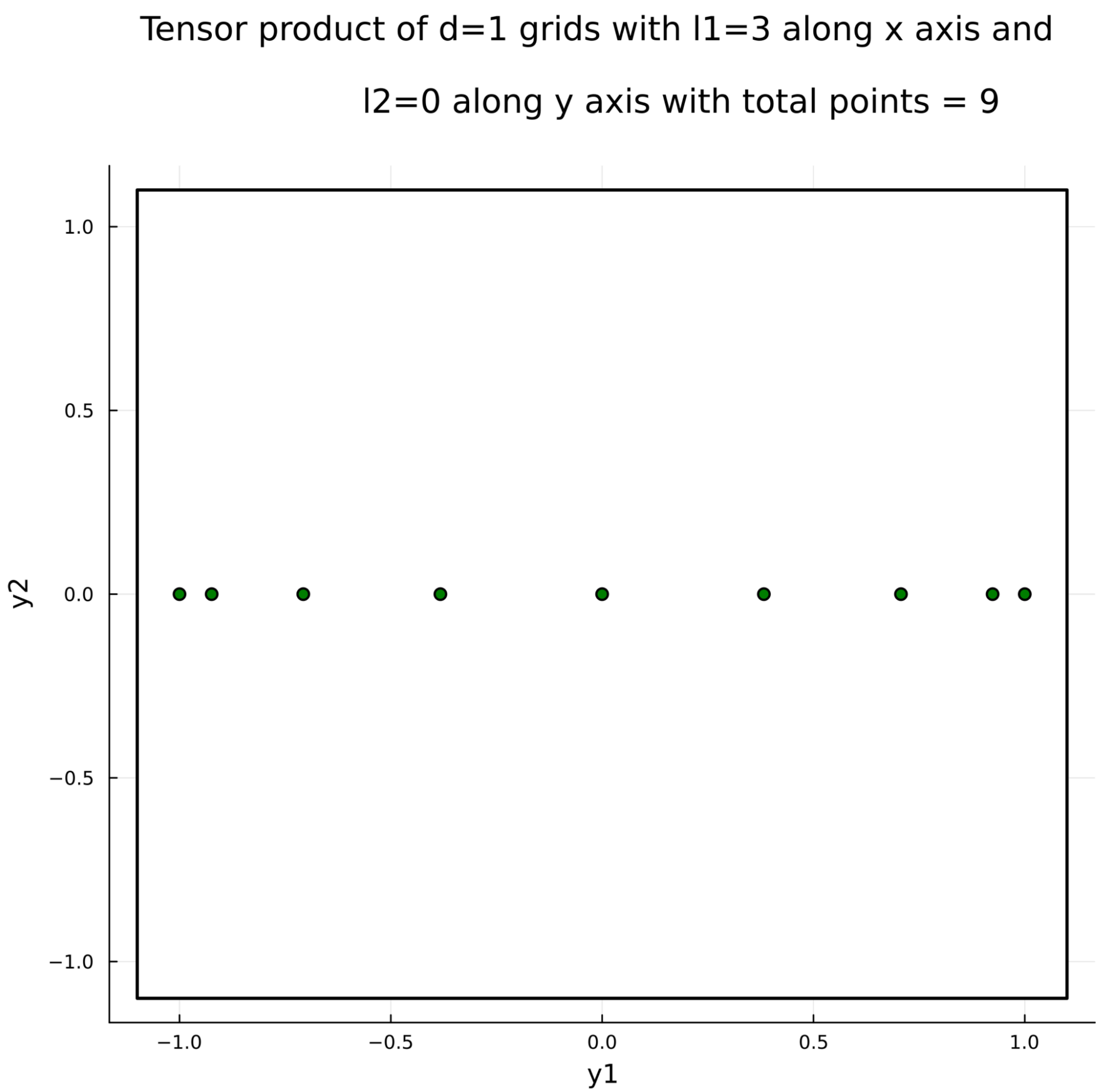
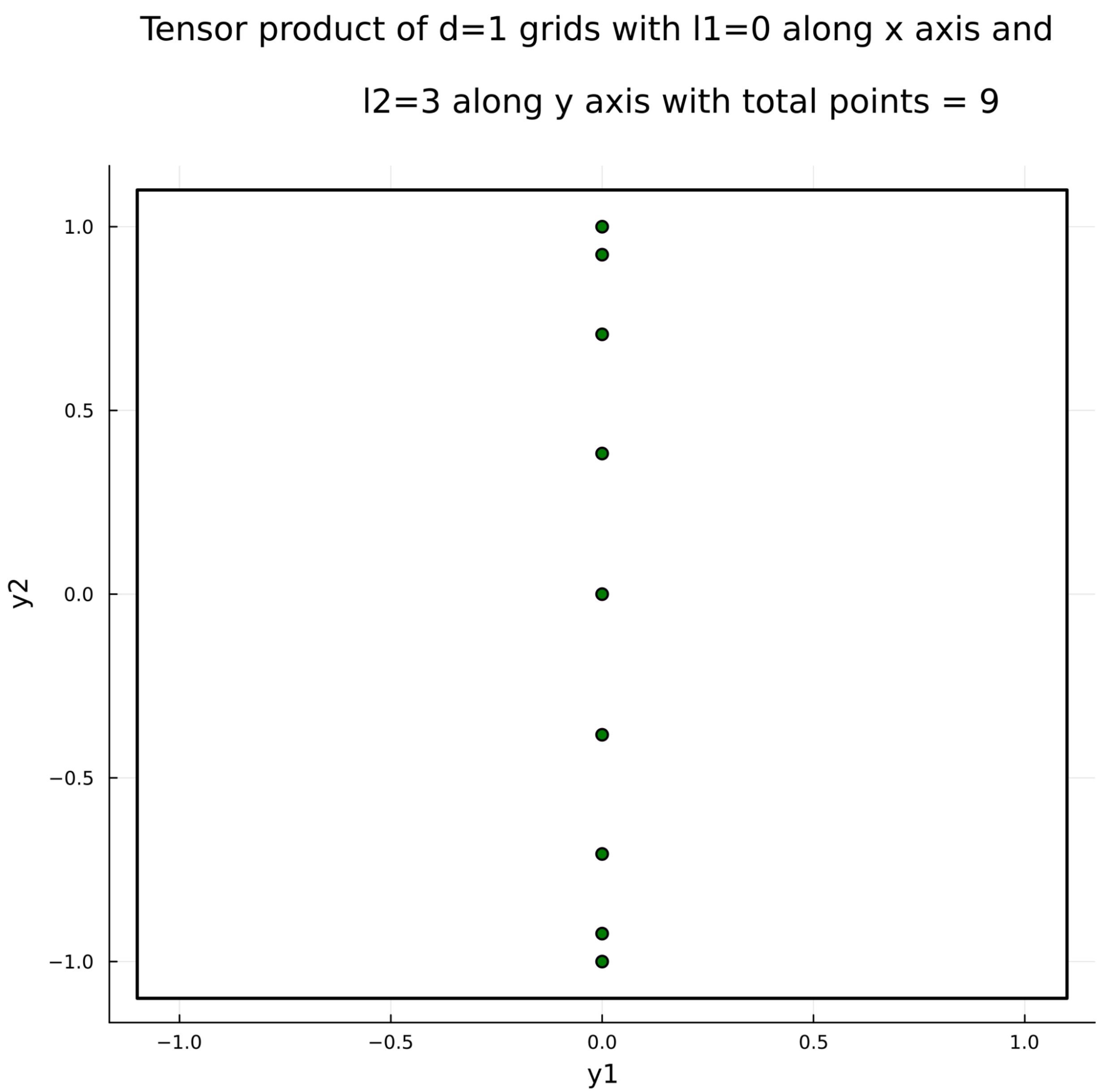
Problem 1

Part 1

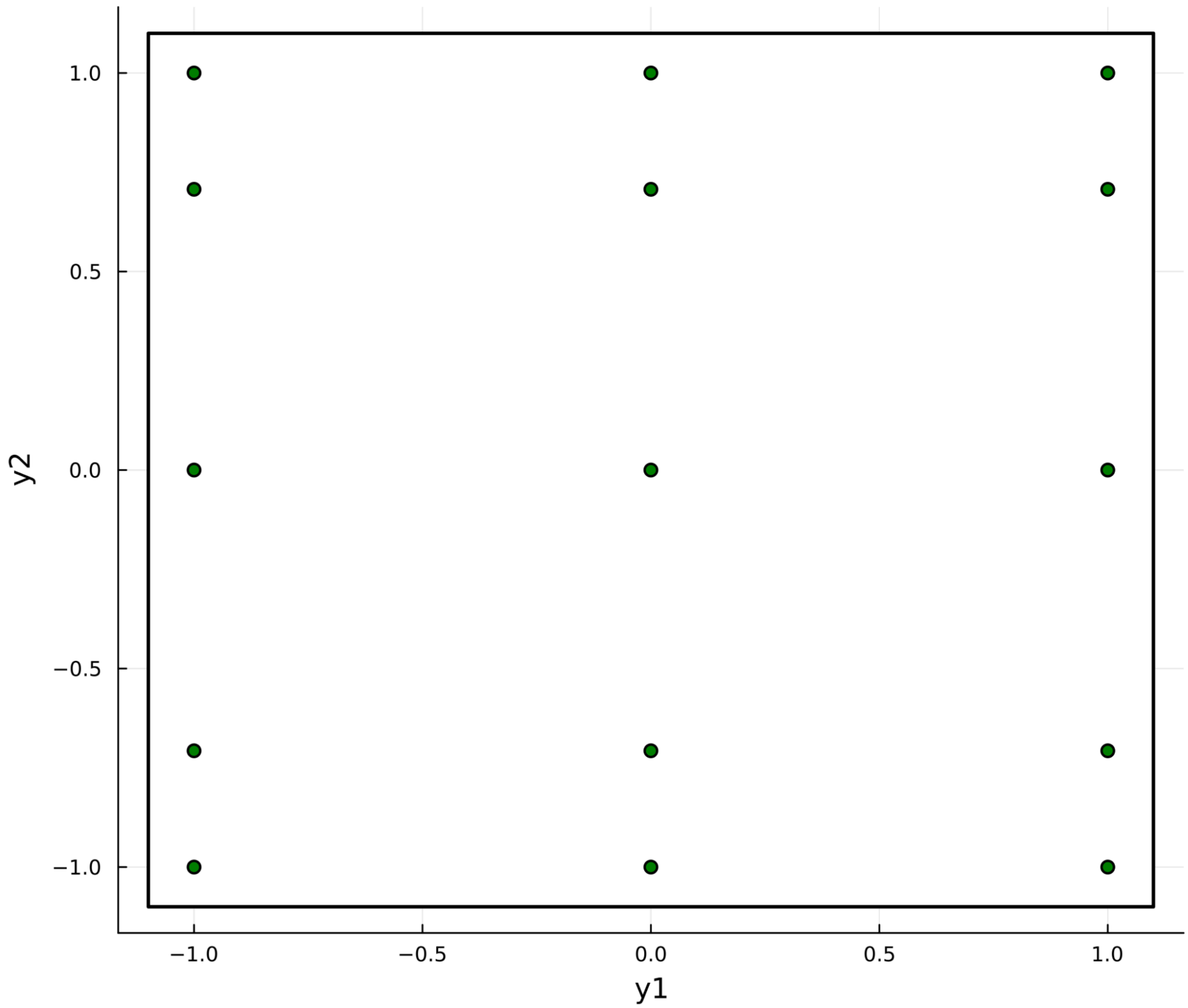
- It is given that q is 4 and d is 2. Therefore, possible values of ‘l’ used in generating the given plot are 3 and 4.
    - With l = 3, possible pairs of (l1,l2) are (3,0); (0,3); (1,2); (2,1)
    - With l = 4, possible pairs of (l1,l2) are (4,0); (0,4); (1,3); (3,1); (2,2)
    - So, a total of 9 pairs of (l1,l2) values were used to generate the given plot.

