5

```
begin
    using LinearAlgebra
    using Plots
    using PlutoUI
    using HypertextLiteral: @htl, @htl_str
    using Distributions
    using Test
    using StatsPlots
    const CARD_SIZE::Int64 = 5
end
```

Bingo simulation

Question: How many steps does it approximately get to successfully fill a winning bingo table?

How does bingo work?

You are given a card with 25 small squares shaped in a one big 5x5 table. You fill in numbers between 1-100 in all tiles, except for the middle one. Later, the moderator starts randomly picking numbers from the same range and you cross yours if they match.

You finish by filling the tables row, column or diagonal (middle one is presumed as filled by default). The first person to fill such table combination wins.

Simulation

This notebook will include simulations of bingo related statistics. We will simulate average draws, before card is successfully filled, winning rate among more players etc.

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

```
#Init filled bingo card as 5x5 matrix of values
function init_bingo_card()
    choices::Set{Int64} = Set(1:100)
    card = Matrix{Int64}(undef, CARD_SIZE, CARD_SIZE)
    for i in 1:CARD_SIZE
        for j in 1:CARD_SIZE
            picked = rand(choices)
            card[i,j] = picked
            delete!(choices, picked)
        end
    end
    card[3,3] = 0
    return card
end
```

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

```
#Return true if bingo card is finished
#Return false if bingo card is not yet finished
function is_finished(card::Matrix{Int64})
    #Flag that does monitor diagonal finished state
    flagd1::Bool = true
    flagd2::Bool = true
    #We mark crossed out tiles as 0, since we have only positive numbers in bingo
    #sum of its rows will only give 0 if all tiles have been crossed out
    for i in 1:5
        if (sum(card[i,:]) == 0)
            return true
        end
        if (sum(card[:,i]) == 0)
            return true
        end
        if !(card[i,i] == 0)
            flagd1 = false
        end
        #Reverse diagonal check
        if !(card[i, (CARD_SIZE + 1) - i] == 0)
            flagd2 = false
        end
    end
    #We have two possible diagonals
    return flagd1 || flagd2
end
```

> We have created two functions. One does create a random bingo card and second will check, if it is finished.
>
> Remember, that we are going to mark crossed out numbers by replacing them with 0 for efficient computations.

Test is_finished() function

Let's test our written is_finished() function if it works correctly.

```
DefaultTestSet("Is finished card test", [], 5, false, false, true, 1.711487366002102e
```

```
Test Summary:          | Pass  Total  Time
Is finished card test  |    5      5  0.1s
```

Bingo game iterative simulation

In this section, we will simulate continual drawing of 100 numbers until we get finished.

Individual results will be kept in a vector, so we can plot them later.

```
@htl("""
<h1>Bingo game iterative simulation</h1>
<ul>In this section, we will simulate continual drawing of 100 numbers until we get finished. </ul>
<ul>Individual results will be kept in a vector, so we can plot them later.</ul>
""")
```

```
cross_out_num! (generic function with 1 method)
```

```
#Cross out given number if present in the card
function cross_out_num!(card::Matrix{Int64}, num::Int64)
    for i in 1:CARD_SIZE
        for j in 1:CARD_SIZE
            if (card[i,j] == num)
                card[i,j] = 0
            end
        end
    end
end
```

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

```
#Return hash map of finished states and their occurences in n simulations of bingo
function simulate_bingo_n_steps(n::Int64)
    res::Vector = Vector{Int64}()
    for i in 1:n
        card = init_bingo_card()
        not_drawn_nums::Set{Int64} = Set{Int64}(1:100)
        iter::Int64 = 0
        while !is_finished(card)
            picked = rand(not_drawn_nums)
            delete!(not_drawn_nums, picked)
            cross_out_num!(card, picked)
            iter += 1
        end
        # Append res to the hash table
        push!(res, iter)
        #print("\n")
        #show(stdout, "text/plain", card)
    end
    return res
end
```

Plotting

We created a function, that will simulate given scenario with n-attempts. How let's create a function to plot the results.

