

Politechnika Warszawska

W Y D Z I A Ł F I Z Y K I



Praca dyplomowa inżynierska

na kierunku Fizyka Techniczna
w specjalności Fizyka komputerowa

Implementacja i analiza wydajności programu do symulacji Particle-in-Cell
w języku Python

Dominik Stańczak

261604

promotor
dr Sławomir Jabłoński

WARSZAWA 2017

Streszczenie

Tytuł pracy: Implementacja i analiza wydajności programu do symulacji 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. Przeprowadzono analizę wydajności i skalowania programu. *Słowa kluczowe:*

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

(podpis opiekuna naukowego)

(podpis dyplomanta)

Abstract

Title of the thesis: Implementation and performance analysis of a Particle-in-Cell simulation code 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. Performance and scaling is analysed. *Keywords:*

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.:

Implementacja i analiza wydajności programu do symulacji 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.:

Implementacja i analiza wydajności programu do symulacji 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	13
2	Część analityczno-teoretyczna	15
2.1	Modelowanie i symulacja plazmy	16
2.2	Modele Particle-in-cell	17
2.2.1	Pętla obliczeniowa PIC	17
2.2.2	Makrocząstki	18
2.3	Problem testowy	20
3	Opis algorytmów PIC	21
3.1	Całkowanie równań ruchu	21
3.2	Komunikacja między cząstkami a siatką — depozycja i interpolacja	22
3.2.1	Depozycja ładunku	23
3.2.2	Interpolacja pól elektrycznego i magnetycznego	23
3.3	Depozycja prądu	24
3.3.1	Depozycja podłużna	25
3.3.2	Depozycja poprzeczna	26
3.4	Warunek początkowy pola elektrycznego: solver globalny, elektrostatyczny	27
3.5	Ewolucja czasowa pola elektromagnetycznego	28
3.5.1	Pole podłużne	29
3.5.2	Pole poprzeczne	29
3.6	Warunki początkowe dla cząstek	30
3.6.1	Warunki brzegowe	30
4	Opis i treść kodu PythonPIC	32
4.1	Grid — siatka Eulera	32
4.2	Species — cząstki	33
4.3	Simulation	34
4.4	Pliki pomocnicze	34
5	Wykorzystane oprogramowanie i biblioteki Pythona	36
5.1	Numpy	36
5.2	scipy	36
5.3	Numba	36
5.4	HDF5	37
5.5	matplotlib	37
5.6	py.test	37
5.7	Travis CI	38
5.8	snakeviz	39

5.9	cProfile	39
5.10	IPython, %timeit	40
6	Weryfikacja	41
6.1	Przypadki testowe	41
6.1.1	Testy algorytmiczne	41
6.2	Testy symulacyjne — przypadki elektrostatyczne	42
6.2.1	oscylacje zimnej plazmy	42
6.2.2	niestabilność dwóch strumieni	43
6.3	Testowe symulacje elektromagnetyczne — propagacja fali	43
6.4	Symulacja elektromagnetyczna — oddziaływanie wiązki laserowej z tarczą wodorową	44
6.5	Parametry uruchomienia	46
7	Wnioski	55
7.1	Potencjalne kierunki dalszej pracy	55
	Literatura	56

1 Wstęp

Algorytmy Particle-in-Cell (cząstka w komórce) to jedno 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 programu.

W ostatnich czasach symulacje Particle-in-Cell zostały wykorzystane między innymi 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. Przykładem takiego projektu jest PIConGPU [4].

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).

Należy też zauważyć [?], że programy takie często są trudne, jeżeli nie niemożliwe do weryfikacji działania, ponownego wykorzystania i modyfikacji przez osoby niezwiązane z pierwotnym autorem z powodów takich jak

- brak dostępności kodu źródłowego
- niedostateczna dokumentacja
- brak jasno postawionych testów pokazujących, kiedy program działa zgodnie z zamiarami twórców i kiedy daje wyniki zgodne z zakładanym modelem fizyki realizowanych w nim zjawisk
- zależność działania kodu od wersji zastosowanych bibliotek, sprzętu i kompilatorów

To sprawia, że pisanie złożonych programów symulacyjnych, zwłaszcza przez osoby zajmujące się głównie pracą badawczą (na przykład w dziedzinie fizyki), bez dogłębnego, formalnego przeszkolenia w programowaniu i informatyce, jest znacznie utrudnione.

Jest to też znacząca przeszkoda dla adaptacji w nauce modelu open science zakładanego przez organizacje unijne do przyjęcia jeszcze w bieżącym dziesięcioleciu, do roku 2020.

Niniejsza praca ma na celu utworzenie kodu symulacyjnego wykorzystującego metodę Particle-in-Cell do symulacji oddziaływania wiązki laserowej z tarczą wodorową w popularnym języku wysokopoziomowym Python, przy użyciu najlepszych praktyk tworzenia reprodukowalnego, otwartego oprogramowania i zoptymalizowanie go w celu osiągnięcia maksymalnej wydajności i

1. WSTĘP

sprawności obliczeniowej.

Utworzenie oprogramowania tego typu może również pozwolić na dalsze zastosowanie kodu w celach badawczych i jego dalszy rozwój, potencjalnie w większej liczbie fizycznych wymiarów lub z użyciem kart graficznych, lub w innych projektach.

Python został wybrany jako język służący do implementacji bieżącej symulacji przez jego rosnącą popularność zarówno w programowaniu ogólnym jak i w nauce, możliwości szybkiego prototypowania (zwłaszcza w rozbudowanych środowiskach jak IPython[13] i Jupyter[?]), oraz łatwość reprodukcji warunków uruchomienia.

