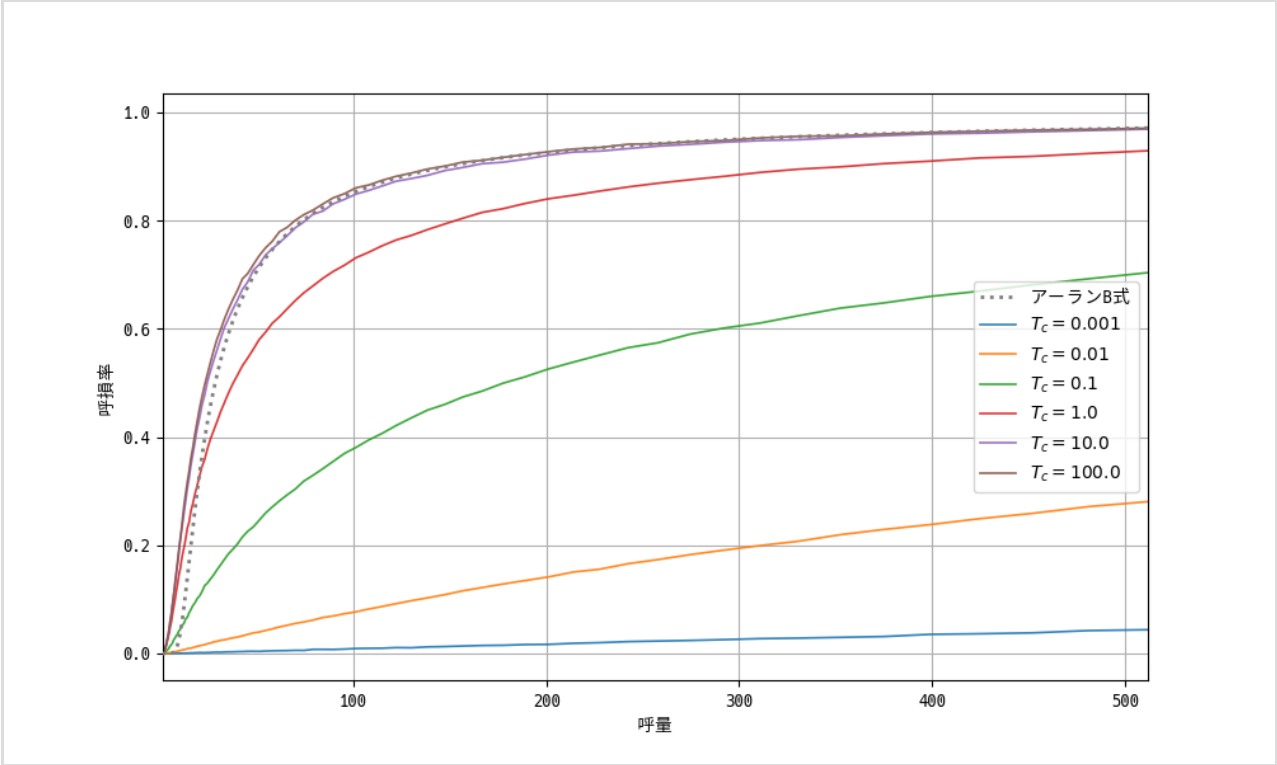
```
241         self._usage_conditions[target_cell_idx] += 1
242
243
244 class Simulator:
245     """
246     シミュレーションを行うためのクラス
247     """
248
249     def __init__(self, prob_of_reach: float, ave_service_time: float, capacity: int = 5, cell_num: int = 5, cell_length: int = 10):
250         """
251         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()):
```

```
                    call_block_num += 1
                time_to_call = self._get_start_time_remaining()
            elif event_name == "close":
                self.sa_manager.close()
            elif event_name == "handoff":
                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
                handoff_num += 1
                if not self.sa_manager.handoff():
                    call_block_num += 1
                    handoff_block_num += 1
            else:
                raise ValueError(f"予期せぬイベント'{event_name}'が発生しました")

        return call_num, call_block_num, handoff_num, handoff_block_num

    def get_traffic_intensity(self):
        return self._traffic_intensity


def main(cell_length):
    prob_of_reach_list = np.logspace(0.1, 9, 100, base=2)
    ave_service_time = 1
    capacity = 3
    cell_num = 5
    stop_all_call = 100000

    # cell_length = 9

    output_list = [{}] * len(prob_of_reach_list)

    for i, prob_of_reach in enumerate(prob_of_reach_list):
        print(f"@ {i:04d} : {prob_of_reach}")

        sim = Simulator(
            prob_of_reach=prob_of_reach,
            ave_service_time=ave_service_time,
