Discrete Dynamical Systems

Problem 1: A Ton of New Facts on Newton

In this problem we will look into Newton's method. Newton's method is the dynamical system defined by the update process:

$$x_{n+1} = x_n - \left(\frac{dg}{dx}(x_n)\right)^{-1} g(x_n)$$

For these problems, assume that $\frac{dg}{dx}$ is non-singular.

Part 1

Show that if x^* is a steady state of the equation, then $g(x^*) = 0$.

Ans:

If x^* is a steady state of the equation, then

$$x^* = x^* - \left(\frac{dg}{dx}(x^*)\right)^{-1} g(x^*) \implies \left(\frac{dg}{dx}(x^*)\right)^{-1} g(x^*) = 0$$

Since $\frac{dg}{dx}$ is non-singular, it rules out the possibility that $\left(\frac{dg}{dx}(x^*)\right)^{-1} = 0$. Therefore, $g(x^*) = 0$.

Part 2

Take a look at the Quasi-Newton approximation:

$$x_{n+1} = x_n - \left(\frac{dg}{dx}(x_0)\right)^{-1} g(x_n)$$

for some fixed x_0. Derive the stability of the Quasi-Newton approximation in the form of a matrix whose eigenvalues need to be constrained. Use this to argue that if x_0 is sufficiently close to x^* then the steady state is a stable (attracting) steady state.

Ans:

Let J_n be the Jacobian of g at x_n and J_0 be the Jacobian of g at x_0.
The Jacobian of the function $x_n - \left(\frac{dg}{dx}(x_0)\right)^{-1} g(x_n)$ is $I - Jo^{-1}J_n$. Therefore, Quasi-Newton approximation is stable if all the eigenvalues of the matrix $I - Jo^{-1}J_n$ have absolute value < 1.

If x_0 is sufficiently close to x^*, then $J_0 \approx J_n \approx J^*$. We can then assume that $Jo^{-1}J_n$ is very close to the identity matrix. More specifically, $Jo^{-1}J_n = I + \mathcal{O}(|x^* - x_0|)$, where $|x^* - x_0|$ is small.

$$\implies I - Jo^{-1}J_n = I - (I + \mathcal{O}(|x^* - x_0|)) = \mathcal{O}(|x^* - x_0|)$$

Since the eigenvalues of a small matrix are small, and $\mathcal{O}(|x^* - x_0|)$ is small, we can imply that the eignevalues of $I - Jo^{-1}J_n$ have absolute values smaller than 1. Therefore, if x_0 is sufficiently close to x^* then the steady state is a stable (attracting) steady state.

Part 3

Relaxed Quasi-Newton is the method:

$$x_{n+1} = x_n - \alpha\left(\frac{dg}{dx}(x_0)\right)^{-1} g(x_n)$$

Argue that for some sufficiently small α that the Quasi-Newton iterations will be stable if the eigenvalues of $(\left(\frac{dg}{dx}(x_0)\right)^{-1} g(x_n))'$ are all positive for every x.

(Technically, these assumptions can be greatly relaxed, but weird cases arise. When $x \in \mathbb{C}$, this holds except on some set of Lebesgue measure zero. Feel free to explore this.)

Ans:

The Jacobian of relaxed quasi-Newton method is $I - \alpha J_0^{-1}J_n$. If the eigenvalues λ_i of $(\left(\frac{dg}{dx}(x_0)\right)^{-1} g(x_n))'$ = $Jo^{-1}J_n$ are all positive for every x, then for some sufficiently small α, we can have all the eigenvalues of $\alpha J_0^{-1}J_n$ satisfy $0 < \alpha\lambda_i < 1$. The eigenvalues of $I - \alpha J_0^{-1}J_n$ are $1 - \alpha\lambda_i$, which also satisfy $0 < 1 - \alpha\lambda_i < 1$. Therefore, the eigenvalues of the Jacobian $I - \alpha J_0^{-1}J_n$ have absolute value < 1, and thus the Quasi-Newton iterations will be stable.

Part 4

Fixed point iteration is the dynamical system

$$x_{n+1} = g(x_n)$$

which converges to $g(x) = x$.

