In [1]:
```
using CSV
using DataFrames
using JuMP
using Plots
using Random
using Statistics
using LinearAlgebra
using Distributions
using BipartiteMatching
using Gurobi
using LinearAlgebra
using SymPy
using NLsolve
using LaTeXStrings
```

Proof of Theorem 7 (i)

Goal

Let SOD stand for the second-order derivative of $\mu^{KS}(1/2, 1/2)$ in the direction $(0, 1)$. The notebook aims to verify if the upper bound of SOD < 0 within each set [alphaf, alphaf + delta) × [alpha,alpha + delta).

Arguments

- `x1_lb` `x2_lb` `x1_ub` `x2_ub` : The respective lower and upper bounds of x_1 and x_2.
- `alphaf` or `alphaf_val` : Value of α^f.
- `alpha` or `alpha_val` : Value of α.
- `delta` or `delta_val` : Small positive increment.

Functions

- `equation_x1!(F, x, alphaf_val, alpha_val)` return the numerical solutions of x_1 and x_2. `ub_concave_sod` uses these values to build an upper bound of SOD for a set [alphaf, alphaf + delta) × [alpha,alpha + delta).
- `sol_f_func2(alphaf_val, alpha_val, delta_val)` returns whether the upper bound < 0.
- `calculate_ub_concave_sod_matrix(delta_val)` examines all alphaf and alpha within the claimed region.

Outputs

- `calculate_ub_concave_sod_matrix(delta_val)` returns a boolean matrix, which we plot in heatmap so that (1) the inequality holds in the red region (2) fails in the blue region, and (3) in the grey region the parameters fall outside the tree-like regime.

In [2]:
```
# Define an expression to compute the upper bound of SOD in each set.
@vars delta alphaf alpha x1_lb x2_lb x1_ub x2_ub
ub_concave_sod = (-2*(x1_lb+x2_lb)*((x1_lb*x2_lb)*(max(alphaf-alpha-de
lta,0))^2)^2 -16*(x1_lb+x2_lb)*(x1_lb*x2_lb)*alphaf*alpha
    + 8*(alphaf+delta)^2 *(x1_ub*x2_ub^2+4*x1_ub^3)+ 8*(alpha+delta)^2
*(x1_ub^2*x2_ub+4*x2_ub^3)
    -24*alphaf^2*x1_lb^2*x2_lb-24*alpha^2*x1_lb*x2_lb^2
    )/(-(x1_ub*x2_ub*(alphaf-alpha + delta)^2)^2 + 8*(alphaf+alpha)^2*
x1_lb*x2_lb +16*alpha^2 *x2_lb^2+16*alphaf^2 *x1_lb^2-16)

# Define a Julia function to solve x1.
function equation_x1!(F, x, alphaf_val, alpha_val)
    F[1] = exp(-0.5*(alphaf_val + alpha_val)*x[1] + 2*alpha_val*(log(x
[1]) + alphaf_val*x[1])/(alphaf_val + alpha_val)) + 2*(log(x[1]) + alp
haf_val*x[1])/(alphaf_val + alpha_val)
end

# Define a Julia function to verify if the upper bound < 0.
tolerance = 1e-5
function sol_f_func2(alphaf_val, alpha_val, delta_val)
    x1_sol = nlsolve((F, x) -> equation_x1!(F, x, alphaf_val, alpha_va
l), [0.2], autodiff=:forward, ftol=tolerance).zero[1]
    x1_sol_ub, x1_sol_lb = x1_sol+tolerance, x1_sol-tolerance
    x2_sol_ub, x2_sol_lb = -2*(log(x1_sol_lb) + alphaf_val*x1_sol_lb)/
(alphaf_val + alpha_val), -2*(log(x1_sol_ub) + alphaf_val*x1_sol_ub)/
(alphaf_val + alpha_val)
    ub_concave_sod_val = subs(ub_concave_sod, (alphaf, alphaf_val), (a
lpha, alpha_val),
        (x1_lb, x1_sol_lb*(1-2*delta)), (x2_lb, x2_sol_lb*(1-2*delt
a)),
        (x1_ub, x1_sol_ub), (x2_ub, x2_sol_ub), (delta,delta_val))
    return ub_concave_sod_val < 0
end
```

Out[2]: sol_f_func2 (generic function with 1 method)

