信州大学工学部

学士論文

情報鮮度の視点から見たバッファで割り込みのある待ち行列通信システムの実験的考察

指導教員　西新 幹彦 准教授

学科　電子情報システム工学科
学籍番号　18T2141F
氏名　細海 俊介

2022 年 2 月 18 日

目次

1　はじめに

防犯カメラ，ネットワークカメラなど遠隔地から観測しモニタへ映し出す場合では，素早く最新の情報を映す必要がある．しかし情報は通信路を通じて送信されている間に古くなり，モニタに表示される時には，程度はともかく過去のものとなっている．さらに，表示された情報も次の情報が届くまで時間の経過とともに古くなり続ける．このような情報の鮮度を定量的に表現する指標として情報鮮度 (Age of infomation:AoI) というものがある．AoI が小さいほど情報が新しいことを意味する．この用語を用いれば，モニタに表示されている情報の AoI は小さいことが望ましいと言える．この問題に対する従来研究 [1] では，センサでサンプリングされた情報が送信機に到着した順番に処理される FIFO(First In First Out) のシステムが考えられている．また，別の研究 [2] では，送信機の処理中に新しい情報が到着したら古い情報の処理を中断し割り込みをして優先的に処理する LIFO（Last In First Out）を用いたシステムをが考えられている．これらのシステムではモニタに表示されている情報の AoI の時間平均は到着間隔とサービス時間を用いて数理的に求められ，LIFO の方が FIFO より AoI が小さくなることが明らかにされている．これらの研究で想定されたシステムでは，情報が符号語に変換されておらず，かつ，サービス時間は指数分布に従っている．この点に注目し，情報を符号語に変換し符号語長ごとにサービス時間を設定した研究 [5] がある．しかし，この研究は，サービス時間が指数分布に従っていないため AoI を数理的に求めることが困難であると考えられる．

本研究では，サービス時間を指数分布に従わせたらどうなるかシミュレーションをして，考察した．2 章では，AoI の定義，時間平均 AoI の計算方法について述べる．3 章では，時間平均 AoI を小さくする割り込みがあるシステムに関する従来研究 [4] と情報を符号語に変換し符号語長ごとにサービス時間を設定した研究 [5] の紹介をする．4 章では，サービス時間が指数分布に従う符号器を追加した通信モデルの詳細について述べる．5 章では，シミュレーションの結果として情報源の分布を変化させたときの時間平均 AoI と，割り込みが発生しどれだけの情報が棄却されたのかという棄却率について様々な観点から見た検証結果を示す．6 章では，本稿のまとめと今後の課題について述べる．

2　AoI の定義

本章では AoI について説明する．

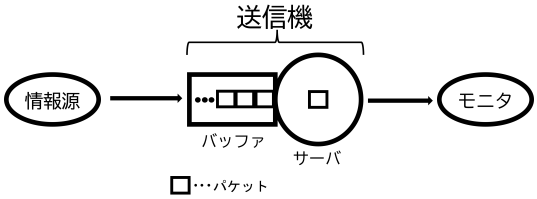

図 1　通信システムの例

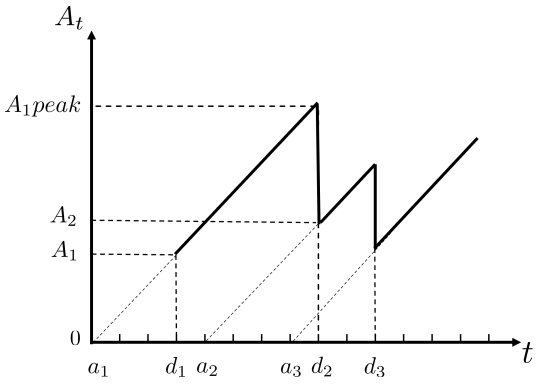

図 2　AoI の遷移図

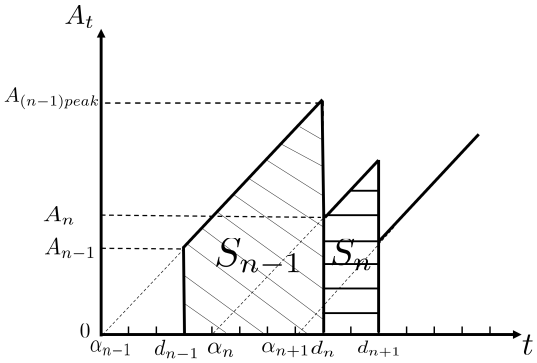

図 3　AoI の時間平均導出

2.1　AoI

AoI を説明するために図 1 の通信システムを考える．このシステムでは情報源から出力されたシンボルは到着順に送信機のバッファに格納され保管される．サーバではバッファに到着した順番でパケットの処理が行われネットワークを通じて遠隔地のモニタに表示される．モニタ

に表示された直後のシンボルは最新のため年齢は小さいが，時間経過とともにモニタに表示中のシンボルの年齢は斜め 45° で増加していく．そしてモニタの表示が新しいシンボルに更新されるとき新しいシンボルの年齢まで瞬時に降下する．この情報の新しさを定量化するための指標を AoI と呼ぶ．時刻 t にモニタに表示されているシンボルの AoI A_t はその情報の発生時刻 a とおくと，

$$A_t = t - a \tag{1}$$

と書ける．図 2 に AoI の時間変化の例を示す．1 番目に発生したシンボル X_1 が時刻 a_1 でバッファに保持される．a_1 から時間が経過するとシンボル X_1 の AoI が増加していき情報は古くなる．時刻 d_1 にサーバの処理が終了するとモニタにシンボル X_1 が表示される．シンボル X_1 が表示された瞬間の値は，

$$A_1 = d_1 - a_1 \tag{2}$$

と書ける．さらに，時刻 a_2 で 2 番目のシンボル X_2 が発生する．ここで時刻 d_2 に注目する．このとき，直前までモニタに表示されていたシンボルは更新されてシンボル X_2 がモニタに表示される．モニタの表示内容が更新されることで，AoI の値が A_1peak から A_2 と小さくなる．このようにモニタの表示内容は更新を繰り返すことから，図 2 のノコギリ状の実線グラフが得られる．これが，モニタに表示された情報の AoI の時間的変化である．次節で AoI を時間平均で評価する方法について述べる．

2.2 時間平均 AoI

AoI の評価方法には，更新される直前のピーク AoI がある目標値以下にとどまっている時間割合で評価する方法や，AoI を計測時間の時間平均で評価する方法などがある．本研究の検討方法として AoI を時間平均で評価する方法を採用した．

観測時間 T の AoI の時間平均を A_{ave} とすると，

$$A_{ave} = \frac{1}{T}\int_0^T A_t dt \tag{3}$$

と書ける．図 3 のグラフから A_{ave} は，

$$A_{ave} = \frac{1}{T}\sum_{n=1}^{l} S_n \tag{4}$$

$$S_n = \frac{1}{2}\{(d_{n+1} - a_n)^2 - (d_n - a_n)^2\} \tag{5}$$

と書ける．ここで，S_n はシンボル x_n の AoI をモニタに表示された時刻 d_n から更新された時刻 d_{n+1} までの時間で囲んだ台形面積であり，l は観測時間 T で最後に表示されたシンボルが発生した順序 (l 番目) である．

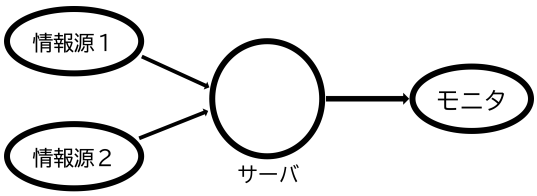

図 4　サーバで割り込みが発生する通信システム

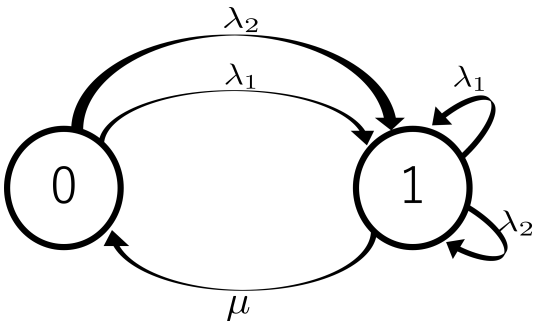

図 5　サーバで割り込みが発生する通信システムの状態遷移図

3　従来研究の紹介

モニタの AoI に注目すると，シンボルの発生時刻と現在時刻の差が小さいほど AoI の値が小さくなりより新しいシンボルが表示されることになる．また，シンボルがモニタに表示された時点での AoI はバッファでの待ち時間とサーバでのサービス時間によって決まる．そこで，それらの時間を短くするために古いシンボルを破棄して新しいシンボルを優先的に処理する方法が考えられる．以下でそのようなシステムに対する従来研究 [4]，さらに，シンボルを符号語に変換し符号語長ごとにサービス時間を設定した研究 [5] の紹介をする．

3.1　サーバで割り込みが発生する研究

サーバで割り込みが発生する通信システムを図 4 に示す．情報源は 2 種類あり，着目したいシンボルは情報源 1 から到着レート λ_1 で出力され，それ以外のシンボルは情報源 2 から到着レート λ_2 で出力される．情報源から出力されたシンボルは送信機にあるサーバに入れられサービスレート μ で処理され遠隔地のモニタにシンボルが表示される．ここで，サーバで処理中に新しいシンボルがサーバに到着すると処理中シンボルを棄却し新しいシンボルを処理することでモニタに表示されているシンボルの AoI を小さくすることができる．サーバの状態