            capacity=capacity,
            cell_num=cell_num,
            cell_length=cell_length
        )

        traffic_intensity = sim.get_traffic_intensity()
        print("---- "*3, "params", "---- "*3)
        print(f"到着率(λ) = {prob_of_reach}")
        print(f"保留時間(1/μ) = {ave_service_time}")
        print(f"呼量(a) = {traffic_intensity}")
        print()

        call_num, call_block_num, handoff_num, handoff_block_num = sim.run(
            stop_all_call=stop_all_call)
        print("---- "*3, "result", "---- "*3)
        print(f"全呼 = {call_num}")
        print(f"呼損 = {call_block_num}")
        print(call_block_num)
        print(call_num)
        print(f"呼損率 = {call_block_num / call_num}")
        print("\n\n")

        output_list[i] = {
            "prob_of_reach": prob_of_reach,
            "ave_service_time": ave_service_time,
            "traffic_intensity": traffic_intensity,
            "capacity": capacity,
            "call_num": call_num,
            "call_block_num": call_block_num,
            "handoff_num": handoff_num,
            "handoff_block_num": handoff_block_num,
            "block_rate": call_block_num / call_num
        }

    with open(f"./output/sim_ave={ave_service_time}_cap={capacity}_cell-len={cell_length}.json",
              mode="w") as f:
        json.dump({
            "segment_time": cell_length / 10.0,
            "output": output_list
        }, f, indent=2)
```

```
393
394
395 if __name__ == "__main__":
396     for cell_length in [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000]:
397         main(cell_length)
398
```