Atutem Pythona w wysokowydajnych obliczeniach jest łatwość użycia w nim zewnętrznych bibliotek napisanych na przykład w C lub Fortranie, co pozwala na osiągnięcie podobnych rezultatów wydajnościowych jak dla kodów napisanych w językach niskopoziomowych przy minimalnej pracy z tymi językami. Istnieje również możliwość implementacji najbardziej intensywnych obliczeniowo fragmentów kodu w językach niskopoziomowych i odwoływanie się do nich z Pythona.

Python znajduje szerokie zastosowania w analizie danych i uczeniu maszynowym (zwłaszcza w astronomii[?]). W zakresie symulacji w ostatnich czasach powstały kody skalujące się nawet w zakres superkomputerów, na przykład w mechanice płynów. Nie można tu nie wspomnieć o utworzonym niedawno hydrodynamicznym kodzie PyFR, łączącym szybkie obliczenia w CUDA, OpenCL oraz OpenMP z wysokopoziomowością Pythona. Uruchomiono go na klastrze Piz Daint i udowodniono jego skalowanie (*weak scaling*) do 2000 kart NVIDIA K20X i 1.3 stabilnych petaflopów na sekundę. [17] [8]

Istotną zaletą Pythona, o której nie można nie wspomnieć, jest inherentna otwartość wśród użytkowników tego języka. Znacząca część projektów tworzonych w tym języku jest udostępnianych za darmo jako projekty *open source* w internecie na platformach takich jak GitHub, GitLab i Bitbucket, co znacząco ułatwia kolaborację i wspólne ich tworzenie przez społeczność. Jest to obszar, którego techniki wydają się być równie korzystne w społeczności naukowej.

2 Część analityczno-teoretyczna

Plazma, powszechnie nazywana czwartym stanem materii, to zbiór zjonizowanych cząstek oraz elektronów wykazujących jako grupa globalną obojętność elektryczną. Innymi słowy, od gazu plazmę odróżnia fakt, że cząstki są zjonizowane, więc oddziałują kolektywnie między sobą na odległość, ale ich pola elektryczne wzajemnie się neutralizują na długich dystansach.

Plazmy występują w całym wszechświecie, od materii międzygwiazdowej 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 Langmuira (1928), który eksperymentował z jonizowaniem gazów w szklanych rurach zwanych rurami Crookesa, służących do generowania promieniowania katodowego, czyli, jak wiemy obecnie, strumieni elektronów.[?]

Globalny wzrost zainteresowania fizyką plazmy na arenie geopolitycznej rozpoczął się w latach czterdziestych 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 obecnych elektrowniach jądrowych. Był to jeden z elementów zimnowojennego wyścigu technologicznego między Stanami Zjednoczonymi a ZSRR, jak również jeden z projektów mających na celu ponowne nawiązanie współpracy naukowej między supermocarstwami po zakończeniu tego konfliktu. Obecnie trwają intensywne badania nad tym problemem, których dotychczasową kulminacją jest budowany we Francji tokamak ITER i planowany reaktor DEMO.

Poza tym ogromnym projektem, plazmy mają szerokie zastosowania w obecnym przemyśle, na przykład:

- metalurgicznym — cięcie metalu przy użyciu łuków plazmowych.
- elektronicznym i materiałowym — żłobienie powierzchni urządzeń półprzewodnikowych, powierzchniowa obróbka materiałów, depozycja aktywnych jonów pod powierzchnią, czyszczenie powierzchni, depozycja cienkich warstw związków chemicznych na powierzchniach (CVD).
- kosmicznym — silniki plazmowe, interakcja z rozgrzanym powietrzem podczas powtórnego wchodzenia w atmosferę.
- użytkowym — ekrany telewizorów, oświetlenie (światłówki).

Należy też zwrócić uwagę, że ze względu na złożoność układów plazmowych pra-komputerowa fizyka miała ogromne problemy z merytorycznymi badaniami zachowania plazmy poza wybranymi, mocno uproszczonymi reżimami. Postęp w badaniach plazmy, jak sugeruje rozwój technologii kontrolowanej syntezy jądrowej, jest silnie skorelowany z rozwojem mocy obliczeniowej oraz algorytmów symulacyjnych.[?]

2. CZĘŚĆ ANALITYCZNO-TEORETYCZNA

2.1 Modelowanie i symulacja plazmy

Modelowanie zjawisk z zakresu fizyki plazmy jest jednym z bardziej złożonych problemów fizyki komputerowej. Konceptyjnie rzecz biorąc, Głównym powodem uniemożliwiającym zastosowanie prostych metod symulacji znanych z newtonowskiej dynamiki molekularnej jest mnogość oddziaływań — każda cząstka oddziałuje z każdą inną nawzajem poprzez niepomijalne oddziaływania kulombowskie, skalujące się z odległością jak $\approx r^{-2}$. Paradoksalnie, na dużych odległościach oddziaływania te znoszą się, co rozumie się w plazmie jako *kwaziobojętność* — jednak bez uwzględnienia oddziaływań od wszystkich cząstek nie osiągnie się tego efektu,

Z powodu dużej liczby cząstek w układach plazmowych, jedynym adekwatnym modelem teoretycznym opierającym się na fundamentalnej fizyce jest opis kinetyczny. Wielkością opisującą plazmę jest tu funkcja dystrybucji (zwana też funkcją rozkładu) zdefiniowana jako $f_s(\vec{x}, \vec{v}, t) d\vec{x} d\vec{v}$ opisująca gęstość rozkładu danej grupy cząstek s plazmy w sześciowymiarowej przestrzeni fazowej (po trzy wymiary na położenia oraz prędkości). Ewolucja czasowa funkcji rozkładu dokonuje się poprzez rozwiązanie wariantu równania Boltzmanna zwanego równaniem Vlasova, które sprzęga gęstości ładunku i prądu otrzymywane z funkcji dystrybucji z równaniami Maxwella na ewolucję pola elektromagnetycznego.

