>>> Assignment #3 for Simulation (CAP 4800) <<<

>>> SOLUTIONS <<<

This assignment covers material from the third week of class lecture.

Problem #1 (25 points)

A typical GET request to a web server requests an HTML page. This HTML page may have embedded images. Each embedded image results in another GET to the server. So, for example, if a given page has two embedded images then there will be two additional GET requests to the server for the images. Suppose that 937 web pages (HTML pages) were characterized for number of embedded images. The below table shows what was observed (that is, 129 pages have 0 embedded images, 247 pages have 1 embedded image, and so on).

Number of embedded images	Number of pages observed
0	129
1	247
2	112
3	332
4	117

Your task is to create a program that generates a simulated workload corresponding to the statistics in the above table. The workload is a series of integer values corresponding to the number of embedded images in a given request. Using your program, generate 1 million simulated requests and determine the mean. Compare the mean of the simulated workload to the expected theoretical mean (which you should calculate in your program) based on the data in the above table. Your program should output the pmf, CDF, and theoretical mean of your empirical distribution, and also the mean of generated requests. Submit your source code and a screen shot of the execution. **Hint:** You may take the printer workload program we discussed in class and modify it for this problem.

A screenshot of the execution is below and the source code is attached as as3_3_1sol.c.

```
VC++ cmd
c:\work>cl as3_3_1sol.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

as3_3_1sol.c
Microsoft (R) Incremental Linker Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:as3_3_1sol.exe
as3_3_1sol.obj

c:\work>as3_3_1sol
The pmf:
  f(0) = 0.137673
  f(1) = 0.263607
  f(2) = 0.119530
  f(3) = 0.354322
  f(4) = 0.124867

The CDF:
  F(0) = 0.137673
  F(1) = 0.401281
  F(2) = 0.520811
  F(3) = 0.875133
  F(4) = 1.000000

Theoretical mean = 2.065101 images per access

Sample mean for 1000000 samples = 2.065446 images per access

c:\work>
```

Problem #2 (25 points)

Here is a C function for a random number generator. Describe how to evaluate an RNG for "goodness". Evaluate the below RNG.

```
//==========================================================================
//= Superduper RNG for Simulation class (summer 2011)                      =
//==========================================================================
double rand_superduper(void)
{
  const long  a =      33312;
  const long  m = 2147483647;
  const long  q =     250001;
  const long  r =       1111;
  static long x = 1;
  long        x_new, x_div_q, x_mod_q;

  x_mod_q = x % q;
  x_div_q = x / q;
  x_new = a * x_mod_q;
  x_new = x_new - (r * x_div_q);
  if (x_new > 0) x = x_new;
  else x = ((-1) * x_new);

  return((double) x / m);
}
```

A good RNG should be unif(0, 1) and all values should be independent. We can plot a histogram to evaluate unif(0, 1) and look for zero autocorrelation for independence. Using the above RNG to generate 1 million samples and using hist.c and autoc.c we get:

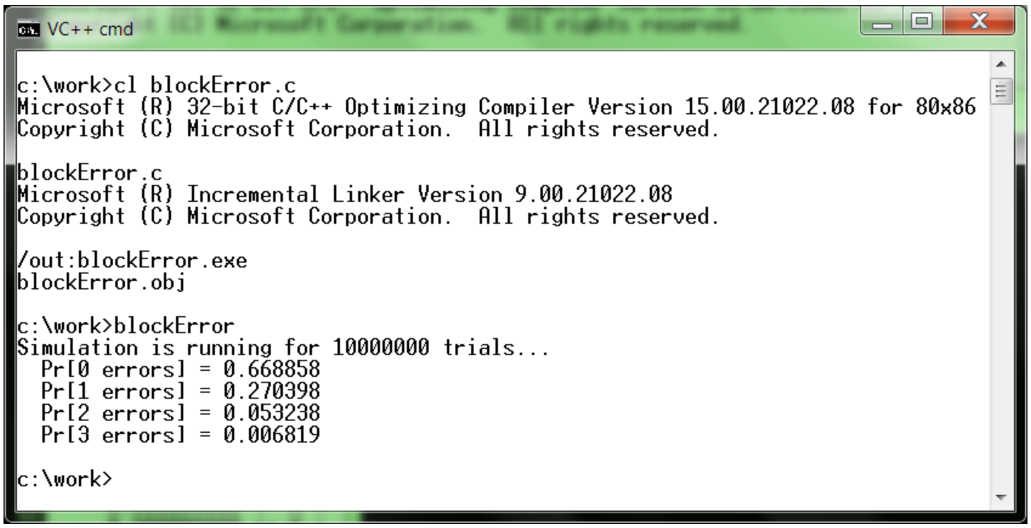
```
VC++ cmd
c:\work>hist < x
------------------------------------------------- hist.c -----
  Frequency for 10 buckets of size 0.100000 (for 1000000 samples)
     0.000000000 <= X <    0.100000000 =  77713 -- 7.771300 %
     0.100000000 <= X <    0.200000000 =  96965 -- 9.696500 %
     0.200000000 <= X <    0.300000000 = 106080 -- 10.608000 %
     0.300000000 <= X <    0.400000000 = 100410 -- 10.041000 %
     0.400000000 <= X <    0.500000000 = 106295 -- 10.629500 %
     0.500000000 <= X <    0.600000000 = 101966 -- 10.196600 %
     0.600000000 <= X <    0.700000000 = 100305 -- 10.030500 %
     0.700000000 <= X <    0.800000000 = 106445 -- 10.644500 %
     0.800000000 <= X <    0.900000000 = 100399 -- 10.039900 %
     0.900000000 <= X <    1.000000000 = 103422 -- 10.342200 %
--------------------------------------------------------------

c:\work>autoc < x
------------------------------------------------ autoc.c -----
  Autocorrelation for lag    1 = 0.004036
  Autocorrelation for lag    2 = -0.001008
  Autocorrelation for lag    3 = -0.002483
  Autocorrelation for lag    4 = 0.006531
  Autocorrelation for lag    5 = 0.008809
  Autocorrelation for lag    6 = 0.008769
  Autocorrelation for lag    7 = 0.007812
  Autocorrelation for lag    8 = 0.004365
  Autocorrelation for lag    9 = 0.004404
  Autocorrelation for lag   10 = 0.008462
--------------------------------------------------------------
```