- The tensor-product grid of all possible (l1,l2) pairs are attached below.

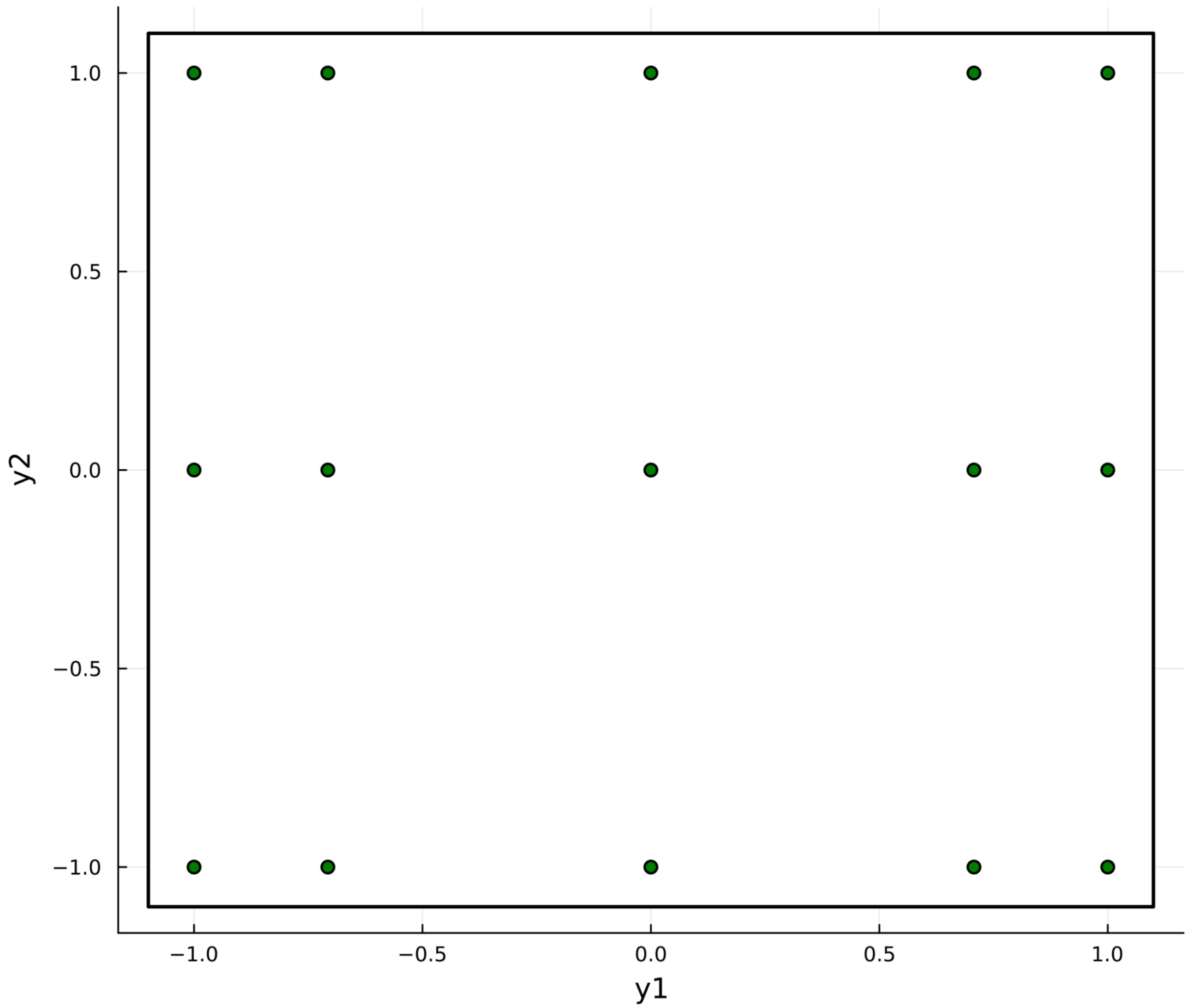

Tensor product of d=1 grids with l1=3 along x axis and
l2=0 along y axis with total points = 9
y2
y1
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

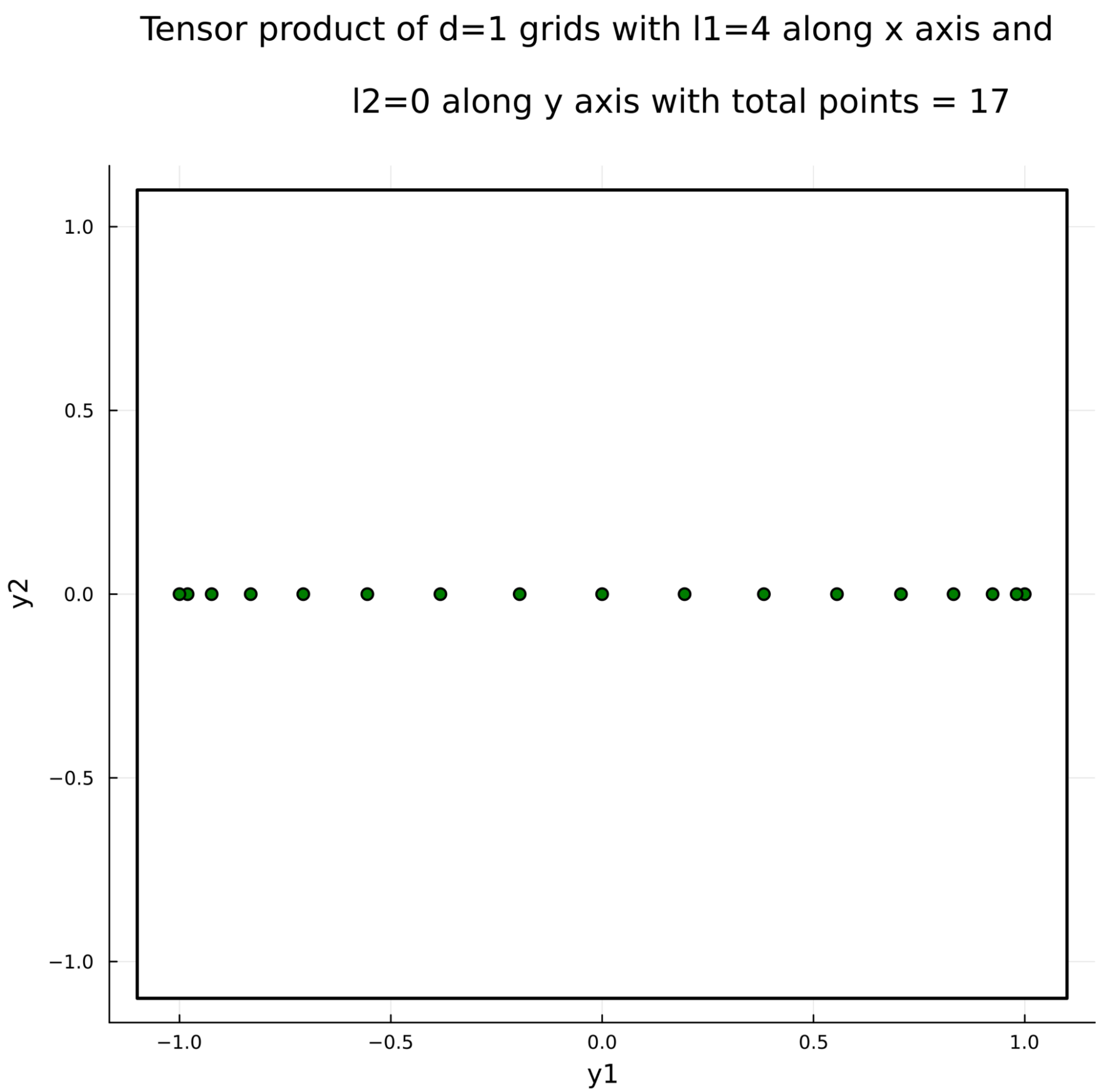

Tensor product of d=1 grids with l1=0 along x axis and
l2=3 along y axis with total points = 9
1.0
0.5
0.0
−0.5
−1.0
y2
−1.0
−0.5
0.0
0.5
1.0
y1

Tensor product of d=1 grids with l1=1 along x axis and l2=2 along y axis with total points = 15

