Simulation の作り方

屋代智之

平成 13 年 12 月 13 日

1 待ち行列理論

ここでは，一つの窓口 (つまり，同時には一人にしかサービスできない) に対して，適当に客が到着し，窓口でなんらかのサービスを受けるのに適当な時間必要として，そのあと，出ていく，というモデルを考える．

この場合，だれかがサービスを受けている間に到着した人は，そのサービスが終わるまで待っていなければならない．この待っている人々(当然，複数の人が待っているケースもある) は，到着した順番に待っていることになる．

この待っている人たち (の列) を**待ち行列**あるいは**待ち列**と呼ぶ．また，サービスを受けている時間を**サービス時間**と呼び，客が到着する頻度 (単位時間当たりに到着する数) を**到着率**と呼ぶ．

窓口でのサービス時間の平均と，客の平均した到着間隔から，客がサービスを受け終わるまでの時間 (待ち列で待っている時間＋サービスを受けている時間) の平均を「理論的に」求めるのが**待ち行列理論** (Queueing Theory:キューイングセオリー) である．

待ち行列のモデルには，客が待ち行列に並んでくれる場合と，すぐにサービスを受けられない場合に，待たずに去っていくケースが考えられる．すぐにサービスを受けられない場合に並んでくれる場合を**待時型モデル**と呼び，待たずに (サービスを受けずに) 去っていく場合を**即時型モデル**と呼ぶ．

待時型モデルで待ち列の長さに (場所の都合などで) 上限があり，上限を越えて到着した場合には，その客は並ばず，サービスも受けずに去っていくモデルを**損失系モデル**，待ち列の長さに上限がない場合を**非損失系モデル**と呼ぶ．

一般に窓口に客が到着するのは，すべてが独立した事象である．つまり，他の人がいつ窓口に行くかに関係なく，ある人は必要となった時点で窓口に赴く．

すなわち，窓口で見ていると，まったくランダムに人々がやってくることになる (図 1)．なお，ここでは数学的な都合上，同時に複数の人がやってくることは一般に考えない．

