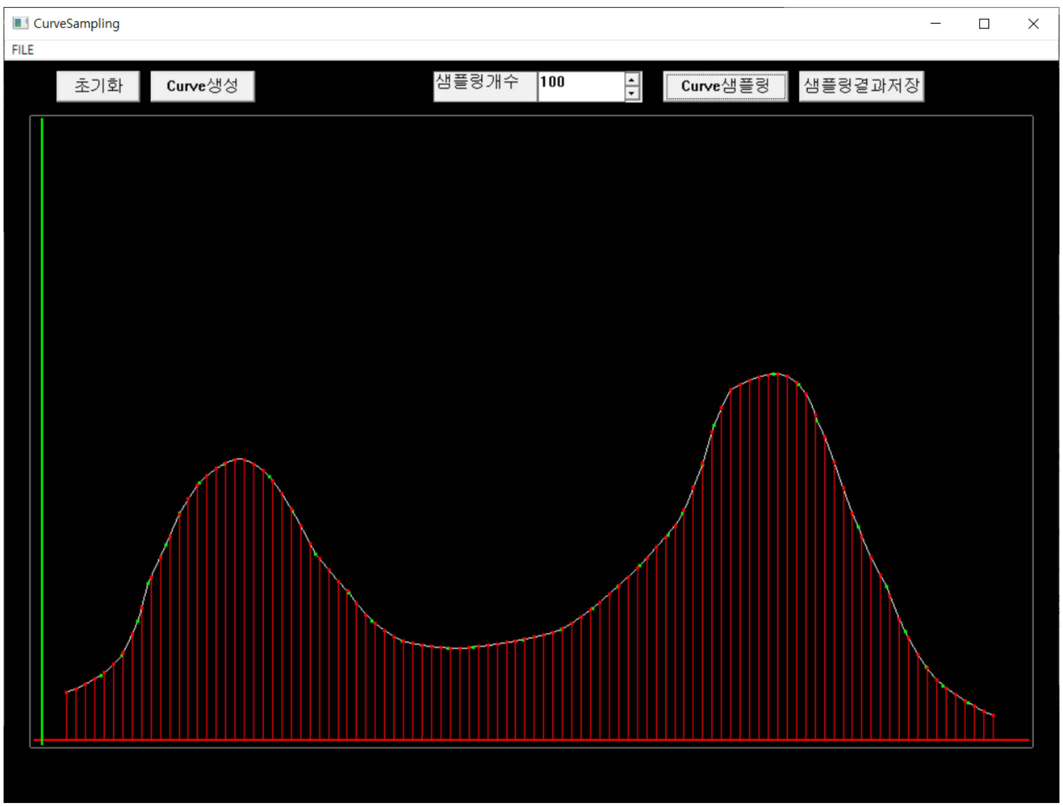
5주차 보고서

Created	@October 20, 2021 10:55 PM
Created by	**20190258 김혜린**

실습 2-1

5주차 실습에서 사용할 확률 밀도함수를 GUI 소프트웨어를 사용해 설계한 곡선은 아래와 같다.



위의 곡선을 통해 얻은 x, y(f(x)) 샘플링 데이터(sampling.txt)를 프로그램 실행파일이 저장된 폴더에 sampling_table.txt로 저장해준 뒤, main에서 `program2_1()`을 실행해주면 파일을 읽어 n과 h 값을 바탕으로 생성된 n개의 데이터를 normalization한다.

1. x 값을 [0, 1] 사이로 normalization ($(n-1)\times h=1$)을 해준다.
2. 재배치된 x 값을 바탕으로 $x_0=0$ 에서 $x_n=1$ 사이의 적분 값이 1이 되도록 y값을 normalization한다.

이렇게 normalization한 pdf_table의 $(x_i, p_x(x_i))$ 쌍은 새로운 `normed_h` 값과 함께 pdf_table.txt에서 확인할 수 있고 my_function.cpp 에 정의된 cdf 함수 `double F(double x, double* y, double h, int n);` 를 이용해 구간별 pdf 적분 값을 구할 수 있다. 함수 F의 parameter는 아래와 같다.

- `double x`
  - $\int_{x_0}^{x} p_t(t)dt$ 에서의 x 값
- `double *y`
  - pdf_table의 샘플링 데이터 $p_x(x)$이 저장된 double 배열
- `double h`
  - pdf의 샘플링 데이터의 간격
- `int n`
  - 샘플링 데이터의 개수

구간 $[x_1, x_2]$ 의 F 값을 구하기 위해서는 $F(x_2, y, h, n) - F(x_1, y, h, n)$를 이용해 구간별 pdf 적분 값을 구해야 한다.

실습 2-2

위의 과정으로 구한 pdf_table.txt 를 이용해 main에서 `program2_2()` 를 실행해 [0, 1]의 균등 분포를 가지는 난수 u_i m개를 inversion 방법을 통해 우리가 만든 pdf 분포를 따르는 난수 x_i m개로 바꾸어준다.

`program2_2()` 가 실행되면 '*Enter the number of random numbers*' 라는 메세지와 함께 사용자로부터 콘솔창을 통해 생성하고자 하는 난수 개수를 입력 받고 입력 받은 난수 개수 `m` 개만큼 `rand()` 를 이용해 0에서 1사이의 난수를 생성한 뒤, my_function.cpp에 정의된 `bisection` 을 이용해 $F_x(x_i) = u_i$ 를 만족하는 x_i를 구한다. 생성된 m개의 x 값은 $\text{random\_event\_table.txt}$ 에서 m값과 함께 확인할 수 있다.