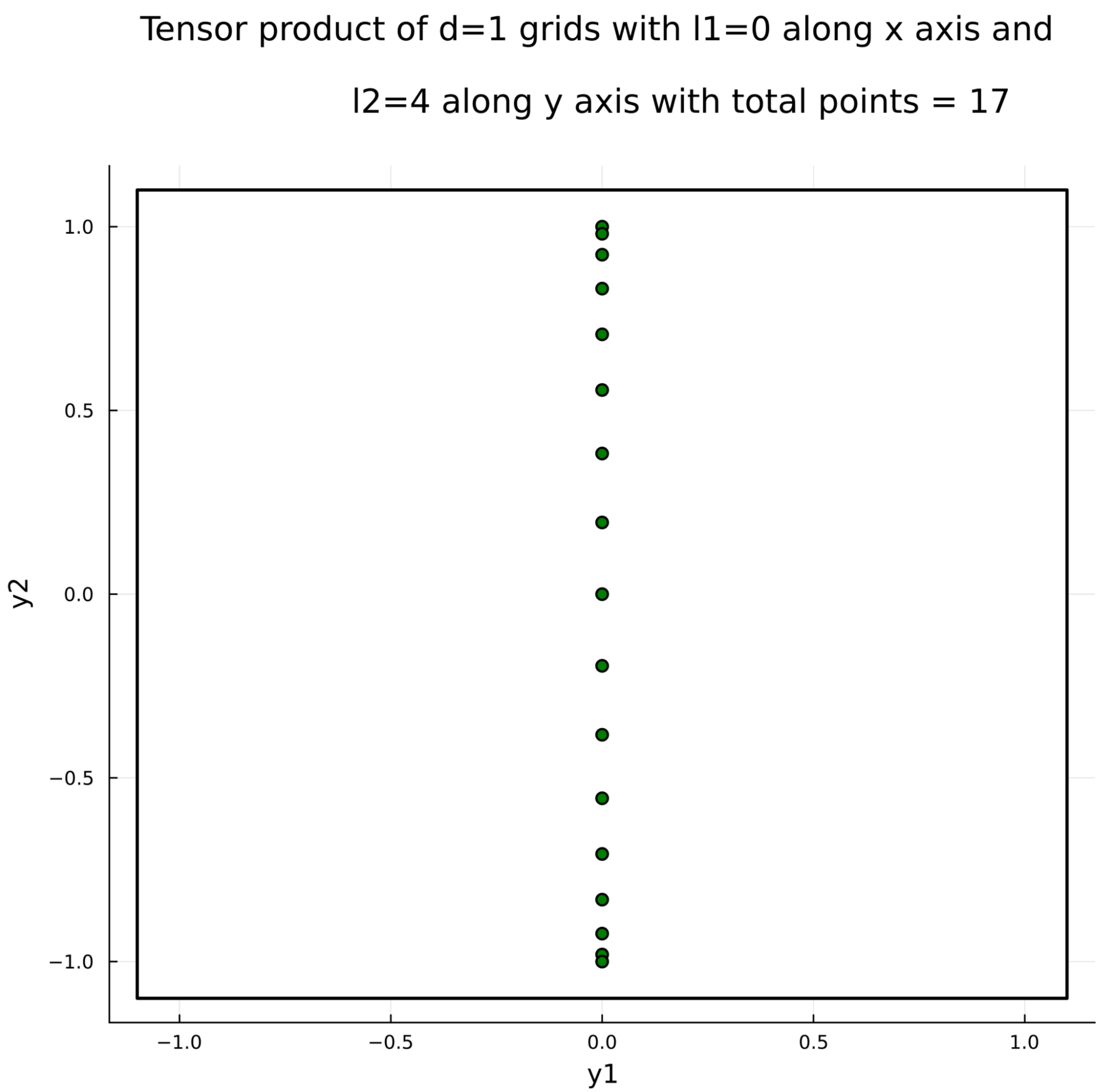


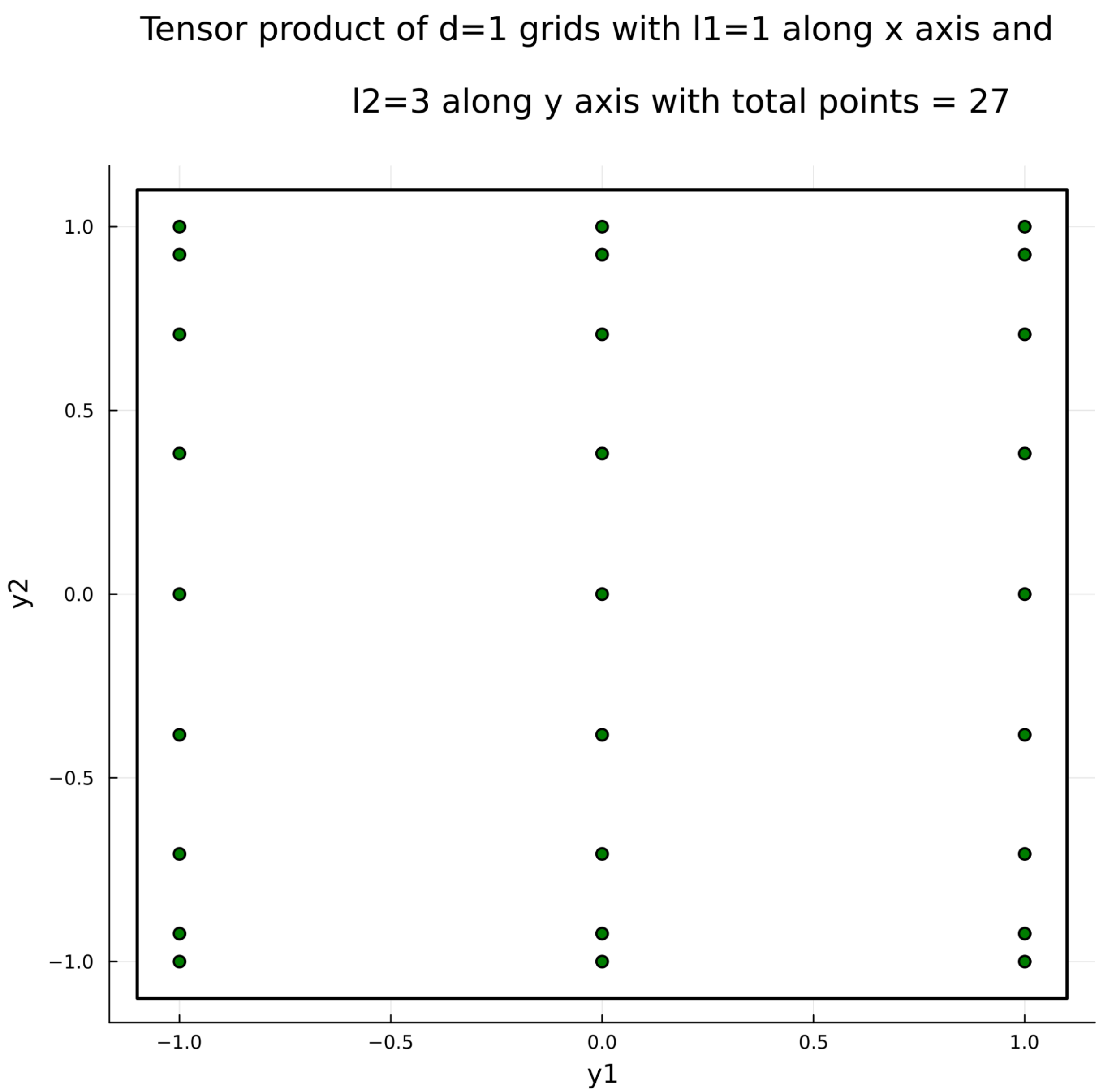
Tensor product of d=1 grids with l1=2 along x axis and l2=1 along y axis with total points = 15

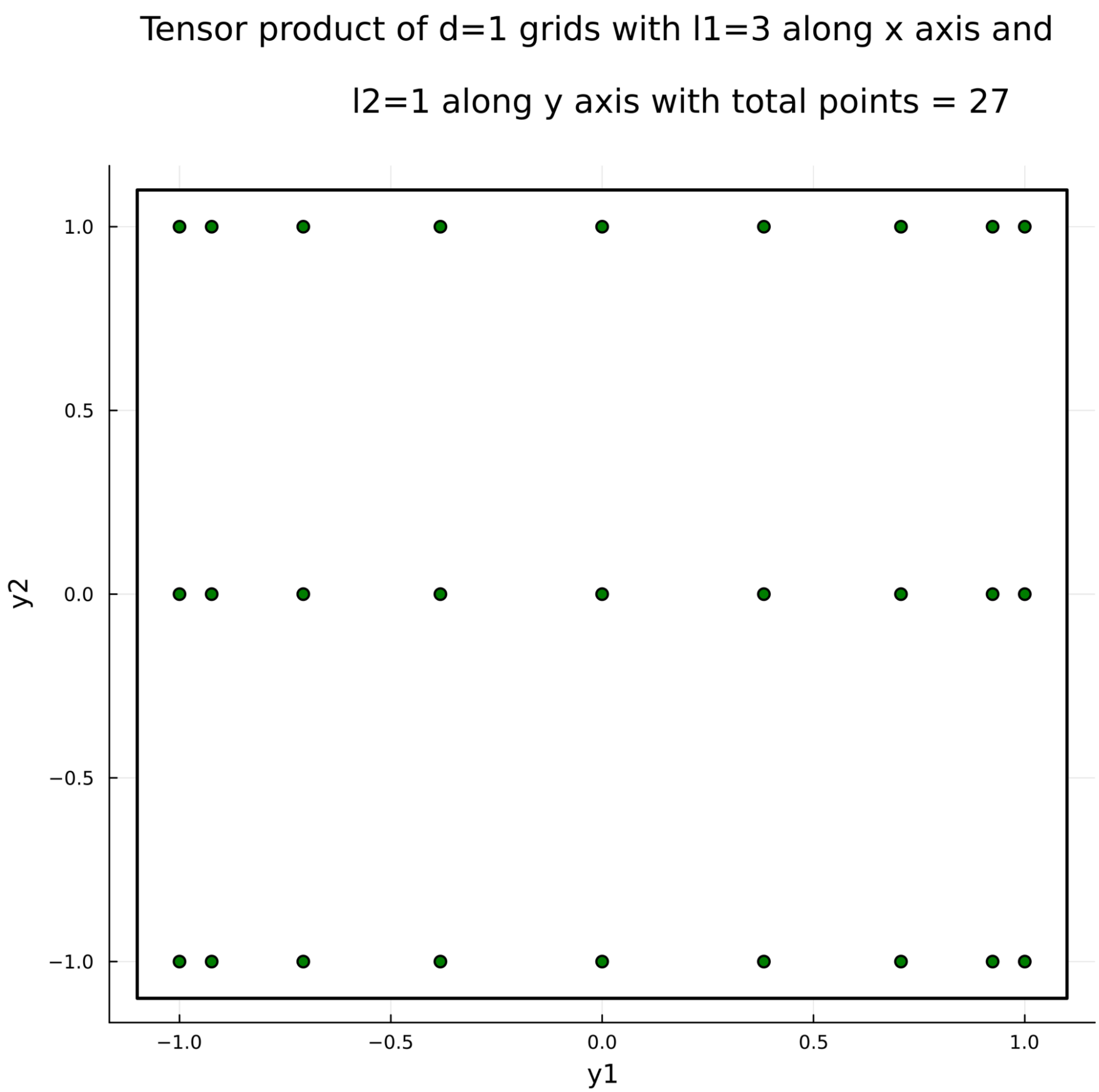


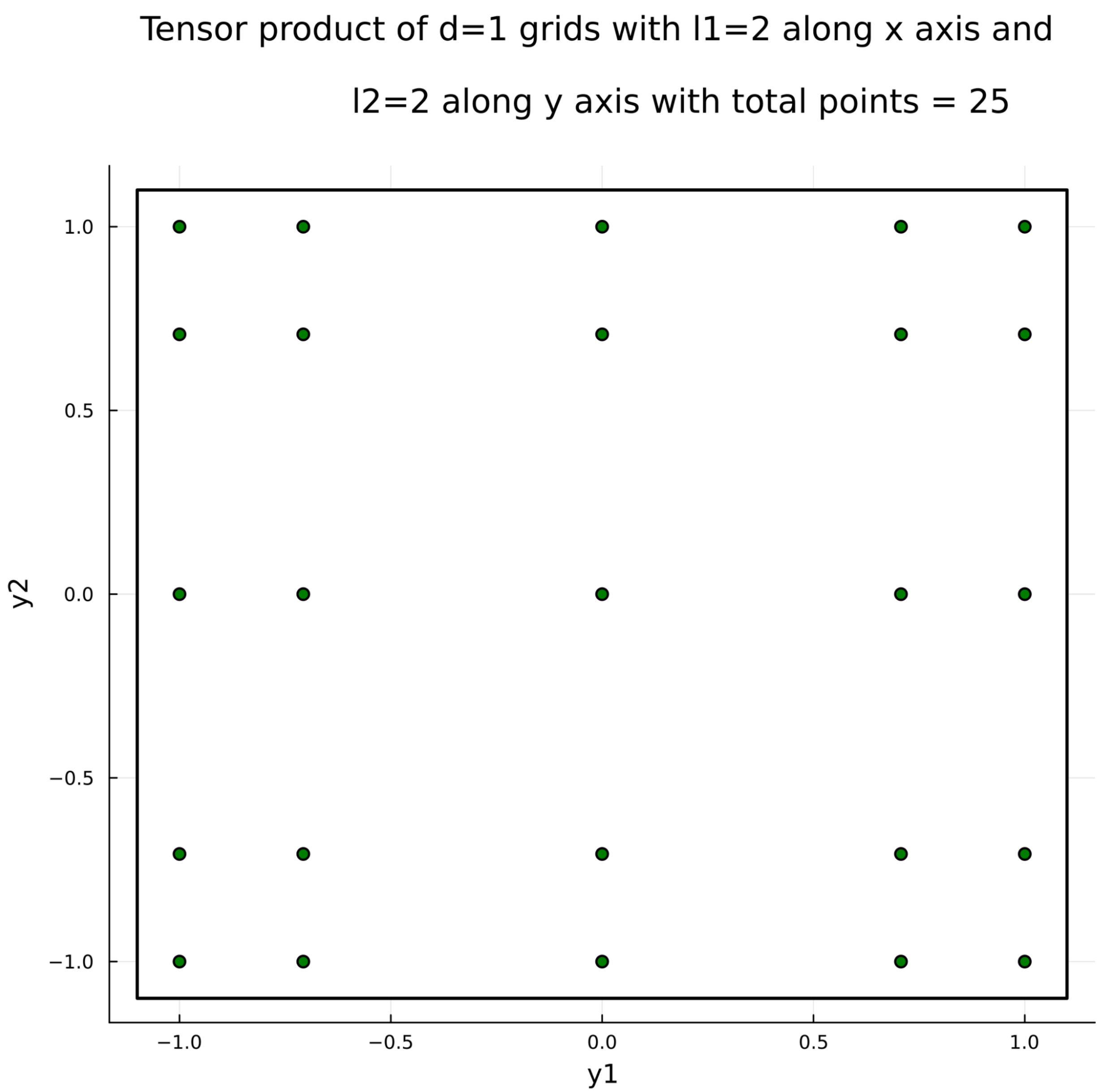


Tensor product of d=1 grids with l1=4 along x axis and
l2=0 along y axis with total points = 17
1.0
0.5
0.0
−0.5
−1.0
y2
−1.0
−0.5
0.0
0.5
1.0
y1


Tensor product of d=1 grids with l1=0 along x axis and
l2=4 along y axis with total points = 17
y2
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
y1


Tensor product of d=1 grids with l1=1 along x axis and
l2=3 along y axis with total points = 27
y2
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
y1


Tensor product of d=1 grids with l1=3 along x axis and
l2=1 along y axis with total points = 27
1.0
0.5
0.0
−0.5
−1.0
y2
−1.0
−0.5
0.0
0.5
1.0
y1


Tensor product of d=1 grids with l1=2 along x axis and
l2=2 along y axis with total points = 25
1.0
0.5
0.0
−0.5
−1.0
y2
−1.0
−0.5
0.0
0.5
1.0
y1

Part 2

From the class discussion, we know that rule $A(d,q)$ with CC and level parameter $q=d+p$ is exact for interpolating polynomials of total order p.