This RNG is not very good – the values are not unif(0, 1). The value are, however, apparently independent.

<u>Problem #3</u> (25 points)

Consider a disk drive that transfers data in micro-blocks of 5 bytes. There is a probability of bit error, *p*, for each block transferred (that is, each bit has a probability of being in error of *p*). Bit errors are independent (that is, there is no correlation between bit errors). What is the probability of a block having 0, 1, 2, or 3 bit errors? The Monte Carlo simulation `blockError.c` (which can be found on the class source code page) models this system. Run the simulation for $p = 10^{-2}$. Submit a screenshot of the results. Also, analytically model this system and compare your analytical and simulation results.

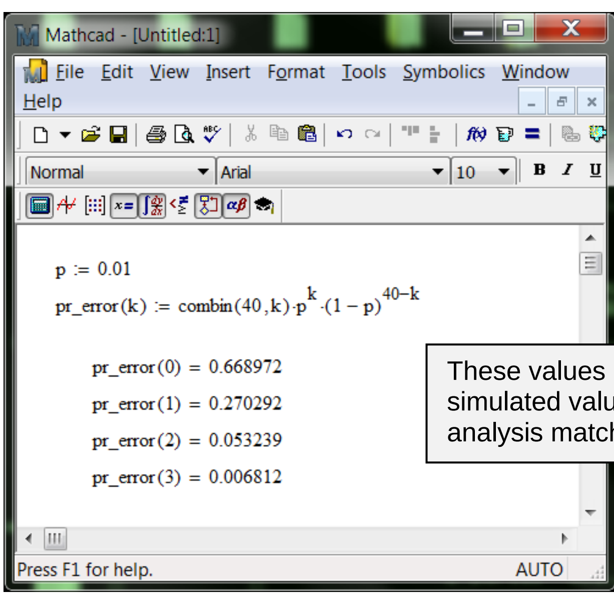
A screenshot of the program execution is:



This problem is a binomial distribution, so:

$$\Pr[k \text{ error in } 40 \text{ bits}] = \binom{40}{k} p^k (1-p)^k$$

where *k* = 0, 1, 2, and 3 for *p* = 0.01. Solving for the above (here using Mathcad, you can use the tool of your choice including a calculator) we get

