Introduction to Modern C++

Homework 2 : PRNGs, Monte-Carlo Integration, and Markov Chains

Question 1. *Not a good idea*: Such numbers are not really random, they are predictable and the seed can in fact be recovered in some situations. If you need a "random" password or key, you should always use a cryptographically secure *true* random number generator (TRNG). Such generators always rely on special-purpose hardware.

Question 2. There are many ways you can generate Gaussian samples. To name just one, inverse sampling: Let F be the cumulative distribution function of our Gaussian distribution, and X a random variable distributed uniformly in $(0, 1)]$, then $Y = F^{-1}(X)$ is distributed according to the Gaussian distribution. Of course you can also use the code provided in the appendix instead.

Question 3. We have the following:

$$\begin{aligned}\int_{y=0}^{1}\int_{x=0}^{1} 1_C \,\mathrm{d}x\,\mathrm{d}y &= \int_{y=0}^{1}\int_{x=0}^{\sqrt{1-y^2}} 1 \,\mathrm{d}x\,\mathrm{d}y \\ &= \int_{y=0}^{1} \sqrt{1-y^2}\,\mathrm{d}y\end{aligned}$$

We can compute this exactly (or make a drawing) to find $\pi/4$. It is now simple to integrate using the Riemann method we saw during one lab session:

```
#include <iostream>
#include <cmath>

int main() {
  int N = 1000;
  double result = 0;
  for (int i = 0; i < N; ++i) {
    double y = i * 1.0/N;
    result = result + sqrt(1 - y*y);
  }
  result = result / N;

  std::cout << "Result:   " << result << std::endl;
  std::cout << "Expected: " << M_PI/4 << std::endl;
  std::cout << "Error:    " << result - M_PI/4 << std::endl;
}
```

This program finds an error of `0.000490703`, which is reasonably low. Note that here we only perform 1000 sums, but this is thanks to using Fubini and getting rid of one of the two integral signs. Otherwise, we would have 1000×1000 sums.

Question 4. We use the Hint 1 to make a `getUnif()` function: This will prove useful in the rest of the homework. Then simply implement the Monte-Carlo algorithm:

```
#include <iostream>  // For std::cout
#include <random>    // For random numbers
#include <cmath>     // For PI

int seed = 314159;
std::mt19937 gen (seed);
std::uniform_real_distribution<> unif (0, 1);

double getUnif() {
  // Returns a random number, distributed uniformly between 0 and 1
  return unif(gen);
```

```
}

double oneC(double x, double y) {
  if (x*x + y*y < 1) {
    return 1;
  }
  return 0;
}

int main() {
  // Monte-Carlo algorithm
  double S = 0;
  int N = 1000;
  for (int i = 0; i < N; ++i) {
    double x = getUnif();
    double y = getUnif();
    S = S + oneC(x, y);
  }
  S = S/N;

  std::cout << "Result:   " << S << std::endl;
  std::cout << "Expected: " << M_PI/4 << std::endl;
  std::cout << "Error:    " << S - M_PI/4 << std::endl;
}
```

This program finds an error of `0.000601837` (slightly more than the Riemann method). Here again, we only perform 1000 sums, but this is true no matter how many integrals there are.

Question 5. This question illustrate the above remark. Note that in the previous code we compute a quarter of the circle. Accordingly, we only compute 1/8 of the 3D sphere (make a drawing to see why). Here is the solution for the 3-dimensional sphere.

```
#include <iostream>  // For std::cout
#include <random>    // For random numbers
#include <cmath>     // For PI

int seed = 314159;
std::mt19937 gen (seed);
std::uniform_real_distribution<> unif (0, 1);

double getUnif() {
  // Returns a random number, distributed uniformly between 0 and 1
  return unif(gen);
}

double oneC(double x, double y, double z) {
  if (x*x + y*y + z*z < 1) {
    return 1;
  }
  return 0;
}

double SphereMC() {
  // Monte-Carlo algorithm
  double S = 0;
  int N = 1000;
  for (int i = 0; i < N; ++i) {
    double x = getUnif();
    double y = getUnif();
    double z = getUnif();
    S = S + oneC(x, y, z);
  }
  return S/N;
}

int main() {
```

```
  double S = SphereMC() * 8; // We only compute 1/8 of the 3D sphere!

  std::cout << "Result:   " << S << std::endl;
  std::cout << "Expected: " << 4*M_PI/3 << std::endl;
  std::cout << "Error:    " << S - 4*M_PI/3 << std::endl;
}
```

This program finds an error of `0.0592098`. For the 10-dimensional sphere (of which we only compute 1/1024) the error gets larger, namely `-0.502164`. Using larger values of N enables us to get smaller errors (for instance, with $N = 1000$ I get an error of `-0.06`).

Question 6. Note: Every time the program is run, the *same* pseudo-random numbers will be generated. It's good practice to keep the seed fixed when debugging, but to change it when performing actual computations (once we know that everything works fine).

```
#include <iostream>  // For std::cout
#include <random>    // For random numbers
#include <cmath>     // For PI

int seed = 314159;
std::mt19937 gen (seed);
std::uniform_real_distribution<> unif (0, 1);

double getUnif() {
  // Returns a random number, distributed uniformly between 0 and 1
  return unif(gen);
}

double oneC(double x, double y, double z) {
  if (x*x + y*y + z*z < 1) {
    return 1;
  }
  return 0;
}

double SphereMC() {
  // Monte-Carlo algorithm
  double S = 0;
  int N = 1000;
  for (int i = 0; i < N; ++i) {
    double x = getUnif();
    double y = getUnif();
    double z = getUnif();
    S = S + oneC(x, y, z);
  }
  return S/N;
}

int main() {
  double variance = 0;

  int M = 1000;
  for (int i = 0; i < M; ++i) {
    double S = SphereMC() * 8; // We only compute 1/8 of the 3D sphere!

    variance = variance + pow(S - 4*M_PI/3, 2);
  }
  variance = variance/(M-1);

  std::cout << "Variance : " << variance << std::endl;
}
```

This program finds a variance of `0.0144904`. Note that increasing N makes the variance go down, as the following table (generated using the program above) shows:

N	10	100	1000	10000
Var	1.57326	0.158464	0.0144904	0.00154892

It is actually a theorem that Monte-Carlo methods halve their variance when we double their running time.