Since for the given plot, d is 2 and q is 4, we can infer that p is 2.

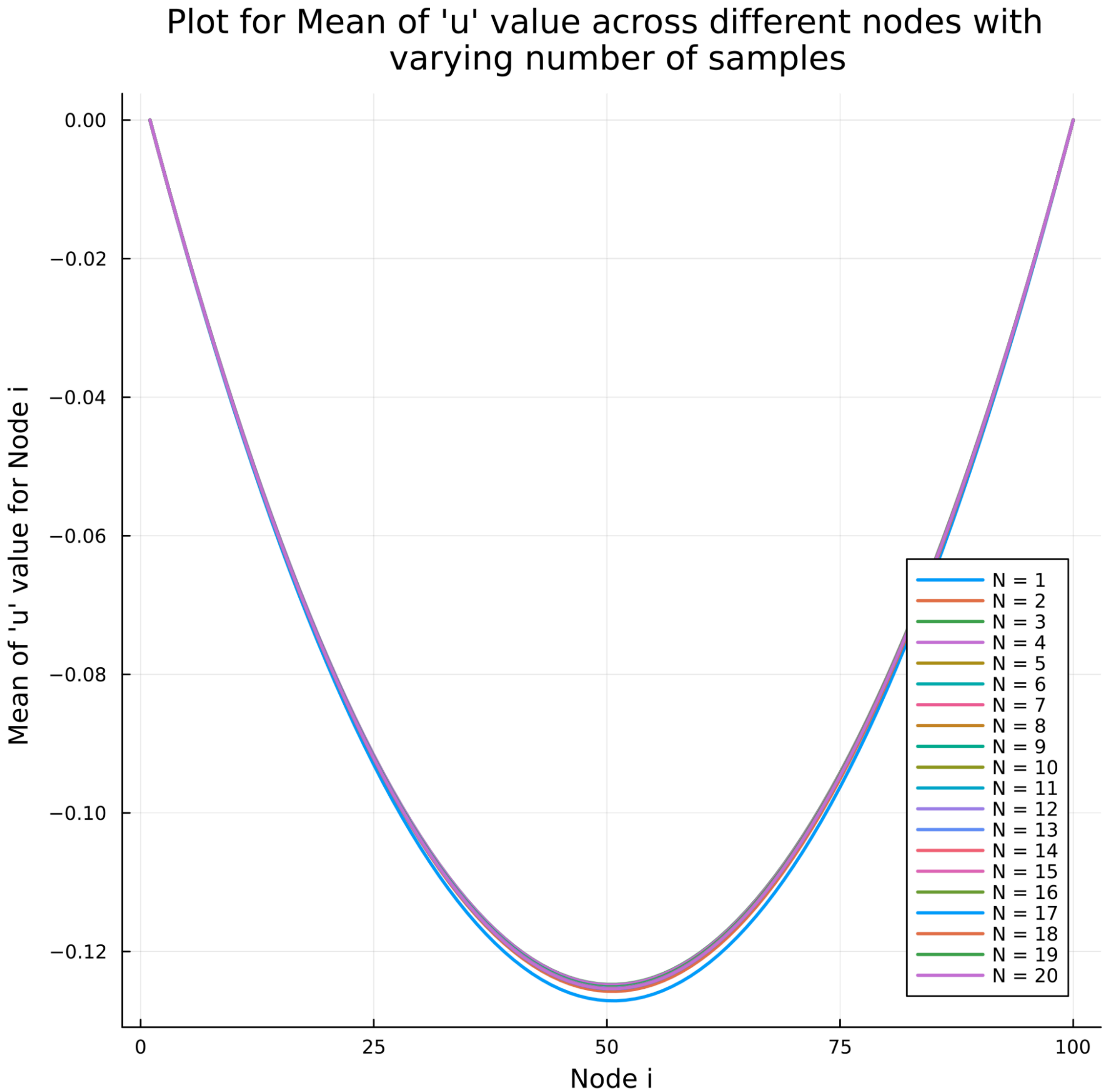
Therefore, the given grid is accurate for interpolating polynomials of total order 2.

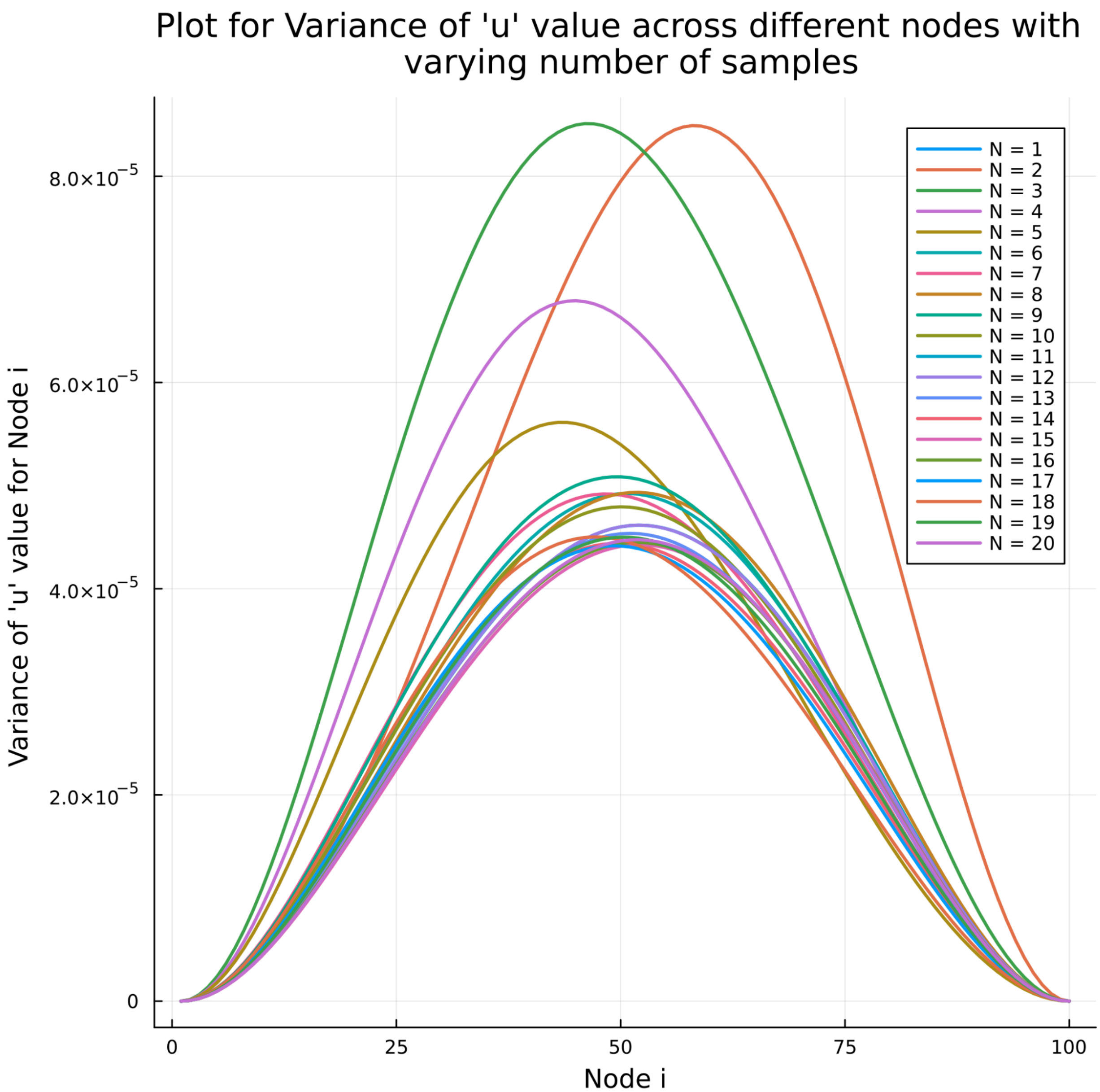
Problem 2

Part 1

In my implementation, I follow the example discussed in the class. To sample N points, I make N partitions in all the d dimensions and get d sequences of N numbers. I then use random permutations of points from those d sequences to sample N d-dimensional points. For the given problem, d=2.

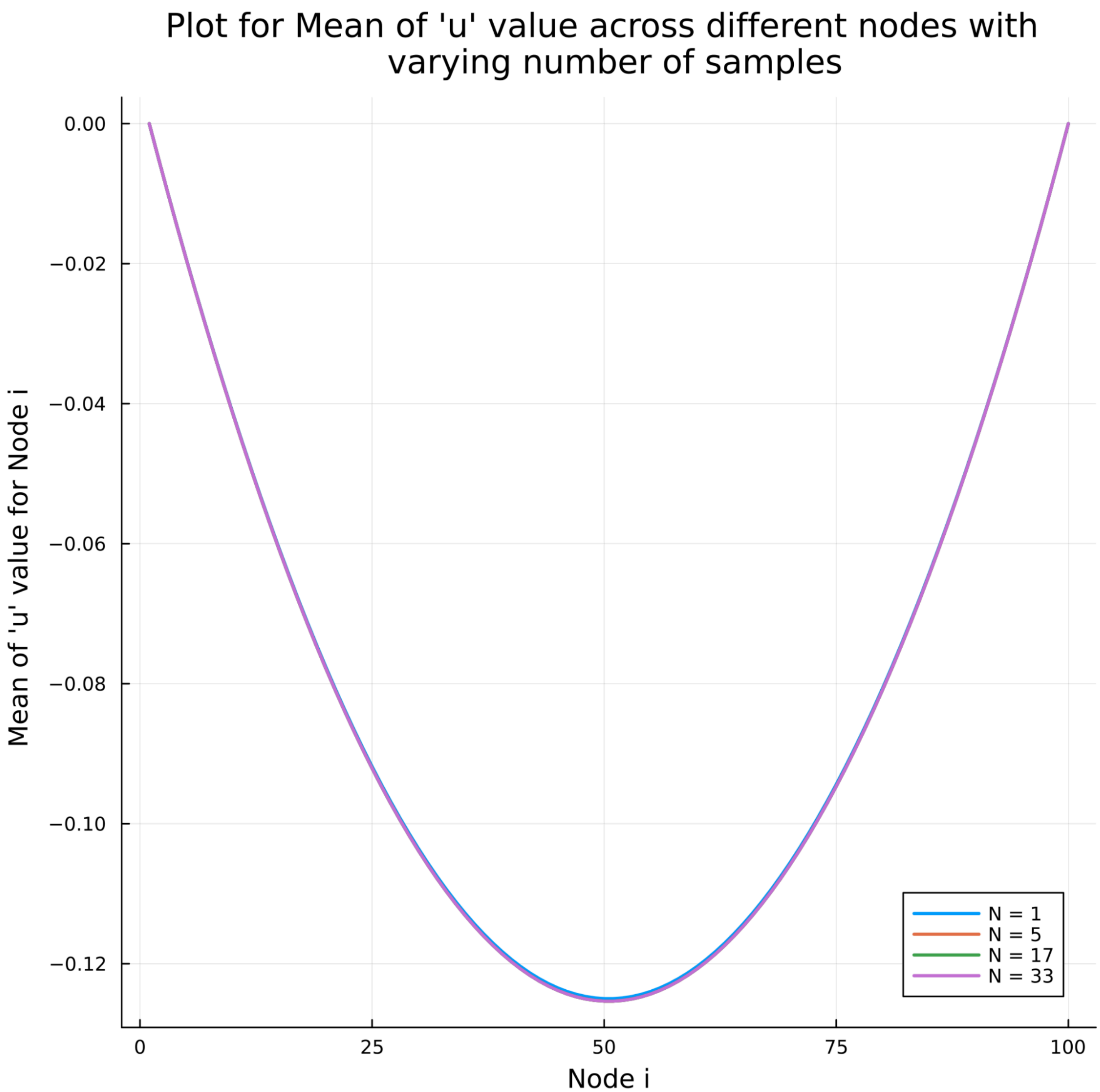
The plots of mean and variance are attached below. We observe a very quick convergence for the mean estimate for the given problem with the number of samples. However, variance takes a slightly larger number of samples to converge.

