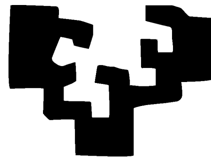
eman ta zabal zazu



Universidad del País Vasco | Euskal Herriko Unibertsitatea

Seminar 1

1 Synopsis.

What we are set about to do. We are going to use simulation to compute an approximation to number π. Not that we will ever compute π using this method —there are much better ways, as you will see below. Next, we will compute an area below the $N(0, 1)$ density curve —the kind of thing you would usually search for in statistical tables.

The beauty of using simulation is that it is quite general, and will enable you to tackle problems that would be hard or impossible to solve in any other way. After you gain some insight on how to use this method, you will be ready for Activity 1.

What you need. In order to benefit from this seminar you need:

1. Access to a computer running R. Most machines in the computer rooms have R loaded. Otherwise, you can download R at no charge[1] and install it in any computer you may own or have access to[2].
2. Some rudiments of R.
3. A working knowledge of the notions of density and distribution functions, the laws of large numbers and convergence in probability.

Aside from that, some background information is provided in the Section 2 next.

2 Background information

2.1 Random number generators.

For a variety of purposes, we would like to have a way of generating a stream of random numbers from a given distribution with a computer. This may strike you as an impossible task, since a computer is a perfectly deterministic machine.

This is true: however you may generate with a computer streams of numbers, called "pseudo-random numbers", which for all practical intents and purposes behave like truly random numbers. How exactly this is done, need not concern us here: you may see for instance [5] and [3] or the summary in the Wikipedia, https://en.wikipedia.org/wiki/Random_number_generation, if you are curious.

In R you have good facilities to generate pseudo-random numbers. A family of functions starting with `r` generate random numbers with different distributions: `runif` (uniform), `rnorm` (normal), `rf` (Snedecor's F), `rchisq` (χ^2), etc. For instance,

[1] From CRAN, http://cran.es.r-project.org or http://www.et.bs.ehu.es/cran.

[2] In which case, we strongly advise you to also install RStudio, which is also a free grab from http://rstudio.org.

```
> runif(5)
```

```
[1] 0.1379541 0.6865969 0.8773901 0.3172691 0.1365857
```

generates 5 random numbers from a uniform distribution $U(0,1)$ (default). Should you want uniform random numbers from, say, a $U(7,10)$, you might obtain them by typing

```
> runif(5,min=7,max=10)
```

```
[1] 9.998017 7.461953 7.158535 7.380092 7.975465
```

2.2 Flow control in R.

In its simplest incantation, a computer program is a sequence of instructions, much as you would type directly in the R console. They are written in a file and submitted at once to the computer, which performs them one after the other.

In practice, this would not be of much use, as we frequently want the flow of the instructions to be dependent on some conditions. Several specialized instructions allow us to do so.

for. Sometimes, we want to do something repeatedly a given number of times. The instruction `for` allows easy iteration. If we wanted to add the first 5 natural numbers, we could write:

```
> Total <- 0
> Total <- Total + 1
> Total <- Total + 2
> Total <- Total + 3
> Total <- Total + 4
> Total <- Total + 5
```

but it would be far easier (and less error prone) to write:

```
> Total <- 0
> for (i in c(1,2,3,4,5)) {
    Total <- Total + i
    }
```

Let's dissect the previous code. The instruction tells: "For all values of `i` in the set, do whatever is written between the opening { and closing } brackets. There is even shorthand notation to describe sets of integers, so the above code could still be rewritten as:

```
> Total <- 0
> for (i in 1:5) {
    Total <- Total + i
    }
```

where `1:5` stands for all integers from 1 to 5.

if. Sometimes we want our code to follow a different path according to some condition. For instance, if we are betting on a game which will produce us a prize of 10 if a uniform random number is greater than 0.7 and a loss (negative price) of 3 if the random number is smaller than or equal 0.7, we might code:

```
> Prize <- 0
> if ( runif(1) > 0.7 ) {
     Prize <- 10
  }  else {
     Prize <- -3
   }
```

Notice that the condition "if a uniform U(0,1) random variable is greater than 0.7" is a convenient way of saying "with probability 0.3". We might as well code:

```
> Prize <- 0
> if ( runif(1) < 0.3 ) {
     Prize <- 10
  }  else {
     Prize <- -3
   }
```

Assume now you want to have an idea of what would be your the total amount in prizes if you play repeatedly that game, say 100000 times. You might code an `if` inside a `for`:

```
> Prize <- 0
> for (i in 1:100000) {
    if ( runif(1) > 0.7 ) {
       Prize <- Prize + 10
    }  else {
       Prize <- Prize - 3
    }
  }
> Prize

[1] 89246
```

so on average you have made 0.89246 per game. This closely agrees with the theoretical computed mean value of the game,

$$10 \times 0.3 + (-3) \times 0.7 = 0.90$$

which is the amount a player would have to pay per game so that the expected return is zero.

