Report of Lab 5: Monte Carlo Integration

Alberto Nieto Sandino

October 26, 2018

1 Introduction

In this lab, we implement the Monte Carlo integration and different methods of variance reduction to improve the estimate. We will also observe how the estimate behaves as the number of simulations N increases.

2 Assignment 1

2.1 Ordinary Monte Carlo integration

Firstly, we are gonna look at the results that performing the ordinary Monte Carlo integration gives for the integral

$$\pi = \int_0^1 \frac{4}{1+x^2} dx. \tag{1}$$

We can observe the results for the different sample sizes in Table 1.

Table 1: Results of the ordinary Monte Carlo integration for equation (1)

N	Estimate S_n	Error estimate	Actual error
10^5	3.142200	± 0.002032	-0.000607
10^6	3.141197	± 0.000643	0.000396
10^7	3.141794	± 0.000203	-0.000201
10^8	3.141428	± 0.000064	0.000165

2.2 Monte Carlo integration with variance reduction techniques

Now, we will observe how the different *variance reduction* techniques improve the approximation of the Monte Carlo integration (Table 2). The idea is transforming the original observations in a way that conserves the expected value and reduces the variance of the estimand. In Table 3 and Table 4, we can see the expected error and the actual error of the estimates for the different methods used to approximate the integral.

Table 2: Results of the Monte Carlo estimate of the integral implementing variance reduction techniques

N	Estimate S_n				
	Standard method	Importance sampling	Control variates	Antithetic variates	Stratified sampling
10^5	3.142200	3.141726	3.141732	3.141733	3.141662
10^6	3.141197	3.141587	3.141634	3.141579	3.141753
10^7	3.141794	3.141617	3.141628	3.141603	3.141644
10^8	3.141428	3.141604	3.141577	3.141595	3.141596

Table 3: Expected and actual errors for the Monte Carlo estimate of the integral implementing variance reduction techniques (Part 1)

	Error					
	Standard method		Importance sampling		Control variates	
N	Expected	Real	Expected	Real	Expected	Real
10^5	± 0.002032	-0.000607	± 0.000253	-0.000133	± 0.000298	-0.000139
10^6	± 0.000643	0.000396	± 0.000080	0.000005	± 0.000094	-0.000041
10^7	± 0.000203	-0.000201	± 0.000025	-0.000024	± 0.000030	-0.000035
10^8	± 0.000064	0.000165	± 0.000008	-0.000011	± 0.000009	0.000016

Table 4: Expected and actual errors for the Monte Carlo estimate of the integral implementing variance reduction techniques (Part 2)

	Error			
	Antithetic variates		Stratified sampling	
N	Expected	Real	Expected	Real
10^5	± 0.000182	-0.000140	± 0.000481	-0.000069
10^6	± 0.000058	0.000014	± 0.000152	-0.000160
10^7	± 0.000018	-0.000010	± 0.000048	-0.000052
10^8	± 0.000006	-0.000002	± 0.000015	-0.000003

As it would be expected, we can observe that the more simulations N that are done, the more accurate the estimate is (Table 2). We can also see that the errors (both the expected and the actual errors) decrease as the number of simulations increases. Furthermore, in Table 3 and Table 4, we see that the expected error and the actual error are not the same but they are a good estimation of the order of magnitude of the error for all of the variance reduction techniques and the ordinary Monte Carlo integration. When looking at the final results, we can see that the antithetic variates method has produced the best results for decreasing the variance and the real error, as well.

For the *importance sampling* method, the variance is reduced because we choose a distribution function in which the density of the samples is similar to that of the shape of the integrand (i.e. the function inside the integral), making the variance of the sample mean decrease. This chosen distribution, that replaces the uniform distribution used in ordinary Monte Carlo integration, overweights the *important* region of the integral (that's where it gets its name). This method will be more effective, the closest that our chosen probability density function (PDF) p is to the distribution of our integrand function f, since our error σ is approximated from

$$\hat{\sigma}^2_{S_n}\left(\frac{f(X)}{p(X)}\right) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{f(x_i}{p(x_i)}\right)^2 - S_n^2. \tag{2}$$

If the shapes of the PDF of p and f are similar, the ratio f/p will be closer to a constant, thus minimizing the variance, making the method more effective.