Dla przypadku plazmy złożonej z elektronów i jednego rodzaju jonów o ładunku Z_i :

$$\frac{\partial f_e}{\partial t} + \vec{v}_e \cdot \nabla f_e - e (\vec{E} + \vec{v}_e \times \vec{B}) \cdot \frac{\partial f_e}{\partial \vec{p}} = 0 \quad (1)$$

$$\frac{\partial f_i}{\partial t} + \vec{v}_i \cdot \nabla f_i + Z_i e (\vec{E} + \vec{v}_i \times \vec{B}) \cdot \frac{\partial f_i}{\partial \vec{p}} = 0 \quad (2)$$

$$\nabla \times \vec{B} = \mu_0 \left(\vec{j} + \epsilon_0 \frac{\partial \vec{E}}{\partial t} \right) \quad (3)$$

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} \quad (4)$$

$$\nabla \cdot \vec{E} = \rho / \epsilon_0 \quad (5)$$

$$\nabla \cdot \vec{B} = 0 \quad (6)$$

Równanie Vlasova może zostać rozszerzone do równania Fokkera-Plancka uwzględniającego bezpośrednie kolizje międzycząsteczkowe.

W praktyce równanie Vlasova jest trudne do rozwiązania poza trywialnymi przypadkami o ułatwiających problem symetriach. Jednym z powodów tej trudności jest konieczność uzyskania dobrej rozdzielczości prędkości przy jednoczesnym zachowaniu zakresów obejmujących prędkości relatywistyczne. Jako równanie różniczkowe cząstkowe, równanie Vlasova jest rozwiązywane na dyskretnych siatkach, należy zauważyć zaś, że skalowanie liczby punktów na siatce tego typu jest proporcjonalne do $N_r^3 N_v^3$, gdzie N_r to liczba punktów przestrzennych, zaś N_v to liczba punktów na siatce prędkości. Jest to więc często niepraktyczne obliczeniowo, między innymi ze względu na istotne w plazmach fuzyjnych zjawisko “uciekających elektronów” o relatywistycznych prędkościach.

2. CZĘŚĆ ANALITYCZNO-TEORETYCZNA

W modelowaniu komputerowym plazmy stosuje się trzy główne koncepcyjne podejścia:

- modele kinetyczne rozwiązujące bezpośrednio równania typu Vlasova na dyskretnych siatkach
- modele płynowe oparte na ciągłym opisie plazmy poprzez uśrednienie po dystrybucji wielkości termodynamicznych, co daje modele takie jak magnetohydrodynamika. Jest to wciąż układ równań różniczkowych cząstkowych, lecz na mniej wymiarowej siatce. Niestety, nie nadają się one do badań plazmy daleko od równowagi z powodu czynionych przy nich założeń takich jak maxwellowski rozkład prędkości. Znajdują za to szerokie zastosowanie w astronomii.
- modele dyskretne oparte na samplowaniu dystrybucji plazmy przy użyciu dyskretnych cząstek, pozwalające w prosty sposób uzyskać dobre przybliżenie faktycznego ruchu cząstek w plazmie i prądów generowanych tym ruchem. Pozwalają na wizualizację efektów działań pól, ale potrafią być mało wydajne do modelowania efektów kolektywnych.

czy na pewno należy rozróżniać kinetyczne i dyskretne?

Istnieje bardzo popularna klasa modeli łączących cechy kategorii pierwszej i trzeciej. Są to tak zwane modele Particle-in-cell.

2.2 Modele Particle-in-cell

Idea modelu *particle-in-cell* (dalej nazywanego *PIC*) jest wyjątkowo prosta i opiera się na przyspieszeniu 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 położenia cząstek. Obliczeniowo, uwzględniając koszty odpowiednich interpolacji, pozwala to zredukować złożoność kroku obliczenia sił międzycząsteczkowych do złożoności $O(n)$ z $O(n^2)$, gdzie n jest liczbą cząstek. Oczywiście, jest to okupione zależnością złożoności od liczby punktów na siatce m , lecz jako że $m \ll n$, jest to akceptowalne i korzystne.

2.2.1 Pętla obliczeniowa PIC

Obliczeniowo algorytm particle-in-cell składa się z czterech elementów powtarzających się cyklicznie:

- Zbierz (Gather)
Depozycja ładunku oraz prądu z położenia cząstek do lokacji na dyskretnej siatce poprzez interpolację, co pozwala na sprawne rozwiązanie na niej równań Maxwella jako układu różnicowych równań cząstkowych zamiast obliczania skalujących się kwadratowo w liczbie cząstek oddziaływań kulombowskich między nimi. W naszym elektromagnetycznym

2. CZĘŚĆ ANALITYCZNO-TEORETYCZNA

przypadku bardziej istotną jest depozycja prądu na siatkę, co szerzej tłumaczy następny fragment.

■ Rozwiąż (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 (prawa Gaussa)⁵⁶ rozwiązywane iteracyjnie (metodami takimi jak Gaussa-Seidela lub gradientów sprzężonych) lub spektralnie, przy użyciu transformat Fouriera lub Hankela [?]. Metody lokalne z kolei wykorzystują równania rotacyjne (prawa Ampera-Maxwella⁴ oraz Faradaya³):

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.

■ Rozprosz (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 sprawia, że algorytmy Particle-in-cell doskonale nadają się do zrównoleglenia (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.

■ Porusz (Push)

iteracja równań ruchu cząstek

$$d\vec{p}/dt = \vec{F} = q(\vec{E} + \vec{v} \times \vec{B}) \quad (7)$$

na podstawie ich prędkości (aktualizacja położeń) oraz działających na nie sił elektromagnetycznych (aktualizacja prędkości). Należy zauważyć, że modele PIC nie modelują bezpośrednich kolizji między cząstkami. Kolizje mogą jednak zostać dodane niebezpośrednio, na przykład poprzez metody Monte Carlo.

