Problem Set 2

February 1, 2024

1 Part 1: Simulation with Metropolis-Hastings and Comparison

Task: Simulate a normal (Gaussian) distribution using the Metropolis-Hastings algorithm, targeting a mean (μ) of 0 and variance (σ^2) of 1.

Approach: Implement the Metropolis-Hastings algorithm within the framework provided by the `Elvis_simple.jl` file.

- *Acceptance Criterion*: Define the acceptance criterion based on the ratio of the target distribution probabilities and the proposal distribution probabilities.

The Metropolis-Hastings algorithm performs a sampling from a target distribution π by generating a Markov chain with a stationary distribution of π.

My solution is based on a proposal distribution with mean 0 and variance 1.5, as seen below. I run the solution 10,000 times.

1.1 Defining an initial proposal distribution

The code in Julia below defines the proposal distribution based on a normal distribution with mean 0 and variance 2.

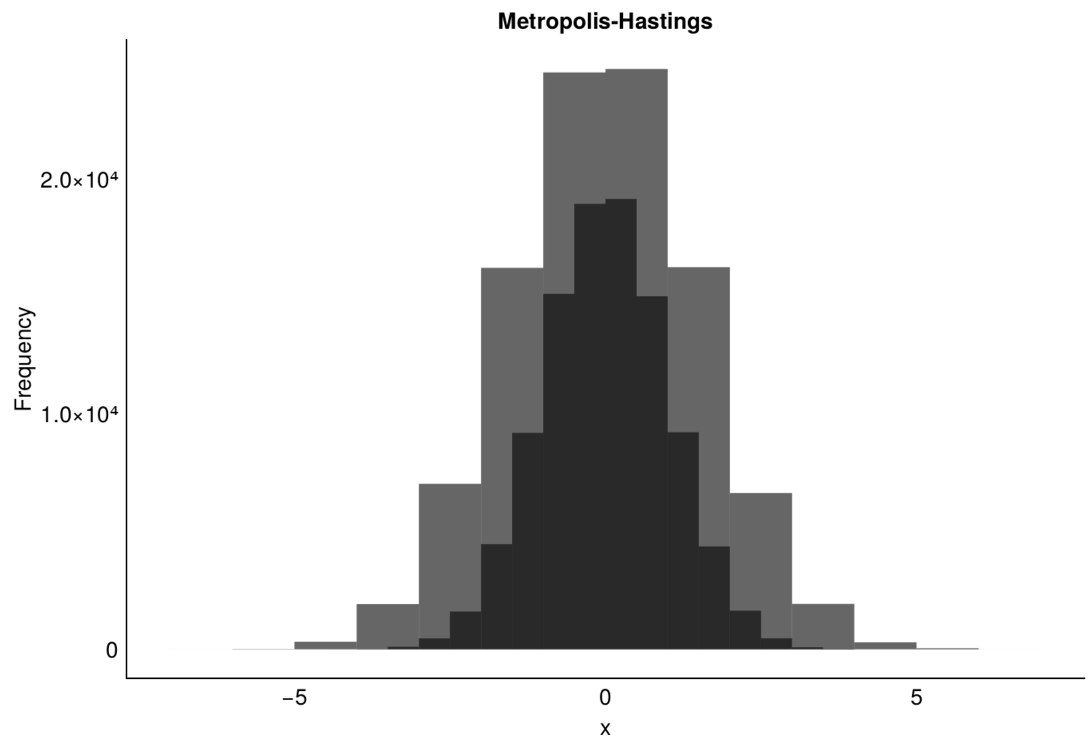
[ ]:
```
using Distributions
using Random
using DataFrames

function metropolis_hastings(n_samples::Int)
    # Initialize the chain
    chain = zeros(n_samples)
    current = randn()  # Start at a random place

    # Target distribution
    target = Normal(0, 1.5)

    for i in 2:n_samples
        proposal = current + randn()  # Add some noise

        # Calculate acceptance probability (as the ratio, given in the
↪instructions)
        p_accept = min(1, pdf(target, proposal) / pdf(target, current))
```

```julia
        # Accept or reject
        if rand() < p_accept
            current = proposal
        end

        chain[i] = current
    end

    return chain
end

# Generate 100,000 samples

samples = metropolis_hastings(100000)

# Make it a dataframe

samples_df = DataFrame(x= samples)
```

	x
	Float64
1	0.0
2	2.24555
3	2.20137
4	1.17754
5	-0.137905
6	-0.137905
7	0.583942
8	1.55178
9	2.22483
10	3.44804
11	3.74749
12	5.29991
13	4.4817
14	4.01859
15	4.01859
16	4.01859
17	3.21273
18	1.7521
19	0.883975
20	-0.0485275
21	0.433413
22	-0.085052
23	-0.216319
24	-0.216319
...	...