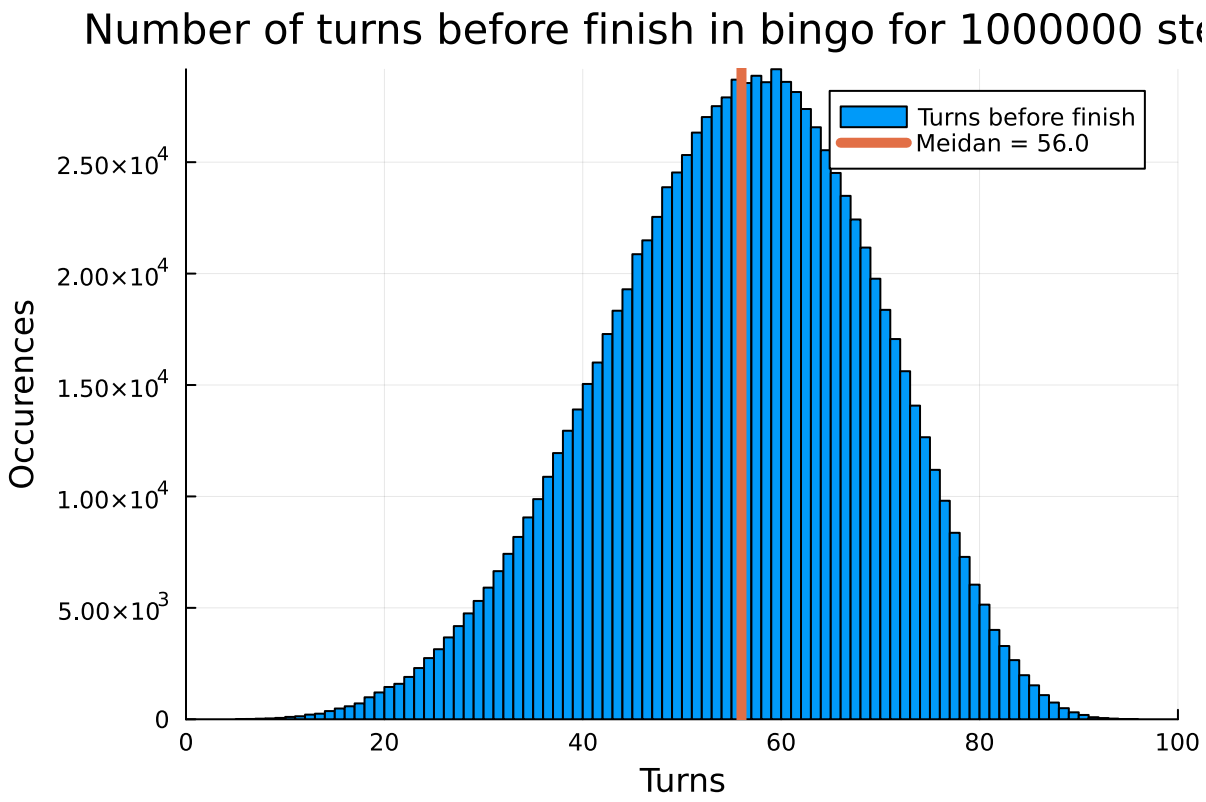
```
plot_simulation_results (generic function with 1 method)
```

```
#Given result dictionary plot histogram of individual numbers before finishing
function plot_simulation_results(res::Vector{Int64}, iterations::Int64)
    histogram(res, label="Turns before finish", bins=100)
    vline!([median(res)], label="Meidan = $(median(res))", linewidth=5)
    xlims!(0, 100)
    title!("Number of turns before finish in bingo for $(iterations) steps.")
    xlabel!("Turns")
    ylabel!("Occurences")
end
```

Interactive simulation

Chose number of simulations by the slider bellow and simulation histogram will be generated.

```
@bind iterations Slider(1:1000000)
```



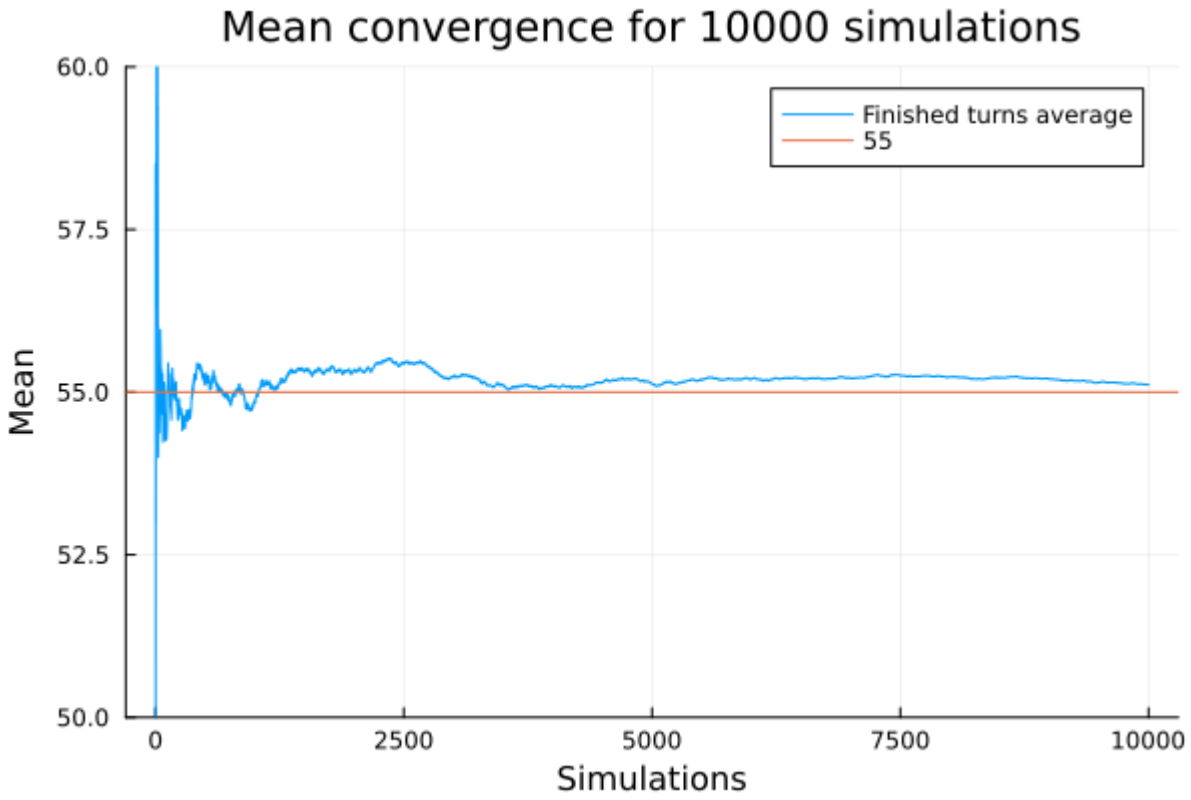
Evolution of average trough simulations

Let's plot, how the average of the steps to finish looks like over the time.

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

```
#Given vector of values, plot how their average converges iteratively
function iterative_average_plot(finishes::Vector{Int64}, steps::Int64)
    val = 0
    plot_seq::Vector{Real} = Vector{Real}()
    for i in 1:length(finishes)
        val += finishes[i]
        push!(plot_seq, val//i)
    end
    plot(plot_seq, label="Finished turns average")
    xlabel!("Simulations")
    ylabel!("Mean")
    ylims!(50, 60)
    hline!([55], label="55")
    title!("Mean convergence for $(steps) simulations")
end
```

```
1 @bind n_steps Slider(1:10000)
```



Convergence

We can visually evaluate, that average does converge to the value of 55. Given the histogram above, we can suspect, that values do follow **normal distribution**. This suspicion can be proved by CLT (central limit theorem), but we can also confirm, that our distribution follows normal one by some statistical tests.

