Politechnika Warszawska

WYDZIAŁ FIZYKI

Praca dyplomowa inżynierska

na kierunku Fizyka Techniczna
w specjalności Fizyka komputerowa

Zoptymalizowana symulacja Particle in Cell w języku Python

Dominik Stańczak
261604

promotor
dr Sławomir Jabłoński

WARSZAWA 2017

Streszczenie

Tytuł pracy: Zoptymalizowana symulacja Particle in Cell w języku Python

1 Streszczenie

Utworzono kod symulacyjny Particle-in-Cell mający modelować interakcję relatywistycznej plazmy wodorowej oraz impulsu laserowego. Kod zoptymalizowano w celu zademonstrowania możliwości użycia wysokopoziomowego języka programowania wywołującego niskopoziomowe procedury numeryczne do osiągnięcia wysokiej wydajności obliczeniowej. *Słowa kluczowe:*

python, plazma, particle in cell, symulacja, optymalizacja, elektrodynamika

(podpis opiekuna naukowego) (podpis dyplomanta)

Abstract

Title of the thesis: Optimised Particle in Cell simulation in Python

2 Abstract

A Python particle-in-cell plasma simulation code is developed to model the interaction between a hydrogen plasma target and a laser impulse. The code is then optimized to demonstrate the possibilities of using a high level programming language to call low level numerical procedures, thus achieving high computational efficiency. *Słowa kluczowe:*

<python, plasma, particle in cell, simulation, optimization, electrodynamics>

(podpis opiekuna naukowego) (podpis dyplomanta)

Oświadczenie o samodzielności wykonania pracy

Politechnika Warszawska
Wydział Fizyki

Ja niżej podpisany/a:

Dominik Stańczak, 261604

student/ka Wydziału Fizyki Politechniki Warszawskiej, świadomy/a odpowiedzialności prawnej przedłożoną do obrony pracę dyplomową inżynierską pt.:

Zoptymalizowana symulacja Particle in Cell w języku Python

wykonałem/am samodzielnie pod kierunkiem

dr Sławomira Jabłońskiego

Jednocześnie oświadczam, że:

- praca nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 o prawie autorskim i prawach pokrewnych, oraz dóbr osobistych chronionych prawem cywilnym,
- praca nie zawiera danych i informacji uzyskanych w sposób niezgodny z obowiązującymi przepisami,
- praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem dyplomu lub tytułu zawodowego w wyższej uczelni,
- promotor pracy jest jej współtwórcą w rozumieniu ustawy z dnia 4 lutego 1994 o prawie autorskim i prawach pokrewnych.

Oświadczam także, że treść pracy zapisanej na przekazanym nośniku elektronicznym jest zgodna z treścią zawartą w wydrukowanej wersji niniejszej pracy dyplomowej.

Warszawa, dnia 2017 (podpis dyplomanta)

Oświadczenie o udzieleniu Uczelni licencji do pracy

Politechnika Warszawska
Wydział Fizyki

Ja niżej podpisany/a:

Dominik Stańczak, 261604

student/ka Wydziału Fizyki Politechniki Warszawskiej, niniejszym oświadczam, że zachowując moje prawa autorskie udzielam Politechnice Warszawskiej nieograniczonej w czasie, nieodpłatnej licencji wyłącznej do korzystania z przedstawionej dokumentacji pracy dyplomowej pt.:

Zoptymalizowana symulacja Particle in Cell w języku Python

w zakresie jej publicznego udostępniania i rozpowszechniania w wersji drukowanej i elektronicznej*.

Warszawa, dnia 2017 (podpis dyplomanta)

---

*Na podstawie Ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (Dz.U. 2005 nr 164 poz. 1365) Art. 239. oraz Ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. z 2000 r. Nr 80, poz. 904, z późn. zm.) Art. 15a. "Uczelni w rozumieniu przepisów o szkolnictwie wyższym przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli uczelnia nie opublikowała pracy dyplomowej w ciągu 6 miesięcy od jej obrony, student, który ją przygotował, może ją opublikować, chyba że praca dyplomowa jest częścią utworu zbiorowego."

