BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Practical Aspects of Software Design 2023/2024

Profiling report

Tomáš Lajda (xlajdat00)

Brno, 20.4.2024

Assingment

Using functions from your mathematical library, create a program (as a standalone executable file) to calculate the sample standard deviation from a sequence of numbers read from standard input (e.g., using scanf in C) until the end of the file, and it must be able to read at least 1000 numbers (we'll verify this). Only numbers separated by whitespace (space, newline, or tab) will be provided as input, and their count is not predetermined. The formula for sample standard deviation to be used is:

$$s = \sqrt{\frac{1}{N-1}\left(\sum_{i=1}^{N} x_i^2 - N\overline{x}^2\right)}$$

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Profile this program with inputs of size 10, 10^3, and 10^6 numerical values. Submit a report containing the profiler output and a brief summary - highlighting where the program spends the most time and indicating what areas to focus on for code optimization.

Approach

The tools I chose to profile are cProfile and snakeviz to visualize data collected by cProfile. I began by profiling my code with cProfile to collect detailed performance data. I executed the program with input sizes of 10, 10^3, and 10^6 numerical values to observe how its performance scales with different input sizes. After each execution, I analyzed the profiler output to identify the functions consuming the most runtime. Using snakeviz, I visualized the profiling data to gain insights into the program's performance

Results

The profiling results indicate that the program spends the majority of its time within the function responsible for calculating (mainly pow and add) the sample standard deviation. This is consistent across all input sizes tested (10, 10^3, and 10^6 numerical values). The results are best viewed from the largest input size. Based on the profiling results, it's evident that our math library is performing well. If we want to make the program faster, we should try to cut down on how often we call functions. One way to do this is by finding a simpler way to calculate the standard deviation.

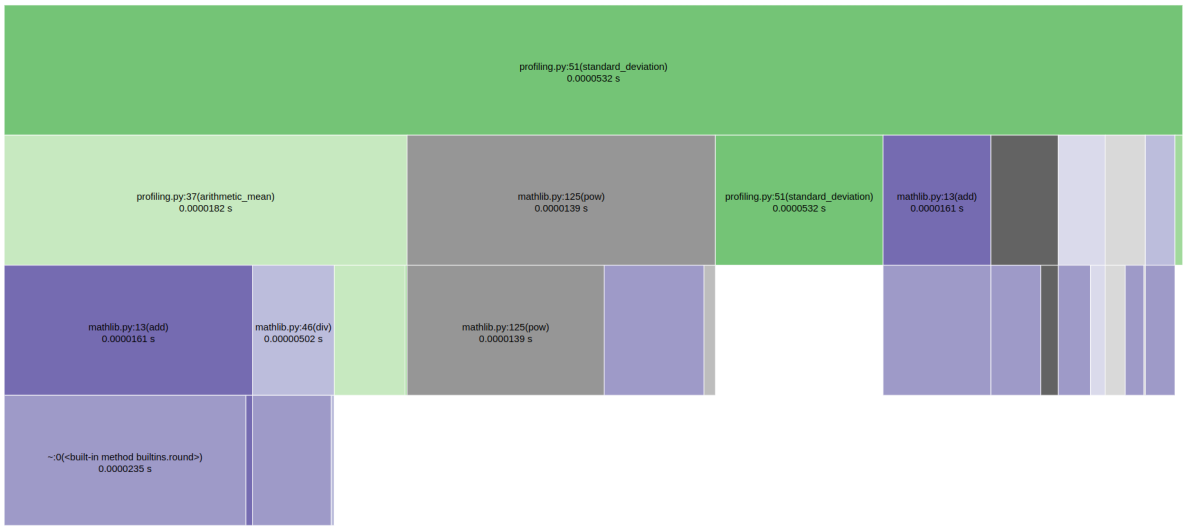

Figure 1: Visualisation of standard deviation with 10 input values by icicle

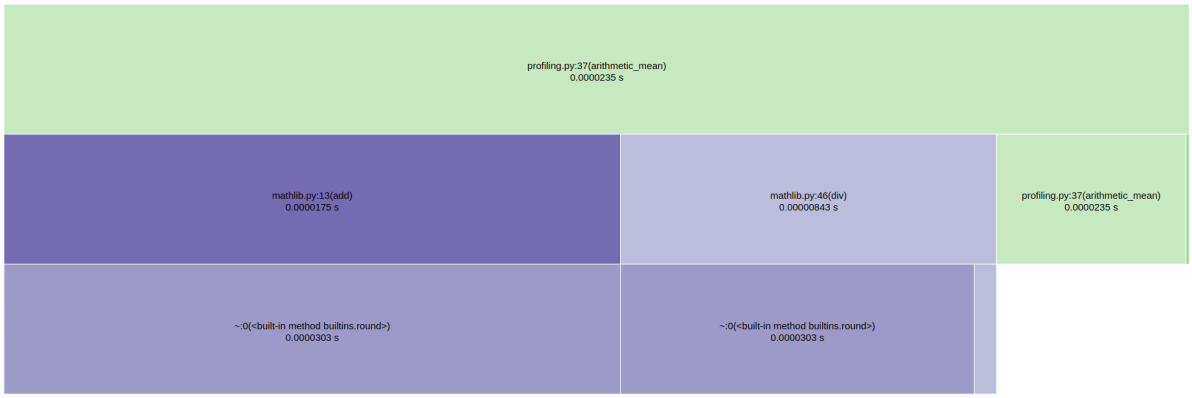

Figure 2: Visualisation of arithmetic mean with 10 input values by icicle

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	5.975e-05	5.975e-05	6.017e-05	6.017e-05	<frozen importlib._bootstrap>:100(acquire)
38	3.025e-05	7.961e-07	3.025e-05	7.961e-07	~:0(<built-in method builtins.round>)
1	1.448e-05	1.448e-05	1.448e-05	1.448e-05	~:0(<built-in method builtins.print>)
1	1.333e-05	1.333e-05	1.333e-05	1.333e-05	~:0(<built-in method marshal.loads>)
1	1.134e-05	1.134e-05	0.0003404	0.0003404	profiling.py:1(<module>)
1	1.004e-05	1.004e-05	1.004e-05	1.004e-05	~:0(<built-in method io.open_code>)
11	9.842e-06	8.947e-07	1.613e-05	1.467e-06	mathlib.py:125(pow)
1	8.2e-06	8.2e-06	6.232e-05	6.232e-05	profiling.py:51(standard_deviation)
1	7.947e-06	7.947e-06	3.198e-05	3.198e-05	<frozen importlib._bootstrap_external>:1536(find_spec)
1	7.656e-06	7.656e-06	1.959e-05	1.959e-05	profiling.py:13(seperate_numbers_from_std)
3	7.115e-06	2.372e-06	7.115e-06	2.372e-06	~:0(<built-in method posix.stat>)
1	6.927e-06	6.927e-06	6.333e-05	6.333e-05	<frozen importlib._bootstrap_external>:950(get_code)
1	5.875e-06	5.875e-06	5.281e-05	5.281e-05	<frozen importlib._bootstrap>:921(_find_spec)
2	5.744e-06	2.872e-06	1.67e-05	8.349e-06	<frozen importlib._bootstrap_external>:380(cache_from_source)
1	5.268e-06	5.268e-06	1.053e-05	1.053e-05	~:0(<method 'read' of '_io.TextIOWrapper' objects>)
20	5.186e-06	2.593e-07	1.754e-05	8.772e-07	mathlib.py:13(add)

Figure 3: Times and number of calls for each function from mathlib on 10 input values

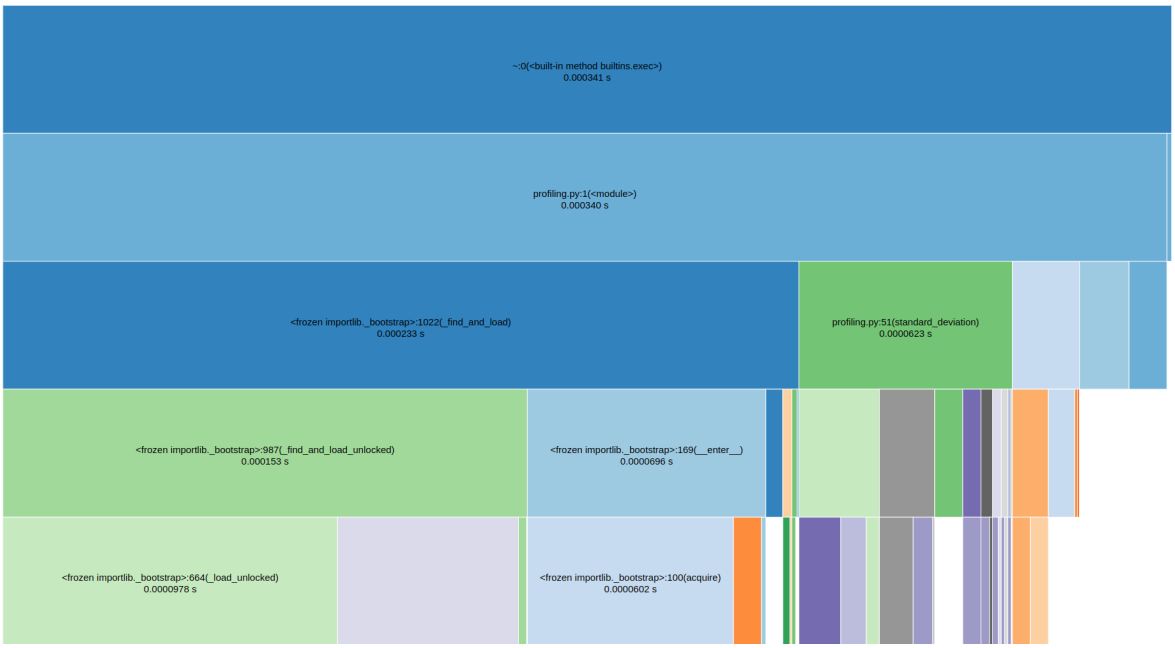

Figure 4: Visualisation of whole program with 10 input values by icicle

Figure 5: Visualisation of standard deviation with 1000 input values by icicle

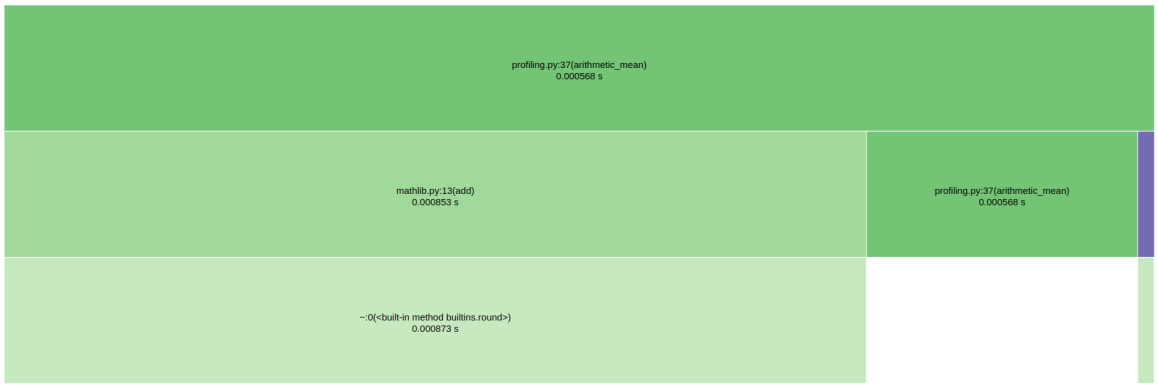

Figure 6: Visualisation of arithmetic mean with 1000 input values by icicle

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
3008	0.0008729	2.902e-07	0.0008729	2.902e-07	~:0(<built-in method builtins.round>)
1001	0.0003597	3.594e-07	0.0006832	6.825e-07	mathlib.py:125(pow)
1	0.0002863	0.0002863	0.001974	0.001974	profiling.py:51(standard_deviation)
2000	0.0002742	1.371e-07	0.0008527	4.264e-07	mathlib.py:13(add)
1	0.0001743	0.0001743	0.0002565	0.0002565	profiling.py:13(seperate_numbers_from_std)
1	0.000134	0.000134	0.0005682	0.0005682	profiling.py:37(arithmetic_mean)

Figure 7: Times and number of calls for each function from mathlib on 1000 input values

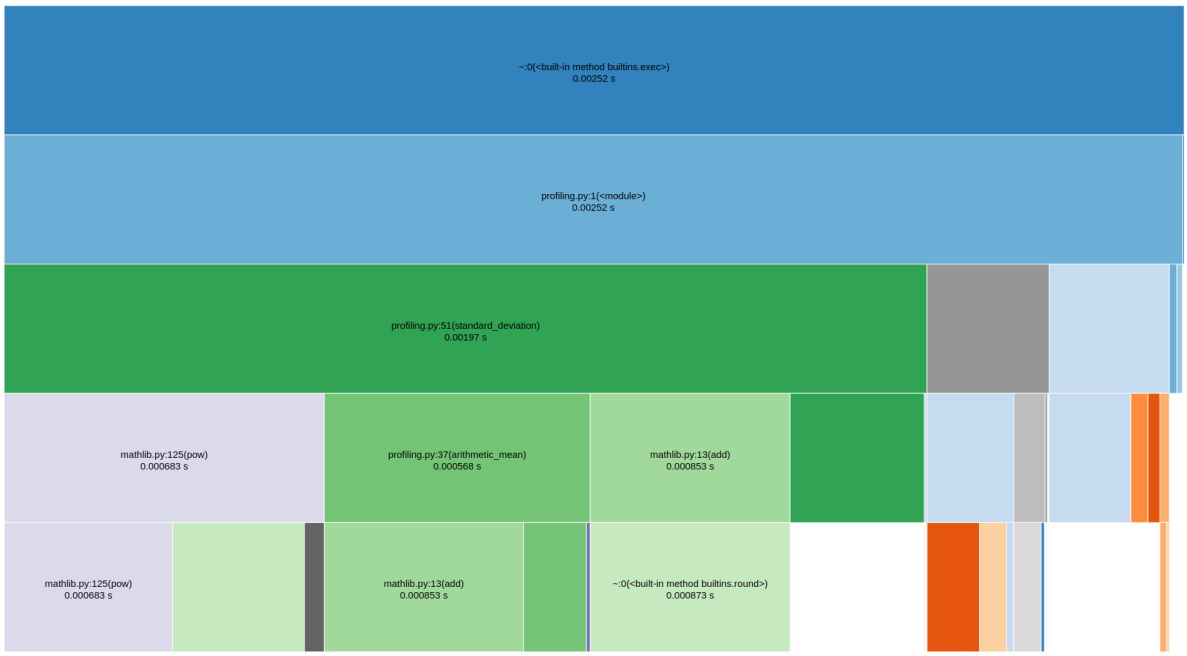

Figure 8: Visualisation of whole program with 1000 input values by icicle

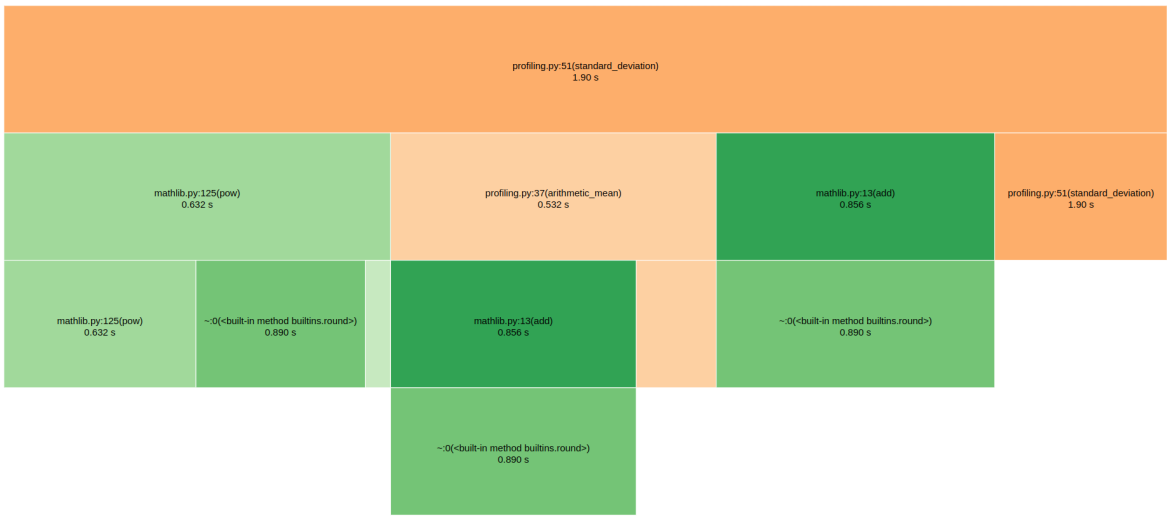

Figure 9: Visualisation of standard deviation with 1000000 input values by icicle

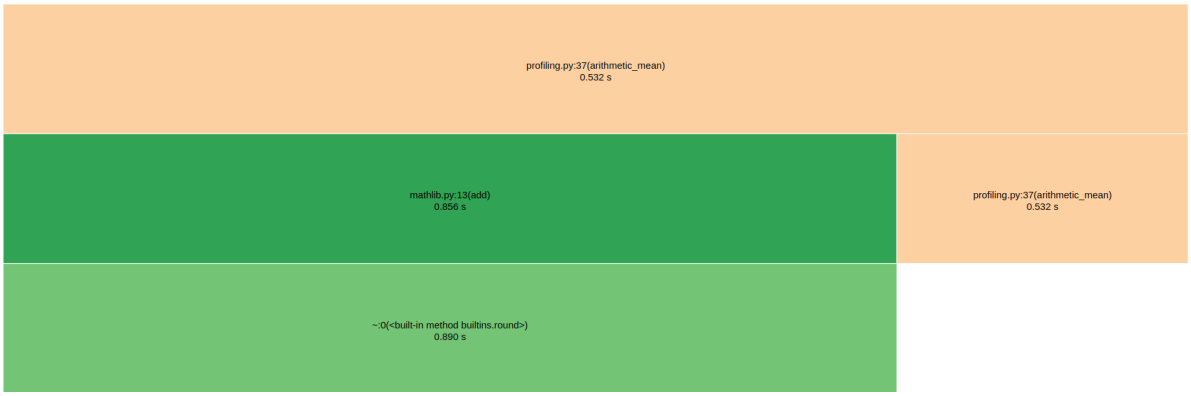

Figure 10: Visualisation of arithmetic mean with 1000000 input values by icicle

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
3000008	0.8896	2.965e-07	0.8896	2.965e-07	~:0(<built-in method builtins.round>)
1000001	0.3135	3.135e-07	0.6316	6.316e-07	mathlib.py:125(pow)
1	0.2817	0.2817	1.9	1.9	profiling.py:51(standard_deviation)
2000000	0.2434	1.217e-07	0.8561	4.281e-07	mathlib.py:13(add)
1	0.1717	0.1717	0.239	0.239	profiling.py:13(seperate_numbers_from_std)
1	0.1308	0.1308	0.5321	0.5321	profiling.py:37(arithmetic_mean)

Figure 11: Times and number of calls for each function from mathlib on 1000000 input values

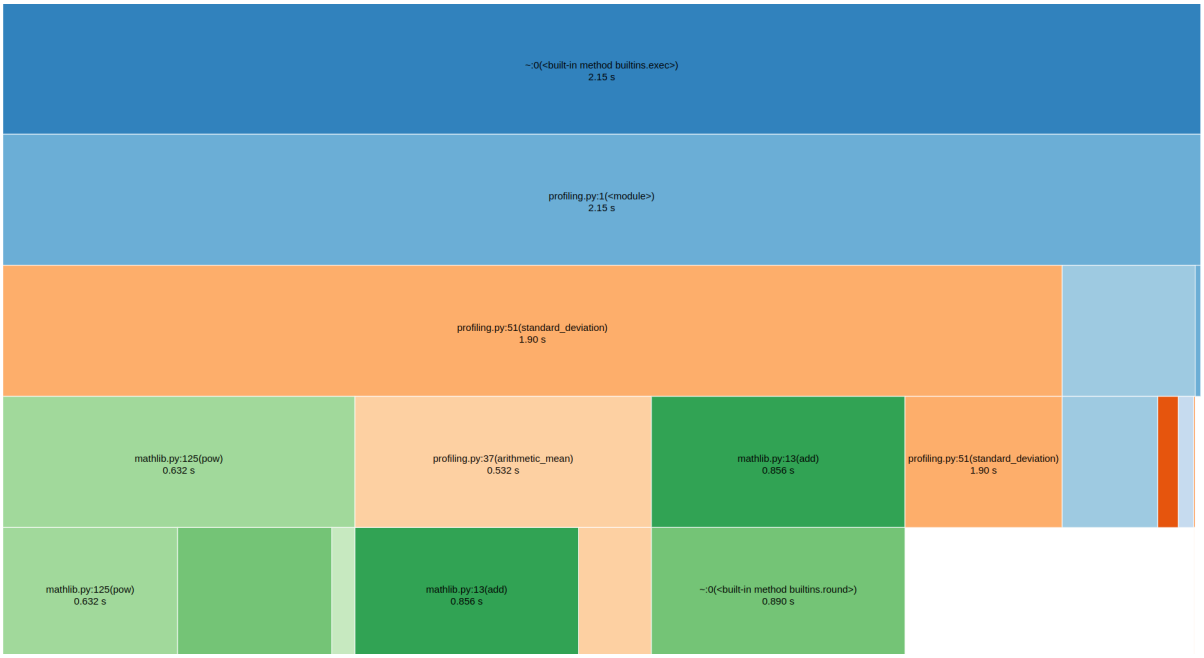

Figure 12: Visualisation of whole program with 1000000 input values by icicle