Algoritmos Computacionales Proyecto intermedio

Mejía Arratia Yalth Estefania

Velasco Gutiérrez Rosa Gabriela

Ejercicio 1

```
using Plots
```

```
(process:11556): GLib-GIO-WARNING **: 11:21:04.678: Unexpectedly, UWP
app `Clipchamp.Clipchamp_2.2.12.0_neutral__yxz26nhyzhsrt' (AUMId
`Clipchamp.Clipchamp_yxz26nhyzhsrt!App') supports 46 extensions but
has no verbs
```

1. Vértices de un triángulo equilátero

Sin dibujarlos, considera tres puntos en un plano que formen los vértices de un triángulo equilátero

```
# Definiremos los 3 puntos dentro de dos vectores
# para expresarlos en sus coordenadas (x,y)

ve= ([0,0], [10, 0], [5,5])
X = [0, 10, 5]
Y = [0, 0, 5]

A= (0, 0)
B= (10, 0)
C= (5, 10)

println(ve)
```

```
([0, 0], [10, 0], [5, 5])
```

2. Posición actual

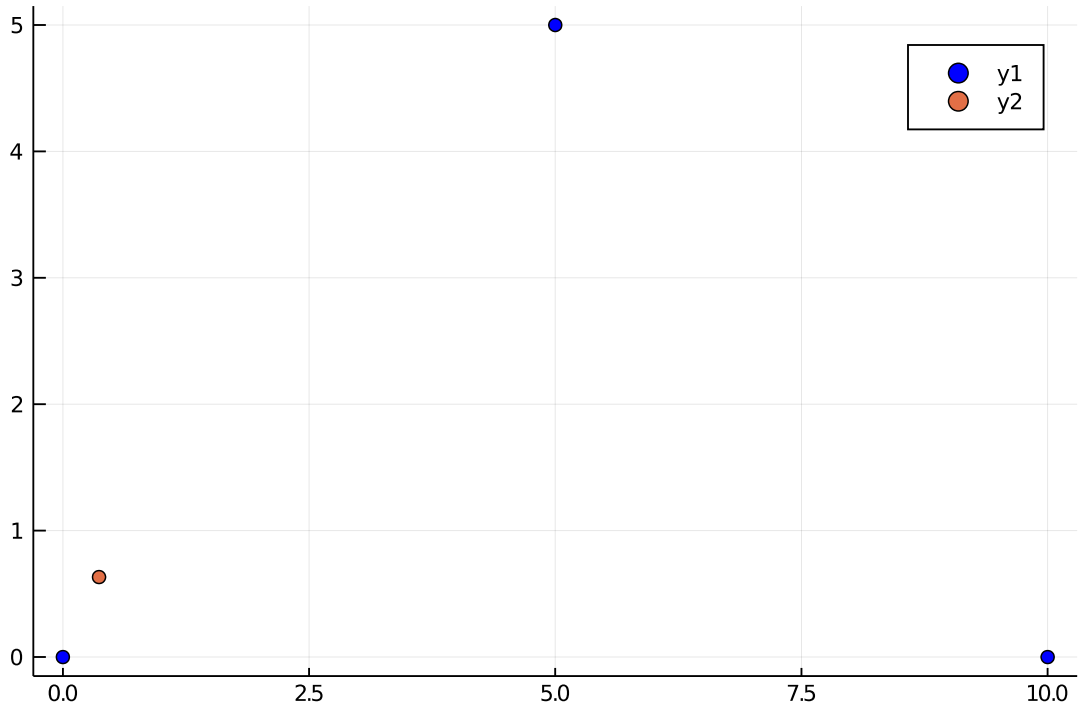
Elige un punto arbitrario dentro de la superficie del triángulo equilátero y considéralo tu posición actual.

```
# utilizaremos la función rand()
p_x = rand()
p_y = 1 - p_x

p_x =[p_x]
p_y =[p_y]

println(p_x)
```

```
println(p_y)

Pi= (p_x, p_y)

println(Pi)
```

```
[0.3673222443933406]
[0.6326777556066594]
([0.3673222443933406], [0.6326777556066594])
```

```
#scatter(p_x,p_y, color= :green)
scatter(X, Y, color= :blue)
scatter!((p_x, p_y))
```



3. Uno de los tres vétices

Elige de forma aleatoria uno de los tres vértices del triángulo equilátero.

```
V= rand((X), 1)
Vy= rand((Y),1)

V= V,Vy

print(V)
```

```
([10], [0])
```

4. Punto medio

Obtén el punto medio entre tu posición actual y el vértice que elegiste en el paso anterior, considéralo tu nueva posición actual.

