CO65: Monte Carlo Integration Lab Report

Alex McInerny, shil5991, 24th January 2023

1 Aims and Methods

In Part 1, I was tasked with determining the volume of a 10-D hyper-sphere using a Monte Carlo simulation. This is a relatively complicated analytical derivation (although the closed form for an n-ball is quite simple; $V_n(R) = \frac{\pi^{n/2}}{\Gamma(n/2+1)}R^n$. Since this derivation is quite tricky, it can be easier to approximate V_{10} using a Monte Carlo simulation (this is why Monte-Carlo is so advantageous). To do this, I shall generate a vector, $\vec{r}$, with ten elements (i.e., $\vec{r} = \sum_{i=1}^{10} r_i \hat{e}_i$) where each $r_i \in [-1, 1]$ is generated by a random uniform distribution. Then I'll check whether this vector is 'inside' the hyper sphere. This is easy to verify, since if it's inside, $\|\vec{r}\| = \sqrt{\sum_{i=1}^{10} r_i^2} \leq 1$. If the point is inside, just add 1 to a counter. Repeating this for a large number and calling the fraction of points inside to points outside R, The volume of the sphere can finally be approximated as this ratio multiplied by the volume of the bounding region (i.e., the volume of a ten dimensional hypercube with side length 2), which is 2^{10}. So concluding, $V_{10,\text{ approx}} = R \times 2^{10}$.

Part 2 required me to approximate the mass, centres of mass in all 3 dimensions, and moments of inertia (again in every dimension) of a section of a torus. In three dimensions, the mass of an object is defined as $M = \int_{\text{across body}} \rho(x, y, z)\ \mathrm{d}x\mathrm{d}y\mathrm{d}z$. Further, the centre of mass and moment of inertia are defined as follows; $x_{\text{cm}} = \int \rho\ \mathrm{d}x\mathrm{d}y\mathrm{d}z$ and $I_{xx} = \int (y^2 + z^2)\rho\ \mathrm{d}x\mathrm{d}y\mathrm{d}z$, where the y and z ones are found with x, y, and z following a cyclic rotation. Once more, these integrals will be quite hard to evaluate, so they can be decomposed into discrete summations. By calculating the following quantities ($\sum \rho$, $\sum \rho^2$, $\sum x\rho$, $\sum (x\rho)^2$, $\sum (y^2 + z^2)\rho$, $\sum ((y^2 + z^2)\rho)^2$ and the other 8 cyclic permutations) you're able to approximate the desired expressions using the equations (again along with their cyclic alternatives for y_{cm}, z_{cm}, I_{yy}, and I_{zz})

$$\begin{cases} M = V\frac{\sum\rho}{N} \pm V\sqrt{\frac{\frac{\sum\rho^2}{N} - \left(\frac{\sum\rho}{N}\right)^2}{N}}, \\ x_{\text{cm}} = \frac{V}{M}\frac{\sum x\rho}{N} \pm \frac{V}{M}\sqrt{\frac{\frac{\sum(x\rho)^2}{N} - \left(\frac{\sum x\rho}{N}\right)^2}{N}}, \\ I_{xx} = V\frac{\sum(y^2+z^2)\rho}{N} \pm V\sqrt{\frac{\frac{\sum((y^2+z^2)\rho)^2}{N} - \left(\frac{\sum(y^2+z^2)\rho}{N}\right)^2}{N}} \end{cases}$$

where V is the volume of the bounding box (in which the random points will be generated), and N is the number of random vectors of points iterated through.

The question asks you to determine the volume of the torus defined by the equation $z^2 + \left(\sqrt{x^2 + y^2} - 3\right)^2 \leq 1$ bounded by the planes $1 \leq x \leq 4$, $-3 \leq y \leq 4$, and $-1 \leq z \leq 1$. It also allows me to assume a constant density through the desired region of $\rho = 1$

Part 3 requires some strategic integration of the 2-dimensional function $f = \sin\log(x + y + 1)$ over the disk $\mathcal{D} = x^2 + y^2 \leq 0.25$! This integral can be approximated by the formula

$$i = \frac{\sum_{j=1}^{N} \sin\log(x + y + 1)\Delta S}{N}$$

where here ΔS is the area of the bounding surface, and N is the number of random position vectors generated within the bounding surface.

The Plan Of Action (PoA):

For all three parts, I'll need two functions; `get_accepted_points` (Appx.(A)), and `calculate_sum` (Appx.(B)). `get_accepted_points` will take a large $N \times D$ matrix (N many D-dimensional vectors each corresponding to a point in a D dimensional space) named `points` and iterate through the rows of that matrix. It will have an if-statement, and if the specified condition (a function handle passed to `get_accepted_points` called `condition_function`) is satisfied, it will take the vector on that iteration of the for-loop and add it to a new matrix named `acceptedPoints`, which by the end will be $M \times D$, where M is the number of accepted points. The `calculate_sum` function will simply take `acceptedPoints` as an input along with a function handle `func` and return a value equal to the that function applied to every point and summed over the length of `acceptedPoints`.

