
Homework 6 Report

You Wang

04/12/2021

Exercise 6.1

In `RandomMain3.cpp`, `GenTwo` use `RandomParkMiller` as inner generator. So to compare convergence of Monte Carlo simulations with and without anti-thetic sampling, we just need to use `generator` and `GenTwo` as generator in `SimpleMonteCarlo6` function and compare their results:

- For anti-thetic sampling:

```
SimpleMonteCarlo6(theOption,  
                  Spot,  
                  VolParam,  
                  rParam,  
                  NumberOfPaths,  
                  gathererTwo,  
                  GenTwo);
```

- For without anti-thetic sampling:

```
SimpleMonteCarlo6(theOption,  
                  Spot,  
                  VolParam,  
                  rParam,  
                  NumberOfPaths,  
                  gathererTwo,  
                  generator);
```

Results:

Let expiry=1, Strike=50, spot=50, vol=0.2, r=0.1, # paths=500000:

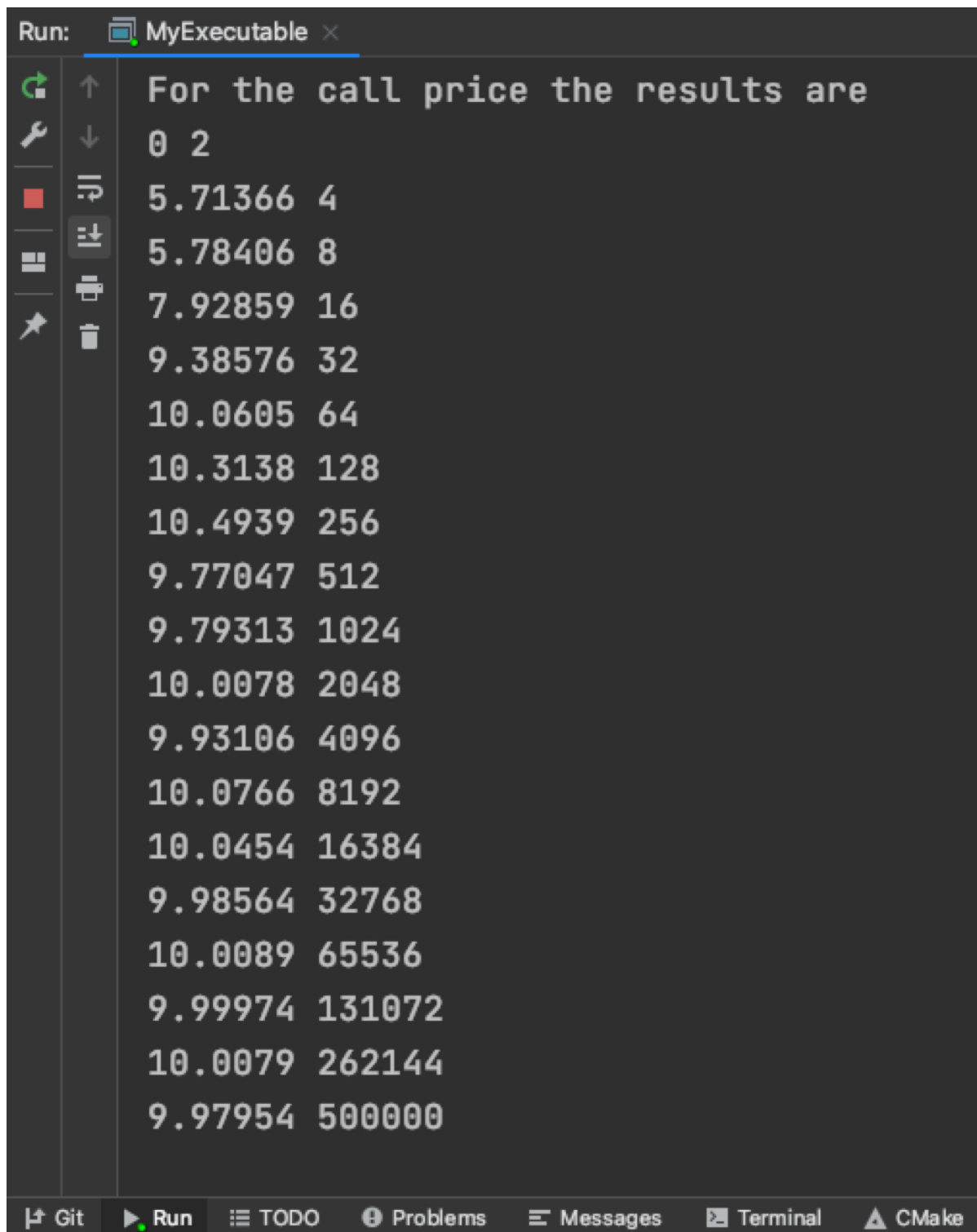
```
Run: MyExecutable x
For the call price the results are
0 2
3.45157 4
3.36582 8
5.0229 16
6.09496 32
6.62233 64
6.93836 128
7.05238 256
6.42126 512
6.39137 1024
6.61295 2048
6.5708 4096
6.71452 8192
6.69303 16384
6.62492 32768
6.64396 65536
6.63806 131072
6.64518 262144
6.62235 500000
|
```