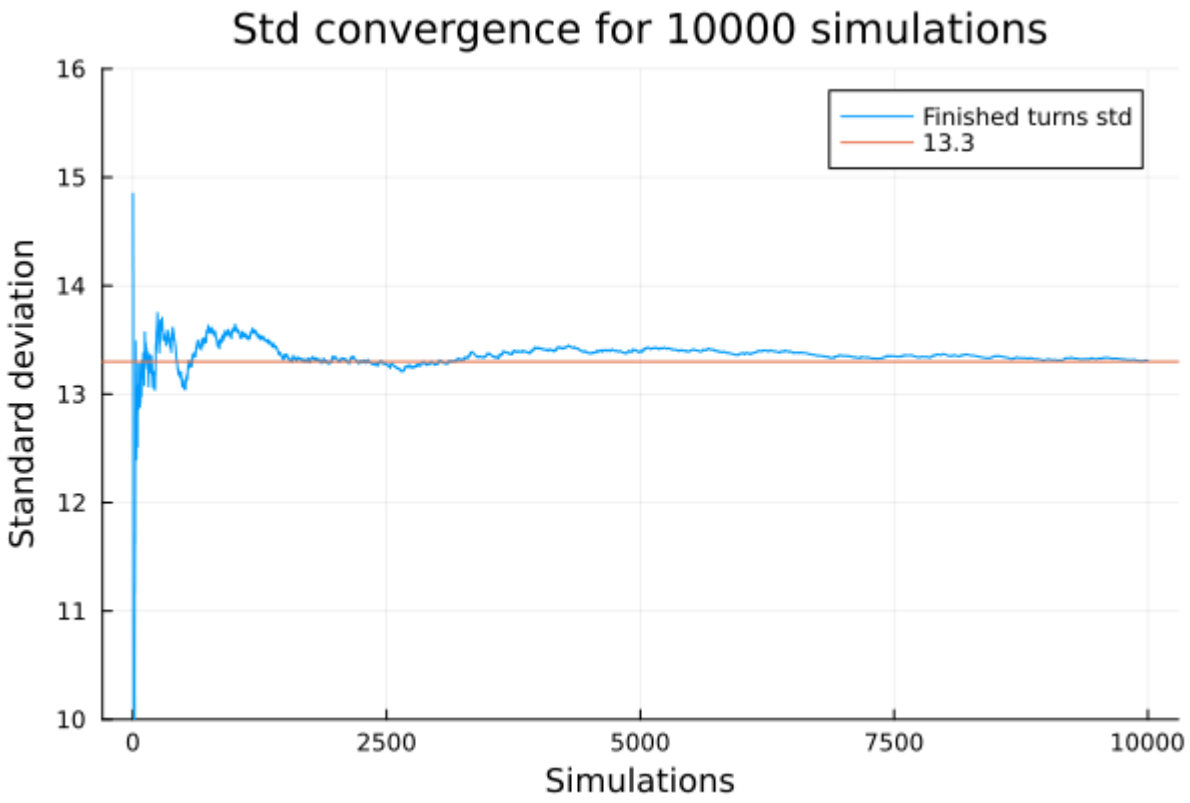
Bingo for n players

> Question: given n players, return mean number of turns, before first wins.
>
> For such simulation, we will make use of the CLT and approximate turns before finish of each player with normal distribution
>
> For such approximation, we first need to get standard deviation (std) of the turns before win.

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

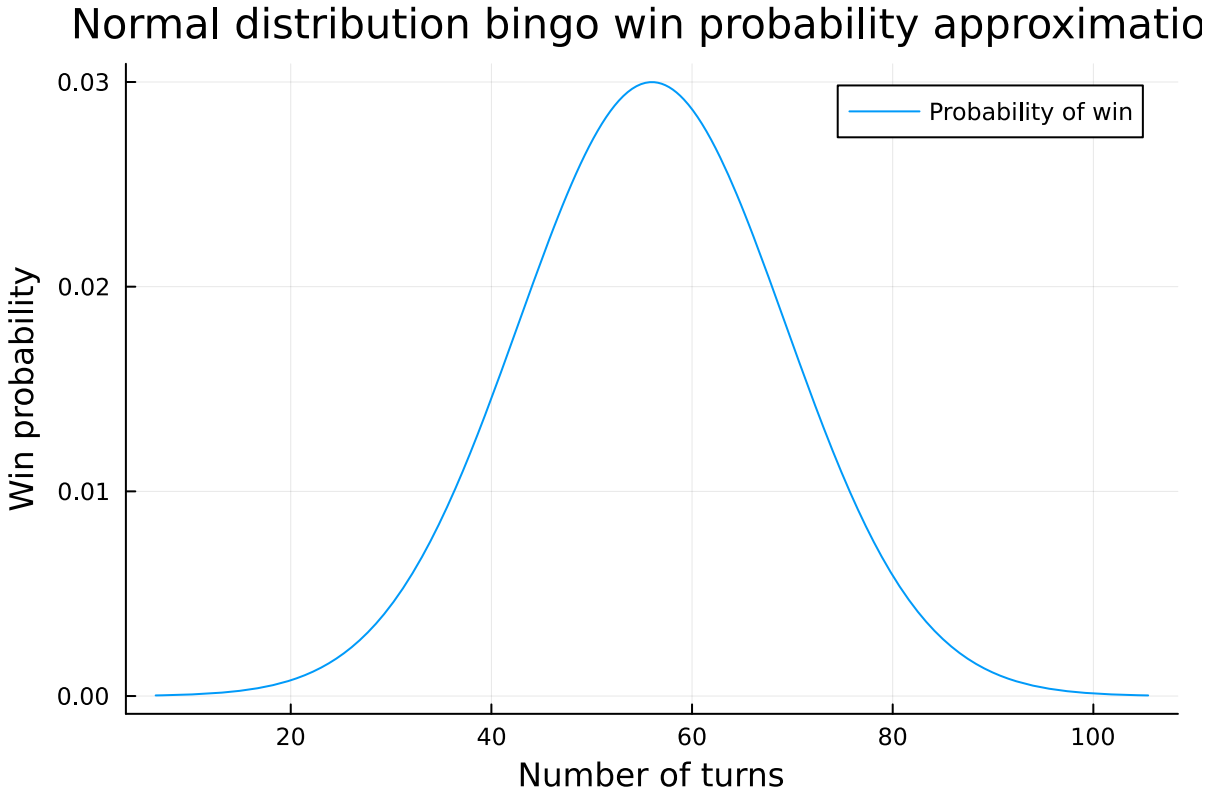
```
#Given result of simulations plot the evolution of variance
function plot_variance_evolution(turns::Vector{Int64}, steps::Int64)
    stds::Vector{Float64} = Vector{Float64}()
    for i in 1:length(turns)
        push!(stds, sqrt(var(turns[1:i])))
    end

    plot(stds, label="Finished turns std")
    xlabel!("Simulations")
    ylabel!("Standard deviation")
    ylims!(10, 16)
    hline!([13.3], label="13.3")
    title!("Std convergence for $(steps) simulations")
end
```

```
@bind n_steps Slider(1:10000)
```



Approxiamtion with normal distribution

Given the std and mean calculations from above, we can approximate the probability of win given number of turns by normal distribution.



Now it's time to evaluate loss of our model using the MSE and RMSE methods.