Spis treści

1 Wstęp

Algorytmy Particle-in-Cell (ćząstka w komórce") to jedne z najbardziej zbliżonych do fundamentalnej fizyki metod symulacji materii w stanie plazmy. Zastosowany w nich lagranżowski opis cząsteczek pozwala na dokładne odwzorowanie dynamiki ruchu elektronów i jonów. Jednocześnie, ewolucja pola elektromagnetycznego na Eulerowskiej siatce dokonywana zamiast bezpośredniego obliczania oddziaływań międzycząsteczkowych pozwala na znaczące przyspieszenie etapu obliczenia oddziaływań międzycząsteczkowych. W większości symulacji cząsteczkowych właśnie ten etap jest najbardziej krytyczny dla wydajności progamu.

W ostatnich czasach symulacje Particle-in-Cell zostały wykorzystane m.in. do

symulacji przewidywanej turbulencji plazmy w reaktorze termojądrowym ITER

modelowania rekonekcji linii magnetycznych w polu gwiazdy

projektowania silników jonowych (Halla)

badania interakcji laserów z plazmą w kontekście tworzenia niewielkich, wysokowydajnych akceleratorów cząstek

Należy zauważyć, że w świetle rosnącej dostępności silnie równoległej mocy obliczeniowej w postaci kart graficznych możliwości algorytmów Particle-in-Cell będą rosły współmiernie, co może pozwolić na rozszerzenie zakresu ich zastosowań.

Inżynieria oprogramowania zorientowanego na wykorzystanie możliwości kart graficznych, jak również w ogólności nowoczesnych symulacji wykorzystujących dobrodziejstwa nowych technologii jest jednak utrudniona poprzez niskopoziomowość istniejących języków klasycznie kojarzonych z symulacją numeryczną (C, FORTRAN) oraz istniejących technologii zrównoleglania algorytmów (MPI, CUDA, OpenCL).

To sprawia, że pisanie złożonych programów symulacyjnych, zwłaszcza przez osoby zajmujące się głównie pracą badawczą (np. fizyką), a nie tylko programowaniem, jest znacznie utrudnione. Należy też zauważyć, że programy takie często są trudne, jeżeli nie niemożliwe do weryfikacji działania, ponownego wykorzystania i modyfikacji przez osoby niezwiązane z oryginalnym autorem z powodów takich jak

- brak dostępności kodu źródłowego
- niedostateczna dokumentacja
- brak jasno postawionych testów pokazujących, kiedy algorytm działa zgodnie ze swoim przeznaczeniem
- zależność działania kodu od wersji zastosowanych bibliotek, sprzętu i kompilatorów

Niniejsza praca ma na celu utworzenie kodu symulacyjnego wykorzystującego metodę Particle-in-Cell w popularnym języku wysokopoziomowym (Pythonie) przy użyciu najlepszych praktyk tworzenia reprodukowalnego, otwartego oprogramowania i zoptymalizowanie go w celu osiągania maksymalnej wydajności i sprawności obliczeniowej.

Ma to dwojaki sens:

1. zweryfikowanie możliwości leżących w obecnych językach wysokopoziomowych bez

wykorzystywania kart graficznych

2. położenie fundamentów pod

2 Część analityczno-teoretyczna

2.1 Fizyka plazmy

Plazma, powszechnie nazywana czwartym stanem materii, to zbiór zjonizowanych cząstek oraz elektronów. Plazmy występują w całym wszechświecie, od materii międzygwiezdnej po błyskawice. Ich istnienie uwarunkowane jest obecnością wysokich energii, wystarczających do zjonizowania atomów gazu.

Fizyka plazmy jest stosunkowo młodą nauką, której rozwój nastąpił dopiero w ostatnim stuleciu, zaczynając od badań Alfvena. Globalny wzrost zainteresowania fizyką plazmy rozpoczął się w latach ’50 ubiegłego wieku, gdy uświadomiono sobie, że można zastosować ją do przeprowadzania kontrolowanych reakcji syntezy jądrowej, które mogą mieć zastosowania w energetyce jako następny etap rozwoju po reakcjach rozpadu wykorzystywanych w "klasycznychęlektrowniach jądrowych.