Jako że każda cząstka, zakładając znane pola elektromagnetyczne w jej położeniu, porusza się niezależnie, jest to kolejny fragment doskonale nadający się do zrównoleglenia.

2.2.2 Makrocząstki

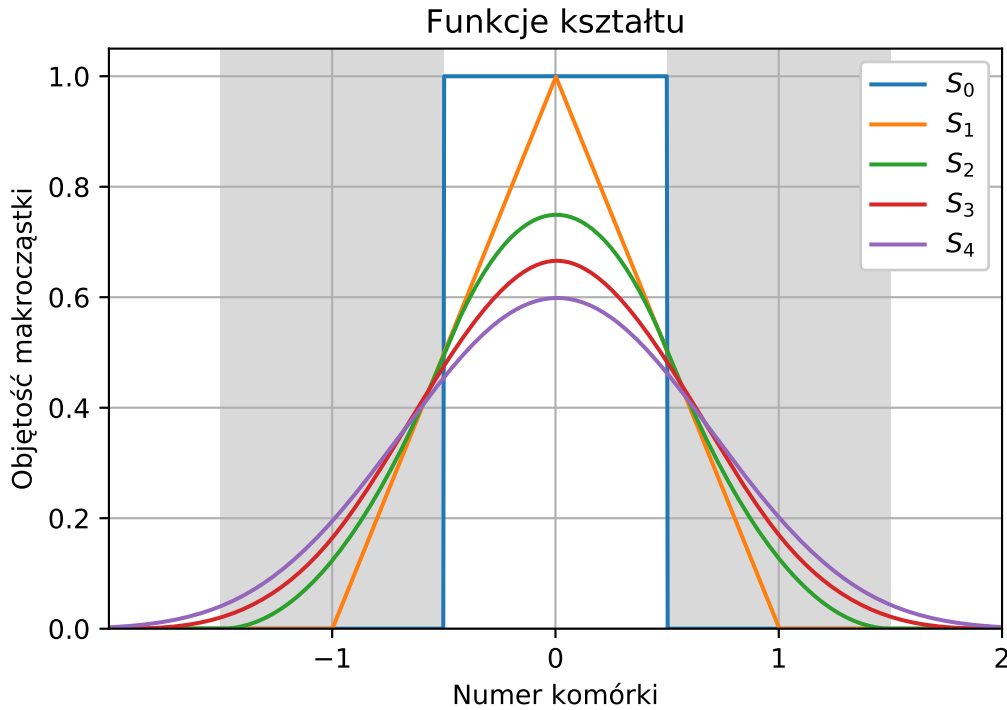
Należy zauważyć, że obecnie nie jest jeszcze możliwe dokładne odwzorowanie dynamiki układów plazmowych w sensie interakcji między poszczególnymi cząstkami ze względu na liczbę cząstek rzędu liczby Avogadro $\approx 10^{23}$. W tym kontekście bardzo korzystnym jest fakt, że wszystkie istotne wielkości 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 koncentracje

2. CZĘŚĆ ANALITYCZNO-TEORETYCZNA

cząstek, gęstości ładunku i prądu zbliżone do rzeczywistych.

Makrocząstki posiadają też kształt, który wyraża się w symulacji poprzez interpolację wielkości fizycznych znanych na siatce Eulera do położenia cząstek i odwrotnie. Matematycznie jest to opisane poprzez funkcję kształtu. Najprostszą jednowymiarową funkcją kształtu, oznaczaną S_0 , jest funkcja “cylindrowa”, która oznacza jednorodny rozkład gęstości cząstki w zakresie szerokości równym długości jednej komórki.

Inne funkcje kształtu tworzy się poprzez kolejne sploty funkcji podstawowej, co obrazuje rysunek 1. Funkcja S_1 (trójkątna, skupiona na środku) jest tradycyjną funkcją kształtu wykorzystywaną historycznie w większości zastosowań modelu PIC.



Rysunek 1: Typowe funkcje kształtu dla makrocząstki

W symulacjach elektromagnetycznych zazwyczaj stosuje się gęstości cząstek (rzeczywistych) rzędu jednej dziesiątej bądź setnej gęstości krytycznej plazmy n_c (wyrażaną wzorem 8), która oznacza taką koncentrację elektronów, przy której fala laserowa zaczyna być tłumiona zamiast być przepuszczaną przez plazmę.[5]

$$n_c = m_e \varepsilon_0 \left(\frac{2\pi c}{e\lambda} \right)^2 \quad (8)$$

gdzie m_e to masa spoczynkowa elektronu, ε_0 to przenikalność elektryczna próżni, c to prędkość światła w próżni, e to ładunek elementarny, zaś λ to długość fali.

Gęstość takiej makrocząstki, oznaczana n_{pic} , oznacza innymi słowy liczbę rzeczywistych

2. CZĘŚĆ ANALITYCZNO-TEORETYCZNA

cząstek, jakie reprezentuje sobą jedna makrocząstka.

2.3 Problem testowy

Głównym problemem testowym, jakiego używamy do przetestowania dokładności i 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 i możliwość dowolnego doboru kierunku polaryzacji promieniowania laserowego w symulacji.

Układ ten jest silnie zbliżony do rzeczywistych eksperymentów prowadzonych w Instytucie Fizyki Plazmy i Laserowej Mikrosyntezy.

