Homework 6 - BIOMATH 205

SIMON LEE

December 2023

1 Chapter 13

P9: Demonstrate that the fixed point $\sqrt{c} - r$ of the fractional linear transformation $f(y) = \frac{c-r^2}{2r+y}$ is locally attractive. This fact increases our confidence that the continued fraction expansion of $\sqrt{c} - r$ converges as expected.

Answer:
We first assume that c and r are positive values as indicated in the chapter.

Now, to demonstrate that the fixed point $\sqrt{c} - r$ of the fractional linear transformation

$$f(y) = \frac{c - r^2}{2r + y}$$

is locally attractive, we need to show that near this point, the function brings values closer to the fixed point. A standard way to do this is to compute the derivative of the function at the fixed point and show that its absolute value is less than 1.

First, let's find the fixed point of the function. A fixed point y of f satisfies $f(y) = y$. Thus, we have:

$$\frac{c - r^2}{2r + y} = y$$

We are given that $\sqrt{c} - r$ is a fixed point, so substituting $y = \sqrt{c} - r$, we should be able to verify this:

$$\frac{c - r^2}{2r + (\sqrt{c} - r)} = \sqrt{c} - r$$

$$\frac{(\sqrt{c} - r)(\sqrt{c} + r)}{\sqrt{c} + r} = \sqrt{c} - r$$

$$\sqrt{c} - r = \sqrt{c} - r$$

Now, let's compute the derivative of $f(y)$ and evaluate it at $y = \sqrt{c} - r$. The derivative of $f(y)$ is:

$$f'(y) = \frac{d}{dy}\left(\frac{c - r^2}{2r + y}\right)$$

$$f'(y) = \frac{-1 \times (c - r^2)}{(2r + y)^2}$$

Evaluating this at $y = \sqrt{c} - r$, we get:

$$f'(\sqrt{c} - r) = \frac{-(c - r^2)}{(2r + (\sqrt{c} - r))^2}$$

For the fixed point to be locally attractive, we need $|f'(\sqrt{c}-r)| < 1$. Calculating this value:

$$f'(\sqrt{c} - r) = \frac{-(c - r^2)}{(\sqrt{c} + r)^2}$$

We therefore evaluate a few values for c and r to see if they remain less than 1.

- **Case 1 (where $c = 4$ and $r = 1$):**

  $f'(\sqrt{4} - 1) = -0.333,$

  $|f'(\sqrt{4} - 1)| < 1\,?$ Yes. The fixed point is locally attractive.

- **Case 2 (where $c = 5$ and $r = 1$):**

  $f'(\sqrt{5} - 1) = -0.382,$

  $|f'(\sqrt{5} - 1)| < 1\,?$ Yes. The fixed point is locally attractive.

- **Case 3 (where $c = 9$ and $r = 2$):**

  $f'(\sqrt{9} - 2) = -0.200,$

  $|f'(\sqrt{9} - 2)| < 1\,?$ Yes. The fixed point is locally attractive.

- **Case 4 (where $c = 1$ and $r = 2$ (with $r > c$)):**

  $f'(\sqrt{1} - 2) = 0.333,$

  $|f'(\sqrt{1} - 2)| < 1\,?$ Yes. The fixed point is locally attractive.

- **Case 5 (where $c = 3$ and $r = 4$ (with $r > c$)):**

  $f'(\sqrt{3} - 4) = 0.396,$

  $|f'(\sqrt{3} - 4)| < 1\,?$ Yes. The fixed point is locally attractive.

However this is not a sufficient enough way to show that it is locally attractive. Therefore we provide an explanation and a plot of why these solutions never exceed 1.

To understand why the expression for the derivative $f'(\sqrt{c} - r)$ at the fixed point does not exceed 1 in absolute value, we analyze its form:

$$f'(\sqrt{c} - r) = \frac{-(c - r^2)}{(\sqrt{c} + r)^2}$$

1. **Numerator Analysis:** The numerator is $-(c-r^2)$. This can be rewritten as $r^2 - c$. As c and r increase, the term r^2 will grow faster than c (since it's quadratic in r). However, the difference between r^2 and c will also grow, making the numerator larger in absolute value.

2. **Denominator Analysis:** The denominator is $(\sqrt{c} + r)^2$. As c and r grow, this term also grows, but notably faster than the numerator. This is because the denominator is a square of a sum, which includes a term r^2, a term $2r\sqrt{c}$, and a term c.

