信州大学工学部

学士論文

算術符号の演算精度と
ポアソン到着シンボルの遅延について

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　12T2092B
氏名　矢口 卓実

2016 年 2 月 25 日

目次

1 序論 1

2 算術符号を用いた伝送システム 1
2.1 システムの概要 1
2.2 算術符号の符号化・復号化 2
2.3 算術符号化・復号化の例 3
2.4 算術符号の演算精度 4
2.5 平均遅延の内訳 5

3 符号化遅延の検証 5
3.1 実験内容 5
3.2 結果と考察 6

4 バッファ遅延の検証 7
4.1 バッファのパンク 7
4.2 到着レートの正規化 7
4.3 実験内容 8
4.4 結果と考察 8

5 状態遷移する情報源 10
5.1 状態遷移 10
5.2 実験内容 10
5.3 結果と考察 10

6 まとめ 13

参考文献 13

付録 A 実験で用いたシステムのソースコード 15

1　序論

インターネットやスマートフォンなどの情報伝送システムにおいて, そのデータ量は膨大である. その膨大なデータをそのまま伝送することは容易ではない. そこで必要になるのがデータの圧縮である. いくつかある圧縮の方法の中でも本研究では算術符号に注目した. 算術符号は圧縮の効率も高く, 逐次符号化することのできる符号化方法である. 昨今, 音声や映像など様々なところでリアルタイムな通信を行う機会が多くなってきている. その中で通信の過程での遅延を小さくすることは望ましいことである. 算術符号を用いることで, 圧縮効率を高く保ちながらシステムの遅延も小さくすることができると考えられる. 算術符号の研究として文献 [1][2] がある. [1] ではエンコーダで圧縮した情報を送信器を用いてデコーダに送るようなシステムでシンボルの到着レートと送信器の送信レートの関係について研究がされている. [2] ではエンコーダとデコーダを直結させたシステムで算術符号の演算精度と平均遅延に関する研究が行われ, 算術符号の演算精度を下げることで遅延を抑制できることが例証されている. 本論文ではエンコーダ・デコーダ間に送信器が設置されている場合でも算術符号の演算精度を下げることが遅延の抑制に有効であるのか, 到着の方法の違いが遅延に影響を与えるのかという問題を実験的に検証する.

2　算術符号を用いた伝送システム

この章では本研究で考察の対象とする伝送システムと算術符号について説明する. 算術符号については文献 [2] に基づいて説明していく.

2.1　システムの概要

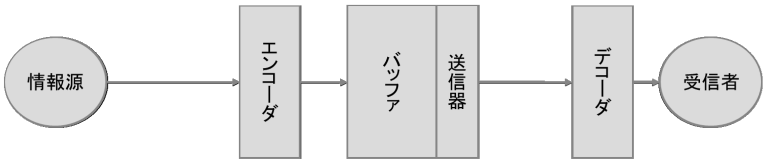

図 1　システムの概要

データ通信などで実際に用いられるシステムは複雑なものであるが, 本研究では従来の研究 [1][2] に基づき簡単化した図 1 のようなシステムを考える. 実際の通信システムでは情報源には記憶がありアルファベットは大きいのが通常であるが, 文献 [2] の指摘に基づき, 考察を 2 元の定常無記憶情報源に限定する. また, 実際の情報源からシンボルが出力されるタイミングも

一定間隔とは限らず情報源の構造によって複雑であるが, 本論文では簡単な数理モデルで表されるものを考察する. 実際の通信では通信路でノイズがのることが考えられるが, ノイズの影響も無いものとする. 算術符号の演算にかかってしまう時間についても本研究では無視できるものとする.

情報源は a,b の 2 つのシンボルをレート λ[シンボル/秒] で出力する. エンコーダ・デコーダでは逐次符号化・復号化がおこなわれる. バッファには送信器が設けられており, 送信レート $\mu = 1$[シンボル/秒] で一定間隔ごとにシンボルをデコーダへ送る.

エンコーダに到着するシンボルのタイミングは計数過程 $N(t)$ に従うとする. 計数過程 $N(t)$ に対しそのレートは

$$\lim_{t\to\infty} \frac{1}{t} E[N(t)] \tag{1}$$

と定義される. これは 1 秒当たりの到着シンボル数の期待値を表す. この意味でシンボルの到着のレートを到着レートと呼ぶ.

本論文で考えるシンボルの到着はポアソン過程に従うかまた一定間隔である. 到着レート λ のポアソン過程の到着間隔はパラメータ λ の指数分布に従う. 一定間隔の到着で到着レートが λ であるとは, 到着間隔が $\frac{1}{\lambda}$ 秒であることである.

情報源から 1 つのシンボルが出力されてからそのシンボルが受信者に届くまでの時間をそのシンボルの遅延と呼ぶことにする. また, 平均遅延とは出力されたすべて情報源シンボルの遅延の算術平均を取ったものである.

2.2 算術符号の符号化・復号化

算術符号 [1][2] は情報源系列を符号化する汎用的な方法のひとつである. 符号化法の原理は文字列から区間（半開区間）への写像に基づいており, 入力される情報源系列と出力される符号語系列はそれぞれに対応する区間を仲介して対応している. 文字列から区間への写像は次の特徴を持つ:(1) 長さゼロの文字列に対する区間は固定（例えば [0,1)）である. (2) 長さの同じ異なる文字列に対して, それらの区間は互いに素である. (3) 文字列 $\boldsymbol{x}$ とその語頭 $\boldsymbol{x'}$ に対して, $\boldsymbol{x}$ に対応する区間は $\boldsymbol{x'}$ に対応する区間に含まれる.