For the *control variates* method, we use a control variate function g that is similar to f, but where the value of its integral is known $I(g) = \int h(x)$. Applying it to the integral by linearity and approximating it by the Monte Carlo integration, we obtain

$$S_n = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - g(x_i)) + I(g). \tag{3}$$

In this equation, we can see that the variance comes from $f - g$ and not from f. This variance should be smaller since $f - g$ is almost constant when compared with f. The method will be more effective, the more similar g is with f, since this will minimize the summation term in equation (3), which is the one where the variation is originating from.

For the *antithetic variates* method, we choose pairs of observations which have negative correlation so that the total variance (i.e. the error) is reduced, since

$$\mathbf{Var}\{X+Y\} = \mathbf{Var}\{X\} + \mathbf{Var}\{Y\} + 2\mathbf{Cov}\{X+Y\}. \tag{4}$$

Thus, the total variance become smaller when the covariance term is negative. This gives a total variance of

$$\hat{\sigma}^2_{S_n} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{f(x_i)}{2} + \frac{f(1-x_i)}{2}\right)^2 - S_n^2, \tag{5}$$

which gives a lower variance that the ordinary Monte Carlo integration. This method works best when the function to integrate is monotonically increasing or decreasing, otherwise, it is not guaranteed that the covariance of the two samples will be negative and the method could produce worst variance than the standard method.

For the *stratified sampling* method, we divide the domain of integration in a way that the integrand f is evaluated more often in the regions where it varies most. The corresponding error for this method is

$$\Delta_{SS} = \sqrt{\sum_{j=1}^{k}\frac{\mathrm{vol}(M_j)^2}{n_j}\sigma^2_{M_j}(f)}. \tag{6}$$

The method performs more effectively if the regions are divided in a way that we compute the integral more times where more variation exist, that is

$$n_j \sim \mathrm{vol}(M_j)\sigma_{M_j}(f). \tag{7}$$

This is because the more regions, the better the change in volume will be catched, thus the shape of the approximation will be more similar to the shape of the integral.

Appendix

Detailed tables for variance reduction techniques

Table 5: Results of the Monte Carlo integration implementing importance sampling

N	Estimate S_n	Error estimate	Actual error
10^5	3.141726	± 0.000253	-0.000133
10^6	3.141587	± 0.000080	0.000005
10^7	3.141617	± 0.000025	-0.000024
10^8	3.141604	± 0.000008	-0.000011

Table 6: Results of the Monte Carlo integration implementing control variates

N	Estimate S_n	Error estimate	Actual error
10^5	3.141732	± 0.000298	-0.000139
10^6	3.141634	± 0.000094	-0.000041
10^7	3.141628	± 0.000030	-0.000035
10^8	3.141577	± 0.000009	0.000016

Table 7: Results of the Monte Carlo integration implementing antithetic variates

N	Estimate S_n	Error estimate	Actual error
10^5	3.141733	± 0.000182	-0.000140
10^6	3.141579	± 0.000058	0.000014
10^7	3.141603	± 0.000018	-0.000010
10^8	3.141595	± 0.000006	-0.000002

Table 8: Results of the Monte Carlo integration implementing stratified sampling

N	Estimate S_n	Error estimate	Actual error
10^5	3.141662	± 0.000481	-0.000069
10^6	3.141753	± 0.000152	-0.000160
10^7	3.141644	± 0.000048	-0.000052
10^8	3.141596	± 0.000015	-0.000003