In [3]:

```
# This function iterates over alphaf and alpha, recording 1 if the ine
quality is satisfied, 0 if it fails, and -1 if the parameters fall out
side the tree-like regime.
function calculate_ub_concave_sod_matrix(delta_val)

    # Adjust the upper limit as needed
    alphaf_range, alpha_range = 0.0:delta_val:exp(1), 0.0:delta_val:ex
p(1)/2
    alphaf_range, alpha_range = collect(alphaf_range), collect(alpha_r
ange)
    alphaf_range[1] = 0.0001
    alpha_range[1] = 0.0001
    results_matrix = Matrix{Int}(undef, length(alphaf_range), length(a
lpha_range))

    for (i, alphaf_val) in enumerate(alphaf_range)
        print(alphaf_val, '\n')
        for (j, alpha_val) in enumerate(alpha_range)
            if alpha_val <= min(alphaf_val, exp(1) - alphaf_val)
                results_matrix[i, j] = sol_f_func2(alphaf_val, alpha_v
al, delta_val)
            else
                results_matrix[i, j] = -1  # Assign -1 when the condit
ion is not satisfied
            end
        end
    end

    return results_matrix
end
```

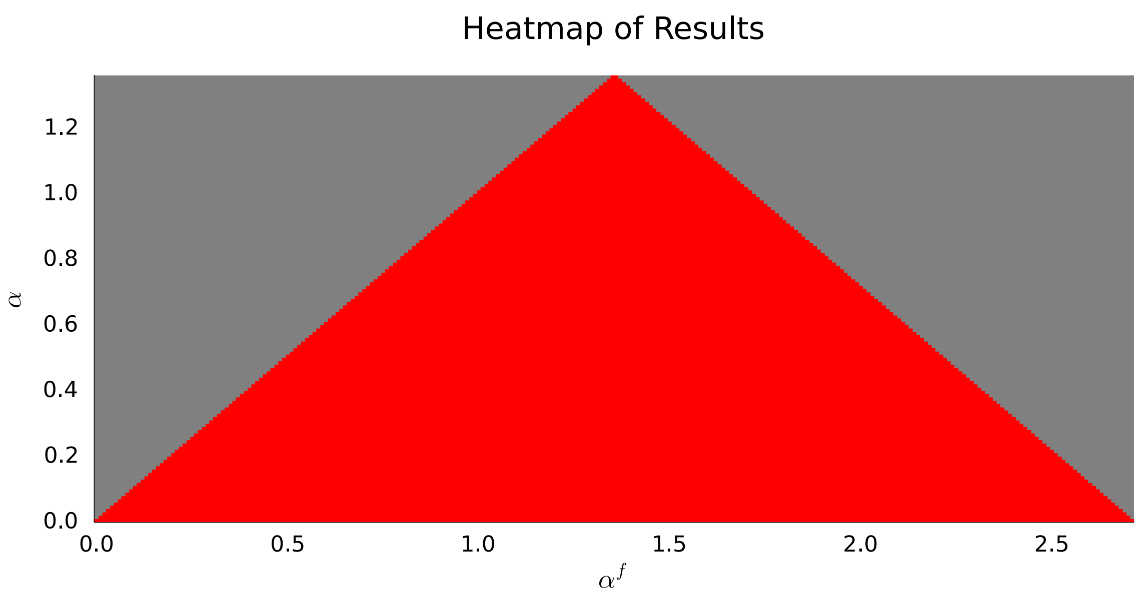
Out[3]: calculate_ub_concave_sod_matrix (generic function with 1 method)

In [ ]:

```
# delta = 0.01: 15 min
delta_val = 0.01
results_matrix = calculate_ub_concave_sod_matrix(delta_val)
CSV.write("concave01.csv", DataFrame(results_matrix, :auto), writehead
er=false)
```

In [4]:
```
# Specify delta and load results
delta_val = 0.01

loaded_dataframe = CSV.File("concave01.csv", header=false) |> DataFram
e
results_matrix = Matrix(loaded_dataframe)

# Create a heatmap to visualize the results
color_palette = cgrad([:gray, :blue, :red], categorical=true)

heatmap(0.0:delta_val:exp(1), 0.0:delta_val:exp(1)/2, transpose(result
s_matrix), color=:auto, colormap=color_palette, colorbar=false, title
="Heatmap of Results",
        xlabel=L"$\alpha^f$", ylabel=L"$\alpha$", tickfont=font(10), g
uidefontsize=12)

plot!(size=(800, 400), margin=5Plots.mm)
```

Out[4]:



Proof of Theorem 7 (ii)

Goal

Let SOD stand for the second-order derivative of $\mu^{KS}(1/2, 1/2)$ in the direction $(1, -1)$. The notebook aims to verify if the lower bound of SOD > 0 within each set [alphaf, alphaf + delta) × [alpha,alpha + delta).