I plot the proposal distribution below to see how "normal" it looks. I use `TidierPlots`, the amazing Julia implementation of R's *ggplot2*. I also overlay a standard normal distribution (`randn()`) to compare the two, where the standard normal distribution is the darker distribution overlaid in the histogram below.

[ ]:
```
using Tidier

standard_normal = DataFrame(normal = randn(100000))

ggplot(samples_df, aes(x=:x)) +
    geom_histogram(binwidth=0.1) +
    geom_histogram(data = standard_normal, aes(x = :normal)) +
    labs(title="Metropolis-Hastings", x="x", y="Frequency") +
    theme_minimal()
```



ggplot options

```
height: 400
x: x
title: Metropolis-Hastings
width: 600
y: Frequency
```

```
geom_histogram
data: inherits from plot
x: not specified

geom_histogram
data: A DataFrame (100000 rows, 1 columns)
x: normal
```

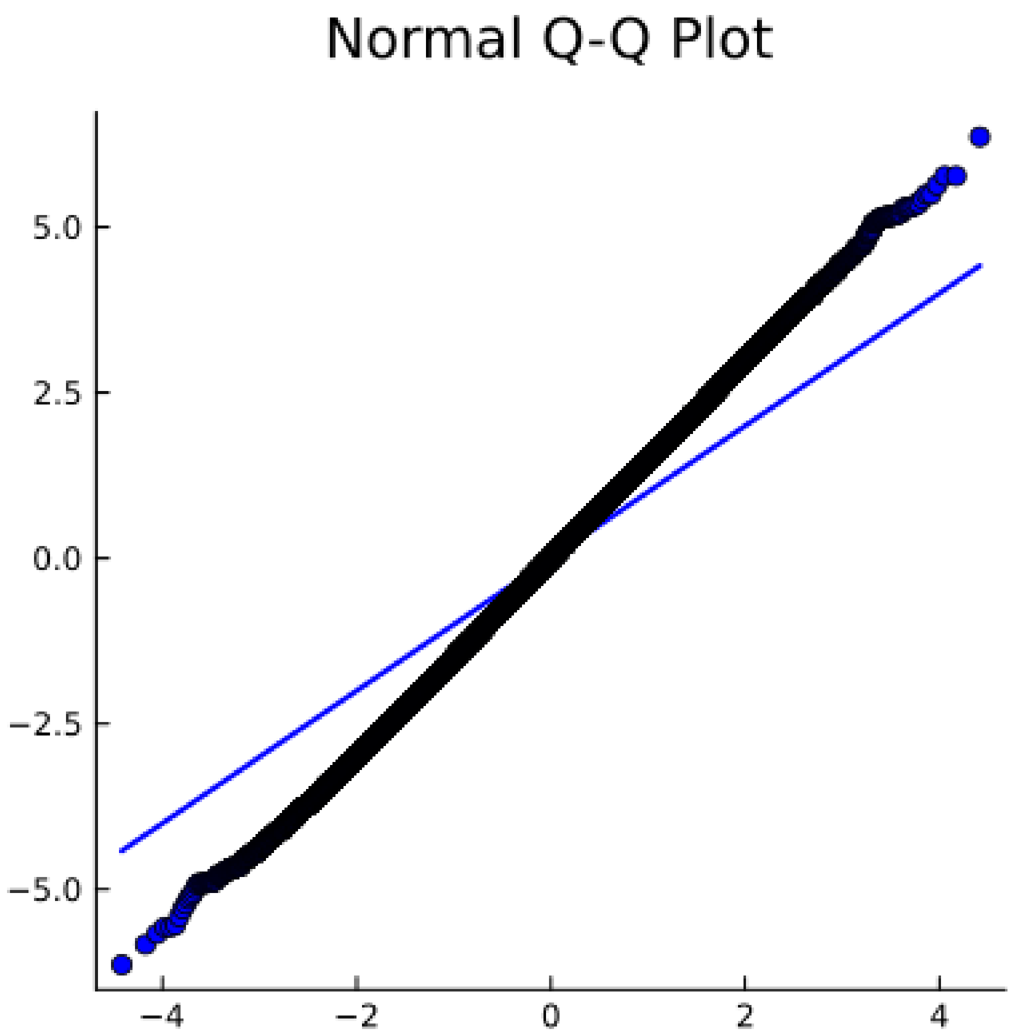
The data looks pretty normal, however, it can noted that my sample has wider tails, probably due to the greater variance. Below, I also computed descriptive statistics using `TidierData`, the Julia implementation of R's `dplyr`.

