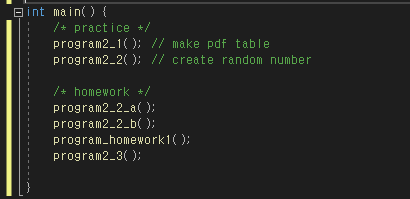
**고급 소프트웨어 실습**

**5주차 실습/과제 보고서**

**20171617 김소연 (2분반)**

1. **프로그램의 구동 방법 및 간략한 소개**

**main.cpp**



각각의 함수는 주석처리를 하면 수행되지 않는다. 주석처리를 통해 프로그램의 수행을 조절할 수 있다.

**program2\_1( )**

* sampling\_table.txt파일에서 sampling된 데이터들을 읽어와, x의 범위가 0~1로, 적분값이 1로 고정되도록 normalize를 진행한다. normalize된 데이터들을 pdf\_table.txt에 저장한다.
* sampling\_table.txt파일이 존재하면 자동으로 수행된다.

**program2\_2( )**

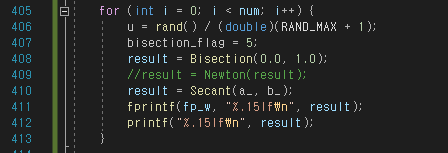
* pdf\_table.txt의 데이터를 읽어와 확률 밀도 함수를 형성하고, 이 확률밀도함수에 기반한 난수들을 생성하여 random\_event\_table.txt에 저장한다.
* 콘솔창에 생성하려는 난수의 수를 입력하면 수행된다.  
  

**program2\_2\_a( )**

* program2\_2( )를 바탕으로 가독성과 함수 연결을 개선한 함수이다. 2\_2와 기능은 같다.
* 콘솔창에 생성하려는 난수의 수를 입력하면 수행된다.  
  

**program2\_2\_b( )**

* program2\_2( )를 바탕으로하여, Bisection method가 아닌 newton method혹은 secant method를 통해 근을 찾는다
* 콘솔창에 생성하려는 난수의 수를 입력하면 수행된다.

program2\_3.cpp  
.

* 409 ~ 410라인에서 newton method, secant method를 조절할 수 있다. 해당하는 부분을 주석 처리해서 Newton, Secant method를 선택한다.

**program2\_3( )**

* 0 ~ 1까지의 구간을 n개의 공간으로 나누어, random\_event\_table.txt에 저장된 난수들을 바탕으로 해당 범위에 해당되는 난수의 수와 전체 난수의 수에 대한 비율을 출력해준다. 그리고 pdf\_table.txt를 읽어 범위에 대해 p(x)를 적분한 결과를 출력해준다. 이에 해당하는 내용들을 historgram.txt에 저장한다.
* 나누려는 공간의 수를 콘솔창에 입력하면 수행된다  
  

**program\_homework1( )**

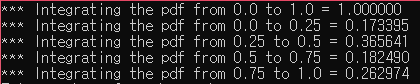
* 지수 분포의 확률에 기반하여 난수를 생성해주고, 생성된 난수들의 기댓값과 분산을, 예상되는 기댓값과 분산과 비교해준다.
* 콘솔창에 생성하려는 난수의 수와 지수 분포를 형성하는 lambda를 입력하면 수행된다.  
  

1. **실습**

**실습문제 2-1**

sampling\_table.txt파일에서 sampling된 데이터들을 읽어와, x의 범위가 0~1로, 적분값이 1로 고정되도록 normalize를 진행한다. normalize된 데이터들을 pdf\_table.txt에 저장한다.

normalize가 잘 수행되었는지를 확인하기 위해, 0~1까지의 구간과 일정 구간들을 나누어 부분적분을 수행하였다.



0.0부터 1.0까지의 적분결과는 1, 그리고 4등분한 구간들의 합이 1에 근사하는 것을 확인할 수 있었다. 이를 통해 normalize가 잘 수행되었다고 판단할 수 있다.