Arguments

- `x1_lb x2_lb x1_ub x2_ub` : The respective lower and upper bounds of x_1 and x_2.
- `alphaf or alphaf_val` : Value of α^f.
- `alpha or alpha_val` : Value of α.
- `delta or delta_val` : Small positive increment.

Functions

- `equation_x1!(F, x, alphaf_val, alpha_val)` return the numerical solutions of x_1 and x_2. `lb_convex_sod` uses these values to build a lower bound of SOD for a set [alphaf, alphaf + delta) × [alpha,alpha + delta).
- `sol_f_func(alphaf_val, alpha_val, delta_val)` returns whether the lower bound > 0.
- `calculate_lb_convex_sod_matrix(delta_val)` examines all alphaf and alpha within the claimed region.

Outputs

- `calculate_lb_convex_sod_matrix(delta_val)` returns a boolean matrix, which we plot in heatmap so that (1) the inequality holds in the red region (2) fails in the blue region, and (3) in the grey region the parameters fall outside the tree-like regime.

In [5]:
```
# Define an expression to compute the lower bound of SOD in each set.
@vars delta alphaf alpha x1_lb x2_lb x1_ub x2_ub
lb_convex_sod = (-(alphaf-alpha+delta)^2 *4*x1_ub*x2_ub*(x1_ub+x2_ub)+
16*max(0,x2_lb-x1_ub)*alphaf*x1_lb-16*(x2_ub-x1_lb)*(alpha+delta)*x2_u
b
    )/(-(max(alphaf-alpha-delta,0))^2 *x1_lb*x2_lb + 4*(1-alpha*x2_lb-
alphaf*x1_lb))

# Define a Julia function to solve x1.
function equation_x1!(F, x, alphaf_val, alpha_val)
    F[1] = exp(-0.5*(alphaf_val + alpha_val)*x[1] + 2*alpha_val*(log(x
[1]) + alphaf_val*x[1])/(alphaf_val + alpha_val)) + 2*(log(x[1]) + alp
haf_val*x[1])/(alphaf_val + alpha_val)
end

# Define a Julia function to verify if the lower bound > 0.
tolerance = 1e-5
function sol_f_func(alphaf_val, alpha_val, delta_val)
    x1_sol = nlsolve((F, x) -> equation_x1!(F, x, alphaf_val, alpha_va
l), [0.2], autodiff=:forward, ftol=tolerance).zero[1]
    x1_sol_ub, x1_sol_lb = x1_sol+tolerance, x1_sol-tolerance
    x2_sol_ub, x2_sol_lb = -2*(log(x1_sol_lb) + alphaf_val*x1_sol_lb)/
(alphaf_val + alpha_val), -2*(log(x1_sol_ub) + alphaf_val*x1_sol_ub)/
(alphaf_val + alpha_val)
    lb_convex_sod_val = subs(lb_convex_sod, (alphaf, alphaf_val), (alp
ha, alpha_val),
        (x1_lb, x1_sol_lb*(1-2*delta)), (x2_lb, x2_sol_lb*(1-2*delt
a)),
        (x1_ub, x1_sol_ub), (x2_ub, x2_sol_ub),(delta,delta_val))
    return lb_convex_sod_val > 0
end
```