Figure 1: Without Anti-thetic

```
Run: MyExecutable x
For the call price the results are
35.5122 2
21.7717 4
13.2112 8
9.8145 16
8.4761 32
7.54843 64
7.01268 128
7.05785 256
6.82617 512
6.62018 1024
6.49969 2048
6.5817 4096
6.62259 8192
6.67419 16384
6.66871 32768
6.62223 65536
6.63037 131072
6.62961 262144
6.63271 500000
```

Figure 2: With Anti-thetic

Let expiry=1, Strike=45, spot=50, vol=0.2, r=0.1, # paths=500000:



```
Run: MyExecutable x
For the call price the results are
0 2
5.71366 4
5.78406 8
7.92859 16
9.38576 32
10.0605 64
10.3138 128
10.4939 256
9.77047 512
9.79313 1024
10.0078 2048
9.93106 4096
10.0766 8192
10.0454 16384
9.98564 32768
10.0089 65536
9.99974 131072
10.0079 262144
9.97954 500000
```

Figure 3: Without Anti-thetic

The screenshot shows a terminal window titled "Run: MyExecutable" with a dark background. The output text is as follows:

```
For the call price the results are
37.7743 2
24.0338 4
16.2833 8
13.0889 16
11.7914 32
10.965 64
10.4449 128
10.4098 256
10.2041 512
10.028 1024
9.9215 2048
9.97217 4096
9.99418 8192
10.0318 16384
10.0206 32768
9.98709 65536
9.9932 131072
9.99048 262144
9.9926 500000
```

The IDE interface includes a left sidebar with "Favorites" and "Structure" views, and a bottom toolbar with icons for Git, Run, TODO, Problems, Messages, Terminal, and CMake.

Figure 4: With Anti-thetic

We can find that simulations with anti-thetic sampling converges relatively faster.

Exercise 6.2

Using boost library and fit it into the RandomBase class:

Random3.h:

```
#ifndef CLIONPROJECT_RANDOM3_H
#define CLIONPROJECT_RANDOM3_H

#include "Random2.h"
#include "Wrapper.h"

class BoostRandom:public RandomBase{
public:
    BoostRandom(unsigned long Dimensionality, unsigned long Seed=1);
    virtual RandomBase* clone() const;
    virtual void GetUniforms(MJArray& variates);
    virtual void Skip(unsigned long numberOfPaths);
    virtual void SetSeed(unsigned long Seed);
    virtual void Reset();

private:
    unsigned long Seed;
};
#endif //CLIONPROJECT_RANDOM3_H
```