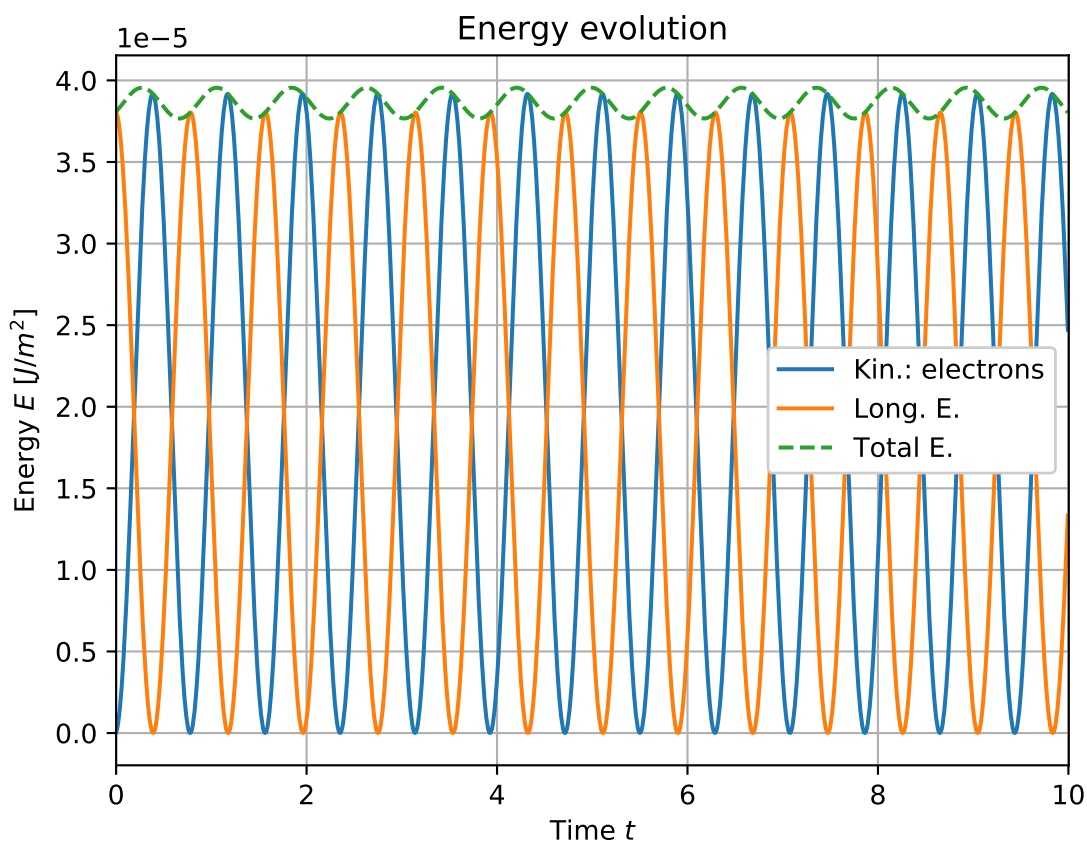
Tu bym chciał prosić o weryfikację.

3 Opis algorytmów PIC

3.1 Całkowanie równań ruchu

Każda symulacja cząstek wymaga zastosowania integratora równań ruchu. Tradycyjnym przykładem takiego integratora jest integrator Rungego-Kutty czwartego rzędu, znajdujący zastosowanie w wielorakich symulacjach.

Niestety, w bieżącym kodzie nie można go zastosować ze względu na jego niesymplektyczność: mimo ogromnej dokładności jest on niestabilny pod względem energii cząstek[12]. W symulacjach typu Particle-in-cell konieczne jest zastosowanie innych algorytmów. Dobrym algorytmem symplektycznym jest na przykład powszechnie znany *leapfrog*, polegający na przesunięciu prędkości o połowę iteracji czasowej względem położenia.[12] Mimo tego, że energia układu w ruchu cząstek obliczonym tym integratorem nie jest lokalnie stała na krótkich skalach czasowych, to jednak zachowuje energię na skali globalnej, co pokazuje rysunek 2.



Rysunek 2: Energia w symulacji elektrostatycznej. Ze względu na brak pola magnetycznego w tej symulacji używany jest efektywnie iterator typu leapfrog.

W przypadku ruchu przy niezerowym polu magnetycznym zwykły algorytm *leapfrog* okazuje

3. OPIS ALGORYTMÓW PIC

się być niewystarczający w celu zachowania energii cząstek¹. Zamiast niego, używa się specjalnej adaptacji tego algorytmu na potrzeby ruchu w zmiennym polu elektromagnetycznym, tak zwanego integratora Borysa (lepiej opisanego w [3]), który rozбивa pole elektryczne na dwa impulsy, między którymi następują dwie rotacje polem magnetycznym. Algorytm jest dzięki temu symplektyczny i długofalowo zachowuje energię cząstek.

$$\vec{v}^- = \vec{v}^{n-1/2} + \frac{qdt}{2m} \vec{E}^n \quad (9)$$

$$\vec{t} = \frac{qdt}{2m} \vec{B}^n \quad (10)$$

$$\vec{v}' = \vec{v}^- + \vec{v}^- \times \vec{t} \quad (11)$$

$$\vec{s} = \vec{t} / (1 + t^2) \quad (12)$$

$$\vec{v}^+ = \vec{v}^- + \vec{v}' \times \vec{s} \quad (13)$$

$$\vec{v}^{n+1/2} = \vec{v}^+ + \frac{qdt}{2m} \vec{E}^n \quad (14)$$

$$\vec{x}^{n+1} = \vec{x}^n + \vec{v}^{n+1/2} dt \quad (15)$$

W naszym przypadku dochodzi jeszcze jedno utrudnienie związane z relatywistycznymi prędkościami osiąganymi przez cząstki (zwłaszcza elektrony) w symulacji. Przed obliczeniem korekty prędkości konieczne jest przetransformowanie prędkości z układu "laboratoryjnego" \vec{v} na prędkość w układzie poruszającym się z cząstką \vec{u} , czego dokonuje się poprzez prostą transformację:

$$\vec{u} = \vec{v} \gamma \quad (16)$$

$$\gamma = \sqrt{1 + (v/c)^2} = 1 / \sqrt{1 + u^2} \quad (17)$$

Aktualizację prędkości należy więc przeprowadzić na \vec{u} , nie na \vec{v} . Po zakończeniu aktualizacji należy również powrócić do \vec{v} jako prędkości używanej do depozycji prądu.

3.2 Komunikacja między cząstkami a siatką — depozycja i interpolacja

Kolejnym krokiem pętli obliczeniowej po rozwiązaniu równań ruchu na aktualizację prędkości, po której — przypomnijmy — dysponujemy położeniami cząstek x^n w chwilach n oraz ich prędkościami $v^{n+1/2}$ w chwilach $n + 1/2$ jest obliczenie prądów podłużnych i poprzecznych potrzebnych do obliczenia wartości pól elektromagnetycznych w kolejnej iteracji.

W bieżącym programie gęstość ładunku jest tak naprawdę niepotrzebna w mechanice symulacji. Ewolucja pola następuje poprzez znajomość gęstości prądu. Jeżeli zaś pole elektromagnetyczne spełniało warunek prawa Gaussa 5 na początku, depozycja prądu w sposób zachowujący ładunek

¹Można to łatwo sprawdzić poprzez umieszczenie cząstki z niezerowym ładunkiem i prędkością w polu magnetycznym.

3. OPIS ALGORYTMÓW PIC

zapewni dalsze zachowanie tego warunku w kolejnych iteracjach. [16]

Wyjątek stanowi początek symulacji, w której pole faktycznie musi zostać obliczone od podstaw na podstawie gęstości ładunku. *PythonPIC* pozwala poradzić sobie z tym problemem. Pierwszą opcją jest ustawienie początkowych położenia ładunków na identyczne między cząstkami ujemnymi i dodatnimi, co pozwala na siłowe wyzerowanie gęstości ładunku i brak pola elektrycznego w rejonie symulacji.

Drugą, bardziej ogólną metodą pozwalającą na niezerowy rozkład gęstości ładunku jest zebranie gęstości ładunku z początkowych położenia cząstek i rozwiązanie równań na pola metodą globalną (na przykład spektralnie). Jest to jedyny moment, gdzie zebranie gęstości ładunku jest faktycznie konieczne. Mimo to, gęstość cząstek (proporcjonalna do gęstości ładunku jako $\rho = \sum_s q_s n_s$) jest wciąż zbierana w symulacji jako wygodna diagnostyka ewolucji przestrzennej plazmy w obszarze symulacji. Ze względu na prostotę algorytmu, nie zabiera ona dużo czasu obliczeniowego (do 2% czasu trwania symulacji).

3.2.1 Depozycja ładunku

Depozycja ładunku odbywa się w prosty sposób. Dla każdego gatunku cząstek obliczana jest ich gęstość liczbowa (koncentracja). Najpierw obliczane jest względne położenie każdej cząstki w komórce, do której przynależy, poprzez

$$x' = (x/dx) - i_x \quad (18)$$

gdzie i_x to indeks komórki, w której znajduje się cząstka.

Następnie gęstość cząstki jest rozkładana pomiędzy bieżącą komórkę a następną w stosunku $n_i = 1 - x'$, $n_{i+1} = x'$. Cząstka będąca w połowie komórki depozytowałaby więc swój ładunek po równo między obie komórki, co ilustruje rysunek 3.

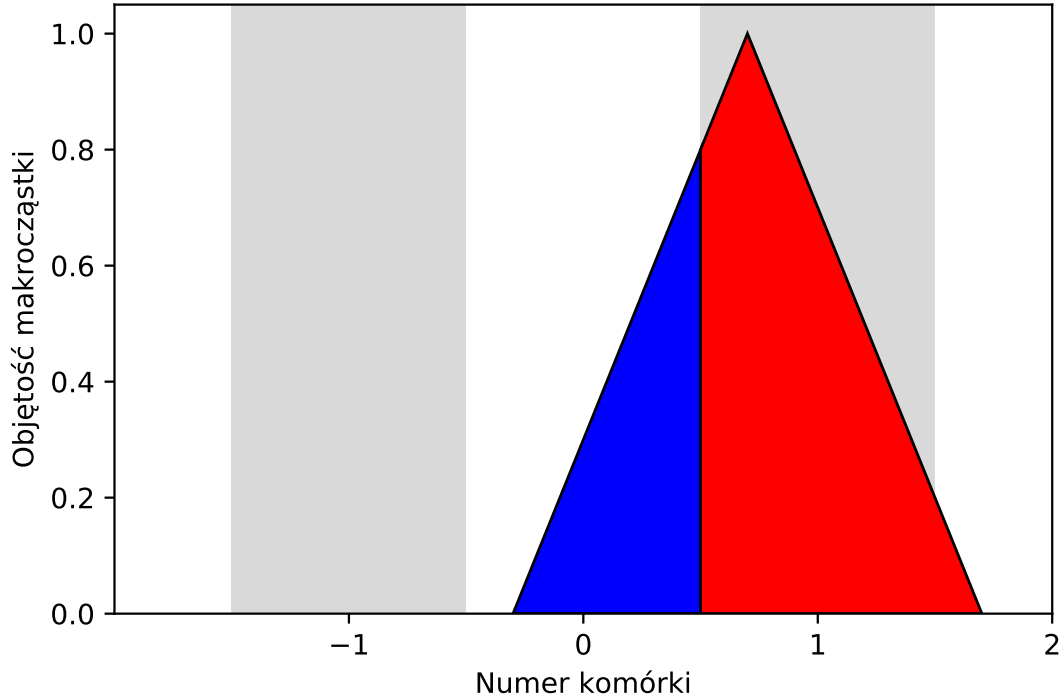
Po obliczeniu otrzymana tablica gęstości *makrocząstek* na siatce jest mnożona przez parametr *scaling* cząstek, co pozwala na obliczenie gęstości rzeczywistych cząstek modelowanych przez makrocząstki. Tablica gęstości ładunku jest otrzymywana poprzez zsumowanie tablic gęstości wszystkich gatunków cząstek w układzie.