は待機中かパケットの処理中かのどちらかであり，次のパケットの到着や処理の終了によって状態が遷移する．この様子を表した状態遷移図を図 5 に示す．図中の状態 0 と状態 1 は，それぞれサーバの待機中と処理中を表している．待機中のサーバには情報源 1 と 2 からそれぞれ到着レート λ_1 と λ_2 でシンボルが到着する．いずれかの情報源からシンボルが到着するとサーバは処理中の状態へ遷移する．サーバは処理を終えると待機中の状態になる．ただしサーバの処理中にもいずれかの情報源から情報が届く可能性があり，その場合は処理中のシンボルを破棄して新しいシンボルの処理が改めて開始される．この状態遷移に基づいて時間平均 AoI の値が従来研究 [3] から次のようにして求められる．

定理 1．時間平均 AoI
情報源 1 の時間平均 AoI を Δ，各状態の AoI に相当する量を v_q，Q を状態の集合とすると，

$$\Delta = \sum_{q \in Q} v_q \tag{6}$$

となる定理がある．

図 5 のシステムで考えると，情報源 1，2，全体の稼働率は ρ_1，ρ_2，ρ は，

$$\rho_1 = \frac{\lambda_1}{\mu} \tag{7}$$

$$\rho_2 = \frac{\lambda_2}{\mu} \tag{8}$$

$$\rho = \rho_1 + \rho_2 \tag{9}$$

と書ける．また，v_0,v_1 は，

$$v_0 = \frac{1}{\rho_1(1+\rho)}(\frac{1}{\mu} + \frac{1+\rho_2}{\mu(1+\rho)}\rho) \tag{10}$$

$$v_1 = \frac{\rho}{\mu(1+\rho)} + \frac{\rho}{\rho_1(1+\rho)}(\frac{1}{\mu} + \frac{1+\rho_2}{\mu(1+\rho)}\rho) \tag{11}$$

と導出される．Q={0,1} より時間平均 AoIΔ は代入すると，

$$\Delta = v_0 + v_1 \tag{12}$$

$$= \frac{1+\rho_1+\rho_2}{\mu\rho_1} \tag{13}$$

$$= \frac{\mu+\lambda_1+\lambda_2}{\mu\lambda_1} \tag{14}$$

と書ける．式 (14) より，時間平均 AoI を小さくするためにはサービスレート μ と到着レート λ_1 を大きくし，λ_2 を小さくすればいいことが分かる．ただし，λ_1 だけを大きくするとサーバが処理中であっても新しいシンボルが到着したら処理を中断しなければならないためシンボルがモニタに表示される頻度が減ってしまうことに注意しなければならない．

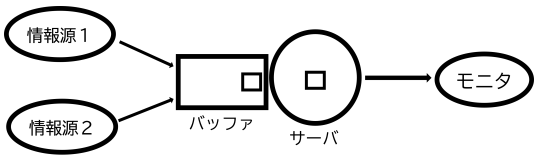

図 6　バッファで割り込みが発生する通信システム

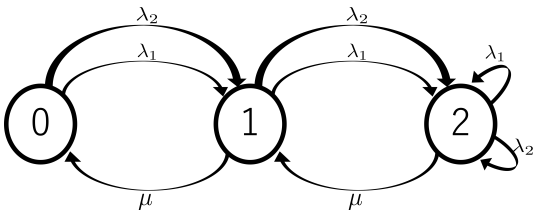

図 7　バッファで割り込みが発生する通信システムの状態遷移図

3.2　バッファで割り込みが発生する研究

バッファで割り込みが発生する通信システムを図 6 に示す．情報源は 2 種類あり，着目したいシンボルは情報源 1 から到着レート λ_1 で出力され，それ以外のシンボルは情報源 2 から到着レート λ_2 で出力される．情報源から出力されたシンボルは送信機にあるサーバへ入れられサービスレート μ で処理され遠隔地のモニタにシンボルが表示される．サーバの前段にはバッファが設置してありサーバが処理中の時に新しいシンボルが出力されたら 1 つだけ保持することができる．ここで，サーバで処理中かつバッファでシンボルが保持されているとき新しいシンボルがバッファに到着するとバッファ内のシンボルを棄却し新しいシンボルを保持することで待機時間が短くなり，モニタに表示されているシンボルの AoI を小さくすることができる．サーバの状態は待機中かパケットの処理中かのどちらかであり，次のパケットの到着や処理の終了によって状態が遷移する．この様子を表した状態遷移図を図 7 に示す．図中の状態 0 と状態 1 と状態 2 は，それぞれバッファが空でサーバ待機中，バッファが空でサーバ処理中，バッファ保持かつサーバ処理中を表している．各状態へ向かう遷移についてはサーバで割り込みが発生する場合と同様である．この状態遷移に基づいて時間平均 AoI の値が次のように求められる．

$$[\pi_0 \quad \pi_1 \quad \pi_2] = C_\pi [1 \quad \rho \quad \rho^2] \tag{15}$$

$$C_\pi = (1 + \rho + \rho^2)^{-1} \tag{16}$$

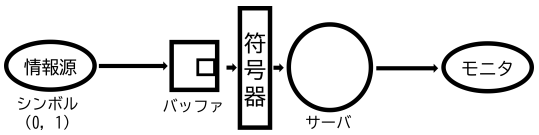

図 8　符号語長ごとにサービス時間を変える通信システム

とする．v_0,v_1,v_2 は，

$$\rho v_0 = \frac{\pi_0}{\mu} + v_{11} \tag{17}$$

$$v_1 = \frac{1}{\mu(1+\rho)} + v_{11} \tag{18}$$

$$v_2 = \frac{\pi_2}{\mu} + \rho v_1 \tag{19}$$

と導出できる．ここで $v_{11} = \frac{\rho}{\mu(1+\rho)}\frac{1}{\rho_1} - \frac{C_\pi(1+\rho+\rho^3)}{\mu(1+\rho)^2}$ である．定理 1，Q={0,1,2} より時間平均 AoIΔ は代入すると，

$$\Delta = v_0 + v_1 + v_2 \tag{20}$$

$$= \frac{1}{\mu} + \frac{\pi_0 + \pi_2}{\rho} + \frac{1+\rho+\rho^2}{\rho} v_{11} \tag{21}$$

と書ける．このシステムではバッファで割り込みが発生するためバッファでの待ち時間が発生してしまうがサーバに到着したシンボルは確実にモニタに表示することができる．サーバで割り込みが発生するシステムと比較すると AoI は大きくなってしまうが，着目したいシンボルの表示される頻度は増えると考えられる．

3.3　符号語長ごとにサービス時間を変えた研究

符号語長ごとにサービス時間を変える通信システムを図 8 に示す．この図 8 はバッファで割り込みが発生する通信システムに符号器を追加したシステムになっている．このシステムは，情報源から発生したシンボルを符号器で符号語に変換してサーバで処理し，ネットワークを通じて遠隔地のモニタに送信する．そして，モニタでは符号語が復号されシンボルが表示される．ここでサーバに処理中のシンボルがある際新しいシンボルはバッファで待機する．バッファが保持できるシンボルは 1 つであり，待機してるところにさらに新しいシンボルがきたらバッファで待機しているシンボルは棄却され新しいシンボルが代わりに待機する．このようなバッファで割り込みが発生する通信システムを考える．情報源についてはシンボルがポアソン過程で到着すると考え，シンボルの種類は簡単のため ‘0’,‘1’ の 2 種類とする．サーバは符号語の長さに比例した時間で処理をする．そのとき，符号語長が短いほどよりはやくシンボルが伝

達されるため AoI は小さくなる．符号として語頭符号を使用する．したがって，符号語長はクラフトの不等式に従う．以下でクラフトの不等式及びその使用方法について説明する．

定理 2．クラフトの不等式
符号語長が $d_0, d_1 ... d_n$ である瞬時復号可能な符号が存在するための必要十分条件は，n を符号語の数，r を符号の種類数とすると，

$$\sum_{i=0}^{n} r^{-d_i} \leq 1 \tag{22}$$

を満たすことである．

この定理 2 を使い，シンボル'0','1' の符号語長を d_0,d_1 とすると，

$$2^{-d_0} + 2^{-d_1} \leq 1 \tag{23}$$

となる．左辺は 1 以下になればよいので等号と設定して，それぞれ変数を与えると，

$$k_0 + k_1 = 1 \tag{24}$$
$$k_0 = 2^{-d_0} \tag{25}$$
$$k_1 = 2^{-d_1} \tag{26}$$

と定義できる．このとき，式 (24) の条件の元でサービス時間を決定している．この従来研究 [5] では，時間平均 AoI と棄却率について計測を行っている．棄却率は発生したシンボルのうちバッファで棄却されたシンボルの割合で求められる．研究結果としては，時間平均 AoI を最小にすると棄却率も最小になることを確認している．