For Part one (Appx.(C)), I just need to create an array of N random 10 dimensional vectors between -1 and 1. The condition function will check that the norm of these vectors is less than (or equal to) 1. Then the ratio R mentioned earlier is just the number of rows of `acceptedPoints` divided by N.

Part two (Appx.(D)) should also be quite simple. I'll create `points` again, this time being an $N \times 3$ matrix composed of random numbers with the conditions specified earlier. I'll then apply `get_accepted_points` to it. I'll create an array of cells, each cell containing one function handle corresponding to one of the summations I need to calculate. I will then iterate through the 14 function handles and apply the `calculate_sum` function to accepted points for each cell in my function handle array, and place each value in it's corresponding position in a vector. Then all I need to do is calculate the values required by the question. I'll do the mass and it's error on it's own, then create a for-loop to calculate the centre of masses, moments of inertia and their respective errors.

The third Part (Appx.(E)) PoA is also pretty simple. Once more I'll create 2-D array, `points`, this time $N \times 2$. I'll also apply the `get_accepted_points` function (with the condition mentioned earlier), getting the `acceptedPoints` matrix. I'll then pass that to the `calculate_sum` function with the condition function f evaluated at some point in `acceptedPoints` point multiplied by ΔS and divided by N (therefore adding every contribution which contributes to the integral). Once that's done, `i` should be the answer!

2 Results

The code in Appx.(C) carries out Part 1. Over three tries with $N = 1000000$, it gives answers of 2.5447424, 2.548224, and 2.5742336. The true value is $\pi^5/120 \approx 2.55016404$. Clearly, mine are pretty close to this. Obviously due to the nature of random number generation they're not bang on, but using higher N and taking many measurements would provide an average which closely converges on the true value, and also allow you to calculate the standard error.

The code for Part 2 is included in Appx.(D). For one iteration at $N = 100000$, the results are shown in Fig.(1).

	M	x_{cm}	y_{cm}	z_{cm}	I_{xx}	I_{yy}	I_{zz}
value	22.06	2.410993	0.15821970	-0.00446287	81.208	144.431	225.444
± error	0.07	0.000007	0.00000008	0.00000008	0.002	0.003	0.008

Figure 1: Sample results for Part 2, $N = 100000$

Note how the z_{cm} coordinate is approximately zero, which is to be expected for the torus since it's symmetric about the $x - y$ plane. Also see, the percentage errors on the values are very small.

Once more, the code for Part 3 is included in Appx.(E). Taking $N = 100000$, it provides approximations to the integral of -0.047396204479612, -0.049946670085657, and -0.046148418113899 over 3 concurrent runs. Although the actual value isn't known, all of these values are constant with each other. When graphing the function in `wolfram mathematica`, it's clear that the integral should be negative at least. Again I could have adapted the code to run the algorithm multiple times, allowing for a mean and standard error to be calculated, but since the true value isn't known there's currently not much reason, since I can't see whether my value agrees with the true one within its errors.

3 Conclusion

All of my results seem reasonable and are consistent with each other over many runs of the code with relatively small variation, although for Parts 2 and 3, I can't know how accurate they are due to the lack of an explicit answer. I can confirm my first solution provided a correct answer (validating my accepted points function), and since the method for the other two is similar, hopefully they're also correct. Clearly due to the nature of random number generation, even with very large sample sizes my answer won't be bang on the true value.

The accuracy could be increased by increasing number of iterations (accuracy of Monte Carlo simulations grows with $1/\sqrt{N}$). Instead of only increasing N, I could adapt it by repeating the whole algorithm N_2 times (getting N_2 answers) and taking a mean of those, where I could also calculate standard error, $\alpha = \sigma/\sqrt{N_2}$.

Also, I believe the code used was somewhat inefficient due to having to store massive arrays in memory (giving a relatively low upper limit on N, and meaning computation time greatly increase for large N). It could be improved by creating a for loop with N iterations, and in the for loop creating one position vector at a time, seeing whether this satisfies the condition in an if statement, and if so completing the rest of the problem (although it would be harder to implement reusable functions sensibly this way, which was a main objective of the project, making this somewhat obsolete). Doing so would mean there was no limit on N due to memory, since the position vector would be reallocated over each iteration of the for loop, although once more for big N, computation times would still be quite long, though I suspect increase at a slower rate than the current method. I actually coded these up for each problem, and found it seemed faster (and even in some cases able) to compute solutions for larger values of N, with no effect on the accuracy. Note the results were constant between both approaches, further increasing my confidence that my solutions are correct.

A get_accepted_points function code

```
function [acceptedPoints] = get_accepted_points(points, condition_function)
%Author: Alex McInerny, Date: 24/01/2023
%Function that returns a matrix for which specific condition
%   is satisfied by every row in the output matrix, and rows which don't
%   satisfy the condition are not returned.
%Inputs:
%   *points: (N x D) matrix containg N points each of dimension D
%   *condition_function: a function which has input of a D dimensional
%       vector, and returns a boolean operator depending on whether the
%       condition of the function is satisifed
%Outputs:
%   *acceptedPoints: (M x D) matrix containg M rows, with each row being a
%       D-dimensional vector which fufills the conditions of
%       condition_function

i1 = 1; %set a counter to 1
for i2 = 1:size(points,1) %iterate through every row of the matrix 'points'
    if condition_function(points(i2,:)) == true %if condition_function is satisfied for specific row..
        acceptedPoints(i1, 1:size(points,2)) = points(i2,:); %add that vector to a new matrix which//
                %shall be returned once all rows in 'points' have been iterated through
        i1 = i1+1;%add one to the counter
    end
end
```

B calculate_sum function code

```
function [value] = calculate_sum(acceptedPoints, func)
%Author: Alex McInerny, Date: 24/01/2023
%Function that returns a value equal to the sum of a defined function, func
%   for each point in acceptedPoints
%Inputs:
%   *acceptedPoints: (M x D) matrix containg M points each of dimension D
%       used in the calculation
%   *func: a function which acceptes a vector and calcules a required value
%       from it's elements
%Outputs:
%   *value: number equal to sum of the function func for every point in
%       acceptedPoints

value = 0; %initialises the value which will be returned to zero
for i1 = 1:size(acceptedPoints, 1) %iterates through every point in acceptedPoints
    value = value + func(acceptedPoints(i1,:)); %adds func to that point and adds it to value
end
```

C Calculate the volume of the unit 10-dimensional sphere code

```
%Author: Alex McInerny, Date: 24/01/2023
%This code is designed to calcluate the volume of a D-Dimensional
%   hypersphere using a Monte-Carlo simulation

%clears command window and variables, changes format of numbers to long
clc; clear variables; format long

%asks user the dimenion of the hypersphere, and the number of 'points' they would like to iterate through
%D = input("What is the dimension, D, of the hypercube you'd like to calculate the volume of? ");
D = 10;
%N = input("How many iterations of different sets of points, N, would you like? ");
N = 1000000;

%defines condition function. i.e., the norm of the vector r is less than one
condition_funciton = @(r) (sum(r.^2) < 1);
```

```
%produces an (NxD) matrix of uniformly distributed random points
points = -1+(1--1)*rand(N, D);
%applies get_accepted_points function to points with required condition function, and stores the output//
        %in a matrix acceptedPoints
acceptedPoints = get_accepted_points(points, condition_funciton);

%calculates and prints the volume of the hypersphere
volume = size(acceptedPoints, 1)/N * 2^D;
fprintf("The volume of the hypersphere is"); disp(volume)
```

Output: `The volume of the hypersphere is 2.544947200000000`

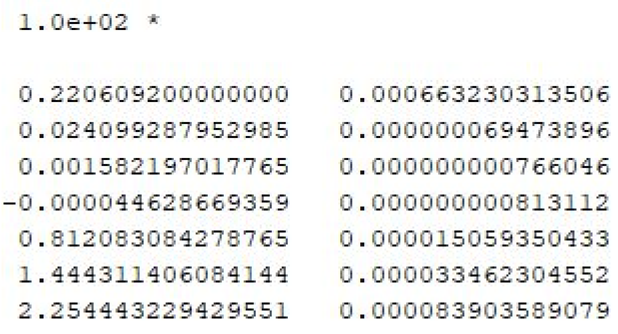
```
The volume of the hypersphere is   2.544947200000000
```