算術符号は文字列から区間への写像を 2 つ持つ. 情報源系列用と符号語系列用である. それぞれの写像を I_s, I_c と表わす. 情報源アルファベットと符号語アルファベットをそれぞれ $\mathcal{X}$, $\mathcal{Y}$ とする.

符号器の仕事は, 与えられた情報源系列 $\boldsymbol{x} \in \mathcal{X}^*$ に対して $I_s(\boldsymbol{x}) \subset I_c(\boldsymbol{y})$ となるような符号語系列 $\boldsymbol{y} \in \mathcal{Y}^*$ を求めることである. 一般にそのような $\boldsymbol{y}$ は複数存在するが, もっとも長いものが唯一あり, 他はそれの語頭になっている. 写像 I_s の特徴から, 情報源系列 $\boldsymbol{x}$ の先頭から順に文字を見ることによって, $I_s(\boldsymbol{x})$ を逐次的に求めていくことができる. すると, I_c の特徴から, 符号語系列 $\boldsymbol{y}$ を先頭の文字から逐次的に決定していくことができる.

復号器の仕事は符号器と逆で, 与えられた符号語系列 $\boldsymbol{y}$ に対して $I_c(\boldsymbol{y}) \subset I_s(\boldsymbol{x})$ となるような情報源系 $\boldsymbol{x}$ を求めることである. 符号語同様, 復号器も符号語系列 $\boldsymbol{y}$ の先頭から順に文字を見ることによって, 情報源系列 $\boldsymbol{x}$ を先頭の文字から逐次的に決定していくことができる. 上記の説明から, 復号器の出力が符号器への入力の語頭になることは明らかである. したがって, 算術符号は逐次符号に分類することができる.

情報源シンボルの確率分布 p が与えられた場合, 写像 I_s として

$$p(a) = \frac{|I_s(\boldsymbol{x}a)|}{|I_s(\boldsymbol{x})|}, \quad \boldsymbol{x} \in \mathcal{X}^*, a \in \mathcal{X} \tag{2}$$

を満たすものを考えるのが通常である.

2.3 算術符号化・復号化の例

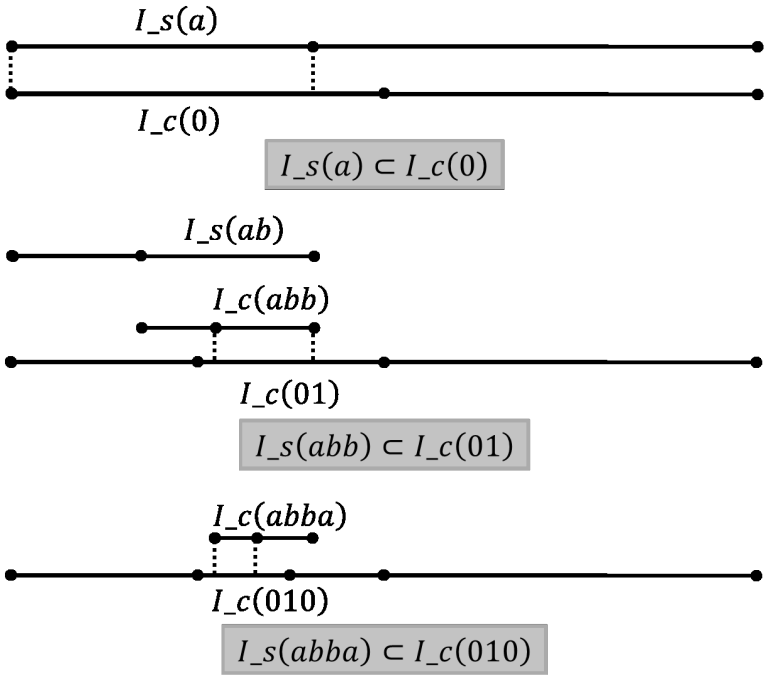

図 2　算術符号の符号化

算術符号の符号化を例をあげて説明していく. 図 2 に沿って説明をしていく. 2 つのシンボル a, b がそれぞれ 0.4,0.6 の確率で生起すると考える. 写像 I_s はシンボルの発生確率によって分割される. 最初に情報源シンボル a が送られてきたとする. a の区間 $I_s(a) = [0, 0.4]$ は符号語 0 の区間 $I_c = [0, 0.5]$ と $I_s(a) \subset I_c(0)$ の関係にあるため, 0 が符号語の語頭として生成さ

れる．2 つ目の情報源シンボルとして b が送られてきたとする．このとき $I_s(ab) = [0.16, 0.4]$ は $I_c(00) = [0, 0.25]$ と $I_c(01) = [0.25, 0.5]$ の区間のどちらにも収まっていないので符号語を生成することができない．そして 3 つ目の情報源シンボルとして b が送られてきたとする．$I_s(abb) = [0.25, 0.4]$,$I_c(01) = [0.25, 0.5]$ であるから $I_s(abb) \subset I_c(01)$ となり符号語の 2 文字目は 1 となる．4 つ目の情報源シンボルとして b が出力されると, $I_s(abba) = [0.256, 0.3136]$ となる．$I_c(010) = [0.25, 0.375]$ であるため, $I_s(abba) \subset I_c(010)$ となり, 符号語の 3 文字目は 0 となる．つまり $abba$ という情報源系列を算術符号で逐次符号化し, 010 という符号語が得られた．