Question 7. The portfolio evaluation works exactly like the sphere volume evaluation, with slightly more complicated random variables (namely, a geometric brownian motion instead of uniform variables).

```
#include <random>
#include <iostream>

int seed = 314159;
std::mt19937 gen(seed);
std::normal_distribution<>       norm (0,1);

double SA0    = 100; double sigmaA = 0.2;
double SB0    =  75; double sigmaB = 0.18;
double alpha  = 100; double muA    = 0.15;
double beta   = 100; double muB    = 0.12;
double T = 0.5;

double getNorm() {
  return norm(gen); // Normally distributed random variable
}

double IL(double SA, double SB) {
  double W0 = alpha * SA0 + beta * SB0;
  double WT = alpha * SA  + beta * SB ;

  if (WT/W0 <= 0.9) {
    return 1;
  }
  return 0;
}

double GeomBrownMot(double mu, double sigma, double T) {
  double B = getNorm() / sqrt(T);

  return exp((mu - sigma*sigma/2) * T + sigma * B);
}

int main() {
  if (alpha*SA0+beta*SB0 == 0) {
    std::cerr << "Error: Incorrect parameters!" << std::endl;
    return -1;
  }
  int N = 1000;
  double theta = 0;
  for (int i = 0; i < N; ++i) {
    double SAT = SA0 * GeomBrownMot(muA, sigmaA, T);
    double SBT = SB0 * GeomBrownMot(muB, sigmaB, T);

    theta = theta + IL(SAT, SBT);
  }
  theta = theta/N;

  std::cout << "Theta = " << theta << std::endl;
}
```

This program computes a final probability $\theta = 0.181$, i.e. there is a 18% probability that the portfolio loses at least 10% of its worth over 6 months. It is easy to modify this program to get more information (e.g. the probability of at least 10% *gain* with that portfolio over the same period, which is 44%).

You can also run the simulation M times to compute the variance, hence the confidence interval of such predictions.

Question 9. (Yes there was no question 8, my mistake). There are again several ways to solve this question — but at any rate it requires the painstaking entry of the transition matrix. I chose the following approach, which makes the matrix clearly visible for inspection and uses `std::array`.

```
#include <random>   // Random numbers
#include <array>    // Arrays
#include <iostream> // std::cout

// Transition matrix

std::array<double,8> AAA {0.9366, 0.0583, 0.004,  0.0009, 0.0002, 0     , 0,      0     };
std::array<double,8> AA  {0.0066, 0.9172, 0.0694, 0.0049, 0.0006, 0.0009, 0.0002, 0.0002};
std::array<double,8> A   {0.0007, 0.0225, 0.9176, 0.0518, 0.0049, 0.002 , 0.0001, 0.0004};
std::array<double,8> BBB {0.0003, 0.0026, 0.0483, 0.8924, 0.0444, 0.0081, 0.0016, 0.0023};
std::array<double,8> BB  {0.0003, 0.0006, 0.0044, 0.0666, 0.8323, 0.0746, 0.0105, 0.0107};
std::array<double,8> B   {0,      0.001,  0.0032, 0.0046, 0.0572, 0.8362, 0.0384, 0.0594};
std::array<double,8> CCC {0.0015, 0,      0.0029, 0.0088, 0.0191, 0.1028, 0.6123, 0.2526};
std::array<double,8> def {0,      0,      0,      0,      0,      0,      0,      1     };

std::array<std::array<double,8>,8> matrix {AAA,AA,A,BBB,BB,B,CCC,def};

// Pseudo-random numbers

int seed = 314159;
std::mt19937 gen (seed);
std::uniform_real_distribution<> unif (0, 1);

double getUnif() {
  return unif(gen);
}

// Credit evolution

int EvolveRatingOneYear(int currentRating) {
  double r = getUnif();
  double cumulatedProb = 0;

  for (int i = 0; i < 8; ++i) {
    cumulatedProb = cumulatedProb + matrix[currentRating][i];

    if (r <= cumulatedProb) {
      return i;
    }
  }
}

int EvolveRating10Years(int currentRating) {
  for (int year = 0; year < 10; ++year) {
    currentRating = EvolveRatingOneYear(currentRating);
  }
  return currentRating;
}

// Main function: Several simulations counting when we get downgraded to CCC or worse

int main() {
  // Note: 0: AAA, 1: AA, ..., 6: CCC and 7: def.

  int M = 1000000;
  double S = 0;
  for (int i = 0; i < M; ++i) {
    if (EvolveRating10Years(0) >= 6) {
      S = S + 1;
    }
  }
  S = S/M;
  std::cout << "Probability of AAA -> CCC or default in 10 yrs = " << S << std::endl;
}
```

This program computes a probability of `0.002376`, i.e. a little bit more than 0.2%. We're safe! You can easily compute the probabilities for other credit ratings to go up or down, by modifying this program.