3. **Balance of Growth:** While both the numerator and denominator grow with larger values of c and r, the denominator's growth is more rapid due to the squared terms and the addition with the cross-product term $2r\sqrt{c}$ and the c term. This means that, although the numerator can become quite large, the denominator increases even more, keeping the overall fraction's absolute value in check. This can be seen from the following equation where the numerator is less than the denominator because we are subtracting a c term from the numerator and adding a c and $2r\sqrt{c}$ term to the same r^2 term in the denominator indicating that the numerator is always less than the denominator.

$$|r^2 - c| < |r^2 + 2r\sqrt{c} + c|$$

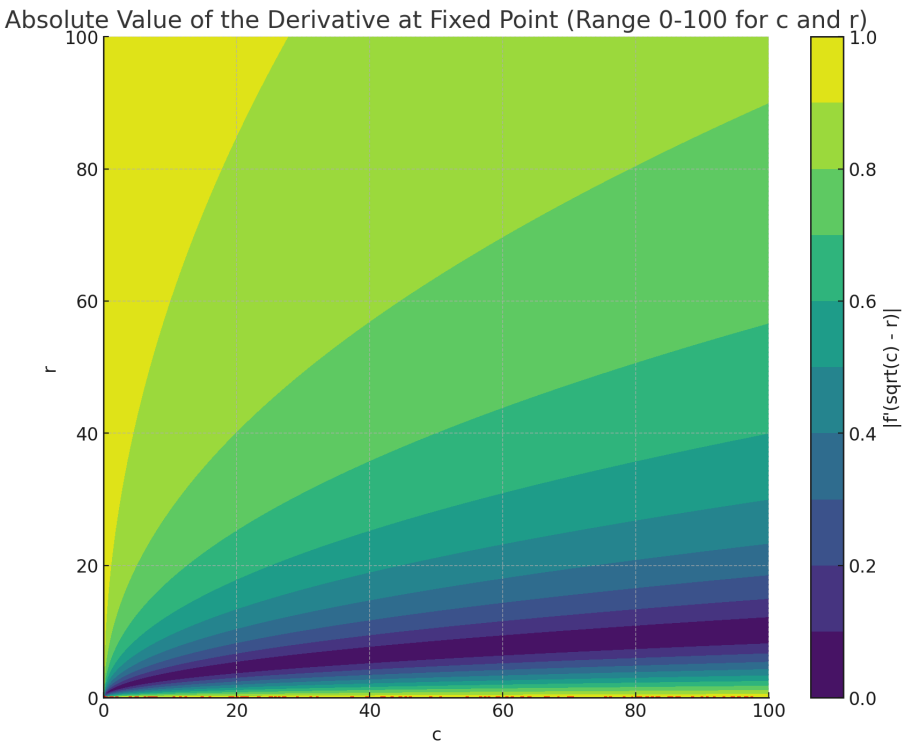

Figure 1: A plot demonstrating the absolute values at varying values of c and r. In this plot we only show from the range 0 to 100.

P4: If the function $f(x)$ has a power series expansion $\sum_{j=0}^{\infty} a_j x^j$ converging in a disc $\{x : |x| < r\}$ centered at 0 in the complex plane, then we can approximate the derivatives $f^{(j)}(0) = j!a_j$ by evaluating $f(x)$ on the boundary of a small circle of radius $h < r$. This is done by expanding the function $t \mapsto f(he^{2\pi it})$ in the Fourier series

$$f(he^{2\pi it}) = \sum_{j=0}^{\infty} a_j h^j e^{2\pi ijt}. \tag{1}$$

Thus, if we take the discrete Fourier transform b_k of the sequence $b_j = f(hu_n^j)$, then equation (11.9) mutates into

$$b_k - a_k h^k = \sum_{l=1}^{\infty} a_{ln+k} h^{ln+k} = O(h^{n+k}) \tag{2}$$

for $0 \leq k \leq n-1$ under fairly mild conditions on the coefficients a_j. Rearranging this equation gives the derivative approximation

$$f^{(k)}(0) = k!a_k = \frac{k!b_k}{h^k} + O(h^n) \tag{3}$$

highlighted by Henrici [112]. Implement this method in computer code and test its accuracy.