Poza tym plazmy mają szerokie zastosowanie w przemyśle metalurgicznym, elektronicznym, kosmicznym itp.

2.2 Modelowanie i symulacja plazmy

Zjawiska z zakresu fizyki plazmy są jednymi z bardziej złożonych problemów modelowanie komputerowej. Głównym, koncepcyjnie, powodem uniemożliwiającym zastosowanie prostych metod symulacji znanych z newtonowskiej dynamiki molekularnej jest mnogość oddziaływań - każda cząstka oddziałowuje z każdą inną nawzajem poprzez niepomijalne na dużych odległościach oddziaływania kulombowskie $\approx r^{-2}$.

Z powodu dużej liczby cząstek w układach plazmowych, jedynymi praktycznymi podejściami są opisy statystyczne, opierające się na modelach kinetycznych. Wielkością opisującą plazmę jest tu funkcja dystrybucji zdefiniowana jako

$$\int \int f(\vec{x}, \vec{v}, t) d\vec{x} d\vec{v} = 1 \tag{1}$$

opisująca gęstość prawdopodobieństwa rozkładu plazmy w N-sześciowymiarowej przestrzeni fazowej (po trzy wymiary na położenia oraz prędkości, powielone dla każdej z N cząstek).

Podstawowym równaniem statystycznym opisującym plazmę jest równanie Vlasova

$$\frac{d}{dt} f_\alpha(\vec{x}, \vec{v}, t) - \nabla f - \nabla_{\vec{v}}(\vec{v} \times \vec{B} + \vec{E}) = f_{coll} \tag{2}$$

W praktyce jest ono również nierozwiązywalne. Jednym z powodów jest konieczność uzyskania dobrej rozdzielczości prędkości przy jednoczesnym zachowaniu zakresów obejmujących prędkości

relatywistyczne.

W modelowaniu komputerowym plazmy stosuje się dwa główne podejścia:

1. modele płynowe oparte na ciągłym opisie plazmy poprzez uśrednienie po dystrybucji wielkości termodynamicznych, co daje modele takie jak magnetohydrodynamikę
2. modele dyskretne oparte na samplowaniu dystrybucji plazmy przy użyciu dyskretnych cząstek (matematycznie jest to równanie Klimontowicza przybliżające równanie Vlasova)

Prawdopodobnie najpopoularniejszym modelem z tej drugiej kategorii są modele Particle-in-cell.

2.3 Modele Particle-in-cell

Idea modelu particle-in-cell jest wyjątkowo prosta i opiera się na idei przyspieszenia najbardziej złożonego obliczeniowo kroku symulacji dynamiki molekularnej, czyli obliczania sił międzycząsteczkowych. Cząstki poruszają się w ciągłej, Lagrange'owskiej przestrzeni. Ich ruch wykorzystywany jest do zebrania informacji dotyczącej gęstości ładunku i prądu na dyskretną, Eulerowską siatkę. Na siatce rozwiązane są (jako równania różniczkowe cząstkowe) równania Maxwella, dzięki którym otrzymuje się pola elektryczne i magnetyczne, które z powrotem są przekazane do położeń cząstek. Obliczeniowo, uwzględniając koszty odpowiednich interpolacji, pozwala to zredukować złożoność kroku obliczenia sił międzycząsteczkowych do $n \log n$ z n^2

Algorytm particle-in-cell składa się z czterech elementów

2.3.1 GATHER

Depozycja ładunku oraz prądu z położeń cząstek do lokacji na dyskretnej siatce poprzez interpolację, co pozwala na sprawne rozwiązanie na tej siatce równań Maxwella jako układu różnicowych równań cząstkowych zamiast obliczania skalujących się kwadratowo w liczbie cząstek oddziaływań kulombowskich między nimi. $1 = \sum_i S_i$<++>