`bisection` 함수의 경우 비선형 방정식 `_f` 에 대해 [0, 1]을 초기구간으로 하여 bisection 방법을 이용해 해를 구하는 함수로 prototype은 `double bisection(double* y, double h, double u, int n);` 이며 각각의 파라미터가 의미하는 바는 다음과 같다.

- `double *y`
  - pdf_table의 샘플링 데이터 $p_x(x)$이 저장된 double 배열

- `double h`
    - pdf의 샘플링 데이터의 간격
- `double u`
    - [0, 1]사이의 난수
- `int n`
    - 샘플링 데이터의 개수

숙제 2-1

`program2_3_1()` 을 통해 임의의 λ에 대해 inversion 방법을 이용해 λ에 해당하는 지수 분포를 따르는 사용자가 정한 개수만큼의 난수가 이론적인 평균, 분산값을 따르는지 확인해보았다

실험에 사용한 λ 값은 아래와 같다.

1. 0.5
2. 1.0
3. 2.5

각각의 λ는 이론상 다음과 같은 평균, 분산 값을 가진다.

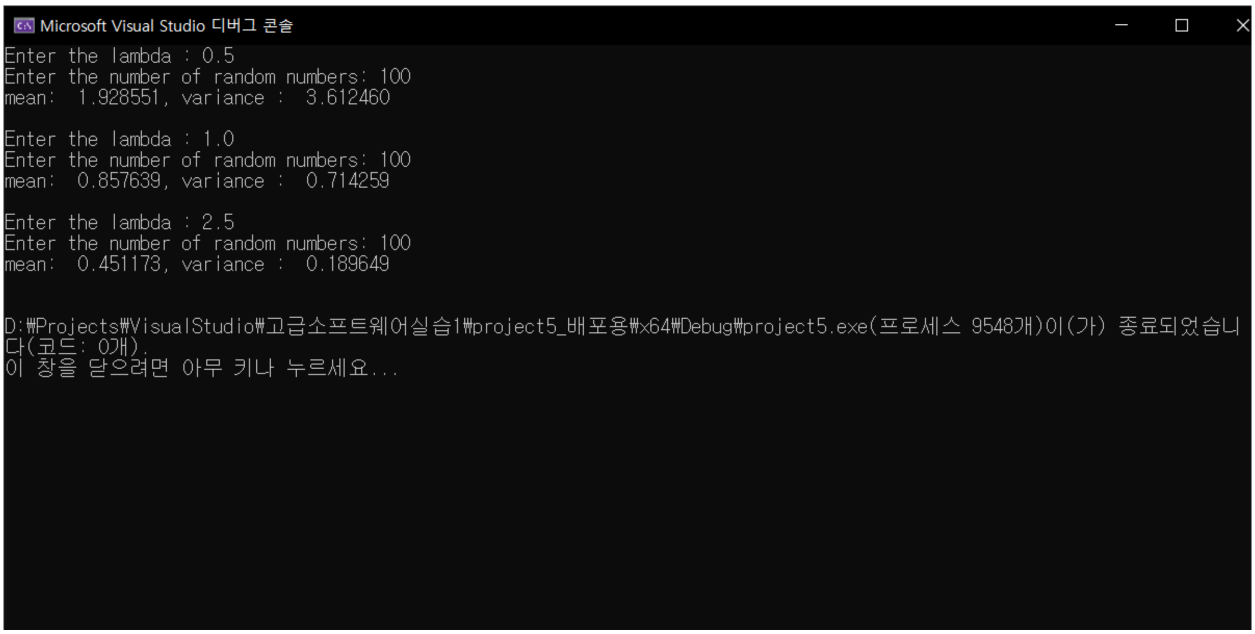
1. 0.5
    a. mean값: 2
    b. variance값: 4
2. 1.0
    a. mean값: 1
    b. variance값: 1
3. 2.5
    a. mean값: 0.4
    b. variance값: 0.16

`program2_3_1()`가 실행되면 콘솔창에는 사용자에게 λ와 생성하고자 하는 난수 개수 `Nr`을 입력하라는 메세지가 뜬다.

입력받은 λ와 `Nr`을 이용해 [0,1] 사이의 균등 분포를 따르는 난수 `u`를 생성한 뒤, $-\frac{ln(1-u)}{\lambda}$ 를 계산하여 지수 분포를 따르는 확률 밀도 함수와 일치하는 `xu`값을 구한다.

`Nr`만큼 생성한 난수 `xu`의 평균과 분산을 확인하기 위해 `mean` 과 `squared_m` 변수를 이용해 $E(X)$와 $E(X^2)$ 를 구한 뒤, $Var(X) = E(x^2) - E(X)^2$ 를 만족하는 분산 값 `var` 를 계산하여 콘솔창에 `mean` 과 `var` 를 출력하였다.

실험에서 난수 개수 `Nr` 은 100, 1000, 10000, 20000, 100000, 500000 순서로 진행하였고 실험 결과는 아래와 같다.