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.
平均サービス提供時間を `1` として, それに対する「1セルを通過するのにかかる時間」に注目し, グラフを作成した(図中実線). また今回のモデルでは, セルの数を `5` つ, 各セル毎の容量を `3` つとしている.
またグラフには容量が `15` の時のアーランB式の結果も載せている(図中破線).



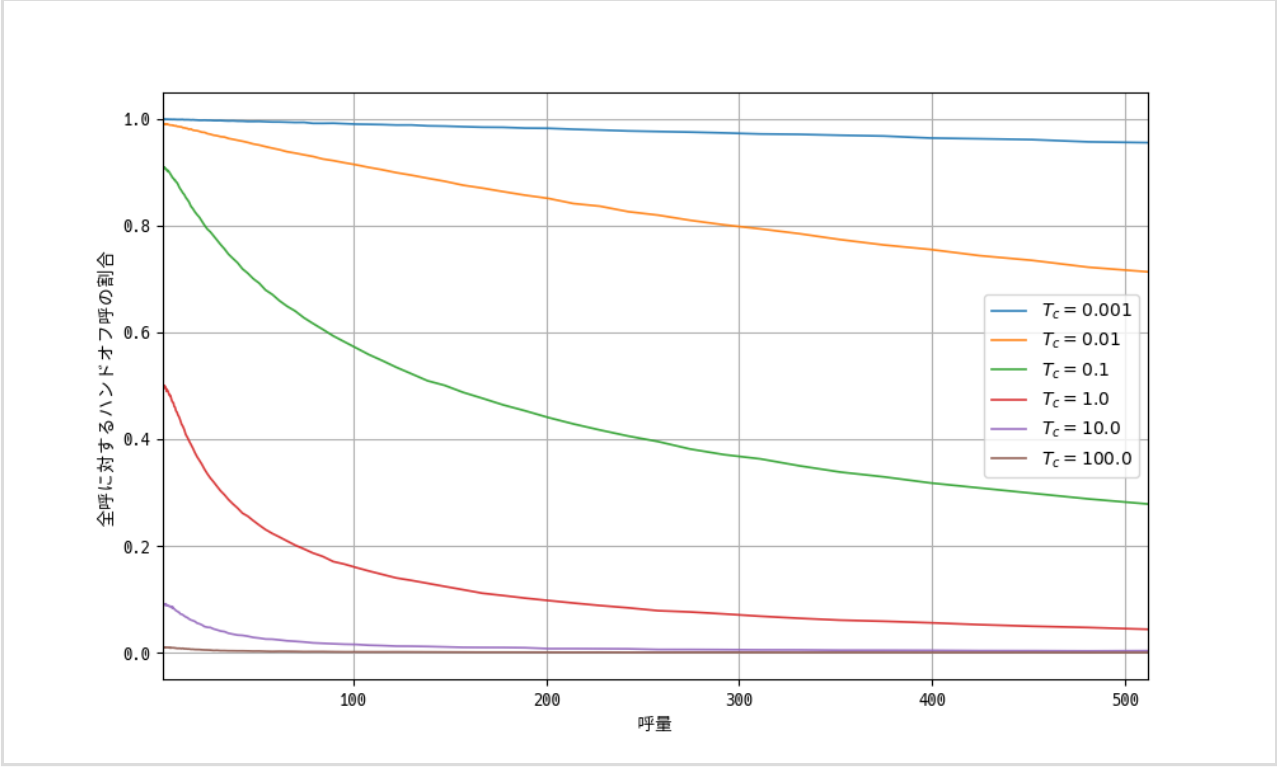
考察

1セルを通過するのにかかる時間における比較

このような比較をした理由としては, 今回の移動のモデルに置いて, 端末の速度とセルの長さはそれぞれ別の関係でなく, 合わせて「1セルを通過するのにかかる時間」としてみたほうが効果的であると考えたためである. ここで言う1セルを通過するのにかかる時間T_cとはセルの長さlと端末の速度vより, 以下のように表される.

$$T_c = \frac{l}{v}$$

結果のグラフを見ると, T_cが小さい値をとるほど呼損率が低くなる傾向が見られる. これは, T_cが小さくなるほど, 全体の呼に対して, ハンドオフ呼の割合が増加するためである(下記の図参照). 具体的には, 今回想定したモデルでは, 移動体の速度がすべて一定であるため, 何回もハンドオフしてもその分呼損が増えるわけではなく, ハンドオフの多くなればなるほど呼損の割合が減り, 呼損率が少なくなるため, ハンドオフ呼の割合が多くなればなるほど全体の呼損率も減少する.



ここからアーランB式を利用して, 上記シミュレーションを数式で出したいと思う.
移動が無い場合の呼損率は生起に成功した呼の数をX_t, 失敗した呼の数をX_fとしたとき, $\frac{X_f}{X_t+X_f}$となる. 先に述べたことから, ハンドオフ呼に対してほとんど呼損が発生しないとすると, ハンドオフ呼の数をYとして, 移動を考慮した場合の呼損率は$\frac{X_f}{X_t+X_f+Y}$となる. このことから, 移動がない場合の呼損率から, 移動を考慮した呼損率を求めるには, $\frac{X_f}{X_t+X_f} \cdot Z = \frac{X_f}{X_t+X_f+Y}$となるような$Z$, つまり$Z = \frac{X_t+X_f}{X_t+X_f+Y}$を求めればよい.
$X_t + X_f$はハンドオフ呼でない呼の総数であるため, 呼量としてみればよいため, Yを具体的に求めればよい.
現状までで, 1セルあたりの通過時間T_cと呼量aから移動を考慮した呼損率を求める式を一旦まとめておくと以下のようになる.

$$f(T_c, a) = E_S(a) \cdot \frac{a}{a+Y}$$

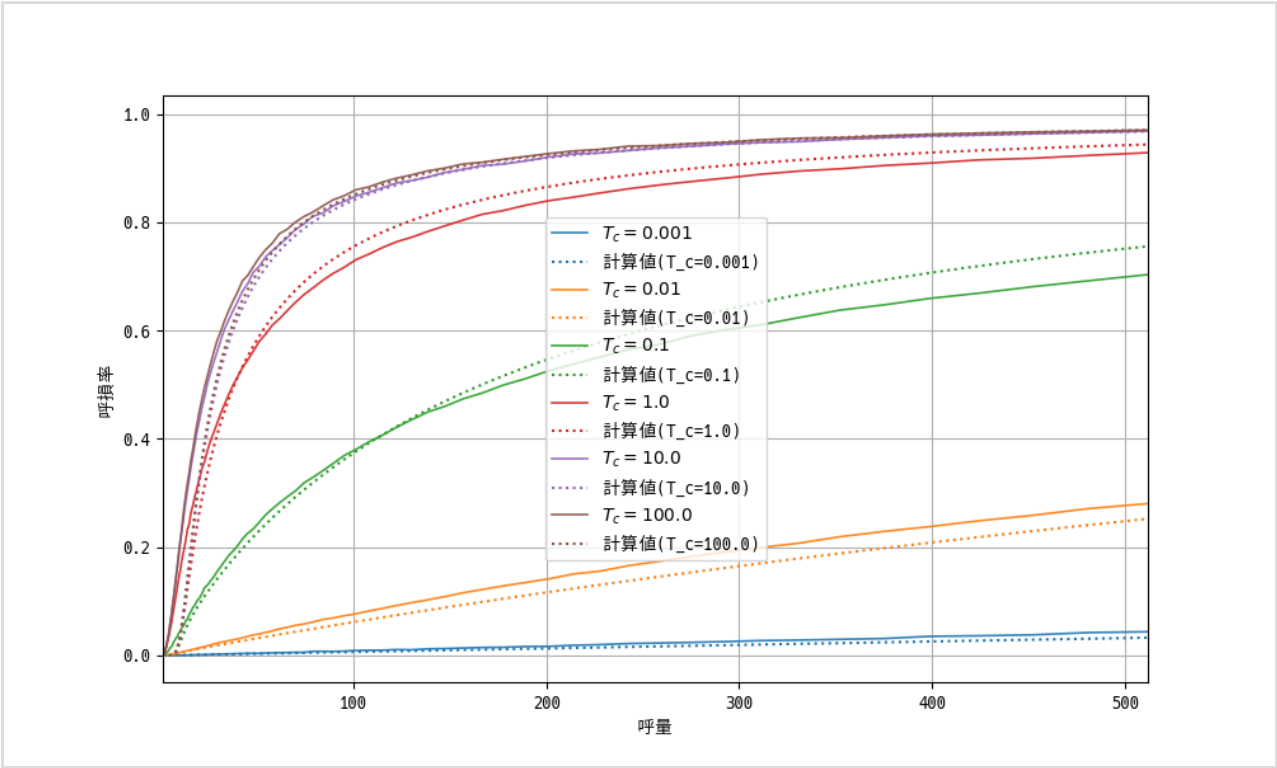
ハンドオフ呼の数Yはサービス下にある平均トークン数N_sと各トークンがハンドオフするまでの平均時間T_hに大きく依存すると考えられる. T_hに関して, サービス提供時間が十分に長いと考えた時, ほとんどのハンドオフにかかる時間がT_cとなるため, $T_h \approx T_c$と近似できる. またサービス下の平均トークン数はアーランB式による呼損率とサービス容量Sより, 大まかに$E_S(a) \cdot S$と置くことができる. これらよりハンドオフ呼の数Yは以下のように示すことができる.

$$Y \approx N_s \cdot \frac{1}{T_h} \approx \frac{E_S(a) \cdot S}{T_c}$$

このことから, 推定式は以下のようになる.

$$f(T_c, a) = E_S(a) \cdot \frac{a}{a + \frac{E_S(a) \cdot S}{T_c}}$$

この推定式とシミュレーションによる値を比較したグラフを以下に示す.



アーランB式との比較

今回, グラフ中に比較のために載せたアーランB式のキャパシティは `15` としたが, これは大まかにみればシステム全体でキャパシティが15だとみなせるためである.
実際に, 1セルを通過するのにかかる時間T_cが長い場合とほとんど一致している様子が見られる.

しかしよく見ると, 呼量が少ない時にアーランB式の値よりも少し高い呼損率をとっていることがわかる.
これはアーランB式が1つのセルで考えているのに対し, 今回のモデルでは複数のセルで考えていることに起因すると思われる. つまり1つのセルが15のキャパシティを持っていれば確実に15までサービス対象を確保できるが, 複数のセルがあるモデルだと1つのセルにアクセスが集中すると全体で15確保する前に呼損が発生する. このため呼量が少ない段階だと, アーランB式の値よりも今回のモデルの方が呼損が大きくなったと考えられる.