Random3.cpp:

```
#include "Random3.h"
#include "boost/random.hpp"
#include "boost/random/random_device.hpp"

BoostRandom::BoostRandom(unsigned long Dimensionality, unsigned long Seed_):
RandomBase(Dimensionality), Seed(Seed_){}

RandomBase* BoostRandom::clone() const {
    return new BoostRandom(*this);
}

void BoostRandom::GetUniforms(MJArray &variates) {
    boost :: random_device dev;
    boost :: mt19937 rng(dev);
```

```

    boost :: uniform_01<> std;
    for (unsigned long j=0; j < GetDimensionality(); j++){
        variates[j] = std(rng);
    }
}

void BoostRandom::Skip(unsigned long numberOfPaths)
{
    MJArray tmp(GetDimensionality());
    for (unsigned long j=0; j < numberOfPaths; j++)
        GetUniforms(tmp);
}

void BoostRandom::SetSeed(unsigned long Seed_) {
    Seed = Seed_;
}

void BoostRandom::Reset() {
    Seed = 1;
}

```

In **RandomMain3.cpp**, we add BoostRandom generator:

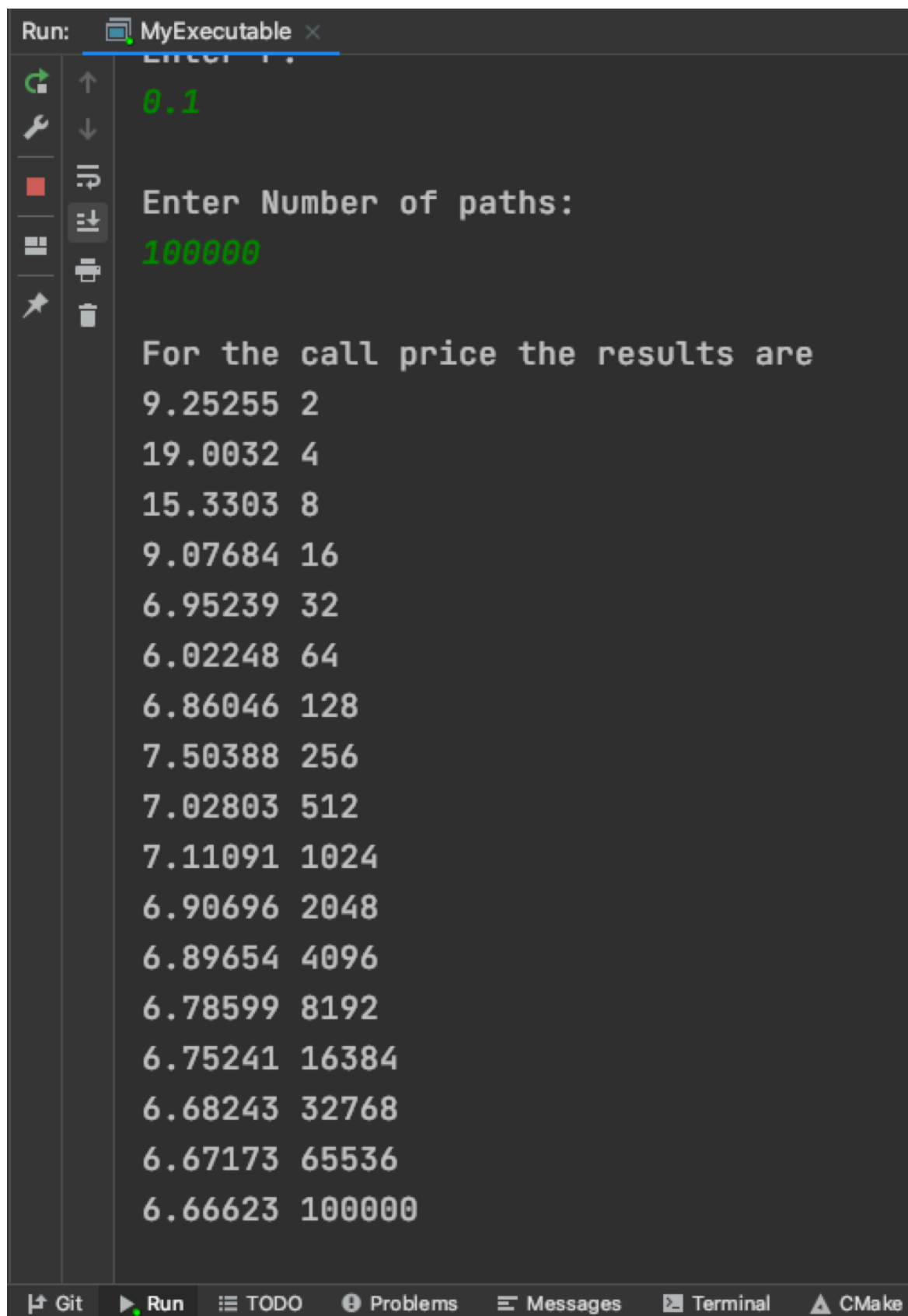
```

#include "Random3.h"
...
BoostRandom generator2(1);
// for call option
SimpleMonteCarlo6(theOption,
                  Spot,
                  VolParam,
                  rParam,
                  NumberOfPaths,
                  gathererTwo,
                  generator2);
...