Out[5]:
```
sol_f_func (generic function with 1 method)
```

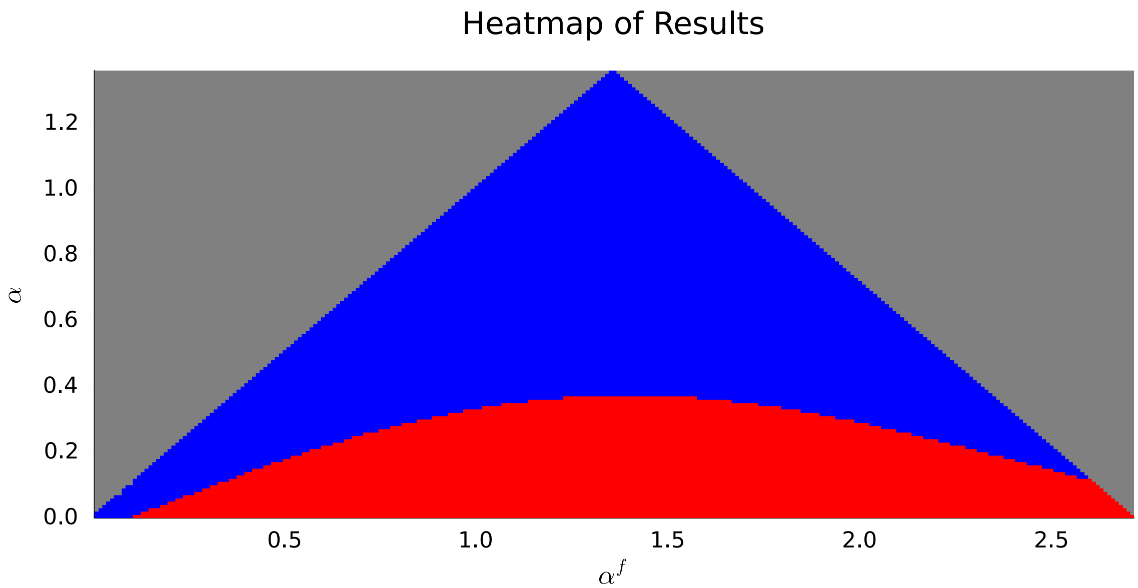
In [6]:
```
# This function iterates over alphaf and alpha, recording 1 if the ine
quality is satisfied, 0 if it fails, and -1 if the parameters fall out
side the tree-like regime.
function calculate_lb_convex_sod_matrix(delta_val)

    alphaf_range = delta_val:delta_val:exp(1)
    alpha_range = 0.0:delta_val:exp(1)/2
    results_matrix = Matrix{Int}(undef, length(alphaf_range), length(a
lpha_range))

    for (i, alphaf_val) in enumerate(alphaf_range)
        print(alphaf_val, '\n')
        for (j, alpha_val) in enumerate(alpha_range)
            if alpha_val <= min(alphaf_val, exp(1) - alphaf_val)
                results_matrix[i, j] = sol_f_func(alphaf_val, alpha_va
l, delta_val)
            else
                results_matrix[i, j] = -1  # Assign -1 when the condit
ion is not satisfied
            end
        end
    end

    return results_matrix

end
```

Out[6]: calculate_lb_convex_sod_matrix (generic function with 1 method)

In [ ]:
```
# delta = 0.01: 15 min
delta_val = 0.01
results_matrix = calculate_lb_convex_sod_matrix(delta_val)
CSV.write("convex01.csv", DataFrame(results_matrix, :auto), writeheade
r=false)
```

In [7]:
```
# Specify delta and load results
delta_val = 0.01

loaded_dataframe = CSV.File("convex01.csv", header=false) |> DataFrame
results_matrix = Matrix(loaded_dataframe)

# Create a heatmap to visualize the results
color_palette = cgrad([:gray, :blue, :red], categorical=true)

heatmap(delta_val:delta_val:exp(1), 0.0:delta_val:exp(1)/2, transpose
(results_matrix), color=:auto, colormap=color_palette, colorbar=false,
title="Heatmap of Results",
        xlabel=L"$\alpha^f$", ylabel=L"$\alpha$", tickfont=font(10), g
uidefontsize=12)

plot!(size=(800, 400), margin=5Plots.mm)
```

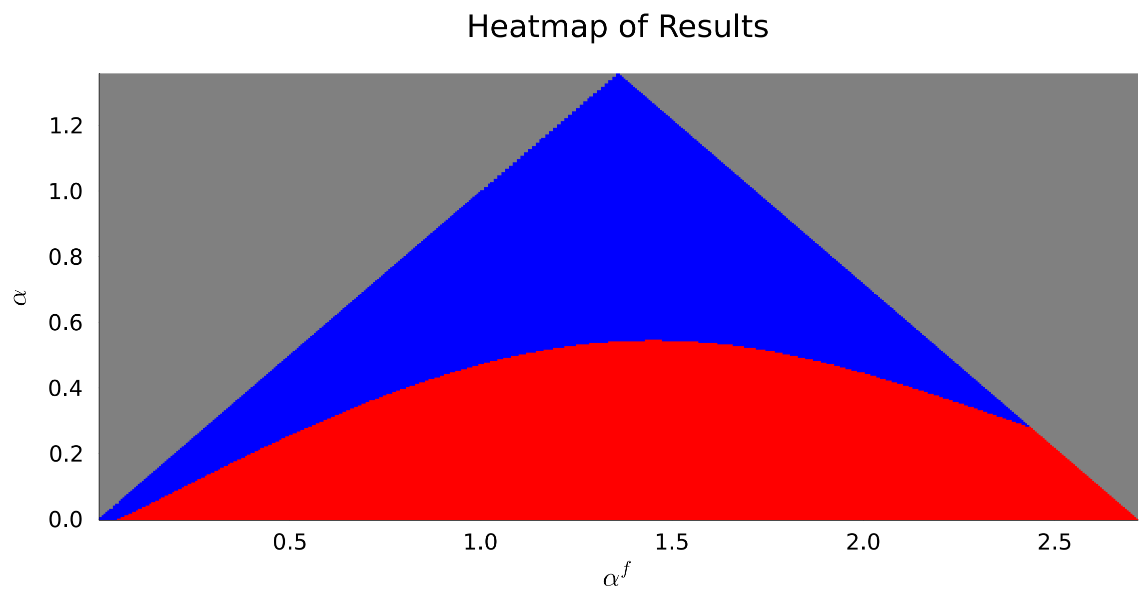
Out[7]:



In [ ]:
```
# delta = 0.005: 60 min
delta_val = 0.005
results_matrix = calculate_lb_convex_sod_matrix(delta_val)
CSV.write("convex005.csv", DataFrame(results_matrix, :auto), writehead
er=false)
```

In [8]:
```
# Specify delta and load results
delta_val = 0.005

loaded_dataframe = CSV.File("convex005.csv", header=false) |> DataFram
e
results_matrix = Matrix(loaded_dataframe)

# Create a heatmap to visualize the results
color_palette = cgrad([:gray, :blue, :red], categorical=true)

heatmap(delta_val:delta_val:exp(1), 0.0:delta_val:exp(1)/2, transpose
(results_matrix), color=:auto, colormap=color_palette, colorbar=false,
title="Heatmap of Results",
        xlabel=L"$\alpha^f$", ylabel=L"$\alpha$", tickfont=font(10), g
uidefontsize=12)

plot!(size=(800, 400), margin=5Plots.mm)
```

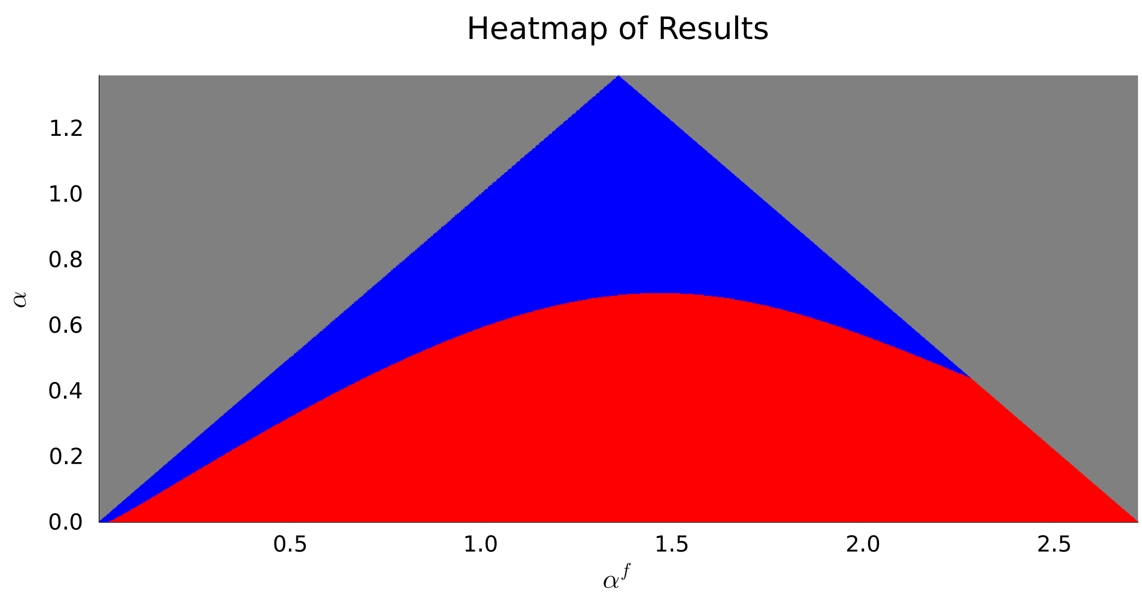
Out[8]:



