Probabilistic Algorithms: Homework 1

Laurent HAYEZ

2 octobre 2016

Exercise 1. *Consider the quarter circle of radius 1 inside the square of edge 1 (see the figure)*

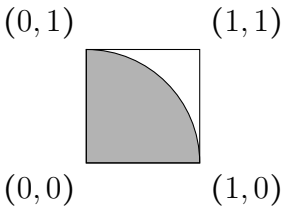

FIGURE 1 – Quarter circle inside unit square

1. *Consider the following experiment : select randomly a point inside the unit square (i.e. generate a tuple (x, y) from the uniform distribution on $[0, 1] \times [0, 1]$). What is the probability that such a point, randomly generated, falls inside the quarter circle ?*

   Solution. The area of the unit square is $\mathcal{A}_1 := 1^2 = 1$ and the area of the quarter disk is $\mathcal{A}_2 := \frac{\pi}{4}$. Let $(x, y) \in [0, 1]^2$ be randomly generated from the uniform distribution. Call C the quarter disk of radius 1 in the unit square, then

   $$\mathbb{P}[(x, y) \in C] = \frac{\mathcal{A}_2}{\mathcal{A}_1} = \frac{\frac{\pi}{4}}{1} = \frac{\pi}{4}.$$ □

2. *Based on the answer for question 1., write a MatLab function* `pi_estimate(N)` *that calculates an estimate of* π *using* N *trials of this experiment.*

   Solution.

```
function pi = pi_estimate( N )
%PI_ESTIMATE generates random points in [0,1] x [0,1] and it is used to
%approximate PI by using the fact that the probability that a point is
%in the quarter circle is PI/4.
% Input:
%   N: an integer > 0 that represents the number of times the estimate
%      must be repeated
% Output:
%   pi: an estimate of PI.

count_in = 0; % nb of pts randomly generated in the quarter disk C

for i=1:N
    rand_point = rand(1,2);
    if norm(rand_point) <= 1 % (x,y) is in C if its 2-norm is <= 1
        count_in = count_in + 1;
    end
end
```

   □

3. *For a fixed* N*, call the* `pi_estimate()` *function* 100 *times and calculates, based on the vector of* 100 *estimates for* π*, the following values : the minimum estimate* (π_{min})*, the maximum estimate* (π_{max})*, the average estimate* (π_{mean})*, the standard deviation of estimates* ($\Delta\pi$)*.*

   Solution. We use the following script (n was replaced by 100, 1000, ..., 10000000 for point 4.

```
N = 100; n = 100;
% creates a 1xn matrix with an estimate of pi using N approx
estimate = arrayfun(@pi_estimate, repmat(N, 1, n));
minn = min(estimate); maxx = max(estimate);
meann = mean(estimate); stdd = std(estimate);
```

   □

4. *Fill the following table :*

N	π_{min}	π_{max}	π_{mean}	$\Delta\pi$
100				
1000				
10000				
100000				
1000000				
10000000				

Solution.

N	π_{min}	π_{max}	π_{mean}	$\Delta\pi$
100	2.72	3.44	3.1344	0.1524
1000	2.36	3.60	3.1399	0.1632
10000	2.36	3.72	3.1401	0.1658
100000	2.32	3.80	3.1416	0.1639
1000000	2.24	3.84	3.1418	0.1641
10000000	2.28	3.88	3.1415	0.1642

□

Exercise 2. *Consider a two-sided error Monte-Carlo algorithm which may return two out- puts (decisions),*

$$\begin{cases} \text{"yes"}, & P(\text{"yes" is wrong}) = \varepsilon < 1/2 \\ \text{"no"}, & P(\text{"no" is wrong}) = \varepsilon < 1/2. \end{cases}$$

Suppose that this algorithm is running 10 times on the same input and generate the following sequence of decisions : "yes", "no", "no", "yes", "yes", "yes", "no", "no", "yes", "yes".

1. *Propose a rule that allows the user to choose the "right" decision after N trials on the same input. What gives your rule for this particular case ($N = 10$) ?*

   Solution. Let "yes" = 1, "no" = 2 and let $e_i \in \{1, 2\}$ represent the output of the algorithm when it is run for the i-th time.

   $$e_i = \begin{cases} 1 & \text{if output = "yes",} \\ 2 & \text{if output = "no".} \end{cases}$$

Compute

$$S := \sum_{i=1}^{N} (-1)^{e_i}.$$

If $S > 0$ then return "no", else if $S < 0$ return "yes", else return "yes" or "no" chosen uniformly at random.

The probability that the answer returned after N rounds is wrong is

$$\varepsilon^{\text{number of "yes"}} \text{ or } \varepsilon^{\text{number of "no"}},$$

hence by choosing the output that appears the most during the N rounds, we ensure that the probability of choosing the wrong output is minimal. If we have the same number of "yes" and "no", then we choose the output uniformly at random, because the probability of choosing the wrong output is the same whether we choose "yes" or "no". □

2. *If $P($ "yes" is wrong$) \neq P($ "no" is wrong$)$, do you think you must modify the rule ?*

Solution. Suppose that $\mathbb{P}[\text{"yes" is wrong}] = \varepsilon_1 \leq 1$ and $\mathbb{P}[\text{"no" is wrong}] = \varepsilon_2 \leq 1$. The rule becomes : compute $a := \varepsilon_1^{\text{number of "yes"}}$ and $b := \varepsilon_2^{\text{number of "no"}}$. If $a < b$ then return "yes", else if $b < a$ then return "no", else return "yes" or "no" uniformly at random.

This ensures that we choose the ouput with which the probability of being wrong is the smallest. □

Exercise 3. *Suppose we have a random number generator that, at each call, returns one of the two values $\{a, b\}$ with a probability of 0.5. Propose a method of how to use this generator to get uniform numbers on $(0, 1)$.*

Solution. We know that every real number has a binary representation, and we know that there is a bijection between $\mathbb{R}$ and $(0, 1)$, which is given by

$$\begin{array}{rccc} \varphi : & (0,1) & \to & \mathbb{R} \\ & x & \mapsto & \tan\left(\pi x - \frac{\pi}{2}\right) \end{array}$$

and its inverse given by

$$\begin{array}{rccc} \varphi^{-1} = \psi : & \mathbb{R} & \to & (0,1) \\ & x & \mapsto & \frac{1}{\pi}\left(\arctan(x) + \frac{\pi}{2}\right) \end{array}$$

Let us identify a with 0 and b with 1, and let $l \in \mathbb{N}$ be the seed of the generator. We start by generating a sequence of l 0s and 1s, then we get the decimal representation

of the generated sequence, which we call $m_1 \in \mathbb{Z}$ (the numbers are in $\mathbb{Z}$ because we used signed binary representation). We store $n_1 = m_1 \pmod{2^{31} - 1} \in \mathbb{N}$ and we return $\psi(m_1) \in (0, 1)$. We then generate a sequence of length n_1 which gives us $m_2 \in \mathbb{Z}$ in decimal representation. We store $n_2 = m_2 \pmod{2^{31} - 1} \in \mathbb{N}$ and return $\psi(m_2) \in (0, 1)$. We continue in this fashion to generate random numbers in $(0, 1)$.

We store numbers $\pmod{2^{31} - 1}$ to prevent generating arbitrary big numbers which can't be represented by a computer. □