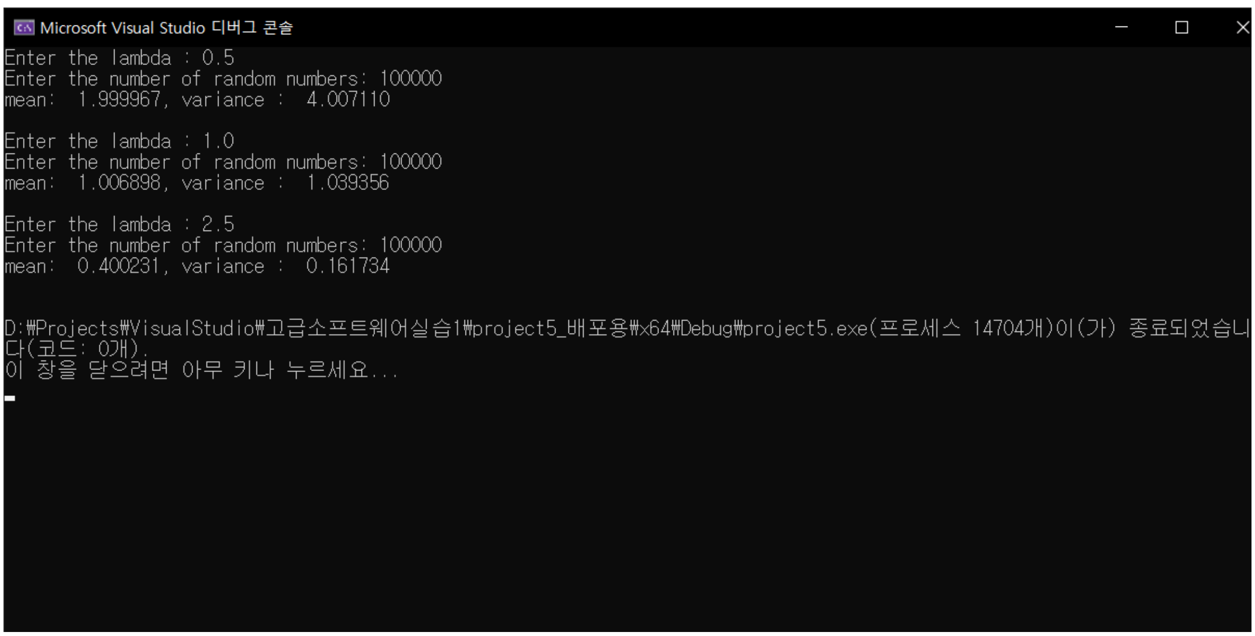


100개로 실험한 결과

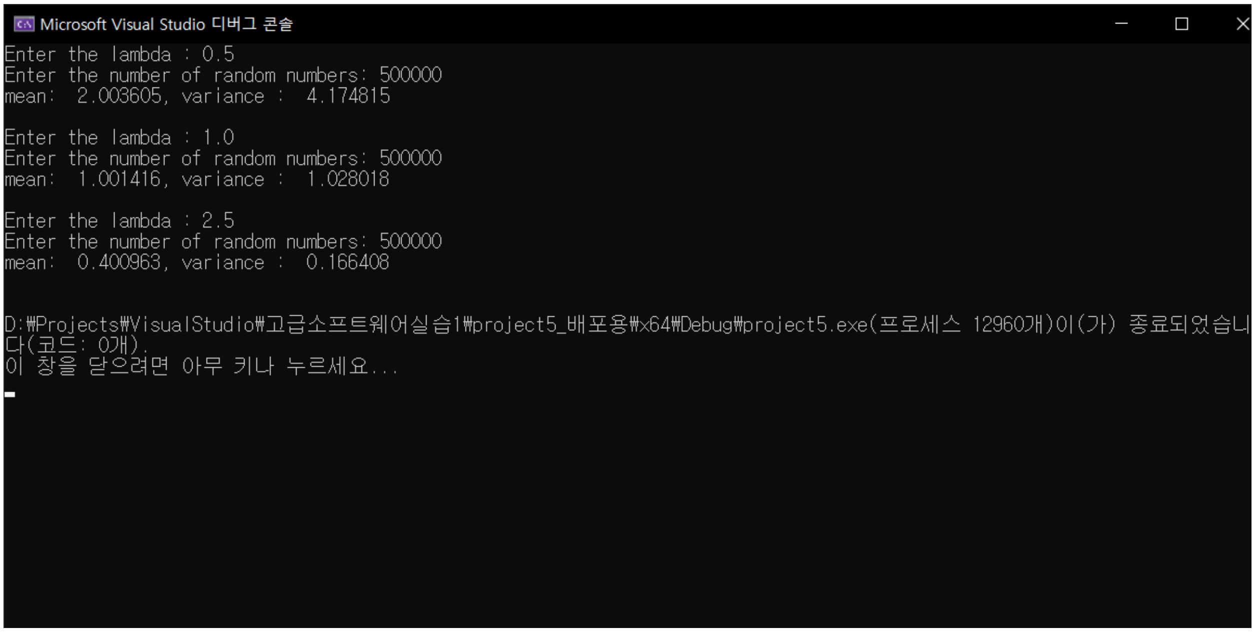
1000개로 실험한 결과

10000개로 실험한 결과

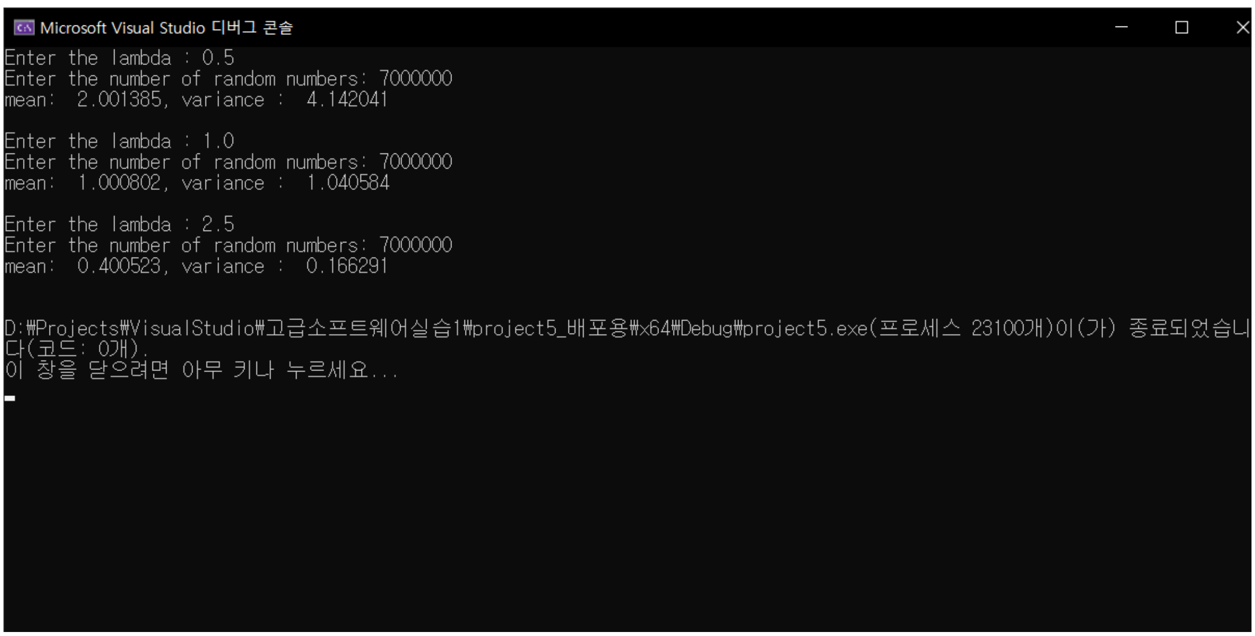
20000개로 실험한 결과



100000개로 실험한 결과



500000개로 실험한 결과



7000000개로 실험한 결과

난수 개수 10000개에서부터 평균 값 오차가 0.02에서 0.01 이내로 줄어드는 것을 확인할 수 있다. 하지만 분산은 아직 오차가 많이 발생하는 것을 확인할 수 있고, 10000개에서부터 소수점 두 세자리 범위 안에서 분산이 일치하는 것을 확인할 수 있다. 500000개로 실험한 결과에서부터 오차가 소수점 세자리 밖에서 발생하는 것을 확인할 수 있다.

따라서 10000개부터 통계적으로 구한 평균값과 분산값이 이론적인 평균값과 분산값과 충분히 일치한다고 분석할 수 있다.

숙제 2-2

숙제 2-2 (i)

(i)에서는 실습 2-2에서 설계한 `program2_2()`를 다듬어 `program2_2_a` 함수로 구현하였다. `prgrogram2_2_a` 함수의 prototype은 `void program2_2_a(FILE* fp_w, double* y, double h, int n, int m);` 이며 각 파라미터는 아래와 같은 의미를 가진다.

- `FILE *fp_w`
    - 생성된 난수 x를 기록할 파일 포인터
- `double *y`
    - $\mathrm{pdf\_table.txt}$ 에 저장된 샘플링 데이터의 $p_x(x)$ 배열
- `double h`
    - $\mathrm{pdf\_table.txt}$ 에 저장된 샘플링 데이터 간격
- `int n`
    - 샘플링 데이터 개수
- `int m`
    - 사용자가 입력한 생성하고자 하는 난수 개수

