Assignment 2: May the Force...

Group: I Miloncini

Tasks:

- **E) Comparison and benchmark**:
  In `data` directory there's a file, `dt_tot.txt`, containing the parameters and the results of the tests.

  In the `plots` folder you can find 3 .pdf files:

  - `plot_dir_model.pdf` contains the benchmark and comparison between the acceleration estimation functions built with the *direct* method. Data gathered with `pyfalcon` are displayed too. These functions were run for a maximum of 5000 particles, due to the long computational time and expensive computational power. You can see that among the three `acc_dir_*`, `acc_dir_diego` is slightly better than `acc_dir_vepe`. Thus we decided to implement it in the main code as our `acceleration_direct`.
  - `plot_vect_model.pdf` contains the benchmark and comparison between the acceleration estimation functions built in a *vectorized* fashion. Data gathered with `pyfalcon` are displayed too. This time we run the simulation till a maximum of 50.000 particles, thanks to the computational power of the **Demoblack** server. You can see that among them, the one with the better performance is `acc_onearray_vepe` (in this case slightly better than `acc_vect_diego`). Thus we decided to implement it in the main code as our `acceleration_direct_vectorized`. The function uses the broadcasting operations of `numpy.array`.
    In this function we also implemented the computation of the jerk: the line `jerk_vepe` refers to this case (further considerations below).
  - `plot_tot.pdf` contains the comparison between all the models we implemented. It's remarkable the increasing in performance (almost 2 orders) relative to the vectorized models. [`pyfalcon`, using C and memory allocation is playing in another league (it's like Pinerolo vs Roma)]

  Speaking about maximum number of particles, `pyfalcon`, based on the tests run by its developers, for $N \geq 10^5$ the scaling of the CPU time required for the mutual forces becomes essentially linear allowing for a substantial improvement in simulations employing large number of bodies. The only disadvantage is the increased requirement of memory.
  Regarding the vectorized functions, a maximum value of N, taking into account a trade-off between computational time and hardware resources (especially RAM, since `numpy.array` requires a lot of it, ~150 GB for $N \geq 5 \cdot 10^3$), could be around $5 \cdot 10^4$ (if you are brave enough, and if you have a lot of time and a lot of RAM, you can try also to reach $N = 10^6$).
  For non-vectorized functions, the maximum number is even lower. Since the computational time is very high, the maximum number is around $5 \cdot 10^3$ (and necessarily $< 10^4$).
  We tried to fit the data we gather (i.e., `dt_tot.txt`) with some functions. The theoretical order of the `acceleration_direct` function should be $O(N^2)$, and we see it from our fitting plot. Regarding the `acceleration_vectorized` function, we expect (looking at the algorithm) that the theoretical order is $O(N^2)$. But understanding how numpy deals with data and operations is difficult. In fact we see that the vectorized funtion is much faster than the direct one, even thoug both functions are well fitted by a 2nd degree polynominal. Therefore, our conclusion is that at worse the vectorized function goes as $O(N^2)$, but from the results we got we expect that the order is much lower.
- **F) (Optional) Jerk estimate**:
  As mentioned above, we implemented the computation of the jerk directly in `acceleration_direct_vectorised`. Since, as you can see from the benchmark, the computation of the jerk is quite computational demanding we implemented a flag `return_jerk` set, as default, `False`. So the performance, for the calculation of the acceleration only, does not decrease.
  Setting the flag to `True`, this is the best that the function can do, since we are using the `numpy.array` broadcasting.
  Regarding the jerk, we implemented new/updated test functions in `test_acceleration_estimate.py`.
- **Challenge (not mandatory): Need for Speed!**
  `acc_onearray_vepe` and `acc_opt_gia` are already the optimized version of the corresponding functions `acc_vect_vepe` and `acc_vect_gia`. The three models of the vectorized function are built in slightly different way, and since the best one, among them, is `acc_onearray_vepe`, probably it's the best result achievable using `numpy.array` and broadcasting.
  Thus, we are trying to improve the `acceleration_direct` function using `Cython`, so as to exploit memory allocation and for loops in a C fashion.