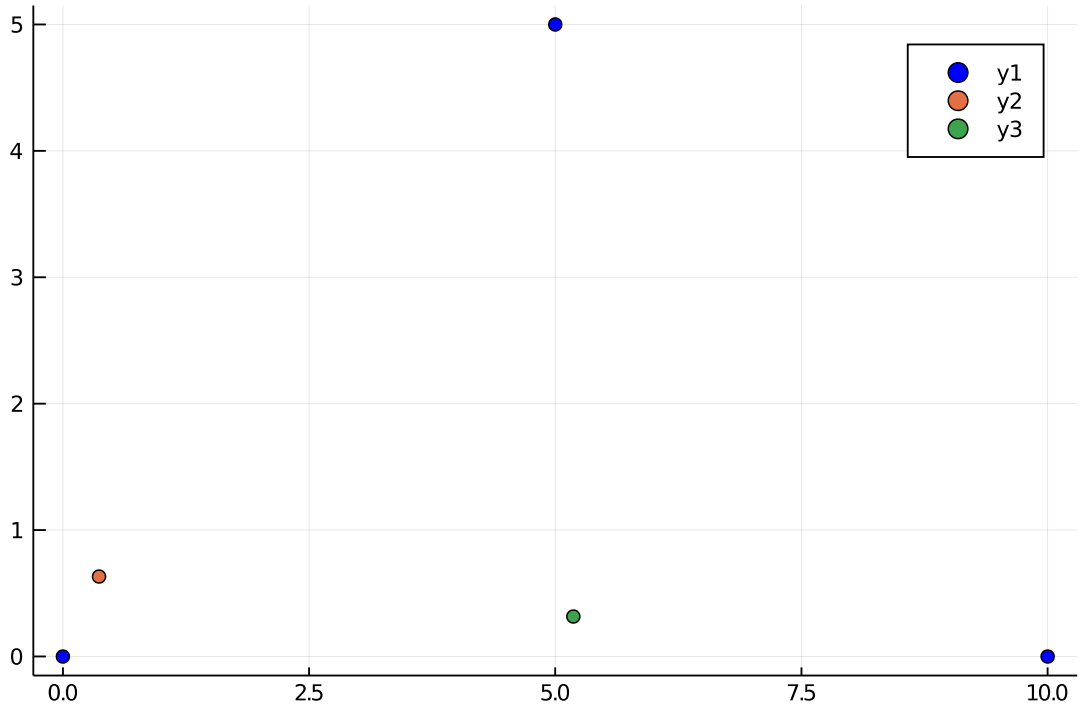
```
function punto_medio(P, V)
    med = (P .+ V) ./2
    return med
end
```

```
punto_medio (generic function with 1 method)
```

```
p_actual= punto_medio(Pi, V)
```

```
([5.18366112219667], [0.3163388778033297])
```

```
scatter(X, Y, color= :blue) #Vértices
scatter!((p_x, p_y)) #Punto aleatorio
scatter!(p_actual) #Posición actual
```