[ ]:
```
@chain samples_df begin
    @summarise(mean = mean(skipmissing(x)),
               std = std(skipmissing(x)))
end
```

	mean	std
	Float64	Float64
1	-0.00779182	1.49748

The mean is approximately 0, and the standard deviation is approximately 1.5, which are close to the target distribution's π defined mean and standard deviation, so the Metropolis-Hastings algorithm is working well in this regard. A Q-Q plot is presented below to see how well it fits a normal distribution.

[ ]:
```
using StatsPlots

qqplot(Normal(), samples, title="Normal Q-Q Plot", legend=false, color=:blue,
 ↪lw=2, grid=false, size=(400, 400))
```



The data has heavier tails than a normal distribution, but it is still a good approximation. As mentioned the heavier tails probably come from the variance of 1.5 in the target distribution π.

[ ]:
```
using HypothesisTests

ks_test = ExactOneSampleKSTest(samples_df.x, Normal(0, 1.5))

pvalue(ks_test)
```

```
 Warning: This test is inaccurate with ties
 @ HypothesisTests
C:\Users\user\.julia\packages\HypothesisTests\r322N\src\kolmogorov_smirnov.jl:68
```

```
0.07336724692448748
```

The test, however, ultimately rejects the possibility that `samples_df` is normally distributed with mean 0 and variance 1.5. So, the sampling is not perfect. The efficiency of the Metropolis-Hastings is smaller than the sampling process of the `randn()` function, given that the proposal distribution fails the Smirnov-Kolmogorov test.

1.2 Enhancing the Metropolis-Hastings algorithm

I enhance my algorithm by fine-tuning the proposal distribution, but also by varying the initial number of the Markov chain, as well as the scale. By making the algorithm a function, I can see how it performs with different parameters and compare how it might do better or worse in the Smilnov-Kolmogorov test.

The function `metro_hastings` takes the following parameters:

- `n`: the number of samples to generate
- π: the target distribution, with μ and σ^2 as parameters (mean and variance)
- `Init`: the initial value of the Markov chain
- `scale`: the scale of the proposal distribution

The code below shows how the function works. I also plot the results of the function with different parameters to see how it performs.

```
[ ]: function metropolis_hastings(n_samples::Int, target::Distribution, init::Real,␣
      ↪scale::Real)
         # Initialize the chain
         chain = zeros(n_samples)
         chain[1] = init

         n_accept = 0
         for i in 2:n_samples
             proposal = chain[i-1] + rand(Normal(0, scale))  # Add some noise

             # Calculate acceptance probability
             p_accept = min(1, pdf(target, proposal) / pdf(target, chain[i-1]))

             # Accept or reject
             if rand() < p_accept
                 chain[i] = proposal
                 n_accept += 1
             else
                 chain[i] = chain[i-1]
             end

             # Adapt the scale of the proposal distribution
             if i % 100 == 0 && i < n_samples/2
                 if n_accept / i < 0.2
                     scale *= 0.9
                 elseif n_accept / i > 0.4
                     scale *= 1.1
                 end
```

```
        end
    end

    return chain
end
```

```
metropolis_hastings (generic function with 2 methods)
```

Below, I set the initial value of the Markov chain to 0, and the scale of the proposal distribution to 1. I run the function 1,000,000 times, with a target distribution of π with mean 1 and variance 2.