2.3.2 SOLVE

Sprawne rozwiązanie równań Maxwella na dyskretnej, Eulerowskiej siatce; znalezienie pól elektrycznego i magnetycznego na podstawie gęstości ładunku i prądu na siatce. Istnieją dwie główne szkoły rozwiązywania tych równań: metody globalne i lokalne. Metody globalne wykorzystują zazwyczaj równania dywergencyjne, rozwiązywane iteracyjnie lub spektralnie, zaś lokalne - równania rotacyjne. Metody globalne nadają się do modeli elektrostatycznych, nierelatywistycznych. Metody lokalne pozwalają na ograniczenie szybkości propagacji zaburzeń do prędkości światła, co przybliża metodę numeryczną do fizyki zachodzącej w rzeczywistym układzie tego typu.

2.3.3 SCATTER

Interpolacja pól z siatki do lokacji cząstek, co pozwala określić siły elektromagnetyczne działające na cząstki. Należy przy tym zauważyć, że jako że interpolacja sił wymaga jedynie

lokalnej informacji co do pól elektromagnetycznych w okolicy cząstki, ta część algorytmu sprawia, że algorytmy Particle-in-cell doskonale nadają się do zrównoleglania (problem jest w bardzo dobrym przybliżeniu “trywialnie paralelizowalny”). Z tego powodu algorytmy Particle-in-cell nadają się doskonale do wykorzystania rosnącej mocy kart graficznych i architektur GPGPU.

2.3.4 PUSH

iteracja równań ruchu cząstek na podstawie ich prędkości (aktualizacja położeń) oraz działających na nie sił elektromagnetycznych (aktualizacja prędkości).

2.3.5 Makrocząstki

Należy zauważyć, że obecnie nie jest możliwe dokładne odwzorowanie dynamiki układów plazmowych w sensie interakcji między poszczególnymi cząstkami ze względu na liczbę cząstek W tym kontekście bardzo szczęśliwym jest fakt, że wszystkie istotne wielkości zależą nie od ładunku ani masy, ale od stosunku q/m. W praktyce stosuje się więc *makrocząstki*, obdarzone ładunkiem i masą będące wielokrotnościami tych wielkości dla cząstek występujących w naturze (jak jony i elektrony, pozwalając jednocześnie zachować gęstości cząstek i ładunku zbliżone do rzeczywistych.

Zazwyczaj (“tradycyjnie”) stosuje się gęstości cząstek (rzeczywistych) rzędu jednej dziesiątej gęstości krytycznej plazmy, która jest opisana wzorem

$$n_c = m_e \varepsilon_0 * (\frac{2\pi c}{e\lambda})^2 \tag{3}$$

2.4 Problem testowy

Problemem testowym, jakiego używamy do przetestowania wydajności działania algorytmu jest interakcja impulsu laserowego z tarczą składającą się ze zjonizowanego wodoru i elektronów.

Układ ten modelowany jest jako jednowymiarowy. Jest to tak zwany w literaturze model 1D-3D. O ile położenia cząstek są jednowymiarowe ze względu na znaczną symetrię cylindryczną układu, cząstki mają prędkości w pełnych trzech wymiarach. Jest to konieczne ze względu na oddziaływania cząstek z polem elektromagnetycznym propagującym się wzdłuż osi układu.

2.5 Python

Python jest wysokopoziomowym, interpretowanym językiem programowania, którego atutami są szybkie prototypowanie,

Python znajduje zastosowania w analizie danych, uczeniu maszynowym (zwłaszcza w astronomii). W zakresie symulacji w ostatnich czasach powstały kody skalujące się nawet w zakres superkomputerów, np. w mechanice płynów

Atutem Pythona w wysokowydajnych obliczeniach jest łatwość wywoływania w nim zewnętrznych bibliotek napisanych na przykład w C lub Fortranie, co pozwala na osiągnięcie

podobnych rezultatów wydajnościowych jak dla kodów napisanych w C.