1. What is a small change to the dynamical system that could be done such that $g(x) = 0$ is the steady state?
2. How can you change the $\left(\frac{dg}{dx}(x_0)\right)^{-1}$ term from the Quasi-Newton iteration to get a method equivalent to fixed point iteration? What does this imply about the difference in stability between Quasi-Newton and fixed point iteration if $\frac{dg}{dx}$ has large eigenvalues?

Ans:

1. A small change that we can make is to turn the dynamical system into $x_{n+1} = g(x_n) + x_n$. Then, in the steady state, we have $g(x) + x = x$, which is $g(x) = 0$.
2. We can change the $\left(\frac{dg}{dx}(x_0)\right)^{-1}$ term from the Quasi-Newton iteration into the identity matrix I to get a method equivalent to fixed point iteration.

   In Quasi-Newton, we have $x_{n+1} = x_n - \left(\frac{dg}{dx}(x_0)\right)^{-1} g(x_n)$, with Jacobian $I - J_0^{-1}J_n$.

   In fixed point iteration, we have $x_{n+1} = x_n - g(x_n)$, with Jacobian $I - J_n$.

   Suppose the case that $\frac{dg}{dx}$ has large eigenvalues:

   In Quasi-Newton method, if we choose x_0 sufficiently close to x^*, then $J_0 \approx J_n$, and $J_0^{-1}J_n \approx I$. The Jacobian $I - J_0^{-1}J_n$ still has small eigenvalues. So, despite the fact that $\frac{dg}{dx}$ has large eigenvalues, the system is stable if we choose x_0 sufficiently close to x^*.

   In fixed point iteration, $I - J_n$ has large eigenvalues (in absolute value) if $\frac{dg}{dx}$ has large eigenvalues, so the system is not stable.

   Therefore, Quasi-Newton is more stable than fixed point iteration. If we want to make the fixed point iteration becomes stable, we can add a sufficiently small α such that $x_{n+1} = x_n - \alpha g(x_n)$, then this can be stable from part 3.

Problem 2: The Root of all Problems

In this problem we will practice writing fast and type-generic Julia code by producing an algorithm that will compute the quantile of any probability distribution.

Part 1

Many problems can be interpreted as a rootfinding problem. For example, let's take a look at a problem in statistics. Let X be a random variable with a cumulative distribution function (CDF) of $cdf(x)$. Recall that the CDF is a monotonically increasing function in $[0, 1]$ which is the total probability of $X < x$. The yth quantile of X is the value x at with X has a y% chance of being less than x. Interpret the problem of computing an arbitrary quantile y as a rootfinding problem, and use Newton's method to write an algorithm for computing x.

(Hint: Recall that $cdf'(x) = pdf(x)$, the probability distribution function.)

Ans:

See part 2.

Part 2

Use the types from Distributions.jl to write a function `my_quantile(y,d)` which uses multiple dispatch to compute the yth quantile for any `UnivariateDistribution` `d` from Distributions.jl. Test your function on `Gamma(5, 1)`, `Normal(0, 1)`, and `Beta(2, 4)` against the `Distributions.quantile` function built into the library.

(Hint: Have a keyword argument for x_0, and let its default be the mean or median of the distribution.)

In [1]:
```
using Distributions

function my_quantile(d, y; x₀=mean(d))

    tolerance = eps()
    g(x) = cdf(d, x) - y
    g_prime(x) = pdf(d, x)
    x_new = x₀ - g(x₀)/g_prime(x₀)
    x_old = x₀

    if y == 0
        return minimum(d)

    elseif y == 1
        return maximum(d)

    elseif 0 < y && y < 1
        while abs(x_new - x_old) > tolerance
            x_old = x_new
            x_new = x_old - g(x_old)/g_prime(x_old)
        end
        return x_new

    else
        return NaN

    end
end
```

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

In [2]:
```
# test
print(my_quantile(Gamma(5, 1), 0.8765), "\n")
print(quantile(Gamma(5, 1), 0.8765))
```

```
7.620780843744852
7.6207808437448525
```

In [3]:
```
print(my_quantile(Beta(2, 4), 0.1), "\n")
print(quantile(Beta(2, 4), 0.1))
```

```
0.11223495854585859
0.11223495854585858
```

In [4]:
```
print(my_quantile(Normal(0,1), 0.39), "\n")
print(quantile(Normal(0,1), 0.39))
```

```
-0.27931903444745404
-0.27931903444745415
```