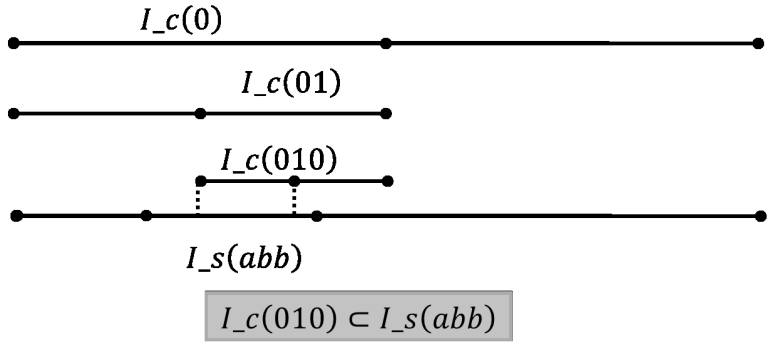

図 3　算術符号の復号化

次に復号化について考えていく．図 3 に沿って説明していく．010 という符号シンボルを復号化して $abba$ というシンボルが得られるかどうか確かめる．まず 1 文字目符号シンボルの 0 をみる．区間 $I_c(0) = [0, 0.5]$ では a か b のどちらかの情報源シンボルの区間かわからない．2 文字目の符号シンボル 1 を含めて $I_c(01) = [0, 0.25]$ を考えてみても判別することができない．3 文字目の符号シンボル 0 を含めて考えると, $I_c(010) = [0.25, 0.355]$ となり, $I_s(abb) = [0.256, 0.4]$ から $I_c(010) \subset I_s(abb)$ となる．abb という 3 つの符号シンボルが一気に復号されることになる．情報源 4 文字目の a が復号されていないが, 符号シンボルをデコーダへ入れていくことで復号することができる．

このように符号化, 復号化の過程で次のシンボルの入力を待たないと符号化, 復号化できない場合があるということがわかる．これがエンコーダ・デコーダでの遅延となって表れる．

2.4　算術符号の演算精度

算術符号を現実的な場合で使用するには, 区間の精度について考えなければならない．現在の計算機アーキテクチャから考えれば, これはすなわち, 区間の端点を何桁の 2 進数で表現するかということである．このとき式 (1) を満たす写像 I_s は一般には存在しない．データ圧縮

の目的からすると, なるべく式 (1) の右辺が左辺に近くなるように, 区間の精度は高いほうが良い.

一方, 区間の状態は逐次符号としての内部状態を表している. したがって, 区間の精度が高ければ, 内部状態として多くの状態をとることができる. さらにこのことは, 内部状態として入力シンボルを多く記憶しておくことができるということを意味するので, 符号化遅延が大きくなるということが予想される. したがって逆に算術符号の精度を下げることで, 符号化遅延を抑制することができると考えられる [2].

その考えのもと, 文献 [2] では精度と平均遅延について研究が行われ, 算術符号の精度を下げることで, 符号化遅延を抑制できるという実証を得た. 本研究ではエンコーダとデコーダの間に送信器があるシステムについても, 算術符号の演算精度を下げることが遅延の抑制に有効であるか実験を行い検証する.

2.5 平均遅延の内訳

図 1 のようなシステム全体の平均遅延は, エンコーダ・デコーダ, バッファ, 送信器での平均遅延を合計したものである. 本研究ではエンコーダ・デコーダでの平均遅延を符号化遅延, バッファ・送信器での平均遅延を, バッファ遅延を呼ぶこととする. つまりシステム全体の平均遅延とは符号化遅延とバッファ遅延の合計である.

エンコーダ・デコーダ間を直結させた全体の平均遅延が符号化遅延であり, 直接計測することが可能である. バッファ遅延は直接計測することができない. そのためシステム全体の平均遅延と符号化遅延の差を取り, バッファ遅延を計測する.

3 符号化遅延の検証

この章では到着の仕方が符号化遅延に与える影響を調べる. そのためにエンコーダ・デコーダを直結させたシステムで実験を行う. 図 1 の送信器部分をなくし, エンコーダ・デコーダを直結させた. このシステムは [2] で考えられたシステムと情報源シンボルの到着の仕方以外は同じものである.

3.1 実験内容

到着には一定間隔での到着とポアソン到着を用いる. 情報源シンボルの到着レートを $\lambda = 1$ とした. そして精度を 2〜11 と情報源のシンボル発生確率を $p = 0.1, 0.2, 0.3, 0.4$ で変化させ, それぞれの確率で 100 万個のシンボルを発生させ, 平均遅延を計測する.