```

Result:

Let expiry=1, Strike=50, spot=50, vol=0.2, r=0.1, # paths=100000:



```
Run: MyExecutable x
0.1
Enter Number of paths:
100000

For the call price the results are
9.25255 2
19.0032 4
15.3303 8
9.07684 16
6.95239 32
6.02248 64
6.86046 128
7.50388 256
7.02803 512
7.11091 1024
6.90696 2048
6.89654 4096
6.78599 8192
6.75241 16384
6.68243 32768
6.67173 65536
6.66623 100000
```

The screenshot shows a terminal window with a dark background. The title bar at the top says "Run: MyExecutable x". On the left, there is a vertical toolbar with icons for running, debugging, and other actions. The main area of the terminal displays the output of a program. It starts with a green "0.1", followed by a prompt "Enter Number of paths:" and a green "100000". Then, it says "For the call price the results are" and lists 16 rows of data, each with a decimal value and an integer value. The bottom of the window has a status bar with icons for Git, Run, TODO, Problems, Messages, Terminal, and CMake.

Value	Count
9.25255	2
19.0032	4
15.3303	8
9.07684	16
6.95239	32
6.02248	64
6.86046	128
7.50388	256
7.02803	512
7.11091	1024
6.90696	2048
6.89654	4096
6.78599	8192
6.75241	16384
6.68243	32768
6.67173	65536
6.66623	100000