In [5]:
```
print(my_quantile(Normal(0,1), 0), "\n")
print(quantile(Normal(0,1), 0))
```

```
-Inf
-Inf
```

In [6]:
```
print(my_quantile(Beta(2, 4), 1), "\n")
print(quantile(Beta(2, 4), 1))
```

```
1.0
1.0
```

In [7]:
```
print(my_quantile(Gamma(5, 1), 10), "\n")
print(quantile(Gamma(5, 1), 10))
```

```
NaN
DomainError with -10.258317557170095:
log will only return a complex result if called with a complex argument. Try log(Complex(x)).

Stacktrace:
  [1] throw_complex_domainerror(f::Symbol, x::Float64)
    @ Base.Math ./math.jl:33
  [2] _log(x::Float64, base::Val{:e}, func::Symbol)
    @ Base.Math ./special/log.jl:304
  [3] log
    @ ./special/log.jl:269 [inlined]
  [4] gamma_inc_inv_qsmall(a::Float64, q::Float64, qgammaxa::Float64)
    @ SpecialFunctions ~/.julia/packages/SpecialFunctions/CQMHW/src/gamma_inc.jl:784
  [5] __gamma_inc_inv(a::Float64, minpq::Float64, pcase::Bool)
    @ SpecialFunctions ~/.julia/packages/SpecialFunctions/CQMHW/src/gamma_inc.jl:1027
  [6] _gamma_inc_inv
    @ ~/.julia/packages/SpecialFunctions/CQMHW/src/gamma_inc.jl:1012 [inlined]
  [7] gamma_inc_inv
    @ ~/.julia/packages/SpecialFunctions/CQMHW/src/gamma_inc.jl:994 [inlined]
  [8] gammainvcdf
    @ ~/.julia/packages/StatsFuns/dTYga/src/distrs/gamma.jl:98 [inlined]
  [9] quantile(d::Gamma{Float64}, q::Int64)
    @ Distributions ~/.julia/packages/Distributions/Ooh1r/src/univariates.jl:627
 [10] top-level scope
    @ In[7]:2
 [11] eval
    @ ./boot.jl:373 [inlined]
 [12] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
    @ Base ./loading.jl:1196
```

Problem 3: Bifurcating Data for Parallelism

In this problem we will write code for efficient generation of the bifurcation diagram of the logistic equation.

Part 1

The logistic equation is the dynamical system given by the update relation:

$$x_{n+1} = rx_n(1 - x_n)$$

where r is some parameter. Write a function which iterates the equation from $x_0 = 0.25$ enough times to be sufficiently close to its long-term behavior (400 iterations) and samples 150 points from the steady state attractor (i.e. output iterations 401:550) as a function of r, and mutates some vector as a solution, i.e. `calc_attractor!(out,f,p,num_attract=150;warmup=400)`.

Test your function with $r = 2.9$. Double check that your function computes the correct result by calculating the analytical steady state value.

In [8]:
```
function calc_attractor!(out, r, x₀; warmup=400)
    num_attract = length(out)
    # first do warmup then write each step to `out`
    x = x₀
    for i in 1:warmup
        x = r * x * (1-x)
    end
    for i in 1:num_attract
        x = r * x * (1-x)
        out[i] = x
    end
end

out = Array{Float64}(undef, 150)
calc_attractor!(out, 2.9, 0.25)
out
```

Out[8]:
```
150-element Vector{Float64}:
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 ⋮
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
 0.6551724137931031
 0.6551724137931038
```

The analytical steady state value:

If $r = 2.9$, we have the update relation:

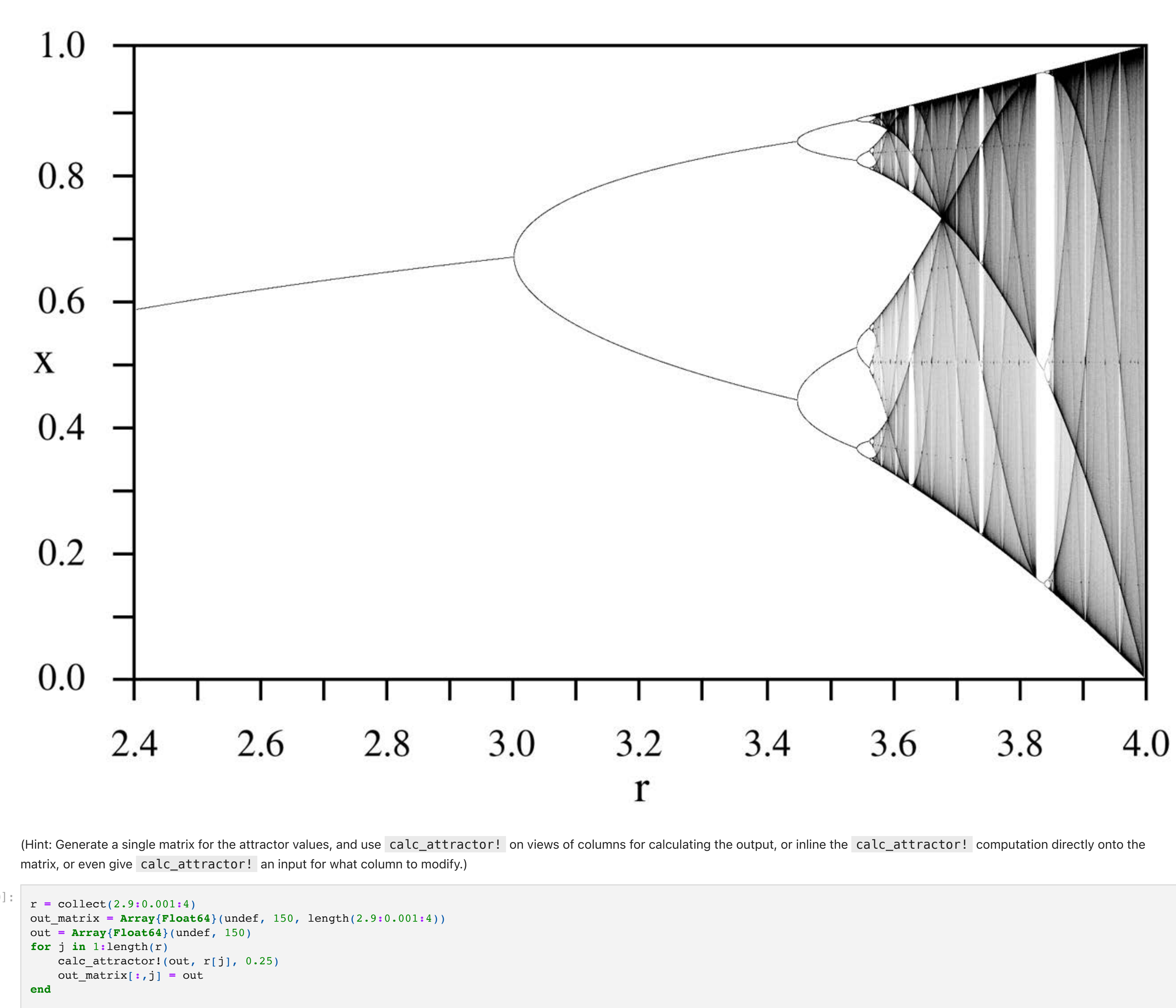
$$x_{n+1} = 2.9x_n(1 - x_n)$$

In steady state, let $x_{n+1} = x_n = x$.

$x = 2.9x(1-x) \implies x = 2.9x - 2.9x^2 \implies 2.9x^2 - 1.9x = 0 \implies x(2.9x - 1.9) = 0 \implies x = 0 \ or \ x = \frac{1.9}{2.9} \approx 0.655$, where $x = 0$ is a trivial solution.

Part 2

The bifurcation plot shows how a steady state changes as a parameter changes. Compute the long-term result of the logistic equation at the values of `r = 2.9:0.001:4`, and plot the steady state values for each r as an r x steady_attractor scatter plot. You should get a very bizarrely awesome picture, the bifurcation graph of the logistic equation.