Answer:
In the code below, I picked the function $f(x) = sin(x), f(x) = tan(x), \& f(x) = e^x$ since it followed the properties of having a power series expansion converging to a disc centered at 0 in the complex plane. Additionally all these derivatives at x=0 is 1 so we can easily check whether the function works.

```
using FFTW  # For Fast Fourier Transform

function approximate_derivative(func, k, h, N=10000)

    # Generate N points on the boundary of the circle of radius h
    t = LinRange(0, 1, N)  # Define the range without the endpoint
    circle_points = h .* exp.(2im *  * t)

    # Evaluate the function on these points
    func_values = func.(circle_points)

    # Compute the Discrete Fourier Transform using FFT
    dft = fft(func_values) / N

    # Extract and scale the coefficient for the k-th derivative
    derivative = k * real(dft[k + 1]) / h^k

    return derivative
```

```
end

# Test the function with various functions
exp_func(x) = exp(x)
sin_func(x) = sin(x)
tan_func(x) = tan(x)

# Test and print the derivatives
println("Derivative of Sin(x) at x=0: ", approximate_derivative(sin_func, 1, 0.001))
println("Derivative of e^x(x) at x=0: ", approximate_derivative(exp_func, 1, 0.001))
println("Derivative of tan(x) at x=0: ", approximate_derivative(tan_func, 1, 0.001))

>>>Derivative of Sin(x) at x=0: 0.9999999341743497
Derivative of e^x(x) at x=0: 1.0000000342143436
Derivative of tan(x) at x=0: 0.9999999342493457
```

P6: Write code in Julia to evaluate $\sqrt{x}$ and its derivative based on its continued fraction expansion

Note: I apologize for doing this problem. I saw Dr. Sobels email and saw this one isn't part of the selected problems list. However, I left it here since I did it anyway.

```
function sqrt_continued_fraction(x, n_terms=10)
    if isperfectsquare(x)
        sqrt_x = sqrt(x)
        derivative = 1 / (2 * sqrt_x)
        return sqrt_x, derivative
    end

    a0 = floor(sqrt(x))
    m = 0
    d = 1
    a = a0

    fraction = 0
    for _ in 1:n_terms
        m = d * a - m
        d = (x - m^2) / d
        a = floor((a0 + m) / d)
        fraction = 1 / (a + fraction)
    end

    sqrt_x = a0 + fraction
    derivative = 1 / (2 * sqrt_x)
```

```
    return sqrt_x, derivative
end

function isperfectsquare(x)
    sqrt_x = sqrt(x)
    return sqrt_x == floor(sqrt_x)
end


x = 10
sqrt_x, derivative = sqrt_continued_fraction(x)
println("Approximate sqrt($x): ", sqrt_x)
println("Derivative of sqrt($x): ", derivative)
>>>Approximate sqrt(10): 3.1622776601683795
Derivative of sqrt(10): 0.15811388300841897
```

Algorithm Limitations

During the development of the code, I found an interesting behavior that tends to occur with perfect squares. Using the continued fraction expansion to approximate the square root of a perfect square, such as $x = 9$, can lead to computational issues in the algorithm. This arises from the distinctive nature of the continued fraction expansion for square roots.

Continued Fraction Expansion for Non-Square Integers

For non-square integers, the continued fraction expansion of their square roots is an infinite, periodic sequence. This allows the algorithm to iteratively approximate the square root by considering a finite number of terms in the sequence.

$$\sqrt{x} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ddots}}}$$

where $a_0 = \lfloor \sqrt{x} \rfloor$ (the integer part of $\sqrt{x}$), and the subsequent a_i follow a pattern that can be determined iteratively.

Issues with Perfect Squares

However, for a perfect square like 9, the square root is exactly 3, and the continued fraction expansion should ideally terminate immediately after the first term, which is the integer part of the square root (i.e., 3 in this case).
When applying the general algorithm for continued fraction expansion to a perfect square like 9, the division by zero error may arise:

1. **Division by Zero:** The algorithm involves division operations, which can lead to division by zero, as the subsequent terms in the continued fraction for a perfect square do not exist.

Handling Perfect Squares

To compute the square root and its derivative accurately for perfect squares using continued fractions, it is more appropriate to handle these cases separately. This approach involves directly returning the exact square root and its derivative without relying on the iterative approximation process, which is only applicable and necessary for non-square integers.