4 サーバでのサービス時間が指数分布に従う符号器を追加した通信モデル

4.1 システムの説明

3 章では，従来研究として 3 種類のシステムについて説明した．3.1 節，3.2 節の研究ではサーバでのサービス時間は指数分布にしたがっている．3.3 節の研究では式 (25),(26) の d_0,d_1，つまり，符号語長に比例してサーバでのサービス時間を決定していた．本研究ではこの違いに着目してサーバでのサービス時間が指数分布に従うバッファで割り込みがある符号器を追加した通信モデルを設定し，AoI はどう変化するのか考えた．通信システムの構成は 3.3

節と同じ図 8 であるがサーバの挙動が変わっている．このシステムは，情報源から発生したシンボルは符号器で符号語に変換され送信機にあるサーバで処理され，ネットワークを通じて遠隔地のモニタに送信される．情報源について，シンボルがポアソン過程で到着すると考え，シンボルの種類は簡単のため ‘0’,‘1’ の 2 種類とする．シンボルはバッファまでポアソン到着し，符号器で符号語に変換されサーバで処理される．サーバでのサービス時間は符号語長を期待値とした指数分布によって決定する．符号語長は語頭符号なので定理 2 のクラフトの不等式を満たす．通常，符号語長は整数でなければならないがサービス時間を実数で表現するため符号語長を実数として扱う．モニタでは符号語が復号され表示される．また，サーバに処理中のシンボルがある際新しいシンボルはバッファで待機する．新しい情報を常に送りたいのでバッファで保持できるシンボルは 1 つとし，待機してるところにさらに新しいシンボルがきたらバッファで待機しているシンボルは棄却され新しいシンボルが待機する．考察するに当たって検討する内容は平均 AoI，棄却率とした．平均 AoI は 2.2 節で説明した方法で導出する．棄却率は情報源から発生したシンボルの個数に対する割りこまれたシンボルの個数の割合で定義する．

4.2 シミュレーションの説明

プログラムで通信システムをシミュレートした．具体的には，サーバのサービス時間を指数分布に従わせ，パラメータとして情報源のシンボルの種類を ‘0’,‘1’，情報源からバッファまでの到着レートを 1.0，サーバにてサービス中に新しいシンボルが到着したら保持されるシンボルの数を 1 つとし，情報源から発生する情報の数を 1000000 と固定した．また，情報源におけるシンボル ‘1’ の発生確率 p_1 を変化させ，符号語長は式 (24) に従うように決定してシミュレーションでは式 (25) の k_0 を変えて実験した．

5 結果と考察

5.1 分布と最小 AoI

シンボル’1’ の発生確率 $p_1 = 0.5$ のときの時間平均 AoI を図 9 に示す．横軸 k_0 は式 (25) で定まる値である．横軸の値が小さいほどシンボル ‘0’ の符号語長が長くなるためサーバでのサービス時間は大きくなる．縦軸は時間平均 AoI の値であり，縦軸の値が小さいほどモニタに表示される情報は新しいことになる．$(k_0,AoI) = (0.5048,2.4136)$ の点を最小値とするグラフが得られた．このときの最小値の導出方法は付録 A に記載した．以後，最小値をあらわす際にはこの方式で導出をしている．まず，モニタに表示されている情報の AoI を小さくするためにはシンボル ‘0’,‘1’ のサービス時間を限りなく小さくすればよい．しかし，クラフトの不等式の制限があるため一方のサービス時間を短くするともう一方のサービス時間が大きくなる関係にある．それゆえ，シンボル ‘1’ の発生確率 $p_1 = 0.5$ のときは，‘0’,‘1’ が等確率で発生す

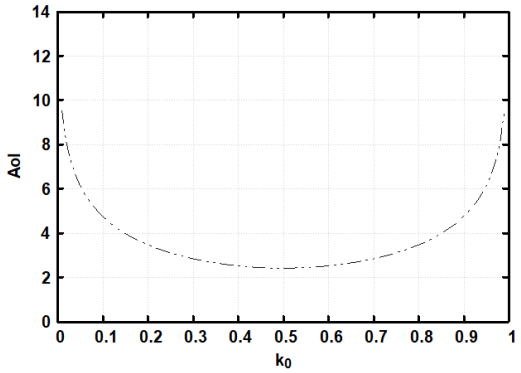

図 9　$p_1 = 0.5$ の時間平均 AoI

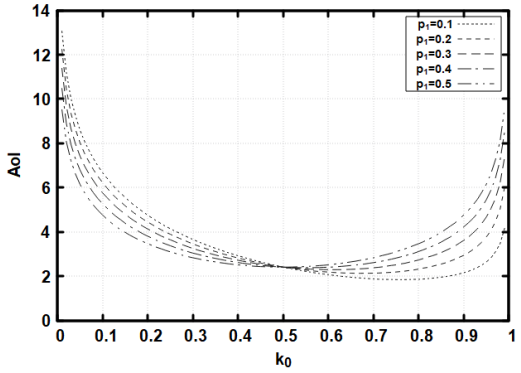

図 10　分布を変更したときの時間平均 AoI

るためサービス時間が同一の $k_0 = 0.5$ にしたときが時間平均 AoI は最小値になると考えられる．実際の最小値は $k_0 = 0.5048$ のときであった．これはサービス時間が指数分布に従っているため極端に大きいまたは小さい数値が現れることがあるからだと推測できる．

次に，シンボル ‘1’ の発生確率 $p_1 = 0.1$ から 0.5 と変化したときの時間平均 AoI の結果を図 10 に示す．結果から各グラフの最小値は $p_1 = 0.1$ のとき $(k_0,AoI) = (0.7598,1.8485)$，$p_1 = 0.2$ のとき $(k_0,AoI) = (0.669,2.133)$，$p_1 = 0.3$ のとき $(k_0,AoI) = (0.598,2.298)$，$p_1 = 0.4$ のとき $(k_0,AoI) = (0.5421,2.3883)$，$p_1 = 0.5$ のとき $(k_0,AoI) = (0.5048,2.4136)$ である．p_1 が小さくなるほど最小値はグラフの右下に向かう傾向が得られた．これは，情報源の分布が偏っているため頻繁に出力されるシンボルはサービス時間を短くし，稀に出力されるシンボルのサービス時間は大きくすることで全体としてモニタには新しい情報が表示され時間平均 AoI は小さくなるためである．分布の偏りが大きい，つまりシンボル ‘1’ の発生確率が小さくなる

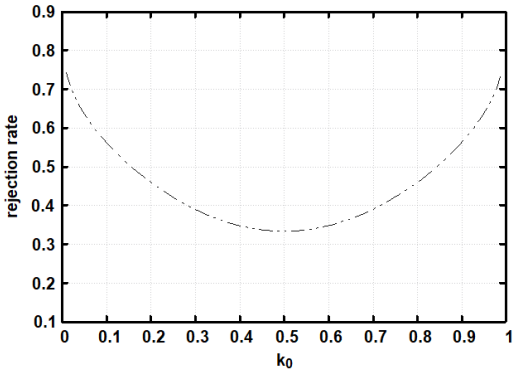

図 11　$p_1 = 0.5$ の棄却率

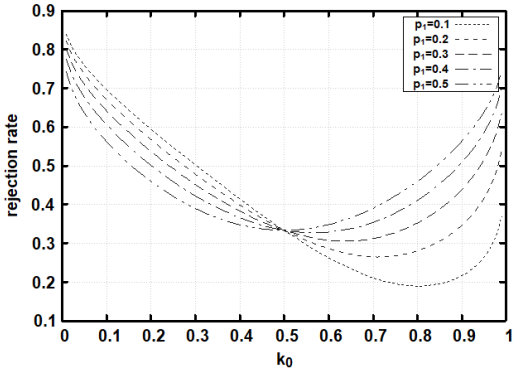

図 12　分布を変更したときの棄却率

ほどサービス時間の配分も偏り，頻繁に出力されるシンボルを短いサービス時間で処理することとなりグラフの最小値が右下へと向かうと考えられる．また，本研究でシュミレーションしたシステムから符号器だけをなくしたシステムを k_0=0.5 のときだと考えることができる。図 10 より k_0 の値を変えると時間平均 AoI の最小値が k_0=0.5 のときより低くなることが分かる．したがって，符号器を追加すると時間平均 AoI を小さくできると言える．

5.2　分布と最小棄却率

前節では時間平均 AoI についての結果を示した．この節ではバッファで割り込みが発生した割合である棄却率について考える．まず，シンボル ‘1’ の発生確率 $p_1 = 0.5$ としたときの棄却率の結果を図 11 に示す．横軸は k_0，縦軸は棄却率 (rejection rate:rr) であり，値が ‘0’ に近づくほど割り込みが少なくなり情報源から出力されたシンボルが棄却されずにモニタに表示さ

れることになる．このとき棄却率はシンボルごとに見ているのではなくシンボル全体の棄却率を計測した．これは，シンボルごとの棄却率を見てもそれはシンボルの発生確率に依存するため計測しても自明であるからである．最小値は $(k_0,rr) = (0.4943,0.3325)$ の点をとる下に凸のグラフが得られた．評価する対象が変わってもクラフトの不等式の制限があるため，一方のサービス時間を短くするともう一方のサービス時間が大きくなる関係は変わらずに存在する．したがって，シンボル '1' の発生確率 $p_1 = 0.5$ のときは，'0','1' が等確率で発生するためサービス時間が同一の $k_0 = 0.5$ にしたときが棄却率は最小値になると考えられる．実際の最小値は $k_0 = 0.4943$ のときであった．これはサービス時間が指数分布に従っているため極端に大きいまたは小さい数値が現れることがあるからだと推測できる．

