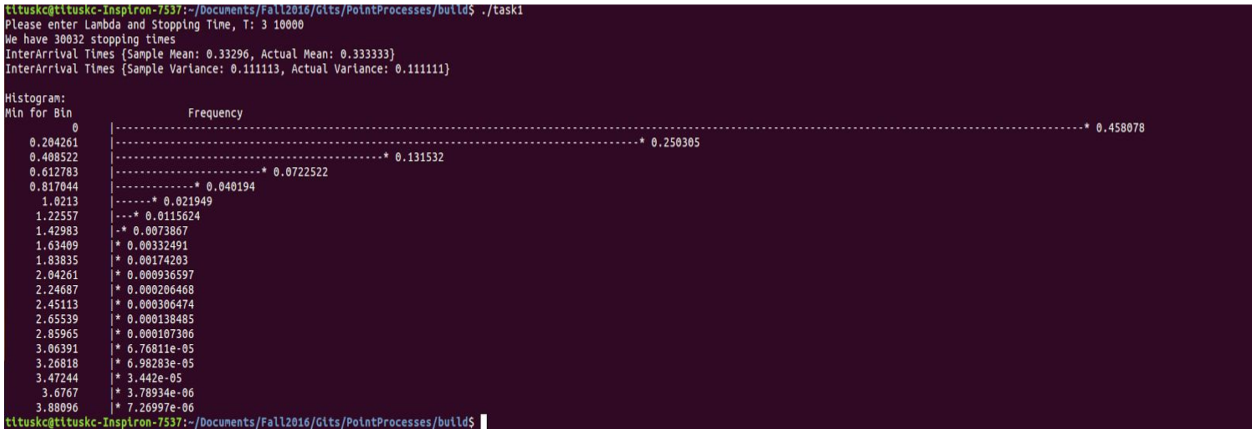
Discrete Time Stochastic Processes with Finite Time Space

The programs outlined in this assignment focus on generating discrete time stochastic processes including the Poisson process, Markov Chains, and Random Walk. Both of these simulations are considered in the finite state space and modelled using the C++ language. The theory and pseudocode for this exercise were obtained from *PoissonProcess.pdf* and *MarkovChain.pdf* files[1]. To run the code packaged, please execute *./runPointProcesses* at the root of the project. This will generate all the executables for the project in *./project/build.* To execute one of them, just go into the build directory and run ./task* (*=1,2,3,4,5).

Task 1: Simulate a Poisson process at the rate $\lambda > 0$ up to T > 0

For this exercise, we simulate a poisson process with input parameters of Lambda and stopping time, T. We also check that the interarrival times are exponentially distributed with mean = $1/\lambda$ and variance = $1/\lambda^2$. We even plot a histogram to demonstrate this fact. Here is the a snapshot of the results:



```
tituskc@tituskc-Inspiron-7537:~/Documents/Fall2016/Gits/PointProcesses/build$ ./task1
Please enter Lambda and Stopping Time, T: 3 10000
We have 30032 stopping times
InterArrival Times {Sample Mean: 0.33296, Actual Mean: 0.333333}
InterArrival Times {Sample Variance: 0.111113, Actual Variance: 0.111111}

Histogram:
Min for Bin              Frequency
        0     |------------------------------------------------------------------------------------------------------------------------------------------------------------* 0.458078
 0.204261     |-------------------------------------------------------------------------------------* 0.250305
 0.408522     |--------------------------------------------* 0.131532
 0.612783     |------------------------* 0.0722522
 0.817044     |-------------* 0.040194
   1.0213     |------* 0.021949
  1.22557     |---* 0.0115624
  1.42983     |-* 0.0073867
  1.63409     |* 0.00332491
  1.83835     |* 0.00174203
  2.04261     |* 0.000936597
  2.24687     |* 0.000206468
  2.45113     |* 0.000306474
  2.65539     |* 0.000138485
  2.85965     |* 0.000107306
  3.06391     |* 6.76811e-05
  3.26818     |* 6.98283e-05
  3.47244     |* 3.442e-05
   3.6767     |* 3.78934e-06
  3.88096     |* 7.26997e-06
tituskc@tituskc-Inspiron-7537:~/Documents/Fall2016/Gits/PointProcesses/build$ 
```

We see that the histogram represents exponential distribution, and the moments are very close to the theoretical values. E.g for λ =3 and T=10000, theoretical mean =0.333333 while sample mean = 0.33296 and theoretical variance = 0.111111 while sample variance=0.11113.

Task 2: Generate Two-State Markov Chain

A two-state Markov chain will apply a 2X2 transition matrix to a given outcome to generate either of two outcomes. More details about the generation process can be found in the pdf files mentioned above. We have implemented a method that takes in the number of steps as an input and outputs an array of Markov chains. Here is the code:

```
Matrix: {{0.3, 0.7},{0.5,0.5}}; Outcomes {0, 1} Initial outcome = 0
std::vector<int> generateTwoStepMarkovChain(int numberOfSteps) {
    int X = S2D[0];
    std::vector<int> markov;
    for (int k = 0; k < numberOfSteps; ++k) {
        markov.push_back(X);
```

[1] Attachments in the FE-522A class module

```
        double random = randomWithDefaultZeroToOne();
        if (X == S2D[0] && random > P[0][0]) {
            X = S2D[1];
        } else if (X == S2D[1] && random <= P[1][0]) {
            X = S2D[0];
        }
    }
    return markov;
}
```

, where *randomWithDefaultZeroToOne*() generates a random number between 0.0 and 1.0. *randomWithDefaultZeroToOne*() is designed so that it can also accept any range of values from the user. We have utilized default parameters here.
(in header file)

```
double randomWithDefaultZeroToOne(double min = 0.0, double max = 1.0);
```

(Implementation)

```
double randomWithDefaultZeroToOne(double min, double max) {
    double range = (max - min);
    double div = RAND_MAX / range;
    return min + (rand() / div);
}
```

Here is a sample result for n=20: 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1

Task 3: Calculate frequency of state 1 for m>n
For this exercise we determine the probability of a state generated by the Markov chain. Specifically, we consider the probability of state 1. To do this, we generate m sample arrays of Markov chains each containing n values, where m>n. The code to generate Markov chain was explored in Task 1. Here is the code to simulate the probability calculation.

```
double probability(int n, int m, bool twoStepMarkov) {
    int m = n + 1;
    double sum = 0.0;
    for (int i = 0; i < m; i++) {
        std::vector<int> markov = twoStepMarkov ? generateTwoStepMarkovChain(n) :
generateThreeStepMarkovChain(n);
        int num = count(markov, 1);
        sum += (double) num / n;
    }
    return sum / m;
}
```

I have added a method *probability(int, bool)* that receives the data size and indicator of whether to generate two state Markov of three state Markov. The code needed for part of Task 4 is essentially the same as that needed for this task except for the actual Markov chain generation. So we reuse code. Here are the results:

```
M       N        Probability
2       1             0
11      10         0.527273
101     100          0.58505
1001    1000          0.583076
```

We observe that for large n and consequently large m, the probability of appearance of state 1 converges to approximately 0.6. This corresponds to the average probability of P(1|0) and P(1|1). That is (0.7 + 0.5)/2 = 0.6.

Task 4: Generate Two-State Markov Chain and Observe frequency of state 1 for m =1 and initial state X_0=0

The steps to generate three state Markov chain here is similar to the two state process except for the fact that there are three possible outcomes of each step driven by three probabilities determined based on the previous outcome and an array of transition probabilities.

The main method in *task4.cpp* prompts the user to enter the number of steps they need for the markov chain before generating the values. Here is the code to generate the three step Markov:

```
std::vector<int> generateThreeStepMarkovChain(int numberOfSteps) {
  int X = S3D[0];
  std::vector<int> markov;
  markov.push_back(X);
  for (int k = 0; k < numberOfSteps; ++k) {
    double random = randomWithDefaultZeroToOne();
    if (X == S3D[0] && random > P3D[0][0]) {
      X = S3D[random <= (P3D[0][0] + P3D[0][1]) ? 1 : 2];
    } else if (X == S3D[1]) {
      if (random <= P3D[1][0]) {
        X = S3D[0];
      } else if (random > (P3D[1][0] + P3D[1][1])) {
        X = S3D[2];
      }
    } else if (random <= (P3D[2][0] + P3D[2][1])) {
      X = S3D[(random <= P3D[2][0] ? 0 : 1)];
    }
    markov.push_back(X);
  }
  return markov;
}
```

Here is a sample outcome for 20 steps:

```
0, 2, 1, 0, 1, 2, 1, 2, 1, 0, 2, 1, 1, 1, 0, 1, 1, 2, 1, 2
```

Similarly to task 3, we simulate the frequency of state 1 but in this case for just one random path. I choose a path with 1,000,000 steps. We find that the frequency of state 1 given 10,000,000 steps is about 0.475. This is close to the expected value of (P(1|0) + P(1|1) + P(1|2))/3 = (1/3+1/2+4/7)/3 = 0.468.

Task 5: Simulating Simple Random Walk

For this exercise, we simulate a simple random walk whereby each step can result in either +1 or -1 to the previous cumulative outcome with $P(X_n = X_{n-1}+1) = 2/3$. This process can therefore be thought of as a Markov chain with state space i = ± 1, ± 2, ± 3,... $\pm \sqrt{n}$. The points achieves are within $\sqrt{n}$ of 0 when p=1/2. And therefore this space is finite. Here is the code to generate the random walk:

```
vector<int> randomWalk(unsigned long steps) {
   vector<int> v(steps);
   return drunkWalk(0, 0.666666666666666, v);
}
vector<int> drunkWalk(int n, double probability, vector<int> vector1)
{
   if(n >= (int)vector1.size())
   {
      return vector1;
   }
   else
   {
      double type = randomWithDefaultZeroToOne();
      int prev = n==0 ? 0 : vector1[n - 1];
      vector1[n] = type < probability ? prev + 1 : prev - 1;
      return drunkWalk(++n, probability, vector1);
   }
}
```

We have used recursion to generate the desired number of points. A sample result for 20 steps is: { -1, -2, -1, -2, -1, 0, 1, 2, 3, 2, 3, 4, 5, 6, 7, 8, 9, 8, 7, 8 }