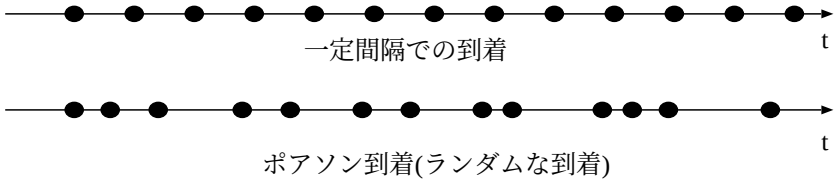


図 1: ポアソン到着

このようにまったくランダムに到着する場合を，ポアソン到着と呼ぶ．ポアソン到着の間隔をすべて調べて，それをヒストグラムにすると，図 2下記のようになる．

その間隔を数学的に解析して，確率密度分布で表すと指数分布となることが知られている (図 3)．

指数分布は数学的には平均値を与えれば定まる関数であり，比較的解析が容易な関数である．

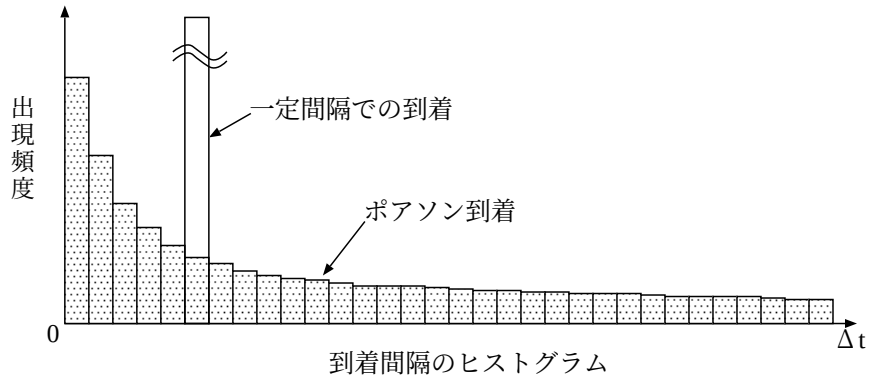


図 2: ポアソン到着のヒストグラム

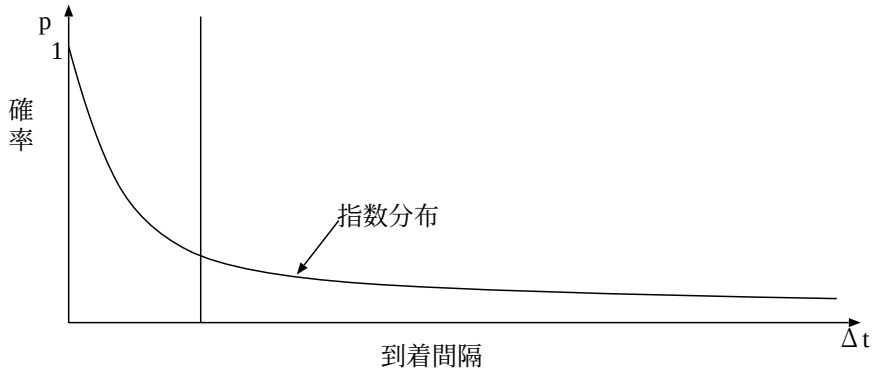


図 3: 指数分布関数

2　M/M/1

到着が完全にランダムであれば，その間隔は指数分布関数となる．同様にサービス処理時間も指数分布関数を用いた場合のこのモデルを M/M/1 と呼ぶ．

ここで M は，Markov の頭文字であり，指数分布関数で表されることを意味する．最初の “M” は，到着間隔が指数分布関数であることを表しており，次の “M” は処理時間が指数分布関数であることを意味する．最後の “1” は窓口が一つであることを意味する．

同様に D(Deterministic:単位分布) は，一定時間で処理されることを表す．M/D/1 は，ランダムに到着し，全員同じ時間で処理される，窓口が一つのモデルである．G(General:一般分布) は一定でも指数分布でもない場合を表す．

一般に M/M/n，M/D/n，D/D/n(n は窓口が n 個あることを表す) は数学的 (解析的) に平均待ち列長を得ることが出来るが，それ以外のものは出来ないことが多い．

そこで，そのようなモデルを解析するためにシミュレーション (コンピュータシミュレーション) が用いられる．

3　時間表現

シミュレーションの話の前に，シミュレーション上で時間をどのように表現するか，を簡単に説明する．例えば，100 秒分のシミュレーションを行うことを考える．この 100 秒は実世界では文字通り 100 秒の長さを持っている．コンピュータ上で，この 100 秒をどのように表したらよいだろうか？

答えは簡単である.「適当に」表せば良いのである．

例えば，`int` 型の `time` という変数を用意する．この変数が表しているものが「シミュレーション開始からの秒数」と決めてしまえば良い．もちろん，`int` 型なので，秒の精度でしか時間を表現することは出来ない．もっと細かく制御したければ，例えば `float` 型を使うとか，同じ `int` 型を使いながら，その意味を「シミュレーション開始からのミリ秒数」と決めればよい．

4　シミュレーション

ここでは，M/M/1 をコンピュータ上で表現することを考える．一般にコンピュータ上でシミュレーションを行うには**タイムドリブン**と呼ばれる手法と，**イベントドリブン**と呼ばれる手法の二つがある．ここでは，それぞれについて簡単に説明する．

4.1　タイムドリブン

タイムドリブン型のシミュレーションは，文字通り時間を少しずつ進めながら，シミュレーションを行っていく方法である．

ここではシミュレーション中の時間のことを実際の時間と区別するために「時刻」と表記する．シミュレーションは，時刻 0 から進んでいくものと考える．

例えば，上記の窓口のシミュレーションで考える．１秒きざみでシミュレーションを動かしていくようなケースでは，ある一秒に着目すると，その一秒で窓口に客が到着する確率は，p_1 で表すことができる．

この確率に応じて乱数を用いて，客が到着するように (例えば，p_1 が 0.01 であれば，平均 100 回に 1 回，客が到着する) すると，客の到着に関するシミュレーションが行える．到着した場合には，まだ窓口でサービスを受けている人がいれば，待ち列に並ぶし，いなければすぐにサービスを受ければよい．

同様に処理 (サービス) が終了する確率を p_2 とすると，**客がいるときには，**p_2 に応じた確率で処理が終了するようにしてあげればよい．

これを簡単なフローチャートで表すと，図 4のようになる．

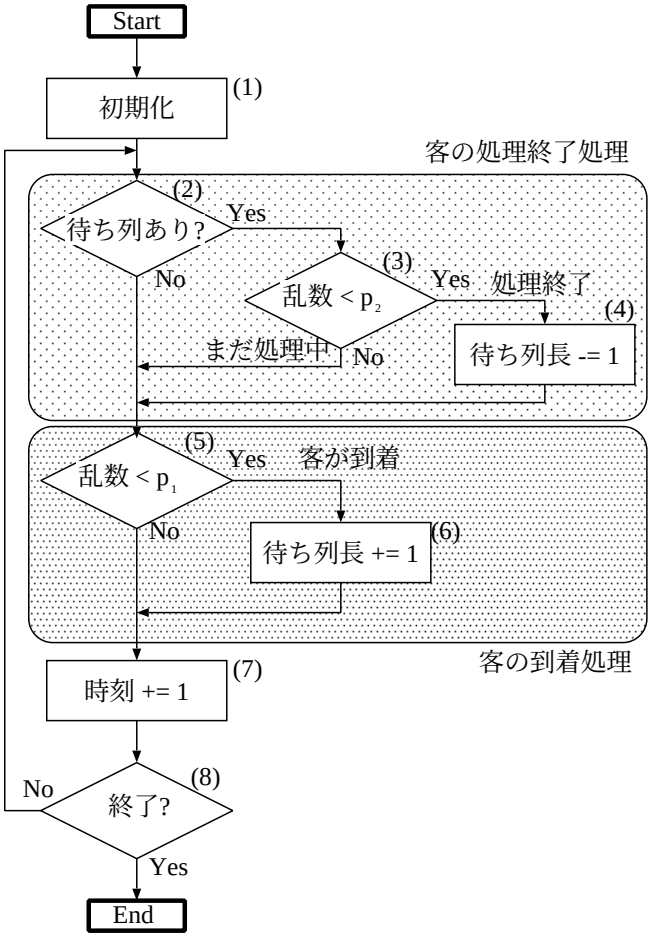


図 4: タイムドリブンのフローチャート

ここで，客の処理終了に関する処理を先に行い，到着処理を後に行っているのは，逆にすると到着した客がすぐに (到着した瞬間に) 処理終了してしまうのを避けるためである．

このフローチャートを見ればわかるように，このようなシミュレーションでは，p_1 と p_2 さえ決まればよい．

実際にこのフローチャートからプログラムを作ると，図 5のようになる．

では，このプログラムの到着間隔は実際にはどうなっているか調べてみよう．図 5のプログラムを変更して，到着間隔のヒストグラムをとってみる (図 6)．

ここでは，統計的なデータを取得するため，終了時間を 1000000 と非常に大きくしてあるが，プログラムの動作的にはほぼ同じである．

出てきた結果 (到着間隔のヒストグラム) をグラフにしてみると，図 7のようになる．

このように，確かに指数分布になっているのが確認できるであろう．

さて，p_1 と p_2 によるが，多くの時刻では，乱数発生と時刻を進める処理しか行っていない．このように一見無駄に思える処理があるものの，タイムドリブン型シミュレーションはかなり強力なシミュレーションであるため，多くのケースで用いられている．

一方，このような無駄な処理をなくしてしまおう，という発想のもとに考えられたのがイベントドリブン型のシミュレーションである．

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define Rand()  ((double)rand()/RAND_MAX)/*     0 〜 1 までの一様分布乱数を返すマクロ */
5  #define END_TIME 1000                    /*     シミュレーション終了時刻 */
6  /* p1, p2 の設定 */
7  #define P1      0.1                      /*     1/10 →到着間隔平均 10 秒 */
8  #define P2      0.2                      /*     1/ 5 →処理時間平均 5 秒 */
9
10  /* グローバル変数 */
11  int time;                               /*     時刻 */
12  int total_queue;                        /*     待ち列長合計 (集計用) */
13  int queue;                              /*     待ち列長 */
14
15  int main()
16  {
17      /* (1) 初期化 */
18      time  = 0;                          /*     時刻を 0 に */
19      queue = 0;                          /*     最初はだれも待っていない */
20      total_queue = 0;                    /*     集計用データ */
21
22      while(time < END_TIME){             /* (8) 終了判定    */
23          /* 客の処理終了処理 */
24          if (queue != 0){                /* (2) 待ち列あり? */
25              /* Yes の場合 (待ち列あり) */
26              if (Rand() < P2){           /* (3) 乱数 < P2 */
27                  /* Yes の場合 (処理終了) */
28                  queue--;                /* (4) 待ち列長 -= 1 */
29              }
30          }
31          /* 客の到着処理 */
32          if (Rand() < P1){               /* (5) 乱数 < P1 */
33              /* Yes の場合 (客が到着) */
34              queue++;                    /* (6) 待ち列長 += 1 */
35          }
36          time++;                         /* (7) 時刻を進める */
37          total_queue += queue;           /*     集計用 */
38      }
39      /* 最後に集計結果表示 */
40      printf("Average = %lf\n", (double)total_queue / time);
41      return 0;
42  }
```

図 5: タイムドリブンのCプログラム

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define Rand()  ((double)rand()/RAND_MAX)/*     0 〜 1 までの一様分布乱数を返すマクロ */
5  #define END_TIME 1000000                 /*     シミュレーション終了時刻 */
6  /* p1 の設定 */
7  #define P1      0.1                      /*     1/10 →到着間隔平均 10 秒 */
8
9  /* グローバル変数 */
10  int time;                               /*     時刻 */
11
12  int main()
13  {
14      /* (1) 初期化 */
15      int i;
16      int interval = 0;
17      int hist[1000];
18      int total = 0;
19      int totalhist = 0;
20
21      for (i = 0; i < 1000; i++) hist[i] = 0;
22      while(time < END_TIME){             /* (8) 終了判定 */
23          if (Rand() < P1){               /* (5) 乱数 < P1 */
24              if (interval < 1000)  hist[interval]++;
25              interval = 0;
26          } else {
27              interval++;
28          }
29          time++;                         /* (7) 時刻を進める */
30      }
31      for (i = 0; i < 1000; i++){
32          printf("%d, %d\n", i, hist[i]);
33          total += hist[i];
34          totalhist += i * hist[i];
35      }
36      printf("average = %lf\n", (double)totalhist / total);
37      return 0;
38  }
```

図 6: ヒストグラム作成プログラム

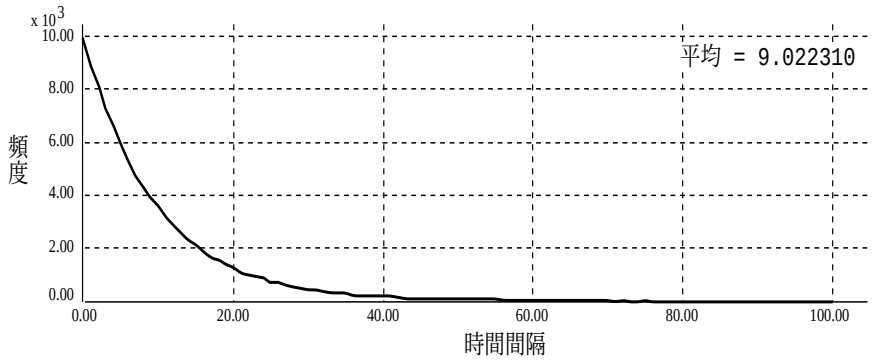


図 7: 指数分布のヒストグラム (タイムド リブン)

4.2 イベントドリブン

イベントドリブン型のシミュレーションでは，なんらかの状態が変化するものを「イベント」と呼ぶ．
先ほどの窓口の例では，

- 客が到着する．
- 客への処理 (サービス) が終了する．

といった場合には，待ち列の長さが変化する．このようなものをイベントと呼ぶ．待ち列の長さを評価するのであれば，これ以外にはイベントはない．

いいかえれば，状況を変えるものがイベントであるので，現状から予定される**全ての**イベントのうち，最も近い将来の (はやく発生する) イベントの時刻までは**状況は変化しない**はずである．

すべてのイベントが，それ以前のイベントによって予定できる場合，イベントドリブン方式は高速なシミュレーションを実行するための有効な手法である．

では，具体的に考えてみよう．

窓口の例で考える．上記の窓口のシミュレーションは，窓口で並ぶ人がどのくらいいるか，ということを知るためのシミュレーションである．このような状況では，状態を変化させるイベントは，

1. 窓口に客が到着する．
2. 客への処理が終了する．
3. 次の客が処理を依頼し，処理が開始される．

の 3 種類となる．このうち，2 番目と 3 番目は (待ち列に人がいれば) 連続して行われるため，より単純化させると，1 番目と 2 番目だけ考えれば良い．

ここで，それぞれのイベントで予定される将来のイベントを考える．

1. 窓口に客が到着する．
    - 客の到着間隔は指数分布で表されるので，次の到着時刻が予定できる．
    - もしも窓口が空いていれば，処理が開始されるので終了時刻が予定できる．
2. 客への処理が終了する．
    - もしも待ち列に人がいれば，次の人の処理が開始されるので，その終了時刻が予定できる．

客が到着する間隔は指数分布となるため，指数分布の乱数が作れれば，最後に客が到着した時点で，次の客の到着時刻は予定可能である．

同様に，処理が開始された時点で，その客への処理が終了する時刻も (指数分布関数を用いて) 予定可能である．

例えば，客が到着したときには，次の客の到着イベントが予定可能である．また，窓口で処理されている人がいなければ，到着と同時に処理されるので，処理が終了するイベントも予定可能である．

このイベントの流れをフローチャートで表してみると，図 8のようになる．

このフローチャートを見れば明らかなように,「客の到着」が起こった場合には必ず次のイベントとして,「客の到着」を予定するので，将来の予定がなくなることはあり得ない．これは重要なことで，将来の予定が無くなってしまった場合，シミュレーションはそこから進められなくなってしまうので，注意が必要である．

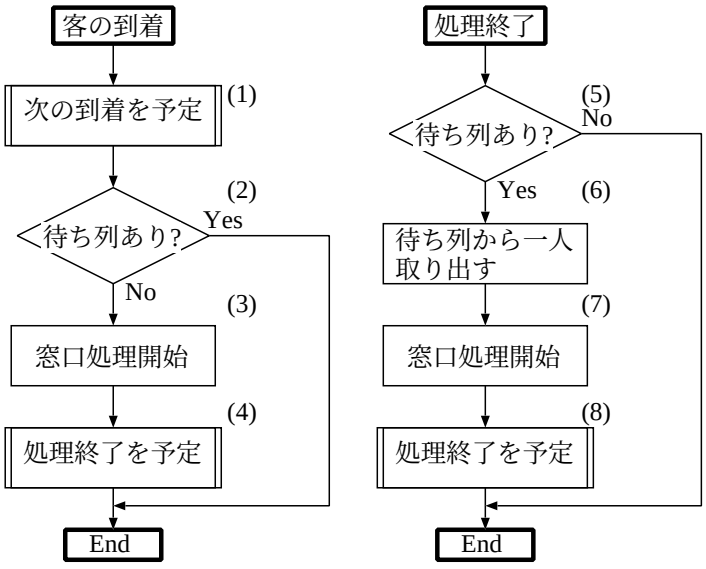
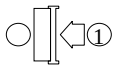
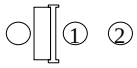
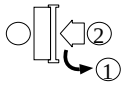
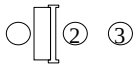
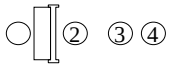
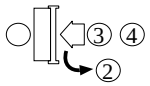


図 8: イベントのフローチャート

予定されたイベントは，予定された時刻順に並べておけば，最初に実行されるイベントを検索するのが容易である．このため，一般にはリスト構造を用いてイベントを管理する (リスト構造については，別紙参照)．一般にこのリストを**イベントリスト**と呼び，イベントを予定することを**イベントをスケジュールする**あるいは**イベントのスケジューリング**と呼ぶ．

表 1にイベントのスケジューリングとその時の状態の例を示す．シミュレーションを作成するときは，この表のように，イベントのある時刻だけなんらかの処理を行う．

表 1: 時刻と状態の変化 (例)

時刻	状態	スケジュール	状態図	イベントリスト
0	初期状態	時刻 8 に客 1 の到着を予定		8：**客 1 の到着**
8	客 1 到着	客 1 の処理開始 時刻 14 に客 2 の到着を予定 時刻 18 に処理終了予定		14：**客 2 の到着** 18：**客 1 の処理終了**
14	客 2 到着	時刻 23 に客 3 の到着を予定 まだ窓口が処理中のため，処理終了は未定		18：客 1 の処理終了 23：**客 3 の到着**
18	客 1 処理終了	客 2 の処理開始 時刻 31 に処理終了予定		23：客 3 の到着 31：**客 2 の処理終了**
23	客 3 到着	時刻 29 に客 4 の到着を予定 まだ窓口が処理中のため，処理終了は未定		29：**客 4 の到着** 31：客 2 の処理終了
29	客 4 到着	時刻 34 に客 5 の到着を予定 まだ窓口が処理中のため，処理終了は未定		31：客 2 の処理終了 34：**客 5 の到着**
31	客 2 処理終了	客 3 の処理開始 時刻 37 に処理終了予定		34：客 5 の到着 37：**客 3 の処理終了**
…	…	…		

では，次に実際にイベントド リブンのシミュレーションを作成する際の全体のフローチャートを作成する．

基本的には，図 8で表した部分が各イベントの処理内容である．このため，これらのイベント処理を正しく呼び出す部分を作れば良い．

図 9にイベント処理内容 (図 8の部分) を除いたフローチャートを示す．

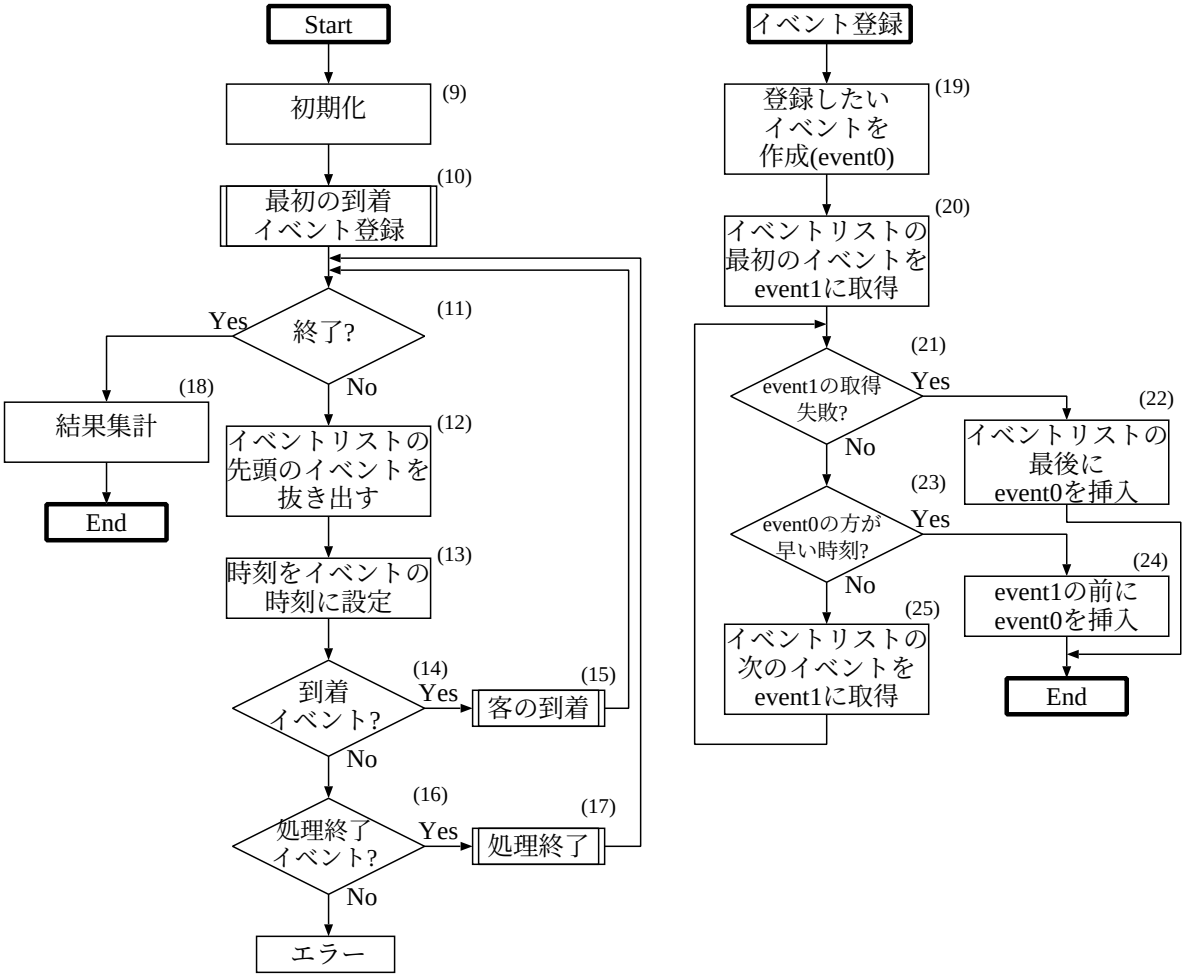


図 9: イベントド リブンのフローチャート

基本的にはこれをそのままプログラムにすれば，シミュレーションは完成する．図 10にこのプログラムの main 関数を示し，図 11にイベント処理部分とイベント登録部分のプログラムを示す．

行数で比較するとイベントドリブンの方がかなり多い．これは現状でプログラムが比較的簡単なためにタイムドリブンのプログラムが短く出来たためである．

イベントドリブンのプログラムはリスト処理など，タイムドリブンには不要な処理があるため，若干長くなっている．しかし，複雑なシミュレーションでは一概にどちらが短くなるとは言えない．

また実行時間を比較すると，この程度のプログラムであるとどちらの方法でもほぼ差はない．しかし，一般にイベントドリブンの方が高速で精度がよい．

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  #define Rand()  ((double)(rand()+1)/RAND_MAX)/*     0 〜 1 までの一様分布乱数を返すマクロ */
6  #define ernd(num)  (-log(Rand())*(num))  /*     平均値が num の指数分布乱数 */
7
8  #define END_TIME 1000.0                  /*     シミュレーション終了時刻 */
9  #define P1      10.0                     /*     到着間隔平均 10 秒 */
10  #define P2      5.0                      /*     処理時間平均 5 秒 */
11
12  typedef enum {
13      E_ARRIVAL = 100,                     /*     到着イベント */
14      E_DEPARTURE                          /*     処理終了イベント */
15  } event_type;
16
17  typedef struct _event {
18      event_type type;                     /*     イベントの種類 */
19      double time;                         /*     イベントの発生時刻 */
20      struct _event *next;                 /*     次のイベント (イベントリスト用) */
21  } event;
22
23  /* 関数のプロトタイプ宣言 */
24  void    e_arrival();
25  void    e_departure();
26  void    schedule(event_type type, int interval);
27
28  /* グローバル変数 */
29  double time;                             /*     時刻 */
30  double l_time;                           /*     直前の処理時刻 (集計用) */
31  int total_queue;                         /*     待ち列長合計 (集計用) */
32  int queue;                               /*     待ち列長 */
33  event top;                               /*     イベントリストの先頭 (空) */
34
35  int main()
36  {
37      event *current;                      /*     処理対象のイベント */
38
39      /* (1) 初期化 */
40      time  = 0.0;                         /*     時刻を 0 に */
41      queue = 0.0;                         /*     最初はだれも待っていない */
42      total_queue = 0;                     /*     集計用データ */
43
44      /* 最初の到着イベント登録 */
45      schedule(E_ARRIVAL, P1);  /* (10) 最初の到着イベント登録 */
46
47      while(time < END_TIME){              /* (11) 終了判定 */
48          /* (12) イベントリストの先頭のイベントを抜き出す */
49          current = top.next;
50          top.next = current->next;
51
52          /* (13) 時刻をイベントの時刻に設定 */
53          l_time = time;
54          time = current->time;
55          /* 集計処理 */
56          total_queue += (time - l_time) * queue;
57
58          switch (current->type){
59            case E_ARRIVAL:                /*  (14) 到着イベント？ */
60              e_arrival();                 /*  (15) 客の到着 */
61              break;
62            case E_DEPARTURE:              /*  (16) 処理終了イベント？ */
63              e_departure();               /*  (17) 処理終了 */
64              break;
65            default:
66              /* エラー処理 */
67              break;
68          }
69          free(current);
70      }
71      /* (18) 最後に集計結果表示 */
72      printf("Average = %lf\n", (double)total_queue / time);
73      return 0;
74  }
```

図 10: イベントドリブンのCプログラム (main 部分)

```
75
76  void e_arrival()
77  {
78      queue++;                            /*     自分を待ち列に追加 */
79      schedule(E_ARRIVAL, P1);            /* (1) 次の到着を予定 */
80      if (queue == 1)                     /* (2) 待ち列無し (自分だけ) */
81          schedule(E_DEPARTURE, P2);      /* (4) 処理終了イベントを予定 */
82  }
83
84  void e_departure()
85  {
86      queue--;                            /*     待ち列から自分を取り除く */
87      if (queue != 0)                     /* (5) 次の客が待っている */
88          schedule(E_DEPARTURE, P2);      /* (8) 処理終了イベントを予定 */
89  }
90
91  void schedule(event_type type, int interval)
92  {
93      event *event0, *event1;
94      event *pre_event1 = (event *)NULL;   /*     event1 の前にあるイベント */
95
96      /* (19) イベントを作成 */
97      event0 = (event *)malloc(sizeof(event));
98      event0->type = type;
99      event0->time = time + ernd((double)interval);
100      event0->next = (event *)NULL;
101
102      /* (20) イベントリストの最初のイベントを取得 */
103      event1 = top.next;
104      pre_event1 = &top;
105
106      while (1){
107          if (event1 == (event *)NULL){    /* (21)event1 の取得に失敗 */
108              pre_event1->next = event0;   /* (22)event1 の前 (=最後) に event0 を挿入 */
109              break;
110          } else if (event0->time < event1->time){ /* (23)event0 の方が早い時刻 */
111              pre_event1->next = event0;   /* (24)event1 の前に event0 を挿入 */
112              event0->next = event1;
113              break;
114          }
115          pre_event1 = event1;
116          event1 = event1->next;           /* (25) イベントリストの次のイベントを event1 に取得 */
117      }
118  }
```

図 11: イベントドリブンのCプログラム (イベント処理部)

5　結果の収束

いままでのシミュレーションプログラムは，終了判定を単に時刻の比較で行ってきた．もちろん，シミュレーション上の時間が短ければ，シミュレーション結果は精度が悪いものとなるが，ある程度シミュレーションを行えば，結果はほぼ収束し，それ以上精度が良くなるわけではない．

そこで，プログラム中で，結果の収束度を判定し，ある程度収束したところでシミュレーションを終了する，ということが一般には行われる．

例えば，図 10〜 11のプログラムの結果をシミュレーション終了時間ごとに出力してみると，図 12のように収束していく．

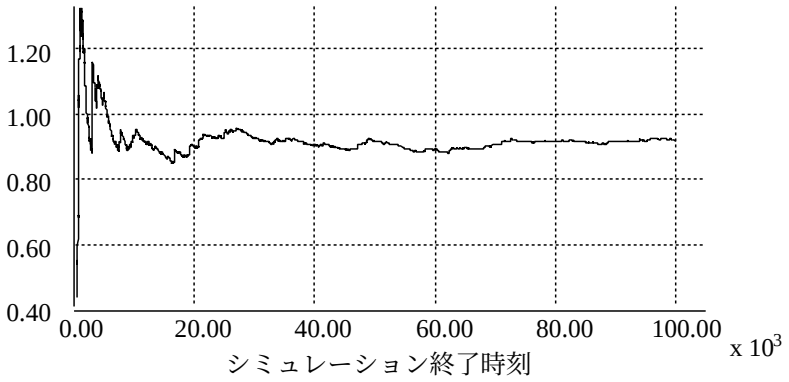


図 12: 結果の収束

では，この収束をどのようにしたら簡単にプログラム上で検出できるであろうか？

そこで，求めた結果 (ここでは平均値) に対して，分散の計算を行ってみる．分散 σ はデータが n 個ある場合には，下記の式で求めることができる．

$$\sigma = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - (\frac{1}{n}\sum_{i=1}^{n} x_i)^2 \tag{1}$$

タイムドリブン形式のシミュレーションであれば，各時刻の結果を x_i とすることで，この式をそのまま適用することが可能である．

イベントドリブン形式の場合，結果が求まる間隔はイベント発生に依存するので，上記のプログラムの例でいえば，

$$\sigma = \frac{1}{T}\sum_{t=0}^{T} \Delta t x_t^2 - (\frac{1}{T}\sum_{t=0}^{T} \Delta t x_t)^2 \tag{2}$$

ただし，T はシミュレーション終了時間，

Δt は今の状態の継続時間 (直後のイベント発生までの時間) を表す．

という式に変形する必要がある．

この結果をグラフにすると，図 13のようになる．

ここで，分散の大きさは平均値に依存する話であり，必ずしもこのような値になるわけではないが，分散自体は 0 に向かって収束していることが分かる．

本日はここまで！

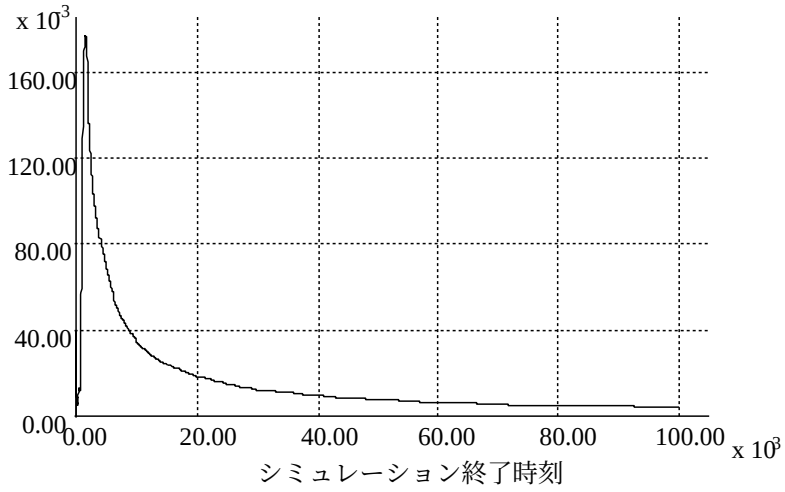

x 10^3
160.00
120.00
80.00
40.00
0.00
0.00
20.00
40.00
60.00
80.00
100.00
x 10^3
シミュレーション終了時刻

図 13: 分散の変化