Universidad Veracruzana

Inteligencia Artificial

Análisis de Algoritmos

Actividad 7

Leonardo Flores Torres

6 de diciembre de 2022

1. Cálculo de π usando el *método de Monte Carlo*.
    - Generar los puntos aleatorios usando *convergencia lineal*.
    - Generar los puntos aleatorios usando *secuencia de Halton*.
    - Generar los puntos aleatorios usando el generador valores aleatorios de su lenguaje de programación de preferencia.
2. Para cada caso graficar las curvas de convergencia, hacer al menos 10^6 iteraciones.
3. Analizar y determinar qué método fue más preciso.
4. Pensar cómo aproximar π con un número de cifras significativas dadas, por ejemplo, con un número de 4 cifras significativas el valor correspondiente sería 3.14159.

    Solución:

    La implementación del algoritmo del *generador lineal congruencial* usada en esta actividad está basada en el trabajo hecho por Schlegel [1] la cual a su vez está basada en una implementación dentro del estándar del lenguaje de programación `C` [2], siendo esta última referencia de la que se obtienen los parámetros adecuados para este generador.

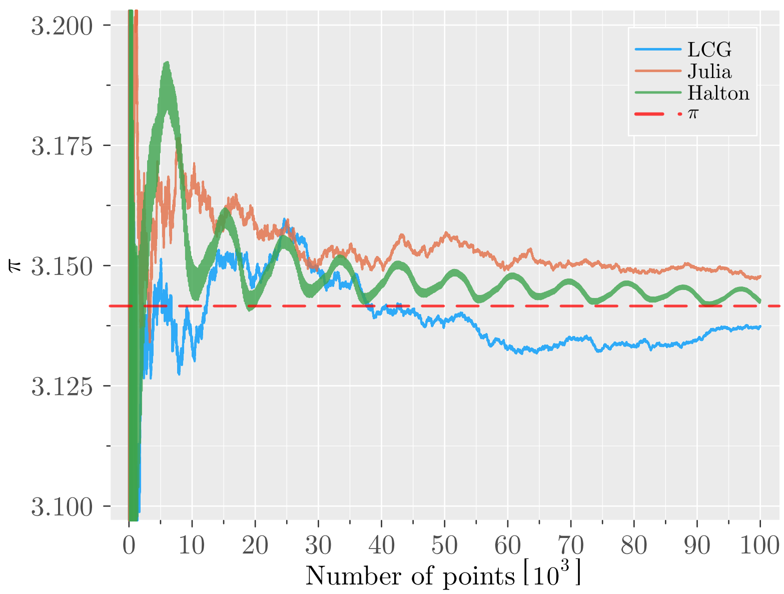
3.200
3.175
3.150
3.125
3.100
π
0 10 20 30 40 50 60 70 80 90 100
Number of points [10^3]
LCG
Julia
Halton
π

Figura 1: xxx

Apéndice

```julia
module ApproxPie

using Random
using FLoops

# Distance from the origin to a point
dist_from_origin(point) = point[1]^2 + point[2]^2
dist_from_origin(x, y) = x^2 + y^2

# Linear congruential generator
# https://aaronschlegel.me/linear-congruential-generator-r.html
"""
    rngLcg(nvals)

Returns a list of pseudo-random numbers using a linear congruential generator. This
implementation uses the default parameters as those from the ANSI C implementation from
the year 2000 by Saucier.
"""
function rngLcg(nvals)
    # Parameters
    m = 2^32
    a = 1103515245
    c = 12345
    # The power of the distributed computing
    seed = time() * 1000 |> BigFloat   # current system's time in micro seconds times 1000

    numbers = zeros(nvals)
    for i in eachindex(numbers)
        seed = (a * seed + c) % m
        numbers[i] = seed / m
    end

    return numbers
end

function rngLcg_floop(nvals)
    # Parameters
    m = 2^32
    a = 1103515245
    c = 12345
    # The power of the distributed computing
    seed = time() * 1000   # current system's time in micro seconds times 1000

    numbers = zeros(nvals)
    @floop for i in 1:nvals
        # s = seed
        # s = (a * s + c) % m
        seed = (a * seed + c) % m
        numbers[i] = seed / m
    end

    return numbers
```

```julia
end


# Halton sequence generator
"""
    rngHalton(nvals; base=2)

Returns a list of pseudo-random numbers generated using the Halton sequence. The
default `base` is 2.
"""
function rngHalton(nvals)
    base = rand(2:150)
    numbers = zeros(nvals)

    for i in eachindex(numbers)
        f = 1
        r = 0

        indx = i
        while indx > 0
            f = f / base
            r = r + f * (indx % base)
            indx = floor(indx / base)
        end

        numbers[i] = r
    end

    return numbers
end

# Julia's default random number generator
rngJulia(nvals) = rand(Float64, nvals)
# function rngJulia(nvals)
#     seed = rand(2:1500)

#     return rand(MersenneTwister(seed), Float64, nvals)
# end

function classifyPoints(points)
    inside_circle = filter(x -> dist_from_origin(x) <= 1, points)
    outside_circle = filter(x -> dist_from_origin(x) > 1, points)

    return (in = inside_circle, out = outside_circle)
end

function classifyPoints(xvals, yvals)
    inside_circle = []
    outside_circle = []

    for (x, y) in zip(xvals, yvals)
        if dist_from_origin(x, y) > 1
            push!(outside_circle, (x, y))
        else
```

```
            push!(inside_circle, (x, y))
        end
    end

    return (in = inside_circle, out = outside_circle)
end

function pieApprox(classification)
    ps = classification
    count_circle = length(ps.in)
    count_square = count_circle + length(ps.out)

    return 4 * count_circle / count_square
end

# function piesApprox(npoints; rng="julia")
#     methods = Dict(
#         "julia" => rngJulia,
#         "halton" => rngHalton,
#         "lcg" => rngLcg)

#     method = methods[rng]
#     xvalues = method(npoints)
#     yvalues = method(npoints)

#     pies = zeros(npoints)
#     for i in eachindex(pies)
#         classify = classifyPoints(xvalues[begin:i], yvalues[begin:i])
#         pie = pieApprox(classify)
#         pies[i] = pie
#     end

#     return (pies=pies, xvalues=xvalues, yvalues=yvalues)
# end

function piesApprox(npoints; rng="julia")
    methods = Dict(
        "julia" => rngJulia,
        "halton" => rngHalton,
        "lcg" => rngLcg)
    method = methods[rng]

    xvalues = method(npoints)
    yvalues = method(npoints)
    pies = zeros(npoints)

    for npoints_ in eachindex(pies)
        xwindow = xvalues[begin:npoints_]
        ywindow = yvalues[begin:npoints_]
        points_inside_circle = count(x -> x[1]^2 + x[2]^2 <= 1, zip(xwindow, ywindow))
        pies[npoints_] = 4 * points_inside_circle / npoints_
    end

    return (pies=pies, xvalues=xvalues, yvalues=yvalues)
```

```
end

function plotFig(classification)
    ps = classification
    dx = 0.01    # extra space to not cut circles at the limits

    fig = plot(size=(600,600),
        xlims=(-dx,1+dx), ylims=(-dx,1+dx),
        xticks=nothing, yticks=nothing)

    scatter!(ps.out, label="", ms=4, ma=0.5, msw=0.5)
    scatter!(ps.in, label="", ms=4, ma=0.5, msw=0.5)

    xaxis!(fig, bordercolor="white")
    yaxis!(fig, bordercolor="white")

    return fig
end

function estimate_pi_floop(nMC)
   radius = 1.
   diameter = 2. * radius

   @floop for i in 1:nMC
       x = (rand() - 0.5) * diameter
       y = (rand() - 0.5) * diameter
       r = sqrt(x^2 + y^2)
       if r <= radius
           @reduce(n_circle += 1)
       end
   end

   return (n_circle / nMC) * 4.
end


function rngLcg_floop(nvals)
    # Parameters
    m = 2^32
    a = 1103515245
    c = 12345
    # The power of the distributed computing
    seed = time() * 1000 |> BigFloat   # current system's time in micro seconds times 1000

    numbers = zeros(nvals)
    @floop for i in eachindex(numbers)
        seed = (a * seed + c) % m
        numbers[i] = seed / m
    end

    return numbers
end

# julia> n = 5000; approximations = ap.piesApprox(n);
```

```julia
# julia> npoints = map(x -> x[1], approximations); pies = map(x -> x[2], approximations);
# julia> scatter(npoints, pies, label="", ms=2, ma=0.8); hline!([π], label="", lw=2.5)

end # module ApproxPie
```

Referencias

[1] Aaron Schlegel. Linear congruential generator for pseudo-random number generation with r. https://aaronschlegel.me/linear-congruential-generator-r.html, 2008. Visitado: 2022-11-20.

[2] Richard Saucier. Computer generation of statistical distributions. Technical report, Army Research Lab Aberdeen Proving Ground MD, 2000.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 9 2017.