Numba Introduction

Themen

- Was Numba kann
- Einführung in JIT-Kompilierung
- Tests
- Verbesserungspotenzial
- Persönliche Meinung & Fazit

Was Numba kann

Kompilierung vom Python-Code

- Optimierung der Rechenoperationen durch CPU-spezifische Anweisungen
- Unterstützt NumPy und meist verwendeten Python-libraries "out of the box"
- Minimale Code-Änderungen notwendig

Weitere Features

- Parallelisierung der Rechenoperationen / Schleifen
- Ermöglicht Einbindung vom Python-Code in C/C++

```python
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit
def go_fast(a):
    trace = 0.0
    # Numba likes loops
    for i in range(a.shape[0]):
        # Numba likes NumPy functions
        trace += np.tanh(a[i, i])
    # Numba likes NumPy broadcasting
    return a + trace

print(go_fast(x))
```

JIT-Kompilierung

```
# JIT (Just-In-Time) Kompilierung wandelt gekennzeichnete Code-Abschnitte
# zum Zeitpunkt der ersten Ausführung in Maschinencode um, um diese
# bei eineutem Aufruf wiederverwenden zu können
```

> "Numba reads the Python bytecode for a decorated function and combines this with information about the types of the input arguments to the function. It analyzes and optimizes your code, and finally uses the LLVM compiler library to generate a machine code version of your function, tailored to your CPU capabilities. This compiled version is then used every time your function is called."

- Kommt nur in interpretierbaren Programmiersprachen (e.g. PHP) zum Einsatz
- Erste Ausführung dauert länger
- Kompilierte Teile können auf der Festplatte gespeichert und wiederverwendet werden

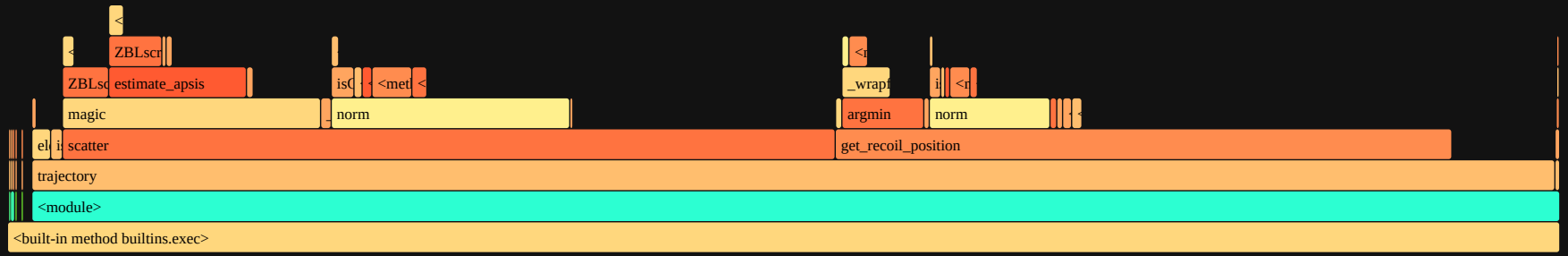
Tests - Ausgangsbedingungen

Unveränderter Code

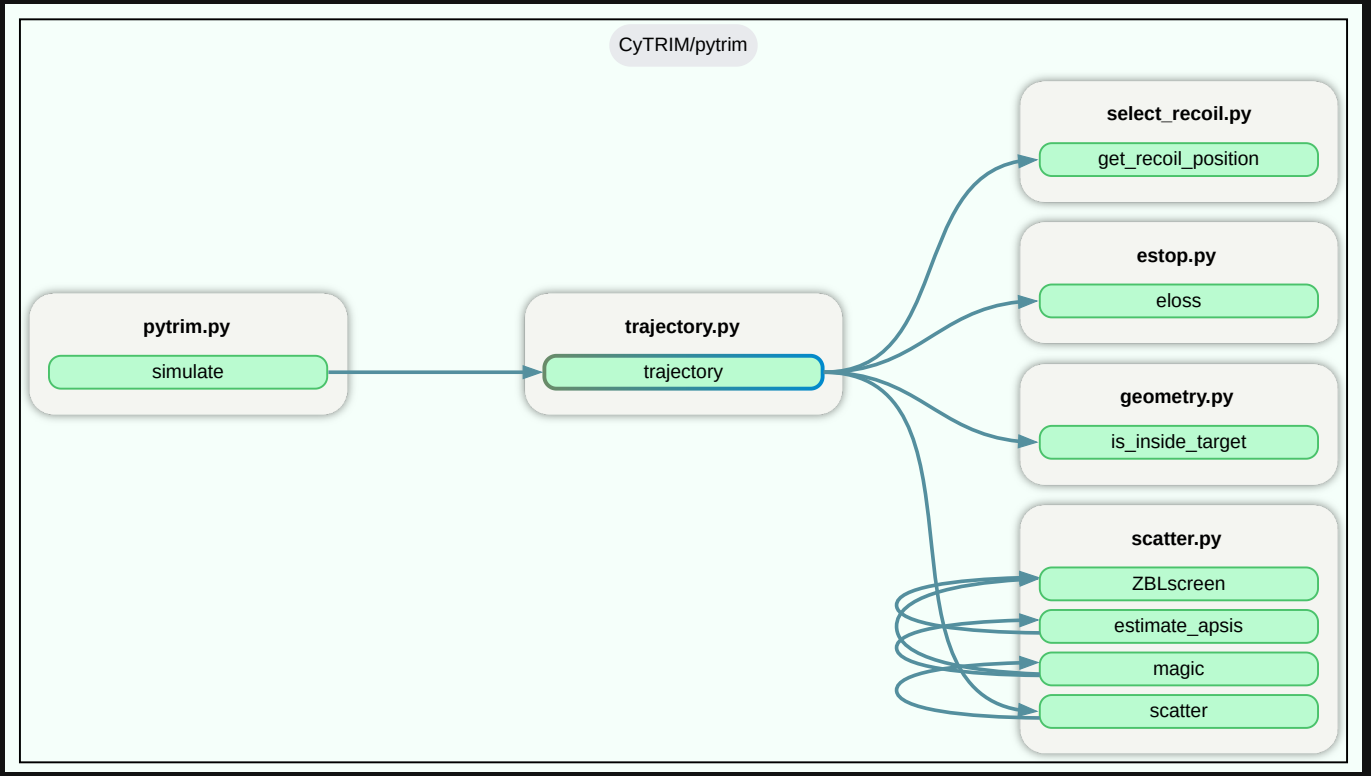
- 14.96s (im Module, Mittelwert, 3 Ausführungen)
- `trajectory.trajectory()` - 99.3% der Ausführungszeit
  - (46.3%) `get_recoil_position()`
  - (47.8%) `scatter()`

Technische Daten

- Intel Core i5-12500H (x86_64)
  - 12 cores, 16 threads
- 24GB DDR4 RAM
- numpy 2.0.2
- numba 0.60.0



Tests - Vorbereitung



Tests - Vorbereitung

```python
from numba import jit

@jit
def trajectory(pos_init, dir_init, e_init):
    """Simulate one trajectory.

    Parameters:
        pos_init (ndarray): initial position of the projectile (size 3)
        dir_init (ndarray): initial direction of the projectile (size 3)
        e_init (float): initial energy of the projectile (eV)

    Returns:
        ndarray: final position of the projectile (size 3)
        ndarray: final direction of the projectile (size 3)
        float: final energy of the projectile (eV)
        bool: True if projectile is stopped inside the target,
            False otherwise
    """
```

```python
@jit
def get_recoil_position(pos, dir)
```

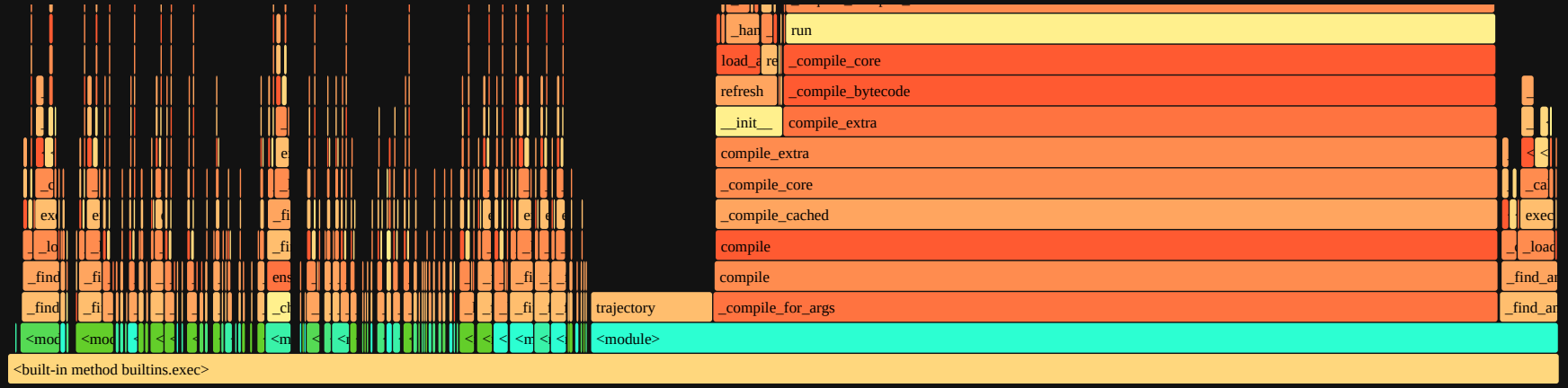
```python
@jit
def scatter(e, dir, p, dirp)
```

...

Tests - Zwischenergebnis

Optimierter Code

- 2.13s (-85.76%) (im Module, Mittelwert, 3 Ausführungen)
- `trajectory.trajectory()` - 11.6% der Ausführungszeit
- `_compile_for_args()` - 82.4%
  - Verbesserungsbedarf



Tests - Optimierung

```python
from numba import jit

@jit(cache=True, fastmath=True)
def trajectory(pos_init, dir_init, e_init):
    """Simulate one trajectory.

    Parameters:
        pos_init (ndarray): initial position of the projectile (size 3)
        dir_init (ndarray): initial direction of the projectile (size 3)
        e_init (float): initial energy of the projectile (eV)

    Returns:
        ndarray: final position of the projectile (size 3)
        ndarray: final direction of the projectile (size 3)
        float: final energy of the projectile (eV)
        bool: True if projectile is stopped inside the target,
            False otherwise
    """
```

```python
@jit(fastmath=True)
def get_recoil_position(pos, dir)
```

```python
@jit(fastmath=True)
def scatter(e, dir, p, dirp)
```

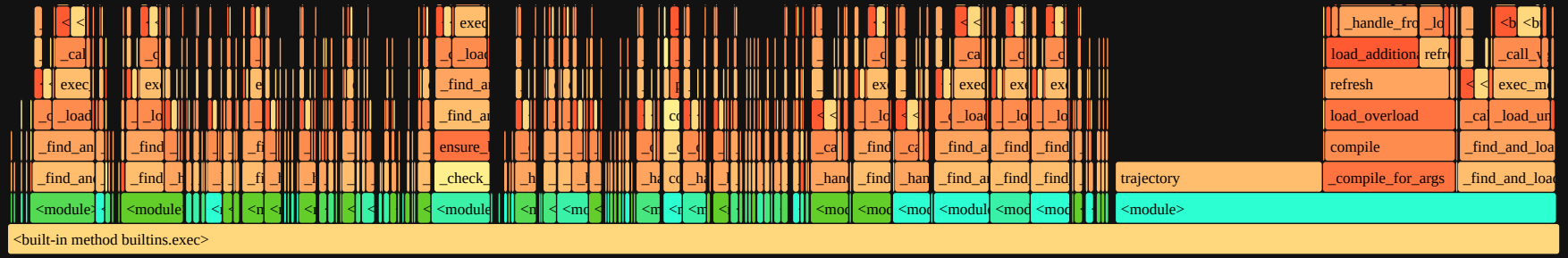
...

- `cache` - Kompilierter code wird auf der Festplatte gespeichert
- `fastmath` - Weitere Optimierung der Rechenoperationen auf Kosten der Präzesion

Tests - Endergebnis

Optimierter Code

- 0.58s (-72.77%) (im Module, Mittelwert, 3 Ausführungen)
- `trajectory.trajectory()` - 46.9% der Ausführungszeit
- `_compile_for_args()` - 29.2%



Verbesserungspotenzial

- Parallelisierung der Schleife mit `trajectory()` Aufruf hatte keinen Einfluss
- `fastmath` hatte nur einen geringen Einfluss
- Anpassung vom Code für Arbeit mit reinen NumPy-Funktionen mit 2D-Arrays kann die Performance mit Numba weiter verbessern

Persönliche Meinung & Fazit

Pros

- Leichte Einbindung
- Deutlich spürbare Optimierung
- Umfangreiche & verständliche Dokumentation

Remarks

- Beschränkte Exception-Handling
  - Array-Zugriff über unzulässige Indizen
- Read-only globals
- Caching
  - Globals werden als konstant angenommen
  - Code-Änderungen in anderen Dateien werden nicht erkannt

Fragen?