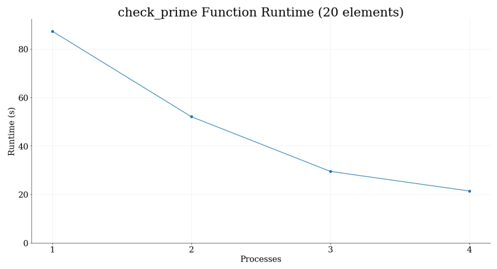
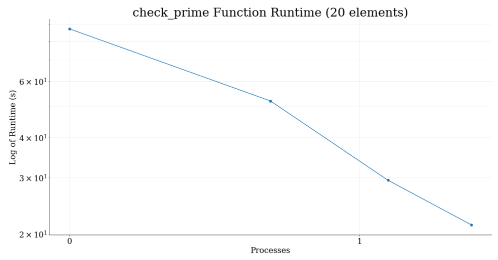
N.B. The CPU used for the following was an Intel Core i7-7700HQ running at 3.4 GHz, allowing processes to be executed in parallel on up to 4 cores.

Prime Numbers

The set of numbers to be tested for primality by the process pool was comprised of 13 numbers known to be prime, along with an additional 7 integers that were generated using the `random.randint()` function. The runtime as a function of the number of processes is shown in Figure (1a). There is a substantial, yet somewhat diminishing, impact on the



(a) `check_prime()` runtime as a function of number of processes.



(b) Logarithm of `check_prime()` runtime as a function of number of processes.

Figure 1: `check_prime()` runtime with various processors.

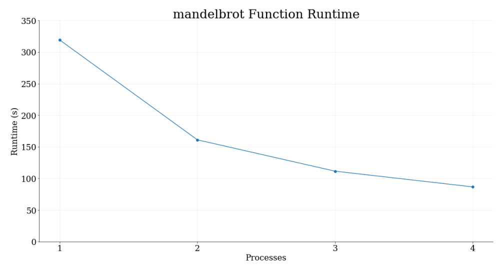
function runtime as the number of processes increases. Quantitatively, this information is represented in Table (1). This performance increase is to be expected, since each process can be executed in parallel on a separate processor within the CPU. In essence, one would expect that by making use of multiprocessing with all 4 processors, the function runtime should decrease by a factor of 4. As can be seen, when one process is used the runtime is 87.34 seconds, compared to 21.38 seconds with 4 processes. This corresponds to a factor of 4.08, almost exactly as expected. As the graph shown in Figure (1a) did not appear to be *exactly* linear, a logarithmic plot was also created, shown in Figure (1b).

Processes	**Runtime (seconds)**	**Performance Increase**
1	87.34	N/A
2	52.05	67.80%
3	29.47	196.40%
4	21.38	308.42%

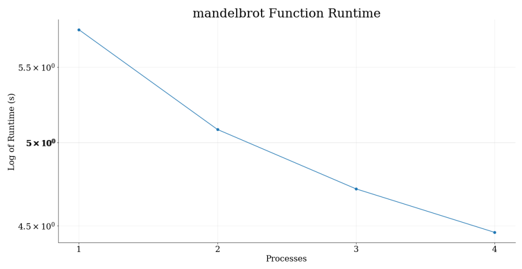
Table 1: Impact of multiprocessing with the `check_prime()` function.

Mandelbrot Set Generation

The alternative task chosen was the computation and generation of the Mandelbrot set. The code within the *Mandlebrot.ipynb* notebook found on Moodle was implemented. The generation of the Mandelbrot set is a computationally expensive task, depending on the dimensions and number of iterations used. The parameters selected for this test were `h = 1080, w = 1920`, and `max_iterations = 1000`. Once again, these results are displayed in



(a) Mandelbrot set generation as a function of number of processors allocated.



(b) Logarithm of Mandelbrot set generation runtime.

Figure 2: Impact of multiprocessing on the runtime of Mandelbrot set generation.

Table (2). This was an interesting case to observe the results of, as the Mandelbrot set is known as an *embarrassingly parallel problem* since each point can be computed independently. Therefore, one would expect there to be an exactly linear increase in performance when multiple processes are created. Clearly however, this was not the case with the tests ran, and perhaps even more interestingly the performance increases are less linear than in the `check_prime()` function. Although the allocation of one extra process resulted in a 98.01% increase in performance, the allocation of 3 additional processes "only" resulted in a 267.98% performance increase – a far cry from the expected 300%.

Processors	**Runtime (seconds)**	**Performance Increase**
1	318.91	N/A
2	161.06	98.01%
3	111.52	185.98%
4	86.66	267.98%

Table 2: Impact of multiprocessing on the generation of the Mandelbrot set.