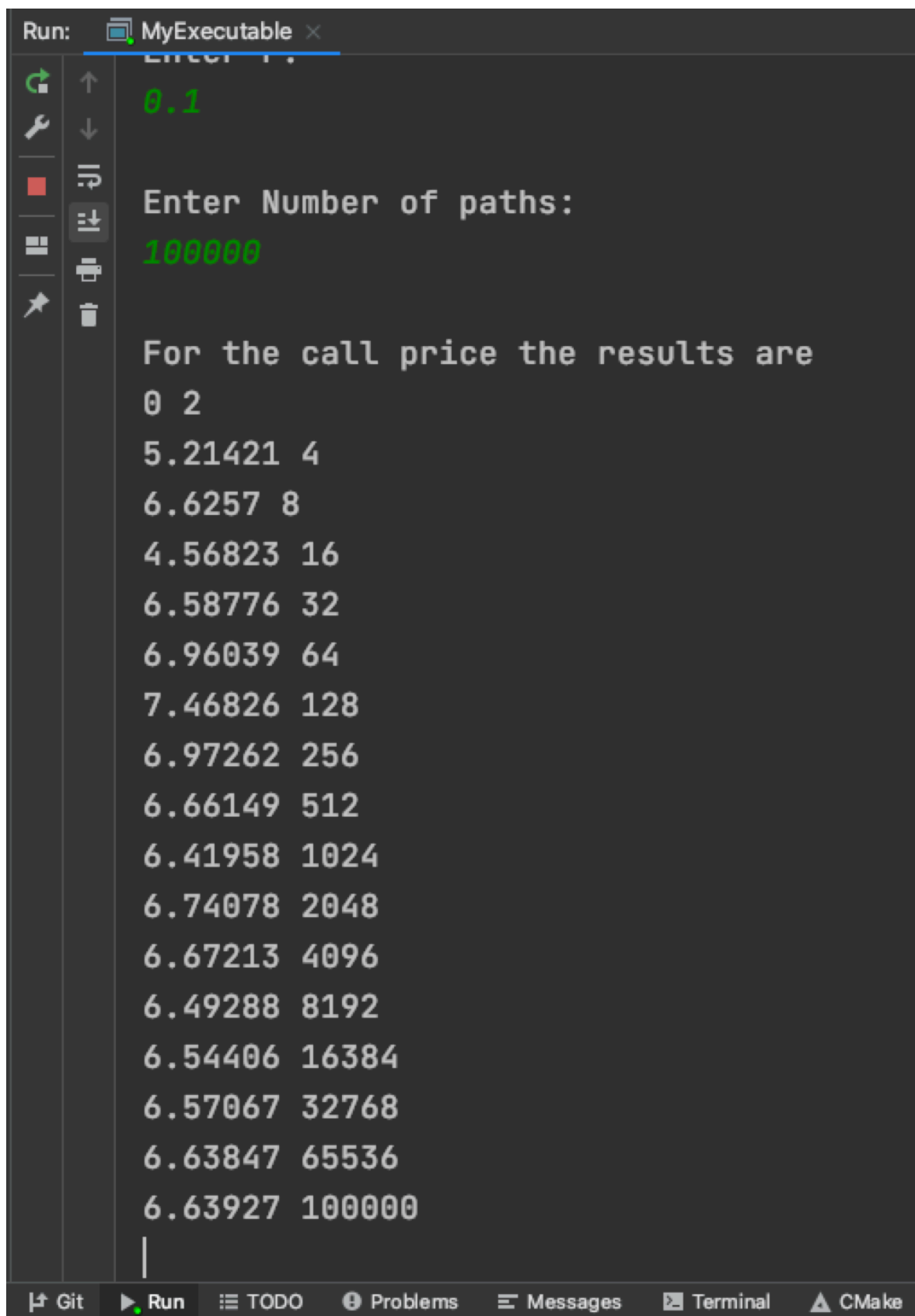
Figure 5: exercise 6.2

Note that here I use `uniform_01` in boost library to generate uniform numbers and still use `GetGaussians` in `RandomBase` to generate normal-distributed numbers from uniform-distributed numbers. We can also use `normal_distribution` to generate normal-distributed numbers directly, by overriding `GetGaussians` and implement it in `Random3.cpp`:

```
void BoostRandom::GetGaussians(MJArray &variates) {
    boost :: random_device dev;
    boost :: mt19937 rng(dev);
    boost :: normal_distribution<> std(0,1);
    for (unsigned long j=0; j < GetDimensionality(); j++){
        variates[j] = std(rng);
    }
}
```

Result:

Let expiry=1, Strike=50, spot=50, vol=0.2, r=0.1, # paths=100000:



```
Run: MyExecutable x
Enter Number of paths:
100000

For the call price the results are
0 2
5.21421 4
6.6257 8
4.56823 16
6.58776 32
6.96039 64
7.46826 128
6.97262 256
6.66149 512
6.41958 1024
6.74078 2048
6.67213 4096
6.49288 8192
6.54406 16384
6.57067 32768
6.63847 65536
6.63927 100000
|
```

The screenshot shows a CMake IDE interface. The top bar indicates the current configuration is 'Run: MyExecutable'. The terminal window displays the output of the program. It starts with a prompt 'Enter Number of paths:' followed by the user input '100000'. Then, it prints 'For the call price the results are' and a table of values. The table has two columns: a numerical value and a corresponding count. The counts are powers of 2, starting from 2 and going up to 100,000. The numerical values represent call prices, which fluctuate between approximately 4.5 and 7.5. The bottom of the IDE shows tabs for 'Git', 'Run', 'TODO', 'Problems', 'Messages', 'Terminal', and 'CMake'.

Figure 6: Exercise 6.2-2