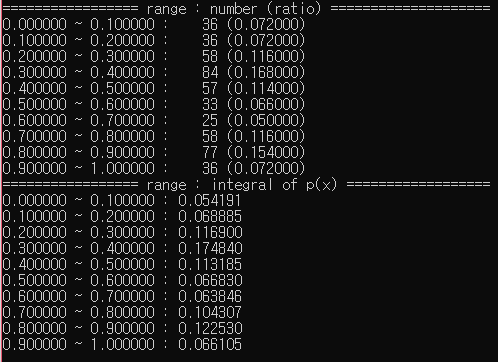
**실습문제 2-2**

실습 2-1에서 생성한 normalize한 data가 저장된 pdf\_table.txt를 읽어와 이 확률밀도함수에 기반한 random number를 생성한다.

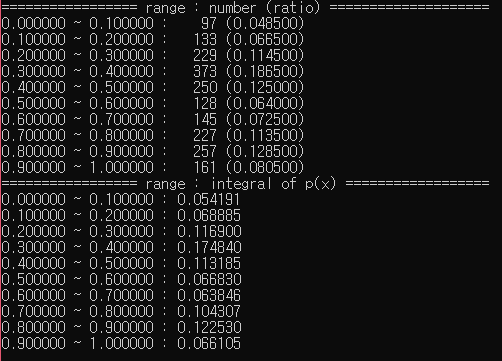
이를 위해서는 확률 밀도 함수 p(x)를 적분한 P(x)의 적분함수가 필요했는데, 이를 구할 수 없었기에 사다리꼴 공식을 이용하여 근사한 수치를 산출하였다.

P(x)의 역함수를 구하는 대신, 0과 1사이의 random한 수를 생성하여(u) f(x) = P(x) – u의 근을 구하는 것으로 확률 밀도 함수에 기반한 random number를 얻을 수 있었다. 이 때, 근을 구하는 데에 사용한 알고리즘은 bisection method였다.

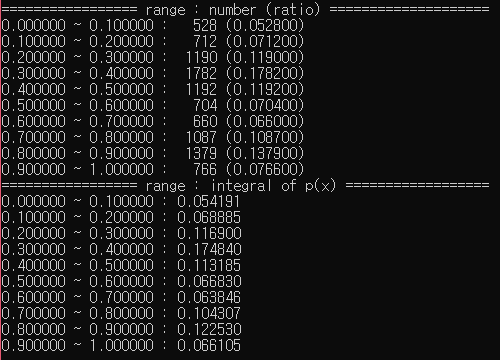
확률 밀도 함수에 기반한 난수가 생성되었는지는 program2\_3에서 구현한 히스토그램을 출력하는 함수를 이용하였다. (아래 참고)



random number : 500



random number : 2000



random number : 10000

(매번 수행마다 새롭게 난수를 생성하고 10개 구간의 히스토그램을 생성해주었다)

random number의 수를 500, 2000, 10000개로 입력해보았다.

난수의 수가 많아질수록, 아래의 p(x)에 대한 부분적분 결과와 비교하였을 때 그 정확도가 높아지는 것을 확인할 수 있었다.

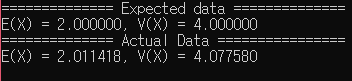
이를 통해 함수가 잘 구현되었음을 알 수 있었으며, 입력하는 난수의 수가 커질수록 그 정확도가 높아지는 것을 알 수 있다.

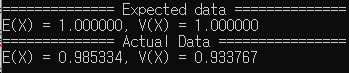
**숙제문제 2-1 - program\_homework1( )**

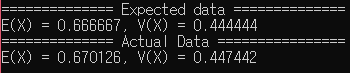
지수 분포의 확률에 기반하여 난수를 생성해주고, 생성된 난수들의 기댓값과 분산을, 예상되는 기댓값과 분산과 비교해주는 프로그램이다.

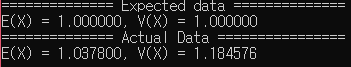
난수의 수와 인자 lambda를 입력받고, 입력받은 난수의 수만큼, 랜덤한 수 u에 대한 x의 수식을 통해 난수를 얻고 이를 화면에 출력해주었다.

