Problem Set 01

Computational Methods in Economics, FGV-EESP

Student: Vinícius de Almeida Nery Ferreira

Professor: Lucas Finamor

TA: Arthur Botinha

Note: this is my first attempt at Julia, so I am going to be commenting A LOT on seemly trivial things.

AI Use: I used ChatGPT to learn basics about Julia and to solve coding errors, as well as to generate markdown code for the tables of the PSET. I did not prompt it to solve any question.

```julia
# Packages
begin
    using Interpolations
    using Distributions
    using Random
    using DataFrames

    using Plots
    using PrettyTables
    using PyFormattedStrings  # Python f-strings

    using BenchmarkTools
end
```

Question 1

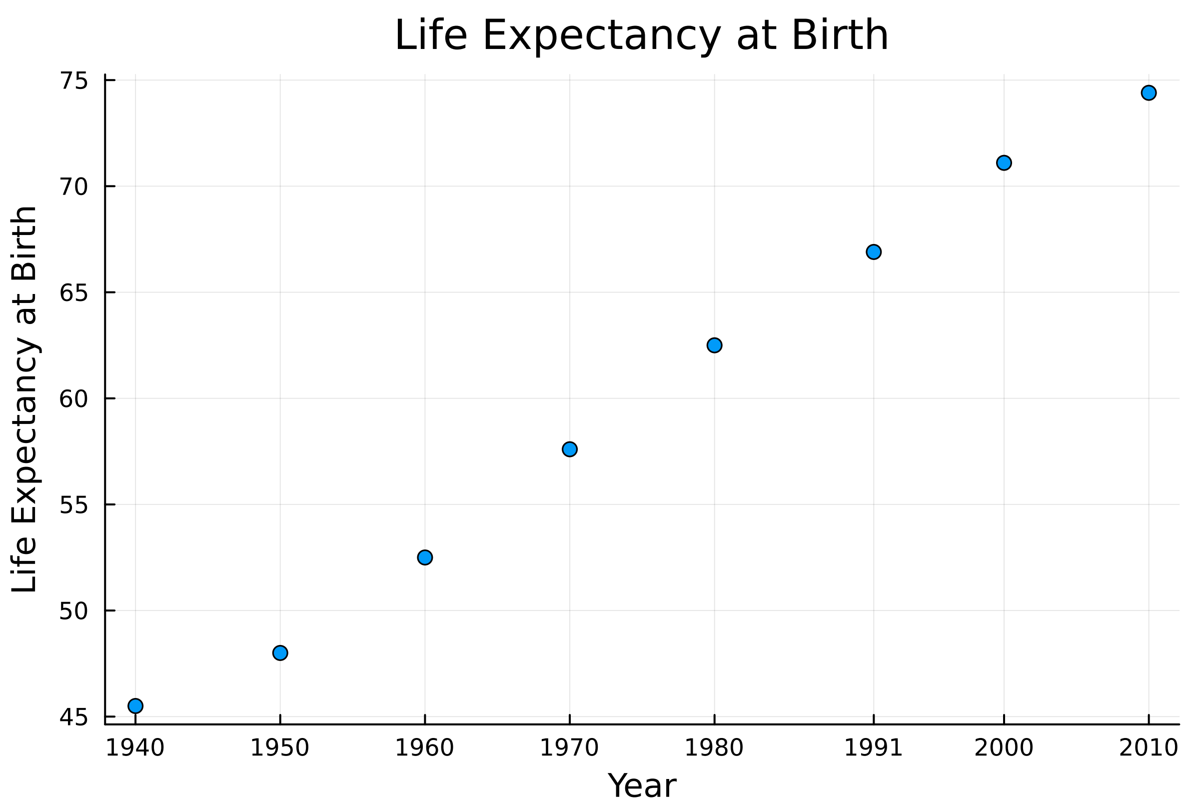
Year	1940	1950	1960	1970	1980	1991	2000	2010
IBGE Life Expectancy	45.5	48.0	52.5	57.6	62.5	66.9	71.1	74.4

Source

(a) Best Guess for 1996 Life Expectancy

```julia
# First, we need to store the data.
## We will do so in a dictionary with vectors, which in Julia are defined by []
## Similar to Python (() also defines tuples)
life_expectancy = Dict(
    "year" => [1940, 1950, 1960, 1970, 1980, 1991, 2000, 2010],
    "expectancy" => [45.5, 48.0, 52.5, 57.6, 62.5, 66.9, 71.1, 74.4]
);
```

It is always useful to plot our data:



We have some pretty linear behavior, specially starting in 1950. However, we get a little concavity over time, so will choose to go for a Pchip, as it preservs monotonicity.

Because the grid is not evenly spaced (thanks 1991), working with splines is a bit more complicated than the convenience constructors.