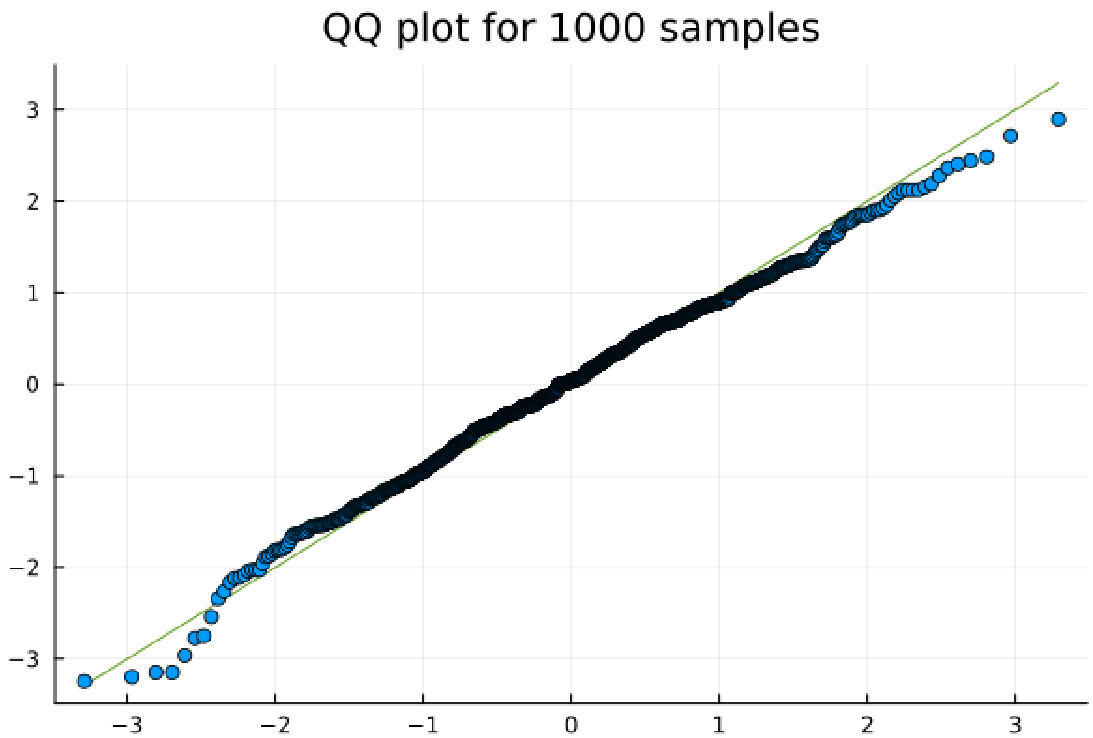
[ ]:
```
new_samples = metropolis_hastings(1000000, Normal(1, 2), 0.0, 1.0)
```

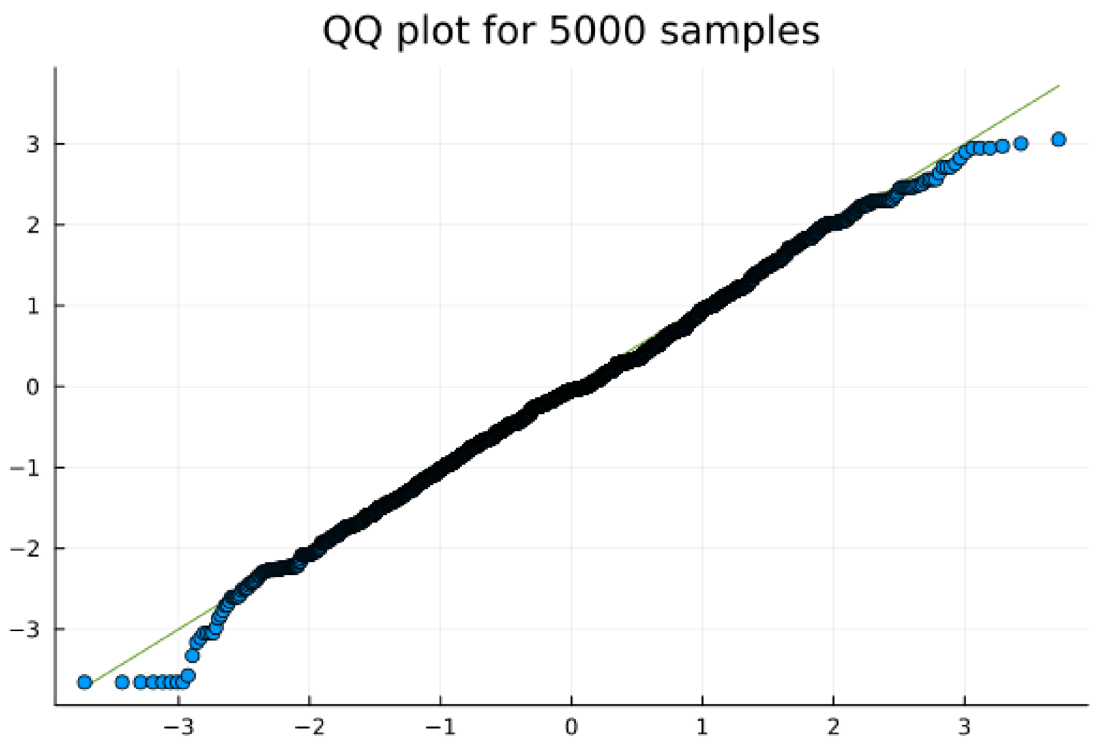
```
1000000-element Vector{Float64}:
  0.0
  1.3398092759012632
  1.1894000985767126
  1.3001204626839074
 -0.5781329214849151
 -0.5702501368014415
 -0.5600397725807035
 -0.17314332965298074
 -0.17314332965298074
 -0.9921124872005007

  0.5421147263083261
  0.5421147263083261
  0.5421147263083261
  0.5421147263083261
 -1.6123943948972208
 -1.6123943948972208
 -2.2443627912739212
  4.906671179041968
  4.906671179041968
```

