Explanation of GIVE ME SOME FOOD(GMSF) function approximation

Gwangju AI school
TEAM 4 of Vision A
Give Me Some Food

1. Real number $\mathbb{R}$

1.1. Mathematical definition : A real number is a value that represents a quantity along a continuous line. The real number system is a unification of the rational and irrational numbers, representing all points on an infinitely long number line. Set of real number is denoted by the symbol $\mathbb{R}$.
Real numbers have the property of being ordered[1), which means that for any two different real numbers, one is necessarily greater than the other. This ordering, combined with their completeness(every bounded set has a least upper bound), gives rise to many foundational results in calculus, analysis, and other branches of mathematics.

1.2. Float point : Computers inherently work with discrete values due to their binary nature, so they can't represent most real numbers exactly. Instead, computers approximate real numbers using a format called floating-point representation. Computers typically use the IEEE 754 standard for floating-point arithmetic. The most commonly used formats under this standard are single precision(usually known as float in many programming languages) and double precision(double). Following figure shows an example of float point representation.

figure 1. *Example of a float point representation*

	0	0 0 0 ··· 0 1 0	1 0 1 0 ··· 0 0 0	
	sign bit	exponent(magnitude)	fraction(significand, mantissa)	bias
32 bit	1 bit	8 bit	23 bit	127
64 bit	1 bit	11 bit	52 bit	1023

2. Min-max scaling(or normalization) : technique in preprocessing data for machine learning. The primary objective of min-max scaling is to transform features to be within a specific range, typically $[0, 1]$.

2.1. Formula : Given a dataset, for each data point x in a feature, the transformed value x' after min-max scaling is computed as $x' = \frac{x - \min}{\max - \min}$ where x is the original value, $\min$ and $\max$ are the minimum and maximum value of the

1) Total ordered set.

feature in the dataset each. The formula can be modified as $x' = a + \frac{(x-\min)(b-a)}{\max - \min}$ for scaling the feature to be within a range $[a, b]$.

2.2. Advantages[2)] : Min-max scaling ensures that all features have the same scale, in short scale invariance. This can be particularly important for algorithms that are sensitive to the scale of input features, like k-means clustering or gradient descent optimization in neural networks. And Min-max scaling preserves the shape of the original distribution, meaning the relative distances between data points remain the same. Also, by scaling features into a [0, 1] range, it can be beneficial for algorithms that expect input features within this range, like neural network.

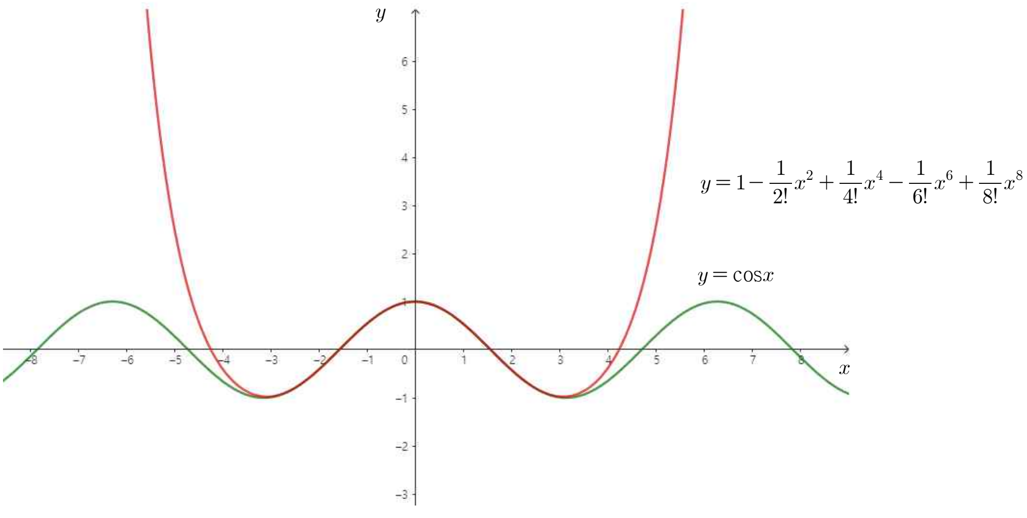
3. Taylor series : A way to representation of a function as an infinite sum of terms, each of which is calculated from the values of the function's derivatives at a single point.

3.1. Definition

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!} + (x-a)^3 + \cdots = \sum_{i=0}^{\infty} \frac{f^{(i)}}{i!}(x-a)^i = \sum_{i=0}^{\infty} a_i x^i$$

3.2. Taylor polynomial : Suppose we take Taylor series $\sum_{i=0}^{\infty} a_i x^i$ as infinite sum. Then partial sum $\sum_{i=0}^{n} a_i x^i$ is Taylor polynomial of nth degree.

figure 2. *Taylor polynomial of cosine at degree* 8.

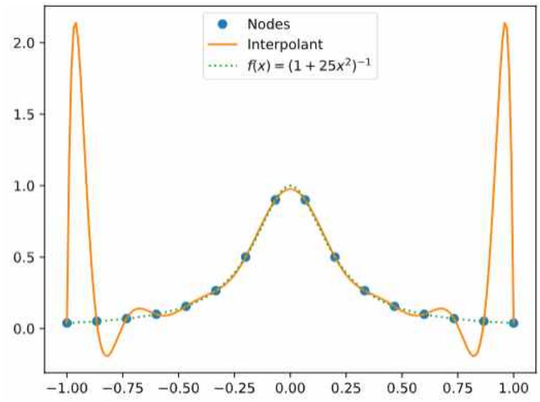


3.3. Runge phenomenon : An unexpected behavior that can occur when trying to approximate certain functions using polynomial interpolation. Take a function

2) Of course, there may be disadvantages, but in this paper, we try to include only the advantages that are necessary here.

and try to approximate it using a polynomial by picking a set of points and finding the polynomial that passes through these points (interpolation), one might intuitively expect that as you add more points (and therefore increase the degree of the interpolating polynomial), the approximation would get better everywhere. However, this is not always the case.

figure 3. *Example of Runge phenomenon*



4. Way of NumPy(C, math.h) calculates functions.

4.1. NumPy uses the computational methods found in the C language's math.h header file.

4.2. In math.h, calculations are structured based on data types (float128, float64, float32). However, the calculations here are strikingly similar.

4.3. math.h also defines numerical functions using the Taylor polynomial. To prevent the Runge phenomenon, it approximates on an interval-by-interval basis.

4.4. Additionally, there are instances where function values are returned using a dictionary (e.g., logarithm functions).

4.5. Functions can also be defined using pointer operations. Furthermore, square root operations are either incorporated in the ALU or use highly complex algorithms.

see examples : https://github.com/bminor/glibc/blob/master/sysdeps/ieee754/ldbl-128/e_asinl.c

5. Actual approximation : We already know that $e^x = \sum_{i=1}^{\infty} \frac{x^i}{i!}$ regarding value and differentiation. Infinite sum would guarantee those two important aspects but computer(needlessly human) can't handle infinite sum even if converges. So rather than use Taylor series, in here uses Taylor polynomial instead.

5.1. Taylor polynomial and adjusted coefficient

$$f(x) = \sum_{i=0}^{n} a_i x^i \cdots = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n + \cdots$$

If x is near 0, terms of high degree such as x^{10} are approximately zero, so we decided to define Taylor polynomial $a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n := \tilde{f}(x)$ since lower degree term is and the difference between f and $\tilde{f}$ as $f(x) - \tilde{f}(x) := error(x)$. We take adjusted coefficient by manually and gradient descent both. Source code is below.

```python
import numpy as np
# 11. tan
# tuning last two coefficients
# original function
def original_function(x):
    return np.tan(x)
# approximation, p and q are coefficient
def approximation_function(x, p, q):
    return x + (0.33333333333333333333333333333333) *x**3 +p * (0.1) *x **5 +q * (0.11) *x **7
# cost function
def cost(p, q):
    # since we want to approximate function on [0, 0.8]
    x =np.linspace(0, 0.8, 800)
    # approximation of integration; mensuration by parts
    # x[1]-x[0] = 0.001 since x = np.linspace(0, 0.8, 800) = [0.000, 0.001, 0,002, ...]
    return (x[1]-x[0]) *np.sum((original_function(x) -approximation_function(x, p, q))**2)
# gradient
def gradients(p, q):
    # h is very small positive number for f'(x)=(f(x+h)-f(x))/h
    h =1e-6
    # partial differentiation by p
    J =cost(p, q)
    J_p =cost(p +h, q)
    dp = (J_p -J) /h
    # partial differentiation py q
    J_q =cost(p, q +h)
    dq = (J_q -J) /h
    return dp, dq
"""
==========================================================================
|  lr = learning rate, eta                                                |
|  Note that initial value p, q where given coeffiecient                  |
|  by taylor approximation can approximate quite well, so p = q = 1       |
==========================================================================
"""
def gradient_descent(lr =0.005, num_iterations =2000):

    p, q =1, 1
    print("==========================================================================================")
    for i in range(num_iterations):
        dp, dq =gradients(p, q)
        p =p -lr *dp     # updating gradient
        q =q -lr *dq     # updating gradient
        # printing processes
        if i %200 ==0:
            print(f"Iteration {i:4d}: Cost = {cost(p, q):.16f}, p = {p:.13f}, q = {q:.13f}")
    return p, q
p, q =gradient_descent()
print("==========================================================================================")
print("OPTIMAL RESULT")
print(f"Optimal p: {p} -> coefficient_1 = {p*(0.1)}")
print(f"Optimal q: {q} -> coefficient_2 = {q*(0.11)}")
```

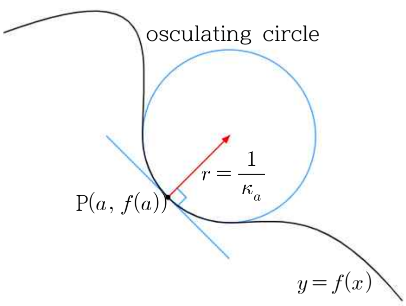
5.2. Tilde approximation[3] : In computer science and algorithm analysis, the tilde is used to describe an approximation, specifically when discussing asymptotics. Saying $f(n) \sim g(n)$ means that $f(n)$ and $g(n)$ are asymptotically equivalent. Formally, this means $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$. In this context, only the highest order term affects the result, and the others can be omitted. Here we got some idea and thought this wat : at point near zero, we take lower order term rather than higher order.

3) This notation helps to express that two functions grow at essentially the same rate, even if they might not be strictly equivalent in terms of big O, big Theta, or big Omega notations.

5.3. Horner’s method : A technique used to evaluate polynomial functions efficiently. Instead of calculating each term of the polynomial individually and summing them up, Horner's method reduces the number of multiplications and additions by leveraging a nested factorized form of the polynomial.

5.4. Osculating circle : The circle that has the same tangent and curvature as the curve at the given point. Because of this, it provides a good approximation of the curve near that point. If the function $f(x)$ is smooth at $x=a$ and curvature of the function $\kappa_a = k$, then osculating circle tangent to the curve at point $\mathrm{P}(a, f(a))$ and its radius $\frac{1}{k}$.

figure 4. *Osculating circle*



STRATAGIES
▶ Take domain of the approximation by $[-1, 1]$ since we want to use min-max scaling for convinience. Also, we can Also we don’t want Runge phenomenon.
▶ We need to adjust coefficient of polynomial.
▶ Since python is not that fast to find values in dictionary compared with C, we don’t use it.
▶ At some intervals, osculating circle works better considering square root operation.

6. Result

6.1. Efficiency test source code example

```
# %%
import numpy as np
import time
import random
# 1. arccos [-0.76, 0.76]
# constants for modified estimation
arccos_b1  =1
arccos_b3  =0.16666666666666666666666666666667
arccos_b5  =0.075
arccos_b7  =0.04464285714285714285714285714286
arccos_b9  =0.03038194444444444444444444444444
arccos_b11 =0.02734375
arccos_b13 =0.02208533653846153846153846153846
arccos_pi_over_2 =  1.5707963267948966192313216916397 5
# modified arccos function with degree 13, some modified coefficients
# For operation test on [-0.76, 0.76]
def arccos(n):
    p =n **2
    result =arccos_pi_over_2 -n * (arccos_b1 +p * (arccos_b3 +p * (arccos_b5 +p *
            (arccos_b7 +p * (arccos_b9 +p * (arccos_b11 +arccos_b13 *p))))))
    return result
```

```python
# Generate a list of random numbers
random_nums1 = [random.random() *0.76 for _in range(1000000)]
def time_test1(test_original, test_modified) :
    # testing cases
    exp_original_start =time.time()
    for num in random_nums1:
        test_original(num)
    exp_original_done =time.time()
    exp_modified_start =time.time()
    for num in random_nums1:
        test_modified(num)
    exp_modified_done =time.time()
    exp_time_comparision
    =100 *round((exp_modified_done-exp_modified_start)/(exp_original_done-exp_original_start), 2)
    # print
    print("original function elapsed time :", exp_original_done -exp_original_start)
    print("modified function elapsed time :", exp_modified_done -exp_modified_start)
    print(f'Result : {100 -exp_time_comparision}% time consumption saved')
    return None
time_test1(np.arccos, arccos)
```

6.2. Accuracy test source code example

```python
import numpy as np
import random
test_value_random =random.random()
# 7. approximation of  exponential x to the base 2 function(2^x or 2 ** x)
# constants for modified estimation
exp2_b0 =1
exp2_b1 =1
exp2_b2 =0.500000000000000000000000000000
exp2_b3 =0.168067226890756302521008403361
exp2_b4 =0.045454545454545454545454545455
exp2_log2 =0.693147180559945309417232121458
# modified exponential x to the base 2 function with degree 4, some modified coefficients
# For operation test
def exp2(n):
    # 0.693147180559945309417232121458 = log(2)
    n =exp2_log2 *n
    result =exp2_b0 +n * (exp2_b1 +n * (exp2_b2 +n * (exp2_b3 +n *exp2_b4)))
    return result
# ----------------Testing values-----------------------
test_original =np.exp2
test_approximation =exp2
test_value1 =0.13
test_value2 =0.35
test_value3 =0.57
print('7. approximation of y=2^x', '\n')
print('Testing value1 x=0.13')
print('original function :', test_original(test_value1))
print('modified function :', test_approximation(test_value1), '\n')
print('Testing value2 x=0.35')
print('original function :', test_original(test_value2))
print('modified function :', test_approximation(test_value2), '\n')
print('Testing value3 x=0.57')
print('original function :', test_original(test_value3))
print('modified function :', test_approximation(test_value3), '\n')
print(f"Testing value_random x={test_value_random}")
print('original function :', test_original(test_value_random))
print('modified function :', test_approximation(test_value_random))
```

6.3. Hypothesis test : Operating source code 6.1. we get around 30% improvement comparing original way(NumPy). We got several samples, and operated t-test. Statistical result is below. Every test shows that modified way lessen time under p<<.05.

table 1. Paired sample statistics

Paired sample statistics					
		Mean	N	Std. Deviation	Std. Error Mean
arccosine	NumPy (original)	.4490	37	.02255	.00371
	modified	.3081	37	.01959	.00322
arcsine	NumPy	.4420	32	.02036	.00360
	modified	.2824	32	.01212	.00214
hyperbolic arcsine	NumPy	.4671	50	.02452	.00347
	modified	.2661	50	.01525	.00216
cosine	NumPy	.4321	59	.02179	.00284
	modified	.2785	59	.01468	.00191
hyperbolic cosine	NumPy	.4470	59	.02893	.00377
	modified	.2174	59	.01251	.00163
exponential	NumPy	.4381	82	.02543	.00281
	modified	.1664	82	.01110	.00123
exponential (base 2)	NumPy	.4679	53	.03557	.00489
	modified	.1793	53	.01301	.00179
exponential - 1	NumPy	.4483	50	.02689	.00380
	modified	.1890	50	.01163	.00164
sine	NumPy	.4383	46	.02760	.00407
	modified	.2693	46	.01354	.00200
hyperbolic sine	NumPy	.4436	65	.02916	.00362
	modified	.2030	65	.01354	.00168
tangent	NumPy	.4362	54	.02312	.00315
	modified	.2248	54	.01268	.00173
hyperbolic sine	NumPy	.4769	66	.02791	.00344
	modified	.2302	66	.01982	.00244

table 2. Paired samples test

Paired Samples Test								
	Paired Differences					t	df	p(2-tailed)
	Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
				Lower	Upper			
difference between arccosine	.14089	.02331	.00383	.13312	.14866	36.762	36	.000
difference between arcsine	.15960	.01677	.00296	.15355	.16565	53.841	31	.000
difference between hyperboilic arcsin	.20098	.02334	.00330	.19435	.20761	60.896	49	.000
difference between cosine	.15368	.02264	.00295	.14778	.15958	52.128	58	.000
difference between hyperbolic cosine	.22956	.02795	.00364	.22227	.23684	63.092	58	.000
difference between exponential	.27165	.02487	.00275	.26619	.27712	98.901	81	.000
difference between exponential(base 2)	.28860	.03010	.00413	.28030	.29690	69.808	52	.000
difference between exponential-1	.25930	.02450	.00347	.25234	.26626	74.826	49	.000
difference between sine	.16897	.02550	.00376	.16139	.17654	44.933	45	.000
difference between hyperbolic sine	.24061	.02801	.00347	.23367	.24755	69.265	64	.000
difference between tangent	.21144	.02290	.00312	.20519	.21769	67.850	53	.000
difference between hyperbolic sine	.24673	.02561	.00315	.24043	.25302	78.266	65	.000