3.2 結果と考察

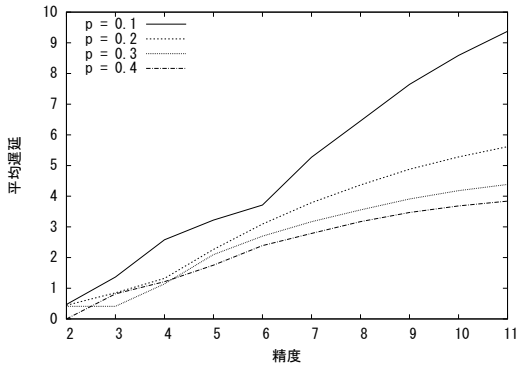

図 4　精度と平均遅延　到着間隔一定

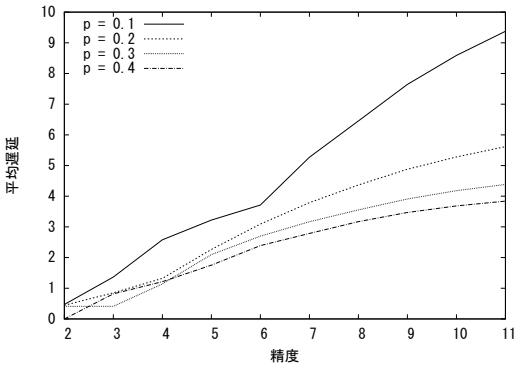

図 5　精度と平均遅延　ポアソン到着

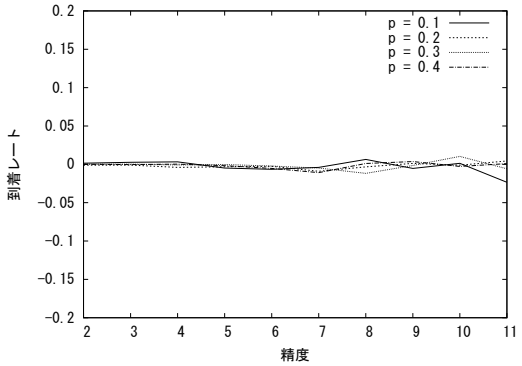

図 6　到着間隔一定 - ポアソン到着

図 4 は到着が一定間隔での到着によるものである．これは文献 [2] の精度と平均遅延についてと同じ実験を行っていることになる．本研究で得られた結果は [2] と同じものである．図 5 は到着がポアソン到着での実験結果である．図 5 と図 4 を比較すると同じ結果が得られているように見える．図 4, 5 二つのグラフの差を取ったものが図 6 となる．図 6 を見ると到着間隔に多少の差は見られるがその値は非常に小さいことがわかる．つまり図 4, 5 の結果はほぼ同じと言える．ポアソン到着は到着がランダムであるものの到着レート [シンボル/秒] は到着間隔一定のものと同じである．非常に多くのシンボルの遅延を計測しているため, ポアソン到着の到着間隔がランダムであっても 1 つ 1 つのシンボルの遅延の差が均される．よって一定間隔のと

きの平均遅延とポアソン到着での平均遅延は, なんら変わらなくなったと考えられる. そこから到着の仕方によって平均遅延が決まるのではなく, 到着レートにより精度と平均遅延の関係が決まると考えられる.

4 バッファ遅延の検証

3 章では符号化遅延について考えたが, この章ではバッファ遅延について詳しく調べる. そこでエンコーダ・デコーダ間に送信器を設置したシステムでの遅延を考える. そしてエンコーダ・デコーダを直結させたシステムでも同様の実験を行い, 2 つの実験の差からバッファ遅延を見る.

4.1 バッファのパンク

バッファに送られてくる符号語のレートに対してバッファから出て行く符号語のレートが小さいとき, バッファにたまる符号シンボルの数が増え続けてしまう. すると, バッファの中での待ち時間が次第に大きくなっていき, 結果として平均遅延が発散してしまう. 本研究ではこの現象をバッファのパンクと呼ぶことにする. 送信器の送信レートが有限の場合, 到着レートとの関係によってはバッファがパンクしてしまう. 一定時間にバッファへ入る符号シンボルの数と, 送信される符号シンボルの数からバッファがパンクしてしまう条件を求める. t 秒間に到着する情報源シンボル数の期待値は $t\lambda$ である. また算術符号の符号化レートを R とすると, バッファに送られる符号化シンボル数は $t\lambda R$ である. よって

$$t\lambda R \leq t\mu \tag{3}$$

を満たせばバッファはパンクしない. これを解くと,

$$\lambda \leq \frac{\mu}{R} \tag{4}$$

が求められる. さらに本実験では送信レート $\mu = 1$ として固定しているため, バッファをパンクさせないためには

$$\lambda \leq \frac{1}{R} \tag{5}$$

を満たせばよい. 式 (5) の範囲の到着レート λ で実験を行う.

4.2 到着レートの正規化

精度を変化させると圧縮効率が変わるため, それに伴い符号化レートも変化する. 式 (5) から情報源の分布が同じであっても, バッファがパンクする到着レートは精度によって異なるとい

うことになる．よって実際の入力シンボルと出力された符号シンボルの数を計測し，それぞれの精度での符号化レートを算出した．式 (5) に従ってバッファがパンクしてしまう到着レートを導きそれを 1 として，それぞれの精度での到着レートを正規化した．この正規化された到着レートを正規化到着レートと呼ぶ．

4.3 実験内容

到着はポアソン到着，到着間隔一定の両方を用いる．送信レートを $\mu = 1$ として固定する．正規化到着レート λ，精度 w，情報源のシンボル発生確率 p をそれぞれ変化させ平均遅延を見る．

4.4 結果と考察

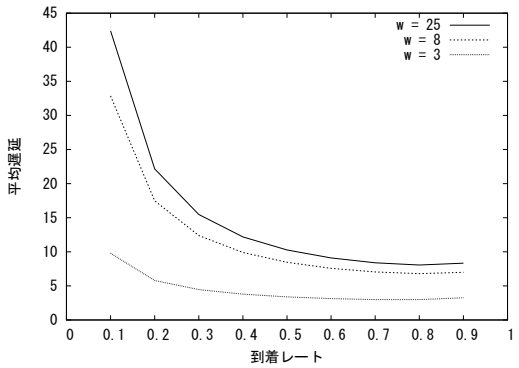

図 7　到着レートと平均遅延　到着間隔一定

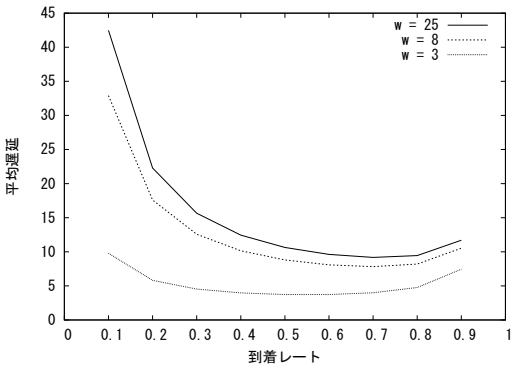

図 8　到着レートと平均遅延　ポアソン到着

図 7 は到着間隔一定，$p = 0.4$, $w = 3, 8, 25$ のときの到着レートと平均遅延である．また図 8 は到着のみポアソン到着になったものである．図 7, 図 8 からどちらを見ても精度を下げることで平均遅延が小さくなっている．システムに送信器を設置した場合でも精度を下げることで平均遅延を小さくすることができるということがわかる．正規化到着レートが 1 に近いところでの平均遅延はポアソン到着のモデルのほうが大きいように見える．