3 Implementacja

3.1 Opis i treść kodu

Cały kod programu w celu reprodukowalności wyników tworzony był i jest dostępny na platformie Github

3.2 Struktura i hierarchia kodu

Program ma obiektową strukturę zewnątrzą, którą w celu łatwości zrozumienia jego działania nakrywa wewnętrzną warstwę składającą się głównie z n-wymiarowych tablic *NumPy* oraz zwektoryzowanych operacji na nich.

Kod składa się z kilku prostych koncepcyjnie elementów:

3.2.1 Species

Klasa reprezentująca pewną grupę makrocząstek (np. odpowiadające elektronom w symulacji).

Zawiera skalary:

- N - liczba makrocząstek
- q - ładunek makrocząstki
- m - masa makrocząstki
- c - prędkość światła

Zawiera tablice rozmiaru N:

- jednowymiarowych położeń makrocząstek x^n, zapisywanych w iteracjach $n, n+1, n+2...$
- trójwymiarowych prędkości makrocząstek $\vec{v}^{n+\frac{1}{2}}$, zapisywanych w iteracjach $n+\frac{1}{2}, n+32, n+52...$
- stanu makrocząstek (flagi boolowskie oznaczające cząstki aktywne bądź usunięte z obszaru symulacji)

Poza tym, zawiera też informacje dotyczące zbierania danych dot. cząstek:

- stringowy identyfikator grupy cząstek, dla potrzeb legend wykresów
- N_T - liczbę iteracji czasowych w symulacji
- odpowiadające poprzednio wymienionym tablice rozmiaru $N_T * N$

3.2.2 Grid

Klasa reprezentująca dyskretną siatkę, na której dokonywane są obliczenia dot. pól elektromagnetycznych.

Klasa zawiera:

- x_i - tablicę położeń lewych krawędzi komórek siatki
- N_G - liczbę komórek siatki

- Δx - krok przestrzenny siatki - $N_G * \Delta x$ daje długość obszaru symulacji
- ρ_i - tablicę gęstości ładunku na siatce
- $\vec{j}_{i,j}$ - tablicę gęstości prądu na siatce
- $E_{i,j}$ - tablicę pola elektrycznego na siatce
- $B_{i,j}$ - tablicę pola magnetycznego na siatce
- n_{species} - liczbę rodzajów cząstek w symulacji, na potrzeby wykresów
- c, ϵ_0 - stałe fizyczne - prędkość światła oraz przepuszczalność elektryczną próżni

3.2.3 Simulation

Klasa zbierająca w całość Grid oraz dowolną liczbę Species zawartych w symulacji, jak również pozwalająca w prosty sposób na wykonywanie iteracji algorytmu i analizy danych. Jest tworzona tak przy uruchamianiu symulacji, jak i przy wczytywaniu danych z plików .hdf5.

- Δt - krok czasowy
- N_T - liczba iteracji w symulacji
- *grid* - obiekt siatki
- *list_species* - lista grup makrocząstek w symulacji

3.2.4 Pliki pomocnicze

Poza powyższymi program jest podzielony na pliki:

- algorithms_grid - zawiera algorytmy dot. rozwiązywania równań Maxwella na dyskretnej siatce
- algorithms_interpolation - zawiera algorytmy dot. interpolacji z cząstek na siatkę i odwrotnie
- algorithms_pusher - zawiera algorytmy integrujące numerycznie równania ruchu cząstek
- animation - tworzy animacje dla celów analizy danych
- static_plots - tworzy statyczne wykresy dla celów analizy danych
- Plotting - zawiera ustawienia dot. analizy danych

Configi testowe są zawarte w plikach run_*:

- run_coldplasma
- run_twostream
- run_wave
- run_beam

Testy jednostkowe są zawarte w katalogu tests:

3.3 Wybrane algorytmy

3.3.1 Leapfrog oraz Borys

Należy zauważyć, że w inicjalizacyjnej iteracji algorytmu pomija się przesunięcie cząstek w przestrzeni, aktualizując jedynie prędkości. Krok czasowy jest wtedy ustawiany na minus połowę swojej zwykłej wartości, co pozwala na obliczenia z wykorzystaniem prędkości znanej w połowie

kroku czasowego między kolejnymi iteracjami czasowymi. Jest to tzw. algorytm leapfrog stosowany tam, gdzie potrzebna jest długofalowa stałość energii symulacji. Wynika to z symplektyczności tego rodzaju integratorów równań ruchu (w przeciwieństwie do, na przykład, standardowej metody Runge-Kutta 4, która mimo swej większej dokładności nie zachowuje energii cząstek.

