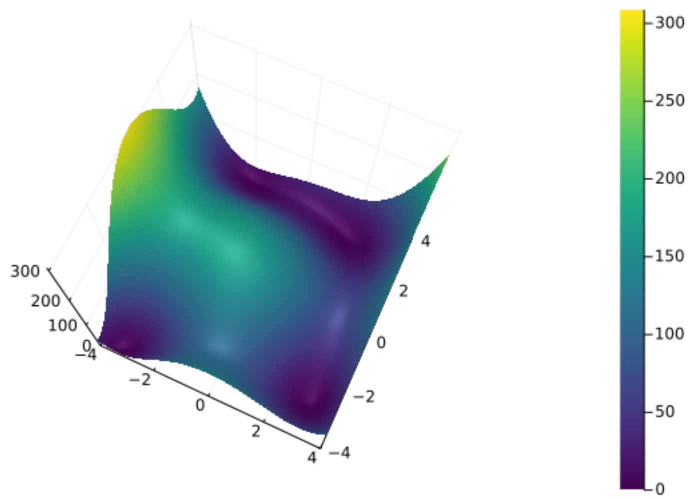
John Higgins
Programming Bootcamp
6 July, 2022

Note: All code and plots can be found here!

Problem 1.

Solution:

a) The surface plot and code to produce it are included below. There appear to be four local minima of the function on this domain.



Code to generate the plot is included below:

```
##problem 1
function f(vec::Vector{Float64})
    #break vector input into x and y components
    x = vec[1]
    y = vec[2]
    #evaluate function
    val =(x^2 + y - 11)^2 + (x + y^2 - 7)^2
    val
end
#create domain grid and matrix for z values
x_grid = collect(-4.0:0.01: 4.0)
nx = length(x_grid)
z_grid = zeros(nx,nx)
```

```
#loop over x and y grids
for i=1:nx, j=1:nx
    #evaluate function at corresponding grid values and set z_grid equal to function value
    z_grid[i,j] = f(x_grid[i], x_grid[j])
end

#plot and save figure
plot_1a = Plots.surface(x_grid, x_grid, z_grid, seriescolor=:viridis, camera=(25,70))
savefig(plot_1a, "plot1a.png")
```

b) The gradient is given by

$$\nabla(x,y) = \langle 4x(x^2+y-11)+2(x+y^2-7), 2(x^2+y-11)+4y(x+y^2-7)\rangle$$

and the Hessian is given by

$$H(x,y) = \begin{bmatrix} 12x^2+4y-42 & 4x+4y \\ 4x+4y & 4x+12y^2-26 \end{bmatrix}$$

Using Newton's method, we find that the four local minima are located at (-3.78, -3.28), (3.58, -1.85), (3,2), and (-2.81, 3.13). These are obtained using initial guesses (-4,-4), (4,-4), (4,4), and (-4, 4). Each of these guesses is located near one of the local minima, so the algorithm ends up finding the closest one to each initial guess.

```
#create gradient function
function g(G, guess::Vector{Float64})
    #split guess vector into component parts
    x,y = guess[1], guess[2]
    G[1] = 4*x*(x^2 + y - 11) + 2(x + y^2 - 7)
    G[2] = 2*(x^2+y-11) + 4*y*(x + y^2 - 7)
```

```
    G
end
#create Hessian function
function h(H, guess::Vector{Float64})
    #split guess vector into component parts
    x,y = guess[1], guess[2]
    #define hessian matrix values
    H[1,1] = 12*x^2 + 4*y - 42
    H[1,2] = 4x + 4y
    H[2,1] = 4x + 4y
    H[1,1] = 4x + 12y^2 - 26
    H
end

using Optim
#create guesses (should have done a loop)
guess1 = [-4.0,-4.0]
guess2 = [4.0,-4.0]
guess3 = [4.0,4.0]
guess4 = [-4.0,4.0]

#find minima given initial guesses
opt1 = optimize(f, g,h, guess1)
opt2 = optimize(f, g,h, guess2)
opt3 = optimize(f, g,h, guess3)
opt4 = optimize(f, g,h, guess4)
println(opt1.minimizer)
println(opt2.minimizer)
println(opt3.minimizer)
println(opt4.minimizer)
>>[-3.7793102534670995, -3.2831859913022914]
>>[3.584428340395532, -1.84812652689675]
>>[3.000000000059233, 1.999999999921049]
```

```
>>[-2.805118087105429, 3.131312518242223]
```