次に，シンボル '1' の発生確率 $p_1 = 0.1$ から 0.5 と変化したときの棄却率の結果を図 12 に示す．結果から各グラフの最小値は $p_1 = 0.1$ のとき $(k_0,AoI) = (0.8,0.188)$,$p_1 = 0.2$ のとき $(k_0,AoI) = (0.7132,0.2639)$,$p_1$=0.3 のとき $(k_0,AoI) = (0.635,0.3044)$,$p_1 = 0.4$ のとき $(k_0,AoI) = (0.5771,0.3266)$,$p_1 = 0.5$ のとき $(k_0,AoI) = (0.4943,0.3325)$ である．p_1 が小さくなるほど最小値はグラフの右下に向かう傾向が得られた．これは，情報源の分布が偏っているため頻繁に出力されるシンボルはサービス時間を短くし，稀に出力されるシンボルのサービス時間は大きくすることで全体として到着間隔よりもサービス時間が短くなり，棄却が発生しづらくなる．分布の偏りが大きいほどサービス時間の配分も偏り，頻繁に出力されるシンボルを短いサービス時間で処理することとなりグラフの最小値が右下に向かうと考えられる．また，前節と同様に符号器のないシステムを k_0=0.5 のときだと考えることができる。図 12 より k_0 の値を変えると棄却率の最小値が k_0=0.5 のときより低くなることが分かる．したがって，符号器を追加すると棄却率を小さくできると言える．

5.3 最小時間平均 AoI となるときの棄却率

モニタに表示される情報の AoI は小さいほうが望ましい．そこで，情報源の分布ごとに時間平均 AoI の最小値を測定し，確率を変えて繰り返し測定を行い，どのようになるか検証することにした．さらに，時間平均 AoI が最小値を取るときの棄却率も同時に測定した．まず，シンボル '1' の発生確率 $p_1 = 0.01$ から 0.5 と変化させたときの確率ごとの最小時間平均 AoI の結果を図 13 に示す．横軸は p_1，縦軸は AoI である．以降の結果は横軸 p_1 の範囲を 0.0 から 0.5 としている．これは，シンボルの種類が 2 種類のため 0.5 から 1.0 の範囲は対称なグラフが得られることが自明である．結果からシンボル '1' の発生確率が等確率に近づくと AoI が増加し，情報源の分布に偏りが大きいほど AoI は小さくなることが分かる．次に，最小時間平均 AoI となるときの棄却率を図 14 に示す．横軸は p_1，縦軸は棄却率である．結果からシンボル '1' の発生確率が等確率に近づくと棄却率が増加し，情報源の分布に偏りが大きいほど棄却率は小さくなることが分かる．図 13 と図 14 から，情報源の分布が分かれば AoI が最小になる

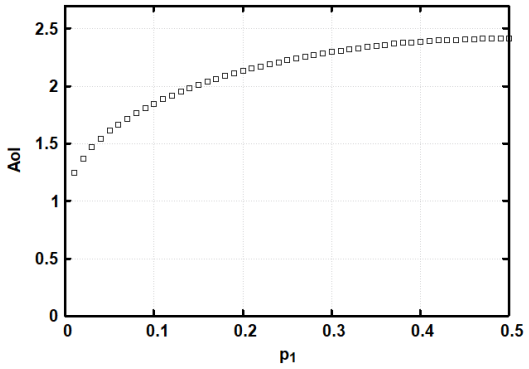

図 13　発生確率ごとの最小時間平均 AoI

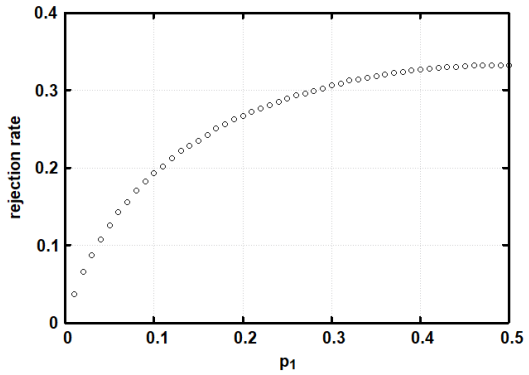

図 14　最小時間平均 AoI となるときの棄却率

システムを設計した際に棄却率の許容値を示すことができると考えられる．

5.4　最小棄却率となるときの時間平均 AoI

バッファが保持できるシンボルの数が少ないとき，なるべく情報を棄却しないで送信したいといった場合の符号器の条件にも興味がある．そこで情報源の分布を変化させたときの最小棄却率とそのときの時間平均 AoI を測定した．まず，シンボル ‘1’ の発生確率 p_1 を変化させ，確率ごとの最小棄却率の結果を図 15 に示す．横軸は p_1，縦軸は棄却率である．次に，最小棄却率になるときの時間平均 AoI を図 16 に示す．横軸は p_1，縦軸は AoI である．AoI，棄却率ともに p_1 の増加に従って値が増加する傾向が得られた．図 15 と図 16 から，情報源の分布が分かれば棄却率が最小になるシステムを設計した際に時間平均 AoI に許容値を示すことができると考えられる．

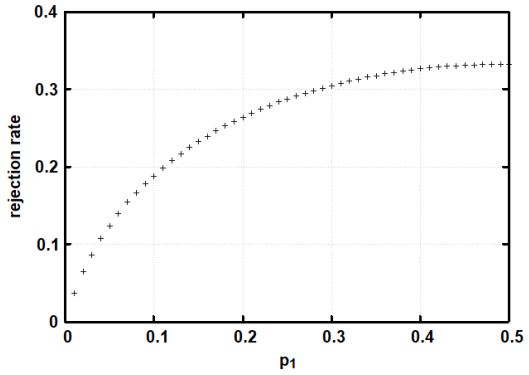

図 15　発生確率ごとの最小棄却率

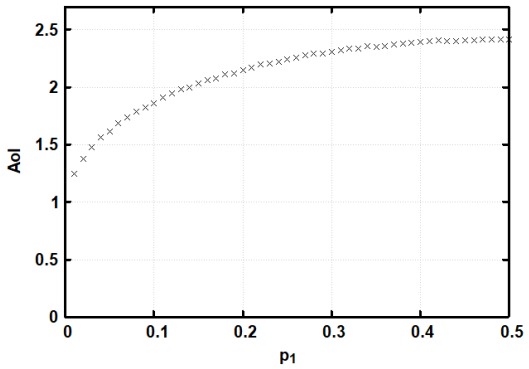

図 16　最小棄却率となるときの時間平均 AoI

5.5　最小値のときの時間平均 AoI と棄却率の比較

5.3 節，5.4 節ではそれぞれが最小値のときのもう一方の値について見ていた．そこで，最小時間平均 AoI と最小棄却率のときの AoI，またその逆で最小棄却率と最小時間平均 AoI のときの棄却率を比較してみたくなった．AoI について図 17，棄却率について図 18 に示す．どちらともほぼ一致するという結果が得られた．しかし，同じ確率のとき，それぞれの値の k_0 を見てみると値が違うため同時に起こらないことが分かった．