W przypadku ruchu w polu magnetycznym nie wystarczy, niestety, użyć zwykłego algorytmu leapfrog. Używa się tutaj specjanej adaptacji tego algorytmu na potrzeby ruchu w zmiennym polu elektromagnetycznym, tzw pushera Borysa który rozbija pole elektryczne na dwa impulsy, między którymi następują dwie rotacje polem magnetycznym. Algorytm jest w ten sposób symplektyczny i długofalowo zachowuje energię cząstek.

W tym przypadku stosuje się relatywistyczny algorytm Borysa, w którym jedyną faktyczną poprawką względem nierelatywistycznego jest operowanie na pędach oraz sprowadzanie pędów do prędkości jako $\vec{v} = \vec{p}/\gamma$

3.3.2 Depozycja gęstości ładunku i prądu

3.3.3 Interpolacja pól elektrycznego i magnetycznego

3.3.4 Field solver

Ewolucja pola elektromagnetycznego opisana jest poprzez równania Maxwella. Jak pokazują Buneman i Ville numerycznie można zastosować dwa główne podejścia: 1. wykorzystać równania na dywergencję pola (prawa Gaussa) do rozwiązania pola na całej siatce. Niestety, jest to algorytm inherentnie globalny, w którym informacja o warunkach brzegowych jest konieczna w każdym oczku siatki 2. wykorzystać równania na rotację pola (prawa Ampera i Faradaya), opisujące ewolucję czasową pól. Jak łatwo pokazać (Buneman), dywergencja pola elektrycznego oraz magnetycznego nie zmienia się w czasie pod wpływem tak opisanej ewolucji czasowej:

Co za tym idzie, jeżeli rozpoczniemy symulację od znalezienia pola na podstawie warunków brzegowych i początkowych (gęstości ładunku), możemy już dalej iterować pole na podstawie równań rotacji. Ma to dwie znaczące zalety: * algorytm ewolucji pola staje się trywialny, zwłaszcza w 1D - ogranicza się do elementarnego dodawania i mnożenia * algorytm ewolucji pola staje się lokalny (do znalezienia wartości pola w danym oczku w kolejnej iteracji wykorzystujemy jedynie informacje zawarte w tym właśnie oczku i potencjalnie jego sąsiadach co zapobiega problemowi informacji przebiegającej w symulacji szybciej niż światło oraz zapewnia stabilność na podstawie warunku Couranta.

W 1D można dokonać dekompozycji składowych poprzecznych pola elektromagnetycznego (tutaj oznaczanych y, z) na propagujące się w przód ($+$) i w tył ($-$) obszaru symulacji. Składowe E_y, B_z są zebrane w

Wychodzimy z rotacyjnych równań Maxwella:

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} \nabla \times \vec{B} = \mu_0(\vec{j} + \epsilon_0 \frac{\partial \vec{E}}{\partial t}) \tag{4}$$

$$F^{+} = E_y + cB_z F^{-} = E_y - cB_z \tag{5}$$

Analogicznie, dla składowych E_z, B_y:

$$G^{+} = E_z - cB_y G^{-} = E_z + cB_y \tag{6}$$

Wyrazem “źródłowym” dla F, G jest prąd poprzeczny. Po dyskretyzacji równania, wyrażenie na ewolucję pól F, G między iteracjami przybiera postać:

$$F^{+}{}_{i+1}^{n+1} = F_n^{+} + j \tag{7}$$

Z tego powodu bardzo istotnym dla dokładnośći i stabilności algorytmu staje się sposób depozycji ładunku - należy pilnować, aby był robiony w sposób który spełnia zachowanie ładunku.

