Research Report

Mark Saleh 103994313

# Abstract

# The aim of this research project was to compare the effectiveness of procedural and object-oriented programming (OOP) paradigms in solving repetitive mathematical functions. The research focused on implementing Gaussian elimination, integration, Newton's method, and a sorting algorithm in both procedural and OOP approaches using Python. The performance of each paradigm was evaluated based on criteria such as speed, code length, elegance, readability, and memory usage.

# Introduction

Within programming, software is guided by paradigms. Procedural and object-oriented programming (OOP) are the two most common such paradigms. Procedural programming is organising the code into procedures, which perform functions (often on objects), while OOP has the user create classes, each with their own variables and method, which then create the objects (whence it’s called “object oriented” in contrast to “function oriented” code).

Procedural and OO programming both have their strengths and weaknesses, and understanding them is crucial for maximising the efficiency, speed, elegance, and conciseness of a program. In this research project, I aim to compare the effectiveness of both paradigms at solving repetitive mathematical functions. Since these functions are generally more about repeating the same functions with different numbers than repeating different functions on the same numbers, I hypothesise that procedural programming will prove to be more suited for this application.

# Method

This research project utilised several different mathematical processes, including Gaussian elimination, Newton’s method, sorting, and integration. Both paradigms were awarded points for speed, code length, readability/elegance, memory usage, and any additional differences that stick out. The comparisons were done in Python, as it is suitable for both procedural and OO programming, and allows for the easy implementation of mathematical functions in both of them. For the memory use, a library called “memory profiler” was downloaded to allow the memory to be measured.

# Results

## Gaussian Elimination

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Speed | Code length | Readability | Memory Use | Extra points | Total |
| Procedural | 1 | 1 |  | 0.5 |  | 2.5 |
| OO |  |  | 1 | 0.5 |  | 1.5 |

The procedural implementation of Gaussian elimination had about 3 less lines of code than the OO one, as the OO version had to define the methods “eliminate” and “solve” within its class, and had to call the solve method in its eliminate one. The procedural one was also consistently about 10-15% faster than the OO (and this rate remains the same as the size of the matrix increases). This could be because Gaussian elimination involves extensive matrix operations, which were not well implemented in OOP. However, it was somewhat difficult to read compared to the OO implementation, which separated the steps and so was clearer. Both paradigms tended to use similar levels of memory usage according to the memory profiler.

## Integration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Speed | Code length | Elegance | Memory Use | Extra points | Total |
| Procedural |  | 1 | 1 |  |  | 2 |
| OO | 1 |  |  | 1 |  | 2 |

Integration (by calculating the area of thousands of rectangular “slices” of a function) turned out to be surprisingly simple to implement. The OO implementation was consistently faster, although the difference in speed decreased as the number of slices increased. For 1000 slices, it was almost twice as fast. It also took significantly less memory usage (which is probably why it was faster). This could be because, unlike in the Gaussian code, there were no nested loops here, in which procedural programming performs better than OOP as they create a lot of overhead. However, the OO class required a constructor with several unwieldy variable declarations like this:

    def \_\_init\_\_(self, f, a, b, n):

        self.f = f

        self.a = a

        self.b = b

        self.n = n

And thus the procedural implementation uses less code and appears more elegant.

## Newton’s method

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Speed | Code length | Elegance | Memory Use | Extra points | Total |
| Procedural | 1 | 1 | 1 | 0.5 |  | 3.5 |
| OO |  |  |  | 0.5 | 1 | 1.5 |

Although the OOP did manage to be faster than the procedural code a few times, the procedural code was on average slightly faster. It also had much shorter code due to not needing a constructor. Since the function for Newton’s method was essentially just the “find\_root” method of the NewtonMethod class, it was faster, shorter, more elegant and used roughly the same amount of memory. However, as this was a resounding victory for procedural programming (and perhaps unwarrantedly so, as OOP was not far behind), it was decided to award one pity point to OOP.

## Sorting

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Speed | Code length | Elegance | Memory Use | Extra points | Total |
| Procedural | 1 | 1 | 1 |  |  | 3 |
| OO |  |  |  | 1 |  | 1 |

Similarly, to Newton’s method, the most concise way to code a sorting algorithm in OOP ended up being simply a class with a single method, which is the same as the function in the procedural implementation. This was not a result of failing to apply OOP concepts, but simply trying to make the code as short as possible. Due to this, the results for sorting were essentially the same as they were for Newton’s method. However, for some reason OOP used 4-5 times less memory than procedural programming here.

# Discussion

Over all four tests, procedural programming earned 11 points while OOP earned 6 points. This means that, overall, procedural programming is better than OOP for performing repetitive maths functions in Python. Some reasons for this include being more elegant (OOP code in Python uses the word “self” too often, and required constructors unlike procedural code), being faster, and the fact that multiple classes (or sometimes even multiple methods within a class) are simply not necessary for mathematical functions. Twice, the most concise way to code something using OOP was to make a single class with a single method, which performs the maths function required. This confirmed my hypothesis that, since repetitive maths functions are moreso about repeating the same functions with different numbers than repeating different functions on the same numbers, procedural programming will “win”.

However, this experiment also revealed some of OOP’s strengths. The separation of variables and methods meant that OOP generally used less memory than procedural programming, sometimes drastically so. This experiment certainly does not diminish OOP overall, but simply its use for repetitive mathematical functions. In situations where multiple classes are necessary, such as my implementation of Tetris, OOP’s ability to encapsulate classes certainly helped with, for example, performing methods on tetrominos without knowing which specific kind of tetromino they are, and with creating necessary objects such as blocks and tetrominos.

# Conclusion

This research project compared procedural and OOP paradigms in solving repetitive mathematical functions. The findings confirm that procedural programming is better suited for this specific application due to its elegance, speed, and simplicity. However, OOP demonstrated strengths in memory optimization and encapsulation, highlighting its value in more complex scenarios. Future work could involve comparing the application of different paradigms in different contexts, especially ones in which OOP would be expected to outperform procedural programming such as game design.