c) For (-3.77, -3.28), Nedler-Mead took 38 iterations and Newton's method took 96 iterations. For (3.58, -1.85), Nedler-Mead took 34 iterations and Newton's method took 72. For (3,2), Nedler-Mead took 39 iterations and Newton's method took 13 iterations. For (-2.81, 3.13), Nedler-Mead took 42 iterations and Newton's method took 758 (!) iterations. Code is included below:

```
opt1n = optimize(f, guess1)
opt2n = optimize(f, guess2)
opt3n = optimize(f, guess3)
opt4n = optimize(f, guess4)
```

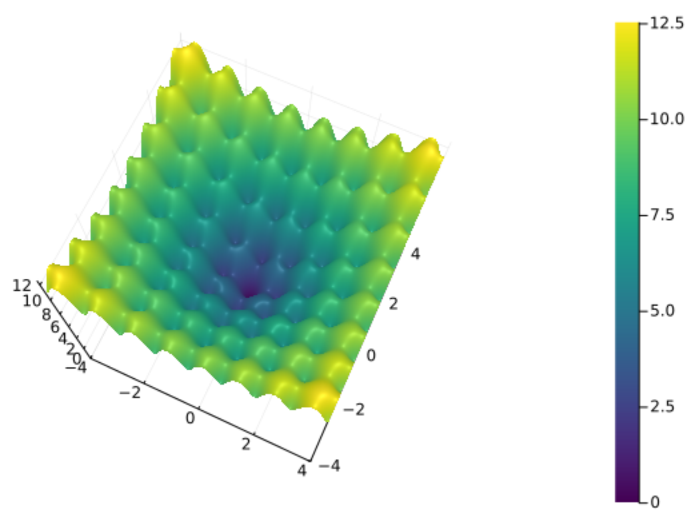
Nedler-Mead offers better performance for all of the minima except for (3,2).

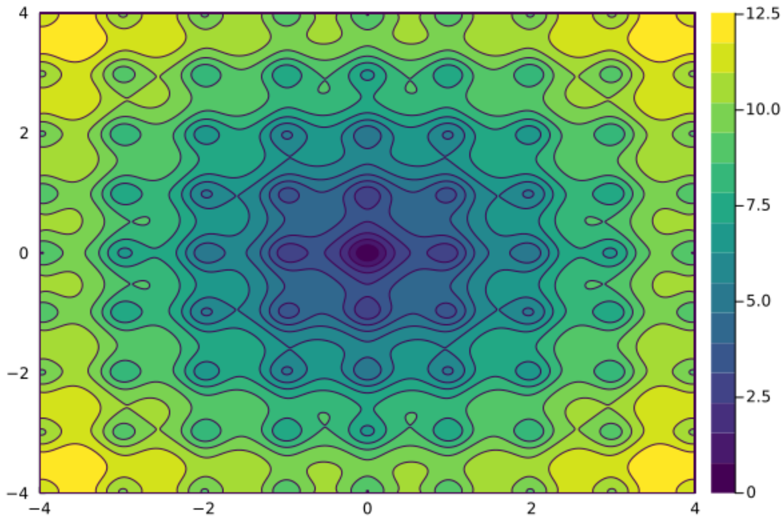
□

Problem 2.

Solution:

a) Surface and contour plots are included below. Upon inspection of the graphs, we find that the global minimum is located at (0,0).






```
#Ackley function
function Ackley(vec::Vector{Float64})
    #take vector and split into x and y components
    x,y = vec[1], vec[2]
    #evaluate function
    val = -20*exp(-0.2*sqrt(0.5*(x^2+y^2))) - exp(0.5(cos(2*pi*x) +
   cos(2*pi*y)))+ exp(1) + 20
    val
end

#create x_grid and 2d grid for function values
x_grid = collect(-4.0:0.01: 4.0)
nx = length(x_grid)
z_grid = zeros(nx,nx)
#loop through x and y arrays and evaluate function at each point then
    store in z_grid
for i=1:nx, j=1:nx
    vec = [x_grid[i], x_grid[j]]
    z_grid[i,j] = Ackley(vec)
end

#plot and save the surface and contour plots
```

```
plot_2a_surf = Plots.surface(x_grid, x_grid, z_grid, seriescolor=:
    viridis, camera=(25,70))
plot_2a_cont = Plots.contourf(x_grid, x_grid, z_grid, seriescolor=:
    viridis)

savefig(plot_2a_surf, "plot2a_surf.png")
savefig(plot_2a_cont, "plot2a_cont.png")


```

b) A table of initial guesses and algorithm performance is included below. Upon inspection, it appears that LBFGS takes far fewer iterations than Nedler-Mead. However, both algorithms seem to miss the global minimum for values farther away from the origin. I chose some guesses that are close to the local minima not at the origin, and it appears that both algorithms can get tripped up at different places. Code to generate the guesses is included below as well.

Guess	NM Min	NM Iter	LBFGS Min	LBFGS Iter
(0.1, 0.1)	(0, 0)	51	(0, 0)	9
(2.0, 1.0)	(1.96, 0.98)	32	(-0.95, 0)	8
(2.0, 1.5)	(0, 0)	57	(0.98, 1.96)	9
(0, 1.0)	(0, 0.95)	25	(0, 0)	9
(2.0, 2.0)	(0, 0)	63	(0.97, 0.97)	4
(2.2, 2.2)	(0, 0)	58	(0, 0)	11
(3.0, 3.2)	(2.96, -1.17)	40	(1.17, 0)	20

Table 1: Guesses, minimizers, and iterations

```
#initialize array of guesses to try
guesses = [[0.1, 0.1], [2,1], [2, 1.5],[0, 1], [2,2], [2.2,2.2],
    [3,3.2]]
#loop through guesses in array
for guess in guesses
    println(guess)
    #run both optimization algorithms
```

```
    opt_nm = optimize(Ackley, guess)
    opt_lb = optimize(Ackley, guess, LBFGS())
    #print results
    println("NM",opt_nm.minimizer, opt_nm.iterations)
    println("LBFGS",opt_lb.minimizer, opt_lb.iterations)
end
```

□

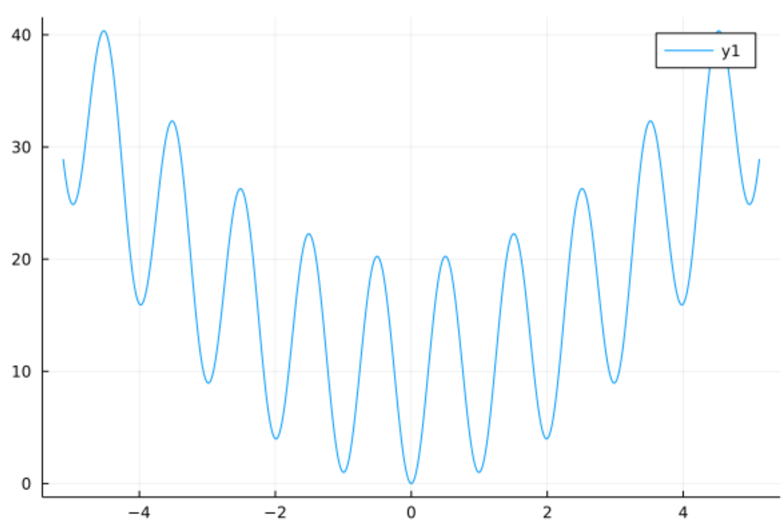
Problem 3.

Solution:

a) The global minimum is zero at $x = 0$. The minimum holds for arbitrary n. Note that $x_i^2 \geq 0$ and $-A\cos(2\pi x_i) \geq -A \; \forall x_i \in \mathbb{R}$. Hence, we have that

$$f(x) = An + \sum_{i=1}^{n}[x_i^2 - A\cos(2\pi x_i)] \geq An - \sum_{i=1}^{n} A = An - An = 0 = f(0)$$

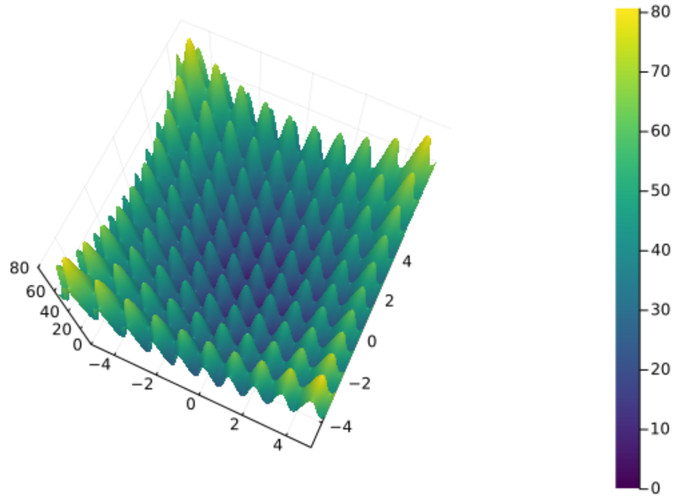
where we can see that the function attains its global minimum at the origin for any arbitrary n. A plot of the function and the code used to generate it are included below:

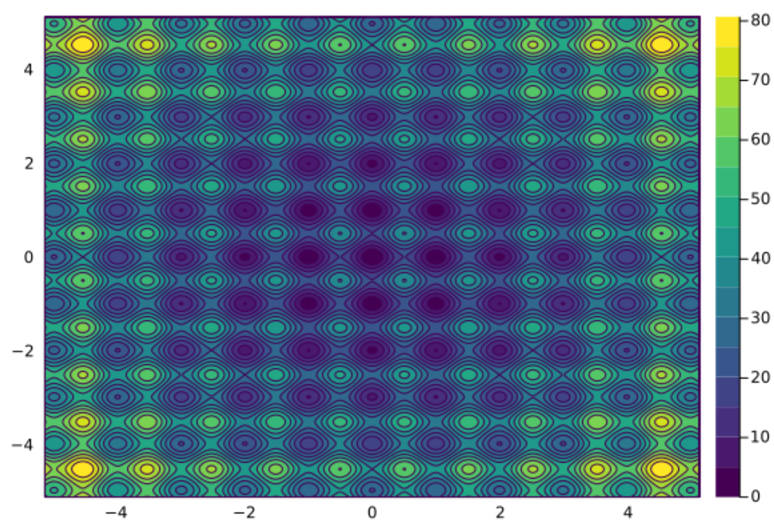



```
function Rastrigin(vec::Vector{Float64})
```

```
    #start with the constant term
    val = 10*length(vec)
    #summation term: iteratively add the x_i terms
    for x_i in vec
        val += x_i^2 - 10*cos(2*pi*x_i)
    end
    val
end

#create arrays for domain and function values
x_grid = collect(-5.12:0.01:5.12)
nx = length(x_grid)
z_grid = zeros(nx)
#loop over domain and evaluate function at each point
for i=1:nx
    z_grid[i] = Rastrigin([x_grid[i]])
end
using Plots
#plot and save figure
plot_3a = plot(x_grid, z_grid)
savefig(plot_3a, "plot3a.png")
```

b) The plots and code for $n = 2$ are included below:





```
#create array of zeros for plotting
z_grid = zeros(nx,nx)
#loop over x and y and evaluate function
for i=1:nx, j = 1:nx
    #create vector to feed into the function
    vec = [x_grid[i], x_grid[j]]
    #evaluate function
    z_grid[i,j] = Rastrigin(vec)
end

#plot and save the surface and contour plots
```

```
plot_3b_surf = Plots.surface(x_grid, x_grid, z_grid, seriescolor=:
    viridis, camera=(25,70))
plot_3b_cont = Plots.contourf(x_grid, x_grid, z_grid, seriescolor=:
    viridis)
savefig(plot_3b_surf, "plot3asurf.png")
savefig(plot_3b_cont, "plot3acont.png")
```

c) A table of initial guesses and algorithm performance is included below. For these guesses, LBFGS finds the global minimum far more consistently than does Nedler-Mead. Furthermore, LBFGS is significantly faster. Code used to generate the table is included below.

Guess	NM Min	NM Iter	LBFGS Min	LBFGS Iter
(0.1, 0.1)	(0, 0)	32	(0, 0)	3
(0.5, 0.5)	(0.99, 0.99)	35	(0, 0)	2
(1.0, 1.0)	(0.99, 0.99)	34	(0, 0)	2
(1.0, 0)	(1, 0)	29	(0, 0)	2
(0, 1.0)	(0, 1.0)	29	(0, 0)	2
(0, 0.5)	(0, 1)	32	(0, 0)	2
(0.5, 0)	(1.0, 0)	32	(0, 0)	2
(2.0, 2.0)	(0, 0)	43	(0, 0)	2
(2.2, 2.2)	(0, 0)	43	(-0.99, -0.99)	4
(3.0, 3.2)	(2.98, 3.98)	35	(3.0, 3.2)	1

Table 2: Guesses, minimizers, and iterations

```
#initialize array of guesses to try
guesses = [[0.1, 0.1], [0.5,0.5], [1,1],[1, 0], [0,1], [0,0.5],
    [0.5,0], [2.0, 2.0], [2.2,2.2], [3.0, 3.2]]

#loop through guesses in array
for guess in guesses
    println(guess)
    #run both optimization algorithms
    opt_nm = optimize(Rastrigin, guess)
    opt_lb = optimize(Rastrigin, guess, LBFGS())
```

```
    #print results
    println("NM",opt_nm.minimizer, opt_nm.iterations)
    println("LBFGS",opt_lb.minimizer, opt_lb.iterations, "\n")
end
```

□

Problem 4.

Solution: Code (and output) for problem 4 is included below. I tested my interpolation function for $f(x) = x^2$ on the interval $[0, 2]$ with 10 grid points at the point $x = 1.3$. The interpolated value is 1.696, whereas the true value of the function at that point is 1.69.

```
function next_highest(x_arr::Vector{Float64}, x::Float64)
    #create boolean array where 1 indicates x is less than y and 0 if y is
     less than x
    compare_arr = [isless(x,y) for y in x_arr]
    #find index of first y such that x < y
    i_first = findfirst(compare_arr)
    #find the corresponding value in x_arr
    x_first = x_arr[i_first]
    x_first, Int(i_first)
end

function lin_approx(f, a::Float64, b::Float64, n::Int64, x::Float64)
    #check if x is lower than a, return a
    if x <= a
        val = f(a)
        return val
    #likewise for b
    elseif x >= b
        val = f(b)
        return val
    else #for values of x in the interval
```

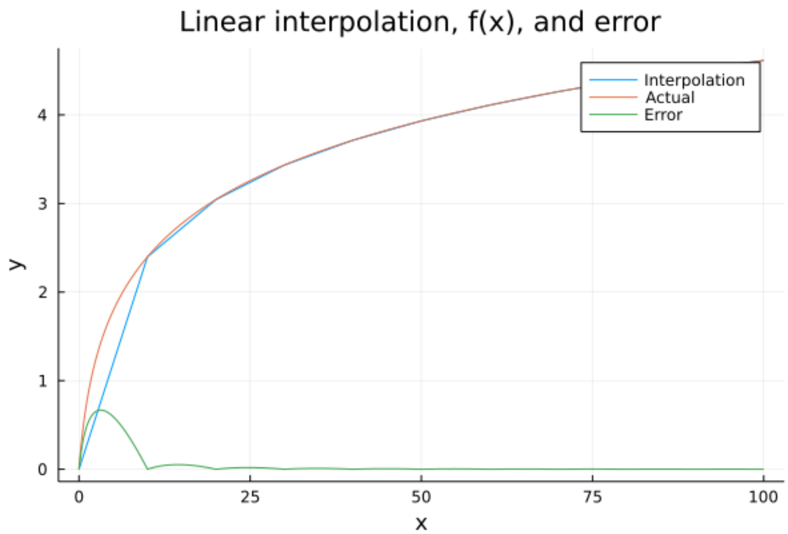
```
        #create domain and empty array for the function values
        x_grid = collect(range(a, length = n , stop= b))
        #fill in the function values
        z_grid = f.(x_grid)
        #find the lowest value in the x grid which is bigger than x, as well as its index
        x_high, i_high = next_highest(x_grid, x)
        #since x_high is the smallest value which is greater than x, the element which comes before it must be less than or equal to x
        x_low = x_grid[Int(i_high-1)]
        #find the weighted average of x_high and x_low to get the interpolated value
        x_interp = ((x-x_low)/(x_high - x_low))*f(x_high) + ((x_high-x)/(x_high - x_low))*f(x_low)
        return x_interp
    end
end

f(x) = x^2

#find interpolation for f at x=1.3 on interval [0,2] with 10 grid points
lin_approx(f, 0.0, 2.0, 10, 1.3)
>> 1.696
```

□

Problem 5.

Solution:

a) Using an evenly-spaced grid, I generated the following plot of the linear interpolation, the actual function values, and the interpolation error. Code is included below the figure:




```
f(x) = log(x+1)
function next_highest(x_arr::Vector{Float64}, x::Float64)
    #create boolean array where 1 indicates x is less than y and 0 if
     y is less than x
    compare_arr = [isless(x,y) for y in x_arr]
    #find index of first y such that x < y
    i_first = findfirst(compare_arr)
    #find the corresponding value in x_arr
    x_first = x_arr[i_first]
    x_first, Int(i_first)
end

function lin_approx(f, x_ar::Vector{Float64}, x::Float64)
    #sort array so that it is ordered
    x_grid = sort(x_ar)
    #check if x lies outside of the boundaries of the grid
    if x <= x_grid[1] #if x is lower than the lowest point in the
    grid:
        val = f(x_grid[1])
        return val
    elseif x >= x_grid[length(x_grid)] #if x is higher than the
    highest point in the grid:
```

```
        val = f(x_grid[length(x_grid)])
        return val
    else #if x is within the grid
        z_grid = f.(x_grid) #fill in the function values at each
    point of the x_grid
        x_high, i_high = next_highest(x_grid, x) #find the value in
    the grid just above x
        x_low = x_grid[Int(i_high-1)] #find the value in the grid
    just below x
        #find the interpolated value using x_low and x_high
        x_interp = ((x-x_low)/(x_high - x_low))*f(x_high) + ((x_high-
    x)/(x_high - x_low))*f(x_low)
        return x_interp
    end
end

function interp_eval(f, x_arr::Vector{Float64})
    #create fine x grid and find its length
    x_fine = collect(0.0:0.01:100.0)
    nx = length(x_fine)
    #create empty grid for interpolated values on the fine grid
    z_fine_approx = zeros(nx)
    #find the actual function values for each point on the fine grid
    z_fine_act = f.(x_fine)
    #loop over x_grid
    for i=1:nx
        #find the interpolated values for each point and record them
    in z_fine_approx
        z_fine_approx[i] = lin_approx(f, x_arr, x_fine[i])
    end
    #find the pointwise interpolation error
    error = z_fine_act - z_fine_approx
    #return the interpolated values, true values, interpolation error
```

```
, and the fine grid used
    z_fine_approx, z_fine_act, error, x_fine
end

#create evenly spaced grid
x_even = [0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
#find interpolation of f using the above grid and store the required variables for plotting
z_fine_approx, z_fine_act, err, x_fine = interp_eval(f, x_even)

using Plots
#create and sove plot
plot_5a = plot(x_fine, [z_fine_approx, z_fine_act, err], labels=["Interpolation" "Actual" "Error"], title="Linear interpolation, f(x), and error", xlabel="x", ylabel="y")
savefig(plot_5a, "plot5a.png")

```

b) Below is my function to take an arbitrary array and return the interpolation error. I chose an arbitrary array to test it and reported the sum of errors for that grid.

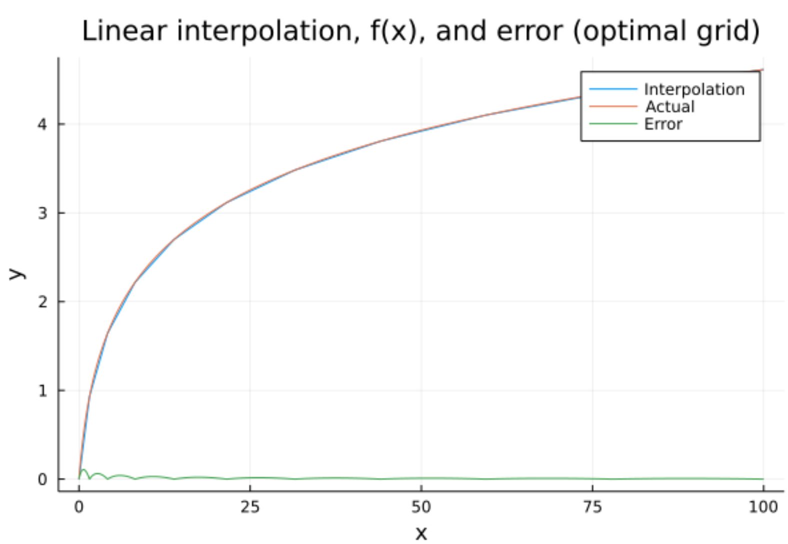
```
function interp_eval_arb(x_arr::Vector{Float64})
    #add 0 and 100 and sort the resulting array so that it is ordered
    x_arr_s = sort( union( 0.0, x_arr, 100.0))
    #for purposes of optimization in part c, return a very high error value if any values of input array are out of bounds
    if x_arr_s[1] < 0.0 || x_arr_s[11] > 100.0
        return 100000
    else
        #create x array
        x_fine = collect(0.0:0.01:100.0)
        nx = length(x_fine)
```

```
        #initialize blank array for interpolated values
        z_fine_approx = zeros(nx)
        #evaluate function for grid points in fine grid
        z_fine_act = f.(x_fine)
        #loop over fine array
        for i=1:nx
            #find interpolated values for each point in array
            z_fine_approx[i] = lin_approx(f, x_arr_s, x_fine[i])
        end
        #find interpolation error
        error = z_fine_act - z_fine_approx
        #find and return sum of the absolute value of interpolation error
        sum_error = sum(abs.(error))
        return sum_error
    end
end

x_arb = [5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 90.0]
#find interpolation error using the above arbitrary array
se = interp_eval_arb(x_arb)
>>419.07

```

c) Below is the result of minimizing the interpolation error using Nedler-Mead. The optimal grid is $(0, 1.54, 4.17, 8.19, 13.89, 21.57, 31.52, 44.05, 59.44, 77.99, 100.0)$. The final error is 110.355, which is down from 504.491 for the evenly-spaced grid. I also include a similar plot to that in problem 5a), and it is clear that this selection of grid points allows the interpolation to fit the actual function far better.

```
using Optim
x_guess = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0]
#find interpolation error using the above evenly-spaced starting
```

```
guess
se_guess = interp_eval_arb(x_guess)
>>504.491
#use Nedler-Mead to find the optimal grid points for interpolation. Our function rejects out-of-bounds guesses by giving a large penalty, and it also sorts the input array so we don't have to worry about checking that the points are in the right order
opt = optimize(interp_eval_arb, x_guess)
#find the optimal grid points from the optimization object
opt_grid = sort(opt.minimizer)
#find the sum of errors associated with optimal grid points
se_opt = interp_eval_arb(opt_grid)
>>110.355

```



Code for plot:

```
#function modified to return the required data for plotting
function interp_eval_arb_plot(x_arr::Vector{Float64})
    x_arr_s = sort( union( 0.0, x_arr, 100.0))
    if x_arr_s[1] < 0.0 || x_arr_s[11] > 100.0
```

```
        return 100000
    else
        z_arr = f.(x_arr_s)
        x_fine = collect(0.0:0.01:100.0)
        nx = length(x_fine)
        z_fine_approx = zeros(nx)
        z_fine_act = f.(x_fine)
        for i=1:nx
            z_fine_approx[i] = lin_approx(f, x_arr_s, x_fine[i])
        end
        error = z_fine_act - z_fine_approx
        sum_error = sum(abs.(error))
        return z_fine_approx, z_fine_act, error, x_fine
    end
end

z_fine_approx, z_fine_act, err, x_fine = interp_eval_arb_plot(opt_grid)
plot_5c = plot(x_fine, [z_fine_approx, z_fine_act, err], labels=["Interpolation" "Actual" "Error"], title="Linear interpolation, f(x), and error (optimal grid)", xlabel="x", ylabel="y")
savefig(plot_5c, "plot5c.png")
```

□