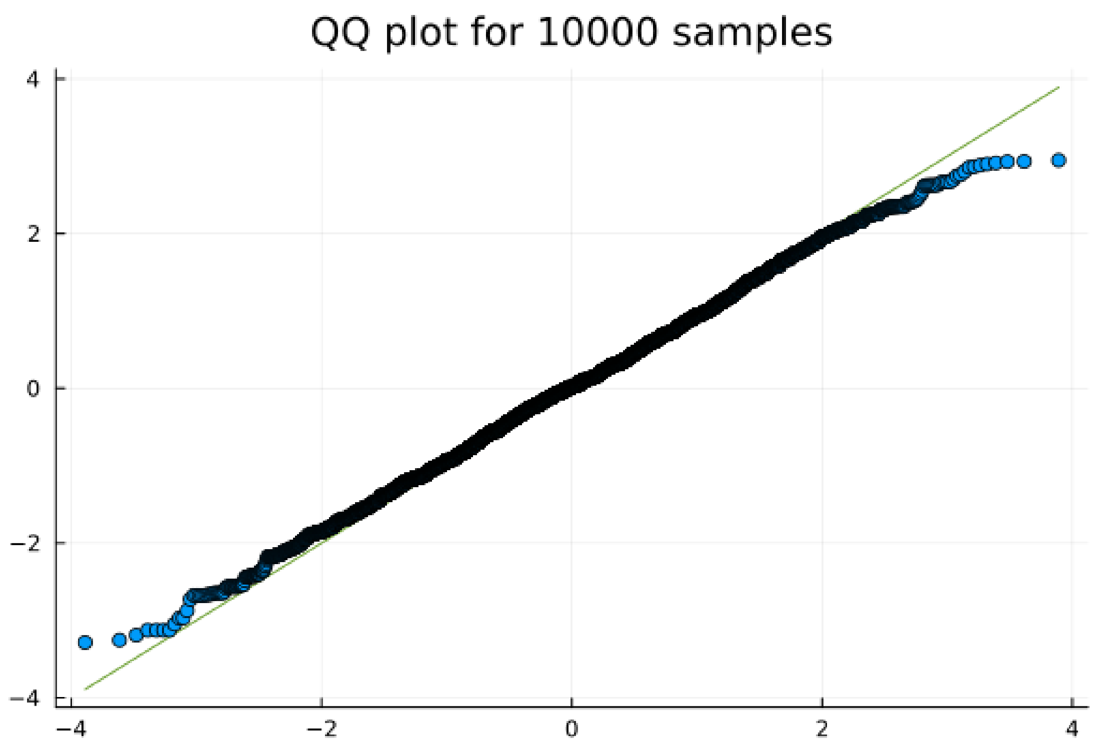
A narrative proof of the algorithm's performance is that when I tried to run the code at the beginning of this Jupyter notebook with 1,000,000 iterations, the kernel crashed. So, I had to run the code with 100,000 iterations. This is much better.

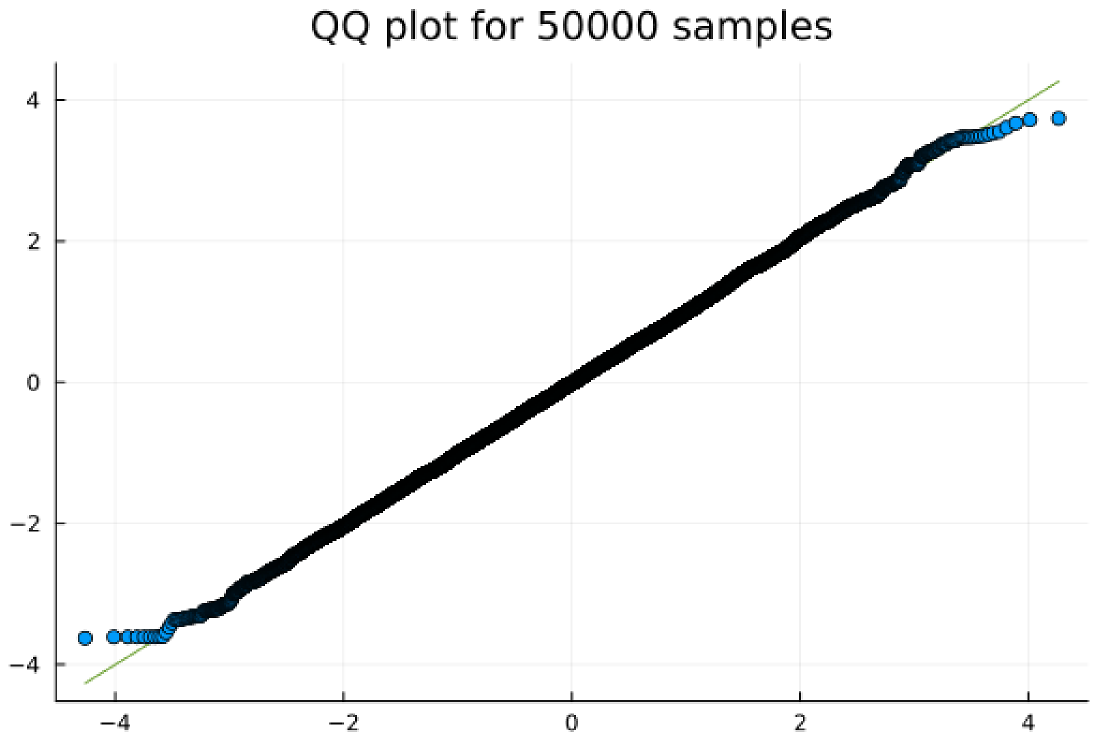
[ ]:
```
# Define the sample sizes
sample_sizes = [1000, 5000, 10000, 50000, 100000]

# Define the target distribution

target = Normal(0, 1)

# Define the initial value and the scale of the proposal distribution

init = 0.0
```