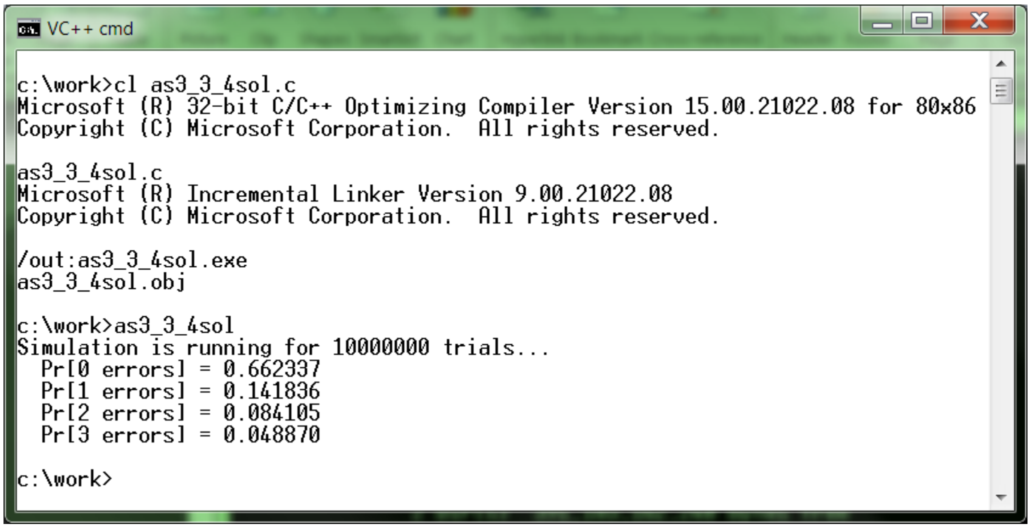


These values are easily within 1% of the simulated values. So, the simulation and analysis match, as expected.

Problem #4 (25 points)

Consider the same system of Problem #3. Modify `blockError.c` to model correlated bit errors as follows. If a given bit is not in error, then the probability that the next bit is in error is p. If a given bit is in error, then the probability that the next bit is in error is 0.5. Just the same as in Problem #3, for $p = 10^{-2}$ determine the probability of a block having 0, 1, 2, or 3 bit errors. This problem cannot (at least not easily) be modeled analytically. This is an example of a problem well suited for a Monte Carlo simulation model. Submit your source code and a screenshot of the results. Discuss your results – compare them to the results of Problem #3 and speculate on why the difference.

A screenshot of the execution is below and the source code is attached as as3_3_4sol.c.