6. Repetir

Repite desde el paso 3

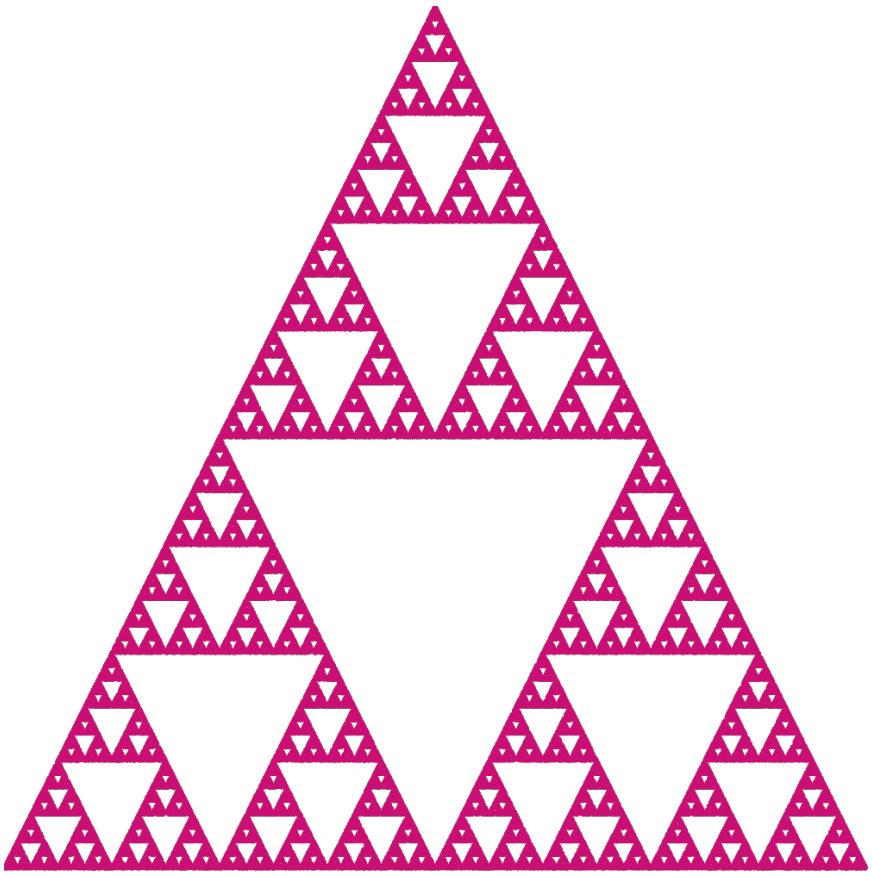
```
using Luxor
```

```
#Para hacer el ciclo recursivo definiremos una función TS
function TS()
```

```
    #Utilizando Drawing(pixeles de ancho, pixeles de alto, nombre de
```

```
la imagen) de Luxor, se guarda la figura en formato png para mostrarla
posteriormente
    Drawing(1300, 1300, "./Triángulo_S.png")

    #Ahora con llamamos Turtle(0,0 pendown=true, orientación=0,
pencolor=()) para dibujar la figura
    T = Turtle(0, 0, true, 0, (0.8, 0.06, 0.46))


    #Definiremos el tamaño de los lados de nuestro triángulo
    L = 1000

    #Punto inicial random
    p_x = rand(1:L) # Con rand() le damos un valor a la coordenada x
dentro del intervalo 1: tamaño de los lados del triángulo
    p_y = rand(1:L) #Lo mismo para la coordenada y


    #Ciclo recursivo
    n =100000 #damos el número de iteraciones a realiza
    #entre mayor sea el triángulo tendrá una mejor resolución de
imagen

    for i in 1:n #utilizamos for para repetir el bloque de código en
cada iteración
        #Vértices
        V = rand(1:3) #usando rand(), elegimos uno de los 3 vértices
        #Casos
        if V == 1 #Para cuando se tenga el primer vértice
            #En este caso las coordenadas de x serán las del punto
inicial entre 2, ya que no hay con que calcular el punto medio
            # con esto tendremos una nueva posición inicial
            p_x = p_x/2
            p_y = p_y/2

        elseif V == 2 #Para cuando se tenga el segundo vértice
            p_x = L/2 + (L/2 - p_x)/2 #calculamos el punto medio entre
el punto p=p_x, p_y y el vértice seleccionado
            p_y = L - (L - p_y)/2

        else #Para el ultimo caso (vértice 3)
            p_x = L - (L - p_x)/2
            p_y = p_y / 2

        end

        # Utilizando Reposition(t::Turtle, pos::Point) de Luxor
colocamos a T dentro de los puntos calculados dentro del ciclo
recursivo
        Reposition(T, p_x, L-p_y) # se usa L-p_y para que los puntos
```

```
de p_y no se grafiquen fuera del triángulo

        #Finalmente usamos Turtle(t::Turtle, radio=1.0) de Luxor para
dibujar cada punto dentro del triángulo
        # en este caso tienen un radio = 2.5
        Circle(T, 2.5)
    end
end

TS (generic function with 1 method)

#Usando nuestra función Ts
TS()

#terminamos la animación con finish()
finish()

#abrimos el png que se guardo anteriormente como "./Triángulo_S.png"
preview()
```



Ejercicio 3

```
import Pkg
Pkg.add("Plots")
```

```
    Updating registry at `C:\Users\gabri\.julia\registries\
General.toml`
   Resolving package versions...
  No Changes to `C:\Users\gabri\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\gabri\.julia\environments\v1.7\
Manifest.toml`
```

```
using Plots
```