`program2_2_a` 함수를 구현하면서 기존에 `F` 함수를 다듬었는데 기존의 실습에서는 파라미터 `x` 에 대해 $x \in [x_{m-1}, x_m]$을 만족하는 `m` 을 while문을 이용해 `xm+h > x` 를 break 구문으로 하여 구한 뒤, `m` 에 해당하는 샘플링 데이터까지 합성 사다리꼴 공식을 이용해 적분값을 구하고 나머지 `x-xm` 에 해당하는 범위의 적분 값(A)은 아래 식을 이용해 구해주었다.

$$(p_{x_m} + \frac{p_{x_{m+1}} - p_{x_m}}{x_{m+1} - x_m} \cdot \frac{x - x_m}{2})(x - x_m)$$

여러 번의 실험을 통해 난수 `u` 의 설계와 실제 값이 다른 것을 확인했다. 설계상 `DELTA` 를 통해 `rand()` 가 가질 수 있는 최댓값 32767에 `DELTA` 를 더해 mod 연산 시, 1이 나오지 않도록 하였

는데 실제로는 my_solver.h 에 설정된 `DELTA` 값이 매우 작았기 때문인지 `u` 값으로 1이 나오는 경우가 생겼다.

따라서 이를 처리하기 위해 함수 `F` 에서 `m` 이 샘플링 데이터의 개수 `n` 과 일치할 경우 A에 대한 연산은 처리하지 않도록 수정하였다. 비선형 방정식 `_f`의 경우 my_function.cpp 에 `F` 연산에 필요한 `x`, `y`, `h`, `n`뿐만 아니라 `u`까지 추가로 파라미터로 받아 `F(x, y, h, n)-u` 값을 반환하도록 설계되었다.

또한 실습에서는 math.h 헤더 파일의 `fabs` 함수를 이용해 두 함수 값이나 x 값의 차이를 계산하였는데 my_solver.h 에 설정된 `DELTA` 와 `EPSILON` 값이 기존의 값보다 훨씬 작았기 때문에 직접 두 값의 차이가 `-DELTA`와 `DELTA`, `-EPSILON`과 `EPSILON` 사이에 들어오는지 확인하였다.

숙제 2-2 (ii)

(ii)에서는 (i)이 앞으로 구현할 난수 생성함수가 올바르게 난수를 생성하는지 통계적으로 확인하는 함수를 설계하였다. `program2_3_2` 함수에 구현되어 있으며 함수가 실행되면 pdf_table.txt 를 읽어 함수 실행에 필요한 `y` , `n` 과 `h` 값을 읽어들이고 결과 값을 작성할 함수 포인터 `fp_w` 를 생성해 함수 포인터 `_f2_3_2` 에 사용자에게 콘솔창을 통해 입력 받은 난수 개수 `m` 과 함께 인자로 넘겨준다.

메인에서 난수 생성 함수로 사용할 함수를 변수 `_f2_3_2` 에 설정하고 `program2_3_2` 함수를 호출하면 해당 함수에 할당된 함수를 이용해 난수를 생성한다.

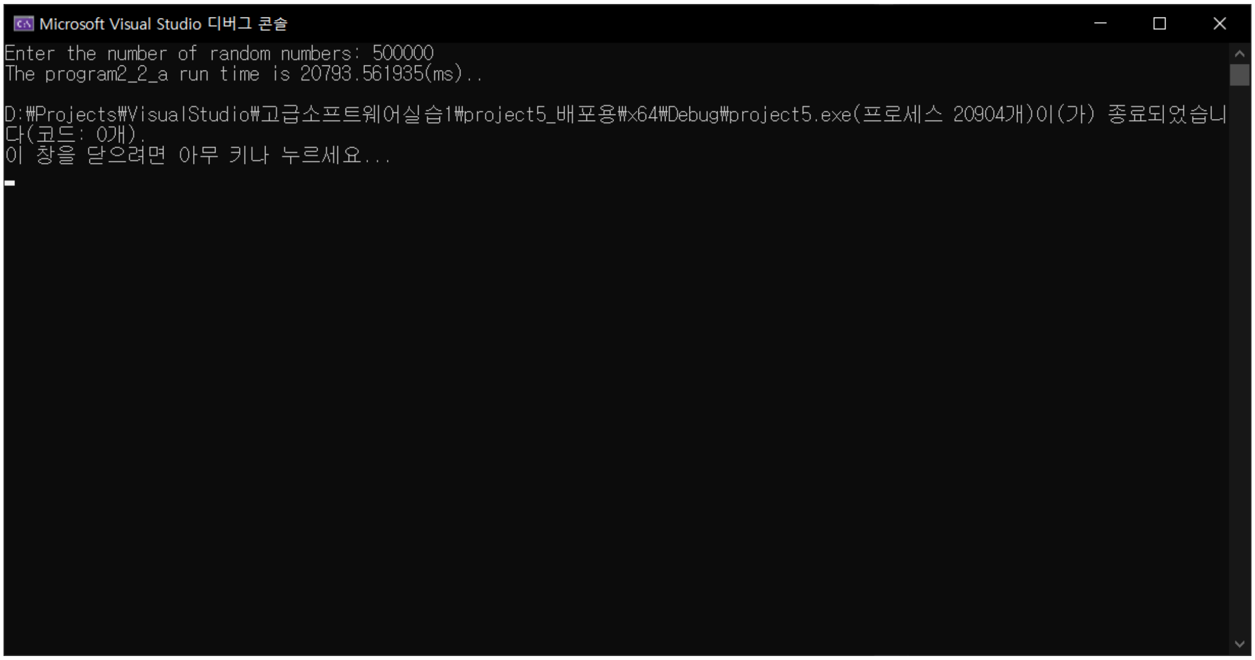
`_f2_3_2` 를 통해 생성된 난수들은 random_event_table.txt 에 저장되어 있다. 따라서 `program2_3_2` 에서는 `_f2_3_2` 가 수행되고 난 뒤, 파일 읽기 포인터 `fp_r` 에 해당 파일을 할당해 생성된 난수를 읽어들여 통계적으로 올바르게 생성되었는지 확인한다.

프로그램이 올바르게 작동했는지는 histogram을 이용해 확인하였는데 $[x_0, x_n] = [0, 1]$ 을 n 개의 구간으로 나누어 각 구간에 난수가 몇 개 존재하는 카운팅하였다.

어떤 구간 $S_i = [x_i, x_{i+1}]$ 에 존재하는지는 해당 난수를 샘플링 데이터 간격인 `h` 로 나눈 몫을 통해 찾아내었다. 몫 i 가 구간 S_i 의 i와 일치하며 이렇게 카운팅한 난수 개수를 `his[n]` 배열에 저장하였다.

구간 S_i 의 이론상 난수 생성 확률은 `F` 함수를 이용해 x_i 에서 x_{i+1}까지 적분한 값을 해당 값으로 사용하였고 통계상 난수 생성 확률은 `his[i]/m` 을 이용해 구하였다.

두 값을 구간과 함께 histogram.txt 에 출력하였다.



500000개를 난수 개수로 설정하여 실험한 콘솔창

histogram.txt - Windows 메모장

[xi, xi+1)	data/Nr	F(xi+1)-F(xi)
[0.000000, 0.010101)	0.002854000000000	0.002870976927000
[0.010101, 0.020202)	0.003106000000000	0.003096486802500
[0.020202, 0.030303)	0.003416000000000	0.003379855206000
[0.030303, 0.040404)	0.003618000000000	0.003703960942500
[0.040404, 0.050505)	0.004162000000000	0.004119829213500
[0.050505, 0.060606)	0.004820000000000	0.004693889245500
[0.060606, 0.070707)	0.005646000000000	0.005578464118500
[0.070707, 0.080808)	0.006914000000000	0.006879351706500
[0.080808, 0.090909)	0.008496000000000	0.008519698551000
[0.090909, 0.101010)	0.009836000000000	0.009998479900500
[0.101010, 0.111111)	0.011282000000000	0.011181973666500
[0.111111, 0.121212)	0.012726000000000	0.012402452244000
[0.121212, 0.131313)	0.013182000000000	0.013471254205500
[0.131313, 0.141414)	0.014234000000000	0.014320369518000
[0.141414, 0.151515)	0.014952000000000	0.014993990056500
[0.151515, 0.161616)	0.015690000000000	0.015472923921000
[0.161616, 0.171717)	0.015918000000000	0.015835873053000
[0.171717, 0.181818)	0.016248000000000	0.016097196024000
[0.181818, 0.191919)	0.016186000000000	0.016194827239500
[0.191919, 0.202020)	0.016300000000000	0.016068696052500
[0.202020, 0.212121)	0.015914000000000	0.015752908489500
[0.212121, 0.222222)	0.015344000000000	0.015273691797000
[0.222222, 0.232323)	0.014876000000000	0.014609061148500
[0.232323, 0.242424)	0.013804000000000	0.013783849852500
[0.242424, 0.252525)	0.012954000000000	0.012864451782000
[0.252525, 0.262626)	0.011708000000000	0.011870806311000
[0.262626, 0.272727)	0.010722000000000	0.010925019378000
[0.272727, 0.282828)	0.010150000000000	0.010162792867500
[0.282828, 0.292929)	0.009550000000000	0.009513106648500
[0.292929, 0.303030)	0.008842000000000	0.008899743625500

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

수행 결과

500000개를 난수 개수로 하여 bisection 방법으로 난수를 생성한 결과 위와 같은 결과를 확인할 수 있으며 소수점 4자리까지 일치하는 정확도를 보이므로 프로그램이 정상적으로 작동하고 있다고 판단할 수 있다.

숙제 2-2 (iii)

`program2_2_a` 에서는 Bisection 방법을 이용해 난수 생성을 구현하였다면 `program2_2_b` 와 `program2_2_c` 에서는 각각 Secant 방법과 Newton-Raphson 방법을 이용해 난수 생성을 구현하였다.

두 함수는 [0, 1] 사이의 난수 `u` 에 대한 비선형 방정식 `_f` 를 my_function.cpp에 구현된 `bisection` , `secant` , `newton` 함수 중 어느 함수를 이용해 풀이할 것인가의 차이만 두기 때문에 `program2_2_a` 와 같은 양식의 prototype을 보인다.

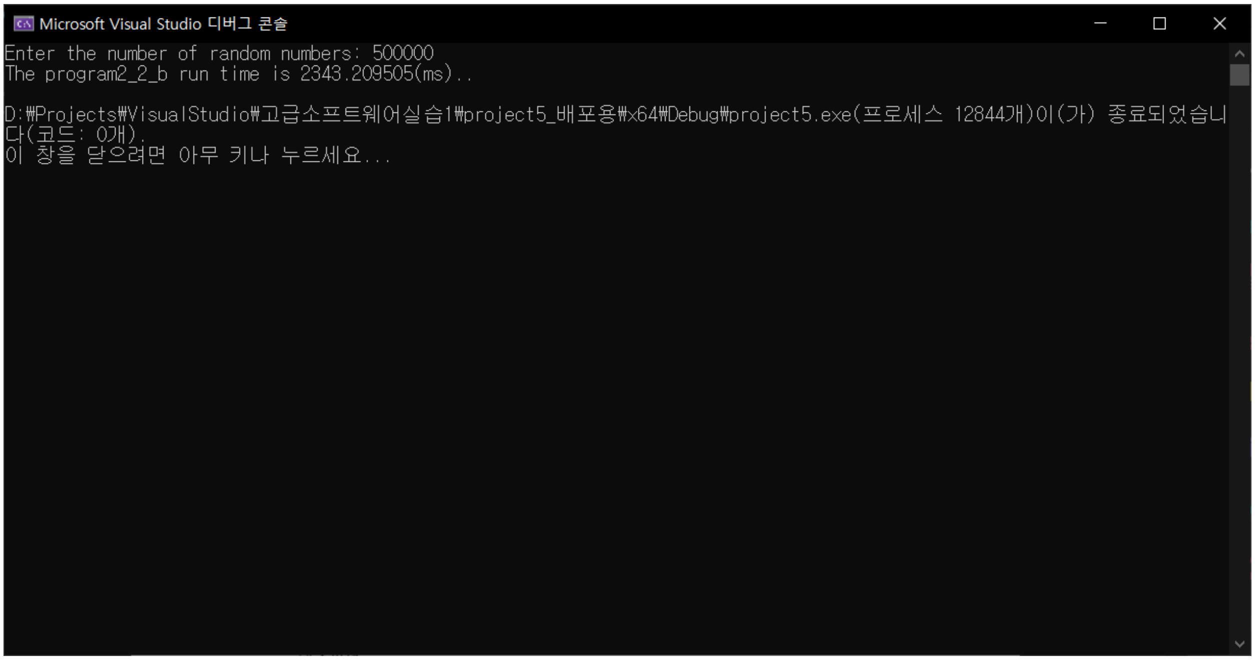
`secant` 와 `newton` 함수의 경우 `bisection` 과 같은 prototype을 가지고 있으며 초기값 설정시 bisection 방법을 이용한다.

초기구간 [0, 1]에 대해 3번의 bisection 방법 수행 후 나온 구간과 중간값을 각각 `secant` 함수와 `newton` 함수의 초기값을 설정하였으며 연산 결과 나온 x에 대한 판단은 `bisection` 방식과 동일하다.

`newton` 함수의 경우 `_f` 의 미분 함수도 필요하기 때문에 my_function.cpp 에 `_fp` 함수를 추가로 구현하였다. `_fp` 함수는 `double _fp(double x, double* y, double h, double u, int n);` 의 prototype을 가지며 파라미터로 넘어온 `x`에 대해 $x \in [x_i, x_{i+1}]$ 를 만족하는 i 를 구해 아래 식을 만족하는 s 를 구해 `_fp` 값을 반환하였다.

$$s = \frac{x - x_i}{x_{i+1} - x_i}$$
$$f'(x) = p_x(x) \approx (1 - s) \cdot P_{x_i} + s \cdot P_{x_{i+1}}$$

Secant 방법



500000개를 난수 개수로 설정하여 실험한 콘솔창

histogram.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

[xi, xi+1)	data/Nr	F(xi+1)-F(xi)
[0.000000, 0.010101)	0.00295200000000	0.00287097692700
[0.010101, 0.020202)	0.00308000000000	0.00309648680250
[0.020202, 0.030303)	0.00334400000000	0.00337985520600
[0.030303, 0.040404)	0.00373800000000	0.00370396094250
[0.040404, 0.050505)	0.00424200000000	0.00411982921350
[0.050505, 0.060606)	0.00473800000000	0.00469388924550
[0.060606, 0.070707)	0.00536800000000	0.00557846411850
[0.070707, 0.080808)	0.00672600000000	0.00687935170650
[0.080808, 0.090909)	0.00867200000000	0.00851969855100
[0.090909, 0.101010)	0.00972000000000	0.00999847990050
[0.101010, 0.111111)	0.01121800000000	0.01118197366650
[0.111111, 0.121212)	0.01195800000000	0.01240245224400
[0.121212, 0.131313)	0.18415600000000	0.01347125420550
[0.131313, 0.141414)	0.01421800000000	0.01432036951800
[0.141414, 0.151515)	0.01523000000000	0.01499399005650
[0.151515, 0.161616)	0.01530600000000	0.01547292392100
[0.161616, 0.171717)	0.01579400000000	0.01583587305300
[0.171717, 0.181818)	0.01609800000000	0.01609719602400
[0.181818, 0.191919)	0.01625400000000	0.01619482723950
[0.191919, 0.202020)	0.01591000000000	0.01606869605250
[0.202020, 0.212121)	0.01591000000000	0.01575290848950
[0.212121, 0.222222)	0.01539600000000	0.01527369179700
[0.222222, 0.232323)	0.01444400000000	0.01460906114850
[0.232323, 0.242424)	0.01364400000000	0.01378384985250
[0.242424, 0.252525)	0.57185800000000	0.01286445178200
[0.252525, 0.262626)	0.00000000000000	0.01187080631100
[0.262626, 0.272727)	0.00000000000000	0.01092501937800
[0.272727, 0.282828)	0.00000000000000	0.01016279286750
[0.282828, 0.292929)	0.00000000000000	0.00951310664850
[0.292929, 0.303030)	0.00000000000000	0.00889974362550

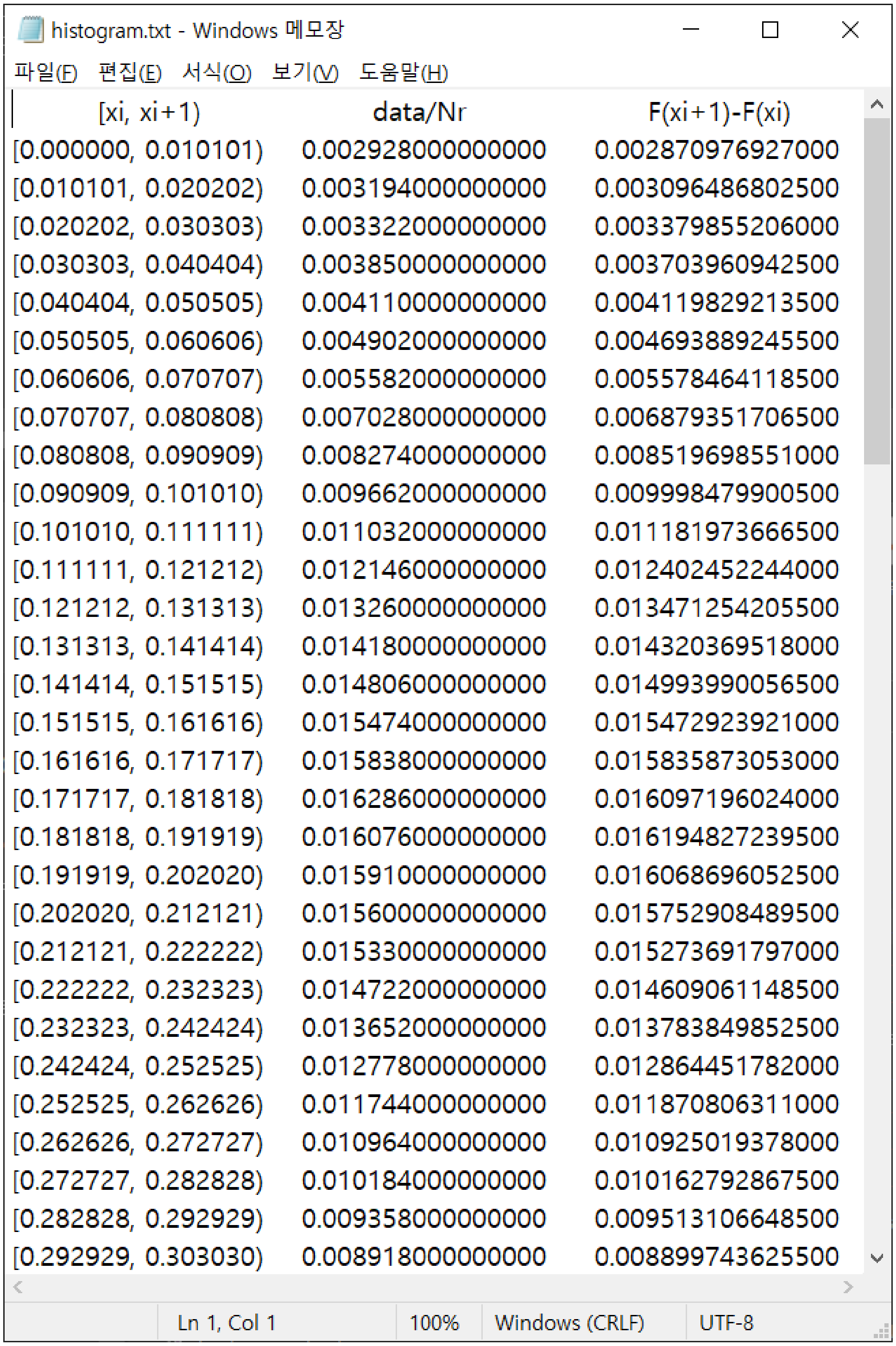
Ln 1, Col 1 100% Windows (CRLF) UTF-8

수행 결과

500000개를 난수 개수로 하여 secant 방법으로 난수를 생성한 결과 위와 같은 결과를 확인할 수 있으며 소수점 3-4자리까지 일치하는 정확도를 보이므로 프로그램이 정상적으로 작동하고 있다고 판단할 수 있다.

Newton-Raphson 방법

500000개를 난수 개수로 설정하여 실험한 콘솔창



histogram.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

[xi, xi+1)	data/Nr	F(xi+1)-F(xi)
[0.000000, 0.010101)	0.002928000000000	0.002870976927000
[0.010101, 0.020202)	0.003194000000000	0.003096486802500
[0.020202, 0.030303)	0.003322000000000	0.003379855206000
[0.030303, 0.040404)	0.003850000000000	0.003703960942500
[0.040404, 0.050505)	0.004110000000000	0.004119829213500
[0.050505, 0.060606)	0.004902000000000	0.004693889245500
[0.060606, 0.070707)	0.005582000000000	0.005578464118500
[0.070707, 0.080808)	0.007028000000000	0.006879351706500
[0.080808, 0.090909)	0.008274000000000	0.008519698551000
[0.090909, 0.101010)	0.009662000000000	0.009998479900500
[0.101010, 0.111111)	0.011032000000000	0.011181973666500
[0.111111, 0.121212)	0.012146000000000	0.012402452244000
[0.121212, 0.131313)	0.013260000000000	0.013471254205500
[0.131313, 0.141414)	0.014180000000000	0.014320369518000
[0.141414, 0.151515)	0.014806000000000	0.014993990056500
[0.151515, 0.161616)	0.015474000000000	0.015472923921000
[0.161616, 0.171717)	0.015838000000000	0.015835873053000
[0.171717, 0.181818)	0.016286000000000	0.016097196024000
[0.181818, 0.191919)	0.016076000000000	0.016194827239500
[0.191919, 0.202020)	0.015910000000000	0.016068696052500
[0.202020, 0.212121)	0.015600000000000	0.015752908489500
[0.212121, 0.222222)	0.015330000000000	0.015273691797000
[0.222222, 0.232323)	0.014722000000000	0.014609061148500
[0.232323, 0.242424)	0.013652000000000	0.013783849852500
[0.242424, 0.252525)	0.012778000000000	0.012864451782000
[0.252525, 0.262626)	0.011744000000000	0.011870806311000
[0.262626, 0.272727)	0.010964000000000	0.010925019378000
[0.272727, 0.282828)	0.010184000000000	0.010162792867500
[0.282828, 0.292929)	0.009358000000000	0.009513106648500
[0.292929, 0.303030)	0.008918000000000	0.008899743625500

Ln 1, Col 1 100% Windows (CRLF) UTF-8

수행 결과

500000개를 난수 개수로 하여 Newton-Raphson 방법으로 난수를 생성한 결과 위와 같은 결과를 확인할 수 있으며 소수점 3-4자리까지 일치하는 정확도를 보이므로 프로그램이 정상적으로 작동하고 있다고 판단할 수 있다.

숙제 2-2 (iv)

비선형 방정식 풀이 방법에 따른 난수 생성 시간을 비교할 때, 세 방법을 구현한 함수 모두 `fp_w` 에 생성된 난수를 출력하는 `fprintf(fp_w, "%.15lf\n", xu);` 를 공통적으로 수행하기 때문에 해당 구문은 실제로 난수 생성에 관여하지는 않지만 연산시간에 포함되어있다.

난수 개수 500000개에 대해 세 방법의 연산시간은 (ii)와 (iii)에서 확인할 수 있다.

- Bisection : 20793.561935ms
- Secant : 2343.209505ms
- Newton-Raphson : 4436.614513ms

연산 시간이 Secant → Newton-Raphson → Bisection으로 느려지는 것을 확인할 수 있는데 이론상 해를 찾는데 걸리는 속도는 Newton-Raphson → Secant → Bisection 순으로 느려지기 때문에 이론과 차이를 보이는 것을 확인할 수 있다.

이런 차이는 초기값 설정시, Newton-Raphson이 좀 더 정밀한 초기값을 요구하기 때문인 것으로 추측된다. Secant와 Bisection의 경우 구간을 이용해 다음 값을 추정하기 때문에 상대적으로 조금 덜 정밀하더라도 구간 조정이 빠르게 이뤄지는 반면, Newton-Raphson의 경우 x값의 미분값을 이용하기 때문에 초기값의 영향을 더 받는다고 판단할 수 있다.