

# Analysis of Numerical Integration Implementations

Connor Churcott - July 24, 2024

## View All Code and Implementations:

<https://github.com/connorchurcott/Numerical-Integration-Analysis>

## The Code Being Analysed:

This paper aims to analyze the running times and overall efficiency of various implementations of numerical integration methods. The two integration methods tested are the Trapezoidal and Simpsons Methods. Each method will be analyzed twice using different implementations. The first implementations will be written by me, using only what I remember from Calculus 2 (presumably this will be the slow/inefficient implementation). The second implementations will either be from an online source or a well-thought-out version I wrote. All implementations will be written in C, unless stated otherwise.

All implementations of these methods will have the same function prototypes as seen below. The data parameter is a scaleable array, which houses all values  $f(x)$  for each sub-interval. Hence the larger the 'n' is the more subintervals you have, causing more calculations. This array is populated outside of the implementations and is not counted in the running time, meaning the implementations are tested for speed solely on their algorithm and are not dependent on the implementation of  $f(x)$ . With this setup, we should be able to accurately compare the speed of each implementation.

```
double integrationMethod(double* data, double lowBound, double highBound, int64_t n);
```

Note: The function being integrated with every implementation is  $f(x) = x^3$

## How it is Being Evaluated:

Each implementation will be timed using the `clock_gettime()` function. Both versions of the same method will use **Valgrind** to analyze cache misses, **Godbolt** to compare to the assembly, and **Perf** to analyze branch mispredictions. Every implementation will be tested with the same parameter values to keep the test fair, and I am choosing to test with a high 'n' value (big array) because the results will likely be more interesting. All results will be the average over three runs. The result times are calculated without running with Valgrind or Perf. In the end, I will compare the fastest implementations of each method to each other and see what is fastest. The following are the compiler commands and the parameter values being used for every implementation.

```
gcc -Wall -Wpedantic -Wextra -march=haswell  
-std=c17 main.c functions.c -o exe -O3
```

```
#define lowBound 0  
#define highBound 100000000  
#define n 500000000
```

## Trapezoidal Method Results:

My first implementation of this method was fairly straightforward, it was just a for loop that accessed memory and did some calculations within it. The Rosetta Code implementation is the same, so for the second implementation, I carefully wrote out a vectorclass solution in C++. Here are the results.

*myTrapezoid time: 0.389 seconds*  
*myTrapezoid L1 cache misses: 50%*  
*myTrapezoid L2 cache misses: 50%*

*myTrapezoidSIMD time: 0.188 seconds*  
*myTrapezoidSIMD L1 cache misses: 50%*  
*myTrapezoidSIMD L2 cache misses: 50%*

*Total Disk Reads: 125 million*

These results display how much faster SIMD instructions are compared to regular instructions. Despite both implementations missing the cache equal amounts of time, the SIMD implementation is still 52% faster than the regular version. I used [Godbolt](#) to ensure that myTrapezoid wasn't being compiled into SIMD instructions, which would make me reasonably believe that the vectorization was the cause of the SIMD version going faster. Sure enough, it shows that no vector operations are being done on the multiplication or addition aspects of the myTrapezoid implementation. This is what we would expect as it is difficult for the compiler to use SIMD because it has to maintain the addition order with doubles. It is also worth noting that both implementations had no if statements, so the branch misses for both implementations were very low and likely only consisted of the one-time for loop break. Hence the speed increase is likely purely from the use of vector operations. An important note is that the SIMD implementation does not account for when the array is not divisible by four. If this feature were implemented it would be slightly slower, which might make it exactly 50% faster than the non-vector instruction version.

I wanted to know why the cache miss rate was so high, and why it was exactly 50%. I tested the functions with a smaller array size, which resulted in both having 50% L1 misses and 0% L2 misses. This tells me that the cache misses are partially due to the size of the array. This makes sense as the bigger the array, the less stuff fits into the cache, hence more misses. Another contributing factor is the use of doubles in the array. Every double is 8 bytes wide, meaning only a few fit into each cache line, causing more misses when compared to other data types. Why does this come out to exactly 50% cache misses? I'm not sure. Doing the math based on my L1 cache size does not indicate that it would miss exactly half the time. So either it is a coincidence or something else that I am unaware of is happening.

### **Simpsons Method Results:**

My first implementation of this method used an if/else block in the for loop to check whether to multiply the calculated value by two or four. The second implementation, which I adapted from [Rosetta Code](#), used two four loops with alternating indices, had two sums, and multiplied both sums by either four or two outside of the loop. Here are the results.

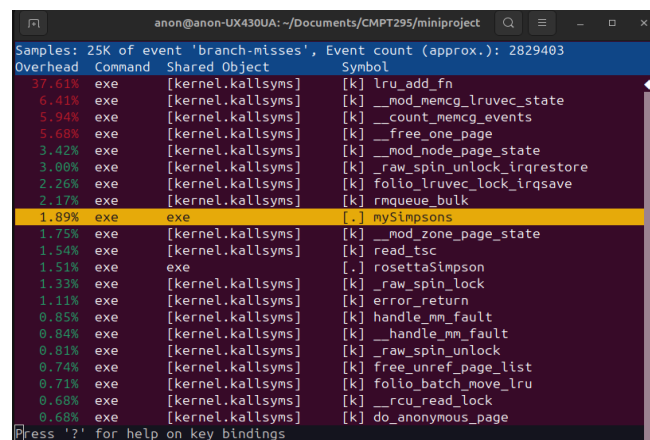
*mySimpsons time: 0.392 seconds*  
*mySimpsons L1 cache misses: 25%*  
*mySimpsons L2 cache misses: 25%*

*rosettaSimpsons time: 0.459 seconds*  
*rosettaSimpsons L1 cache misses: 50%*  
*rosettaSimpsons L2 cache misses: 50%*

*Total Disk Reads: 500 million*

These results display that the implementation with a potential conditional branch is faster than the version using two loops. The most clear reason for this is because mySimpsons is missing caches half as much as rosettaSimpsons. Comparing the assembly for the [mySimpsons](#) and [rosettaSimpsons](#) using Godbolt reveals why. The mySimpsons version is compiled with jumps, while the rosettaSimpsons version is compiled with two loops. It literally does the loop twice, hence twice as many cache misses. It is quite surprising that the compiler doesn't optimize out one of the loops and do some sort of trick. Another interesting note, the compiler does not use conditional moves with the mySimpsons implementation. It might be possible to squeeze some more performance out of this implementation by changing the conditional jumps to conditional moves.

Further, we can hypothesize that the branch predictor must do pretty well considering the total time of the mySimpsons function is very similar to myTrapezoid. Using Perf to analyze the branch mispredictions confirms these suspicions. mySimpsons accounts for 45% of the total branches, but only has a 1.5% misprediction rate. Logically, this makes sense. The branch is predictable as it alternates between the if and else blocks every time around.



Overhead	Command	Shared Object	Symbol
37.61%	exe	[kernel.kallsyms]	[k] lru_add_fn
6.41%	exe	[kernel.kallsyms]	[k] __mod_memcg_lruvec_state
5.94%	exe	[kernel.kallsyms]	[k] __count_memcg_events
5.68%	exe	[kernel.kallsyms]	[k] __free_one_page
3.42%	exe	[kernel.kallsyms]	[k] __mod_node_page_state
3.00%	exe	[kernel.kallsyms]	[k] __raw_spin_unlock_irqrestore
2.26%	exe	[kernel.kallsyms]	[k] folio_lruvec_lock_irqsave
2.17%	exe	[kernel.kallsyms]	[k] rmqueue_bulk
1.89%	exe	exe	[.] mySimpsons
1.75%	exe	[kernel.kallsyms]	[k] __mod_zone_page_state
1.54%	exe	[kernel.kallsyms]	[k] read_tsc
1.51%	exe	exe	[.] rosettaSimpson
1.33%	exe	[kernel.kallsyms]	[k] __raw_spin_lock
1.11%	exe	[kernel.kallsyms]	[k] error_return
0.85%	exe	[kernel.kallsyms]	[k] handle_mm_fault
0.84%	exe	[kernel.kallsyms]	[k] __handle_mm_fault
0.81%	exe	[kernel.kallsyms]	[k] __raw_spin_unlock
0.74%	exe	[kernel.kallsyms]	[k] free_unref_page_list
0.71%	exe	[kernel.kallsyms]	[k] folio_batch_move_lru
0.68%	exe	[kernel.kallsyms]	[k] __rcu_read_lock
0.68%	exe	[kernel.kallsyms]	[k] do_anonymous_page

## Conclusion, Learnings, and Potential Optimizations:

Through analyzing various implementations of the Trapezoid and Simpson's integration methods, we have found that the Trapezoid method implemented with SIMD operations is very fast relative to the other implementations. I did not get to try Simpsons method with SIMD operations. I hypothesize that a carefully written assembly implementation using conditional moves instead of conditional jumps, as well as SIMD operations, could be faster than my SIMD Trapezoid implementation. If you want to improve the speed of all implementations, use floats instead of doubles and this should reduce the amount of cache misses by a significant amount.

A key takeaway from this analysis is that the compiler does not compile things 100% efficiently all the time. If you want SIMD operations you have to organize your code around that. If you want minimal cache misses you have to organize your code around that. Further, if you want to know that your code is as fast as possible, you are going to have to analyze it and find out. Something else that was very apparent is that doubles inflate your cache misses by a lot. Use them if you need precision or if you don't care about speed.