C code

```
/* Lab 05 - Monte Carlo integration
By Alberto Nieto
2018-10-25
*/
#include <math.h> //for mathfunctions
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // for timing


double ordinary_montecarlo(double n)
    {
      //seed rand function
        srand(time(NULL));
        // Define parameters to be used in the code
        double x=0, y=0, S_n=0, errorterm=0, error=0;
      // loop according to sample size n
      for (int i = 0; i < n; i++) {
          x = rand() / (RAND_MAX + 1.0); //samples
          y = 4/(1+x*x); // approximation

          S_n= S_n+y; // uppdate for each loop according to sample size
          errorterm= y*y + errorterm;
      }
      S_n= S_n/n; // Divide by sample size to get the approximated value
      error=sqrt(errorterm/n-(S_n*S_n))/sqrt(n); // approximated error
    printf("Ordinary Monte Carlo Integration:\n");
    printf("S_n= %f, Estimated error = +/- %f, Real error = %f\n", S_n, error, (M_PI-S_n));
    }


double importance_sampling(double n)
  {
    //seed rand function
   srand48(time(NULL)); //Make a new seed for the random number generator
    double x=0, y=0, S_n=0, errorterm=0, error=0, P=0, z=0;
    // loop according to sample size n
    for (int i = 0; i < n; i++) {

        z = drand48(); // randomize
        x = 2-sqrt(4-3*z); // define sample
        P = (4 - 2*x)/3; // define the p(x) function
        y = (4/(1+x*x)); // define the y function

        S_n += y/P; // do opertion and update according to sample size
        errorterm += ((y/P)*(y/P));
    }
    S_n = S_n/n;
    error = sqrt(errorterm/n - S_n*S_n)/sqrt(n);
  printf("Importance Sampling:\n");
  printf("S_n= %f, Estimated error +/- %f, Real error =%f\n", S_n, error, (M_PI-S_n));
  }
```

```
double control_variates(double n)
    {
      //seed rand function
     srand48(time(NULL)); //Make a new seed for the random number generator
         double x=0, y=0, S_n=0, errorterm=0, error=0, g=0, I=0;
      // loop according to sample size n
      for (int i = 0; i < n; i++) {

          x = drand48();
          //x = 2-sqrt(4-3*z);
          g = (4 - 2*x);
          y = (4/(1+x*x));
          I = 3.0;

          S_n += y-g+I;
          errorterm += ((y-g+I)*(y-g+I));
      }
      S_n = S_n/n;
      error = sqrt(errorterm/n - S_n*S_n)/sqrt(n);
    printf("Control Variates:\n");
    printf("S_n= %f, Estimated error +/- %f, Real error =%f\n", S_n, error, (M_PI-S_n));
    }


double antithetic_variates(double n)
    {
      //seed rand function
     srand48(time(NULL)); //Make a new seed for the random number generator
     double x=0, y=0, S_n=0, errorterm=0, error=0, g=0, y_comp=0;
      // loop according to sample size n
      for (int i = 0; i < n; i++) {

          x = drand48();
          //x = 2-sqrt(4-3*z);
          y = (4/(1+x*x));
          y_comp = (4/(1+(1-x)*(1-x)));

          S_n += (y + y_comp)/2;
          errorterm += (((y + y_comp)/2)*((y + y_comp)/2));
      }
      S_n = S_n/n;
      error = sqrt(errorterm/n - S_n*S_n)/sqrt(n);
    printf("Antithetic Variates:\n");
    printf("S_n= %f, Estimated error +/- %f, Real error =%f\n", S_n, error, (M_PI-S_n));
    }


      double stratified_sampling(double n)
          {
            //seed rand function
           srand48(time(NULL)); //Make a new seed for the random number generator
          int k = 4; // Number of domains
```

```
          double x=0, y=0, S_n=0, errorterm=0, error=0;
            // loop according to sample size n
              double M_j = (double)1/(double)4; // domain size
                    int nj = n/k; // domain sample size
                        // loop according to number of domains
                        for (int i = 1; i < k+1; i++)
                        {
                            double S_n1=0, S_n2=0, sigma=0;
                                for (int j = 0; j < nj; j++)
                                {

                            x = drand48()/(double)k + (double)(i-1)/(double)k;
                            // uniformly distributed
                            y = (4/(1+x*x));
                            S_n1 += y*y;
                            S_n2 += y;

                                }
                   S_n += M_j*(S_n2/nj);
                   S_n1 = S_n1/nj;
                   S_n2 = S_n2/nj;
                   sigma = (S_n1 - (S_n2*S_n2));
                   errorterm += (M_j*M_j)*(sigma/nj);
                    }

            error = sqrt(errorterm);
          printf("Stratified Sampling:\n");
          printf("S_n= %f, Estimated error +/- %f, Real error =%f\n", S_n, error, (M_PI-S_n));
          }


int main() {

      for (int exp = 5; exp <9; exp++)
      {
      double size = pow(10,exp);
      printf("\n\n");
      printf("####################################################\n");
      printf("#########     For n = 10^%d        #####################\n", exp);
        printf("######################################################\n");
    ordinary_montecarlo(size);
      importance_sampling(size);
      control_variates(size);
      antithetic_variates(size);
      stratified_sampling(size);
        }

return 0;
}
```