信州大学工学部

学士論文

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

指導教員　西新　幹彦 准教授
学科　電気電子工学科
学籍番号　03T2047B
氏名　杉原　辰徳

2008 年 2 月 27 日

目次

1　序章

1.1　背景

私たちの生活に欠かす事のできないインターネットなどの情報通信システムにおいてそのデータ量は莫大になってきている．これらの莫大なデータを, そのまま伝送することは不可能である．この莫大なデータを伝送する際に, 誤りがなくかつ効率よく伝送するにはどうすべきか．本研究ではデータを圧縮しかつ遅延を最小限にすることにより最適にデータを伝送することを考えた．符号化する方法はいくつかあるが, その中で, エンコーダに入力シンボルが到着しつつ符号化し, デコーダはエンコーダから符号語を受け取りながら復元することができれば, リアルタイムなデータ伝送ができる．それを実用的に実現することのできる一つに算術符号があり, これにより遅延も最小限に留めることができるのではないかと考えた．本研究は, データがランダムに送られてくるというシステムを考察の対象としている．例としては, 降水量や交通量の測定などであるが, 本研究ではそのような不規則なデータ送信一般を簡単化して考える．まずはじめに本研究の概要について述べる．

1.2　研究の概要

本研究で考えた算術符号のシステムのモデルを図 1 に示した．本研究において情報源は, 情報源シンボル 0,1 をもち, レート λ のポアソン到着でシンボルを出力する．出力されたシンボルは算術符号によるエンコーダに入力される．そこでは, 情報源シンボルの出現確率を使って符号語を生成し圧縮する．これらの符号語は送信バッファに貯められ, そこからレート μ でデコーダに送られる．すなわち 1 符号シンボルを送信するのに要する時間は $1/\mu$[秒] である．デコーダは, 符号語を受け取りながら復元していく．つまりエンコーダに全ての入力を待たずしてデコーダは復元を開始することができるのである．

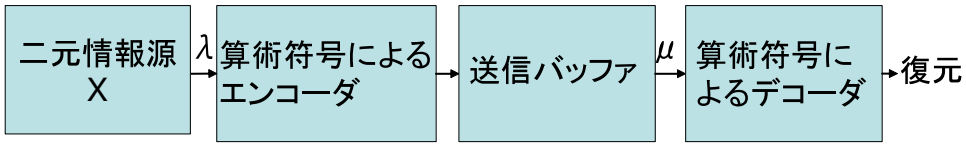

図 1　算術符号のシステム

本研究における遅延とは, 一つのシンボルがエンコーダに入力されて符号語となり送信バッファに送られてまたここからデコーダに送られシンボルが復元されるまでに要する時間をいう．入力シンボルの到着レート λ が送信バッファレート μ より大きいと送信バッファ内に符号語が蓄積されてここで遅延が生じてしまう. そして送信バッファに到着する符号語が多すぎると最終的には送信バッファに符号語が入りきらなくなってしまう．本研究においてこの状態をパンク状態と定義する．このようなことを考慮して遅延を小さくするための入力シンボルの到着レート λ と送信バッファレート μ の関係を知ることが本研究の目標

である.

算術符号の遅延を測定する実験としてまず, 実験 1 では本研究で使用するプログラムの動作確認を目的に, 入力シンボルを等間隔に入力し, 送信バッファレート $\mu = \infty$ とすることによって算術符号における遅延の分布を求めて既存のデータ [1] と比較した.

実験 2 では, 入力シンボルの到着レート λ と送信バッファレート μ を変化させて遅延の分布および平均遅延を求めた. この際に λ と μ を共に 2 倍に変化させたときに, 平均遅延が $1/2$ になった.

実験 3 では, 実験 2 の結果を踏まえて平均遅延と入力シンボルの到着レート λ と送信バッファレート μ の関係について調べた.

実験 4 では, 送信バッファがパンクする様子をグラフに示すために送信バッファのパンクする条件式を求めてこれを元に検証を行った.

実験 5 では, 入力シンボルの到着レート λ を大きくすれば平均遅延は小さくなるのかということを検証するために送信バッファレート μ を固定して入力シンボルの到着レート λ を変化させて平均遅延を求めた. この結果より送信バッファにおける遅延を考慮しなければならないことがわかった.

実験 6 では, 実験 5 の結果を踏まえて μ が既に設定されている伝送システムを仮定して平均遅延が最小となる λ を求めることを目的とした. 送信バッファレートを $\mu = 1$ に固定し, 入力シンボルの到着レート λ を変化させたときの送信バッファにおける遅延, 算術符号における遅延, および全体の遅延を求めて最も平均遅延が小さくなる入力シンボルの到着レート λ を求めた.

実験 7 では, 送信バッファ内における遅延の分布を調べた.

2 算術符号について

本研究で用いる符号化の方法である算術符号について説明する.

2.1 算術符号とは

算術符号 [2] とは, 実数軸上の半開区間 [0,1) 内の有限小数に対応した符号語で構成された, 無歪みデータ圧縮符号である. 算術符号においては, 入力データ系列が与えられるごとに, その系列に対応した符号語が, 数の四則演算と大小比較によって求められるので, あらかじめ符号語全体の表を用意する必要がない. さらに符号化, 復号化に要する計算量がデータ長に比例すること, 入力データから確率構造を推定しつつ符号化を行う適応符号化に適していることなど実用上優れた特長をもつために, 現在算術符号は画像やテキストデータなどの圧縮に広く用いられるようになってきている. もともと算術符号は, 閉区間を確率の比に応じて再帰的に分割するという Elias によるアイデアを出発点として, 区間を分割する際に必要な計算誤差のない固定桁の整数演算に帰着させることによって完成した.

2.2 算術符号の符号化、復号化の例

2.2.1 算術符号の符号化

算術符号は, 記号列を実数と 0 と 1 の区間を用いて表す. 以下の例では, 記号は {0,1} の 2 種類があり, 出現確率はそれぞれ 0.2,0.8 とする. 算術符号は, 区間を記号の出現確率に比例した小区間に分割していくことで符号化を行う. ここで, 記号列 01111 を符号化する. この過程を図 2 に示す. x 以上 y 未満の区間を $[x, y)$ と表すこととする. 区間の初期値は [0,1)