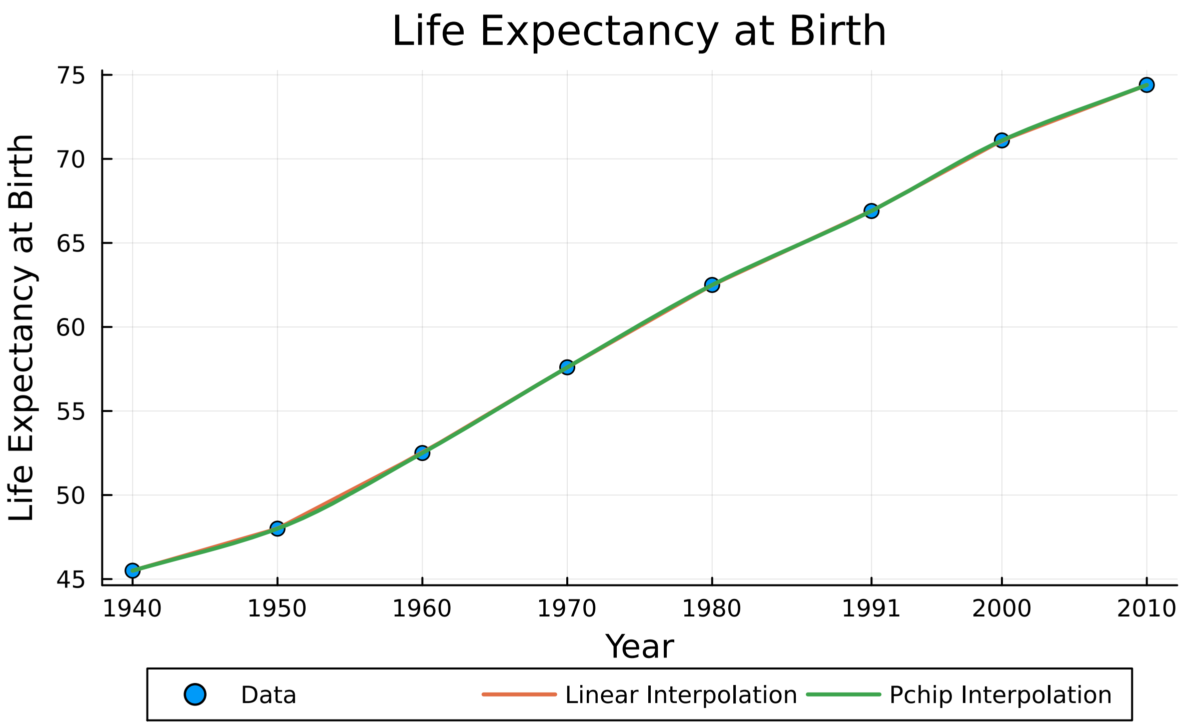
```
begin
    # Creating interpolation object (using ; to avoid printing)
    linear_interp_life_exp = linear_interpolation(Int64.(life_expectancy["year"]),
    life_expectancy["expectancy"])

    pchip_interp_life_exp = interpolate(Int64.(life_expectancy["year"]),
    life_expectancy["expectancy"], FritschButlandMonotonicInterpolation())
end;
```

With the object, we can now have the guess for the value of 1996:

```
"Guess for Life Expectancy at 1996 is 69.3 years with Pchip interpolation."
```

(b) Plot



Question 2

We are interested in the function

$$f(x) = \frac{(x+\varepsilon)^{1-\sigma}}{1-\sigma}$$

where $\sigma = 3$ and $\varepsilon = 10^{-8}$.

```
# Creating the function
function func_question2(x, σ, ε)
    return (x + ε)^(1 - σ) / (1 - σ)
end;
```

```
# Assigning parameters
begin
    σ = 3.0
    ε = 10e-8
end;
```

```
# Since we are going to be working with log grids (which are not evenly unspaced and thus do not work with the convenience constructor of cubic spline), will be using a different package (yes I'm that lazy)
using DataInterpolations
```

(a) Random Draws

```
begin
    # Drawing random uniforms
    ## First, setting seed
    Random.seed!(121019)

    ## Drawing (sort! sorts inplace, as ! modifies objects in Julia)
    number_samples = 2500
    x = sort!(rand(Uniform(0, 10), number_samples))
end;
```

(b) Creating a Grid

Here, we will try to create a function to encompass itens (b)-(d), as we will be repeating them 3 times.

In Julia, `;` separates keyword function arguments. I always like to name the arguments when passing, so everything will be keywords.

```julia
function evaluate_interpolation(; func::Function, grid_t::AbstractArray,
x::AbstractArray, param1::Number, param2::Number)
    # Computing the function, vectorizing using .
    ## At grid points
    y = func.(grid_t, param1, param2)

    ## At all points x
    f_x = func.(x, param1, param2)

    # Dataframe to store results
    df_mse = DataFrame(
        "Method" => ["Linear", "Cubic Spline", "Pchip"],
        "MSE" => [0.0, 0.0, 0.0],
        "Average Time (μs)" => [0, 0, 0]
    )

    # Looping across methods
    ## To display the average time, will do 10,000 repetitions
    ## Could use benchmark and display it on separate cells, but chose to keep
    everything inside this function and put it directly in the table
    times = zeros(10000)

    for (index, interp_method) in enumerate([
        DataInterpolations.LinearInterpolation,
        DataInterpolations.CubicSpline,
        DataInterpolations.PCHIPInterpolation])

        # Initializing mse variable so we can access outside the following loop
        # Note that, in each repetition, the MSE is the same
        mse = 0.0

        for repetition in 1:10000
            # Calculating interpolation and error and time
            time = @elapsed begin
                # Interpolation object
                interp = interp_method(y, grid_t)

                # Interpolation on all points
                f_hat_x = interp.(x)

                # Error
                error = f_x - f_hat_x

                # MSE (using * instead of ^ as it is faster; . vectorizes)
                mse = mean(error .* error)
            end

            # Appending time
            times[repetition] = time
        end

        # Appending to DataFrame (time as integer)
        df_mse[index, "MSE"] = mse
        df_mse[index, "Average Time (μs)"] = trunc(Int, 10e6 * mean(times))
    end

    return df_mse
end;
```

```julia
# Creating the grid from 0 to 10 using 10 points
grid_t = collect(range(start = 0, stop = 10, length = 10));
```

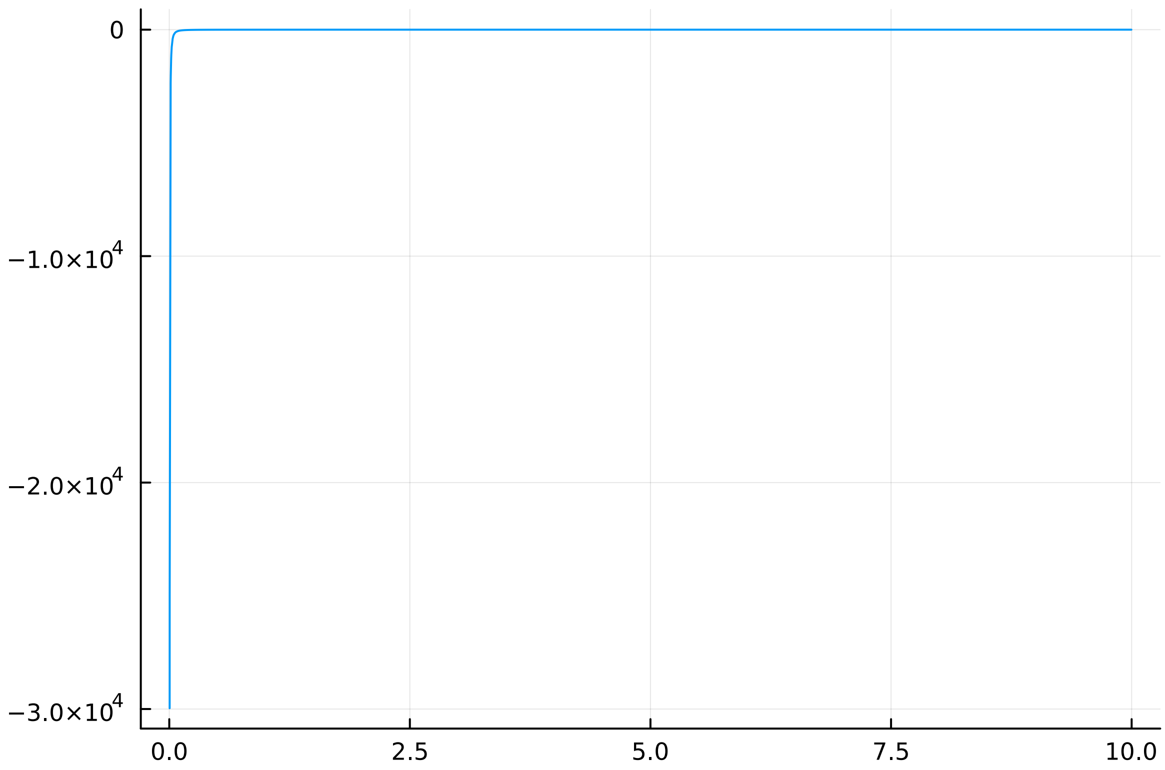
(c) Results with Linear Grid and 10 Points

```julia
# Calling function and displaying results (will omit the rest of coding cells and only show output from hereon)
results_q2_linear_grid_10points = evaluate_interpolation(func = func_question2, grid_t = grid_t, x = x, param1 = σ, param2 = ε);
```

	Method	MSE	Average Time (μs)
1	"Linear"	9.42164e25	2582
2	"Cubic Spline"	7.77873e25	2081
3	"Pchip"	5.41226e25	2478

We see that the error is absolutely huge, but this is because of the shape of our function (which we plot below). PChip has the smallest error, while Linear has the largest.

Moreover, we see that there are little differences in average run times, but cubic spline seems to be the slowest, while linear interpolation is the fastest.



(d) Results with More Points

With $n = 15$,

	Method	MSE	Average Time (µs)
1	"Linear"	5.79726e25	2146
2	"Cubic Spline"	4.76671e25	2663
3	"Pchip"	3.32436e25	2577

With $n = 20$,

	Method	MSE	Average Time (µs)
1	"Linear"	4.09689e25	3989
2	"Cubic Spline"	3.39643e25	2943
3	"Pchip"	2.41199e25	3268

With $n = 30$,

	Method	MSE	Average Time (µs)
1	"Linear"	2.60646e25	2670
2	"Cubic Spline"	2.20878e25	2993
3	"Pchip"	1.59853e25	2964

With $n = 50$,

	Method	MSE	Average Time (μs)
1	"Linear"	1.58068e25	3115
2	"Cubic Spline"	1.34497e25	3139
3	"Pchip"	9.75326e24	3767

As expected, when we increase the number of error points, the MSE falls, although it is still large in magnitude.

(e) Log-Spaced Grids

```
# Creating function that makes the log grid
function log_grid(initial, final, n)
    # Have to check if initial is <= 0
    if initial <= 0
        initial = .0001
    end

    return exp.(range(log(initial), log(final), length=n))
end;
```

With $n = 10$,

	Method	MSE	Average Time (μs)
1	"Linear"	6.76422e5	2230
2	"Cubic Spline"	4.94135e16	1898
3	"Pchip"	10061.7	1666

With $n = 15$,

	Method	MSE	Average Time (μs)
1	"Linear"	2.01538e5	2562
2	"Cubic Spline"	7.89621e11	2989
3	"Pchip"	32030.7	3236

With $n = 20$,

	Method	MSE	Average Time (μs)
1	"Linear"	10995.4	2064
2	"Cubic Spline"	1.00897e8	2326
3	"Pchip"	2694.84	2492

With $n = 30$,

	Method	MSE	Average Time (μs)
1	"Linear"	7310.72	2577
2	"Cubic Spline"	9313.06	2570
3	"Pchip"	298.166	3177

With $n = 50$,

	Method	MSE	Average Time (μs)
1	"Linear"	470.198	1956
2	"Cubic Spline"	0.127214	1749
3	"Pchip"	6.19932	1835

```
begin
    results_q2_log_grid_50points = evaluate_interpolation(func = func_question2,
    grid_t = log_grid(0, 10, 50), x = x, param1 = σ, param2 = ε)

    results_q2_log_grid_50points
end
```

(f) Conclusions

With the log grid, which is denser at small values, our errors decrease by orders of magnitude! This is because our function is very concave, and so placing more points at areas with larger curvature helps the interpolation procedure.

In terms of performance, we see a difference based on whether the grid is linear or log-spaced, which is intriguing since the grids are calculated outside of the function.

In the linear grid, all methods alternate in terms of best performance. This is striking, since we expect that the linear interpolation method be the fastest.

However, in the log grids, this prediction holds true (more or less): Linear interpolation is often the fastest, followed by PChip and the Cubic Spline. It is worth noting that the difference is only around 100μs on average.

To confirm this timing result, will also do some tests with benchmark (with $n = 50$).

```julia
function mse_interpolation(; func::Function, grid_t::AbstractArray, x::AbstractArray, param1::Number, param2::Number, interp_method)
    # Computing the function, vectorizing using .
    ## At grid points
    y = func.(grid_t, param1, param2)

    ## At all points x
    f_x = func.(x, param1, param2)

    # Dataframe to store results
    df_mse = DataFrame(
        "Method" => ["Linear", "Cubic Spline", "Pchip"],
        "MSE" => [0.0, 0.0, 0.0],
        "Average_Time" => [0, 0, 0]
    )

    # Interpolation object
    interp = interp_method(y, grid_t)

    # Interpolation on all points
    f_hat_x = interp.(x)

    # Error
    error = f_x - f_hat_x

    # MSE (using .* because it is faster than .^2)
    mse = mean(error .* error)

    # Just assigning to df so that is more comparable with the other function
    df_mse[1, "MSE"] = mse

    return mse
end;
```

Linear Grids

Linear:

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation per sample.
 Range (min … max):  158.316 μs … 147.360 ms  ┊ GC (min … max): 0.00% … 99.65%
 Time  (median):     300.649 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   471.847 μs ±   2.129 ms  ┊ GC (mean ± σ):  5.84% ±  1.41%
```

```
  158 μs        Histogram: log(frequency) by time       4.31 ms <

 Memory estimate: 83.25 KiB, allocs estimate: 99.
```

Cubic Spline:

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation per sample.
 Range (min … max):  166.232 μs … 113.289 ms  ┊ GC (min … max): 0.00% … 99.56%
 Time  (median):     296.159 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   452.971 μs ±   1.806 ms  ┊ GC (mean ± σ):  6.23% ±  1.72%
```

```
  166 μs        Histogram: log(frequency) by time        2.8 ms <

 Memory estimate: 92.11 KiB, allocs estimate: 145.
```

PChip:

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation per sample.
 Range (min … max):  156.727 μs … 136.942 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     254.275 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   476.160 μs ±   2.326 ms  ┊ GC (mean ± σ):  4.76% ± 1.41%
```

```
  157 μs        Histogram: log(frequency) by time       3.97 ms <

 Memory estimate: 85.67 KiB, allocs estimate: 109.
```

Log Grids

Linear:

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation per sample.
 Range (min … max):  118.559 μs … 161.354 ms  ┊ GC (min … max): 0.00% … 99.85%
 Time  (median):     203.736 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   365.448 μs ±   2.216 ms  ┊ GC (mean ± σ):  8.13% ±  1.41%
```

```
  119 μs        Histogram: log(frequency) by time       4.34 ms <

 Memory estimate: 83.25 KiB, allocs estimate: 99.
```

Cubic Spline:

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation per sample.
 Range (min … max):  139.250 μs … 140.344 ms  ┊ GC (min … max): 0.00% … 99.37%
 Time  (median):     230.918 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   398.047 μs ±   1.847 ms  ┊ GC (mean ± σ):  5.78% ±  1.41%
```

```
  139 μs         Histogram: log(frequency) by time       4.51 ms <

 Memory estimate: 92.11 KiB, allocs estimate: 145.
```

Pchip:

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation per sample.
 Range (min … max):  142.201 μs … 176.575 ms  ┊ GC (min … max): 0.00% … 99.81%
 Time  (median):     249.601 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   412.041 μs ±   2.336 ms  ┊ GC (mean ± σ):  7.21% ±  1.41%
```

```
  142 μs         Histogram: log(frequency) by time       3.07 ms <

 Memory estimate: 85.67 KiB, allocs estimate: 109.
```

Results hold: looking at both the median and the mean, the linear is the slowest of them in case of the linear grid (the same result we got with $n = 50$), while it is the fastest using log-spaced grids.

I suspect this has something to do with its larger MSEs, which slows down allocation in the dataframe. Don't know though...

Making use of Pluto's Interactivity

As a test, will try to make the function interactive in the number of points using PlutoUI.

```
using PlutoUI
```

Choose the number of points...

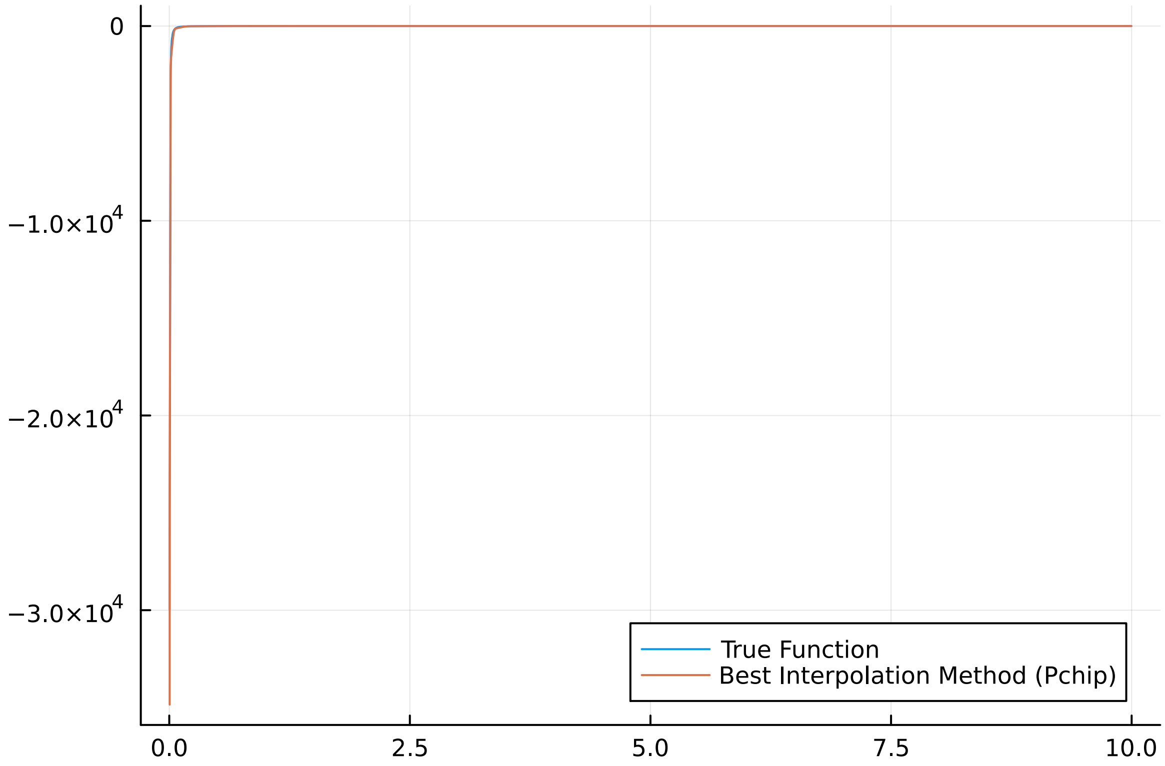
```
"Number of grid points: 10"
```

...and the type of grid...

log

... and see the magic happen!

	Method	MSE	Average Time (μs)
1	"Linear"	6.76422e5	1225
2	"Cubic Spline"	4.94135e16	1139
3	"Pchip"	10061.7	1251



Question 3

We now have the function

$$g(x) = \frac{1}{1 + \exp\left[k(-x + \xi)\right]}$$

where $k = 2.5$, $\xi = 5$ and $x \in [1, 10]$.

```
# Creating the function
function func_question3(x, k, ξ)
    return 1 / (1 + exp(k * (-x + ξ)))  # avoiding using ^(-1)
end;
```

```
# Assigning parameters
begin
    k = 2.5
    ξ = 5
end;
```

```
# Drawing random uniforms from [1, 10]
x_q3 = sort!(rand(Uniform(1, 10), number_samples));
```

Repeating Question 2

Linear Grids

With $n = 10$,

	Method	MSE	Average Time (μs)
1	"Linear"	0.000503557	3447
2	"Cubic Spline"	0.000122881	3782
3	"Pchip"	9.04105e-5	3419

With $n = 15$,

	Method	MSE	Average Time (μs)
1	"Linear"	7.09472e-5	3778
2	"Cubic Spline"	1.77892e-6	4591
3	"Pchip"	5.6738e-6	3409

With $n = 20$,

	Method	MSE	Average Time (μs)
1	"Linear"	2.18288e-5	3998
2	"Cubic Spline"	5.91472e-8	3335
3	"Pchip"	9.85413e-7	3525

With $n = 30$,

	Method	MSE	Average Time (μs)
1	"Linear"	4.12796e-6	4102
2	"Cubic Spline"	1.23897e-9	3775
3	"Pchip"	7.86532e-8	4619

With $n = 50$,

	Method	MSE	Average Time (μs)
1	"Linear"	5.19299e-7	3358
2	"Cubic Spline"	9.77902e-12	3261
3	"Pchip"	3.20806e-9	3083

Log Grids

With $n = 10$,

	Method	MSE	Average Time (µs)
1	"Linear"	0.000851208	2073
2	"Cubic Spline"	0.000722996	2612
3	"Pchip"	0.000227089	2878

With $n = 15$,

	Method	MSE	Average Time (µs)
1	"Linear"	0.000244346	2291
2	"Cubic Spline"	2.5363e-5	2859
3	"Pchip"	3.48728e-5	3205

With $n = 20$,

	Method	MSE	Average Time (µs)
1	"Linear"	6.46422e-5	2404
2	"Cubic Spline"	6.05927e-7	2774
3	"Pchip"	5.26742e-6	2981

With $n = 30$,

	Method	MSE	Average Time (µs)
1	"Linear"	1.23025e-5	2492
2	"Cubic Spline"	1.55735e-8	2907
3	"Pchip"	4.20402e-7	2971

With $n = 50$,

	Method	MSE	Average Time (μs)
1	"Linear"	1.56318e-6	2464
2	"Cubic Spline"	1.11514e-10	2614
3	"Pchip"	1.97013e-8	3140

(a) Different Conclusions

Now, we see that the performance with the linear grid is slightly better than with the log grid, although it is slower. To make it easier to see, we will show the tables for $n = 50$ below each other:

With a linear grid:

	Method	MSE	Average Time (μs)
1	"Linear"	5.19299e-7	3358
2	"Cubic Spline"	9.77902e-12	3261
3	"Pchip"	3.20806e-9	3083

With a log grid:

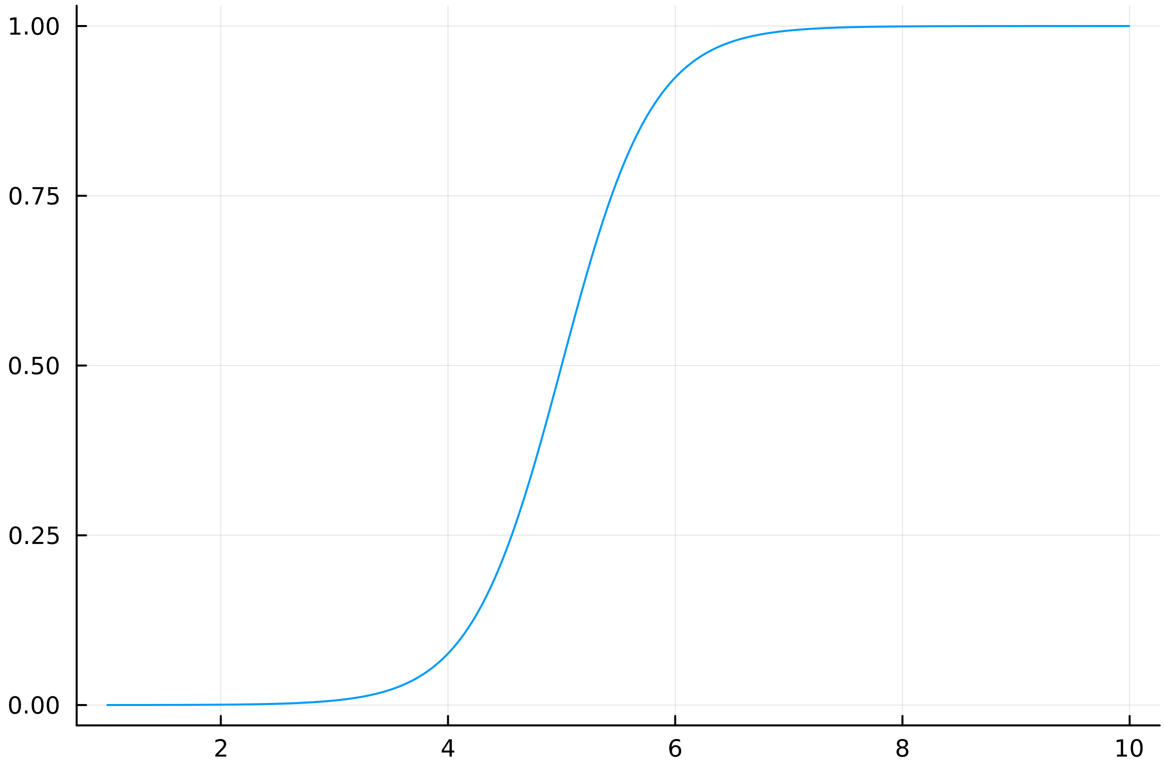
	Method	MSE	Average Time (μs)
1	"Linear"	1.56318e-6	2464
2	"Cubic Spline"	1.11514e-10	2614
3	"Pchip"	1.97013e-8	3140

Moreover, we see that the spline does slightly better in this case relative to the Pchip, but it is slightly slower than the other two methods.

(b) Best Grid

The curvature of the function depending on whether x is greater or less than ξ: if it is greater, $\exp(k(-x + \xi)) < 1$ and the function will be concave; if x is less than ξ, the function will be convex.

Thus, $g(x)$ is S-shaped, as we show below:



Therefore, the best shaped grid would be one that is denser in the middle of the interval, as it is the area with the larger curvature.

Question 4

Friend	Human Capital in HS	College	Wage
A	1.0	0	1.00
B	1.5	0	1.50
C	3.1	1	4.22
D	4.5	1	25.50

Maria has a human capital $h = 2.8$, and will decide on whether to go to college or not based on her friends.

(a) Nearest-Neighbour Interpolation Decision

```
# First, we need to create the data; will follow what we did in Question 1
college_decision = Dict(
    "h" => [1.0, 1.5, 3.1, 4.5],
    "college" => Int32.([0, 0, 1, 1]),
    "wage" => [1.0, 1.5, 4.22, 25.50]
);
```

```
# Interpolation object
interp_nn_q4 = constant_interpolation(college_decision["h"], college_decision["college"]);
```

```
"Using NN interpolation, Maria WILL go to college."
```

```
# Finding college decision
begin
    maria_college_decision = interp_nn_q4(maria_h)
    college_decision_string = ifelse(maria_college_decision == 1, "WILL", "WILL NOT")

    f"Using NN interpolation, Maria {college_decision_string} go to college."
end
```

(b.1) Wage with Linear Interpolation

```
# Interpolation object
interp_linear_wage_q4 = linear_interpolation(college_decision["h"], college_decision["wage"]);
```

"Using linear interpolation, Maria's wage estimate is 3.7."

```
# Finding wage
f"Using linear interpolation, Maria's wage estimate is
{interp_linear_wage_q4(maria_h):.1f}."
```

Now, Maria learned new information: she is now a econometrician doing simulations and thus knows the counterfactual wages of her friends.

Friend	Human Capital in HS	College	Wage if HS	Wage if College
A	1.0	0	1.00	-13.00
B	1.5	0	1.50	-10.50
C	3.1	1	3.10	4.22
D	4.5	1	4.50	25.50

```
# Creating the data
# First, we need to create the data; will follow what we did in Question 1
college_decision_cf = Dict(
    "h" => [1.0, 1.5, 3.1, 4.5],
    "college" => Int32.([0, 0, 1, 1]),
    "wage_hs" => [1.0, 1.5, 3.1, 4.50],
    "wage_col" => [-13, -10.5, 4.22, 25.50]
);
```

(b.2) Estimates of Wages

```
# Interpolation objects
begin
    interp_linear_wage_hs_q4 = linear_interpolation(college_decision_cf["h"],
    college_decision_cf["wage_hs"])

    interp_linear_wage_col_q4 = linear_interpolation(college_decision_cf["h"],
    college_decision_cf["wage_col"])
end;
```

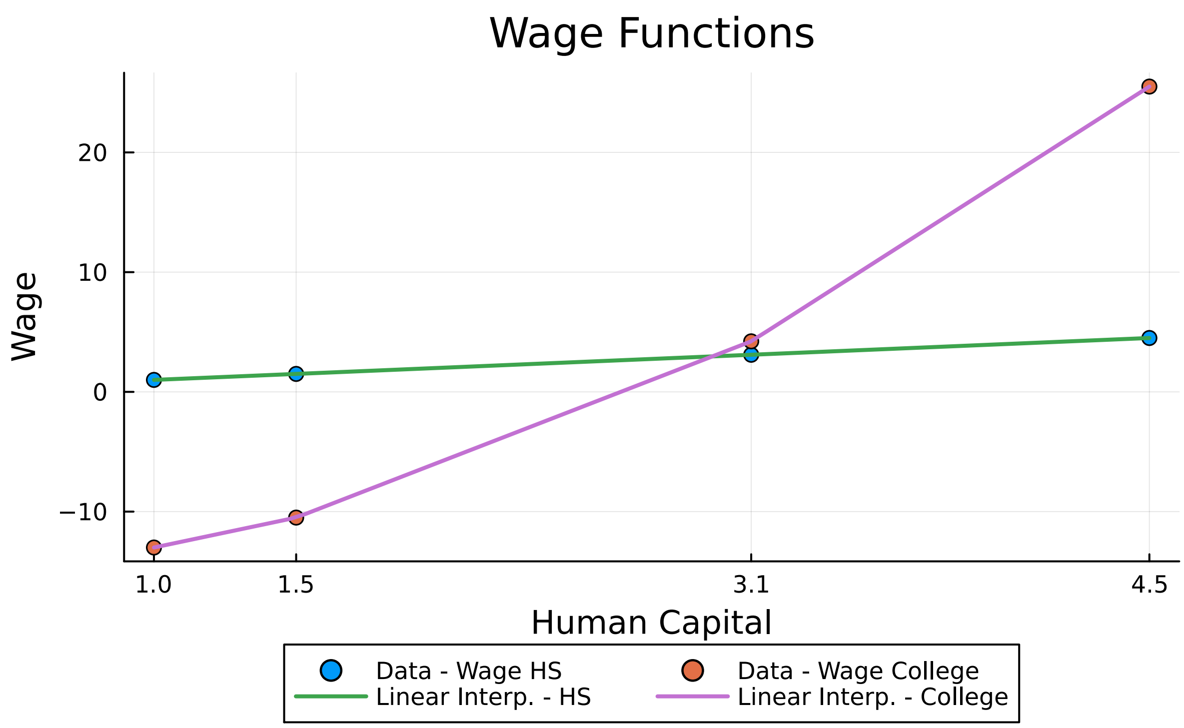
"Using linear interpolation, Maria's wage estimate if she stays in HS is 2.8."

```
# Finding wage if HS
f"Using linear interpolation, Maria's wage estimate if she stays in HS is
{interp_linear_wage_hs_q4(maria_h):.1f}."
```

"Using linear interpolation, Maria's wage estimate if she goes to college is 1.5."

```
# Finding wage if college
f"Using linear interpolation, Maria's wage estimate if she goes to college is
{interp_linear_wage_col_q4(maria_h):.1f}."
```

(c) Plot of Wage Functions



(d) Lowest Human Capital to Go to College - NN College Decision Interpolation

Based on NN interpolation, the lowest value of human capital that would make Maria go to college is the midpoint between h_B and h_C, which is

$$h_{min} = \frac{h_B + h_C}{2} = 2.3$$

(e) Lowest Human Capital to Go to College - Linear Wage Functions

Based on the linear interpolation of wage functions, Maria goes to college if her estimated wage going to college is higher than the estimated wage staying in high school.

This happens if

$$w_B^{HS} + \frac{(w_C^{HS} - w_B^{HS})}{h_C - h_B}(h - h_B) = w_B^{Col} + \frac{(w_C^{Col} - w_B^{Col})}{h_C - h_B}(h - h_B)$$

$$1.5 + \frac{(3.1 - 1.5)}{3.1 - 1.5}(h - 1.5) = -10.5 + \frac{(4.22 - (-10.5))}{3.1 - 1.5}(h - 1.5)$$

$$10.5 + h = \frac{14.72}{1.6}(h - 1.5)$$

$$1.6h + 16.8 = 14.72h - 22.08$$

$$13.12h = 38.88$$

$$h = 2.9634$$

We could also find this value with a root finding algorithm, such as bisection:

```
using Roots
```

```
function diff_wage_functions(x)
    return interp_linear_wage_hs_q4(x) - interp_linear_wage_col_q4(x)
end;
```

```
# Finding 0
min_h_go_to_col = find_zero(diff_wage_functions, (college_decision_cf["h"][2], college_decision_cf["h"][3]), Bisection());
```

```
# Printing
begin
    maria_output_hc_string = f"Minimum human capital such that Maria goes to college based \n on linear interpolation of wage functions: {min_h_go_to_col:.4f}"

    println(maria_output_hc_string)
end
```

```
Minimum human capital such that Maria goes to college based
 on linear interpolation of wage functions: 2.9634
```