In [ ]:
```
# delta = 0.0025: 4 hours
delta_val = 0.0025
results_matrix = calculate_lb_convex_sod_matrix(delta_val)
CSV.write("convex0025.csv", DataFrame(results_matrix, :auto), writehea
der=false)
```

In [9]:
```
# Specify delta and load results
delta_val = 0.0025

loaded_dataframe = CSV.File("convex0025.csv", header=false) |> DataFra
me
results_matrix = Matrix(loaded_dataframe)

# Create a heatmap to visualize the results
color_palette = cgrad([:gray, :blue, :red], categorical=true)

heatmap(delta_val:delta_val:exp(1), 0.0:delta_val:exp(1)/2, transpose
(results_matrix), color=:auto, colormap=color_palette, colorbar=false,
title="Heatmap of Results",
        xlabel=L"$\alpha^f$", ylabel=L"$\alpha$", tickfont=font(10), g
uidefontsize=12)

plot!(size=(800, 400), margin=5Plots.mm)
```

Out[9]:

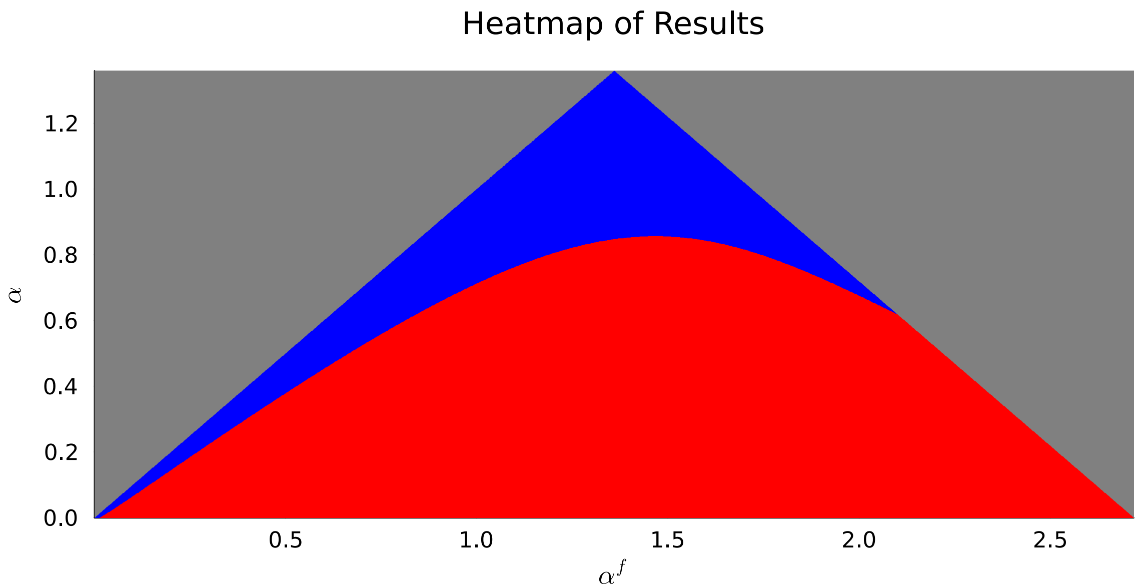


In [ ]:
```
# delta = 0.001: 20 hours
delta_val = 0.001
results_matrix = calculate_lb_convex_sod_matrix(delta_val)
CSV.write("convex001.csv", DataFrame(results_matrix, :auto), writehead
er=false)
```

In [10]:

```
# Specify delta and load results
delta_val = 0.001

loaded_dataframe = CSV.File("convex001.csv", header=false) |> DataFram
e
results_matrix = Matrix(loaded_dataframe)

# Create a heatmap to visualize the results
color_palette = cgrad([:gray, :blue, :red], categorical=true)

heatmap(delta_val:delta_val:exp(1), 0.0:delta_val:exp(1)/2, transpose
(results_matrix), color=:auto, colormap=color_palette, colorbar=false,
title="Heatmap of Results",
        xlabel=L"$\alpha^f$", ylabel=L"$\alpha$", tickfont=font(10), g
uidefontsize=12)

plot!(size=(800, 400), margin=5Plots.mm)
```

Out[10]:



In [ ]: