گزارش پروژه اول درس هوش مصنوعی

نگین اسماعیل زاده ۹۷۱۰۴۰۳۴

مختصر توضیحات :

نحوه ی نگاشت این مسئله به درخت به این صورت انجام شده که یک کلاس Node تعریف شده که با خود سه عضو دارد (مقدار خانه ، عضو راست و عضو چپ) و برای تشکیل یک درخت کافست مشخص کنیم مقدار هر گره چیست (که متناظر با عملگر ها عملوند و اعداد نوشته شده در هر عضو درخت است) و از گره ریشه عضو راست و چپ گره را در صورت وجود مشخص کنیم. اما از آنجایی که محدودیت هایی برای ایجاد این درخت وجود دارد (برای مثال بعضی گره ها یا حتما یک عضو دارند یا حتما عضوی ندارند مانند عملگر های تک عملوندی و متغیر یا اعداد ثابت که فقط میتوانند در برگ درخت باشند) لازم است در هر مرحله انتخاب رندم مقدار خانه ی یک گره از درخت از میان مجموعه ی عملگر ها و عملوند ها شرایط معتبر بودن درخت (معادل ریاضی داشته باشد) رعایت شود. در این پیاده سازی فرض شده است برای تقریب تابع مجاز به استفاده از عملگر های + ، - ، * ، / ، ^ ، sin ، cos ، abs هستیم. همچنین فرض شده است تقریب تک بعدی است و از اعداد ثابت در بازه ی (5,5-) میتوان استفاده نمود (فرض این است که اعداد بزرگ تر با جهش یا ضرب و تقسیم ایجاد خواهند شد اما میتوان این بازه را نیز بزرگ کرد). به صورت کلی نحوه ی تولید جمعیت اولیه با تعداد مشخص شده و توسط کاربر به صورت رندم از بین عملگر ها، عملوند ها و متغیر مسئله و با فرض یک آستانه ی حداکثری برای عمق درخت های تولید شده است. پس از تولید نسل اولیه شایستگی هر عضو از این نسل توسط تابع شایستگی متناسب با معکوس خطای کمترین مربعات خروجی محاسبه میشود و به صورت یکنواخت متناسب با شایستگی نسبی اعضای جمعیت احتمال حضور آن ها در نسل بعد مشخص می شود و فرض شده است که ماکزیمم ۲۰ عضو در جمعیت انتخاب شده قرار خواهند داشت (که این عدد قابل تغییر است). در بین اعضای انتخاب شده برای حضور در نسل بعد (با احتمالی که توضیح داده شد) عملیات ترکیب متقاطع به این صورت انجام میشود که به ازای هر ۲ انتخاب عضو به عنوان والدین ، دو درخت به عنوان فرزند تولید میشوند. نحوه ی تولید فرزندان به این صوزت است که ابتدا یک گرده رندم از درخت والد اول و یک گره رندم از درخت والد دوم انتخاب میشود، سپس محتویات درخت اول، بعد از گرده انتخاب شده ی متناظر با آن قطع شده و به جای آن محتویات درخت دوم، بعد از گره انتخاب شده ی متناظر با آن قرار میگیرد و فرزند اول را تشکیل میدهد و همین کار نیز به صورت معکوس برای تولید فرزند دوم از هر دو والد مشخص شده انجام می شود. پس از انجام ترکیب با احتمالی که مقدار اولیه آن ورودی کاربر است (در حال حاضر ۱۰ درصد نظر گرفته شده) و به صورت خطی با افزایش نسل افزایش پیدا میکند یک گره از درخت های هر عضو یک نسل تغییر خواهد کرد . منتهی از آنجایی که از تغییرات بیش از حد درخت جلوگیری شود یک گره تنها میتواند با مقداری از نوع خودش جابه جا شود (اگر عمگر دو عملوندی بود با یک عملگر دو عموندی دیگر ، اگر عملگر تک عملوندی بود با یک عملگر تک عملوندی دیگر و اگر عدد ثابت یا متغیر بود با یک عدد ثابت یا متغیر) و به این صورت پس از اعمال جهش فرزندان نسل بعد تولید خواهند شد و این الگورتیم تکرار می شود تا زمانی که یکی از شروط خاتمه اعمال شود :

۱) خطای کمترین مربعات خروجی برای یک عضو از یک نسل از استانه ۰.۰۰۱ کمتر شود (معادل با شایستگی بیشتر از ۱۰۰۰).
۲) پس از گذشت از حد ماکزیمم تعداد نسل های تولید شده که توسط کاربر مشخص میشود و مقدار آن در حال حاضر برابر با ۲۰ فرض شده است.

چالش ها :

اولین چالش در رابطه با تکرار شدن فرایند خواندن یک درخت برای هر نقطه ی x ورودی بود که اگر برای هر نقطه مجدد درخت را میخواندیم بسیار زمانبر می شد ، به همین دلیل تابعی نوشتم که یک درخت را به معادل رشته ی خود تبدیل میکند و سپس با استفاده از تابع eval خروجی این رشته را به ازای متغیر معلوم شده در تعداد زیادی نقطه میتوانیم بدست آوریم.

چالش دوم در نحوه ی برسی درستی نتایج هر تابع بود که برای این موضوع تابعی زده شد تا درخت معادل را رسم کند تا بتوان راحت تر دیباگ کرد.

چالش سوم در پیاده سازی عملیات ترکیب متقاطع بود که چگونه این عملیات انجام شود که فرزندان از نظر ریاضی معتبر باشند، نهایتا راه حل این شد که از یک گره رندم به بعد والدین را باهم ترکیب کنیم و از هر دو والد دو فرزند تولید کنیم

چالش چهارم در مرحله ی جهش بود که جهش ها میتوانست فرزندان نا معتبر تولید کند و از طرفی جهش ها نباید باعث تغییرات در تعداد زیادی از کروموزوم ها میشد. برای رفع این موضوع جهش ها به انواع هم جنس خود (به نحوی که بالاتر توضیح داده شد) محدود شد.

چالش پنجم این است که متاسفانه مخرج کسر ها در یک تابع فرض شده در نقاط ورودی مسئله میتواند به نحوی صفر شود و این باعث ایجاد خطا در برخی مواقع اجرا میشود و این موضوع را با اکسپشن هندل کردم که اگر این رخ داد مقدار x را به میزان کوچکی مثلا ۰.۱ جابجا کند و اگر مجددا رفع نشد این تابع به احتمال زیاد متلوب نیست مانند : 1/(x-x) بنابراین خطای آن را برابر با ماکزیمم ممکن قرار دادم تا در نسل بعد انتخاب نشود.

چالش ششم هم این هست که تابع eval نمیتواند به ازای مقادیر بزرگ ورودی کار کند و باعث میشود گاهی کد با خطا مواجه شود و این موضوع را هم مجبور شدم مانند قبلی با اکسپشن رفع کنم که این اعضا را تقریبا به عنوان اعضای نابهینه با احتمال خوبی دور میریزم.

توابع :

```
GP_function_approximator(initial_population_number,Max_generation,mutation_pr
obability,x_list,y_list)
…
return current_generation , best_function , best_output,best_output_fitness
```

این تابع تعداد جمعیت اولیه ، تعداد ماکزیمم نسل های قابل گذشت ، احتمال جهش ، لیستی از نقاط ورودی x و لیستی از f(x) های متناظر را به عنوان ورودی میگیرد و الگوریتم GP را اجرا میکند و در نهایت تعداد نسل های طی شده ، مناسب ترین تابع پیدا شده، لیستی از خروجی تقریب زده شده ینی g(x) های متناظر با ورودی و مقدار شایستگی خروجی را میدهد.

```
Def random_generation_initialization(root,layer,Max_layers):
…
return root
```

این تابع ماکزیمم عمق درخت را دریافت میکند و به صورت بازگشتی یک درخت اولیه معتبر ریاضی تولید میکند و در خروجی میدهد که تعداد و مقدار گره های آن کاملا تصادفی است.

```
Def fitness(population,y_list,x_list):
    return scores,efficiancy_flag,best_function,best_output,best_output_fitness
```

این تابع یک نسل و لیستی از x و f(x) را در ورودی گرفته و در خروجی مقدار امتیاز توابع تخمینی، درستی برقراری شرط خاتمه از نظر دقت، شایسته ترین تابع و لیستی از g(x) های مربوط به آن و مقدار شایستگی شایسته ترین فرد را در خروجی برمیگرداند. توجه شود که میزان شایستگی در بازه ی ۰ تا بینهایت است که مقدار بینهایت با Inf نمایش داده میشود و نشانه ی دقیق بودن تخمین با این معیار شایستگی است.

```
Def expression_constructor(root):
return exp
```

این تابع ریشه یک درخت را میگیرد و به صورت بازگشتی string مربوط به آن را تشکیل میدهد و در خروجی بر میگرداند.

```
Def evaluator(expression,x_list):
return out_list
```

این تابع یک string که به فرم رابطه ی ریاضی است و لیست x های وروی مسئله را میگیرد و در خروجی جواب رابطه ی ریاضی داده شده را در نقاط داده شده بر میگرداند.

```
Def is_single_operator(node):
    return  (node.left is None and node.right != None) or (node.right is None
and node.left != None)
```

این تابع یک گره از درخت را دریافت میکند و در خروجی مشخص میکند این گره یک عملگر تک عملوندی اس یا خیر.

```
Def is_leaf(node):
    return node.left is None and node.right is None
```

این تابع یک گره از درخت را دریافت میکند و در خروجی مشخص میکند این گره یک برگ است یا خیر(برگ ها نشان دهنده عملوند در مسئله ما هستند)

```
def selection (population,scores,method):
return selected_population
```

این تابع جمعیت فعلی ، امتیازات مربوط با شایستگی اعضا و روش انتخاب اعضای والد را دریافت کرده و در خروجی والدین منتخب را برمیگرداند.

```
Def crossover(selected_population):
return crossedover_population
```

این تابع والدین انتخاب شده را در ورودی میگیرد و در خروجی فرزندان را از ترکیب متقاطع تولید کرده و بر میگرداند.

```
Def mutation(crossedover_population,mutation_propability):
return next_generation
```

این تابع فرزندان نسل جدید را دریافت میکند و در خروجی نسل بعد نهایی شده را بر میگرداند (تعدادی از فرزندان تغییر نمیکنند و برخی از آنها به نحوی که در ابتدا توضیح داده شد جهش می یابند.

نمونه ها :

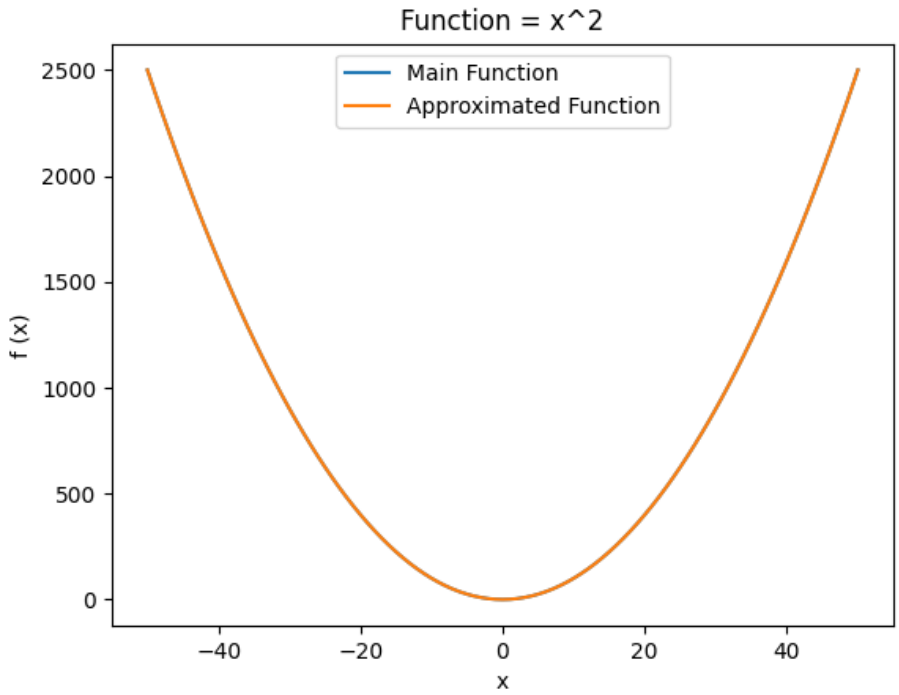
ورودی۱ : نمونه های تابع y = x^2 به این صورت :

```
## Initializations
initial_population_number = 100
Max_generation =  200
mutation_probability = 10

## Inputs
x_list = []
y_list = []
for i in range(-50,51):
    x_list.append(i)
    y_list.append(i**2)
```

خروجی های ۱ :

```
approximated function =  (x*x)
fitness =  Inf
0 generations were passed
execution_duration =  0.9069080352783203 s
```



این نمونه با موفقیت تقریب زده شد علت آن این است که ساده هست و حتی در ۱۰۰ جمعیت تصادفی اولیه با احتمال بالایی تولید میشود.

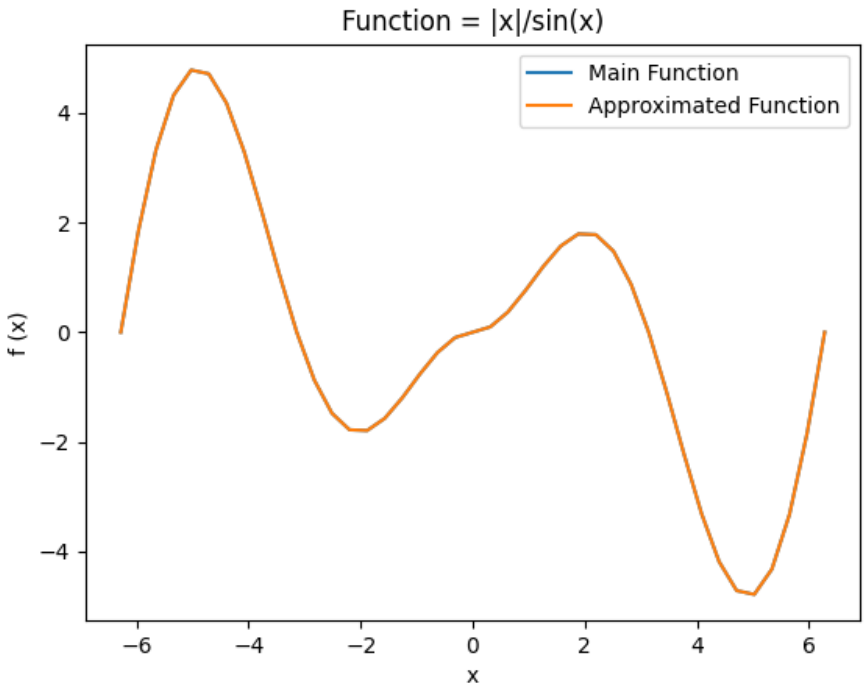
ورودی ۲ : نمونه های تابع y = abs(x)*sin(x) به این صورت :

```
## Initializations
initial_population_number = 100
Max_generation =  200
mutation_probability = 10

## Inputs
x_list = []
y_list = []
for i in range(-20,21):
    x = 4*pi * i/40
    x_list.append(x)
    y_list.append(abs(x)*sin(x))
```

خروجی های ۲ :

```
approximated function =  (x*(sin((abs(x)))))
fitness =  Inf
6 generations were passed
execution_duration =  9.919869184494019 s
```



همانطور که مشاهده میشود الگوریتم در این تابع نیز در زمان کم و در طی ۹ نسل به جواب اپتیمم تقریب رسید. علت بیشتر شدن نسل ها نسبت به مثال قبل نسبتا پیچیده تر شدن درخت (عمیق تر شدن) هست.

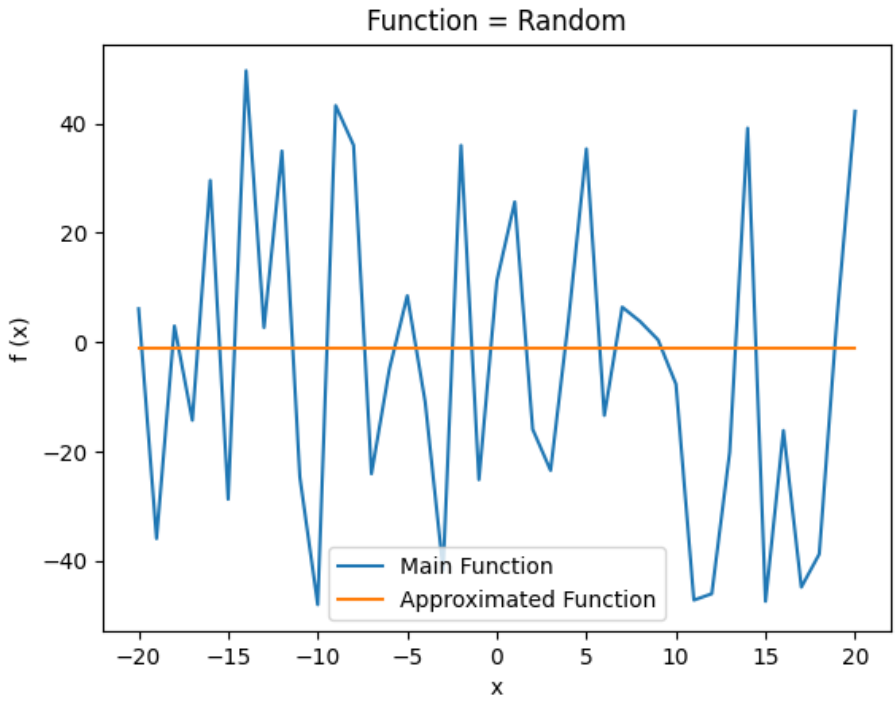
ورودی ۳ : نمونه های تابع رندم (مانند مثال خط خطی) به این صورت :

```
## Initializations
initial_population_number = 100
Max_generation =  200
mutation_probability = 10

## Inputs
x_list = []
y_list = []
for i in range(-20,21):
    x_list.append(i)
    y_list.append(-50+100*random())
```

خروجی های ۳ :

```
approximated function =  (sin(67.53261184535957))
fitness =  2.9040171908146423e-05
200 generations were passed
execution_duration =  289.7842597961426 s
```



در این مثال مشاهده می شود که نتوانستیم تابع مناسبی ارائه دهیم و تقریبا به نوعی میانگین تابع به عنوان خروجی داده شده که خب از نظر خطا قابل دفاع هست اما خروجی مناسبی نیست. بنظر میرسد در تشخیص توابع بدرفتار با مشکل رو به رو هستیم.

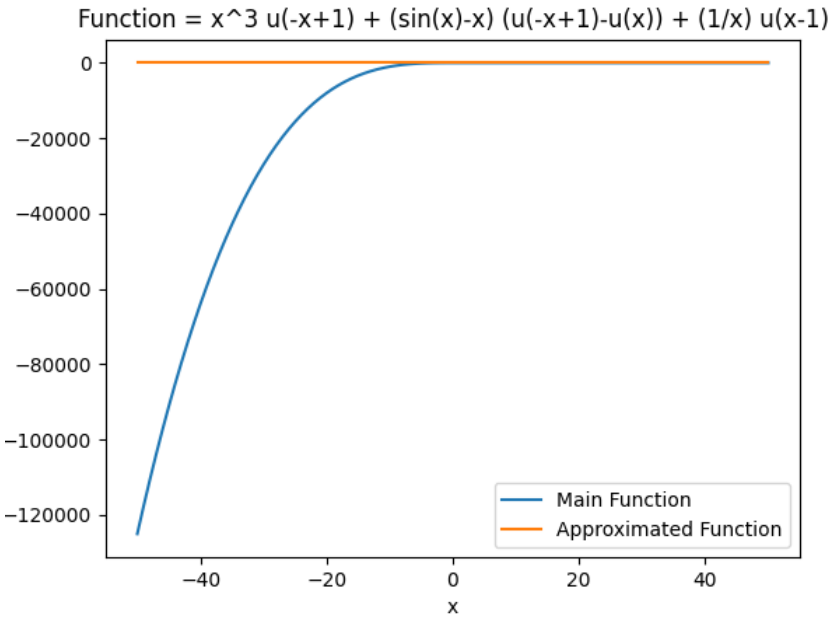
ورودی ۴ : نمونه های تابع ناپیوسته ی که برای نقاط کمتر از صفر ضابطه ی y = x^3 دارد و برای نقاط بین صفر و یک ضابطه ی y = -x + sin(x) دارد و برای نقاط بیشتر از یک ضابطه ی y = 1/x به این صورت :

```
## Initializations
initial_population_number = 100
Max_generation =  200
mutation_probability = 10

## Inputs
x_list = []
y_list = []
for i in range(-50,51):
    x_list.append(i)
    if (i<0):
        y_list.append(i**3)
    elif (i>=0 and i<1):
        y_list.append(-i+sin(i))
    else:
        y_list.append(1/i)
```