The probability of 0 errors remains the same as for the independent case. This is not a surprise as only when a bit error occurs do the correlation effects (which increase the probability of additional bit errors) come into play. The probability of "few" bit errors (compare Pr[1 errors] for each case) is lower than for the independent case. However, for "many" bit errors (compare Pr[2 errors] for each case) the correlated case has higher probability (than the independent case). That is, the correlation or bit errors pushes more mass into the tail of the distribution.

```
//================================================== file = as3_3_1sol.c =====
//=  Program to generate simulated web accesses (# of images per access)      =
//=============================================================================
//=  Build: bcc32 as3_3_1sol.c                                                 =
//=---------------------------------------------------------------------------=
//=  Execute: as3_3_1sol                                                       =
//=============================================================================
#include <stdio.h>              // Needed for printf()
#include <stdlib.h>             // Needed for rand() and RAND_MAX

//----- Constants -------------------------------------------------------------
#define NUM_MEAS          5     // Number of measurements in "bins"
#define NUM_SAMPLES 1000000     // Number of Web access samples to simulate

//----- Globals ---------------------------------------------------------------
struct web_access               // Structure for holding web access sample
{
  int      num_images;          //  ** Number of images for this Web access
  double   ratio_observed;      //  ** Observed ratio (0 to 1) for this sample
};

// Observed # of images per Web access (sum of ratios must equal 1.0)
struct web_access size[NUM_MEAS] = {{0, 129.0 / 937.0}, {1, 247.0 / 937.0},
                                    {2, 112.0 / 937.0}, {3, 332.0 / 937.0},
                                    {4, 117.0 / 937.0}
                                   };

//===== Main program ==========================================================
void main(void)
{
  double dist_func[NUM_MEAS];      // The CDF for the Web access samples
  int    bin_num;                  // Measurement "bin" number
  int    image_count;              // Image count for this sample
  double mean_theory;              // Mean image count (theory)
  double mean_sample;              // Mean image count (sample)
  double z;                        // Uniform random number from 0 to 1
  int    i;                        // Loop counter

  // Output the pmf for the observed measurements
  printf("The pmf: \n");
  for (i=0; i<NUM_MEAS; i++)
    printf("  f(%d) = %f \n", size[i].num_images, size[i].ratio_observed);

  // Build and output the CDF for the observed measurements
  printf("\nThe CDF: \n");
  dist_func[0] = size[0].ratio_observed;
  printf("  F(%d) = %f \n", size[0].num_images, dist_func[0]);
  for (i=1; i<NUM_MEAS; i++)
  {
    dist_func[i] = dist_func[i-1] + size[i].ratio_observed;
    printf("  F(%d) = %f \n", size[i].num_images, dist_func[i]);
  }
```

```
  // Calculate and output the theoretical mean
  mean_theory = 0.0;
  for (i=1; i<NUM_MEAS; i++)
    mean_theory = mean_theory + (size[i].ratio_observed * size[i].num_images);
  printf("\nTheoretical mean = %f images per access \n", mean_theory);

  // Generate NUM_SAMPLES simulated Web accesses
  mean_sample = 0.0;
  for (i=0; i<NUM_SAMPLES; i++)
  {
    // Pull a uniform RV (0 < z < 1)
    do
    {
      z = ((double) rand() / RAND_MAX);
    }
    while ((z == 0) || (z == 1));

    // Map z to a bin number
    for (bin_num=0; bin_num<NUM_MEAS; bin_num++)
      if (z < dist_func[bin_num]) break;

    // Get image_count from selected bin
    image_count = size[bin_num].num_images;

    // Running sum for the mean_count
    mean_sample = mean_sample + ((double) image_count / NUM_SAMPLES);
  }

  // Output the mean image count
  printf("\nSample mean for %d samples = %f images per access \n",
    NUM_SAMPLES, mean_sample);
}
```

```
//================================================== file = as3_3_4sol.c =====
//= A Monte Carlo simulation of dedependent bit errors in a block            =
//=   - For Assignment #3, problem #4                                        =
//============================================================================
//= Notes: See problem #4 in assignment #3                                   =
//=--------------------------------------------------------------------------=
//= Build: bcc32 as3_3_4sol.c                                                =
//=--------------------------------------------------------------------------=
//= Execute: as3_3_4sol                                                      =
//============================================================================
//----- Include files --------------------------------------------------------
#include <stdio.h>             // Needed for printf()

//----- Constants ------------------------------------------------------------
#define      FALSE        0    // Boolean false
#define       TRUE        1    // Boolean true
#define    P_ERROR     0.01    // Probability of a bit error
#define  BLOCK_LEN       40    // Number of bits in a block
#define NUM_TRIALS 10000000    // Number of trials to run

//----- Function prototypes --------------------------------------------------
double rand_val(int seed);     // RNG

//===== Main program =========================================================
int main(void)
{
  int    error_count;          // Error count for a message
  int    error[BLOCK_LEN + 1]; // Error count vector
  int    lastBitInError;       // Flag for last bit in error
  int    i, j;                 // Loop counters

  // Initialization error vector to zero
  for (i=0; i<BLOCK_LEN; i++)
    error[i] = 0;

  // Seed the RNG
  rand_val(1);

  // Main simulation loop
  printf("Simulation is running for %d trials... \n", NUM_TRIALS);
  lastBitInError = FALSE;
  for (i=0; i<NUM_TRIALS; i++)
  {
    error_count = 0;
    for (j=0; j<BLOCK_LEN; j++)
    {
      // If last bit was not in error, then Pr[bit error] = p
      if (lastBitInError == FALSE)
      {
        if (rand_val(0) < P_ERROR)
        {
          error_count++;
          lastBitInError = TRUE;
        }
```

```
        else
          lastBitInError = FALSE;
      }
      // If last bit was in error, then Pr[bit error] = 0.5
      else if (lastBitInError == TRUE)
      {
        if (rand_val(0) < 0.5)
        {
          error_count++;
          lastBitInError = TRUE;
        }
        else
          lastBitInError = FALSE;
      }
    }

    // Increment appropriate element in error vector
    error[error_count]++;
  }

  // Determine and output probabilities of interest
    printf("  Pr[0 errors]  = %f \n", (double) error[0] / NUM_TRIALS);
    printf("  Pr[1 errors]  = %f \n", (double) error[1] / NUM_TRIALS);
    printf("  Pr[2 errors]  = %f \n", (double) error[2] / NUM_TRIALS);
    printf("  Pr[3 errors]  = %f \n", (double) error[3] / NUM_TRIALS);

  return(0);
}

//===========================================================================
//= Multiplicative LCG for generating uniform(0.0, 1.0) random numbers    =
//=   - x_n = 7^5*x_(n-1)mod(2^31 - 1)                                      =
//=   - With x seeded to 1 the 10000th x value should be 1043618065       =
//=   - From R. Jain, "The Art of Computer Systems Performance Analysis," =
//=     John Wiley & Sons, 1991. (Page 443, Figure 26.2)                  =
//=   - Use seed == 0 to generate random value, use seed > 0 for seeding  =
//===========================================================================
double rand_val(int seed)
{
  const long  a =      16807;  // Multiplier
  const long  m = 2147483647;  // Modulus
  const long  q =     127773;  // m div a
  const long  r =       2836;  // m mod a
  static long x;               // Random int value (this is the seed)
  long        x_div_q;         // x divided by q
  long        x_mod_q;         // x modulo q
  long        x_new;           // New x value

  // Seed the RNG
  if (seed > 0) x = seed;

  // RNG using integer arithmetic
  x_div_q = x / q;
  x_mod_q = x % q;
  x_new = (a * x_mod_q) - (r * x_div_q);
  if (x_new > 0)
```

```
    x = x_new;
  else
    x = x_new + m;

  // Return a random value between 0.0 and 1.0
  return((double) x / m);
}
```