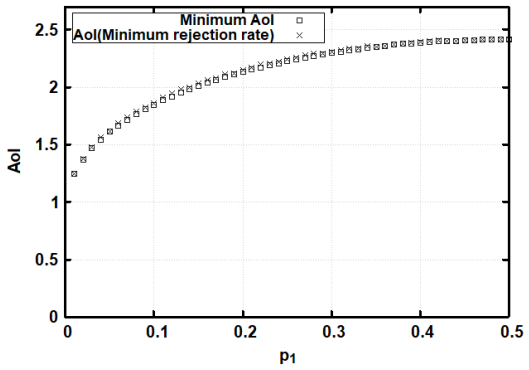

図 17　最小時間平均 AoI と最小棄却率のときの AoI の比較

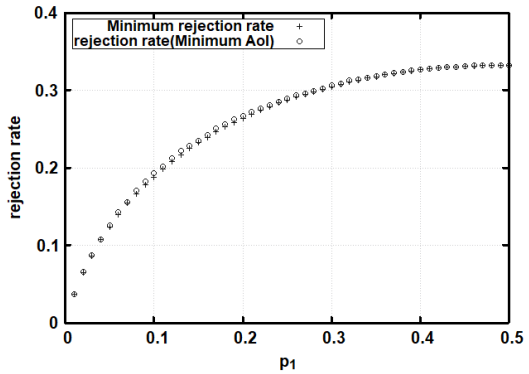

図 18　最小棄却率と最小時間平均 AoI のときの棄却率の比較

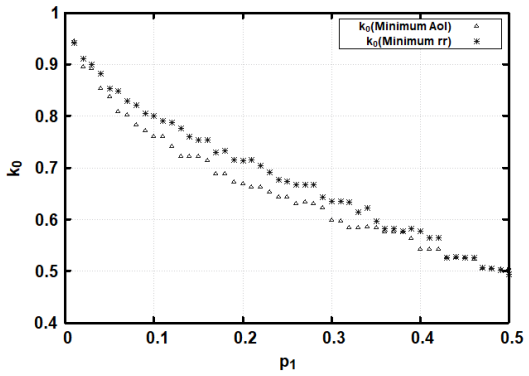

図 19　最小 AoI，最小棄却率時の k_0

5.6 最小時間平均 AoI，最小棄却率時の k_0 の比較

前節で最小値ともう一方の最小値のときの値について見たところ同じ確率のとき k_0 が違う，つまり，シンボル '0' のサービス時間が違うことが分かった．この k_0 はシステムを設計する際エンジニアが手を加えられる場所なのでそれぞれが最小値のときはどんな値になるか検証を行った．最小時間平均 AoI，最小棄却率時の k_0 を図 19 に示す．横軸は p_1，縦軸は k_0 である．グラフより両者 p_1 の増加とともに k_0 は現象する傾向が得られ，等確率に近い場所では k_0 の差があまり見られなかった．また，グラフが綺麗な直線状にプロットされなかった理由としてはサービス時間が指数分布に従っているため極端に大きいまたは小さい値が出るため安定しないと考えた．ここで，k_0 の違いによって時間平均 AoI と棄却率にどれだけ影響しているのかを考える．図 17，図 18 を見てみると分布の偏りが変化してもどちらもほぼ一致することが分かる．したがって，分布の偏りが分かれば図 19 で示した 2 つのグラフの間の k_0 の値に設定すれば時間平均 AoI，棄却率ともに最小値に近い値が得られると考えられる．

6 まとめ

本研究では AoI という視点に基づきバッファで割り込みが発生する通信システムに符号器を追加し，サービス時間が符号語長を期待値とする指数分布に従い決定するモデルについて検証を行った．また，バッファで割り込みが発生するため割り込みの頻度である棄却率についても検証をした．時間平均 AoI と棄却率はシンボル '1' の発生確率が等確率に近づくと増加し，分布に偏りがあると減少する傾向が得られた．符号器を追加しなかったシステムを k_0=0.5 のときだと考えられ，そのときより時間平均 AoI，棄却率ともに低い値をとることができた．これらは符号器を追加した利点だと言える。また，情報源の分布が分かれば時間平均 AoI，棄却率ともに最小値に近い値を符号器の設定を変えることで得られることが分かった．今後の課題として，本研究でシミュレーションした通信システムを数理的に解析できないか検討していきたい．

謝辞

本研究に際して，様々なご指導をしていただいた指導教員の西新幹彦先生，また助言をしていただいた研究室の先輩方に深く感謝申し上げます．

参考文献

[1] 井上文彰，滝根哲哉，「Age of Information (AoI) 基本概念と研究動向」，http://www.ieice.org/ess/sita/forum/article/2018/201810111910.pdf，2022年1月閲覧.

[2] Y. Inoue, H. Masuyama, T. Takine and T. Tanaka, "A general formula for the stationary distribution of the age of information and its application to single-server queues", IEEE Trans. Inf. Theory, vol.65, no.12, pp.8305–8324, Dec. 2019.

[3] R. D. Yates and S. K. Kaul, "The age of information: Real-time status updating by multiple sources", IEEE Trans. Inf. Theory, vol.64, no.3, pp.1807–1827, Mar. 2019.

[4] S. Kaul, R. Yates and M. Gruteser, "Real-time status: How often should one update?", in Proc. of IEEE INFOCOM, pp.2731–2735, Mar. 2012.

[5] 千葉直紀，「情報鮮度の観点に基づくバッファで割り込みがある通信システムの実験的評価」，信州大学大学院総合理工学研究科修士論文（指導教員：西新幹彦），2021年2月.

[6] mt19937ar:Mersenne Twister with improved initialization, http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/MT2002/mt19937ar.html, 2022年1月閲覧.

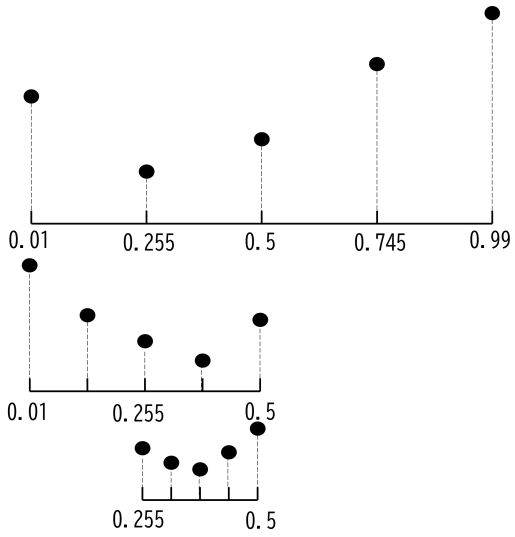

図 20　最小値を探すアルゴリズムの考え方

付録 A　最小値を探すアルゴリズムの説明

プログラムを設計しシミュレーションをするにあたって高い精度で値の最小値を求めたくなった．そこで，最小値を探すアルゴリズムを作った．以下ではそのアルゴリズムの説明をする．このときの考え方を図 20 に示す．本論文のシミュレーションではクラフトの不等式を満たし ‘0’ の発生割合を 0.1 から 0.99 へと変化させ平均 AoI，棄却率の最小値を探している．このときどちらのグラフも下に凸なグラフを得ることが分かった．したがって，等間隔に値を 5 つ取り，その中での最小値の割合を中央値として再度等間隔に値を 5 つ取ることを繰り返して行けば平均 AoI，棄却率の最小値が求まるのではと考えた．割合の中央値と最小値の差が 0.000001 以下になるまで繰り返すように設定した．このアルゴリズムを作ることでシミュレーションの効率は良くなった．

付録 B　ソースコード

B.1　サーバでのサービス時間が指数分布に従う通信モデルをあらわしたプログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "MT.h"
```

```
double OCCUR_PROBABILITY_1 = 0.0;       //1が出る確率

/*lambda に従う指数分布*/
double Random_Exp(double Lambda) {
    return -log(genrand_real2()) / Lambda;
}

/*確率 p で1を乱数生成*/
int Random_Binary(double Probability) {
    int Symbol;
    if ((double)genrand_real2() >= (1.0 - Probability)) {
        Symbol = 1;
    }
    else {
        Symbol = 0;
    }
    return Symbol;
}

/*100 万回まわして平均 AoI を出す関数 (サービスレートは指数分布に従い決定)*/
double AoI_Func(double Kraft_0) {

    /*クラフトの不等式によりパラメータ導出*/
    double Kraft_Exp_0;
    double Kraft_Exp_1;
    double Kraft_1;
    double Service_Rate_0;
    double Service_Rate_1;
    Kraft_1 = 1 - Kraft_0;
    Kraft_Exp_0 = log2(Kraft_0);
    Kraft_Exp_1 = log2(Kraft_1);
    Service_Rate_0 = -1 / Kraft_Exp_0;
    Service_Rate_1 = -1 / Kraft_Exp_1;

    /*初期設定*/
    double Time = 0.0;                  //現在時刻
    double Next_Arrive_Time = 0.0;      //次の到着
    double Service_End_Time = -1.0;     //サービス終了時刻 (ないとき-1.0)
    double Occur_Srv_Time = 0.0;        //サーバ処理中の情報の発生時刻
    double Occur_Buf_Time = 0.0;        //バッファの中の情報の発生時刻
    int Count_Arrival = 0;              //到着数
    double Count_Occur = 0.0;           //発生数
#define NUM 1000000                     //上限値
    int Type_Srv = -1;                  //サーバの中身 (0 or 1 or 空なら-1)
    int Type_Buf = -1;                  //バッファの中身 (0 or 1 or 空なら-1)
#define ARRIVE_RATE 1.0                 //到着レート
/*AoI 計算必要*/
    double Occur_Time = 0.0;                //発生時刻
    double Display_Time = 0.0;              //表示時刻
    double Occur_One_Before_Time = 0.0;     //ひとつ前の発生時刻
    double Display_One_Before_Time = 0.0;   //ひとつ前の表示時刻
    double AoI = 0.0;                       //AoI
    double Total_AoI = 0.0;                 //AoI 全体を足したもの
    double Ave_AoI = 0.0;                   // 平均 AoI
    double First_Display_Time = 0.0;        //最初の表示時刻
    double Last_Display_Time = 0.0;         //最後の表示時刻


    init_genrand(19991202);
    Next_Arrive_Time = Random_Exp(ARRIVE_RATE);
    while (Next_Arrive_Time >= 0 || Service_End_Time >= 0) {
        if (Count_Arrival == NUM) {
            Ave_AoI = Total_AoI / (Display_Time - First_Display_Time);
            break;
        }
        else if (Type_Srv == -1 || Service_End_Time > Next_Arrive_Time) {
            Time = Next_Arrive_Time;
            if (Type_Buf == -1) {
                if (Type_Srv == -1) {
```

```
                    Occur_Srv_Time = Time;
                    Type_Srv = Random_Binary(OCCUR_PROBABILITY_1);
                    Count_Occur++;
                    if (Type_Srv == 0) {
                        Service_End_Time = Time + Random_Exp(Service_Rate_0);
                    }
                    else if (Type_Srv == 1) {
                        Service_End_Time = Time + Random_Exp(Service_Rate_1);
                    }


                }
                else {
                    Occur_Buf_Time = Time;
                    Type_Buf = Random_Binary(OCCUR_PROBABILITY_1);
                    Count_Occur++;

                }

            }
            else {
                Occur_Buf_Time = Time;
                Type_Buf = Random_Binary(OCCUR_PROBABILITY_1);
                Count_Occur++;

            }

            Next_Arrive_Time = Time + Random_Exp(ARRIVE_RATE);
        }
        else {
            Time = Service_End_Time;
            Count_Arrival++;

            /*AoI 導出関係*/
            Occur_One_Before_Time = Occur_Time;
            Occur_Time = Occur_Srv_Time;
            Display_One_Before_Time = Display_Time;
            Display_Time = Time;
            if (Occur_One_Before_Time == 0.0) {
                First_Display_Time = Display_Time;
            }
            if (Occur_One_Before_Time != 0.0) {
                AoI = 0.5 * (pow(Display_Time - Occur_One_Before_Time, 2) -
                (pow(Display_One_Before_Time - Occur_One_Before_Time, 2)));
                Total_AoI += AoI;
            }
            if (Type_Buf == -1) {
                Type_Srv = -1;
            }
            else {
                Type_Srv = Type_Buf;
                Occur_Srv_Time = Occur_Buf_Time;
                if (Type_Srv == 0) {
                    Service_End_Time = Time + Random_Exp(Service_Rate_0);
                }
                else if (Type_Srv == 1) {
                    Service_End_Time = Time + Random_Exp(Service_Rate_1);
                }
                Type_Buf = -1;
            }
        }
    }
    return Ave_AoI;
}

/*100 万回まわして棄却率を出す関数 (サービスレートは指数分布に従い決定)*/
double Rejection_Rate_Func(double Kraft_0) {

    /*クラフトの不等式によりパラメータ導出*/
```

```
    double Kraft_Exp_0;
    double Kraft_Exp_1;
    double Kraft_1;
    double Service_Rate_0;
    double Service_Rate_1;
    Kraft_1 = 1 - Kraft_0;
    Kraft_Exp_0 = log2(Kraft_0);
    Kraft_Exp_1 = log2(Kraft_1);
    Service_Rate_0 = -1 / Kraft_Exp_0;
    Service_Rate_1 = -1 / Kraft_Exp_1;

    /*初期設定*/
    double Time = 0.0;                //現在時刻
    double Next_Arrive_Time = 0.0;    //次の到着
    double Service_End_Time = -1.0;   //サービス終了時刻（ないとき-1.0）
    double Occur_Srv_Time = 0.0;      //サーバ処理中の情報の発生時刻
    double Occur_Buf_Time = 0.0;      //バッファの中の情報の発生時刻
    int Count_Arrival = 0;            //到着数
    double Count_Occur = 0.0;         //発生数
#define NUM 1000000                   //上限値
    int Type_Srv = -1;                //サーバの中身（0 or 1 or 空なら-1）
    int Type_Buf = -1;                //バッファの中身（0 or 1 or 空なら-1）
#define ARRIVE_RATE 1.0               //到着レート
/*AoI 計算必要*/
    double Occur_Time = 0.0;               //発生時刻
    double Display_Time = 0.0;             //表示時刻
    double Occur_One_Before_Time = 0.0;    //ひとつ前の発生時刻
    double Display_One_Before_Time = 0.0;  //ひとつ前の表示時刻
    double AoI = 0.0;                      //AoI
    double Total_AoI = 0.0;                //AoI 全体を足したもの
    double Ave_AoI = 0.0;                  // 平均 AoI
    double First_Display_Time = 0.0;       //最初の表示時刻
    double Last_Display_Time = 0.0;        //最後の表示時刻
    double Rejection_Rate = 0.0;           //棄却率


    init_genrand(19991202);
    Next_Arrive_Time = Random_Exp(ARRIVE_RATE);
    while (Next_Arrive_Time >= 0 || Service_End_Time >= 0) {
        if (Count_Arrival == NUM) {
            Ave_AoI = Total_AoI / (Display_Time - First_Display_Time);
            Rejection_Rate = (Count_Occur - NUM) / Count_Occur;
            break;
        }
        else if (Type_Srv == -1 || Service_End_Time > Next_Arrive_Time) {
            Time = Next_Arrive_Time;
            if (Type_Buf == -1) {
                if (Type_Srv == -1) {
                    Occur_Srv_Time = Time;
                    Type_Srv = Random_Binary(OCCUR_PROBABILITY_1);
                    Count_Occur++;
                    if (Type_Srv == 0) {
                        Service_End_Time = Time + Random_Exp(Service_Rate_0);
                    }
                    else if (Type_Srv == 1) {
                        Service_End_Time = Time + Random_Exp(Service_Rate_1);
                    }


                }
                else {
                    Occur_Buf_Time = Time;
                    Type_Buf = Random_Binary(OCCUR_PROBABILITY_1);
                    Count_Occur++;

                }

            }
            else {
```

```
                Occur_Buf_Time = Time;
                Type_Buf = Random_Binary(OCCUR_PROBABILITY_1);
                Count_Occur++;

            }

            Next_Arrive_Time = Time + Random_Exp(ARRIVE_RATE);
        }
        else {
            Time = Service_End_Time;



            /*AoI 導出関係*/
            Occur_One_Before_Time = Occur_Time;
            Occur_Time = Occur_Srv_Time;
            Display_One_Before_Time = Display_Time;
            Display_Time = Time;
            if (Occur_One_Before_Time == 0.0) {
                First_Display_Time = Display_Time;
            }
            if (Occur_One_Before_Time != 0.0) {
                AoI = 0.5 * (pow(Display_Time - Occur_One_Before_Time, 2) -
                (pow(Display_One_Before_Time - Occur_One_Before_Time, 2)));
                Total_AoI += AoI;
            }
            if (Type_Buf == -1) {
                Type_Srv = -1;
            }
            else {
                Type_Srv = Type_Buf;
                Occur_Srv_Time = Occur_Buf_Time;
                if (Type_Srv == 0) {
                    Service_End_Time = Time + Random_Exp(Service_Rate_0);
                }
                else if (Type_Srv == 1) {
                    Service_End_Time = Time + Random_Exp(Service_Rate_1);
                }
                Type_Buf = -1;
            }
        }
    }
    return Rejection_Rate;
}

/*最小の AoI を求める関数*/
double Min_AoI_Func(double  OCCUR_PROBABILITY_1) {

    double AoI[5] = { AoI_Func(0.01),AoI_Func(0.255),AoI_Func(0.5),AoI_Func(0.745),AoI_Func(0.99) };
    double Kraft_0_AoI_Search_0 = 0.01;
    double Kraft_0_AoI_Search_1 = 0.255;
    double Kraft_0_AoI_Search_2 = 0.5;
    double Kraft_0_AoI_Search_3 = 0.745;
    double Kraft_0_AoI_Search_4 = 0.99;

    int Count_AoI_Search = 0;

    while (Count_AoI_Search < 20) {

        int mini_AoI = 1;
        if (AoI[2] < AoI[mini_AoI]) {
            mini_AoI = 2;
        }
        if (AoI[3] < AoI[mini_AoI]) {
            mini_AoI = 3;
        }

        switch (mini_AoI) {
        case 1:
```

```
            AoI[4] = AoI[2];
            AoI[2] = AoI[1];
            Kraft_0_AoI_Search_4 = Kraft_0_AoI_Search_2;
            Kraft_0_AoI_Search_2 = Kraft_0_AoI_Search_1;
            break;
        case 2:
            AoI[0] = AoI[1];
            AoI[4] = AoI[3];
            Kraft_0_AoI_Search_0 = Kraft_0_AoI_Search_1;
            Kraft_0_AoI_Search_4 = Kraft_0_AoI_Search_3;
            break;
        case 3:
            AoI[0] = AoI[2];
            AoI[2] = AoI[3];
            Kraft_0_AoI_Search_0 = Kraft_0_AoI_Search_2;
            Kraft_0_AoI_Search_2 = Kraft_0_AoI_Search_3;
            break;
        }
        Kraft_0_AoI_Search_1 = (Kraft_0_AoI_Search_0 + Kraft_0_AoI_Search_2) * 0.5;
        Kraft_0_AoI_Search_3 = (Kraft_0_AoI_Search_2 + Kraft_0_AoI_Search_4) * 0.5;
        AoI[1] = AoI_Func(Kraft_0_AoI_Search_1);
        AoI[3] = AoI_Func(Kraft_0_AoI_Search_3);
        Count_AoI_Search += 1;

    }

    int mini_AoI = 1;
    if (AoI[2] < AoI[mini_AoI]) {
        mini_AoI = 2;
    }
    if (AoI[3] < AoI[mini_AoI]) {
        mini_AoI = 3;
    }

    double Min_AoI = 0.0;
    Min_AoI = AoI[mini_AoI];
    return Min_AoI;
}

/*最小の棄却率を求める関数*/
double Min_Rejection_Rate_Func(double  OCCUR_PROBABILITY_1) {

    double Rejection_Rate[5] = { Rejection_Rate_Func(0.01),Rejection_Rate_Func(0.255),Rejection_Rate_Func(0.5),
    Rejection_Rate_Func(0.745),Rejection_Rate_Func(0.99) };
    double Kraft_0_Rejection_Rate_Search_0 = 0.01;
    double Kraft_0_Rejection_Rate_Search_1 = 0.255;
    double Kraft_0_Rejection_Rate_Search_2 = 0.5;
    double Kraft_0_Rejection_Rate_Search_3 = 0.745;
    double Kraft_0_Rejection_Rate_Search_4 = 0.99;

    int Count_Rejection_Rate_Search = 0;

    while (Count_Rejection_Rate_Search < 20) {

        int mini_Rejection_Rate = 1;
        if (Rejection_Rate[2] < Rejection_Rate[mini_Rejection_Rate]) {
            mini_Rejection_Rate = 2;
        }
        if (Rejection_Rate[3] < Rejection_Rate[mini_Rejection_Rate]) {
            mini_Rejection_Rate = 3;
        }

        switch (mini_Rejection_Rate) {
        case 1:
            Rejection_Rate[4] = Rejection_Rate[2];
            Rejection_Rate[2] = Rejection_Rate[1];
            Kraft_0_Rejection_Rate_Search_4 = Kraft_0_Rejection_Rate_Search_2;
            Kraft_0_Rejection_Rate_Search_2 = Kraft_0_Rejection_Rate_Search_1;
            break;
```

```
        case 2:
            Rejection_Rate[0] = Rejection_Rate[1];
            Rejection_Rate[4] = Rejection_Rate[3];
            Kraft_0_Rejection_Rate_Search_0 = Kraft_0_Rejection_Rate_Search_1;
            Kraft_0_Rejection_Rate_Search_4 = Kraft_0_Rejection_Rate_Search_3;
            break;
        case 3:
            Rejection_Rate[0] = Rejection_Rate[2];
            Rejection_Rate[2] = Rejection_Rate[3];
            Kraft_0_Rejection_Rate_Search_0 = Kraft_0_Rejection_Rate_Search_2;
            Kraft_0_Rejection_Rate_Search_2 = Kraft_0_Rejection_Rate_Search_3;
            break;
        }
        Kraft_0_Rejection_Rate_Search_1 = (Kraft_0_Rejection_Rate_Search_0 + Kraft_0_Rejection_Rate_Search_2) * 0.5;
        Kraft_0_Rejection_Rate_Search_3 = (Kraft_0_Rejection_Rate_Search_2 + Kraft_0_Rejection_Rate_Search_4) * 0.5;
        Rejection_Rate[1] = Rejection_Rate_Func(Kraft_0_Rejection_Rate_Search_1);
        Rejection_Rate[3] = Rejection_Rate_Func(Kraft_0_Rejection_Rate_Search_3);
        Count_Rejection_Rate_Search += 1;

    }

    int mini_Rejection_Rate = 1;
    if (Rejection_Rate[2] < Rejection_Rate[mini_Rejection_Rate]) {
        mini_Rejection_Rate = 2;
    }
    if (Rejection_Rate[3] < Rejection_Rate[mini_Rejection_Rate]) {
        mini_Rejection_Rate = 3;
    }
    //printf("棄却率最小値は  %f\n", Rejection_Rate[mini_Rejection_Rate]);
    double Min_Rejection_Rate = 0.0;
    Min_Rejection_Rate = Rejection_Rate[mini_Rejection_Rate];
    return Min_Rejection_Rate;
}

/*最小の AoI の時の Kraft_0 を求める関数*/
double Kraft_0_Min_AoI_Func(double  OCCUR_PROBABILITY_1) {

    double AoI[5] = { AoI_Func(0.01),AoI_Func(0.255),AoI_Func(0.5),AoI_Func(0.745),AoI_Func(0.99) };
    double Kraft_0_AoI_Search_0 = 0.01;
    double Kraft_0_AoI_Search_1 = 0.255;
    double Kraft_0_AoI_Search_2 = 0.5;
    double Kraft_0_AoI_Search_3 = 0.745;
    double Kraft_0_AoI_Search_4 = 0.99;

    int Count_AoI_Search = 0;

    while (Count_AoI_Search < 20) {

        int mini_AoI = 1;
        if (AoI[2] < AoI[mini_AoI]) {
            mini_AoI = 2;
        }
        if (AoI[3] < AoI[mini_AoI]) {
            mini_AoI = 3;
        }

        switch (mini_AoI) {
        case 1:
            AoI[4] = AoI[2];
            AoI[2] = AoI[1];
            Kraft_0_AoI_Search_4 = Kraft_0_AoI_Search_2;
            Kraft_0_AoI_Search_2 = Kraft_0_AoI_Search_1;
            break;
        case 2:
            AoI[0] = AoI[1];
            AoI[4] = AoI[3];
            Kraft_0_AoI_Search_0 = Kraft_0_AoI_Search_1;
            Kraft_0_AoI_Search_4 = Kraft_0_AoI_Search_3;
            break;
```

```
        case 3:
            AoI[0] = AoI[2];
            AoI[2] = AoI[3];
            Kraft_0_AoI_Search_0 = Kraft_0_AoI_Search_2;
            Kraft_0_AoI_Search_2 = Kraft_0_AoI_Search_3;
            break;
        }
        Kraft_0_AoI_Search_1 = (Kraft_0_AoI_Search_0 + Kraft_0_AoI_Search_2) * 0.5;
        Kraft_0_AoI_Search_3 = (Kraft_0_AoI_Search_2 + Kraft_0_AoI_Search_4) * 0.5;
        AoI[1] = AoI_Func(Kraft_0_AoI_Search_1);
        AoI[3] = AoI_Func(Kraft_0_AoI_Search_3);
        Count_AoI_Search += 1;

    }
    double Kraft_0_Min_AoI = Kraft_0_AoI_Search_1;
    int mini_AoI = 1;
    if (AoI[2] < AoI[mini_AoI]) {
        mini_AoI = 2;
        Kraft_0_Min_AoI = Kraft_0_AoI_Search_2;
    }
    if (AoI[3] < AoI[mini_AoI]) {
        mini_AoI = 3;
        Kraft_0_Min_AoI = Kraft_0_AoI_Search_3;
    }

    return Kraft_0_Min_AoI;
}

/*最小の棄却率の時の Kraft_0 を求める関数*/
double Kraft_0_Min_Rejection_Rate_Func(double  OCCUR_PROBABILITY_1) {

    double Rejection_Rate[5] = { Rejection_Rate_Func(0.01),Rejection_Rate_Func(0.255),Rejection_Rate_Func(0.5),
    Rejection_Rate_Func(0.745),Rejection_Rate_Func(0.99) };
    double Kraft_0_Rejection_Rate_Search_0 = 0.01;
    double Kraft_0_Rejection_Rate_Search_1 = 0.255;
    double Kraft_0_Rejection_Rate_Search_2 = 0.5;
    double Kraft_0_Rejection_Rate_Search_3 = 0.745;
    double Kraft_0_Rejection_Rate_Search_4 = 0.99;

    int Count_Rejection_Rate_Search = 0;

    while (Count_Rejection_Rate_Search < 20) {

        int mini_Rejection_Rate = 1;
        if (Rejection_Rate[2] < Rejection_Rate[mini_Rejection_Rate]) {
            mini_Rejection_Rate = 2;
        }
        if (Rejection_Rate[3] < Rejection_Rate[mini_Rejection_Rate]) {
            mini_Rejection_Rate = 3;
        }

        switch (mini_Rejection_Rate) {
        case 1:
            Rejection_Rate[4] = Rejection_Rate[2];
            Rejection_Rate[2] = Rejection_Rate[1];
            Kraft_0_Rejection_Rate_Search_4 = Kraft_0_Rejection_Rate_Search_2;
            Kraft_0_Rejection_Rate_Search_2 = Kraft_0_Rejection_Rate_Search_1;
            break;
        case 2:
            Rejection_Rate[0] = Rejection_Rate[1];
            Rejection_Rate[4] = Rejection_Rate[3];
            Kraft_0_Rejection_Rate_Search_0 = Kraft_0_Rejection_Rate_Search_1;
            Kraft_0_Rejection_Rate_Search_4 = Kraft_0_Rejection_Rate_Search_3;
            break;
        case 3:
            Rejection_Rate[0] = Rejection_Rate[2];
            Rejection_Rate[2] = Rejection_Rate[3];
            Kraft_0_Rejection_Rate_Search_0 = Kraft_0_Rejection_Rate_Search_2;
            Kraft_0_Rejection_Rate_Search_2 = Kraft_0_Rejection_Rate_Search_3;
```

```
            break;
        }
        Kraft_0_Rejection_Rate_Search_1 = (Kraft_0_Rejection_Rate_Search_0 + Kraft_0_Rejection_Rate_Search_2) * 0.5;
        Kraft_0_Rejection_Rate_Search_3 = (Kraft_0_Rejection_Rate_Search_2 + Kraft_0_Rejection_Rate_Search_4) * 0.5;
        Rejection_Rate[1] = Rejection_Rate_Func(Kraft_0_Rejection_Rate_Search_1);
        Rejection_Rate[3] = Rejection_Rate_Func(Kraft_0_Rejection_Rate_Search_3);
        Count_Rejection_Rate_Search += 1;