3.4 Wykorzystane biblioteki i technologie

3.4.1 Numpy

Numpy to biblioteka umożliwiająca wykonywanie obliczeń macierzowych. Pod zewnętrzną powłoką zawiera odwołania do wypróbowanych modułów LAPACK, BLAS napisanych w szybkich, niskopoziomowych językach C oraz FORTRAN. Jest to de facto standard większości obliczeń numerycznych w Pythonie.

3.4.2 scipy

Kolejną podstawową biblioteką w numerycznym Pythonie jest *scipy*, biblioteka zawierająca wydajne implementacje wielu podstawowych algorytmów numerycznych służących między innymi całkowaniu, optymalizacji, algebrze liniowej czy transformatom Fouriera.

3.4.3 Numba

Numba to biblioteka służąca do kompilacji just-in-time kodu. W wielu przypadkach pozwala na osiągnięcie kodem napisanym w czystym Pythonie wydajności marginalnie niższej bądź nawet równej do analogicznego programu w C. Jednocześnie

3.4.4 HDF5

HDF5 jest wysokowydajnym formatem plikow służącym przechowywaniu danych liczbowych w drzewiastej, skompresowanej strukturze danych. W Pythonie implementuje go biblioteka h5py. Używa się go na przykład w W bieżącej pracy wykorzystuje się go do przechowywania danych dot. symulacji, pozwalających na postprocessinge<F9><F8>

3.4.5 matplotlib

Do wizualizacji danych z symulacji (oraz tworzenia schematów w sekcji teoretycznej niniejszej pracy) użyto własnoręcznie napisanych wykresów w uniwersalnym pakiecie graficznym `matplotlib`. `matplotlib` zapewnie wsparcia zarówno dla grafik statycznych w różnych układach współrzędnych (w tym 3D), jak również dla dynamicznie generowanych animacji przedstawiających przebiegi czasowe symulacji.

3.4.6 py.test

Przy pracy nad kodem użyto frameworku testowego py.test Obsługa testów jest trywialna:

Należy zaznaczyć, że w numeryce, gdzie błędne działanie programu nie objawia się zazwyczaj błędem wykonywania programu, a jedynie błędnymi wynikami, dobrze zautomatyzowane testy jednostkowe potrafią zaoszczędzić bardzo dużo czasu na debugowaniu poprzez automatyzację uruchamiania kolejnych partii kodu i lokalizację błędnie działających części algorytmu. Dobrze napisane testy są praktycznie koniecznością w dzisiejszych czasach, zaś każdy nowo powstały projekt numeryczno-symulacyjny powinien je wykorzystywać, najlepiej do weryfikacji każdej części algorytmu z osobna.

3.5 Implementacja podstawowych algorytmów numerycznych stosowanych w symulacji

3.5.1 Depozycja ładunku i prądu

Depozycja ładunku i prądu polega na obliczeniu na podstawie położenia i prędkości każdej cząstki jej wkładu do gęstości prądu i ładunku na siatce, oraz zwiększeniu bieżących wartości o ten wkład.

W symulacji stosowana jest liniowa interpolacja - cząstki mają “szerokość” jednej komórki siatki, zaś ich wkład do i komórki jest proporcjonalny do przekrycia między powierzchnią cząstki a powierzchnią siatki.

Stosowany algorytm jest rekurencyjny i uwzględnia zachowanie ładunku/prądu co pozwala na uniknięcie stosowania poprawek Dzięki temu algorytm obliczania zmian w polu elektromagnetycznym znacznie się upraszcza.

$$\varepsilon_3 = \frac{x_n - (x_i + \frac{dx}{2})}{x_{n+1} - x_n} \tag{8}$$

$$\rho_i^{n+1} = \frac{q}{\Delta x}(x_i + \frac{\Delta x}{2} - x^{n+1}) \tag{9}$$

$$\rho_{i-1}^{n+1} = \frac{q}{\Delta x}(x^{n+1} - (x_i - \frac{\Delta x}{2})) \tag{10}$$

$$j_{x,i+1}^{n+1/2} = q(\frac{1}{2} + \frac{x^n - x_{i+1}}{\Delta x}) \tag{11}$$

$$j_{x,i}^{n+1/2} = q(\frac{1}{2} - \frac{x^{n+1} - x_i}{\Delta x}) \tag{12}$$

$$< \rho_{i+1} >= \frac{q\varepsilon}{2}(\frac{1}{2} - \frac{x_{i+1} - x^n}{\Delta x}) \tag{13}$$

$$< \rho_{i-1} >= q\frac{1-\varepsilon}{2}(\frac{1}{2} - \frac{x^n - x_i}{\Delta x}) \tag{14}$$

$$< \rho_i >= q\left(\frac{\varepsilon}{2}(\frac{3}{2} + \frac{x^n - x_i}{\Delta x}) + \frac{1-\varepsilon}{2}(\frac{3}{2} - \frac{x^{n+1} - x_{i+1}}{\Delta x})\right) \tag{15}$$

4 Część weryfikacyjna

Niniejsza analiza przeprowadzona została na “finalnej” w chwili pisania niniejszej pracy wersji programu. W repozytorium gita na Githubie jest to commit “placeholder” identyfikowany również jako wersja 1.0.

4.1 Przypadki testowe

Kod przetestowano w dwojaki sposób. Pierwszym z nich są testy jednostkowe. Poszczególne algorytmy podlegały testom przy użyciu ogólnodostępnego pakietu pytest

Testy polegały na przeprowadzeniu fragmentu symulacji - w przypadku testów algorytmów było to na przykład wygenerowanie pojedynczej cząstki o jednostkowej prędkości oraz zdepozytowanie jej gęstości prądu na siatkę, co pozwala porównać otrzymany wynik z przewidywanym analitycznie dla danego rozmiaru siatki i położenia cząstki. Automatyczne testy jednostkowe uruchamiane po każdej wymiernej zmianie kodu pozwalają kontrolować działanie programu znacznie ułatwiają zapobieganie błędom.

- Gather
  - Depozycja prądu z pojedynczej cząstki na niewielką siatkę
  - Depozycja prądu z dwóch pojedynczych cząstek na niewielką siatkę i porównanie z sumą prądów dla obu pojedynczyczh cząstek
  - Depozycja prądu z dużej ilości równomiernie rozłożonych cząstek
- Solve
  - Symulacja fali sinusoidalnej, obwiedni impulsu i złożenia tych dwóch propagujących się w próżni
- Scatter
  -
- Push

- Ruch w jednorodnym polu elektrycznym wzdłuż osi układu
- Ruch w jednorodnym polu magnetycznym z polem magnetycznym

Aby zweryfikować działanie kodu, zastosowano kod do symulacji kilku znanych problemów w fizyce plazmy:

4.1.1 oscylacje zimnej plazmy

4.1.2 niestabilność dwóch strumieni

4.2 Symulacja oddziaływania lasera z tarczą wodorową

Jako warunki początkowe przyjęto plazmę z liniowo narastającą funkcją rozkładu gęstości (jest to tak zwany obszar prejonizacji)

Gęstość rozkładu plazmy przyjęto jako

Początkowe prędkości cząstek przyjęto jako zerowe.

Za moc lasera przyjęto $10^{23} W/m^2$, zaś za jego długość fali 1.064 μm (jest to laser Nd:YAG)

Długość obszaru symulacji to

Prędkość światła c, stałą dielektryczną ε_0, ładunek elementarny e, masy protonu i elektronu m_p, m_e przyjęto według tablic, jak obrazuje następująca tabela:

4.3 Benchmarki - szybkość, zasobożerność

Do przeprowadzenia testów wydajności kodu użyto

4.4 Problemy napotkane w trakcie pisania kodu

5 Zakończenie

Utworzono kod symulacyjny implementacyjny algorytm particle-in-cell w Pythonie przy użyciu wszystkich dostępnych możliwości, jakie daje ekosystem open-source. Kod zoptymalizowano przy użyciu Otrzymane wyniki benchmarków pozwalają sądzić, że