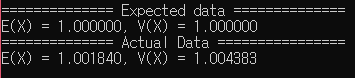
이 때 만들어진 난수들을 통해 기대값과 분산을 계산하고, 이를 이론값과 비교해보고, 난수들이 지수 분포 확률에 기반하여 생성되었는지를 점검하였다.

  
random number : 5000, lambda = 0.5

  
random number : 5000, lambda = 1.0

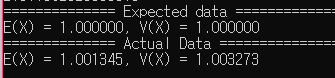
  
random number : 5000, lambda = 1.5

  
random number : 500, lambda = 1.0

  
random number : 50000, lambda = 1.0

lambda 값을 달리해가며 관찰해봤을 때, 기댓값과 분산은 적은 오차를 갖고 예상 값에 근사하는 것을 확인할 수 있었다.

특히 이러한 오차는, random number의 수가 커질수록 적어지는 경향을 위의 결과 이미지를 통해 확인할 수 있다.



random number : 1,000,000, lambda = 1.0

50000개였을 때와 비교하면 오차가 적어지긴 했지만, 난수의 수가 20배가 된 것에 비하면 그리 유의미한 숫자는 아닌 것으로 판단했다. 이는 수가 더 커져도 마찬가지이다. 따라서 난수의 개수가 많아질수록 오차는 줄어들지만, 50000개 가량의 난수를 생성했을 때, 예상 값에 비교적 근접한 값을 얻을 수 있다는 결론을 내었다.

**숙제문제 2-3 – program2\_2\_a( )**

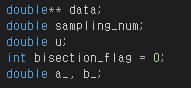
실습 2-2에서 구현하였던 난수 생성 프로그램의 가독성과 범용성을 개편하였다.

**실습 2-2에서의 함수**





**숙제 2-3-a에서의 함수**

f1에서 필요한 u, data, sampling\_num을 넘겨주기 위해 bisection을 포함해 f1의 함수의 parameter가 불필요하게 길어지는 경향이 있었다. 이러한 부분을 전역변수로 해결하였다.

**integral 함수**

입력한 범위의 부분적분을 구하는 integral함수는 program2\_1, 2\_2, 2\_3 모두에서 사용하였는데, 2-2에서는 따로 구현하였던 integral 함수를, 2\_2\_a에서는 2\_1에서 구현했던 integral 함수를 그대로 사용하였다. 이를 통해 불필요하게 반복되는 부분을 줄여주었다.

  
random number : 500

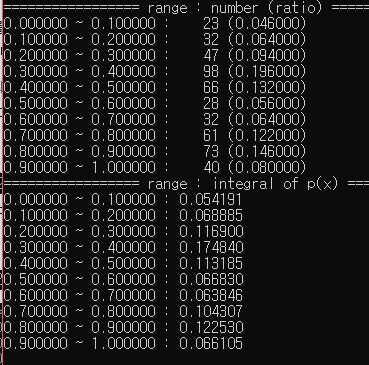
  
random number : 2000

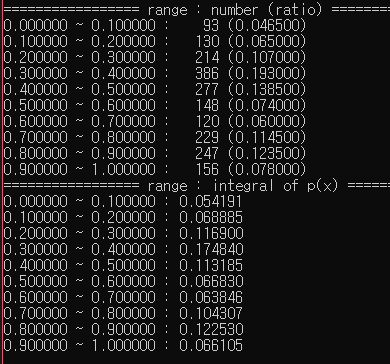
  
random number : 10000

**숙제문제 2-3 – program2\_2\_b**

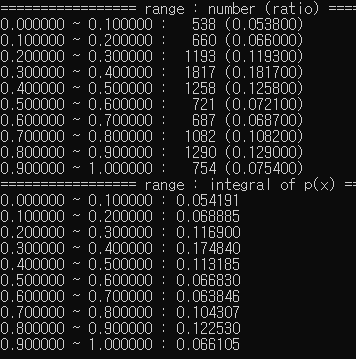
program2\_2( )를 바탕으로하여, Bisection method가 아닌 newton method혹은 secant method를 통해 근을 찾는다.

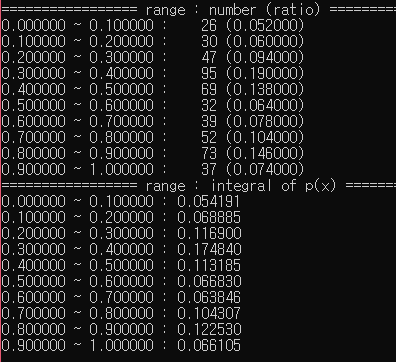
이 때, newton과 secant method는 초기값을 정해주어야 했는데, 이 때의 초기값을 bisection 과정을 n번 수행하고 나온 mid값과 a, b값으로 정해주었다.

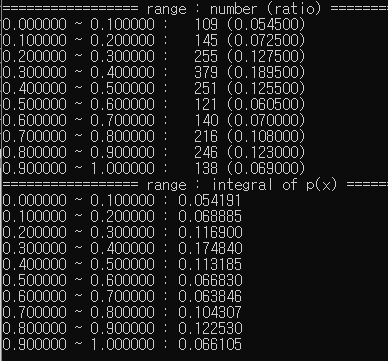
  
  
Newton Method, random number : 500

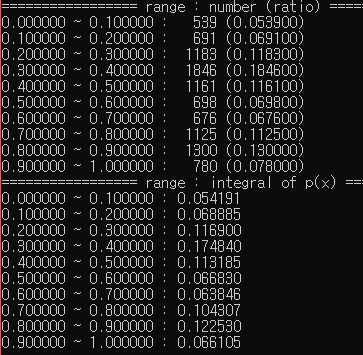
  


Newton Method, random number : 2000

  
  
Newton Method, random number : 10000

  
Secant Method, random number : 500

  
  
Secant Method, random number : 2000

Secant Method, random number : 10000

우선 오차에 대해 이야기해보자면, 대부분의 결과가 p(x)의 부분적분 결과에 근사하는 것을 확인할 수 있고, 실습 2\_2에서와 마찬가지로 난수의 수가 크면 클수록 오차는 적어지는 것을 확인할 수 있다.

다음은 속도를 확인해 볼 수 있다. 난수의 수에 따라 조금씩 다르지만, 대체로 bisection보다는 newton과 secant가 좀 더 빠른 것을 확인할 수 있다. newton과 secant중에서는 newton이 실행에 있어 더 빠른 속도를 보여주고 있다.

이를 통해 근을 찾는 알고리즘의 수행속도는 newton이 가장 빠르고, 그 다음으로 secant, 그 다음으로 bisection이라는 결론을 내릴 수 있다.

또한 수치로 확인하기는 어려우나, Secant method와 Newton method의 방식을 생각해보았을 때, secant method가 좀 더 정확할 것이라는 추측을 할 수 있다.

**실습 문제 2-3**

난수들을 일정 범위로 나누어 해당 범위에 존재하는 난수들의 개수를 세고, 이를 히스토그램으로 출력해주는 함수를 구현하였다.

이 함수는 다른 함수들을 수행할 때, 구현한 함수가 주어진 확률 밀도 함수를 따르는 지에 대한 판단을 할 수 있는 함수이다.

히스토그램은 해당 범위의 난수의 개수와 함께, 전체 난수의 개수에 대한 비율을 같이 출력해주었다.

생성한 히스토그램과 비교할 수 있는 수단으로, 히스토그램을 나눈 범위에 따라 p(x) 함수를 부분적분하고 그 결과를 출력해주었다. 이러한 부분적분의 출력결과는 앞서 말한 전체 난수의 개수에 대한 비율과 직접적으로 비교할 수 있는 수치이며, 이 수치가 비슷할수록 이 프로그램의 정확도가 높다는 것을 나타낸다.