P11: Program Search.jl to perform logistic regression. No derivatives need be input. Apply your code to the Titanic data, which can be extracted via the commands

```
Using RDatasets
df = dataset("count", "titanic") # data frame
y = convert(Vector{Float64}, df[:, 1]); # responses
X = Tables.matrix(df[:, 2:4]) # cases by predictors
```

If necessary, convert the predictors to real numbers.

Answer:

Below I have added the code which should work in conjunction with Search.jl to perform a logistic regression on the Titanic Dataset.

```
using LinearAlgebra, ForwardDiff, Printf, SpecialFunctions, RDatasets, Tables, DataFrames

include("Search.jl") # Include the Search.jl file

# Define logistic regression cost function
function logistic_regression(par, X, y)
    m = length(y)
    predictions = 1 ./ (1 .+ exp.(-X * par))
    cost = -sum(y .* log.(predictions) + (1 .- y) .* log.(1 .- predictions)) / m
    return cost
end

# Initialize logistic regression problem for Titanic dataset
function initial_titanic()
    df = dataset("count", "titanic") # data frame
    y = convert(Vector{Float64}, df[:, 1]); # responses
    X = Tables.matrix(df[:, 2:4]) # cases by predictors

    pars = size(X, 2)
```

```
    constraints = 0
    points = 1
    travel = "search"
    title = "Titanic Logistic Regression"
    cases = 0
    standard_errors = false

    grid, pname, par, pmin, pmax, constraint, level, goal = set_search_defaults(constraints,
    par .= 0.0
    goal = "minimize"

    return (grid, pname, par, pmin, pmax, constraint, level, travel, goal, title, cases, sta
end

# Run logistic regression on the Titanic dataset
(par, f) = search((par) -> logistic_regression(par, X, y), initial_titanic, stdout)

>>>                   Search, Julia Version

                (c) Copyright Kenneth Lange, 2022


Title: Titanic Logistic Regression
Grid or search option: search
Minimize or maximize: minimize

Parameter minima and maxima:

    par 1       par 2       par 3

    -Inf        -Inf        -Inf
     Inf         Inf         Inf

iter  steps   function    par 1       par 2       par 3

  1    0    6.9315e-01   0.0000e+00  0.0000e+00  0.0000e+00

  2    0    6.4951e-01  -1.2272e-01 -1.9719e-01 -4.0540e-01

  3    0    6.2174e-01  -5.4431e-02 -2.0595e-01 -2.6056e-01

  4    0    6.0783e-01   2.4813e-02 -3.5325e-01 -1.8486e-01

  5    0    5.6767e-01   2.5310e-01 -1.0058e+00 -1.4770e-01

  6    0    5.4547e-01   6.4849e-01 -1.9460e+00 -3.3034e-04
```

```
 7    0    5.4504e-01   6.5083e-01 -1.9478e+00 -1.1206e-02

 8    0    5.4392e-01   6.9933e-01 -1.9857e+00 -4.5397e-02

 9    0    5.4173e-01   8.6246e-01 -2.0239e+00 -1.1157e-01

10    0    5.3919e-01   1.2067e+00 -1.9907e+00 -2.3259e-01

11    0    5.3917e-01   1.2291e+00 -1.9889e+00 -2.3922e-01

12    0    5.3917e-01   1.2291e+00 -1.9890e+00 -2.3922e-01

13    0    5.3917e-01   1.2293e+00 -1.9893e+00 -2.3921e-01

14    0    5.3917e-01   1.2295e+00 -1.9897e+00 -2.3921e-01

15    0    5.3917e-01   1.2295e+00 -1.9897e+00 -2.3921e-01

The minimum function value of 0.53917 occurs at iteration 15.
```

P12: Implement this chapter's neural net code on data of your choice with multiple hidden layers. The data should be available in a public repository reachable by Julia.

In the code below, I used the Iris dataset from the RDatasets package. The Iris dataset is a classic dataset in the field of machine learning and statistics, often used for demonstrating classification algorithms. This dataset is the most popular binary classification problem. The goal of this competition is to predict whether or not an Iris flower belongs to the Iris Setosa species.