システム全体の遅延と符号化遅延，バッファ遅延の関係を調べるために平均遅延の内訳をみる．

図 9 が到着間隔一定，$p = 0.4$, $w = 3$ のときの平均遅延の内訳である．また図 10 は到着がポアソン到着になったものである．図 9, 10 を比較すると符号化遅延は一致しているように見える．3 章で符号化遅延には到着の仕方が影響してこないことがわかっているので，それらはほぼ一致していると言える．それに対してバッファ・送信器での平均遅延に差が見られる．図

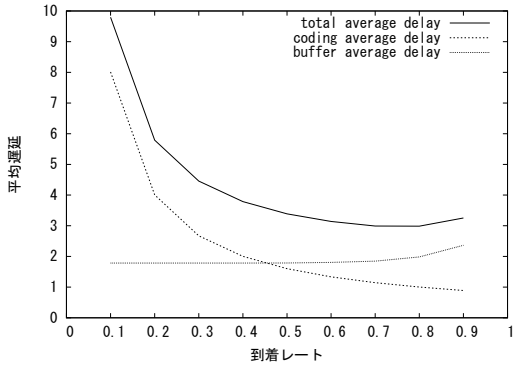

図 9　到着レートと平均遅延　到着間隔一定
平均遅延内訳

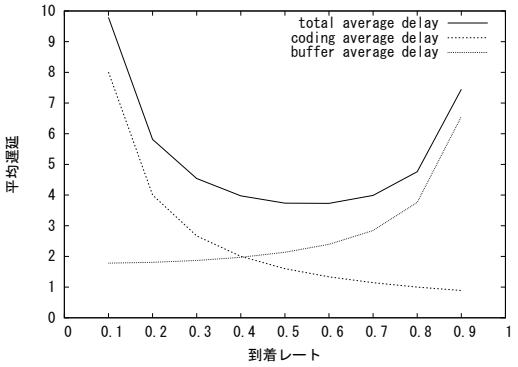

図 10　到着レートと平均遅延　ポアソン到着
平均遅延内訳

9, 10 を見るとどちらも到着レートが 0 に近いところでは符号化遅延が大きい. これはシンボルの到着間隔が長くなるためにエンコーダ・デコーダ内で符号化復号化によって起きる待ちの時間が大きくなり, 遅延が大きくなったためと考えられる. また到着レートが 1 に近いところではバッファ遅延が大きくなっている. これはバッファがパンクする到着レートに近づいたことが原因と言える. つまり到着の仕方に関わらず, 送信レートに対して到着レートが小さいとき符号化遅延に影響を与え, バッファのパンクする到着レートに近いところではバッファ遅延に影響を与える.

次にバッファ遅延を見る.

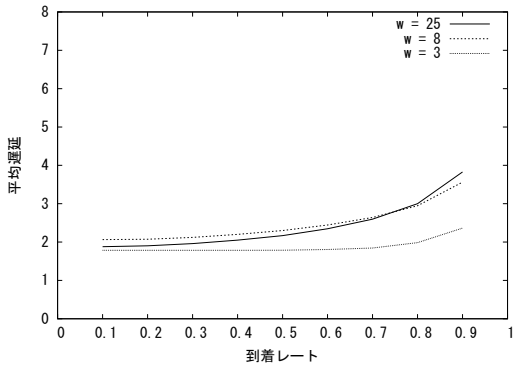

図 11　到着レートと平均遅延　到着間隔一定
バッファ遅延

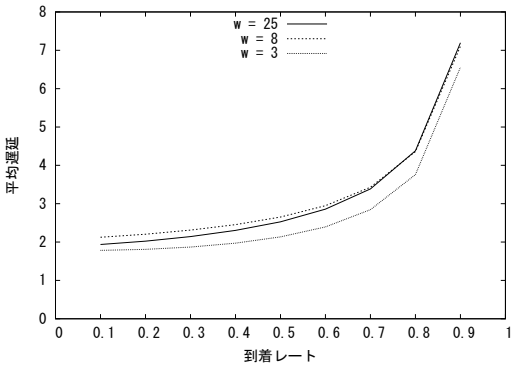

図 12　到着レートと平均遅延　ポアソン到着
バッファ遅延

図 11, 12 がそれぞれ到着間隔一定とポアソン到着の時のバッファ遅延である. 図 11 を見る

と精度を下げることでバッファ遅延も小さくなっていることがわかる．図 12 でも精度を下げたときのバッファ遅延が小さくなっているが, 図 11 ほど小さくなっているわけではない．また図 11, 12 からポアソン到着のほうが到着間隔一定のものよりバッファ遅延が大きくなっている．これらは送信器の送信間隔が一定であることに起因していると考えられる．精度が低ければ符号シンボルがバッファへ入る間隔が到着間隔に近くなる．ここから送信器の送信間隔が一定であるため, ランダムな到着間隔のポアソン到着ではバッファ内で符号シンボルに待ちが生じ, 平均遅延が大きくなったと考えられる．さらに精度を下げてもバッファ遅延があまり小さくならないことの理由もこのことから説明できる．バッファ遅延を抑制するためにはバッファへ入る符号シンボルの間隔と送信間隔を近くすることが必要だと考えられる．