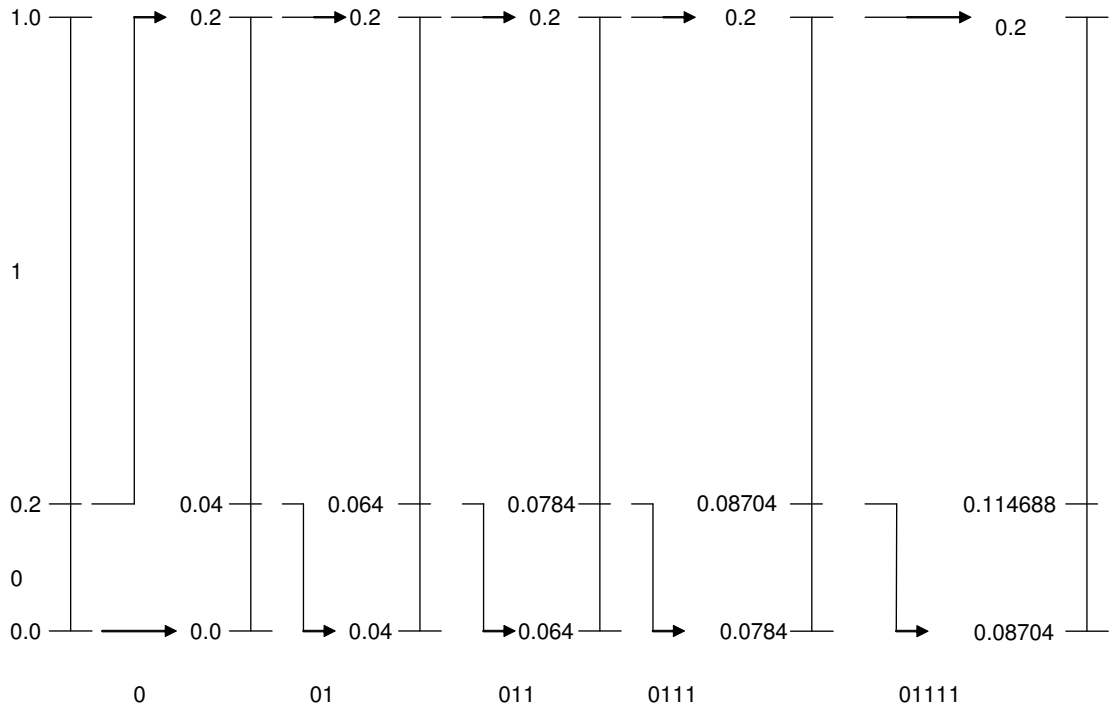

図 2　算術符号の符号化の過程

である. 記号を読み込んだら区間 [0,1) を分割する. 記号が 0 ならば区間の 0 から 0.2 までの部分, 1 ならば 0.2 から 1.0 までの部分に分割する.

最初の記号は 0 なので区間は [0,0.2) である. 次の記号は 1 なので, 区間 [0,0.2) の 0.2 から 1.0 の部分 [0.04,0.20) が新しい区間である. このように, 記号を読み込むたびに区間を分割していくと, 記号列 01111 を表す区間は [0.08704,0.2) となる. そして, 実際の算術符号の符号語は, この区間に含まれる一つの実数を指定する.

ここで符号語を 2 進数で表して, 区間内で小数点以下のビット数の少ない値を選ぶことにする. 例えば 0.1171875 を 2 進数で表すと次のようになる.

$$0.1171875 = 1/16 + 1/32 + 1/64 + 1/128 = (0.0001111)_2$$

0001111 の 7 ビットを符号語として出力すれば, 記号列 01111 の 5 文字を 7 ビットに圧縮することができるわけである. この例では 1 文字が 1.4 ビットに圧縮されたが, 記号の出現確率によっては 1 文字が 1 ビット未満に圧縮できる場合もある.

2.2.2 算術符号の復号

次に復号について説明する. ここでは説明の都合上, 符号語は下限値の 0.08704 とする. 0.08704 は,[0,0.2) の間にあるので, 最初の記号は 0 であることがわかる. 次に 0 を表す区間 [0,0.2) を [0,1.0) になるように拡大すると, 符号語は次のように変換できる.

$$\begin{aligned} \text{新しい符号語} &= (\text{符号語} - \text{記号の下限値})/\text{記号の区間幅} \\ &= (0.08704 - 0)/0.2 \\ &= 0.4352 \end{aligned}$$

新しい符号語 0.4352 は [0.2,0.8) の間にあるので、次の記号は 1 であることがわかる. このような操作を繰り返すことにより記号列 01111 を復号することができる.

3 ポアソン到着について

ポアソン到着とは, ランダムに到着するシンボルを表す確率過程のひとつである. ポアソン到着のシンボルの到着間隔は指数分布にしたがう.

3.1 ポアソン到着の定義と基本的性質

ポアソン到着 [3][4] は, シンボルの到着が次の 3 つの条件を満たすとして仮定した 1 つの到着の仕方のモデルである. 到着間隔は指数分布に従う. ポアソン到着は以下の性質をもつ.

- 定常性
  $t > 0, x \geq 0$ なる任意の実数 t, 任意の整数 x に対し, 時間間隔 $(a, a+t)$ の間に x 人到着する確率は, 全ての a について同一で,$v_x(t)$ と表すことができる. つまり, 時間区間の幅が同じであれば, どこをとってもシステムの確率的状態は同じである. この性質を定常性という.
- 独立性
  上記に記した $v_x(t)$ は, 時刻 a までにどれだけきたか, またいつきたかには無関係である. もし前に到着したシンボルの時刻や個数によってこの確率 $v_x(t)$ が影響を受けるならば, 前のシンボルはあとに残る影響, つまり残留効果をもつことになるが, そういうことがないというのが, この独立性である.
- 希少性
  $\psi(t) = \sum_{x=2}^{\infty} v_x(t)$ と置くとき,t が十分小さければ $\psi(t)$ は無視してよい. つまり,

$$\psi(t) = o(t)$$
$$(t \to 0)$$

が成り立つ. この式は,

$$\lim_{t \to 0} \frac{\psi(t)}{t} = 0$$

を意味する. $o(t)$ とは t が十分小さければ無視してよい程度であることを示している. つまり $\psi(t) = o(t)$ は,2 つ以上のシンボルが同時に連れだってこないことを意味する.

4 システムにおける遅延

4.1 実験用プログラム

本研究における入力シンボルの数は 2^{20} 個である. 2 つのシンボル 0,1 をそれぞれ確率 p, $1-p$ で発生させるために Knuth 法で $[0,1)$ 上の一様乱数 u を発生させ, $u < p$ ならば 0, $u \geq p$ ならば 1 を発生させた. また, パラメータ λ の指数分布は密度関数 $f(x) = \lambda e^{-\lambda x}$ に従う. よって分布関数は $F(x) = \int_0^x f(t)dt = 1 - e^{-\lambda x}$ となる. これより, 逆関数 $F^{-1}(u) = -\frac{1}{\lambda}\log_e(1-u)$ の u に $[0,1)$ 上の一様乱数を代入することで指数分布を得る. 一様乱数は Knuth 法で発生させる.

遅延については, 各情報源シンボルがエンコーダに入力された時刻を記録しておき, デコーダにより復元された時刻との差をとることにより測定している.

4.2 実験用プログラムの動作確認

本研究で実装したプログラムの動作確認として既存のデータ [1] との比較をした.

4.2.1 実験内容

入力シンボルの到着レート λ をポアソン到着ではなく 1 秒間に 1 シンボルという等間隔に設定し, 送信バッファのレート $\mu = \infty$ として実験を行った. 送信バッファのレート μ を ∞ とするということは, 送信バッファの存在を無視すると仮定したことと同等である. つまり, エンコーダに入力されたシンボルは符号語となり直接デコーダに送られ, 算術符号における遅延が観測される. ここでは, 入力シンボルの出現確率を $p = 0.1, 0.2, 0.3$ とする. 測定した遅延は, 整数値となる.

4.2.2 結果と考察

図 3 に遅延の相対頻度を示す.

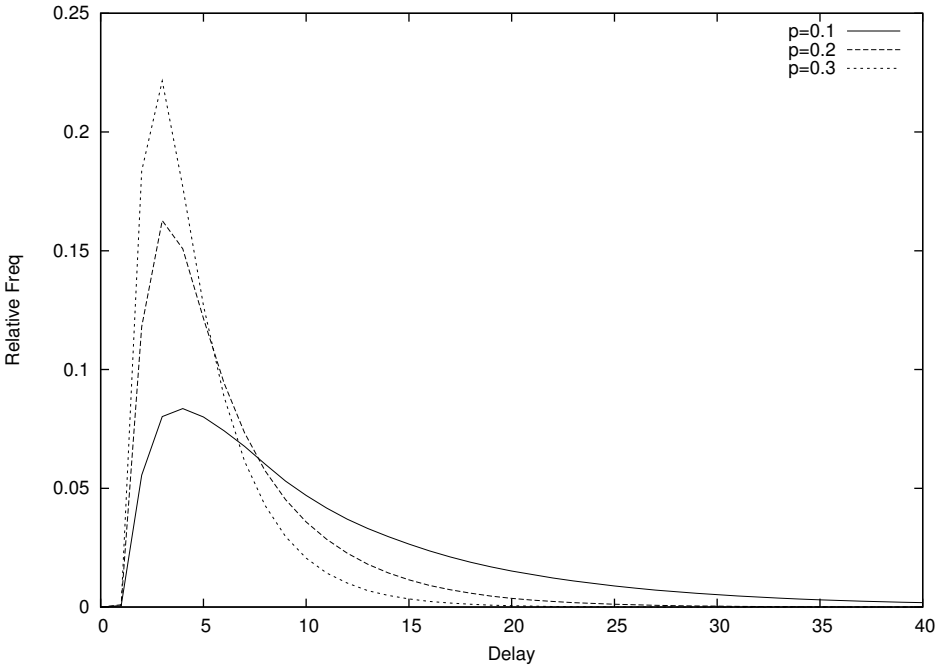

図 3　遅延の相対頻度 1

図 3 に示した遅延の相対頻度のグラフは, 既存の研究結果 [1] と同等である. このことより実装したプログラムに大きな間違いはないということが確認できた.

図 3 より分かる事は, 入力シンボルの出現頻度 $p = 0.1$ の時が他と比べて最も平均遅延が大きいということである. この原因は, エントロピーが他と比べて最も小さいからであると考られる.

4.3　処理速度と遅延の関係

入力シンボルの到着レート λ と送信バッファのレート μ を共に大きくすることは, システム全体の処理速度を早めたことと同等になるのではないかということを確認することを目的に, 入力シンボルの到着レート λ と送信バッファのレート μ を変化させた際の平均遅延のついて計測した. この際は, 入力シンボルの出現頻度を p=0.1 とした.

4.3.1　実験内容

入力シンボルの到着レート $\lambda = 0.1$, 送信バッファのレート $\mu = 1.0$ の場合と $\lambda = 0.2$, $\mu = 2.0$ の場合の平均遅延を求めてグラフにして比較した.

4.3.2　結果と考察

図 4 に, 遅延の相対頻度を密度関数とした表したグラフを示す. 遅延は少数点以下を切り捨てて正の整数をとるとする.

図 4 のグラフを見てみると入力シンボルの到着レート $\lambda = 0.1$ と送信バッファのレート

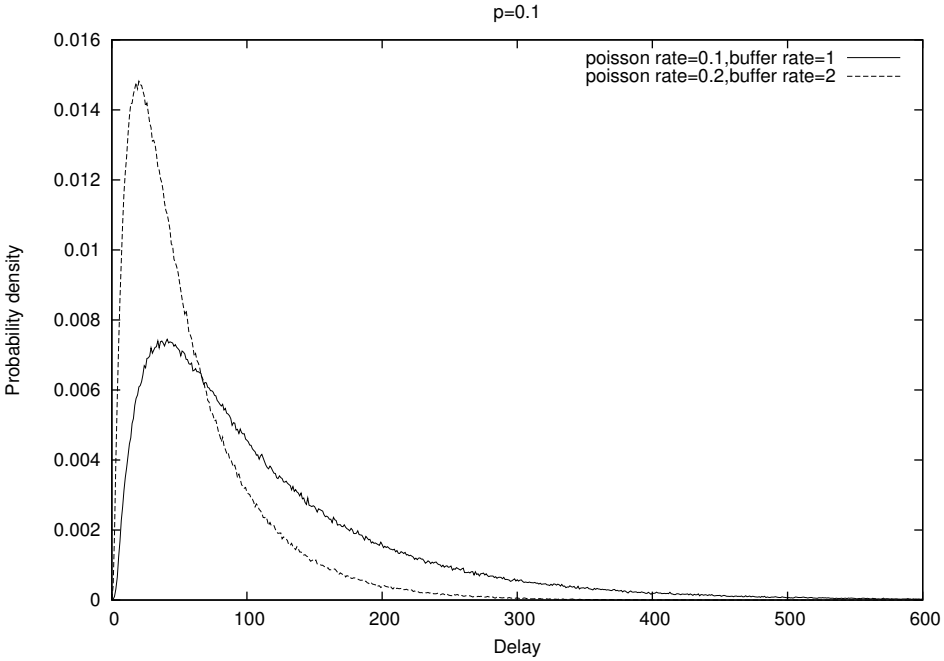

図 4　遅延の相対頻度 2

$\mu = 1$ の時に比べて入力シンボルの到着レート $\lambda = 0.2$ と送信バッファのレート $\mu = 2$ の時のグラフの方が, 遅延の確率密度が全体的に小さくなっているのではないかと推測される. 結果を表 1 に示す.

表 1　λ と μ と平均遅延

λ	μ	平均遅延
0.1	1.0	114.38
0.2	2.0	56.94

入力シンボルの到着レート $\lambda = 0.1$ と送信バッファのレート $\mu = 1$ の時の平均遅延に比べて, 入力シンボルの到着レート $\lambda = 0.2$ と送信バッファのレート $\mu = 2$ の時の平均遅延の方が $1/2$ になっているのではないかと推測した. 入力シンボルの到着レートと送信バッファのレートを同時に大きくするということは, システム全体の動きを早くするということと同等であると考えた. そして次の実験で, 入力シンボルの到着レートと送信バッファのレートと平均遅延の関係について反比例の関係にあるのかを確認する.

4.4　入力シンボルの到着レートと送信バッファのレートと平均遅延の関係

処理速度と遅延の関係の実験の内容を踏まえて, 平均遅延と入力シンボルの到着レート λ と送信バッファのレート μ の関係が反比例にであるかについて調べた.

4.4.1 実験内容

入力シンボルの出現頻度 p=0.1 として, 入力シンボルの到着レート λ の値を, 0.05, 0.1, 0.15, 0.2, 0.25 と変化させてその際の平均遅延を求めて逆数をとりグラフを描いて関係について調べた. $\lambda \to 0$ の際は入力間隔が限りなく広くなるために遅延は ∞ となりグラフが原点を通るのは明らかなので計測は行わなかった. グラフについては, 横軸に入力シンボルの到着レート λ, 縦軸に平均遅延の逆数をとることにした. また送信バッファのレート μ の値については, 入力シンボルの到着レート λ との比によって決定される値,$\mu = 5\lambda$, $\mu = 10\lambda$, $\mu = 20\lambda$, $\mu = 30\lambda$ とする.

4.4.2 結果と考察

図 5 に平均遅延と入力シンボルの到着レート λ と送信バッファのレート μ の関係について求めた結果をグラフにして示した.

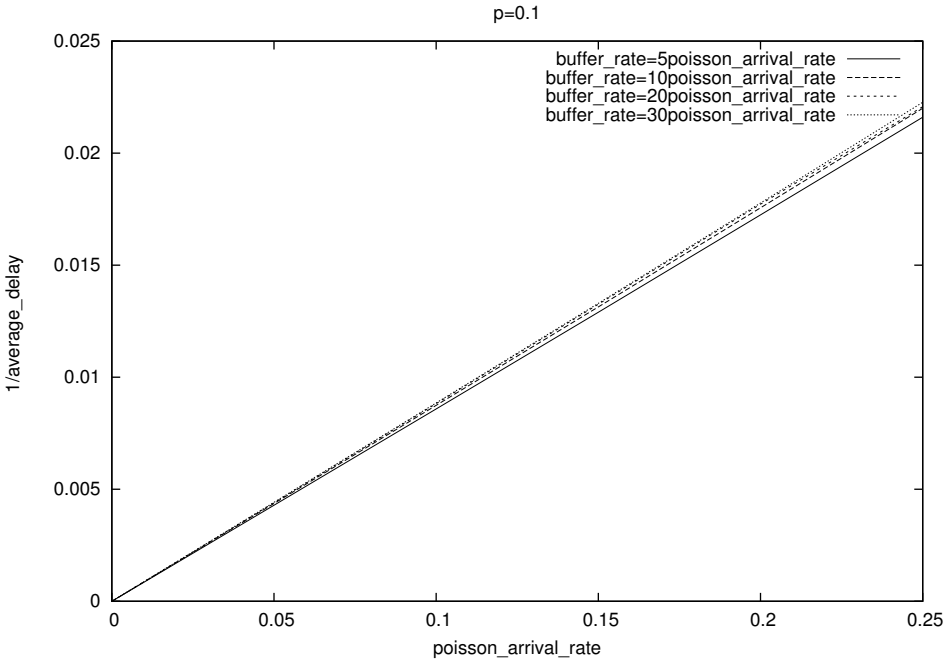

図 5　平均遅延と入力シンボルの到着レート λ と送信バッファのレート μ の関係

縦軸に平均遅延の逆数をとったのは, 入力シンボルの到着レート λ と送信バッファのレート μ と平均遅延の関係を求める際に入力シンボルの到着レート λ と送信バッファのレート μ を共に 2 倍すると平均遅延は 1/2 倍になるという反比例の関係を実証するために有効な手段であるためである. 結果は全ての比において右上がりの直線のグラフとなり, 平均遅延と入力シンボルの到着レートと送信バッファのレートの関係は, 反比例の関係になることがわかった. また以降の実験において λ と μ のどちらか一方を固定して実験を行っても一般性を損なうことがないということがわかった.

4.5 送信バッファのパンク状態

送信バッファがパンクする条件について考察し実験によって確認した.

4.5.1 実験内容

時刻 0 から t において, 入力シンボルの到着レート λ とすると到着したシンボル数の期待値 k は, $k = t\lambda$ となる. ここで到着したシンボル列を, $x_1, x_2, \cdots, x_k$ とすると算術符号の性質より符号語長 $l_k(x_1, x_2, \cdots, x_k)$ は, 到着したシンボル列のエントロピーに近似することができる. つまり,

$$l_k = l_k(x_1, x_2, \cdots, x_k) \simeq H(x_1, x_2, \cdots, x_k) = kH(x) \tag{1}$$

となる. エンコーダから送信バッファに送られるレートは,

$$\frac{l_k}{t} = \frac{\lambda}{k} l_k \simeq \lambda H(x) \tag{2}$$

となる. つまり,

$$\lambda H(x) > \mu \tag{3}$$

ならば, 送信バッファの中身は増加していくので,$t \to \infty$ で, システムがパンク状態になる. 以上のことを考慮して, 横軸に送信バッファのレート μ をとり縦軸に平均遅延をとるグラフを描いてパンク状態となることを確認する. 入力シンボルの出現頻度 p=0.1 とし入力シンボルの到着レートを $\lambda = 2$ とした.

4.5.2 結果と考察

図 6 に送信バッファの中身が増加してパンク状態になる様子をグラフで示す. グラフを見てわかる通り, 送信バッファのレートを小さくしていくと, 式 (3) が示すように入力シンボルのエントロピー 0.48 と入力シンボルの到着レートを $\lambda = 2$ を代入することにより求まる $\mu = 0.96$ 付近では平均遅延が急激に増加していることが観測された. つまりこれは, 平均遅延が無限大まで大きくなり送信バッファがパンク状態であるということを示している.

4.6 入力シンボルの到着レート λ を変化させた際の遅延

平均遅延を小さくするには送信バッファがパンクしない範囲の中で λ を大きくすればよいのではないかと考えた. このことを確認することを目的に送信バッファのレート μ を固定して入力シンボルの到着レート λ を変化させて平均遅延を求めた.

4.6.1 実験内容

送信バッファのレート $\mu = 1.0$ として入力シンボルの到着レート $\lambda = 0.1, 0.5, 1.0$ とした. そしてその 3 つのグラフそれぞれの平均遅延を求めた. 入力シンボルの出現頻度を p=0.1

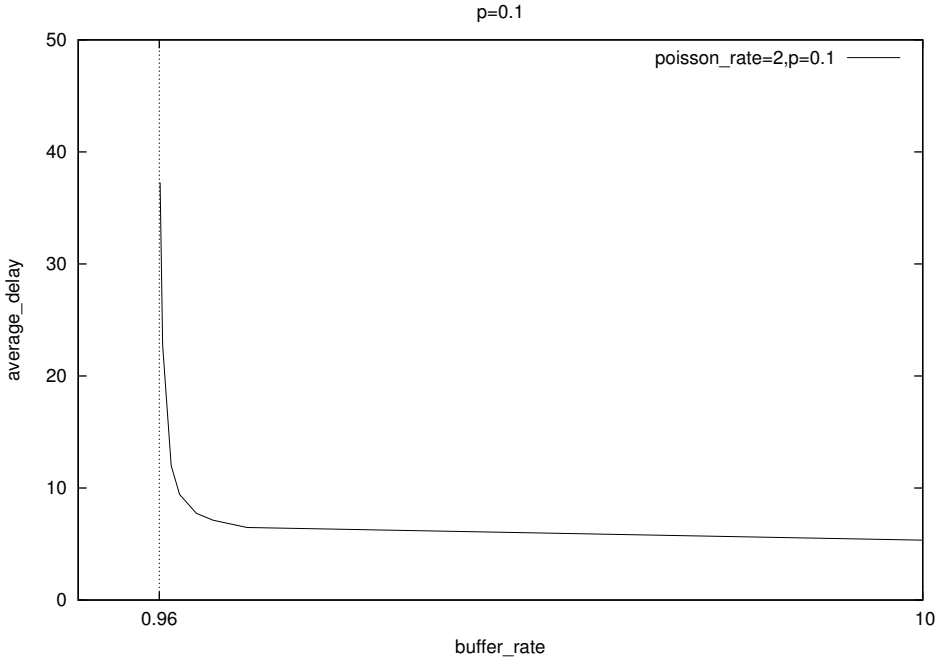

図 6　送信バッファのレートと平均遅延の関係

とした.

4.6.2　結果と考察

図 7 に送信バッファのレート μ を固定して入力シンボルの到着レート λ を変化させた際における遅延と相対頻度を密度関数として表したグラフを示す.

またそれぞれの平均遅延の結果を表 2 に示す.

表 2　λ と $\mu = 1.0$ と平均遅延

λ	μ	平均遅延
0.1	1.0	114.38
1.0	1.0	11.89
2.0	1.0	22.75

平均遅延を小さくするには λ を送信バッファがパンクしない範囲の中で大きくすればいのではないかと考えたが, $\lambda = 2.0$, $\mu = 1$ の際の平均遅延が 22.75 と $\lambda = 1.0$, $\mu = 1$ の時の 11.89 より大きくなってしまった. つまりこれは入力シンボルの到着レートを大きくしすぎたことにより送信バッファにシンボルが溜まってしまいそこで遅延が発生してしまったためと考えられる. また表 2 の結果は, 遅延の最小値の存在を示唆しているといえる.

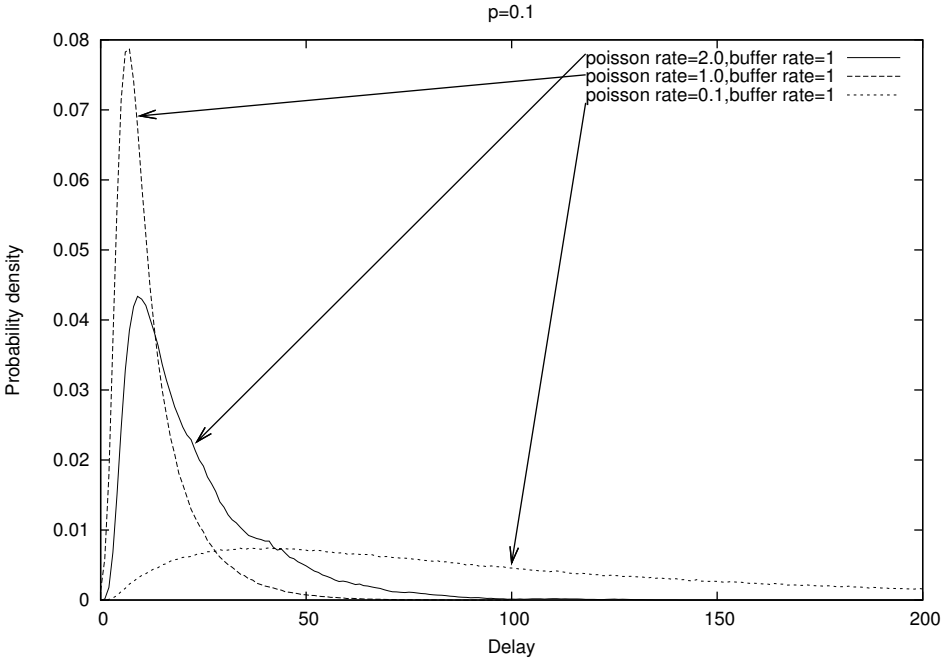

図 7　送信バッファのレート $\mu = 1.0$ とした際に入力シンボルの到着レート λ を変化させた時の遅延と相対頻度を密度関数として表したグラフ

4.7　遅延の最小値と遅延増加の原因

入力シンボルの到着レートを大きくしすぎたことにより送信バッファにおける遅延が発生した. 平均遅延を小さくするためには送信バッファにおける遅延も考慮しなければならないと考えた. 全体の遅延は算術符号における遅延と送信バッファにおける遅延の和であるので, 送信バッファにおける遅延を求めるには,

$$\text{送信バッファにおける遅延} = \text{全体の遅延} - \text{算術符号における遅延} \tag{4}$$

となる. 送信バッファにおける平均遅延を求め, また全体の遅延を求めることにより平均遅延が最小となる値が存在することを確認すると共に平均遅延が最小となる λ を求めることを目的とした.

4.7.1　実験内容

送信バッファレート $\mu = 1.0$ で式 (4) を用いて λ を変化させた際のシステム全体の平均遅延から, λ を変化させたときの算術符号における平均遅延を引くことにより送信バッファにおける平均遅延を求めてグラフにした. 算術符号における遅延は, 4.2 節と同様に $\mu = \infty$ とすることで計測した.

4.7.2 結果と考察

図 8 に送信バッファにおける平均遅延と算術符号における平均遅延および全体の平均遅延のグラフを示した. グラフより $\lambda = 1.3$ で最小の平均遅延を得た. 考察として, まず $\lambda \to 0$ の場合を考える. このとき全体の平均遅延は ∞ となる. これは算術符号における平均遅延が ∞ となることからもわかる. 一方, 送信バッファにおける平均遅延は小さくなる. 次に λ を大きくする場合を考える. 理論的には $\lambda \to \infty$ とすると算術符号における平均遅延は 0 に収束する. 一方, 送信バッファにおける平均遅延は送信バッファのパンク条件式 (3) より,$\lambda = 2.1$ で,∞ に発散する. よって全体の平均遅延も発散する.

以上のことより, どのような情報源に対しても平均遅延の最小値が存在するということがわかった.

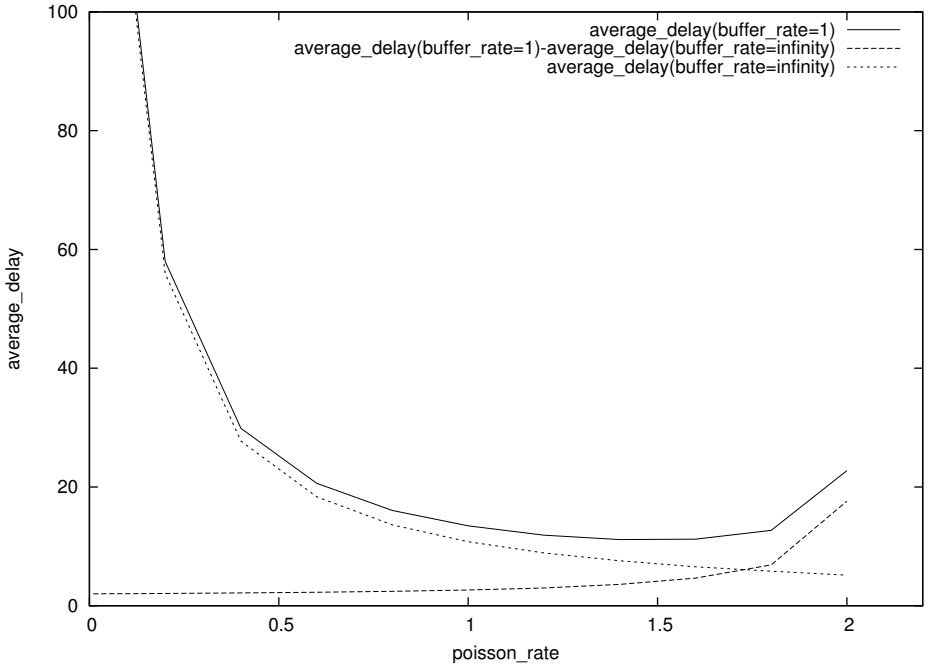

図 8　送信バッファにおける平均遅延と送信バッファレート $\mu = 1$ の時の全体の平均遅延

4.8 送信バッファにおける遅延の分布

送信バッファにおける遅延の分布について調べる. 送信バッファにおける遅延は直接計測することができないので図 8 を参考にする必要がある.

4.8.1 実験内容

送信バッファにおける遅延の分布を調べるにあたりそれぞれの期待値と分散を求めた. 算術符号における遅延および全体の遅延の期待値, 分散は実測値である. 送信バッファにおける遅延は実測値である上記の差をとることにより求める. その結果を表 3 に示す.

表 3　遅延の期待値と分散 ($\lambda = 0.1, \mu = 1.0$)

	期待値	分散
算術符号における遅延	112.35	11046.59
送信バッファにおける遅延	2.03	73.18
全体の遅延	114.38	11119.77

この表 3 の結果からもわかるように送信バッファにおける遅延の分散は 73.18 と他の分散に比べて小さいことから送信バッファにおける遅延の分布は 2.03 付近のインパルスになるのではないかと予想される. 送信バッファにおける遅延の分布は直接計算することが困難なので, 全体の遅延と算術符号における遅延から間接的に求める. 以下にその方法を説明する.

ここで X を算術符号における遅延, Y を送信バッファにおける遅延, Z を全体の遅延とすると式 (4) より $Z = X + Y$ が成り立つ. X, Y, Z の分布をそれぞれ $P_X(t)$, $P_Y(t)$, $P_Z(t)$ とする. すると,

$$
\begin{aligned}
P_Z(t) &= \Pr\{Z = t\} \\
&= \int \Pr\{X = \tau, Z = t\} d\tau \\
&= \int \Pr\{X = \tau, X + Y = t\} d\tau \\
&= \int \Pr\{X = \tau, Y = t - \tau\} d\tau \\
&= \int \Pr\{X = \tau\} \Pr\{Y = t - \tau\} d\tau \\
&= \int P_X(\tau) P_Y(t - \tau) d\tau \\
&= P_X(\tau) * P_Y(\tau)
\end{aligned}
$$

となる. 5 番目の等号で, X と Y は独立であると仮定している. 以上に式に示したように全体の遅延の分布 $P_Z(t)$ は, 算術符号における遅延の分布 P_X と送信バッファにおける遅延の分布 P_Y のたたみ込み積分で表すことができる.

そして, 全体の遅延の分布 $P_Z(t)$, 算術符号における遅延の分布 P_X, 送信バッファにおける遅延の分布 P_Y のフーリエ変換をそれぞれ,

$$
\begin{aligned}
P_Z(t) &\longleftrightarrow p_Z(\omega) \\
P_X(t) &\longleftrightarrow p_X(\omega) \\
P_Y(t) &\longleftrightarrow p_Y(\omega)
\end{aligned}
$$

とおくと,

$$p_Z(\omega) = p_X(\omega)p_Y(\omega)$$

すなわち,

$$p_Y(\omega) = \frac{p_Z(\omega)}{p_X(\omega)}$$

が成り立つ.

観測によって $P_Z(t)$,$P_X(t)$ を求め,$p_Z(\omega)$,$p_X(\omega)$ を計算した後, $p_Y(\omega)$ を逆フーリエ変換することにより送信バッファにおける遅延の分布 P_Y を求める.

4.8.2 結果と考察

図 9 にフーリエ変換によって求めた送信バッファにおける遅延の分布を示す. グラフはジンク関数 $\mathrm{sinc}(x) = \frac{\sin x}{x}$ のように見える. 送信バッファにおける遅延の分布はインパルスであると予想を立てていた. しかし周波数領域においてフーリエ逆変換の際の誤差を除くために高周波成分を取り除いたことによりインパルスではなくジンク関数となったのではないかと考えられる. つまりフーリエ変換の際の誤差が生じなければ送信バッファの遅延の分布はインパルスになったと考えられる. また遅延が 2 の付近で最も確率密度が大きくなっていることも表 3 の送信バッファの期待値が 2.03 であることから確認することができた.

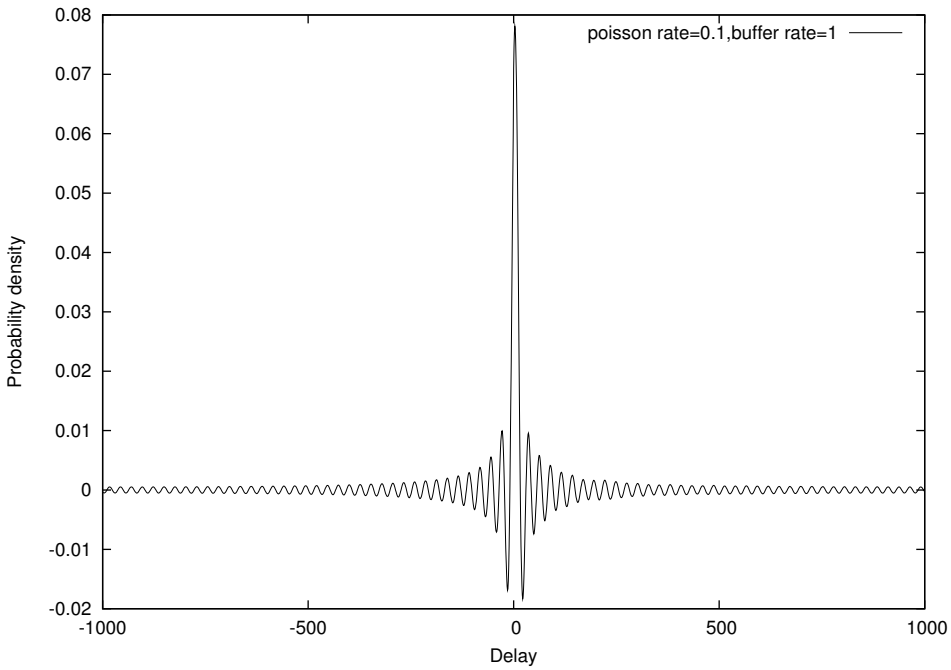

図 9　送信バッファにおける遅延の分布

5 結論とまとめ

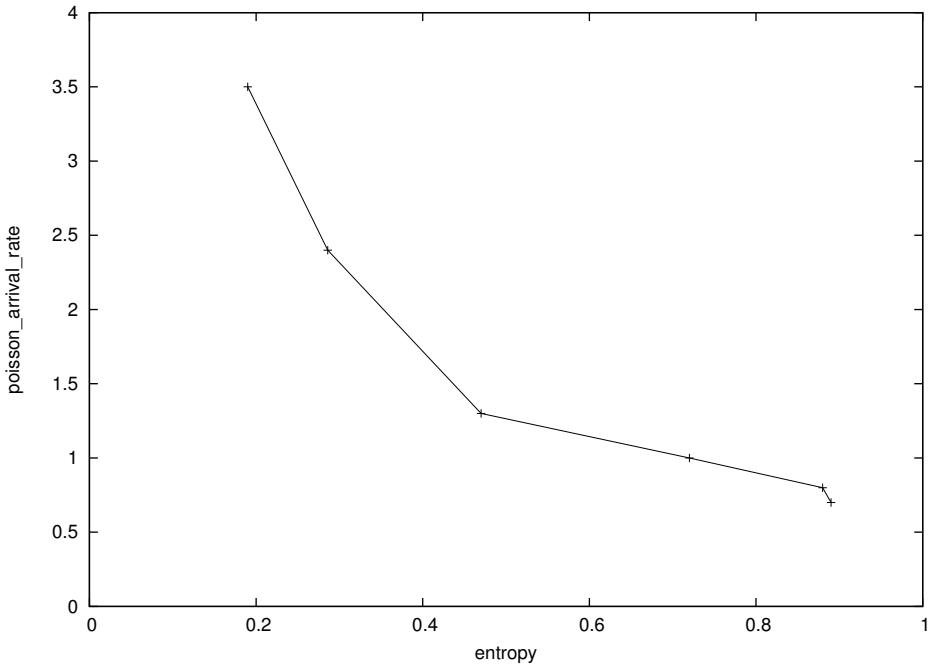

図 10　最適な λ とエントロピーの関係

本研究において送信バッファのレート $\mu = 1.0$ が固定で入力シンボルの到着レート λ を設定することができるシステムを仮定した. その際における最適な λ を設定するために入力シンボルの出現頻度を変化させてその時の平均遅延が最小になる λ を求めて, エントロピーと λ の点を結んだ結果を図 10 に示す. この図 10 を参考に λ を設定することにより遅延を最小限に留めることが可能となる. 図 10 は $\mu = 1$ の時についてだが,μ を α 倍に変化させたときはこの結果のグラフで得られた λ を α 倍することにより平均遅延を最小とする λ の目安を求めることができる. これは平均遅延と λ と μ の関係が 4.4 節で証明されているからである.

本研究で得られた知見は以下の通りである.

1. 入力シンボルの到着レートと送信バッファのレートを同時に大きくすることはシステム全体の処理速度を早めたことと同等である.
2. 平均遅延と入力シンボルの到着レートと送信バッファのレートの関係は反比例となる.
3. 入力シンボルの到着レートをただ単に大きくすると送信バッファに符号語が蓄積され遅延を生じ最終的にはパンク状態となる.
4. 送信バッファのレートが固定されているシステムを仮定した際, 算術符号における平均遅延は入力シンボルの到着レートを大きくすることにより小さくなる.
5. システム全体の遅延を考える際は, 算術符号のエンコーダから送信バッファに符号語

が送られるレート $\lambda H(X)$ が $\lambda H(x) < \mu$ の条件を考慮しなければならない.

6. 送信バッファにおける遅延の分布はインパルスとなる.
7. μ が決まっている伝送システムにおける平均遅延が最も小さくなる λ を設定する目安がわかった.

本研究において平均遅延を最も小さくする目安の λ を求めることができた. 今後の課題として, 情報源とエントロピーと平均遅延が最小となる λ の関係を数学的に解明することが挙げられる. これにより正確な λ を設計できるようになるのではないかと考えられる.

付録 A　算術符号を用いて実装した符号化のソースコード

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

/**********指数分布乱数を発生**********************************************/

#define MRND 1000000000L
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 rand_(void)
{
    return ((-100)*log(1-(rnd())));
}

/*************************************************************************/

double present_time;      /*  現在時刻  */
double arrival_time;        /*  到着時刻  */
double sending_time;         /*送信終了時刻*/

#define ARRIVING_LIST_MAX 1000
#define SENDING_BUFFER_LIST_MAX 1000
int next_ch;

double arriving_list[ARRIVING_LIST_MAX];
int arriving_count=0;
int arriving_in=0;
int arriving_out=0;

int sending_buffer_list[SENDING_BUFFER_LIST_MAX];
int sending_buffer_count=0;
int sending_buffer_in=0;
int sending_buffer_out=0;

int bit;
int freq[NO_OF_CHARS] = { 1, 9 };
int cum_freq[NO_OF_CHARS + 1];

/*************************************************************************/

void add_arriving_list(double time)
{
    if (arriving_count >= ARRIVING_LIST_MAX){

        printf("warning1!!");
        exit(0);
    }
    arriving_list[arriving_in] = time;
    arriving_in = (arriving_in + 1) % ARRIVING_LIST_MAX;
    arriving_count++;
    return;
```

```
}

double get_arriving_list()
{
    double time;

    if (arriving_count == 0){

        printf("warning2!!");
        exit(0);
    }
    time = arriving_list[arriving_out];
    arriving_out = (arriving_out + 1) % ARRIVING_LIST_MAX;
    arriving_count--;
    return(time);
}

void add_sending_buffer_list(int bit)
{
    if (sending_buffer_count >= SENDING_BUFFER_LIST_MAX){

        printf("warning3!!");
        exit(0);
    }
    sending_buffer_list[sending_buffer_in] = bit;
    sending_buffer_in = (sending_buffer_in + 1) % SENDING_BUFFER_LIST_MAX;
    sending_buffer_count++;
    return;
}

int get_sending_buffer_list()
{
    int bit;


    if(sending_buffer_count == 0){

        printf("warning4!!");
        exit(0);
    }
    bit=sending_buffer_list[sending_buffer_out];
    sending_buffer_out=(sending_buffer_out+1)%SENDING_BUFFER_LIST_MAX;
    sending_buffer_count--;
    return(bit);
}


/*****************************************************************************/
void
start_model()
{
    cum_freq[0] = 0;
    cum_freq[1] = freq[0];
    cum_freq[2] = freq[0] + freq[1];
    if (cum_freq[NO_OF_CHARS] > MAX_FREQUENCY)
        abort();
    return;
```

```
}

/*****************************************************************************/

int low;
int high;
long bits_to_follow;
unsigned long input_length=0;
unsigned long output_length=0;

void
start_encoding()
{
    low = 0;
    high = TOP_VALUE;
    bits_to_follow = 0;
    return;
}

/*****************************************************************/

int d_low;
int d_high;
int v_low;
int v_high;

void
start_decoding()
{
         d_low=0;
         d_high=TOP_VALUE;
         v_low=0;
         v_high=TOP_VALUE;
         return;
}


/*****************************************************************************/

void
decode_symbol(
    int bit,
    int cum_freq[])

{
    if( bit==0 ){
        v_high=(v_high + v_low+1)/2-1;
    } else {
        v_low=(v_high + v_low+1)/2;
    }
    for(;;){
        int d_b;
        double delay;
        d_b=((d_high-d_low+1)*cum_freq[1])/cum_freq[2]+d_low;
        if (v_high+1 <= d_b){
            if(output_length < input_length){
                delay = present_time - get_arriving_list();
```

```
                printf("%f\n",delay);
                output_length++;
            }
            d_high=d_b-1;
        } else if (v_low>=d_b){
            if(output_length<input_length){
                delay = present_time - get_arriving_list();
                printf("%f\n",delay);
                output_length++;
            }
            d_low=d_b;
        } else {
            break;
        }
        for (;;){
            if (d_high+1<=HALF){
                d_low=d_low*2;
                d_high=(d_high+1)*2-1;
                v_low=v_low*2;
                v_high=(v_high+1)*2-1;
            } else if (d_low>=HALF){
                d_low=(d_low-HALF)*2;
                d_high=(d_high-HALF+1)*2-1;
                v_low=(v_low-HALF)*2;
                v_high=(v_high-HALF+1)*2-1;
            } else if(d_low>=FIRST_QTR && d_high+1<=THIRD_QTR){
                d_low=(d_low-FIRST_QTR)*2;
                d_high=(d_high-FIRST_QTR+1)*2-1;
                v_low=(v_low-FIRST_QTR)*2;
                v_high=(v_high-FIRST_QTR+1)*2-1;
            } else {
                break;
            }
        }
    }
    return;
}




void
sending_finish()       /* 送信終了処理 */
{
    int bit;

    bit = get_sending_buffer_list();
    decode_symbol(bit, cum_freq);            /*デコード処理*/
    if(sending_buffer_count == 0){
        sending_time = -1.0;
    } else {
        sending_time = present_time + 0 ;
    }
    return;
}
```

```
void
bit_plus_follow(
    int bit)
{
    add_sending_buffer_list(bit);
    if ( sending_time < 0.0){
        sending_time = present_time + 0 ;
    }
    while (bits_to_follow > 0){
        add_sending_buffer_list(1 - bit);
        bits_to_follow--;
    }
    return;
}

void
encode_symbol(
    int symbol,
    int cum_freq[])
{
    long range;

    range = (long)(high - low) + 1;
    high = low + (range * cum_freq[symbol+1])/cum_freq[NO_OF_CHARS]-1;
    low  = low + (range * cum_freq[symbol  ])/cum_freq[NO_OF_CHARS];
    for(;;){
        if (high < HALF){
            bit_plus_follow(0);
        } else if (low >= HALF){
            bit_plus_follow(1);
            low  -= HALF;
            high -= HALF;
        } else if (low >= FIRST_QTR && high < THIRD_QTR){
            bits_to_follow += 1;
            low  -= FIRST_QTR;
            high -= FIRST_QTR;
        } else {
            break;
        }
        low  = 2 * low;
        high = 2 * high + 1;
    }
    return;
}
/********************************************************************************/
void
done_encoding()
{
    bits_to_follow += 1;
    bit_plus_follow((low < FIRST_QTR)? 0: 1);
    return;
}
/********************************************************************************/


void
```

```
arrival_symbol()        /*到着シンボル処理*/
{

    input_length++;
    add_arriving_list(present_time);
    encode_symbol(next_ch - '0', cum_freq);     /*エンコード処理*/
    if ((next_ch = getchar()) == EOF){
        arrival_time = -1.0;
    } else {
        arrival_time = present_time + rand_();
    }
     return;
}




int
main(void)
{
    present_time = 0.0;      /*  現在時刻  */
    arrival_time = rand_();        /*  到着時刻  */
    sending_time = -1.0;         /*送信終了時刻*/
    init_rnd(3);
    start_model();
    start_encoding();
    start_decoding();
    next_ch = getchar();


    while ((arrival_time >= 0.0) || (sending_time >= 0.0)){

        if (arrival_time<0 || (sending_time>=0 && arrival_time>sending_time)){

            present_time = sending_time;
            sending_finish();   /*送信終了処理*/
        } else {

            present_time = arrival_time;
            arrival_symbol(); /*到着シンボル処理*/
        }
    }
    return(0);
}
```

付録 B　付録 A のヘッダファイル

arithmetic_coding.h

```
#define CODE_VALUE_BITS 16
#define TOP_VALUE (((long)1 << CODE_VALUE_BITS) - 1)
#define FIRST_QTR (TOP_VALUE / 4 + 1)
#define HALF (2 * FIRST_QTR)
```

```
#define THIRD_QTR (3 * FIRST_QTR)

#define NO_OF_CHARS (2)
#define MAX_FREQUENCY (16383)
```

付録 C　Knuth の乱数発生法

```
#define MRND 1000000000L
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();
}

int
main(void)
{
    int i;
    init_rnd(50);


    for(i=0;1024*1024>i;i++){
        double r;
        r=rnd();
        if(r<0.4)
        putchar('0');
        else
        putchar('1');
    }
        return(0);
}
```

付録 D　期待値算出のソースコード

```
#include <stdio.h>

#define MIN_num 0   // カウントする数字の最小値
#define MAX_num 10000 // カウントする数字の最大値
#define DENOMINATOR (1024*1024)    // 分布を示すための分母

int main(){
    int i, // 一時的に使用する変数
        count[MAX_num - MIN_num + 1] = {0}, // 各数字のカウント数
        sum = 0; // 数字×カウント数
    float input_num, // 標準入力から読み取る数字
          except; // 期待値

    while(scanf("%f", &input_num) != EOF){ // 標準入力が終わるまで読む
        i = (int)input_num; // int 型に型変換
        if(i < MIN_num || i > MAX_num){ // 入力値が指定範囲を越えたらエラー
            printf("ERROR : input number is too large or small\n");
            return -1;
        }
        count[i - MIN_num]++;
    }

    for(i = 0; i < MAX_num - MIN_num + 1; i++){ // 結果出力
        // 分布確率
        // printf("%d %f\n", i + MIN_num, (float)count[i] / DENOMINATOR);

        // 期待値のための合計値算出
        sum += (i + MIN_num) * count[i];

        // 単純なカウントの場合
        // printf("%d %d\n", i + MIN_num, count[i]);
```

```
    }

    // 期待値出力
    except = (float)sum / DENOMINATOR;
    printf("expectation value = %f\n", except);

    return 0;
}
```

付録 E　相対頻度算出のソースコード

```
#include <stdio.h>

#define MIN_num 0   // カウントする数字の最小値
#define MAX_num 1000 // カウントする数字の最大値
#define DENOMINATOR (1024*1024)    // 分布を示すための分母

int main(){
    int i, // 一時的に使用する変数
        count[MAX_num - MIN_num + 1] = {0}; // 各数字のカウント数
    float input_num; // 標準入力から読み取る数字

    while(scanf("%f", &input_num) != EOF){ // 標準入力が終わるまで読む
        i = (int)input_num; // int 型に型変換
        if(i < MIN_num || i > MAX_num){ // 入力値が指定範囲を越えたらエラー
            printf("ERROR : input number is too large or small\n");
            return -1;
        }
        count[i - MIN_num]++;
    }

    for(i = 0; i < MAX_num - MIN_num + 1; i++){ // 結果出力
        printf("%d %f\n", i + MIN_num, (float)count[i] / DENOMINATOR);

        // 単純なカウントの場合
        // printf("%d %d\n", i + MIN_num, count[i]);
    }

    return 0;
}
```

謝辞

本論文を終えるにあたり, 本論文の作成において終始一貫して丁寧なご指導を賜りました指導教員の西新幹彦先生, 卒業論文発表の際にアドバイスを頂いた杉村先生に心より感謝と敬意の意を申し上げます.

参考文献

[1] 西新幹彦, 渡邊大, 森田啓義,“算術符号における符号語の生成過程の確率モデルについて,” 電子情報通信学会論文誌,Vol.J90-A, No.2, pp.170–174,2007.

[2] Timothy C. Bell, John G. Cleary, Text Compression, Prentice Hall, 1990.

[3] 森村英典, 大前義次, 応用待ち行列理論, 日本技連, 1975.

[4] 西田俊夫, 待ち行列の理論と応用, 朝倉書店, 1971.

[5] 奥村晴彦, C 言語によるアルゴリズム辞典, 技術評論社, 1991.