3.2.2 Interpolacja pól elektrycznego i magnetycznego

Interpolacja pól elektrycznego i magnetycznego odbywa się na bardzo podobnej zasadzie, co depozycja ładunku. Wartości pól są liniowo skalowane do pozycji makrocząstek według ich względnych położenia (równanie 18) wewnątrz komórek.

$$F = F_i(1 - x') + F_{i+1}x' \quad (19)$$

3. OPIS ALGORYTMÓW PIC



Rysunek 3: Ilustracja depozycji ładunku poprzez skalowanie pola (area weighting). Do komórki 0 przypisana zostanie część ładunku proporcjonalna do niebieskiej części pola trójkąta, zaś do 1 - reszta.

3.3 Depozycja prądu

Depozycja prądu jest bardziej złożonym zagadnieniem niż depozycja ładunku. Prądy potrzebują bowiem informacji o prędkości cząstek w połówkowych iteracjach $v^{n+1/2}$, ale sama interpolacja do komórek siatki wymaga położeń w iteracji całkowitej x^n . Poza tym jest też kwestia, że zwykle liniowe przeskalowanie ładunku i parametru scaling przez prędkość w danym kierunku nie jest wystarczające z tego powodu, że taki sposób depozycji nie spełnia warunku zachowania ładunku. Strumień prądu podłużny powinien reprezentować zmianę gęstości ładunku w czasie. W postaci różniczkowej można to zapisać, za [16]:

$$\nabla \cdot \vec{J} = -\partial \rho / \partial t \quad (20)$$

To zaś dyskwalifikuje prostą liniową interpolację jako metodę depozycji prądu, ponieważ chcemy rozwiązywać lokalne równania Ampere'a-Maxwella zamiast liczyć globalne równania Poissona i Gaussa. Jeżeli chcemy uniknąć obliczania poprawek korygujących dywergencję pól, musimy to okupić większą złożonością algorytmu depozycji prądu.

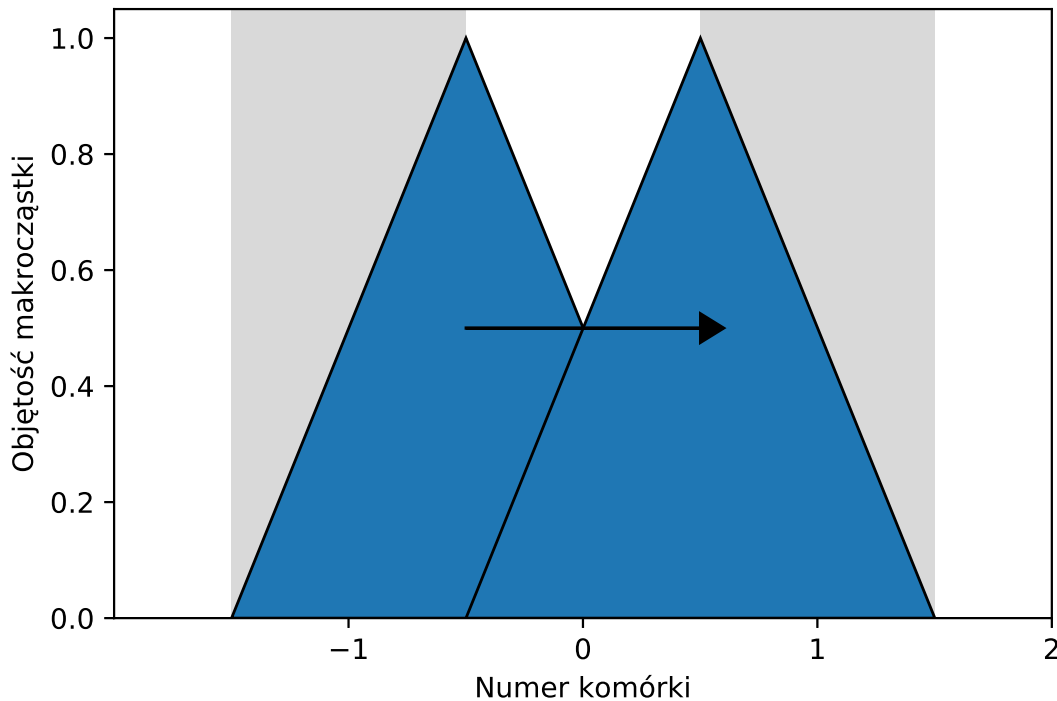
Rozważamy tu zastosowany w bieżącej symulacji kształt cząstki S_1 (trójkątny).

Depozycja prądu jest rozbita na dwie części — prąd podłużny i prąd poprzeczny. Jest to

3. OPIS ALGORYTMÓW PIC

spowodowane tym, że ruch w kierunkach y, z nie powoduje w symulacji przesuwania cząstek, a w x już tak. Obliczenie prądu w kierunkach poprzecznych jest więc jednoznaczne z obliczeniem skalowanego *przekrycia* cząstek z komórkami, przez które przechodzą w trakcie ruchu.

Prędkość cząstek jest ograniczona przez prędkość światła, zaś za krok czasowy przyjmujemy $\Delta t = \Delta x/c$, można więc łatwo zauważyć, że maksymalna odległość, jaką cząstka mogłaby przebyć w jednej iteracji, to Δx . Jeżeli cząstka wystartowałaby środkiem w lewej krawędzi komórki (swoją własną lewą krawędzią sięgając środka poprzedniej komórki, a prawą połowy obecnej komórki), to w czasie ruchu przemieściłaby się przez łącznie trzy komórki. Ilustruje to rysunek 4.