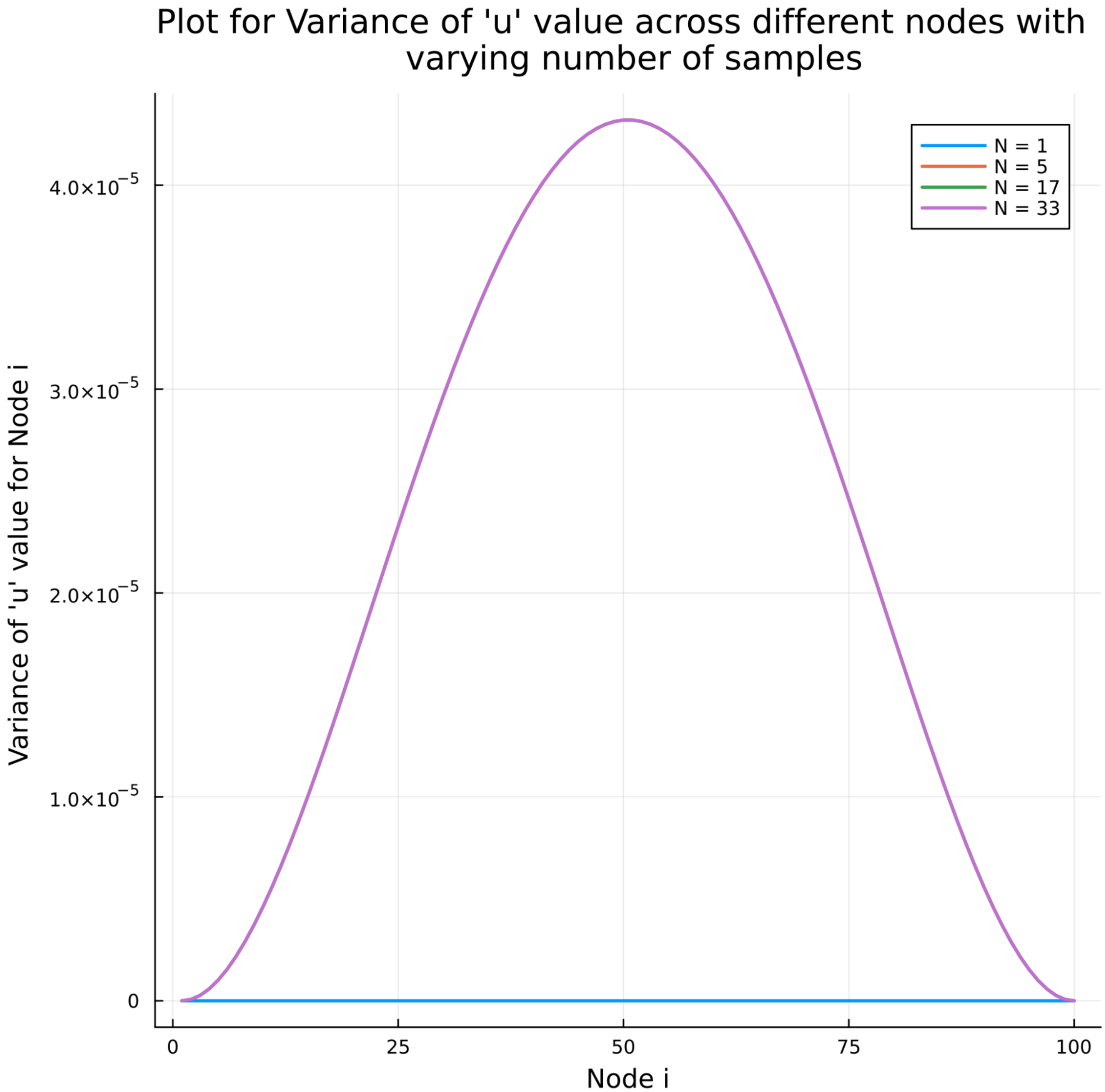

Plot for Mean of 'u' value across different nodes with varying number of samples
Mean of 'u' value for Node i
Node i
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
0
25
50
75
100
N = 1
N = 2
N = 3
N = 4
N = 5
N = 6
N = 7
N = 8
N = 9
N = 10
N = 11
N = 12
N = 13
N = 14
N = 15
N = 16
N = 17
N = 18
N = 19
N = 20


Plot for Variance of 'u' value across different nodes with varying number of samples
Variance of 'u' value for Node i
Node i
8.0×10^-5
6.0×10^-5
4.0×10^-5
2.0×10^-5
0
0
25
50
75
100
N = 1
N = 2
N = 3
N = 4
N = 5
N = 6
N = 7
N = 8
N = 9
N = 10
N = 11
N = 12
N = 13
N = 14
N = 15
N = 16
N = 17
N = 18
N = 19
N = 20

<u>Part 2</u>

The plots of mean and variance using the tensor product grid are attached below. Unlike the output of Part 1, we observe a very quick convergence for both the mean and the variance estimate for the given problem with the number of samples.

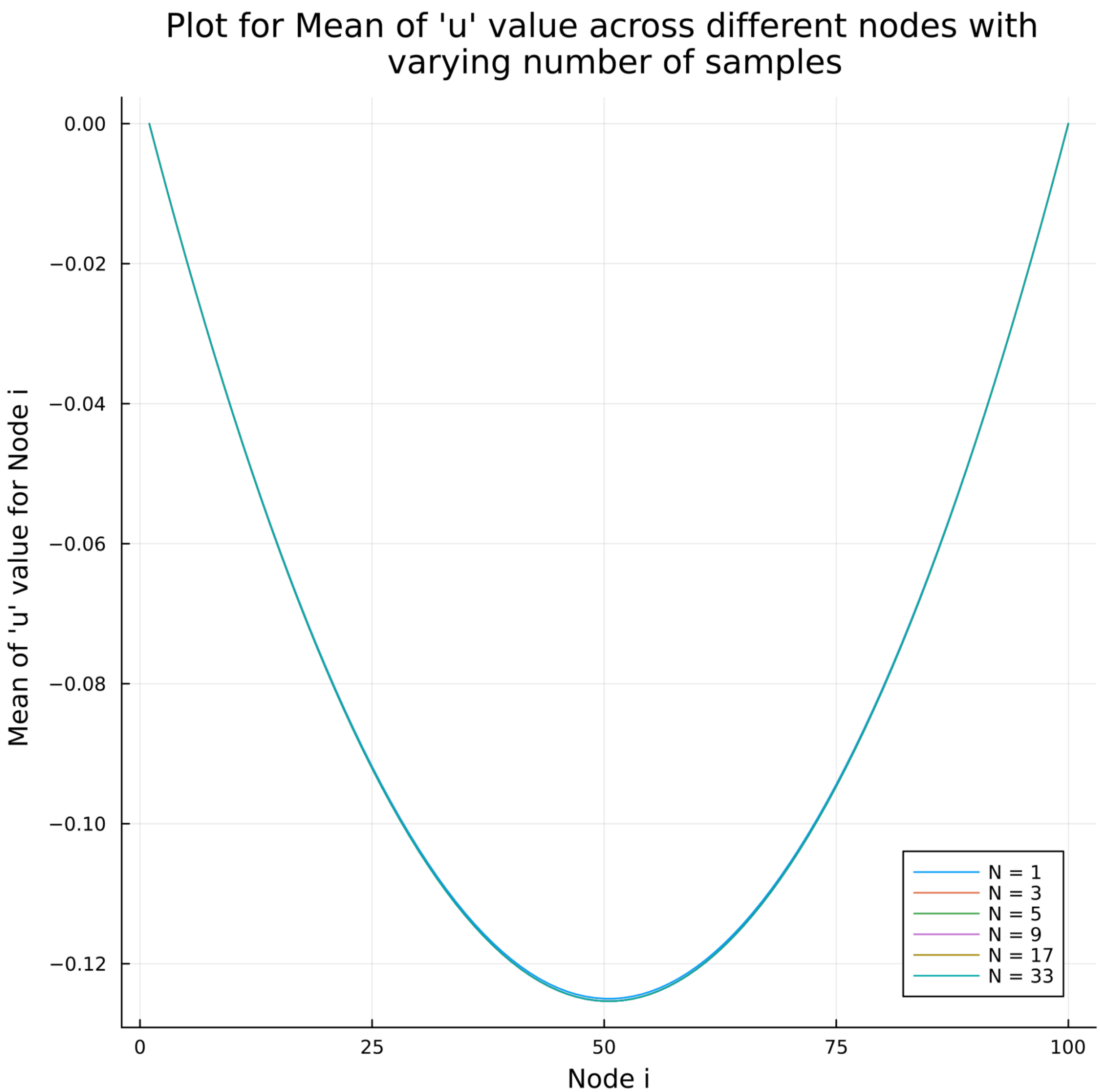

Plot for Mean of 'u' value across different nodes with varying number of samples
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
Mean of 'u' value for Node i
0
25
50
75
100
Node i
N = 1
N = 5
N = 17
N = 33