    }
    double Kraft_0_Min_Rejection_Rate = Kraft_0_Rejection_Rate_Search_1;
    int mini_Rejection_Rate = 1;
    if (Rejection_Rate[2] < Rejection_Rate[mini_Rejection_Rate]) {
        mini_Rejection_Rate = 2;
        Kraft_0_Min_Rejection_Rate = Kraft_0_Rejection_Rate_Search_2;
    }
    if (Rejection_Rate[3] < Rejection_Rate[mini_Rejection_Rate]) {
        mini_Rejection_Rate = 3;
        Kraft_0_Min_Rejection_Rate = Kraft_0_Rejection_Rate_Search_3;
    }
    return Kraft_0_Min_Rejection_Rate;
}

/*最小の AoI の時の棄却率を求める関数*/
double Rejection_Rate_of_Min_AoI_Func(double  OCCUR_PROBABILITY_1) {

    double AoI[5] = { AoI_Func(0.01),AoI_Func(0.255),AoI_Func(0.5),AoI_Func(0.745),AoI_Func(0.99) };
    double Kraft_0_AoI_Search_0 = 0.01;
    double Kraft_0_AoI_Search_1 = 0.255;
    double Kraft_0_AoI_Search_2 = 0.5;
    double Kraft_0_AoI_Search_3 = 0.745;
    double Kraft_0_AoI_Search_4 = 0.99;

    int Count_AoI_Search = 0;

    while (Count_AoI_Search < 20) {

        int mini_AoI = 1;
        if (AoI[2] < AoI[mini_AoI]) {
            mini_AoI = 2;
        }
        if (AoI[3] < AoI[mini_AoI]) {
            mini_AoI = 3;
        }

        switch (mini_AoI) {
        case 1:
            AoI[4] = AoI[2];
            AoI[2] = AoI[1];
            Kraft_0_AoI_Search_4 = Kraft_0_AoI_Search_2;
            Kraft_0_AoI_Search_2 = Kraft_0_AoI_Search_1;
            break;
        case 2:
            AoI[0] = AoI[1];
            AoI[4] = AoI[3];
            Kraft_0_AoI_Search_0 = Kraft_0_AoI_Search_1;
            Kraft_0_AoI_Search_4 = Kraft_0_AoI_Search_3;
            break;
        case 3:
            AoI[0] = AoI[2];
            AoI[2] = AoI[3];
            Kraft_0_AoI_Search_0 = Kraft_0_AoI_Search_2;
            Kraft_0_AoI_Search_2 = Kraft_0_AoI_Search_3;
            break;
        }
        Kraft_0_AoI_Search_1 = (Kraft_0_AoI_Search_0 + Kraft_0_AoI_Search_2) * 0.5;
        Kraft_0_AoI_Search_3 = (Kraft_0_AoI_Search_2 + Kraft_0_AoI_Search_4) * 0.5;
        AoI[1] = AoI_Func(Kraft_0_AoI_Search_1);
        AoI[3] = AoI_Func(Kraft_0_AoI_Search_3);
        Count_AoI_Search += 1;
```

```
    }
    double Kraft_0_Min_AoI = Kraft_0_AoI_Search_1;
    int mini_AoI = 1;
    if (AoI[2] < AoI[mini_AoI]) {
        mini_AoI = 2;
        Kraft_0_Min_AoI = Kraft_0_AoI_Search_2;
    }
    if (AoI[3] < AoI[mini_AoI]) {
        mini_AoI = 3;
        Kraft_0_Min_AoI = Kraft_0_AoI_Search_3;
    }

    return Rejection_Rate_Func(Kraft_0_Min_AoI);
}

/*最小の棄却率の時の AoI を求める関数*/
double AoI_of_Min_Rejection_Rate_Func(double  OCCUR_PROBABILITY_1) {

    double Rejection_Rate[5] = { Rejection_Rate_Func(0.01),Rejection_Rate_Func(0.255),Rejection_Rate_Func(0.5),
    Rejection_Rate_Func(0.745),Rejection_Rate_Func(0.99) };
    double Kraft_0_Rejection_Rate_Search_0 = 0.01;
    double Kraft_0_Rejection_Rate_Search_1 = 0.255;
    double Kraft_0_Rejection_Rate_Search_2 = 0.5;
    double Kraft_0_Rejection_Rate_Search_3 = 0.745;
    double Kraft_0_Rejection_Rate_Search_4 = 0.99;

    int Count_Rejection_Rate_Search = 0;

    while (Count_Rejection_Rate_Search < 20) {

        int mini_Rejection_Rate = 1;
        if (Rejection_Rate[2] < Rejection_Rate[mini_Rejection_Rate]) {
            mini_Rejection_Rate = 2;
        }
        if (Rejection_Rate[3] < Rejection_Rate[mini_Rejection_Rate]) {
            mini_Rejection_Rate = 3;
        }

        switch (mini_Rejection_Rate) {
        case 1:
            Rejection_Rate[4] = Rejection_Rate[2];
            Rejection_Rate[2] = Rejection_Rate[1];
            Kraft_0_Rejection_Rate_Search_4 = Kraft_0_Rejection_Rate_Search_2;
            Kraft_0_Rejection_Rate_Search_2 = Kraft_0_Rejection_Rate_Search_1;
            break;
        case 2:
            Rejection_Rate[0] = Rejection_Rate[1];
            Rejection_Rate[4] = Rejection_Rate[3];
            Kraft_0_Rejection_Rate_Search_0 = Kraft_0_Rejection_Rate_Search_1;
            Kraft_0_Rejection_Rate_Search_4 = Kraft_0_Rejection_Rate_Search_3;
            break;
        case 3:
            Rejection_Rate[0] = Rejection_Rate[2];
            Rejection_Rate[2] = Rejection_Rate[3];
            Kraft_0_Rejection_Rate_Search_0 = Kraft_0_Rejection_Rate_Search_2;
            Kraft_0_Rejection_Rate_Search_2 = Kraft_0_Rejection_Rate_Search_3;
            break;
        }
        Kraft_0_Rejection_Rate_Search_1 = (Kraft_0_Rejection_Rate_Search_0 + Kraft_0_Rejection_Rate_Search_2) * 0.5;
        Kraft_0_Rejection_Rate_Search_3 = (Kraft_0_Rejection_Rate_Search_2 + Kraft_0_Rejection_Rate_Search_4) * 0.5;
        Rejection_Rate[1] = Rejection_Rate_Func(Kraft_0_Rejection_Rate_Search_1);
        Rejection_Rate[3] = Rejection_Rate_Func(Kraft_0_Rejection_Rate_Search_3);
        Count_Rejection_Rate_Search += 1;

    }
    double Kraft_0_Min_Rejection_Rate = Kraft_0_Rejection_Rate_Search_1;
    int mini_Rejection_Rate = 1;
```

```
    if (Rejection_Rate[2] < Rejection_Rate[mini_Rejection_Rate]) {
        mini_Rejection_Rate = 2;
        Kraft_0_Min_Rejection_Rate = Kraft_0_Rejection_Rate_Search_2;
    }
    if (Rejection_Rate[3] < Rejection_Rate[mini_Rejection_Rate]) {
        mini_Rejection_Rate = 3;
        Kraft_0_Min_Rejection_Rate = Kraft_0_Rejection_Rate_Search_3;
    }
    return AoI_Func(Kraft_0_Min_Rejection_Rate);
}

/*本文*/
int main(void) {


    OCCUR_PROBABILITY_1 = 0.5;
    double a = 0.01;
    while (a <= 1.0) {
        printf("%f\n", AoI_Func(a));
        a += 0.01;
    }

    /*OCCUR_PROBABILITY_1 = 0.01;
    while (OCCUR_PROBABILITY_1 <= 0.51) {
        //printf("%f\n", OCCUR_PROBABILITY_1);
        //printf("%f\n", Rejection_Rate_of_Min_AoI_Func(OCCUR_PROBABILITY_1));
        //printf("%f\n", AoI_of_Min_Rejection_Rate_Func(OCCUR_PROBABILITY_1));
        //printf("%f\n", Min_AoI_Func(OCCUR_PROBABILITY_1));
        //printf("%f\n", Min_Rejection_Rate_Func(OCCUR_PROBABILITY_1));
        //printf("%f\n", Kraft_0_Min_AoI_Func(OCCUR_PROBABILITY_1));
        printf("%f\n", Kraft_0_Min_Rejection_Rate_Func(OCCUR_PROBABILITY_1));

        OCCUR_PROBABILITY_1 += 0.01;
    }*/
    /*printf("%f\n", Min_AoI_Func(OCCUR_PROBABILITY_1));
    printf("%f\n", Kraft_0_Min_AoI_Func(OCCUR_PROBABILITY_1));
    printf("%f\n", Min_Rejection_Rate_Func(OCCUR_PROBABILITY_1));
    printf("%f\n", Kraft_0_Min_Rejection_Rate_Func(OCCUR_PROBABILITY_1));
    printf("%f\n", Rejection_Rate_of_Min_AoI_Func(OCCUR_PROBABILITY_1));
    printf("%f\n", AoI_of_Min_Rejection_Rate_Func(OCCUR_PROBABILITY_1));*/


}
```