I also initialized 4 hidden layers as you will see from the following code line.

```
    nodes = [size(X, 1), 10, 8, 6, 4, 1]  # input layer, 4 hidden layers, and output layer
```

I took the neural net code from the book as well as the logistic loss. I wrote the sigmoid activation function in a similar style to what we saw in the book. I didn't know what we had to show so I simply plotted the cost by iteration. Feel free to copy paste the code and you can see other attributes of the model.

```
using LinearAlgebra, Plots, RDatasets, StatsBase

function logistic(x, y) # loss
    (u, du) = (similar(x), similar(x))
    @. u = - (y * log(x) + (1 - y) * log(1 - x))
    @. du = - (y - x) / (x * (1 - x))
```

```
    return (u, du)
end

function sigmoid(z)
    a = similar(z)
    da = similar(z)
    @. a = 1 / (1 + exp(-z))
    @. da = a * (1 - a)
    return (a, da)
end

function initialize(nodes)
    par = Dict()
    for l = 1:length(nodes) - 1
        (W, b) = ("W" * string(l), "b" * string(l))
        par[W] = randn(nodes[l + 1], nodes[l]) / sqrt(nodes[l])
        par[b] = zeros(nodes[l + 1], 1)
    end
    return par
end

function update(par, grad, layers, t)
    for l = 1:layers - 1
        (W, b) = ("W" * string(l), "b" * string(l))
        par[W] = par[W] - t * grad[W]
        par[b] = par[b] - t * grad[b]
    end
    return par
end

function forward(X, par, cases, activation, layers)
    z = X
    cache = [(z, zeros(0,0))]
    for l = 1:layers - 1
        (W, b) = ("W" * string(l), "b" * string(l))
        (a, da) = activation(par[W] * z .+ (par[b] * ones(cases)'))
        push!(cache, (a, da))
        z = a
    end
    return cache
end

function backward(X, y, par, cache, activation, loss, layers)
    grad = Dict()
    (a, da) = cache[layers]
    (obj, dZ) = loss(a, y)
```

```
    cost = sum(obj)
    dZ = da .* dZ
    for l = reverse(1:layers - 1)
        (a, da) = cache[l]
        (W, b) = ("W" * string(l), "b" * string(l))
        grad[W] = dZ * a'
        grad[b] = sum(dZ, dims = 2)
        if l > 1
            dZ = par[W]' * dZ
            dZ = da .* dZ
        end
    end
    return (cost, grad)
end

function train(nodes, X, y, t, iters, fun, plot_name)
    (activation, loss) = fun
    (cases, layers) = (size(X, 2), length(nodes))
    par = initialize(nodes)
    cost = zeros(iters)
    for i = 1:iters
        cache = forward(X, par, cases, activation, layers)
        (cost[i], grad) = backward(X, y, par, cache, activation, loss, layers)
        par = update(par, grad, layers, t)
    end
    cost_plot = plot(1:iters, cost, legend = false, title = "Cost per Iteration", xlabel = '
    display(cost_plot)
    savefig(plot_name)
    return par, cost
end

# Using the Iris dataset
df = dataset("datasets", "iris")

# binarizes 1 for setosa, 0 for rest
y = convert(Vector{Float64}, df[:, "Species"] .== "setosa")
y = reshape(y, 1, length(y))

XT = Tables.matrix(df[:, 1:4])
X = Matrix(XT')
X = convert(Matrix{Float64}, X)

# Define network structure with four hidden layers
nodes = [size(X, 1), 10, 8, 6, 4, 1]  # Input layer, four hidden layers, and output layer
fun = (sigmoid, logistic)
(t, iters, plot_name) = (0.001, 1000, "Iris.png")
```

```
(par, cost) = train(nodes, X, y, t, iters, fun, plot_name)
```