1. Algoritmo que estime el valor de π

Escribe un algoritmo que estime el valor de π y que te permita visualizar algo similar al gráfico de la Figura 2, asegúrate de incluir el conteo del número de puntos rojos, número de puntos totales, y la respectiva estimación de π.

```
using Random #usaremos la paqueria Random para crear intervalos de números aleatorios

Random.seed!(31417) #utilizamos la función sedd() para iniciar el generador de números aleatorios
# el valor con el que iniciara el intervalo de números será el ubicado en 31415

TaskLocalRNG()

n = 100000 # cuantos puntos se van a producir en total
#entre más grande se tendra un mejor acercamiento a pi

100000

x = [rand() * rand((-0.5,0.5)) for _ in 1:n] #para el eje x se generaron una cantidad de números aleatorios
# se aplicará de -0.5 a 0.5

100000-element Vector{Float64}:
  0.013587815620244748
 -0.04688929565577443
 -0.43252294592963136
 -0.13998116230267688
 -0.052018975604578355
 -0.3298373294697698
 -0.2419991070833214
 -0.2753592383827538
 -0.1552284295545157
 -0.26220996223538456
 -0.003951924814897145
 -0.08610916394519202
  0.2724532571755319
  ⋮
  0.3805206576300762
  0.2194335217462498
  0.1986634784684156
 -0.41898947713373985
  0.11540120463000009
 -0.45500798145800464
  0.4280361210930058
  0.11455480741906882
 -0.06328069767821543
  0.19450837529025417
 -0.3641710371617009
  0.36217065216791183
```

```
y = [rand() * rand((-0.5,0.5)) for _ in 1:n] # lo mismo para el eje y
```

```
100000-element Vector{Float64}:
 -0.3025612578525958
  0.35569508949328704
 -0.4869239297188697
 -0.47402129826549994
 -0.13437035648522455
  0.3668061280343085
 -0.23157915724109152
 -0.3259699147175246
  0.38979912897005364
  0.16197736193663947
 -0.4925036217494282
 -0.4889890328582646
 -0.1029161798226032
  ⋮
 -0.21073864682875992
 -0.36020430139452825
 -0.4861595578342957
 -0.24919367338683807
  0.08533037604034666
 -0.11681021524774182
 -0.04461747778288566
  0.4173634453432005
  0.47696138058914994
 -0.4526717906482567
  0.4814289004578762
  0.2594067162561517
```

```
# Ahora determinaremos si los puntos (x,y) caen dentro del circulo o
no
cir= [x[i]^2 + y[i]^2 <= 0.25 for i in 1:n]
```

```
100000-element Vector{Bool}:
 1
 1
 0
 1
 1
 1
 1
 1
 1
 1
 1
 1
 1
 1
 ⋮
 1
 1
```

```
 0
 1
 1
 1
 1
 1
 1
 1
 0
 1
```

```
cir_sum = Int64[] # creamos una lista donde irán entrando los puntos
que caigan dentro del circulo
push!(cir_sum, cir[1])
```

```
1-element Vector{Int64}:
 1
```

```
for i in 1:(n-1) #usando un ciclo for decimos que para cada valor en
nuestro intervalo n
    #vamos sumando los puntos que caen en el circulo y los agregamos a
circulo_sum
    new_sum= cir_sum[i] + cir[i+1]
    push!(cir_sum, new_sum)
end
```

```
#comprobamos que el valor sea el mismo usando la función sum y
circulo=sum
cir_sum[end]
```

```
78462
```

```
sum(cir)
```

```
78462
```

```
cuad= [x[i]^2 + y[i]^2 > 0.25 for i in 1:n]
```

```
100000-element Vector{Bool}:
 0
 0
 1
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
```

```
 ⋮
 0
 0
 1
 0
 0
 0
 0
 0
 0
 0
 1
 0
```

```
cuad_sum = Int64[] # creamos una lista donde irán entrando los puntos
que caigan dentro del cuadrado
push!(cuad_sum, cuad[1])
```

```
1-element Vector{Int64}:
 0
```

```
for i in 1:(n-1) #usando un ciclo for decimos que para cada valor en
nuestro intervalo n
    #vamos sumando los puntos que caen en el circulo y los agregamos a
circulo_sum
    newcu_sum= cuad_sum[i] + cuad[i+1]
    push!(cuad_sum, newcu_sum)
end
```

```
cuad_sum[end]
```

```
21538
```

```
sum(cuad)
```

```
21538
```

```
pi_est = [4* cir_sum[i]/ i for i in 1:n]
```

```
100000-element Vector{Float64}:
 4.0
 4.0
 2.6666666666666665
 3.0
 3.2
 3.3333333333333335
 3.4285714285714284
 3.5
 3.5555555555555554
 3.6
 3.6363636363636362
 3.6666666666666665
```

```
 3.6923076923076925
 ⋮
 3.138465231175429
 3.1384738473847387
 3.138442459821384
 3.138451076086087
 3.1384596921784524
 3.138468308098486
 3.1384769238461923
 3.138485539421577
 3.1384941548246448
 3.138502770055401
 3.138471384713847
 3.13848
```

```
pi_est[end]
```

```
3.13848
```

```
println(pi_est[end])
println("Puntos dentro del circulo:", cir_sum[end])
println("Puntos dentro del cuadrado:", cuad_sum[end])
```

```
3.13848
Puntos dentro del circulo:78462
Puntos dentro del cuadrado:21538
```

```
# porcentaje de diferencia
(pi - pi_est[end]) / pi *100
```

```
0.09907884098966345
```