D Calculate the mass, centre of mass and moment of inertia of a complicated shape code

```
%Author: Alex McInerny, Date: 24/01/2023
%This code is designed to calcluate mass, centre of mass coordinate and
%   moments of inertia of a section of a torus using a Monte Carlo
%   simulation

%clears command window and variables, changes format of numbers to long
clc; clear variables; format long

%asks user for the number of 'points' they would like to iterate through and defines used variables
%N = input("How many iterations of different sets of x, N, would you like?");
N = 100000;
rho = 1;
volumeOfBoundingBox = (4-1)*(4--3)*(1--1);

%defines condition function. i.e., the equation of the torus
condition_function = @(r) (r(3)^2+(sqrt(r(1)^2+r(2)^2)-3)^2) <= 1;

%creates a matrix of uniformly distributed random points, each column having a different condition//
        %to satisfy the bounding box
points(:,1) = 1+(4-1)*rand(N,1); points(:,2) = -3+(4--3)*rand(N,1); points(:,3) = -1+(1--1)*rand(N,1);
%applies get_accepted_points function to points with the required condition function, storing result//
        %in acceptedPoints
acceptedPoints = get_accepted_points(points, condition_function);

%defines all function handles used to calculate the quantities requried in the formulas for//
        %M, x_cm and I_xx etc.
f1 = @(r) (rho);f2 = @(r) (rho^2);f3 = @(r) (r(1)*rho);f4 = @(r) (r(2)*rho);f5 = @(r) (r(3)*rho);
f6 = @(r) ((r(1)*rho)^2);f7 = @(r) ((r(2)*rho)^2);f8 = @(r) ((r(3)*rho)^2);
f9 = @(r) (((r(2)+r(3))^2)*rho);f10 = @(r) (((r(3)+r(1))^2)*rho);f11 = @(r) (((r(1)+r(2))^2)*rho);
f12 = @(r) ((((r(2)+r(3))^2)*rho)^2);f13 = @(r) ((((r(3)+r(1))^2)*rho)^2);//
        %f14 = @(r) ((((r(1)+r(2))^2)*rho)^2);
%stores all these function handles in an array of cells, ready to be called in a for loop
sum_functions = {f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14};

%iterate through the different function handlesplace the result of the calculate_sum function //
        %into a corrisponding point in an array called summations
for i1 = 1:length(sum_functions)
    summations(i1, 1) = calculate_sum(acceptedPoints, sum_functions{i1}); %place the result of//
            %the calculate_sum function (with the desired func) into a corrisponding point in//
            %an array called summations
end

%calculate desired quantities and their errors and place them in an array
solutions(1,1) = volumeOfBoundingBox*summations(1)/N; %mass
solutions(1,2) = volumeOfBoundingBox*sqrt(abs(((summations(2)/N)-(summations(1)/N)^2)/N));%error on mass
for i2 = 1:3 %iterate through cyclic permutations, x,y,z
    solutions(i2+1,1) = volumeOfBoundingBox*summations(i2+2)/(N*solutions(1,1));%centre of mass
    solutions(i2+1,2) = volumeOfBoundingBox*sqrt(abs(((summations(i2+3)/N^2)* ...
            (summations(i2+2)/N)^2)/N))/solutions(1,1); %error on centre of mass
    solutions(i2+4,1) = volumeOfBoundingBox*summations(i2+8)/N;%moment of inertia
    solutions(i2+4,2) = volumeOfBoundingBox*sqrt(abs(((summations(i2+9)/N^2)* ...
            (summations(i2+8)/N)^2)/N));%error on moment of inertia
```

```
end

%display desired quantities in a table
disp(solutions)
```

Output:

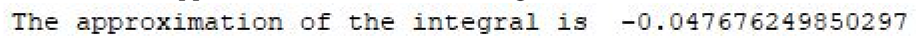


```
 1.0e+02 *

 0.220609200000000   0.000663230313506
 0.024099287952985   0.000000069473896
 0.001582197017765   0.000000000766046
-0.000044628669359   0.000000000813112
 0.812083084278765   0.000015059350433
 1.444311406084144   0.000033462304552
 2.254443229429551   0.000083903589079
```

E Calculate an integral over a surface code

```
%Author: Alex McInerny, Date: 24/01/2023
%This code is designed to approximate the integral of multivariable function
%   over a specified disk using a Monte Carlo simulation

%clears command window and variables, changes format of numbers to long
clc; clear variables; format long

%asks user for the number of 'points' they would like to iterate through and//
        %sets variables used through code
%N = input("How many iterations of different sets of r, N, would you like? ");
N = 100000;
surfaceAreaOfBoundingArea = (0.5--0.5)*(0.5--0.5);

%defines condition function. i.e., checking if a point is contained within the disk
condition_function = @(r) (sum(r.^2) < 0.25);

%creates a (Nx2) matrix of uniformly distributied random points between -0.5 and 0.5
points = -0.5+rand(N,2);
%applies get_accepted_points function to points with the required condition function,//
        %stores result in acceptedPoints
acceptedPoints = get_accepted_points(points, condition_function);

%defines the function handle used to calcluate
f1 = @(r) (sin(log(r(1)+r(2)+1))*surfaceAreaOfBoundingArea/N);

%uses the calcualte_sum function to approdumate the value of the integral using the//
        %accepted points and desired function to integrate, f1
i = calculate_sum(acceptedPoints, f1);

%displays the approximation to the integral
fprintf("The approximation of the integral is"); disp(i);
```

Output: `The approximation of the integral is -0.047676249850297`



```
The approximation of the integral is  -0.047676249850297
```