IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Comparative Study of ChatGPT Generated and Hand-Optimized Code for Different Applications

Daniel Niktash[1], Nader Bagherzadeh[2], Life Fellow, IEEE
[1]University High School, Irvine, CA 92612, USA
[2] Department of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697, USA

Corresponding author: D. Niktash (e-mail: dniktash@gmail.com).

ABSTRACT In recent years, natural language models have revolutionized software programming by enabling chatbots to generate code snippets through human-like text interactions. These models, trained on vast datasets, comprehend and generate text responses, making them useful tools for programmers. OpenAI's ChatGPT is one of the pioneers of Artificial Intelligence (AI) language models and is widely used in programming. However, the quality of generated codes is a major concern. In this study, the functionality and performance of Python codes generated by ChatGPT are examined and compared to its human-written hand-optimized code. Five different classes of algorithms were benchmarked. For sorting algorithms, ChatGPT was able to effectively generate functional code and generally performed 10-20% slower compared to its hand-optimized equivalent. This trend continued for the statistical analysis algorithm containing the mean, median, mode, and variance calculations where ChatGPT performed around 5-10% slower. When testing the linear regression algorithm, ChatGPT's performance ranges from 70% slower to match the hand-optimized performance. However, for the low-pass audio filter, ChatGPT performed around 18 times slower for all tested tap sizes. Finally, for the debayering algorithm, ChatGPT was unable to create functional code in most interpolation methods and performed between 10-50 times slower. Based on these experiments, ChatGPT may be reliable for simpler tasks, but as the complexity grows, the generated code is prone to functionality and syntax errors as well as significant speed degradation.

INDEX TERMS Audio Filter, Bayer Filter, ChatGPT, Debayering, FIR Filter, IIR Filter, Linear, Code Optimization, Regression, Python, Sort Algorithms, Statistical Analysis,

I. INTRODUCTION

ChatGPT, introduced in 2022, is a language model developed by OpenAI [1] that is capable of generating human-like text responses with prompting. One of the key features of ChatGPT is that it translates the natural language to code, making it an Artificial Intelligence (AI) programmer. Both professional programmers and beginners can greatly benefit from coding with ChatGPT. For beginners, interacting with ChatGPT provides an excellent opportunity to learn coding concepts conversationally and interactively.

Beginners can ask questions, seek clarification on programming principles, and receive instant feedback on their code, which accelerates the learning process. For professional programmers, ChatGPT serves as a valuable resource as well for troubleshooting a bug, exploring new algorithms, or refining code syntax. Additionally, ChatGPT can assist in generating pseudocode, providing code snippets, or suggesting best practices, saving time and enhancing productivity.

But how good is the code generated by ChatGPT? How is it compared with human-generated optimized code? Are generated codes free from syntax or functional errors? Are there differences in the efficiency of the code generated for different applications?

Since ChatGPT is very new, there are very prior works to explore these questions. In [2], the authors discuss how AI models are trained with publicly available existing codes and argue that an AI-generated code may not be held to a higher standard compared to human-generated codes.[3] provides some tips about using AI programming such as ways to approach prompting as well as being critical of generated code due to potential inaccuracies and errors. [4] provides examples of generated codes in Python, Javascript, and C++ without any discussion of the quality of the code. [5] tested ChatGPT with 184 programming exercises from an introductory bioinformatics course and reported the success rate. [6] proposed an iterative model to fine-tune instructions for guiding a chatbot in generating code for bioinformatics data analysis tasks.

In this article, we focus on Python coding language and are seeking to find answers to those questions by prompting ChatGPT to create Python codes for 5 different frequently used algorithms in various areas from mathematics and statistics to audio and image processing. The generated codes are tested and compared with human-generated optimized

ChatGPT 3.5

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quicksort(left) + middle + quicksort(right)
```

Hand-optimized

```python
def sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = [x for x in arr[1:] if x < pivot]
        right = [x for x in arr[1:] if x >= pivot]
        return sort(left) + [pivot] + sort(right)
```

FIGURE 1. Comparison of the code generated for Quicksort by ChatGPT-3.5 with a hand-optimized implementation.

ChatGPT 3.5

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    merged = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    while i < len(left):
        merged.append(left[i])
        i += 1

    while j < len(right):
        merged.append(right[j])
        j += 1
    return merged
```

Hand-optimized

```python
def merge_sort(seq):
    if len(seq) == 1:
        return seq
    left = merge_sort(seq[:len(seq) // 2])
    right = merge_sort(seq[len(seq) // 2:])

    return merge(left, right)

def merge(left, right):
    result = []
    left_count = 0
    right_count = 0
    try:
        while True:
            if left[left_count] > right[right_count]:
                result.append(right[right_count])
                right_count += 1
            else:
                result.append(left[left_count])
                left_count += 1
    except:
        return result + left[left_count:] + right[right_count:]
```

FIGURE 2. Comparison of the code generated for Merge Sort by ChatGPT-3.5 with a hand-optimized implementation.

ChatGPT 3.5

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Hand-optimized

```python
def sort(our_list):
    has_swapped = True

    num_of_iterations = 0

    while(has_swapped):
        has_swapped = False
        for i in range(len(our_list) - num_of_iterations - 1):
            if our_list[i] > our_list[i+1]:
                # Swap
                our_list[i], our_list[i+1] = our_list[i+1], our_list[i]
                has_swapped = True
        num_of_iterations += 1
    return our_list
```

FIGURE 3. Comparison of the code generated for Bubble Sort by ChatGPT-3.5 with a hand-optimized implementation.

code. The main criteria for comparison are code functionality and execution speed. Both ChatGPT 3.5 and 4.0 as well as Python Interpreter were explored. All codes are tested on a Lenovo Legion 5i PC with a Core i7-10750H 2.6GHz, 16GB memory with Windows 11 and results are reported. Python 3.8.8 was the interpreter tested in this experiment.

II. PROMPTING CHATGPT TO GENERATE CODE

Prompting ChatGPT to generate Python code involves providing clear and specific instructions. To begin, it's essential to define the problem clearly. In general, the input parameters, expected output, and any constraints are required to be specified. If a particular algorithm or approach is in mind, it needs to be explained briefly. Sometimes providing

relevant context helps the model understand the requirements better. In addition to the code, ChatGPT generates an example usage and a description/explanation of what the code does. If the initial response is not what is expected, we can refine the prompt based on the model's output or rephrase the prompt. Furthermore, it can also be challenged to correct a given response.

In this experiment, grammatical wording did not make a large difference in the results. However, descriptions such as "shorter" or "faster" did impact the outcome. In several instances, the code was not functional or had slight errors. Prompting again or challenging the answer by saying something such as "are you sure" often resolved the issue. In general, when the same prompt is repeated, ChatGPT may produce different responses. However, there are consistent results for very simple tasks or operations (e.g., multiplying numbers in a list).

III. CASE STUDY 1: SORTING ALGORITHMS

Sorting algorithms are crucial components of computer science, and their efficiency is often measured by their time complexity, O() [7], denoting the amount of time they require to process data. *Quicksort*, with an average-case time complexity of O(n log n), is highly efficient and widely used due to its speed. *Merge Sort*, also O(n log n), excels in stability and is commonly utilized in applications requiring predictable performance. *Bubble Sort*, though simple, has a worst-case and average-case time complexity of O(n^2), making it less suitable for large datasets. *Insertion Sort*, with an average-case time complexity of O(n^2), is adaptive and performs well for nearly sorted data. *Selection Sor*t, with a time complexity of O(n^2), is simple to implement but less efficient for large datasets. *Timsort*, a hybrid algorithm used in Python's sorting functions, combines Merge Sort and Insertion Sort, offering a reliable O(n log n) performance and excelling in diverse datasets. Understanding the complexities of these algorithms is crucial for selecting the appropriate sorting method based on the specific requirements of applications, ensuring optimal performance and resource utilization. In this study, each of the above-mentioned sort algorithms is investigated.

A. PROMPTING AND TEST SETUP

A general and simple prompt was used such as "Create a Python script for a sort algorithm". With the same prompt, each time, ChatGPT generated a completely different script using different sorting algorithms ranging from the basic sorted() function to longer Merge Sort algorithms. Also, even when the code for the same sorting algorithm was generated, there were slight variations in the code itself for the same algorithm. A follow-up prompt to “Make it faster”, resulted in ChatGPT generating a Quicksort algorithm. Each of these scripts generated by Chat GPT was functional and compared to equivalent hand-optimized implementation. For benchmarking, a fixed random seed was used to generate the same input data. The data type was integers, and 4 different sizes of unsorted random data were selected. The execution speed is measured with a PC running Windows 11 as described earlier.

B. QUICKSORT

This algorithm works by partitioning large arrays into smaller sub-arrays that are sorted and recombined. Figure 1 shows the code snippet generated by ChatGPT-3.5 and the hand-optimized implementation. Table 1 shows the run-time comparison for different data sizes. The hand-optimized Quicksort code is consistently faster and shows 20-30% shorter execution time.

TABLE I
COMPARISON OF THE RUN-TIME FOR QUICKSORT BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)
10k	0.017	0.014
100k	0.226	0.181
1M	3.370	2.727
10M	54.452	41.852

C. MERGE SORT

The Merge Sort algorithm works by dividing a list into smaller sublists that are sorted and merged to form a sorted list. Figure 2 demonstrates the code snippet generated by ChatGPT-3.5 and the hand-optimized implementation. Table 2 summarizes the run-time comparison for different data sizes. The hand-optimized Merge sort code is consistently faster and shows 25-40% shorter execution time. Also, by comparing Tables 1 and 2, the Merge sort is observed to be always slower than the Quicksort as expected.

TABLE II
COMPARISON OF THE RUN-TIME FOR MERGE SORT BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)
10k	0.026	0.020
100k	0.325	0.235
1M	4.200	3.148
10M	54.765	43.293

D. BUBBLE SORT

This algorithm works by repeatedly examining and swapping adjacent elements as it goes through the data from left to right. Figure 3 demonstrates the code snippet generated by ChatGPT-3.5 and the hand-optimized implementation. Table 3 summarizes the run-time comparison for different data sizes. As this algorithm is very slow, smaller sizes of input data were used for testing.In this example, the hand-optimized code performed slightly slower, about 4-5% compared to the code generated by ChatGPT. Also, by comparing Tables, 2 and 3, Bubble sort is observed to be significantly slower than the other sort algorithms as expected.

TABLE III

ChatGPT 3.5

```
def sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

Hand-optimized

```
def sort(argShuffledList):
    for index in range(1,len(argShuffledList)):

        currentvalue = argShuffledList[index]
        position = index

        while position>0 and argShuffledList[position-1]>currentvalue:
            argShuffledList[position]=argShuffledList[position-1]
            position = position-1

        argShuffledList[position]=currentvalue
    return argShuffledList
```

FIGURE 4. Comparison of the code generated for Insertion Sort by ChatGPT-3.5 with a hand-optimized implementation.

ChatGPT 3.5

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
```

Hand-optimized

```
def sort(l: list):
    arr = l[::]
    for i in range(len(arr)):
        m = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[m]:
                m = j
        if i != m:
            arr[i], arr[m] = arr[m], arr[i]
    return arr
```

FIGURE 5. Comparison of the code generated for Selection Sort by ChatGPT-3.5 with a hand-optimized implementation.

COMPARISON OF THE RUN-TIME FOR BUBBLE SORT BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)
1k	0.048	0.049
10k	5.205	5.495
25k	32.654	35.367
100k	528.617	563.795

E. INSERTION SORT

This simple algorithm works by iterating through an array and inserting it into its correct position one at a time, similar to the way to sort a deck of playing cards. For this example, the hand-optimized version is very similar to what ChatGPT originally generated. Figure 4 demonstrates the code snippet generated by ChatGPT-3.5 and the hand-optimized implementation. Table 4 summarizes the run-time comparison for different data sizes. As this algorithm is very slow, smaller sizes of input data were used for testing. For this algorithm, both ChatGPT's code and the hand-optimized version performed very similarly in speed with hand-optimized code slightly outperforming in most tests about 1-2%. This is due to their similarity in code structure. Also, by comparing Tables 1-3 and 4, Insertion sort is observed to be about 2x faster than Bubble sort and yet significantly slower than the other previously mentioned sort algorithms as expected.

TABLE IV
COMPARISON OF THE RUN-TIME FOR INSERTION SORT BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)
1k	0.024	0.023
10k	2.498	2.517
25k	15.880	15.241
100k	247.713	243.62

F. SELECTION SORT

This algorithm selects the smallest (or largest) element from an unsorted portion of a data set and moves it to the sorted position. Similar to Bubble sort and Insertion sort, the performance of this algorithm is known to significantly decrease with increasing data set sizes and has an O(n^2) time complexity regardless of how sorted the input is. Figure 5 demonstrates the code snippet generated by ChatGPT-3.5 and the hand-optimized implementation. Table 5 summarizes the run-time comparison for different data sizes. As this algorithm is very slow, smaller sizes of input data were used for testing.

For this algorithm, the ChatGPT-generated code performed slightly slower about 4% than the hand-optimized code. Also, by comparing Tables 1-4 and 5, Selection sort is observed to be about 3% faster than Bubble sort and yet

significantly slower than Quicksort and Merge sort as expected.

TABLE V

ChatGPT generated. This is also the fastest implementation of the sort algorithm compared to the other ones reported in previous sections.

ChatGPT 3.5

```
def timsort(arr):
    min_run = 32
    n = len(arr)

    for i in range(0, n, min_run):
        insertion_sort(arr, i, min((i + min_run - 1), (n - 1)))

    size = min_run
    while size < n:
        for start in range(0, n, size * 2):
            mid = start + size - 1
            end = min((start + 2 * size - 1), (n - 1))
            merge(arr, start, mid, end)
        size *= 2
    return arr
```

Hand-optimized

```
sorted_numbers = sorted(numbers)
```

FIGURE 6. Comparison of the code generated for Timsort by ChatGPT-3.5 with a hand-optimized implementation.

ChatGPT 3.5

```
def calculate_mean(numbers):
    total = sum(numbers)
    mean = total / len(numbers)
    return mean
def calculate_median(numbers):
    sorted_numbers = sorted(numbers)
    n = len(sorted_numbers)
    if n % 2 == 0:
        middle_1 = n // 2
        middle_2 = middle_1 - 1
        median = (sorted_numbers[middle_1] + sorted_numbers[middle_2]) / 2
    else:
        middle = n // 2
        median = sorted_numbers[middle]
    return median
def calculate_mode(numbers):
    count = Counter(numbers)
    max_count = max(count.values())
    mode = [num for num, freq in count.items() if freq == max_count]
    return mode
def calculate_variance(numbers):
    mean = calculate_mean(numbers)
    squared_diff = [(x - mean) ** 2 for x in numbers]
    variance = sum(squared_diff) / len(numbers)
    return variance
```

Hand-optimized

```
def calcmean(numbers):
    mean = sum(numbers)/len(numbers)
    return mean
def calcmedian(numbers):
    test_list =sorted(numbers)
    mid = len(test_list) // 2
    res = (test_list[mid] + test_list[~mid]) / 2
    return res
def calcmode(array):
        counts = Counter(array)
        maximum = max(counts.values())
        return [key for key, value in counts.items()
                if value == maximum]
def calcvariance(data, ddof=0):
    n = len(data)
    mean = sum(data) / n
    var = sum((x - mean) ** 2 for x in data) / (n - ddof)
    return var
```

Statistics Module

```
def calculate_mean(numbers):
    mean = statistics.mean(numbers)
    return mean

def calculate_median(numbers):
    median = statistics.median(numbers)
    return median

def calculate_mode(numbers):
    mode = statistics.mode(numbers)
    return mode

def calculate_variance(numbers):
    variance = statistics.variance(numbers)
    return variance
```

FIGURE 7. Comparison of the code generated for mean, median, mode, and variance by ChatGPT-3.5 with a hand-optimized implementation and statistics package.

COMPARISON OF THE RUN-TIME FOR SELECTION SORT BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION

Size	ChatGPT3.5 Time (sec)	Hand-optimized Tine (sec)
1k	0.0210	0.020
10k	2.116	2.083
25k	14.175	13.538
100k	240.733	231.884

G. TIMSORT

Timsort works by dividing the array into subsets that are sorted and merged. This algorithm is made using a hybrid of an Insertion sort and Merge sort algorithm. To reduce the number of comparisons, this algorithm exploits the original orientation of the data which helps to reduce time. Timsort is the default sort algorithm used in the Python sorted() function. For comparison purposes, the sorted() function is considered to represent the hand-optimized code and ChatGPT was prompted to generate a Timsort function as shown in Figure 6. As seen in Table 6, the sorted() function performs more than 10x faster than the coded version of Timsort generated by ChatGPT, which shows that this internal function is significantly optimized compared to what

TABLE VI
COMPARISON OF THE RUN-TIME FOR TIMSORT BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION

Size	ChatGPT3.5 Time (sec)	Hand-optimized Tine (sec)
10k	0.022	0.0009
100k	0.254	0.015
1M	3.670	0.285
10M	48.280	4.292

IV. CASE STUDY 2: STATISTICAL ANALYSIS

Statistical analysis is a vital aspect of data interpretation and decision-making in various fields, ranging from science and finance to social sciences. *Mean, median, mode, and variance* are fundamental statistical measures that offer valuable insights into data sets and are chosen for this experiment. The mean, often referred to as the average, provides a general overview of the data's central tendency. The median is the middle value of a dataset when it is sorted in ascending or descending order. Unlike the mean, the median is not affected by extreme values, making it a useful measure when dealing with skewed data distributions. The mode represents the most frequently occurring value in a dataset. Mode is particularly useful in categorical data

analysis. Variance, on the other hand, quantifies the spread or dispersion of data points around the mean. It measures how much each number in the dataset differs from the mean. Understanding these statistical measures helps researchers and analysts gain a deeper understanding of data patterns, enabling them to make informed decisions, draw meaningful conclusions, and identify trends within the data.

A. PROMPTING AND TEST SETUP

When prompting ChatGPT, phrases such as "Create a Python script to find the mean of a list" were chosen. Additionally, it can be prompted to create all four measures at once by asking to "generate a Python script to find the mean, median, mode, and variance of a list."
By default, ChatGPT created the code using the statistics package which is a single-line code that calls the relevant function for each measure. However, when asked to not use this package, it created its own coded algorithms for each measure which was also used in the comparison.
For testing, I used a fixed random seed to ensure that the generated data values are the same across all tests.

For benchmarking, four different sizes of data ranging from 100,000 to 100,000,000 values were selected. Additionally, both a random (uniform) distribution and normal distribution of data values were tested as input. The uniform distribution data were integer values generated between 1 and 10000. The normal distribution data were floating-point values with mean = 5000.0 and Standard Deviation = 1500.0.

B. MEAN, MEDIAN, MODE, AND VARIANCE

Figure 7 shows the code snippet generated by ChatGPT-3.5 and the hand-optimized implementation as well as function call using the statistics package of Python. Table 7 and 8 summarize the run-time comparison for different cases.As shown in Tables 7 and 8, ChatGPT and hand-optimized code both exhibited nearly identical speeds, although the hand-optimized version had a slight edge in average performance. Surprisingly, the built-in Statistics module consistently required significantly more time for all tests, underscoring its slower processing speed. Moreover, when testing with normal distribution data values using floating points, the processing time was over three times longer in contrast to the uniform distribution data using integers, highlighting the impact of data type and distribution on computational efficiency.

TABLE VII
COMPARISON OF THE RUN-TIME FOR MEAN CALCULATION BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION AND STATISTICS PACKAGE WITH UNIFORM DISTRIBUTION INTEGER DATA

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)	Statistics Package Time (sec)
100k	0.00099	0.00099	0.0009
1M	0.015	0.013	0.015
10M	0.188	0.197	0.285
100M	2.116	2.041	4.292

TABLE VIII
COMPARISON OF THE RUN-TIME FOR MEAN CALCULATION BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION AND STATISTICS PACKAGE WITH NORMAL DISTRIBUTION FLOATING POINT DATA

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)	Statistics Package Time (sec)
100k	0.0060	0.0051	0.102
1M	0.059	0.064	0.956
10M	0.574	0.587	9.772
100M	5.968	5.953	97.212

Tables 9-12 show the execution time for calculating the Median and Mode. When testing the calculation of Median and Mode with uniform distribution integer data, ChatGPT, hand-optimized, and the Statistics Module all performed at remarkably similar speeds. With normal distribution floating point data as input, all three experienced a notable slowdown in processing. However, the Statistics Module surpassed ChatGPT and the hand-optimized code, indicating a specific improvement in handling floating point data with normal distribution.

TABLE IX
COMPARISON OF THE RUN-TIME FOR MEDIAN CALCULATION BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION AND STATISTICS PACKAGE WITH UNIFORM DISTRIBUTION INTEGER DATA

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)	Statistics Package Time (sec)
100k	0.013	0.012	0.012
1M	0.187	0.171	0.175
10M	1.928	2.026	1.933
100M	20.256	20.199	20.273

TABLE X
COMPARISON OF THE RUN-TIME FOR MEDIAN CALCULATION BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION AND STATISTICS PACKAGE WITH UNIFORM DISTRIBUTION INTEGER DATA

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)	Statistics Package Time (sec)
100k	0.031	0.031	0.030
1M	0.545	0.487	0.462
10M	7.175	7.103	7.412
100M	104.064	104.998	94.523

TABLE XI
COMPARISON OF THE RUN-TIME FOR MODE CALCULATION BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION AND STATISTICS PACKAGE WITH UNIFORM DISTRIBUTION INTEGER DATA

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)	Statistics Package Time (sec)

100k	0.0060	0.0059	0.0059
1M	0.0591	0.0555	0.05455
10M	0.671	0.647	0.654
100M	6.734	6.758	6.798

The test result for calculating Variance is reported in Tables 13 and 14. In both data types, the hand-optimized code demonstrated superior speed, outperforming both ChatGPT, which operated slightly slower, and the Statistics Module, which required significantly more time for processing.

TABLE XII
COMPARISON OF THE RUN-TIME FOR MODE CALCULATION BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION AND STATISTICS PACKAGE WITH NORMAL DISTRIBUTION FLOATING POINT DATA

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)	Statistics Package Time (sec)
100k	0.0210	0.0200	0.0210
1M	0.335	0.277	0.251
10M	3.984	3.797	3.679
100M	63.0714	57.814	50.537

TABLE XIII
COMPARISON OF THE RUN-TIME FOR VARIANCE CALCULATION BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION AND STATISTICS PACKAGE WITH UNIFORM DISTRIBUTION INTEGER DATA

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)	Statistics Package Time (sec)
100k	0.0170	0.0149	0.1860
1M	0.185	0.173	1.762
10M	1.9165	1.698	17.470
100M	19.095	16.850	172.716

TABLE XIV
COMPARISON OF THE RUN-TIME FOR VARIANCE CALCULATION BY CHATGPT-3.5 WITH A HAND-OPTIMIZED IMPLEMENTATION AND STATISTICS PACKAGE WITH NORMAL DISTRIBUTION FLOATING POINT DATA

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)	Statistics Package Time (sec)
100k	0.0285	0.02801	0.317
1M	0.292	0.282	3.243
10M	2.928	2.804	31.958
100M	29.910	27.599	320.772

V. CASE STUDY 3: LINEAR REGRESSION

Linear regression is a statistical technique used to model the relationship between two variables by fitting a linear first-degree function based on the observed data. The line of best fit is generated by minimizing the mean square error of the data. This linear equation can be used for predicting the values of one variable based on the other and can help with making predictions and understanding the direction of the relationship. This method of regression is very simple due to its linearity assumption, which assumes that a change in one variable will result in a constant change in the other. Linear regression is widely used in various fields such as economics, finance, engineering, biology, and social sciences.

A. PROMPTING AND TEST SETUP

ChatGPT was prompted to "Create a Python script to find the linear regression of a data set" Originally, ChatGPT created the code using the Sklearn package to create the regression. However, when asked to not use this module, it created its own coded algorithms for each measure. Both codes are used in the experiment and are compared to the optimized ScPy package which is a single-line code that calls the linear regression function from the package

In this experiment, a fixed random seed was used to ensure that the generated data values were the same across all tests. For benchmarking, five different sizes of data from 100 to 1,000,000 were selected. The data was uniformly distributed with ±10 variation. The installed Sklearn and SciPy packages for this experiment were 1.3.0 and 1.10.1 respectively.

A. LINEAR EQUATION

Figure 8 shows the code snippets generated by ChatGPT-3.5 with and without the Sklearn package. The hand-optimized implementation is also shown using the imported SciPy package. Table 15 summarizes the run-time comparison for different cases and the ratio of ChatGPT 3.5 to hand-optimized time.

The result of this experiment is demonstrated in Table 15. ChatGPT utilized the Sklearn module while the hand-optimized version employed the SciPy module. As shown, the hand-optimized script exhibited significantly better performance, operating at speeds 2-4 times faster than ChatGPT. Surprisingly, the ChatGPT code without Sklearn performed faster compared to the hand-optimized code with Scipy. The difference decreases as the size of the data grows. The difference in performance might be attributed to the additional error-checking and overhead associated with calling a module, affecting the efficiency of the ChatGPT implementation in larger-scale data processing scenarios.

TABLE XV
RUN-TIME COMPARISON FOR DIFFERENT SIZES FOR LINEAR REGRESSION

Size	ChatGPT3.5 Time (sec)	Hand-optimized Time (sec)	ChatGPT w/o Package Time (sec)
100k	0.408	0.0805	0.0255
1M	0.428	0.0914	0.0320
10M	0.530	0.157	0.0745
100M	3.212	1.060	1.765

VI. CASE STUDY 4: AUDIO FILTER

ChatGPT 3.5

```
def calculate_linear_regression(x_values, y_values):
    # Create a Linear Regression object
    regression = LinearRegression()

    # Reshape the input arrays to match the expected format
    x_values = np.array(x_values).reshape(-1, 1)
    y_values = np.array(y_values).reshape(-1, 1)
    # Fit the regression model to the data
    regression.fit(x_values, y_values)

    # Extract the slope and intercept of the linear regression line
    slope = regression.coef_[0][0]
    intercept = regression.intercept_[0]

    return slope, intercept
```

ChatGPT w/o Modules

```
def calculate_linear_regression(x, y):
    # Compute the mean of x and y
    mean_x = np.mean(x)
    mean_y = np.mean(y)

    # Calculate the terms needed for the num and denom of beta
    num = np.sum((x-mean_x)*(y-mean_y))
    denom = np.sum((x-mean_x)**2)

    # Calculate beta coefficients
    beta1 = num / denom
    beta0 = mean_y - (beta1 * mean_x)
    return beta1, beta0
```

Hand Optimized

```
def linearreg(x,y):
    slope, intercept, _, _, _ = stats.linregress(x, y)
    return slope, intercept
```

FIGURE 8. Comparison of the code generated for linear regression by ChatGPT-3.5 with and without packages and a hand-optimized implementation.

Audio filters are widely used in digital signal processing and play a major role in audio electronics as well as telecommunication. There are two main types of audio filters: Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters [8]. FIR filters have a finite duration impulse response, meaning they only consider a finite window of input samples to calculate the output. Some design methods include the Bartlett filter, Hanning filter, and Kaiser filter. On the other hand, IIR filters have an infinite duration impulse response, allowing feedback within the filter structure. Some examples of IIR filter designs include the Butterworth, Chebyshev, and Elliptic filters. In digital filters, "taps" refer to the individual coefficients or weights applied to the input samples in the filter's algorithm. The number of taps in a filter corresponds to the filter's order and determines how many past input samples are considered to calculate the current output sample. There are four main types of filters: low-pass, high-pass, band-pass, and notch filters. Each of these types serves unique purposes in signal processing [8]. In this experiment, the IIR Butterworth low-pass filter, a very common type of filter, was focused on for benchmarking.

A. PROMPTING AND TEST SETUP

ChatGPT was prompted to "generate a Python script to make white noise, apply a low pass filter to it, and then save the output." By default, ChatGPT used the Butterworth filter which was tested. The generated code to design the filter is shown in Figure 9. When prompted specifically, ChatGPT was also able to create other different filter designs with some having errors and some correctly running. Originally, ChatGPT used the SciPy package (tested with version 1.10.1) which also has optimized filter functions that are generally used in hand-optimized filter code It was also asked to create a coded version instead of using the SciPy package and it created its own coded script. The result of this is compared to the optimized SciPy package which represents the hand-optimized interpretation.

For input, a 5-second white noise input sampled at 44.1 kHz was used. The filter tested was an IIR Butterworth low-pass filter with a cutoff frequency of 1 kHz. Four different tap sizes were benchmarked and the total time to apply the filter is recorded as shown in Table 16.

B. BUTTERWORTH LOW-PASS FILTER

Figure 10 shows the low-pass Butterworth filter design code with a cutoff frequency of 1 kHz. Figure 10 shows the SciPy and ChatGPT coded applications of the filters to the input data. Table 16 summarizes the run-time comparison of both scripts for tap values 1, 2, 4, and 8.

```
# Define the filter
nyquist_rate = samplerate / 2.
cutoff_frequency = 1000.0  # Set your cut-off frequency here
normalized_cutoff = cutoff_frequency / nyquist_rate

# Use butter to create a low-pass filter
b, a = signal.butter(1, normalized_cutoff, btype='low')
```

FIGURE 9. Filter design code

Application of filter without modules

```
filtered_noise = np.zeros(len(noise))
for i in range(1, len(noise)):
    filtered_noise[i] = b[0] * noise[i] + b[1] * noise[i-1] - a[1] * filtered_noise[i-1]
```

Application of filter using SciPy Signal filter

```
filtered_noise = signal.lfilter(b, a, noise)
```

FIGURE 10. Application of both filters to the input audio file

The performance result is depicted in Table 16. The execution time for the application of the filter using the module was about 17x faster. The number of taps did not affect the execution time in the SciPy implementation of the filter. In the ChatGPT coded script, increasing the number of taps slightly increased the execution time.

TABLE XVI
RUN-TIME COMPARISON FOR DIFFERENT NUMBER OF TAPS IN A LOW-PASS FILTER IMPLEMENTATION

Number of Taps	SciPy Package Time (ms)	Coded ChatGPT Filter Time (ms)

1	0.989	17.102
2	0.989	18.603
4	0.989	19.999
8	0.989	19.051

VII. CASE STUDY 5: DEBAYERING

The Bayer format is a popular color filter array pattern used in most digital imaging devices, including digital cameras and smartphone cameras [9]. In this format, when an image is captured, individual pixels on an image sensor are covered with color filters in a specific arrangement. This Bayer pattern consists of alternating rows of green and red filters, and blue and green filters, which creates a mosaic of RGB values [10]. Figure 11 shows an example of the Bayer pattern.

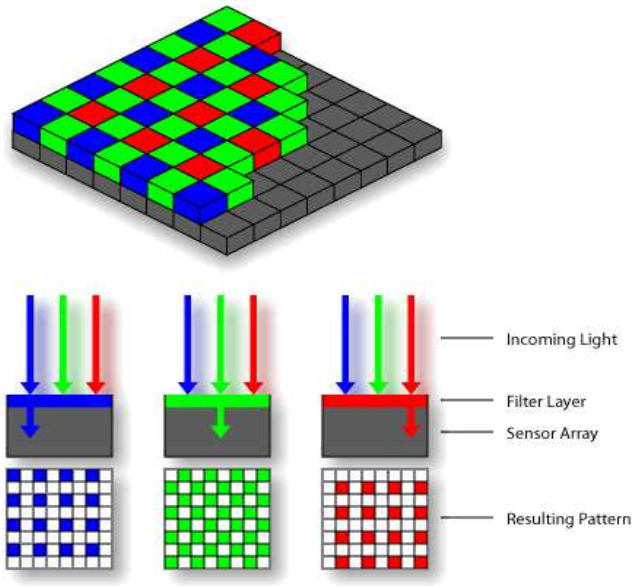

FIGURE 11. Bayer pattern (image taken from [11] without any changes under Creative Commons licensing)

This format is used because it requires less memory and bandwidth, captures spatial details better, and reduces the cost of producing camera sensors. To produce a full-color image based on this Bayered image, a process called debayering (or Demosaicing) is applied. Debayering algorithms interpolate the missing color information in each pixel by analyzing the neighboring pixels' color values. After debayering, a complete and detailed color image can be created. Debayering techniques vary in complexity, ranging from simple methods such as nearest-neighbor interpolation to more advanced algorithms like bilinear, bicubic, and directional interpolation. These four debayering methods were tested in this case study.

A. PROMPTING, BAYERING, AND DEBAYERING

In order to test the debayering, first an image is converted to Bayer format using ChatGPT to represent the raw Bayer images captured by image sensors by prompting "Create a Python script to convert an RGB image to Bayer". The generated code shown in Figure 12 was then tested on the pure red, green, and blue images to verify its correctness. As demonstrated in Figure 13, only one color is present in the correct pixel location for each image as expected.

```
def rgb_to_bayer_display(img):
    """
    pick bayer channels only
    """
    img = np.array(img)
    # red only
    img[0::2, 0::2, 1] = 0
    img[0::2, 0::2, 2] = 0

    #green only
    img[0::2, 1::2, 0] = 0
    img[0::2, 1::2, 2] = 0

    img[1::2, 0::2, 0] = 0
    img[1::2, 0::2, 2] = 0

    # blue only
    img[1::2, 1::2, 0] = 0
    img[1::2, 1::2, 1] = 0



    plt.figure(figsize=(10, 10))
    plt.imshow(img)
    plt.axis('off')
    plt.title("Bayer Patterned Mona Lisa Image")
    plt.show()

    return
```

FIGURE 12. ChatGPT generated script to convert an RGB image to Bayer

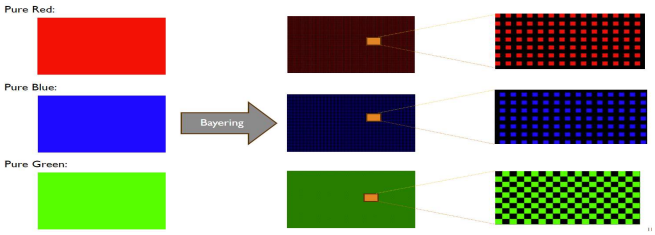

FIGURE 13. Applying generated rgb_to_bayer_display function on pure color image to verify

F\

IGURE 14. Bayered image of Mona Lisa (original colored image was taken from [12] with public domain licensing)

Next, the same function is applied to convert a colored image to Bayer format to be used as the test image as shown in Figure 14. This image is used for testing each of the Debayering methods. For each test, ChatGPT was prompted to create a Python code with a specific debayering algorithm such as "Create a Python script to Debayer a Bayered image using nearest-neighbor interpolation".

B. NEAREST NEIGHBOR INTERPOLATION

The Nearest Neighbor interpolation is the simplest form of interpolation used in debayering, where the value of a missing pixel is assigned the value of the nearest neighboring pixel. This method is generally the fastest but can result in artifacts, especially in images with sharp edges. After being prompted, ChatGPT was able to make an error-free and functional code successfully as depicted in Figure 15. The code was tested on the Bayered image shown in Figure 16 and the debayered image was very accurate compared to the original color image as demonstrated in Figure 17.

```
def debayer_nearest_neighbor(bayered_array):

    # Allocate an array for the RGB image
    rgb_image = np.zeros((bayered_array.shape[0],
                          bayered_array.shape[1], 3),
                         dtype=bayered_array.dtype)

    # R channel
    rgb_image[0::2, 0::2, 0] = bayered_array[0::2, 0::2]
    rgb_image[1::2, 0::2, 0] = bayered_array[0::2, 0::2]
    rgb_image[0::2, 1::2, 0] = bayered_array[0::2, 0::2]
    rgb_image[1::2, 1::2, 0] = bayered_array[0::2, 0::2]

    # G channel
    rgb_image[0::2, 0::2, 1] = bayered_array[0::2, 1::2]
    rgb_image[1::2, 0::2, 1] = bayered_array[1::2, 0::2]
    rgb_image[0::2, 1::2, 1] = bayered_array[0::2, 1::2]
    rgb_image[1::2, 1::2, 1] = bayered_array[1::2, 1::2]

    # B channel
    rgb_image[0::2, 0::2, 2] = bayered_array[1::2, 1::2]
    rgb_image[1::2, 0::2, 2] = bayered_array[1::2, 1::2]
    rgb_image[0::2, 1::2, 2] = bayered_array[1::2, 1::2]
    rgb_image[1::2, 1::2, 2] = bayered_array[1::2, 1::2]

    # Convert the array back to an image and save
    rgb_image_pil = Image.fromarray(rgb_image)
    rgb_image_pil.save('temp.jpg')
    return rgb_image
```

FIGURE 15. ChatGPT generated script to convert an RGB image to Bayer using Nearest Neighbor Interpolation

FIGURE 16. The result of Debayering with the ChatGPT generated script using Nearest Neighbor Interpolation

C. BILINIEAR INTERPOLATION

In this form of interpolation, the average of the nearest 2x2 neighborhood of known pixel values around a pixel is used for the missing colors. This method provides smoother results than nearest-neighbor but may blur sharp edges.

ChatGPT was prompted: "Create a Python script to Debayer a Bayered image using bilinear interpolation". The created code was incorrect code and did not execute. After repeating a similar prompt a few times and providing the errors that the code produced, it was finally able to create a functional code. However, the debayered image was incorrect with a heavy green shade. Next, the ChatGPT was challenged by prompting “Are you sure”, or “Double check”, but the result was still not correct. Using the code interpreter feature of ChatGPT 4.0 also produced a similar result. At last, the code interpreter was given the expected output image (the original colored image) to be used as the reference to fix the code. However, the produced code was not able to correctly debayer the image. The generated code and converted image are shown in Figures 17 and 18.

```python
def debayer_bilinear(bayered_array):
    # Allocate an array for the RGB image
    rgb_image = np.zeros((bayered_array.shape[0],
                          bayered_array.shape[1], 3), dtype=bayered_array.dtype)
    # R channel
    r_mask = np.zeros_like(bayered_array)
    r_mask[0::2, 0::2] = 1
    rgb_image[:, :, 0] = r_mask * bayered_array
    for i in range(1, bayered_array.shape[0] - 1, 2):
        for j in range(1, bayered_array.shape[1] - 1, 2):
            rgb_image[i, j, 0] = (int(bayered_array[i - 1, j])
                                  + int(bayered_array[i + 1, j])) // 2
    # G channel
    g_mask = np.zeros_like(bayered_array)
    g_mask[0::2, 1::2] = 1
    g_mask[1::2, 0::2] = 1
    rgb_image[:, :, 1] = g_mask * bayered_array
    for i in range(1, bayered_array.shape[0] - 1, 2):
        for j in range(1, bayered_array.shape[1] - 1, 2):
            rgb_image[i, j, 1] = (int(bayered_array[i, j - 1])
                                  + int(bayered_array[i, j + 1])) // 2
    # B channel
    b_mask = np.zeros_like(bayered_array)
    b_mask[1::2, 1::2] = 1
    rgb_image[:, :, 2] = b_mask * bayered_array
    for i in range(0, bayered_array.shape[0] - 1, 2):
        for j in range(0, bayered_array.shape[1] - 1, 2):
            rgb_image[i, j, 2] = (int(bayered_array[i + 1, j])
                                  + int(bayered_array[i, j + 1])) // 2
    # Convert the array back to an image
    return Image.fromarray(rgb_image)
```

FIGURE 17. ChatGPT generated script to convert an RGB image to Bayer using Bilinear Interpolation. The converted image was not correct.

FIGURE 18. The result of Debayering with the ChatGPT generated script using Bilinear Interpolation. The image has a heavy shade of green.

D. DIRECTIONAL INTERPOLATION

In this method, the missing color in a pixel is estimated by considering the direction of edges or features in the image to reduce the artifacts along the edges.

After being prompted to "Create a Python script to Debayer a Bayered image using directional interpolation", ChatGPT generated a non-functional script with multiple errors. Next, it was challenged with additional prompts to fix the errors dozens of times. However, it was still not able to generate a functional code.

E. BICUBIC INTERPOLATION

In Bicubic Interpolation, the values of the nearest 4x4 neighborhood pixels are used to estimate the missing colors. This method generally produces smoother images and preserves edges better. The trade-off is the computational complexity of this method compared to the other simpler algorithms.

By prompting ChatGPT to "Create a Python script to Debayer a Bayered image using Bicubic interpolation", and challenging it several times, it was finally able to make a functional code as reported in Figure 19. However, the debayered image was not correct and had a red tint as displayed in Figure 20.

```python
def debayer_bicubic(bayered_image_path):
    # Load the image using PIL
    bayered_image = Image.open(bayered_image_path)
    bayered_array = np.array(bayered_image)

    # Extract the partial channels from the Bayer pattern
    R = bayered_array[0::2, 0::2]
    G1 = bayered_array[0::2, 1::2]
    G2 = bayered_array[1::2, 0::2]
    B = bayered_array[1::2, 1::2]

    # Upscale each channel using bicubic interpolation
    R_upscaled = zoom(R, 2, order=3)
    G1_upscaled = zoom(G1, 2, order=3)
    G2_upscaled = zoom(G2, 2, order=3)
    B_upscaled = zoom(B, 2, order=3)

    # Merge the channels
    rgb_image = np.zeros((bayered_array.shape[0], bayered_array.shape[1], 3), dtype=np.uint8)
    rgb_image[:,:,0] = R_upscaled
    rgb_image[0::2, 1::2, 1] = G1_upscaled[0::2, 1::2]
    rgb_image[1::2, 0::2, 1] = G2_upscaled[1::2, 0::2]
    rgb_image[:,:,2] = B_upscaled

    # Interpolate missing Green values
    for i in range(1, bayered_array.shape[0]-1, 2):
        for j in range(1, bayered_array.shape[1]-1, 2):
            rgb_image[i, j, 1] = (int(rgb_image[i, j-1, 1]) + int(rgb_image[i, j+1, 1]) +
                                  int(rgb_image[i-1, j, 1]) + int(rgb_image[i+1, j, 1])) // 4
    return Image.fromarray(np.clip(rgb_image, 0, 255))
```

FIGURE 19. ChatGPT generated script to convert an RGB image to Bayer using Bicubic interpolation. The converted image was not correct.

FIGURE 20. The result of Debayering with the ChatGPT generated script using Bicubic Interpolation. The image has a shade of red.

F. USING OPENCV AND SCIPY

OpenCV and SciPy are two of the most common packages that include optimized functions for debayering. By default, they both use Bilinear Interpolation, and they are frequently used in hand-optimized code implementation of Debayering implementations.

In this exercise, ChatGPT's coded Bilinear Interpolation algorithm without using any packages is compared to optimized SciPy (1.10.1) and OpenCV (4.8.0.74) implementations. The Debayering speed is measured for four image sizes and the result is reported in Table 17. Compared to ChatGPT's Bilinear coded script, SciPy performed about 10x faster and OpenCV performed around 50x faster.

TABLE XVII
RUN-TIME COMPARISON OF BILINEAR INTERPOLATION IMPLEMENTED BY CHATGPT CODED SCRIPT, OPENCV, AND SCIPY FOR DIFFERENT IMAGE RESOLUTIONS

Resolution	ChatGPT Time (sec)	OpenCV Time (sec)	SciPy Time (sec)
854x480	0.141	0.0029	0.0140
1280x720	0.340	0.0070	0.0280
1920x1080	0.735	0.0129	0.0625
3840x2160	2.848	0.0575	0.2375

VIII. CONCLISON

ChatGPT is a new AI language model by OpenAI, capable of creating code based on descriptive prompts provided by the user. In this study, the functionality and speed of ChatGPT's Python code and its hand-optimized equivalent were compared for five algorithms across different disciplines. For simpler algorithms like sorting and linear regression, ChatGPT demonstrated a performance level closely mirroring that of hand-optimized code. However, as the complexity of algorithms increased, ChatGPT exhibited slower speeds, a higher likelihood of errors in the code, and at times, an inability to create the desired output. Notably, in the debayering case study, the most challenging algorithm in this experiment, ChatGPT often failed to produce functional code even with assistance.

REFERENCES

[1] https://openai.com
[2] J.Vaydya, H.Asif, "A Critical Look at AI-generated Software", *IEEE Spectrum for the Technology Insider*, Jun. 11, 2023, Available: https://spectrum.ieee.org/ai-software
[3] R.D. Caballar, "How Coders Can Survive—and Thrive—in a ChatGPT World", *IEEE Spectrum for the Technology Insider*, Jul. 3, 2023, Available: https://spectrum.ieee.org/ai-programming
[4] Zibtek-Custom Software Development Company, "Just How Good is ChatGPT at Writing Code?", Available: https://www.linkedin.com/pulse/just-how-good-chatgpt-writing-code-zibtek
[5] S.R.Piccolo et al. "Many bioinformatics programming tasks can be automated with ChatGPT", Preprint, Mar. 7, 2023, Available: https://arxiv.org/abs/2303.13528
[6] E. Shue, et al. Quant. Biol. "Empowering beginners in bioinformatics with ChatGPT", Quantitative Biology, Vol. 11, Issue 2, 2023, Available: https://doi.org/10.15302/J-QB-023-0327
[7] https://en.wikipedia.org/wiki/Big_O_notation
[8] J. O. Smith III, "Introduction to Digital Filters with Audio Applications", Center for Computer Research in Music and Acoustics (CCRMA), Department of Music, Stanford University, California 94305 USA, Available: https://www.dsprelated.com/freebooks/filters/
[9] https://www.red.com/red-101/bayer-sensor-strategy
[10] https://en.wikipedia.org/wiki/Bayer_filter#cite_note-1
[11] https://commons.wikimedia.org/wiki/File:BayerPatternFiltration.png
[12] https://en.wikipedia.org/wiki/File:Mona_Lisa,_by_Leonardo_da_Vinci,_from_C2RMF_retouched.jpg

DANIEL NIKTASH is a junior in University Highschool, Irvine with dual enrollment at Irvine Valley College. His interests include math, science, and code development. He also has a passion for exploring new technologies in artificial intelligence and AI generated codes. In the summer of 2023, he conducted research on the quality of AI-generated codes using ChatGPT as the platform at University of California, Irvine showcasing algorithms from different disciplines which is presented in this work.

NADER BAGHERZADEH (Life Fellow, IEEE) received the PhD degree from the University of Texas at Austin, in 1987. He is a professor of computer engineering with the Department of Electrical Engineering and Computer Science, University of California, Irvine, where he served as a chair from 1998 to 2003. He has been involved in research and development in the areas of computer architecture, reconfigurable computing, VLSI chip design, network on chip, 3D chips, sensor networks, computer graphics, memory and embedded systems. Dr. Bagherzadeh has published more than 325 articles in peer-reviewed journals and conferences. His former students have assumed key positions in software and computer systems design companies in the past 30 years.