خروجی های ۴ :

```
approximated function =  (sin(-39.24307205371181))
fitness =  8.363118436271318e-12
200 generations were passed
execution_duration =  77.22853899002075 s
```



همانطور که در مثال قبل هم دیدیم در تشخیص توابع بد رفتار این الگوریتم انچنان موفق نیست. در اینجا هم اگر تابع چند ضابطه ای به ورودی داده شود در واقع یکی از ضابطه ها را فقط میتوان با پاسخ فیت کند تا خطا کم باشد اما قادر به دنبال کردن تغییرات همه ی قسمت های تابع نیست.

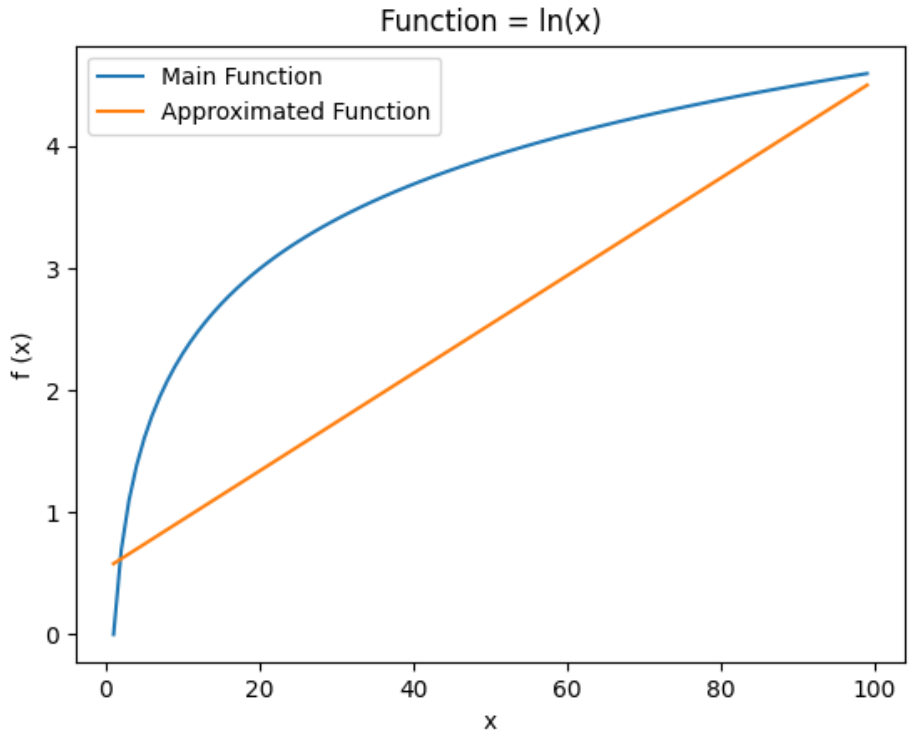
ورودی ۵ : نمونه های تابع y = ln(x) (که در کد این عملگر وجود ندارد) به این صورت :

```
## Initializations
initial_population_number = 100
Max_generation = 10
mutation_probability = 10

## Inputs
x_list = []
y_list = []
for i in range(1,100):
    x_list.append(i)
    y_list.append(log(i))
```

خروجی های ۵ :

```
approximated function =  (((x/5)/5)+(cos((x/x))))
fitness =  0.006965578173657713
10 generations were passed
execution_duration =  3.492017984390259 s
```



تابع ln در عملگر های ابتدا پیاده سازی نشده است بنابراین مجددا احتمالا الگورتیم نمیتواند به جواب دقیقی از ان برسد اما پاسخ تقریبا قابل قبولی مشاهده شد که در واقع یک نوع تقریب خطی از این تابع است.

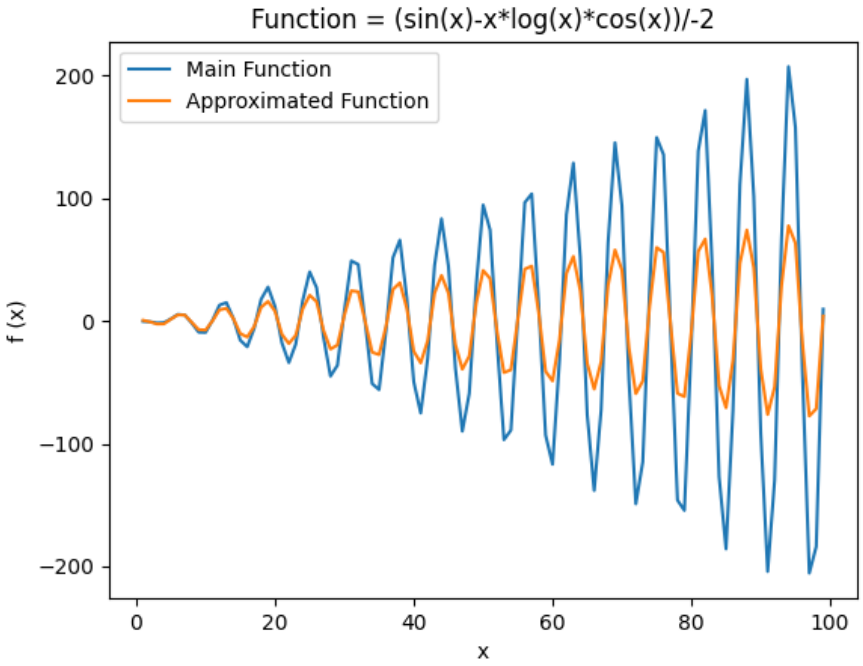
ورودی ۵ : نمونه های تابع $y = -1/2\ (\sin(x)-x\cos(x)\ln(x))$ (هم عملگر ln را نداریم هم پیچیدگی بیشتر شده) به این صورت :

```
## Initializations
initial_population_number = 100
Max_generation = 20
mutation_probability = 10

## Inputs
x_list = []
y_list = []
for i in range(1,100):
    x_list.append(i)
    y_list.append((sin(i)-i*log(i)*cos(i))/-2)
```

خروجی های ۵ :

```
approximated function =  (x*(sin((cos(x)))))
fitness =  3.8851467719241754e-06
20 generations were passed
execution_duration =  10.082016944885254 s
```



همانطور که مشاهده میشود مجددا الگوریتم قدرت تشخیص دقیق این تابع را هم نداشت و مانند سه مثال قبل میزان شایستگی نیز در این حالات بسیار کم است اما مجددا از شکل متوجه میشویم سعی شده تا نزدیک ترین تقریب ممکن زده بشود.

برسی کلی و نتیجه :

به نظر میرسد این الگوریتم به طور کلی وابستگی به جمعیت تصادفی اولیه دارد و این موضوع در هنگام ساخت توابع پیچیده تر متاسفانه بیشتر هم میشود. بنابراین مشاهده شد که گاهی نیاز بود مثلا ۲ بار برای یک ورودی اجرا کرد و نتایج میتوانست کاملا متفاوت باشد. اما به نظر میرسد این الگوریتم بسیار روش خوبی میتواند باشد، حتی با وجود زمان بالا به نسبت دریای بیکران حالت های موجود روش هوشمندانه ای برای نزدیک شده به جواب ارائه میدهد که احتمالا تشخیص آن در حالت تصادفی متوانست غیر ممکن باشد. اما عیب دیگر این روش مقدار تکرار ها و حجم به نسبت سنگین این کار هست خصوصا زمانی که تابع قدری پچیده تر میشود و به تبع آن درخت عمیق تر و دارای حالات بیشتر میشود (برای مثال من با سیستم خودم نتوانستم برای تعداد نسل های بیشتر از ۲۰۰ اجرا بگیرم چون بسیار زمانبر میشد) و این موضوع که لازم هست تعداد نسل های خوبی بگذرد کمی چالش برانگیز است. از طرفی با تکرار آزمون ها متوجه میشویم که لزوما تعداد نسل های زیاد هم خوب نیست چرا که میتواند والدین شایسته تر را از بین ببرد، بنابراین در نهایت تصمیم بر این شد که حدود تعداد ۲۰ نسل عدد مناسبی میتواند باشد.

بطور کلی از تست ها متوجه میشویم، هرچقدر کارکرد این الگورتیم برای توابع خوش رفتار یا قابل شکسته شدن به المان ها خوب بود در مقابل توابع دیگر اصلا خوب نبود.