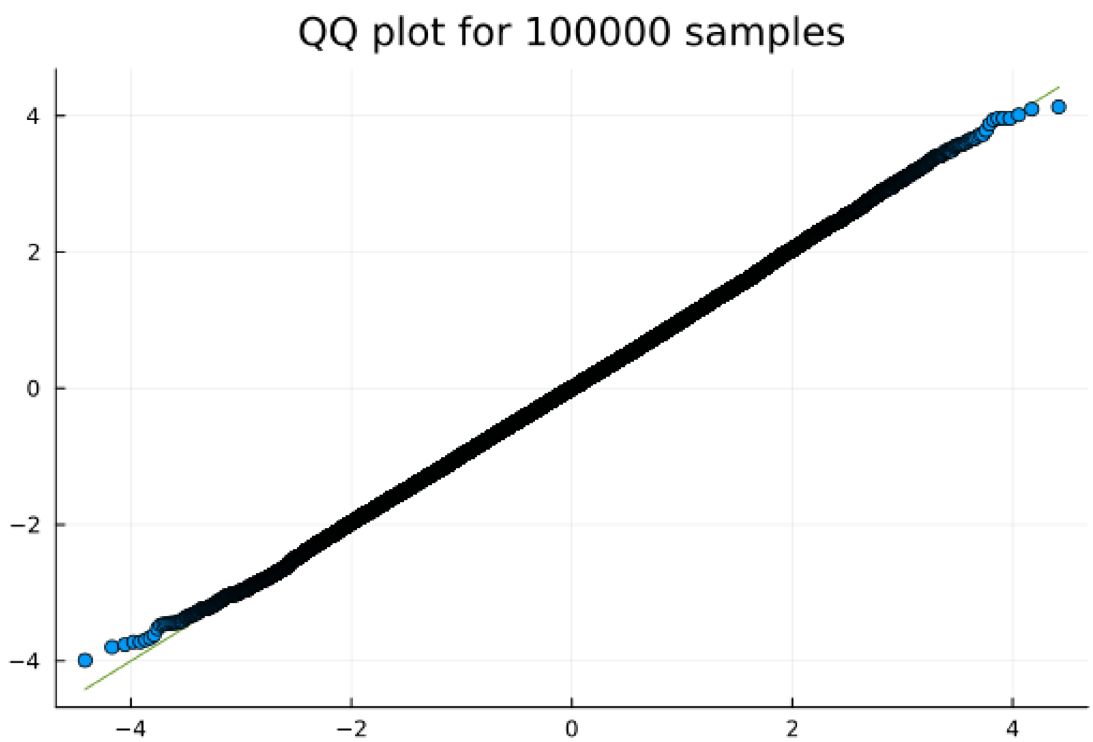
```julia
scale = 1.0


# Initialize an empty Dict to store the DataFrames
dfs = Dict()

# Run the Metropolis-Hastings function for each sample size and store the
 ↪samples in a separate DataFrame
for n_samples in sample_sizes
    samples = metropolis_hastings(n_samples, target, init, scale)
    dfs["samples_$n_samples"] = DataFrame(samples = samples)
end

# Run the Metropolis-Hastings function for each sample size and create a QQ plot
for n_samples in sample_sizes
    samples = metropolis_hastings(n_samples, target, init, scale)
    p = qqplot(Normal(0,1), samples, title = "QQ plot for $n_samples samples",
 ↪legend = false)
    display(p)
end
```




QQ plot for 5000 samples


QQ plot for 10000 samples


QQ plot for 50000 samples
4
2
0
−2
−4
−4
−2
0
2
4


QQ plot for 100000 samples
4
2
0
−2
−4
−4
−2
0
2
4

This is a much better result than the previous one. In terms of the Smirnov-Kolmogorov test, I perform the analysis below.

[ ]:
```
# Perform the Smirnov-Kolmogorov test for each sample size
for n_samples in sample_sizes
    samples = dfs["samples_$n_samples"].samples
    ks_test = ExactOneSampleKSTest(samples, target)
    p_value = pvalue(ks_test)
    println("Sample size: $n_samples, p-value: $p_value")
end
```

```
Sample size: 1000, p-value: 0.15464286515578862
Sample size: 5000, p-value: 0.0023610555858638396
Sample size: 10000, p-value: 1.458157170760516e-17
Sample size: 50000, p-value: 5.149515593524394e-9
Sample size: 100000, p-value: 0.017501710791874975
```

```
 Warning: This test is inaccurate with ties
 @ HypothesisTests
C:\Users\user\.julia\packages\HypothesisTests\r322N\src\kolmogorov_smirnov.jl:68
 Warning: This test is inaccurate with ties
 @ HypothesisTests
C:\Users\user\.julia\packages\HypothesisTests\r322N\src\kolmogorov_smirnov.jl:68
 Warning: This test is inaccurate with ties
 @ HypothesisTests
C:\Users\user\.julia\packages\HypothesisTests\r322N\src\kolmogorov_smirnov.jl:68
 Warning: This test is inaccurate with ties
 @ HypothesisTests
C:\Users\user\.julia\packages\HypothesisTests\r322N\src\kolmogorov_smirnov.jl:68
 Warning: This test is inaccurate with ties
 @ HypothesisTests
C:\Users\user\.julia\packages\HypothesisTests\r322N\src\kolmogorov_smirnov.jl:68
```

There are now cases where I can definitely not reject null, that is, my approximation may allow to say the data is normally distributed with mean 0 and variance 1.

2 Part II: Expanding the ELVIS script

Please see the `problem_set_2_part_2_elvis.jl` file for the expanded ELVIS script.