5　状態遷移する情報源

今まで行った実験ではそれぞれ確率を固定して平均遅延を計測していた．しかしより一般には情報のシンボルの発生確率は常に一定とは限らず, たえず変化する．とはいえ情報源シンボルの発生確率が遷移していくシステムでも平均遅延の傾向は確率が一定のものと同じような傾向となると思われる．これを検証するために 4 章と同様の実験を情報源の状態が遷移していくモデルで行う．

5.1　状態遷移

図 13 は情報源の状態遷移を表し, その中の数はシンボルの発生確率を表している．情報源が a を出力した場合は実線, b を出力した場合は破線の向きに状態が遷移する．

5.2　実験内容

到着はポアソン到着を用いた. 送信レート $\mu = 1$ で固定. 情報源のシンボル発生確率は初期値 $p = 0.1$ で上記のように遷移させる．精度 w, 正規化到着レート λ をそれぞれ変化させたときの平均遅延を見る．

5.3　結果と考察

図 14 は $w = 3, 8, 25$ の平均遅延を描画したものである．図 15 は $w = 3$ のときの遅延内訳の図である．

図 14 から精度が下がると平均遅延も小さくなっていることがわかる．図 15 を見ると到着レートが 0 に近いところでは符号化遅延が大きい．また正規化到着レートが 1 に近いところではバッファ遅延が大きい．予想通り, 状態を遷移させたシステムでも 4 章の実験結果と同じよ

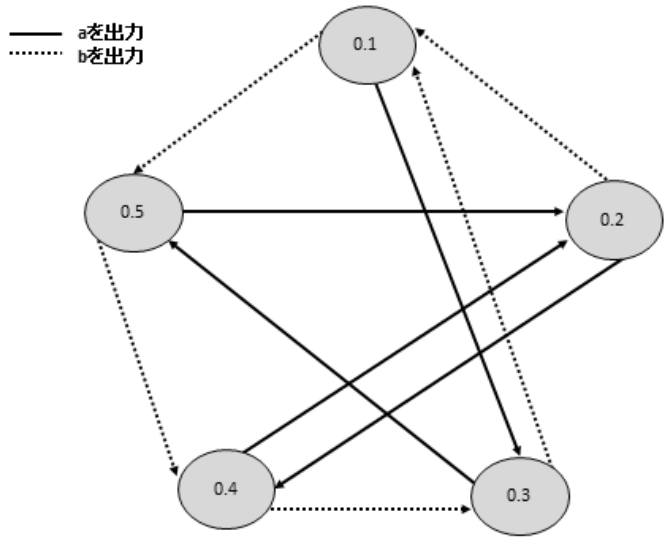

図 13　情報源の状態遷移

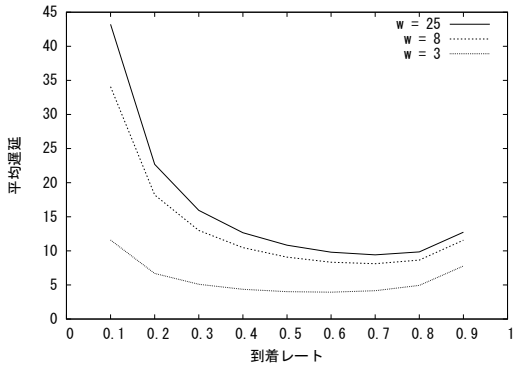

図 14　到着レートと平均遅延　ポアソン到着確率遷移

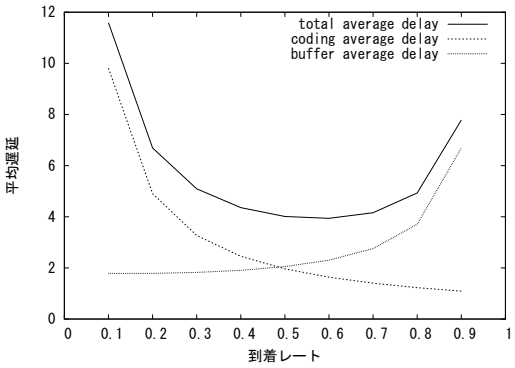

図 15　到着レートと平均遅延　ポアソン到着確率遷移　遅延内訳

うな傾向があることがわかった.

また $p = 0.1, 0.2, 0.3, 0.4, 0.5$ のときの平均遅延の合成から図 14 のグラフと同じ結果を導き出せるのではないかと考えた. 情報源の定常分布を導き表 1 に示した. 表 1 に従い平均遅延を合成する.

情報源の状態	定常分布 [%]
$p = 0.1$	24.66
$p = 0.2$	19.77
$p = 0.3$	12.63
$p = 0.4$	16.95
$p = 0.5$	25.99

表 1　情報源の定常分布

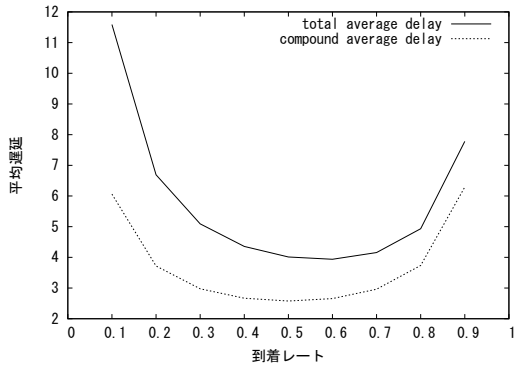

図 16　到着レートと平均遅延　合成による平均遅延

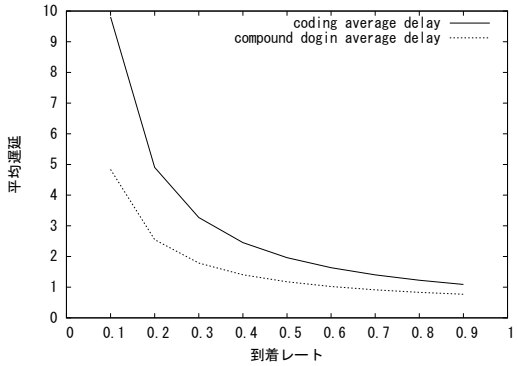

図 17　到着レートと平均遅延　合成による符号化遅延

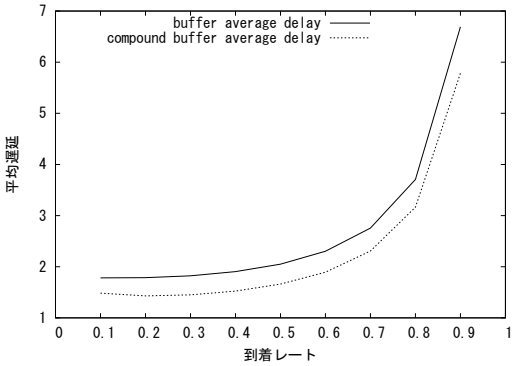

図 18　到着レートと平均遅延　合成によるバッファ遅延

図 14 の $w = 3$ のときのシステム全体の平均遅延と合成によって求めた平均遅延を図 16 に, 符号化遅延と合成による符号化遅延を図 17, バッファ遅延と合成による符号化遅延を図 18 に

描画した.

図 16 を見ると合成したものは全体的に平均遅延が小さい. 図 17, 18 を見ると, 符号化遅延, バッファ遅延のどちらも合成によって求めたものが小さいことがわかる. 情報源の状態が遷移しない時, エンコーダ・デコーダでの符号化・復号化に用いる確率はそれぞれ一定であり, 状態数の組み合わせも一定である. それに対して情報源の状態が遷移する場合, エンコーダ・デコーダでの符号化・復号化に用いる確率はそれぞれ常に変化している. そのため状態数の組み合わせが多くなり, エンコーダ・デコーダで取りうる状態数が多くなると考えられる. とりうる状態数が多いということは符号化遅延が大きくなるということであり, バッファ遅延も大きくなったと考えられる.

6 まとめ

エンコーダとデコーダの間に送信器を入れたシステムでも演算精度を下げることで遅延を小さくできることがわかった. 到着レートが送信レートに対して小さいときに符号化遅延が大きくなり, バッファがパンクしてしまう到着レートに近づくとバッファ遅延が大きくなる. レートが同じであれば到着の仕方が符号化遅延に影響を及ぼすことはない. 送信器の送信間隔と, バッファに入る符号シンボルの間隔が近いものであるほどバッファ遅延が小さくなると考えられる.

情報源の状態を遷移させるとエンコーダ・デコーダで取りうる状態数が多くなり, 結果としてシステムの遅延が大きくなると考えられる. 情報源の状態の定常分布から平均遅延を求めても, 状態を遷移させたモデルとは結果が異なる.

今後の課題として, 最大遅延の検討が挙げられる.

謝辞

本研究を行うに当たり, 丁寧なご指導をして下さった西新幹彦准教授に感謝と敬意の意を申し上げます.

参考文献

[1] 杉原辰徳,「算術符号を用いた伝送システムにおけるポアソン到着シンボルの遅延」, 信州大学工学部, 学士論文, 2008.

[2] 西新幹彦,「算術符号の演算精度と状態数と遅延に対する考察」, The 8th Shannon Theory Workshop (STW13), pp.35–40, Oct. 2013.

[3] 情報理論とその応用学会, 情報源符号化=無歪みデータ圧縮, 培風館,1998.

[4] 宮沢政清, 待ち行列の数理とその応用, 牧野書店, pp.78–79, 2006.
[5] 奥村晴彦, C 言語による最新アルゴリズム事典, 株式会社技術評論社, 1991.