1. Parte de la gráfica

```
n = 10000 #utilizaremos 10000 en vez de 100000 para que corra bien
Plots
x = [rand() * rand((-0.5,0.5)) for _ in 1:n]
```

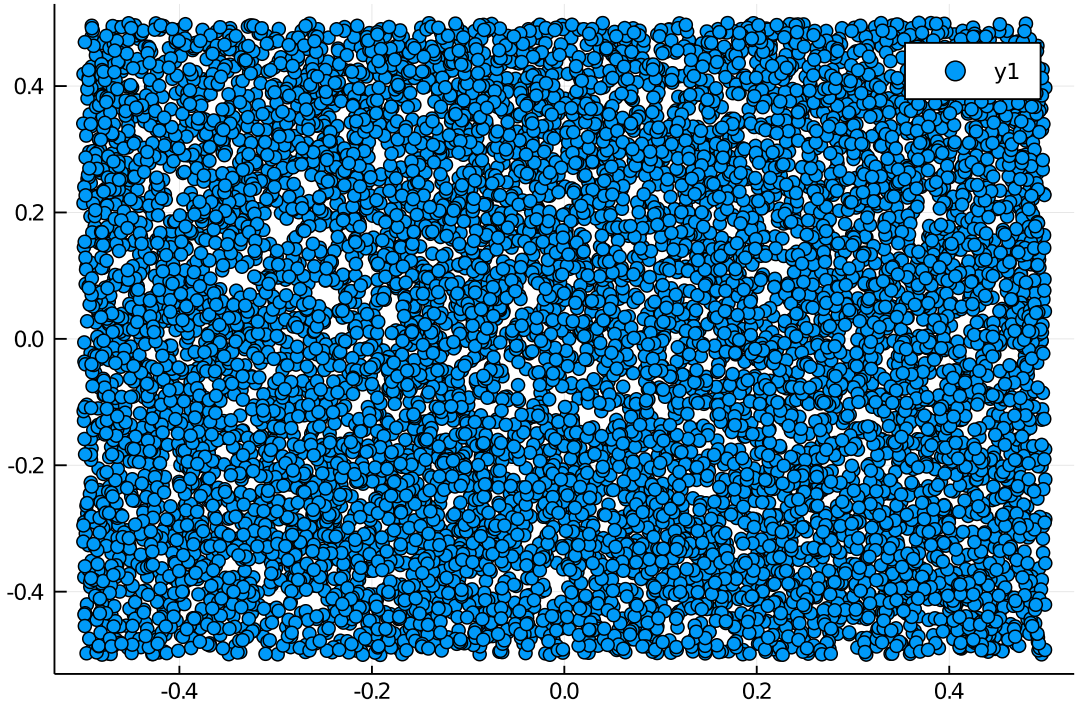
```
10000-element Vector{Float64}:
  0.36366716369647756
 -0.42165124780655927
 -0.2915751356614074
 -0.34697684550968116
  0.2834531414049347
  0.35552361041230784
  0.4956228734672266
 -0.20759013135619947
  0.18741142624328633
  0.23359886059905893
  0.08613625241589712
  0.33341529442951445
 -0.42693213448319206
```

```
  ⋮
 -0.1950768383638854
  0.17932606844950127
 -0.18175409147912103
 -0.23074125998360834
  0.18804089547468678
 -0.11569898771625464
 -0.33392985400273434
 -0.2172589839050591
 -0.11299744558260522
 -0.029040861667458007
  0.29599858336359763
  0.3455389606461736
```

```
y = [rand() * rand((-0.5,0.5)) for _ in 1:n]
```

```
10000-element Vector{Float64}:
  0.4704216508953132
 -0.49613403623243374
  0.026319189262549225
 -0.42688876179880675
 -0.2644633874123551
 -0.43436562800450595
  0.3642273258306128
 -0.14405914715049684
  0.4164867684978923
 -0.3891178264848749
  0.03954999886876398
  0.07700192057969268
 -0.11514302513338698
  ⋮
 -0.42372124198496935
  0.15128215237359094
  0.23427008404986438
 -0.41938573946691665
  0.12839673832936277
 -0.19283074958318364
 -0.18408920684368807
  0.15308283935190203
  0.3555800089456318
  0.18897978174155877
  0.1112615937414424
  0.03451823172028384
```

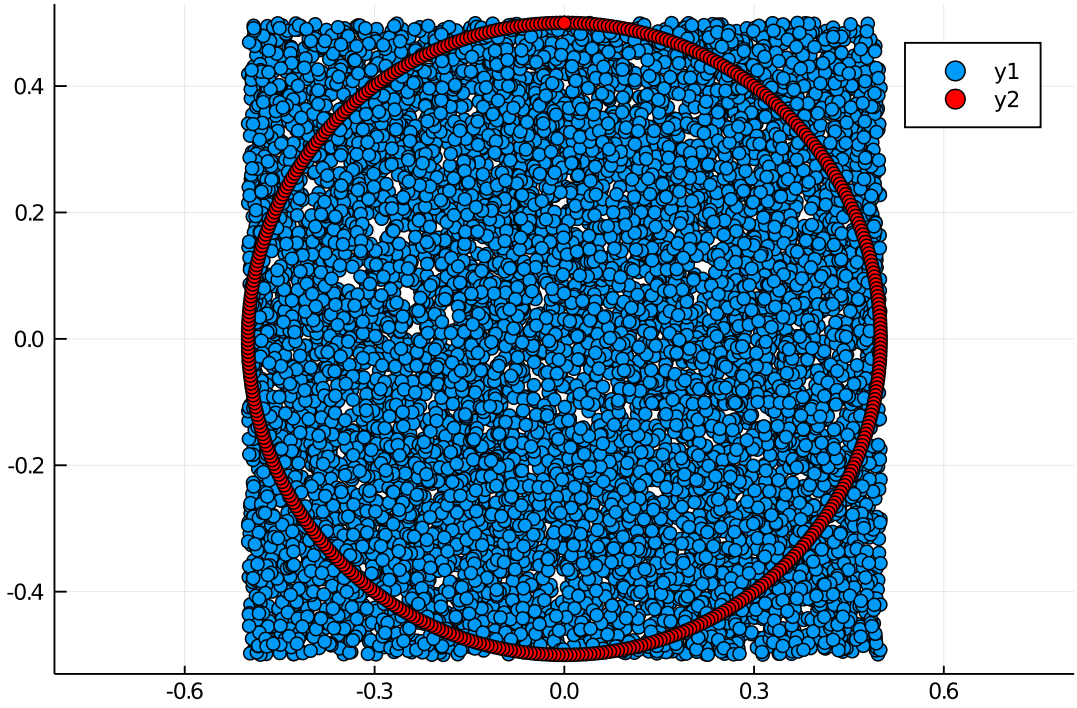
```
using Plots
scatter(x,y)
```




```
function circulo(x, y, r)
    theta= LinRange(0, 2*pi, 500)
    x .+ r*sin.(theta), y .+ r*cos.(theta)
end
```

```
circulo (generic function with 1 method)
```

```
scatter!(circulo(0,0, 0.5), color= :red, aspect_ratio= :equal)
```



$y^2=\left(r^2-x^2\right)\$\ \$\^{}\$\ \$\left(0.5\right)$

Para encontrar los que estan dentro y fuera del circulo

3. Gráfica del error de la estimación

Realiza una gráfica del error de la estimación en función del número de puntos comparando contra el valor predeterminado de π de Julia (que se obtiene llamando a la constante pi).

```
pi
```

```
π = 3.1415926535897...
```

```
n = 100000
estimacion= pi_est
error = [pi .- estimacion]
```