Aside from the brief in-class introduction to R, you may benefit from using any of the available introductory books: among them, [6], [2], [7] and the freely available documents in `http://cran.r-project.org`.

2.3 The Monte Carlo method.

The Monte Carlo method is a way to solve problems through simulation. The principle is quite simple, and is best demonstrated by way of example.

2.3.1 Computing number π.

You know that the surface of a circle is given by the formula πr^2, where r is the radius and $\pi = 3.14159265\ldots$, an irrational number. There are quite simple and fast methods to compute decimal digits of π to any desired accuracy[3].

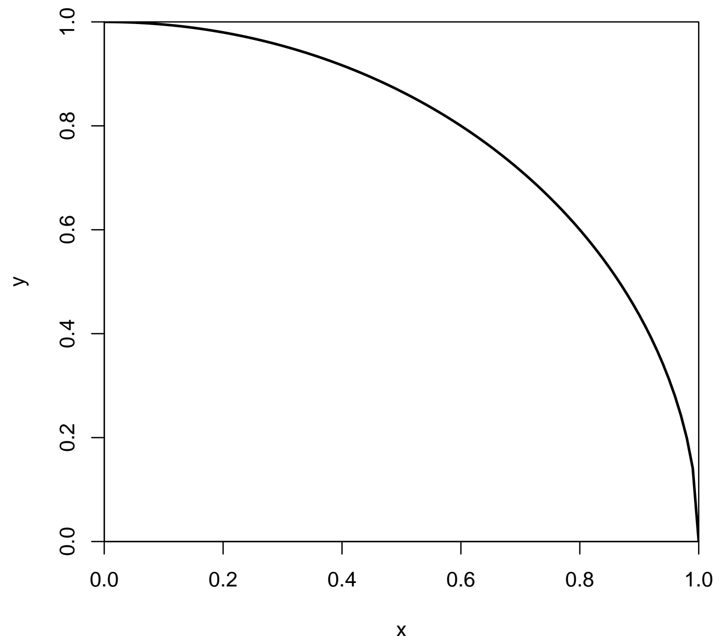
Now, suppose these methods were unavailable to you. You could approximate π easily by simulation reasoning as follows (refer to Figure 1): "The surface inside the quarter circle is $\pi/4$; if I generate a great many points uniformly distributed in the square $[0,1] \times [0,1]$ and I compute the proportion of those falling inside the quarter circle, that should be approximately the proportion of areas of the quarter circle and the inscribing square."

In other words, you could do something as follows:

```
> N <- 10000
> Inside <- 0
> for (i in 1:N) {
```

[3]For instance, it can be shown that $\sum_{k=1}^{\infty}(-1)^k \frac{1}{2k-1} = \frac{\pi}{4}$ so taking enough terms in the sum π can be approximated as closely as desired. You can see many more ways of computing π and other constants in books like [4] or [1].

Figure 1: Quarter circle inscribed in $[0,1] \times [0,1]$.




```
    x <- runif(1) ; y <- runif(1)
  if (x^2 + y^2 < 1)
    Inside <- Inside + 1
}
```

Let's look at the code: `N` is the number of points you want to generate. You generate first the `x` coordinate, then the `y`, in both cases uniformly between 0 and 1 (the default of `runif`). When the `for` loop ends, `Inside` counts the number among those points falling inside the quarter circle (those points must verify $x^2+y^2 < 1$).

Now, your approximation of $\pi/4$ (ratio of the area of the quarter circle to the total area of the square, which is 1) would be[4]:

```
> Inside / N
```

```
[1] 0.7891
```

You may check that you are not far off (and you could do still better by increasing `N`):

```
> pi / 4
```

```
[1] 0.7853982
```

The example presented is certainly artificial, as there are much faster and simpler ways to calculate π; the importance of the method lies in that it is completely general, and allows approximate integration[5] of functions were all other methods fail.

[4] Your results may vary slightly from those shown, from those obtained by a fellow student, and even from the ones you obtain in different runs. The streams of pseudo-random numbers generated are different each time you run the program. You might obtain the same stream by fixing the "seed": type `help(set.seed)` while in R to see how this is done.