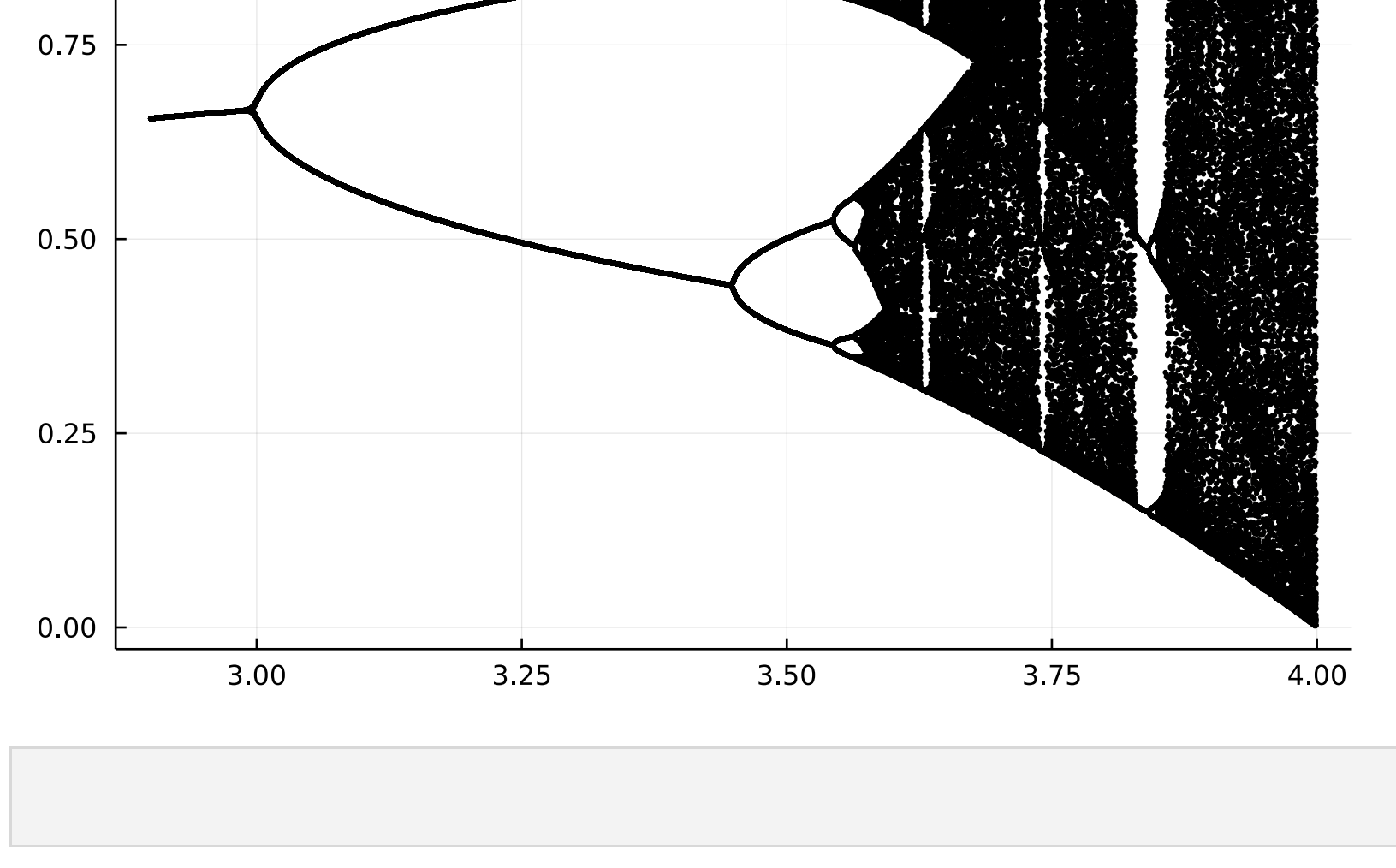


(Hint: Generate a single matrix for the attractor values, and use `calc_attractor!` on views of columns for calculating the output, or inline the `calc_attractor!` computation directly onto the matrix, or even give `calc_attractor!` an input for what column to modify.)

In [9]:
```
r = collect(2.9:0.001:4)
out_matrix = Array{Float64}(undef, 150, length(2.9:0.001:4))
out = Array{Float64}(undef, 150)
for j in 1:length(r)
    calc_attractor!(out, r[j], 0.25)
    out_matrix[:,j] = out
end

using Plots
scatter(r, out_matrix', legend=false, markersize=1, zcolor=0)
```

Out[9]:



In [ ]: