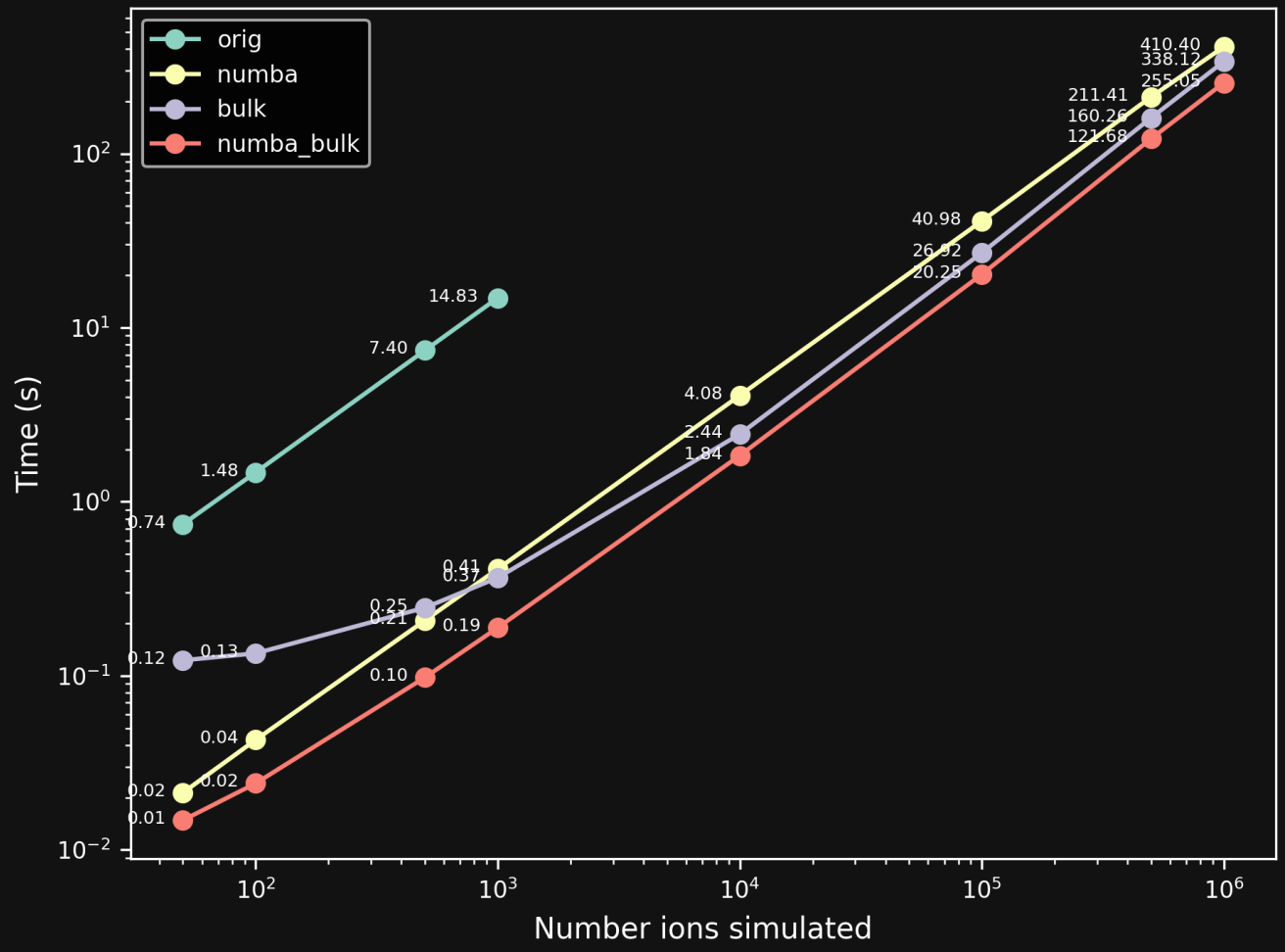
# Optimizing PyTrim (Pt. 2)

Using a combination of NumPy and Numba

# Content

- Benchmarks
- Implementation
  - Improving performance using pure NumPy
  - Adding Numba
  - Parallelizing
- Conclusion


Overview of simulation times
orig
numba
bulk
numba_bulk
Time (s)
Number ions simulated
$10^{2}$
$10^{1}$
$10^{0}$
$10^{-1}$
$10^{-2}$
$10^{2}$
$10^{3}$
$10^{4}$
$10^{5}$
$10^{6}$
14.83
7.40
1.48
0.74
0.13
0.12
0.04
0.02
0.02
0.01
0.41
0.37
0.25
0.21
0.19
0.10
4.08
2.44
1.84
40.98
26.92
20.25
211.41
160.26
121.68
410.40
338.12
255.05

# Improving performance using pure NumPy

- Current version of PyTrim heavily relies on python loops
  - NumPy can’t optimize them -> no SIMD optimizations possible
  - Numba can, but has its own limitations (e.g. reduced [Num]Py feature support)

## Proposal

- Vectorize everything and rely on more efficient NumPy methods
  - Adding Numba later should be possible and may be beneficial

# Improving performance using pure NumPy

- Current state: a maximum of 3 "parallel" arithmetical operations
  - Full simulation for a single ion per iteration
- *Proposal*: as much as possible independent "parallel" arithmetical operations
  - Single simulation step for *all* particles per iteration
  - Parallelization may be possible / beneficial on larger number of particles

```python
import numpy as np

a = np.random.rand(5, 3)
b = np.random.rand(5, 3)
# 15 simultaneous operations
a += b
# Time with 100k rows: 0.003s
print(a)
```

# DEMO

Improving performance using pure NumPy

# Adding Numba

The following methods / signatures did not compile with Numba:

```
np.linalg.norm(axis)
np.ndarray[list, list]
np.ndarray[:, list[list]]
np.put_along_axis()
np.argmin(keepdims)
```

(full list can be found here)

Those parts had to be re-written, sometimes in ugly ways

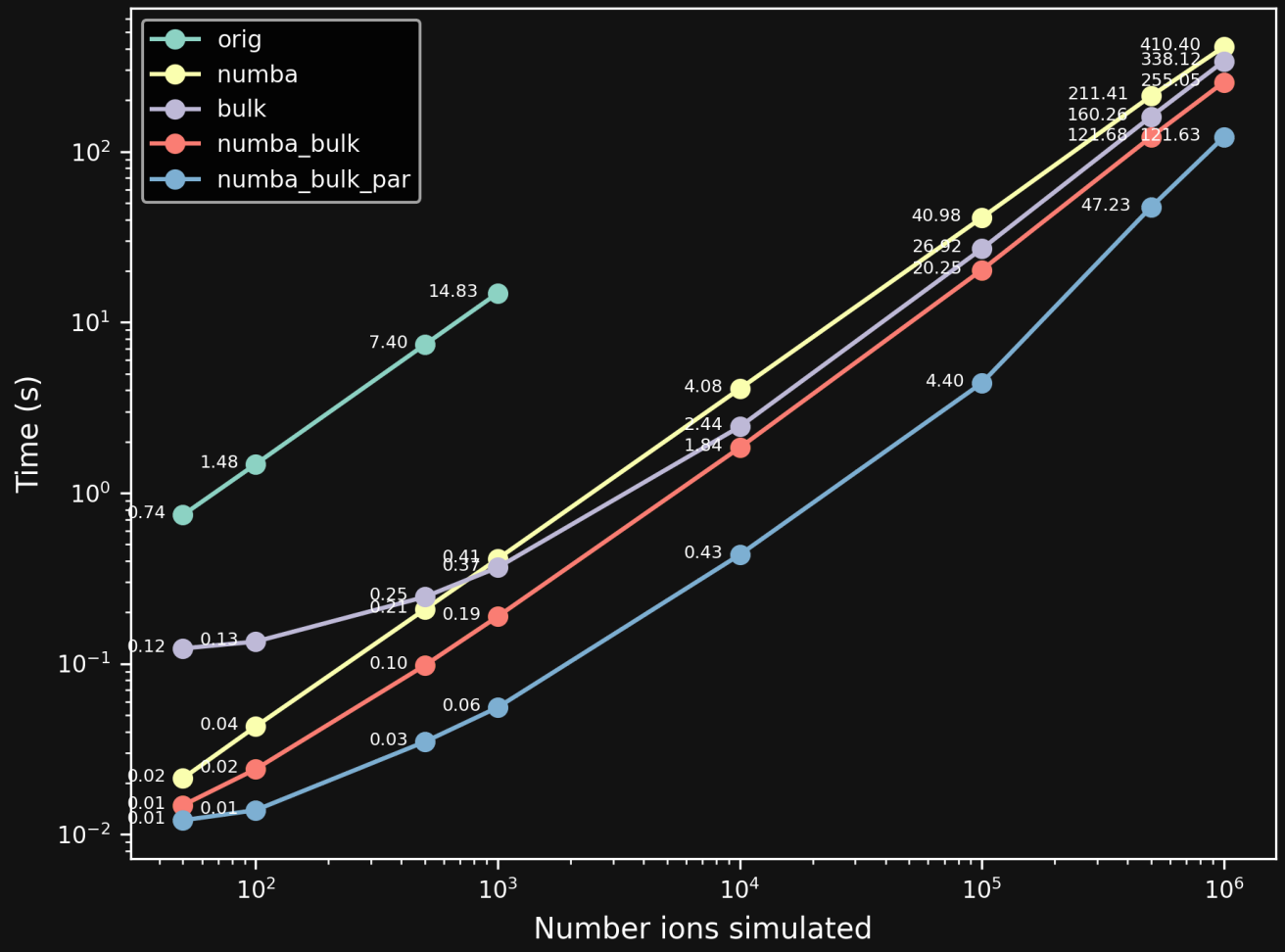
PyQT integration: **Adding Numba prevents the use of Queues to send simulation updates to UI thread**

# Adding Numba - Parallelization

```python
@jit(fastmath=True, parallel=True, nogil=True, cache=True)
def simulate(nion: int, nthreads: int = 16):
    # Initial conditions of the projectile
    # NOTE In reality a check is performed to split data in equal parts
    threads = nthreads
    col_cnt = nion / nthreads
    e = np.full((threads, col_cnt), 50000.0)  # energy (eV)
    pos = np.zeros((threads, col_cnt, 3))  # position (A)
    dir = np.zeros((threads, col_cnt, 3))
    dir[:, :, 2] = 1.0  # direction (unit vector)
    is_inside = np.full((threads, col_cnt), True)

    for i in prange(threads):
        trajectory.trajectories(pos[i], dir[i], e[i], is_inside[i])

    pos = pos.reshape((nion, 3))
    dir = dir.reshape((nion, 3))
    e = e.flatten()
    is_inside = is_inside.flatten()
    count_inside = np.count_nonzero(is_inside)
    mean_z = pos[is_inside, 2]
    std_z = np.std(mean_z)
    mean_z = np.mean(mean_z)
```


Overview of simulation times
orig
numba
bulk
numba_bulk
numba_bulk_par
Time (s)
Number ions simulated
$10^{2}$
$10^{1}$
$10^{0}$
$10^{-1}$
$10^{-2}$
$10^{2}$
$10^{3}$
$10^{4}$
$10^{5}$
$10^{6}$
14.83
7.40
1.48
0.74
410.40
338.12
255.05
211.41
160.26
121.68
121.63
40.98
26.92
20.25
47.23
4.08
2.44
1.84
4.40
0.43
0.41
0.37
0.25
0.21
0.19
0.13
0.12
0.10
0.06
0.04
0.03
0.02
0.02
0.01
0.01
0.01
0.01

# Conclusion

- Provided NumPy implementation brings a significant performance boost
  - Additional 2x possible by reducing precesion from `` `float64` `` -> `` `float32` ``
- Adding Numba brings a slight improvement, but also significant limitations and less readable code
  - Limited [Num]Py feature set
  - Increased codebase size / complexity
- *UI integration options*
  - Vanilla NumPy implementation (in background) notifying about finished simulations (processing in main / other thread)
  - Numba with data splitting, but with Python threads

# Questions?