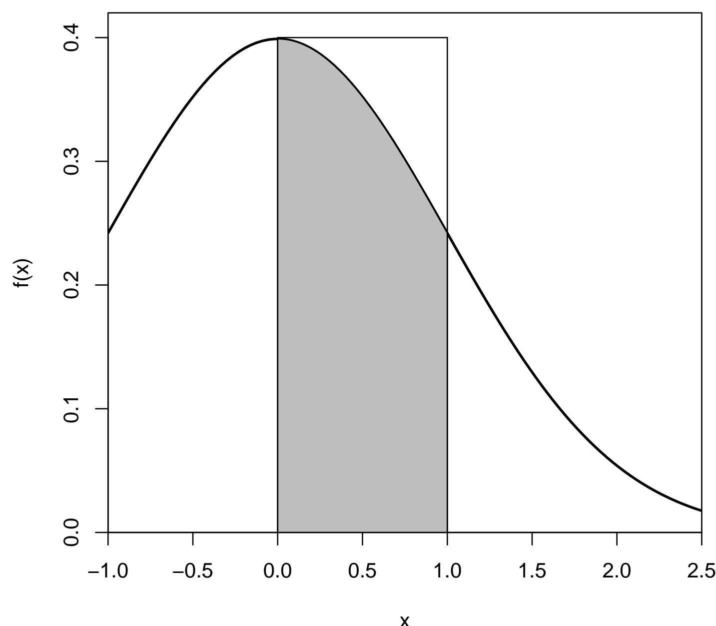
[5] In fact, the value of $\pi/4$ that we have approximated can be seen as the integral $\int_0^1 \sqrt{1-x^2}dx$; make sure you understand why.

Computing the probability under the $N(0,1)$ distribution of a certain interval. As a second example, consider the $N(0,1)$ density function,

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}. \quad (1)$$

We want to obtain the approximate integral of such function over the interval $[0,1]$ (see Figure 2), using simulation.

Figure 2: Normal density; shaded area is the probability of the interval $[0,1]$.



We will mimic what we did in the first example. We know that the maximum value of the density is $1/\sqrt{2\pi} = 0.3989423$, so the shaded area we are interested in is completely inside the box $[0,1] \times [0,0.4]$ represented in Figure 2. We will generate random points inside that box and count how many lie below the density curve —in the greyed area—. The proportion of points in that area times 0.4 (the area of the box) will give us a good approximation of the greyed area..

The following short code would compute the desired probability to a good approximation:

```
> N    <- 10000    # try 20000, 50000....
> Hits <- 0
> for (i in 1:N) {
    x <- runif(1, min=0, max=1)
    y <- runif(1, min=0, max=0.4)
    if (y < dnorm(x))
      Hits <- Hits + 1
  }
> Prob <- ( Hits / N ) * 0.4
```

We may compare the value in `Prob` with the "exact" value which can be computed directly with function `pnorm` in R:

```
> Prob
```

```
[1] 0.3408

> pnorm(1) - pnorm(0)

[1] 0.3413447
```

3 Some comments and questions

Now, you can play at will changing `N` in the little code fragment above, and running it over and over. Think about and try to answer the following questions:

1. With larger `N` you should get a better a approximation of π. What statistical result or theorem makes you confident that as you increase `N` you will likely be closer to the true value of $\pi/4$?
2. The approximation of $\pi/4$ you obtain is random: it has a variance. Is that variance going to zero?
3. If your answer to your previous question is "yes", how fast?
4. We would be much happier with our newly acquired method to estimate $\pi/4$ if we could make an statement of the sort: "If we use this procedure with `N` larger than such and such, the probability that we commit an error larger than 0.01 in estimating $\pi/4$ will be smaller than 0.02."

   Can we make such an statement? (Hint: Remember Tchebychev inequality?)
5. Regarding the second example (area under the normal curve between 0 and 1.0): we have taken care to enclose the desired area into a box $[0,1] \times [0,0.4]$ as small as possible. What would be the effect of using a larger box? For instance, we could generate random numbers in $[0,1] \times [0,1]$ and the proportion of "hits" below the density curve would be directly our estimate of the grey area.

References

[1] M. Abramovitz and I. Stegun, editors. *Handbook of Mathematical Functions*. Dover Pub., 1965.

[2] P. Dalgaard. *Introductory Statistics with R*. Statistics and Computing. Springer-Verlag, 2002. Signatura: 519.682 DAL.

[3] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

[4] A. Jeffrey. *Handbook of Mathematical Formulas and Integrals*. Academic Press, 1995.

[5] D. K. Knuth. Fundamental algorithms. In *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Mass., 1968.

[6] M.D. Ugarte, A.F. Militino, and A.T. Arnholt. *Probability and Statistics with R*. CRC Press, 2008.

[7] W.N. Venables and B.D. Ripley. *Modern Applied Statistics with S-Plus*. Springer-Verlag, New York, third edition, 1999.