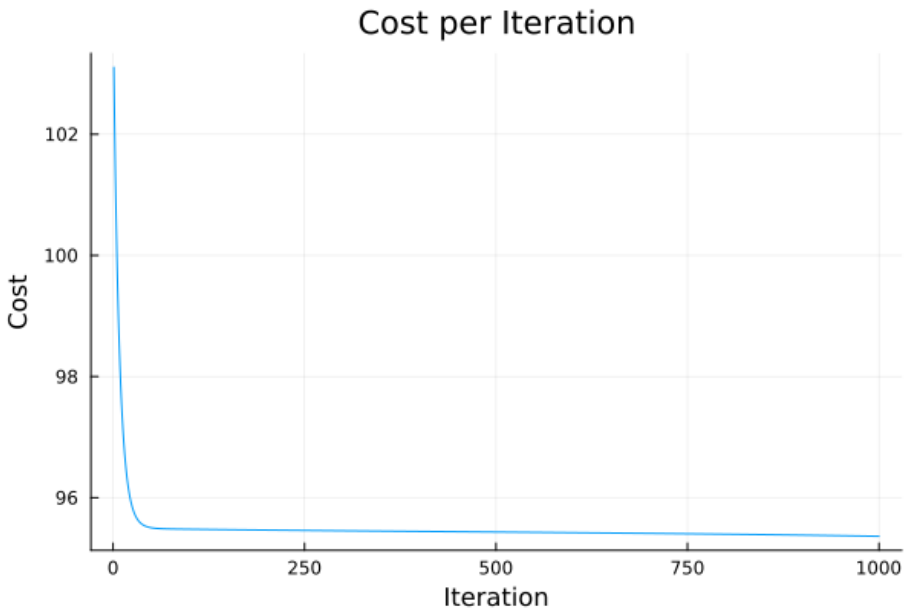

Figure 2: The Cost per Iteration Plot of the Iris dataset generated similar to the examples from the Book

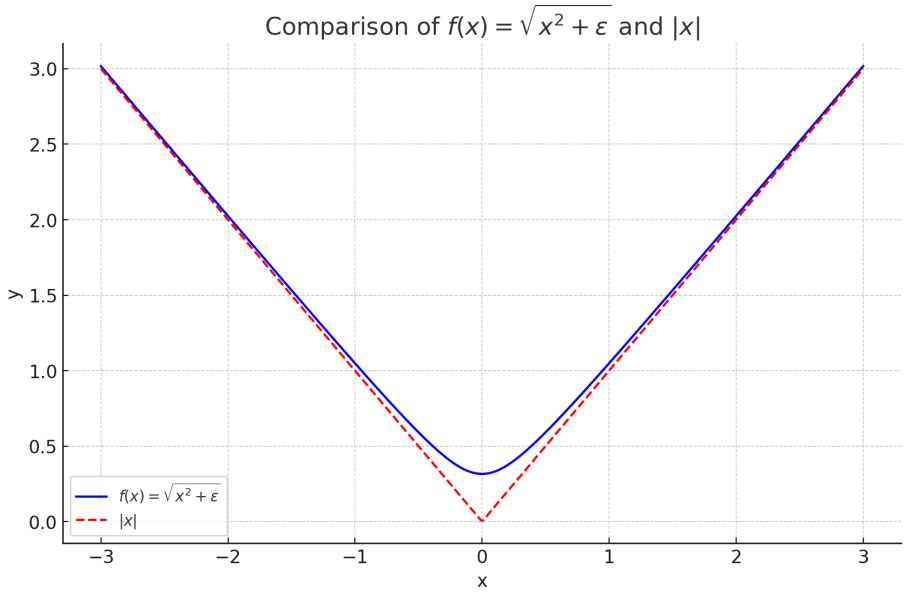

Figure 3: The function $\sqrt{x^2 + \epsilon}$ compared to $|x|$

P13: Prove that the function $f(x) = \sqrt{x^2 + \epsilon}$ for $\epsilon > 0$ small is majorized by the quadratic $g(x) = \sqrt{x_n^2 + \epsilon} + \frac{1}{2\sqrt{x_n^2+\epsilon}}(x^2 - x_n^2)$. Plot $f(x)$ and note how well it approximates $|x|$. Finally prove $f(x)$ is convex.

Comparison with Absolute Value Function

The function $f(x)$ approximates the absolute value function $|x|$ closely, especially near the origin. This can be visualized in the plot below, where the function $f(x)$ is shown alongside $|x|$ in Figure 2.

Convexity of $f(x)$

To prove that $f(x)$ is convex, we consider its second derivative. The second derivative of $f(x)$ is given by:

$$f''(x) = \frac{d^2}{dx^2}\sqrt{x^2 + \epsilon}$$

We plug in the value $\epsilon = 0.0001$ to make this computation easier After computation, the second derivative is found to be:

$$f''(x) = \frac{(-x^2/(x^2 + 0.0001) + 1)}{\sqrt{x^2 + 0.0001}}$$

This expression can be simplified to:

$$f''(x) = \frac{0.0001}{(x^2 + 0.0001)\sqrt{x^2 + 0.0001}}$$

Since the numerator and denominator are both positive for all x, the second derivative $f''(x)$ is non-negative. Therefore, the function $f(x)$ is convex.