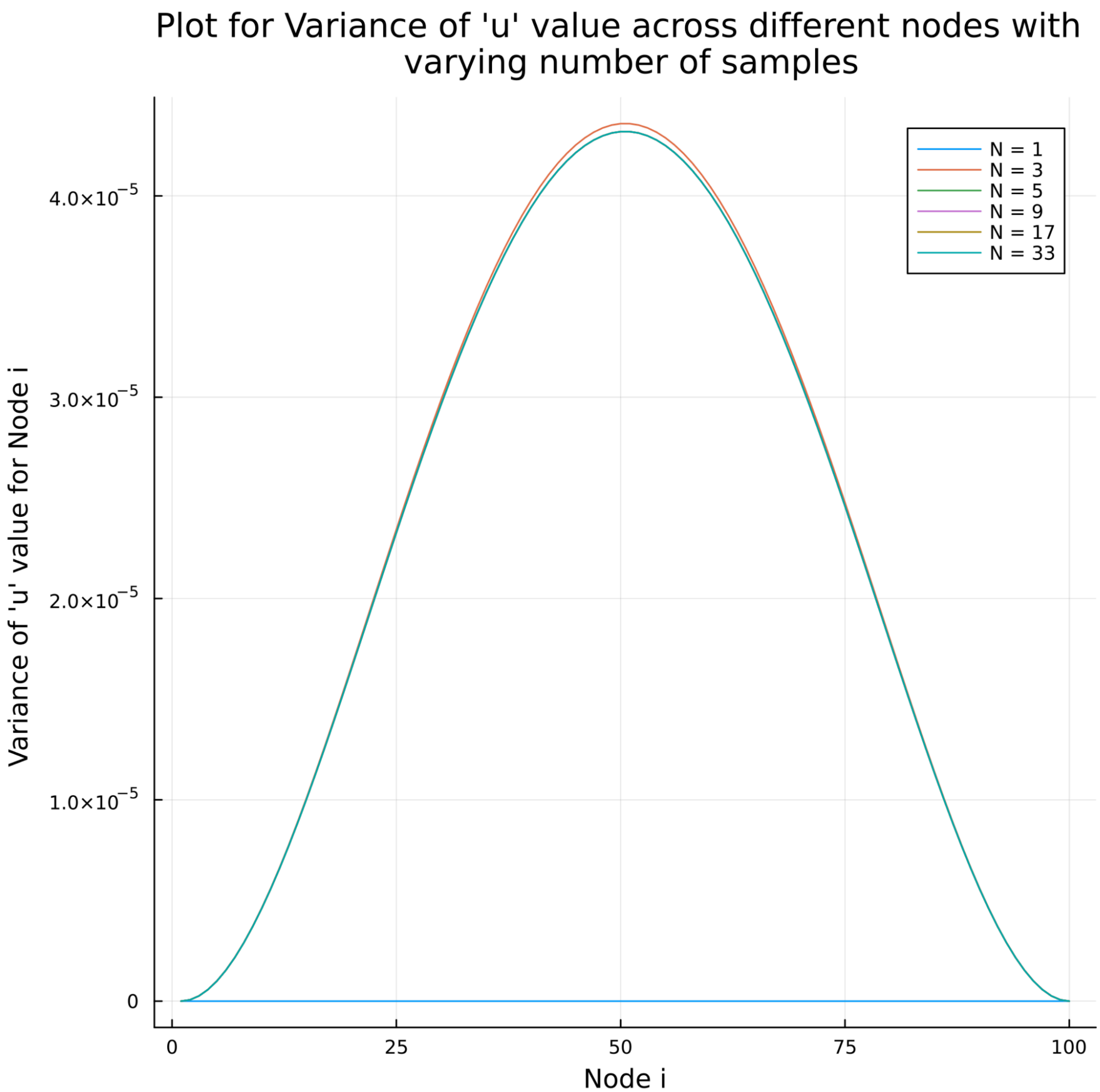


Plot for Variance of 'u' value across different nodes with varying number of samples
N = 1
N = 5
N = 17
N = 33
4.0×10⁻⁵
3.0×10⁻⁵
2.0×10⁻⁵
1.0×10⁻⁵
0
Variance of 'u' value for Node i
0
25
50
75
100
Node i

Part 3

The plots of mean and variance using the Smolyak sparse grid with CC abscissas are attached below. Similar to Part 2, we observe a very quick convergence for both the mean and the variance estimate for the given problem with the number of samples.

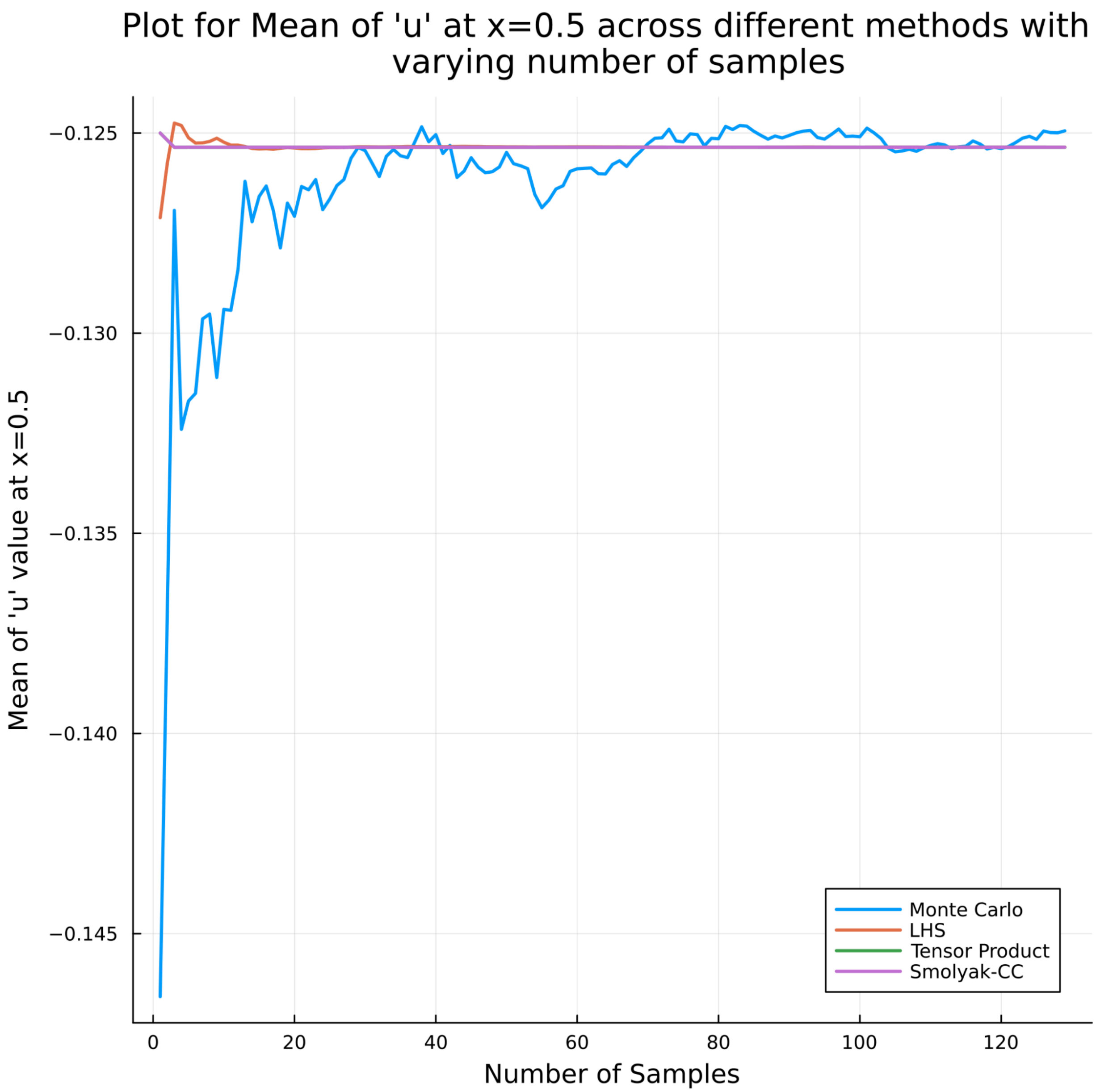

Plot for Mean of 'u' value across different nodes with varying number of samples
Mean of 'u' value for Node i
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
0
25
50
75
100
Node i
N = 1
N = 3
N = 5
N = 9
N = 17
N = 33


Plot for Variance of 'u' value across different nodes with varying number of samples
Variance of 'u' value for Node i
Node i
4.0×10^{-5}
3.0×10^{-5}
2.0×10^{-5}
1.0×10^{-5}
0
0
25
50
75
100
N = 1
N = 3
N = 5
N = 9
N = 17
N = 33

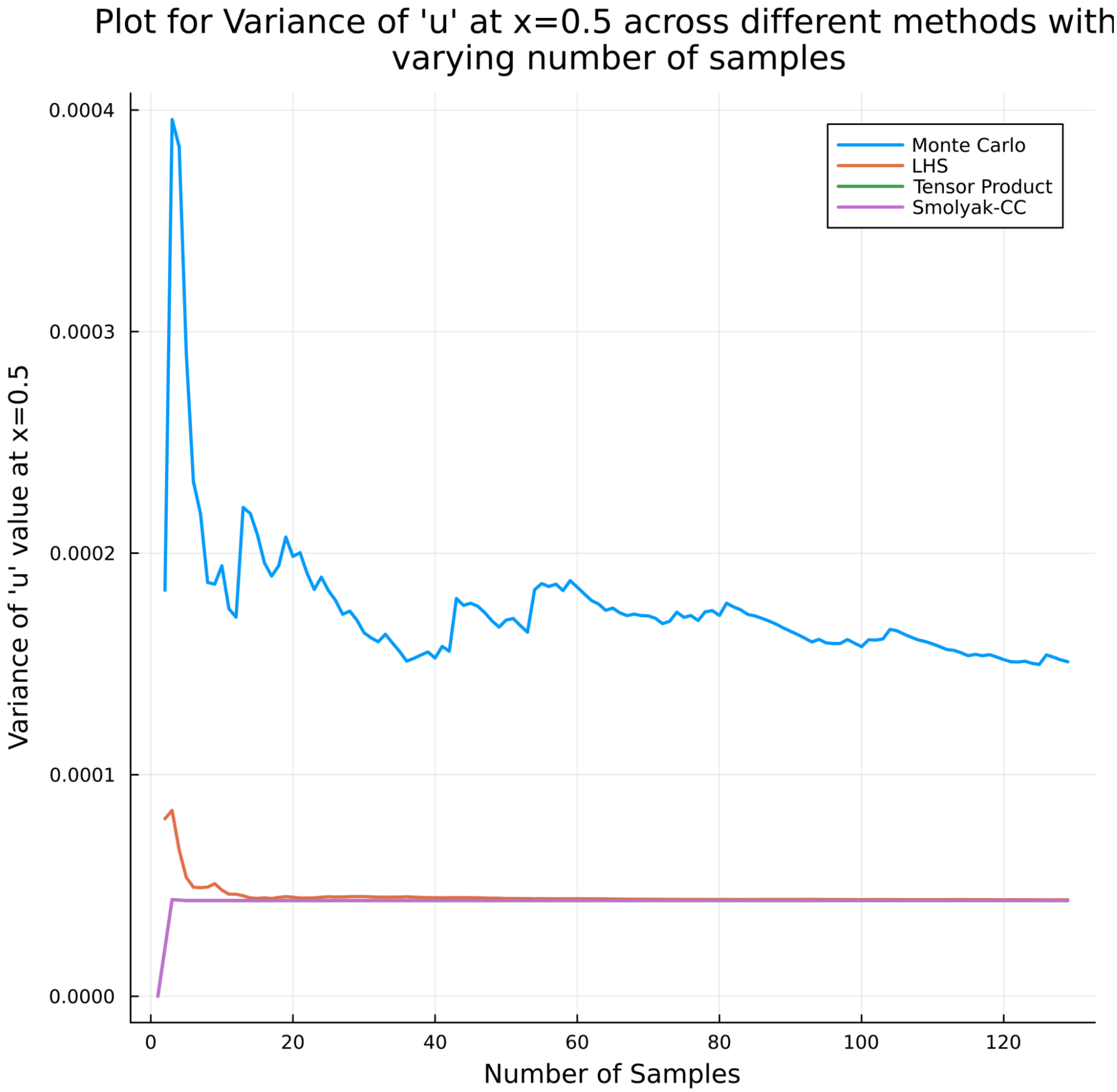
Part 4

The plots of mean and variance of u at x=0.5 using all the different methods are attached below. As expected, we observe that Monte Carlo takes the largest amount of samples to get an accurate estimate, followed by LHS and then by the Tensor product grid and Smolyak grid.

We also observe that the convergence rate of standard Monte Carlo for both mean and variance looks pretty similar to the plot of 1/sqrt(N) which is following the theory discussed in class.



Plot for Mean of 'u' at x=0.5 across different methods with varying number of samples
Mean of 'u' value at x=0.5
Number of Samples
−0.125
−0.130
−0.135
−0.140
−0.145
0
20
40
60
80
100
120
Monte Carlo
LHS
Tensor Product
Smolyak-CC



Plot for Variance of 'u' at x=0.5 across different methods with varying number of samples
Variance of 'u' value at x=0.5
Number of Samples
0.0004
0.0003
0.0002
0.0001
0.0000
0
20
40
60
80
100
120
Monte Carlo
LHS
Tensor Product
Smolyak-CC

Part 5

When the correlation length is decreased, the number of eigenpairs used in computing K(x,w) will increase. That means d will be more than 2. As a result, more number of Y points will need to be sampled. As a result, the number of Smolyak abscissas that will need to be computed will increase, thus making it more difficult computationally to do stochastic collocation for the given problem.

```
#=
By Counting the number of points on each axis, we can tell that there are 17 points.
This means l=4 because the number of points on each axis is 2^l+1.
So, we need to find 17 points in one dimension and take the tensor product of these points.
=#
include("spquad.jl")
using Plots


function get_grid_points(l1,l2)

    l1_one_d_points, l1_one_d_weights = spquad(1,l1)
    N1 = length(l1_one_d_points)  # N = 2^l + 1

    l2_one_d_points, l2_one_d_weights = spquad(1,l2)
    N2 = length(l2_one_d_points)  # N = 2^l + 1

    total_points = N1*N2
    grid_points = Array{NTuple{2,Float64},1}(undef,total_points)
    grid_weights = zeros(total_points)

    # println(N1, " ", N2, " ", total_points)

    index = 1
    for i in 1:N1
        for j in 1:N2
            # index = (i-1)*N1 + j
            # println(index, " ", i, " ", j)
            grid_points[index] = (l1_one_d_points[i],l2_one_d_points[j])
            grid_weights[index] = l1_one_d_weights[i]*l2_one_d_weights[j]
            index += 1
        end
    end

    return grid_points, grid_weights
end


function Q1_part1()

    q = 4
    d = 2

    # Possible values for l are 3 and 4
    possible_l1_l2_pairs_where_l_is_3 = SVector((3,0),(0,3),(1,2),(2,1))
    possible_l1_l2_pairs_where_l_is_4 = SVector((4,0),(0,4),(1,3),(3,1),(2,2))

    points_when_l_is_3 = Dict()
    for ele in possible_l1_l2_pairs_where_l_is_3
        l1, l2 = ele
        grid_points, grid_weights = get_grid_points(l1,l2)
        points_when_l_is_3[ele] = grid_points
    end
    points_when_l_is_4 = Dict()
    for ele in possible_l1_l2_pairs_where_l_is_4
        l1, l2 = ele
        grid_points, grid_weights = get_grid_points(l1,l2)
        points_when_l_is_4[ele] = grid_points
    end

    max_val = 1.1
    x_boundary = SVector{4,Float64}(-max_val,max_val,max_val,-max_val)
    y_boundary = SVector{4,Float64}(-max_val,-max_val,max_val,max_val)
    boundary = Shape(x_boundary, y_boundary)

    # Plot the points
    for ele in possible_l1_l2_pairs_where_l_is_3
        l1, l2 = ele
        grid_points = points_when_l_is_3[ele]
        total_points = length(grid_points)
        snapshot = plot(
            size=(700,700),
            dpi=300,
            xticks=-1.0:0.5:1.0,
            yticks=-1.0:0.5:1.0,
```

```
            xlabel="y1", ylabel="y2",
            title="Tensor product of d=1 grids with l1=$l1 along x axis and \n
                l2=$l2 along y axis with total points = $total_points \n",
            # axis=([], false),
            # legend=:bottom,
            legend=false
            # Distribution
            # Density
            )
        plot!(snapshot,boundary,linewidth=2,color=:white)
        for i in 1:total_points
            point = grid_points[i]
            scatter!(snapshot,point,linewidth=2,color=:green)
        end
        display(snapshot)
        savefig(snapshot, "./HW4/Q1_part1_l1=$l1"*"_l2=$l2.png")
    end

    for ele in possible_l1_l2_pairs_where_l_is_4
        l1, l2 = ele
        grid_points = points_when_l_is_4[ele]
        total_points = length(grid_points)
        snapshot = plot(
            size=(700,700),
            dpi=300,
            xticks=-1.0:0.5:1.0,
            yticks=-1.0:0.5:1.0,
            xlabel="y1", ylabel="y2",
            title="Tensor product of d=1 grids with l1=$l1 along x axis and \n
                l2=$l2 along y axis with total points = $total_points \n",
            # axis=([], false),
            # legend=:bottom,
            legend=false
            # Distribution
            # Density
            )
        plot!(snapshot,boundary,linewidth=2,color=:white)
        for i in 1:total_points
            point = grid_points[i]
            scatter!(snapshot,point,linewidth=2,color=:green)
        end
        display(snapshot)
        savefig(snapshot, "./HW4/Q1_part1_l1=$l1"*"_l2=$l2.png")
    end

end

#=
Q1_part1()
=#
```

```julia
include("utils.jl")
using Plots
using StaticArrays
import Distributions as DT

#=
Q2 part 1
=#
function sample_LHS_points(num_partitions,num_dimensions,seeds_array)

    #=
        num_partitions is essentially the value of N in each dimension.
    =#

    dim_start = -1.0
    dim_end = 1.0
    Δx = (dim_end-dim_start)/(num_partitions)
    dim_sampled_points = zeros(num_partitions)

    sampled_points = Dict{Int,Array{Float64,1}}()

    for d in 1:num_dimensions
        rng = MersenneTwister(seeds_array[d])
        dim_sampled_points = zeros(num_partitions)
        for i in 1:num_partitions
            a = dim_start + (i-1)*Δx
            b = dim_start + i*Δx
            point = rand(rng, DT.Uniform(a,b))
            dim_sampled_points[i] = point
            # println((a,b))
        end
        sampled_points[d] = dim_sampled_points
    end

    return sampled_points
end
#=
seeds_array = (11,111)
num_partitions = 10
num_dimensions = 2
sample_LHS_points(num_partitions,num_dimensions,seeds_array)
=#


function generate_Ys(num_samples,num_dimensions,rng=MersenneTwister(1))

    num_partitions = num_samples
    seeds_array = (11,111)
    sampled_points = sample_LHS_points(num_partitions,num_dimensions,seeds_array)

    Ys = Array{NTuple{num_dimensions,Float64},1}(undef,num_samples)
    shuffled_indices = Dict{Int,Array{Int64,1}}()

    for i in 1:num_dimensions
        shuffled_indices[i] = shuffle(rng,1:num_partitions)
    end

    for j in 1:num_samples
        sampled_Y = MVector{num_dimensions,Float64}(undef)
        for d in 1:num_dimensions
            sampled_Y[d] = sampled_points[d][shuffled_indices[d][j]]
        end
        Ys[j] = Tuple(sampled_Y)
    end

    return Ys
end


function do_LHS_sampling(N,num_dimensions,num_nodes,rng = MersenneTwister(9))
    Ys = generate_Ys(N,num_dimensions,rng)
    u_samples = [generate_u_sample(num_nodes,Ys[i]) for i in 1:N]
    return u_samples
end
```

```julia
function Q2_part_1(N,num_nodes=100,rng = MersenneTwister(19))
    num_dimensions = 2
    u_samples = do_LHS_sampling(N,num_dimensions,num_nodes,rng)
    M = ST.mean(u_samples)
    V = ST.var(u_samples)
    # println("Sample Mean of U samples over ",num_nodes," nodes : ",M)
    # println("Sample Variance of U samples: over ",num_nodes," nodes : ",V)
    return u_samples,M,V
end


function visulization_Q2_part1()

    num_nodes = 100
    num_dimensions = 2
    # num_samples_array = (100,200,500,1000,2000,5000)
    # num_samples_array = (100,200,500,1000,2000,5000,10000,20000,50000)
    # num_samples_array = (100,200)
    num_samples_array = (1:1:20)
    means = []
    variances = []
    seed = 13
    for N in num_samples_array
        rng = MersenneTwister(seed)
        u_samples,M,V = Q2_part_1(N,num_nodes,rng)
        push!(means,M)
        push!(variances,V)
    end

    snapshot1 = plot(size=(700,700), dpi=300,
        # xticks=:0.1:1.0, yticks=0:0.2:3,
        xlabel="Node i", ylabel="Mean of 'u' value for Node i ",
        title="Plot for Mean of 'u' value across different nodes with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
    # legend=false
    # Distribution
    # Density
    )

    for i in 1:length(num_samples_array)
        M = means[i]
        N = num_samples_array[i]
        plot!(snapshot1,1:num_nodes,M,label="N = $N",linewidth=2)
    end
    display(snapshot1)

    snapshot2 = plot(size=(700,700), dpi=300,
    # xticks=:0.1:1.0, yticks=0:0.2:3,
    xlabel="Node i", ylabel="Variance of 'u' value for Node i ",
    title="Plot for Variance of 'u' value across different nodes with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
    # legend=false
    # Distribution
    # Density
    )

    for i in 1:length(num_samples_array)
        V = variances[i]
        N = num_samples_array[i]
        plot!(snapshot2,1:num_nodes,V,label="N = $N",linewidth=2)
    end
    display(snapshot2)

    return snapshot1,snapshot2
end

#=
num_samples = 10
num_nodes = 101
Q2_part_1(num_samples,num_nodes,MersenneTwister(11))
s1,s2 = visulization_Q2_part1()
=#
```

```julia
include("spquad.jl")
include("utils.jl")
using StaticArrays
using Plots


function tensor_product_grid(one_d_points, one_d_weights, num_dimensions)

    d = num_dimensions
    N = length(one_d_points)  # N = 2^l + 1
    total_points = N^d

    grid_points = Array{NTuple{d,Float64},1}(undef,total_points)
    grid_weights = zeros(total_points)

    for i in 1:N
        for j in 1:N
            index = (i-1)*N + j
            grid_points[index] = (one_d_points[i],one_d_points[j])
            grid_weights[index] = one_d_weights[i]*one_d_weights[j]
        end
    end

    return grid_points, grid_weights
end


function do_tensor_product_quadrature_clenshaw_curtis(num_dimensions, l, f)

    # Get the points and weights for the 1D quadrature
    one_d_points, one_d_weights = spquad(1,l)

    # Get the grid points and weights for the tensor product quadrature
    grid_points, grid_weights = tensor_product_grid(one_d_points, one_d_weights, num_dimensions)

    # Compute the integral
    start_index = 1
    integral = f(grid_points[start_index])*grid_weights[start_index]
    for i in (start_index+1):length(grid_points)
        integral += (f(grid_points[i])*grid_weights[i])
    end

    return integral
end


function Q2_part2_mean(num_nodes,d,l)
    f(Y) = generate_u_sample(num_nodes,Y)
    integral = do_tensor_product_quadrature_clenshaw_curtis(d, l, f)
    mean = 0.5*0.5*integral
    return mean
end

function compute_matrix_square(x)
    return x.*x
end

function Q2_part2_mean_square(num_nodes,d,l)
    f(Y) = compute_matrix_square(generate_u_sample(num_nodes,Y))
    integral = do_tensor_product_quadrature_clenshaw_curtis(d, l, f)
    mean_square = 0.5*0.5*integral
    return mean_square
end


function visulization_Q2_part2()

    num_nodes = 100
    d = 2 #num_dimensions
    num_l = 4
    l_array = SVector(0,2,4,5)
    num_samples_array = MVector{num_l,Int}([2^i+1 for i in l_array])
    num_samples_array[1] = 1

    means = []
```

```
    variances = []

    for l in l_array
        M = Q2_part2_mean(num_nodes,d,l)
        M_square = Q2_part2_mean_square(num_nodes,d,l)
        V = M_square .- compute_matrix_square(M)
        push!(means,M)
        push!(variances,V)
    end

    snapshot1 = plot(size=(700,700), dpi=300,
        # xticks=:0.1:1.0, yticks=0:0.2:3,
        xlabel="Node i", ylabel="Mean of 'u' value for Node i ",
        title="Plot for Mean of 'u' value across different nodes with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
    # legend=false
    # Distribution
    # Density
    )

    for i in 1:length(num_samples_array)
        M = means[i]
        N = num_samples_array[i]
        plot!(snapshot1,1:num_nodes,M,label="N = $N",linewidth=2)
    end
    display(snapshot1)

    snapshot2 = plot(size=(700,700), dpi=300,
    # xticks=:0.1:1.0, yticks=0:0.2:3,
    xlabel="Node i", ylabel="Variance of 'u' value for Node i ",
    title="Plot for Variance of 'u' value across different nodes with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
    # legend=false
    # Distribution
    # Density
    )

    for i in 1:length(num_samples_array)
        V = variances[i]
        N = num_samples_array[i]
        plot!(snapshot2,1:num_nodes,V,label="N = $N",linewidth=2)
    end
    display(snapshot2)

    return snapshot1,snapshot2
end

#=
d = 2 #num_dimensions
num_nodes = 101
l = 4  #N = 2^l + 1

Q2_part2_mean(num_nodes,d,l)
Q2_part2_mean_square(num_nodes,d,l)
s1,s2 = visulization_Q2_part2()
=#
```

```julia
include("spquad.jl")
include("utils.jl")
using StaticArrays
using Plots


function do_smolyak_sparse_grid_clenshaw_curtis(num_dimensions, l, f)

    # Get the points and weights for the 1D quadrature
    points, weights = spquad(num_dimensions,l)
    points = [Tuple(c) for c in eachrow(points)]

    # Compute the integral
    start_index = 1
    integral = f(points[start_index])*weights[start_index]
    for i in (start_index+1):length(points)
        integral += (f(points[i])*weights[i])
    end

    return integral
end


function Q2_part3_mean(num_nodes,d,l)
    f(Y) = generate_u_sample(num_nodes,Y)
    integral = do_smolyak_sparse_grid_clenshaw_curtis(d, l, f)
    mean = 0.5*0.5*integral
    return mean
end

function compute_matrix_square(x)
    return x.*x
end

function Q2_part3_mean_square(num_nodes,d,l)
    f(Y) = compute_matrix_square(generate_u_sample(num_nodes,Y))
    integral = do_smolyak_sparse_grid_clenshaw_curtis(d, l, f)
    mean_square = 0.5*0.5*integral
    return mean_square
end


function visulization_Q2_part3()

    num_nodes = 100
    d = 2 #num_dimensions
    num_l = 6
    l_array = SVector(0,1,2,3,4,5)
    num_samples_array = MVector{num_l,Int}([2^i+1 for i in l_array])
    num_samples_array[1] = 1

    means = []
    variances = []

    for l in l_array
        M = Q2_part3_mean(num_nodes,d,l)
        M_square = Q2_part3_mean_square(num_nodes,d,l)
        V = M_square .- compute_matrix_square(M)
        push!(means,M)
        push!(variances,V)
    end

    snapshot1 = plot(size=(700,700), dpi=300,
        # xticks=:0.1:1.0, yticks=0:0.2:3,
        xlabel="Node i", ylabel="Mean of 'u' value for Node i ",
        title="Plot for Mean of 'u' value across different nodes with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
    # legend=false
    # Distribution
    # Density
    )

    for i in 1:length(num_samples_array)
        M = means[i]
```

```
        N = num_samples_array[i]
        plot!(snapshot1,1:num_nodes,M,label="N = $N",linewidth=1)
    end
    display(snapshot1)

    snapshot2 = plot(size=(700,700), dpi=300,
    # xticks=:0.1:1.0, yticks=0:0.2:3,
    xlabel="Node i", ylabel="Variance of 'u' value for Node i ",
    title="Plot for Variance of 'u' value across different nodes with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
    # legend=false
    # Distribution
    # Density
    )

    for i in 1:length(num_samples_array)
        V = variances[i]
        N = num_samples_array[i]
        plot!(snapshot2,1:num_nodes,V,label="N = $N",linewidth=1)
    end
    display(snapshot2)

    return snapshot1,snapshot2
end


#=
d = 2 #num_dimensions
num_nodes = 101
l = 4  #N = 2^l + 1

Q2_part3_mean(num_nodes,d,l)
Q2_part3_mean_square(num_nodes,d,l)
s1,s2 = visulization_Q2_part3()
=#
```

```julia
include("Q2_part1.jl")
include("Q2_part2.jl")
include("Q2_part3.jl")

#=
Do MC
Do LHS
Do Tensor Product
Do Smolyak with CC
=#


function generate_u_sample_MC(num_nodes,rng=MersenneTwister(7))

    D = 2
    a = 0.5
    l = 2.0
    σ = 1
    eigenpairs = generate_eigenpairs(D,a,l,σ)
    Y = randn(rng,D)

    # num_nodes is n in the function fem1d_heat_steady
    slab_start = 0.0 #a
    slab_end = 1.0 #b
    boundary_condition_start = 0.0 #ua
    boundary_condition_end = 0.0 #ub
    K(x) = calculate_K_xω(x,eigenpairs,Y) #k
    F(x) = forcing_function(x) #f
    node_points = range(slab_start,stop=slab_end,length=num_nodes) #x

    sampled_u = fem1d_heat_steady(num_nodes,slab_start,slab_end,boundary_condition_start,
                                boundary_condition_end,K,F,node_points)

    return sampled_u
end

function do_monte_carlo_simulations(N,num_nodes,rng = MersenneTwister(77))
    u_samples = [generate_u_sample_MC(num_nodes,rng) for i in 1:N]
    return u_samples
end

function main_MC(N,num_nodes=100,rng=MersenneTwister(777))
    u_samples = do_monte_carlo_simulations(N,num_nodes,rng)
    M = ST.mean(u_samples)
    V = ST.var(u_samples)
    # println("Sample Mean of U samples over ",num_nodes," nodes : ",M)
    # println("Sample Variance of U samples: over ",num_nodes," nodes : ",V)
    return u_samples,M,V
end

function Q2_part4()

    num_dimensions = 2
    num_nodes = 101
    x_index = 51 #index of x = 0.5
    #For MC and LHS
    num_samples_array = collect(0:1:128)
    num_samples_array[1] = 1
    push!(num_samples_array,129)
    #For TP-CC and Smolyak-CC
    l_array = SVector(0,1,2,3,4,5,6,7)
    num_l = length(l_array)
    num_samples_array_using_l = MVector{num_l,Int}([2^i+1 for i in l_array])
    num_samples_array_using_l[1] = 1


    # MC
    means_MC = []
    variance_MC = []
    seed = rand(UInt32)
    for N in num_samples_array
        rng = MersenneTwister(seed)
        u_samples,M,V = main_MC(N,num_nodes,rng)
        push!(means_MC,M[51])
        push!(variance_MC,V[51])
```

```
    end

    #LHS
    means_LHS = []
    variance_LHS = []
    seed = 13
    for N in num_samples_array
        rng = MersenneTwister(seed)
        u_samples,M,V = Q2_part_1(N,num_nodes,rng)
        push!(means_LHS,M[51])
        push!(variance_LHS,V[51])
    end

    #Tensor Product
    means_TP = []
    variance_TP = []
    d = num_dimensions
    for l in l_array
        M = Q2_part2_mean(num_nodes,d,l)
        M_square = Q2_part2_mean_square(num_nodes,d,l)
        V = M_square .- compute_matrix_square(M)
        push!(means_TP,M[51])
        push!(variance_TP,V[51])
    end

    #Smolyak
    means_SG = []
    variance_SG = []
    d = num_dimensions
    for l in l_array
        M = Q2_part3_mean(num_nodes,d,l)
        M_square = Q2_part3_mean_square(num_nodes,d,l)
        V = M_square .- compute_matrix_square(M)
        push!(means_SG,M[51])
        push!(variance_SG,V[51])
    end

    return means_MC,variance_MC,means_LHS,variance_LHS,
            means_TP,variance_TP,means_SG,variance_SG,
            num_samples_array,num_samples_array_using_l

end


function visulization_Q2_part4()


    means_MC,variance_MC,means_LHS,variance_LHS,
    means_TP,variance_TP,means_SG,variance_SG,
    num_samples_array,num_samples_array_using_l = Q2_part4()


    snapshot1 = plot(size=(700,700), dpi=300,
        # xticks=:0.1:1.0, yticks=0:0.2:3,
        xlabel="Number of Samples", ylabel="Mean of 'u' value at x=0.5",
        title="Plot for Mean of 'u' at x=0.5 across different methods with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
    # legend=false
    # Distribution
    # Density
    )

    plot!(snapshot1,num_samples_array,means_MC,label="Monte Carlo",linewidth=2)
    plot!(snapshot1,num_samples_array,means_LHS,label="LHS",linewidth=2)
    plot!(snapshot1,num_samples_array_using_l,means_TP,label="Tensor Product",linewidth=2)
    plot!(snapshot1,num_samples_array_using_l,means_SG,label="Smolyak-CC",linewidth=2)
    display(snapshot1)

    snapshot2 = plot(size=(700,700), dpi=300,
        # xticks=:0.1:1.0, yticks=0:0.2:3,
        xlabel="Number of Samples", ylabel="Variance of 'u' value at x=0.5",
        title="Plot for Variance of 'u' at x=0.5 across different methods with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
```

```
    # legend=false
    # Distribution
    # Density
    )

    plot!(snapshot2,num_samples_array,variance_MC,label="Monte Carlo",linewidth=2)
    plot!(snapshot2,num_samples_array,variance_LHS,label="LHS",linewidth=2)
    plot!(snapshot2,num_samples_array_using_l,variance_TP,label="Tensor Product",linewidth=2)
    plot!(snapshot2,num_samples_array_using_l,variance_SG,label="Smolyak-CC",linewidth=2)
    display(snapshot2)

    return snapshot1,snapshot2

end

#=
Q2_part4()
visulization_Q2_part4()
=#


function visulization_just_MC()

    num_nodes = 101
    # num_samples_array = (100,200,500,1000,2000,5000,10000,20000,50000)
    num_samples_array = (1:1:5)
    means = []
    variances = []
    for N in num_samples_array
        u_samples,M,V = main_MC(N,num_nodes)
        push!(means,M)
        push!(variances,V)
    end

    snapshot1 = plot(size=(700,700), dpi=300,
        # xticks=:0.1:1.0, yticks=0:0.2:3,
        xlabel="Node i", ylabel="Mean of 'u' value for Node i ",
        title="Plot for Mean of 'u' value across different nodes with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
    # legend=false
    # Distribution
    # Density
    )

    for i in 1:length(num_samples_array)
        M = means[i]
        N = num_samples_array[i]
        plot!(snapshot1,1:num_nodes,M,label="N = $N",linewidth=2)
    end
    display(snapshot1)

    snapshot2 = plot(size=(700,700), dpi=300,
    # xticks=:0.1:1.0, yticks=0:0.2:3,
    xlabel="Node i", ylabel="Variance of 'u' value for Node i ",
    title="Plot for Variance of 'u' value across different nodes with \n varying number of samples"
    # axis=([], false),
    # legend=:bottom,
    # legend=false
    # Distribution
    # Density
    )

    for i in 1:length(num_samples_array)
        V = variances[i]
        N = num_samples_array[i]
        plot!(snapshot2,1:num_nodes,V,label="N = $N",linewidth=2)
    end
    display(snapshot2)

    return snapshot1,snapshot2
end
```

```julia
include("../steady_state_math.jl")
include("../helper_functions.jl")

#Function that gives you K(x,ω) at any given x for a fixed ω determined by λ,ϕ and Y
function calculate_K_xω(x,eigenpairs,Y)
    K_bar = 1
    σ_K = 0.1
    d = length(eigenpairs)
    a = 0.5   #IS this needed? Should it be zero for this Q?

    s = 0.0
    T = x - a
    for i in 1:length(eigenpairs)
        (index,λ,ϕ) = eigenpairs[i]
        s += sqrt(λ)*ϕ(T)*Y[i]
    end
    s = K_bar + σ_K*s
    return s
end

function forcing_function(t)
    return -1.0
end

function generate_u_sample(num_nodes,Y)

    D = 2
    #D = length(Y)
    a = 0.5
    l = 2.0
    σ = 1
    eigenpairs = generate_eigenpairs(D,a,l,σ)

    # num_nodes is n in the function fem1d_heat_steady
    slab_start = 0.0 #a
    slab_end = 1.0 #b
    boundary_condition_start = 0.0 #ua
    boundary_condition_end = 0.0 #ub
    K(x) = calculate_K_xω(x,eigenpairs,Y) #k
    F(x) = forcing_function(x) #f
    node_points = range(slab_start,stop=slab_end,length=num_nodes) #x

    sampled_u = fem1d_heat_steady(num_nodes,slab_start,slab_end,boundary_condition_start,
                            boundary_condition_end,K,F,node_points)

    return sampled_u
end
```