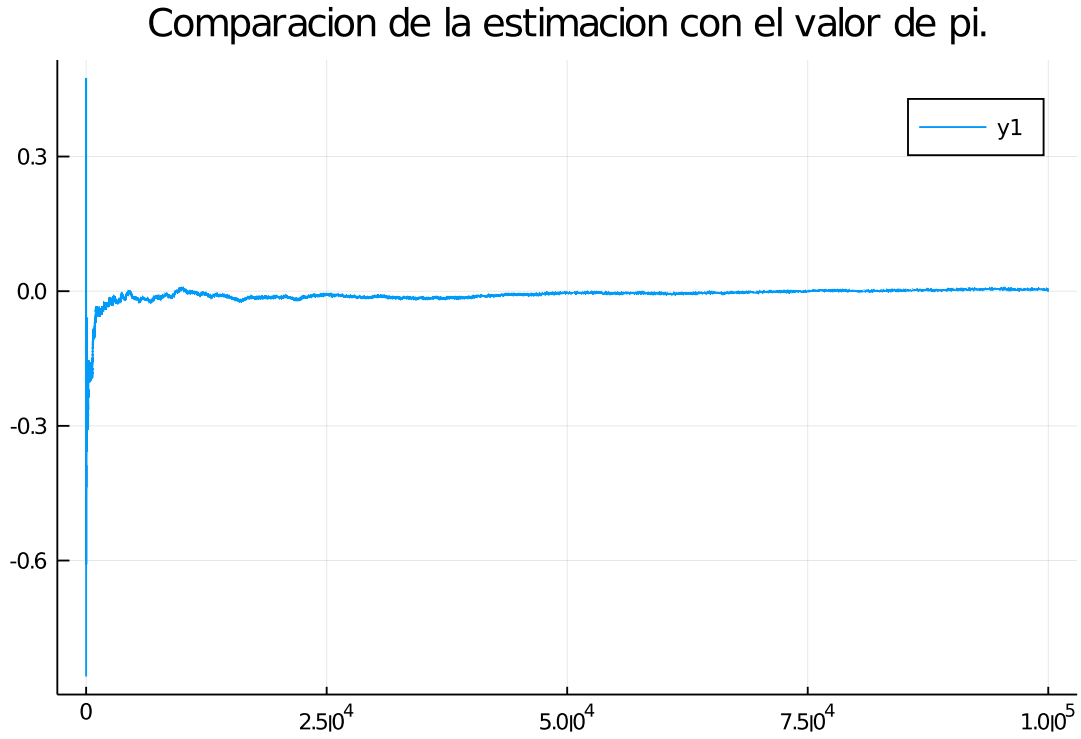
```
1-element Vector{Vector{Float64}}:
 [-0.8584073464102069, -0.8584073464102069, 0.4749259869231266,
0.14159265358979312, -0.05840734641020706, -0.19174067974354037, -
0.28697877498163527, -0.3584073464102069, -0.41396290196576224, -
0.458407346410207  …  0.003150193768409082, 0.003141577503706028,
0.0031329614113406734, 0.0031243454913072455, 0.003115729743600859,
0.003107114168216185, 0.0030984987651483387, 0.0030898835343919906,
0.003012126887594610 4, 0.0031126535897931795]
```

```
n = [1:100000] #volvemos nuestro n= 1000000 un vector con 1000000
valores
```

```
estimacion= pi_est
error = [pi .- estimacion]
```

```
1-element Vector{Vector{Float64}}:
 [-0.8584073464102069, -0.8584073464102069, 0.4749259869231266,
0.14159265358979312, -0.05840734641020706, -0.19174067974354037, -
0.28697877498163527, -0.3584073464102069, -0.41396290196576224, -
0.458407346410207  …  0.003150193768409082, 0.003141577503706028,
0.0031329614113406734, 0.0031243454913072455, 0.003115729743600859,
0.003107114168216185, 0.0030984987651483387, 0.0030898835343919906,
0.003012126887594 6104, 0.0031126535897931795]
```

```
plot(n, error, title= "Comparacion de la estimacion con el valor de
pi.")
```




```
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
```

```
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
GKS: character ignored due to unicode error
```