付録 A　実験で用いたシステムのソースコード

以下は実験で用いたエンコーダとデコーダ間に送信器を持たせたシステムのソースコードである. エンコーダ, デコーダのプログラムは文献 [2] から引用した. また一様乱数発生のプログラムは文献 [5] から引用した.

```
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double prob;    /* 確率 p */
int prec; /* 精度 w */
#define HIGHVAL (1 << prec)
#define HLFVAL (1 << (prec - 1))
int low, high; /* 区間 [lowES, highES) */
int low2, high2;
int low_c, high_c; /* [lowDC, highDC) */

double ct;  /* 現在時刻 */
double rambda;  /* 到着レート */
double st;  /* シンボル到着時刻 */
double et;  /* 送信完了時刻 */
double m = 0;   /* 復号シンボル数 */
double D;   /* 遅延 */
double A;   /* 平均遅延 */
double p;   /* 確率 */
double ec = 0;  /*  符号化シンボル数  */
double sc = 0;  /* 情報源から発生したシンボル数 */
double tr = 0;  /* 送信レートの逆数 */

FILE *fp;   /* 出力ファイル */

#define MRND 1000000000L    /* 乱数発生 */
static int jrand;
static long ia[56];
static void irn55(void)
{
    int i;
    long j;
    for (i = 1; i <= 24; i++) {
        j = ia[i] - ia[i + 31];
        if (j<0) j += MRND;
        ia[i] = j;
    }
    for (i = 25; i <= 55; i++) {
        j = ia[i] - ia[i - 24];
        if (j<0) j += MRND;
        ia[i] = j;
    }
}
void init_rnd(unsigned long seed)
{
    int i, ii;
    long k;
    ia[55] = seed;
    k = 1;
    for (i = 1; i <= 54; i++) {
        ii = (21 * i) \% 55;
        ia[ii] = k;
        k = seed - k;
        if (k<0)k += MRND;
        seed = ia[ii];
    }
```

```
    irn55(); irn55(); irn55();
    jrand = 55;
}
long irnd(void)
{
    if (++jrand>55) { irn55(); jrand = 1; }
    return ia[jrand];
}
double rnd(void)
{
    return((1.0 / MRND) * irnd());
}
double exp_rnd(void)
{
    return -(1 / rambda)*log(1 - rnd());
}


#define QUEUE_SIZE 1000000      /* リングバッファ１（シンボルの時刻を入れる）*/

double queue[QUEUE_SIZE];
int front;
int rear;

void
enqueue(double x)
{
    if ((rear + 1) \% QUEUE_SIZE == front) {
        printf("buffer1 is full.");
        exit(1);
    }
    queue[rear] = x;
    rear = (rear + 1) \% QUEUE_SIZE;
    return;
}

double
dequeue()
{
    double x;
    if (front == rear){
        printf("buffer1 is empty.");
        exit(1);
    }
    x = queue[front];
    front = (front + 1) \% QUEUE_SIZE;
    return (x);
}

#define QUEUE_SIZE2 1000000     /* リングバッファ２（シンボルを入れる）*/
char queue2[QUEUE_SIZE2];
int front2;
int rear2;

void
enqueue2(char c)
{
    if ((rear2 + 1) \% QUEUE_SIZE2 == front2) {
        printf("buffer2 is full.");
        exit(1);
    }
    queue2[rear2] = c;
    rear2 = (rear2 + 1) \% QUEUE_SIZE2;

    if (et < 0){
        et = ct + 1;
    }
    return;
}
```

```
char
dequeue2()
{
    char c;
    if (front2 == rear2){
        printf("buffer2 is empty.");
        exit(1);
    }
    c = queue2[front2];
    front2 = (front2 + 1) \% QUEUE_SIZE2;
    return(c);
}

void decode(char symbol);
void encode(char symbol);

void
symbol()
{
    sc += 1;
    ct = st;
    enqueue(ct);
    if (rnd() < p){
        encode('a');
    }
    else {
        encode('b');
    }
    st = ct + exp_rnd();

    if (et <= 0){
        et = ct + tr;
    }
    return;
}

void
transmission()
{
    ct = et;

    if (front2 == rear2){
        et = 0;
    }
    else {
        decode(dequeue2());
        if (front2 != rear2){
            et = ct + tr;
        }
    }
    return;
}

void
encode(char symbol)
{
    int div = (high - low) * p + low + 0.5;
    if (div <= low) div = low + 1;
    if (div >= high) div = high - 1;
    if (symbol == 'a'){
        high = div;
    }
    else {
        low = div;
    }

    for (;;){
        if (high <= HLFVAL){
```

```
            enqueue2('0');
            ec += 1;
        }
        else if (low >= HLFVAL){
            enqueue2('1');
            ec += 1;
            low -= HLFVAL;
            high -= HLFVAL;
        }
        else {
            break;
        }
        low *= 2;
        high *= 2;
    }
    return;
}

void
calc_delay()
{
    double arr = dequeue();
    D = ct - arr;
    //  printf("\%f \%f \%f \n", D ,ct ,arr);
    A = A + D;
    //  if (A < D){
    //      A = D;
    //  }
    return;
}

void
decode(char symbol)
{
    if (symbol == '0'){
        high_c = (low_c + high_c) / 2;
    }
    else {
        low_c = (low_c + high_c) / 2;
    }
    for (;;){
        int div = (high2 - low2) * p + low2 + 0.5;
        if (div <= low2) div = low2 + 1;
        if (div >= high2) div = high2 - 1;
        if (high_c <= div){
            calc_delay();
            m++;
            high2 = div;
        }
        else if (div <= low_c){
            calc_delay();
            m++;
            low2 = div;
        }
        else {
            break;
        }
        for (;;){
            if (high2 <= HLFVAL){
                ;
            }
            else if (low2 >= HLFVAL){
                low2 -= HLFVAL;
                high2 -= HLFVAL;
                low_c -= HLFVAL;
                high_c -= HLFVAL;
            }
            else {
                break;
```

```
            }
            low2 *= 2;
            high2 *= 2;
            low_c *= 2;
            high_c *= 2;
        }
    }
    return;
}


int
main(void)
{

    int w;
    int l;
        for (w = 2; w <= 11; w++)
//  for (l = 1; l <= 10; l++)
    {
        p = 0.1;
        ec = 0;
        sc = 0;
        rambda = 1;
        prec = w;
        high = HIGHVAL;
        low = 0;
        high_c = HIGHVAL;
        low_c = 0;
        high2 = HIGHVAL;
        low2 = 0;
        ct = 0;
        st = (1 / rambda);
        et = 0;
        D = 0;
        A = 0;
        ec = 0;
        front = rear = 0;
        front2 = rear2 = 0;

        init_rnd(2092);

        //  fp = fopen("delay.txt", "wt");
        m = 0;
        while (m < 1000000){
            if ((et <= 0) || st < et){
                symbol();
            }
            else {
                transmission();
            }
        }
        printf("\%f\n", A / 1000000);
        //      printf("\%f\n", ec / sc);
    }
    //  fclose(fp);

    return(0);
}
```