Measuring Time in Python

Timing With `time.perf_counter`

We will use the `time` module to perform our benchmarks. More specialized modules exist, but for this course we will keep benchmarking simple.

The `time` module gives access to the computers system clocks, as well as functions for converting time formats.

The documentation for `time` can be found here.

The function that is of particular importance to us is `perf_counter()` :

[2]:
```
from time import perf_counter
```

It uses the most precise system wide clock available to it to return a time in seconds. The starting point of the timer is arbitrary and system dependent, so only time differences are of use to us. When timing a block of code make sure to take the time directly before and after the block.

Worked Example

Let's say we want to compare the time it takes to perform the sum

$$\sum_{n=1}^{1000} \frac{(-1)^n}{n}$$

using a for loop to using NumPy array functions.

[5]:
```
#Timing the use of a loop
start_time = perf_counter()

s = 0
for n in range(1, 1001):
    s += (-1)**n/n

loop_time = perf_counter() - start_time

print('Using a loop:')
print('The value of the sum is:', s)
print('The time taken to compute the sum is:', loop_time)
```

```
Using a loop:
The value of the sum is: -0.6926474305598223
The time taken to compute the sum is: 0.0018592540000099689
```

[4]:
```python
#Timing the use of numpy functions
import numpy as np

start_time = perf_counter()

n_arr = np.arange(1, 1001)
s = np.sum( (-1)**n_arr/n_arr )

np_time = perf_counter() - start_time

print('Using NumPy:')
print('The value of the sum is:', s)
print('The time taken to compute the sum is:', np_time)
```

```
Using NumPy:
The value of the sum is: -0.6926474305598204
The time taken to compute the sum is: 0.001659153999753471
```

Note that the times for both the looping method and NumPy method are very similar. Every time the code is run the values also fluctuate wildly. This makes it difficult to compare the performance of these solutions.

As a solution to this problem, we can take many runs of the code block in question and quote the total time taken to execute all of them. This will limit the effects of the fluctuation on our measurement and it will also make it less likely for us to run into floating point errors (the times of individual code runs can be very small).

Something to keep in mind when running a benchmark is to limit the number of background processes you have running on your computer, in particular those who's resource requirements fluctuate.

Don't time what you don't intend to measure, only time the code you are interested in benchmarking. If you use a loop to repeat measurements (i.e. it isn't an essential part of the code you're testing) don't include the overhead from the loop in your timing. If you are using print function calls for debugging purposes, you should also exclude those from the timing (in the example above the print calls were performed **after** the time was taken).