Rysunek 4: Ilustracja ruchu cząstki poruszającej się z maksymalną prędkością dozwoloną w symulacji.

Algorytm rozбивa ruch cząstek w bieżącej iteracji na etapy. W każdym z etapów cząstka obejmuje swoim zasięgiem inne komórki. Pętla obliczeniowa akumuluje prąd dla cząstek w następujący sposób, zaczynając z przydzielonym czasem w bieżącej iteracji $t = \Delta t$.

3.3.1 Depozycja podłużna

1. obliczany jest czas potrzebny cząstce na dotarcie do kolejnej granicy obszaru

$$t_l = (x - x_g)/v \quad (21)$$

3. OPIS ALGORYTMÓW PIC

Granica obszaru jest definiowana jako najbliższa połowa komórki Eulera w kierunku ruchu cząstki.

2. ■ jeżeli cząstka ma wystarczająco czasu t (spełnia warunek $t > t_l$) aby dotrzeć do kolejnej granicy obszaru, jej prąd zostaje zdepoztowany przez czas t_l ,
■ W przeciwnym razie prąd jest depozytowany przez czas t i ruch cząstki kończy się.

Wkład cząstki do depozytowanego prądu wynosi

$$j_x = qnv_x t / \Delta t \quad (22)$$

gdzie q to ładunek cząstki, a n to parametr `scaling` oznaczający liczbę rzeczywistych cząstek w makrocząstce.

3. Jeżeli ruch cząstki się nie skończył, zostaje ona (niezależnie od jej faktycznego ruchu w symulacji) w pętli wewnętrznej, przesunięta za granicę obszaru o $\varepsilon = 10^{-10} \Delta x$. Dodanie ε jest wykonywane w celu zwiększenia jednoznaczności porównywania liczb zmiennoprzecinkowych przy wybieraniu gałęzi algorytmu.

Pętla obliczeniowa wykonuje kolejną iterację dla takiej cząstki, która tym razem ma przydzielony czas $t' = t - t_l$.

Depozytowany prąd w danej iteracji trafia do komórki, w której cząstka zaczynała ruch. W ten sposób można zagwarantować, że cząstka zdeponuje prąd do wszystkich komórek, w których się poruszała.

W każdej iteracji pętli obliczeniowej prąd ze wszystkich cząstek jest akumulowany do tablicy prądów na siatce. Tablica ma rozmiar NG+3 w przypadku depozycji podłużnej i NG+4 w przypadku depozycji poprzecznej, co spowodowane jest koniecznością obsłużenia warunków brzegowych. W obecnej symulacji nieokresowej prąd na zewnątrz wewnętrznego obszaru właściwego NG jest zerowany, zaś w symulacji okresowej - odbijany na obszar wewnętrzny po przeciwnej stronie siatki.

3.3.2 Depozycja poprzeczna

Algorytm depozycji poprzecznej jest identyczny do depozycji podłużnej, poza samym etapem obliczania wielkości depozytowanego prądu. Jako że symulacja jest jednowymiarowa i cząstki nie wykonują faktycznego ruchu w kierunkach y , z , przyjmujemy liniową interpolację prądów w tych kierunkach jako iloczynów $\vec{j} = q\vec{v}$ ważonych przez przekrycie cząstki z komórkami oraz czas, jaki cząstka spędza w ruchu obejmując dane komórki.

Przekrycie z centrum bieżącej komórki w danej chwili obliczamy jako

$$s_0 = 1 \pm (x' + 0.5) \quad (23)$$

gdzie znak to $+$, gdy cząstka znajduje się w lewej połowie komórki. Zmianę przekrycia w trakcie

3. OPIS ALGORYTMÓW PIC

ruchu, jak i końcowe przekrycie, obliczamy z tym samym znakiem jako

$$s_1 = s_0 \pm \Delta s = s_0 \pm v_x t / dx \quad (24)$$

Aby uzyskać wartość średniego przekrycia cząstki z komórką w trakcie ruchu uśredniamy przekrycie na końcu ruchu z przekryciem na początku ruchu,

$$w = (s(x^{n+1}) + s(x^n)) / 2 \quad (25)$$

Ostatecznie wkład do bieżącej komórki w ciągu jednej iteracji pętli obliczeniowej to $wqnv_{y,z}t/\Delta t$. Zaś do dalszej komórki, w której stronę skierowany jest ruch, jest to $(1-w)qnv_{y,z}t/\Delta t$.

Tablica prądów poprzecznych, do których zostają depozytowane, ma rozmiar $(NG+4, 2)$. Jest to spowodowane faktem, że w liniowej depozycji cząstki mogą objąć kształtem maksymalnie dwie komórki, przez co należy dodać komórkę "graniczną" na każdej krawędzi rejonu symulacji.

3.4 Warunek początkowy pola elektrycznego: solver globalny, elektrostatyczny

Jako że symulacja jest jednowymiarowa, musimy jedynie znaleźć równowagowe pole elektryczne E_x .

Za Birdsall i Langdonem [3] wychodzimy z równania Poissona i dokonujemy obustronnej transformaty Fouriera w przestrzeni na polu elektrycznym i gęstości ładunku, zakładając możliwość rozłożenia obu tych wielkości na składowe mające następującą zależność od położenia:

$$y(x) = \sum_k y(k) e^{-ikx} \quad (26)$$

Przy tym założeniu dywergencja na skutek zmienności przestrzennej jedynie na osi x zamienia się na operator

$$\nabla \cdot = \frac{\partial}{\partial x} = -ik \quad (27)$$

W związku z tym równanie Poissona zamienia się na (po przekształceniu):

$$E(k) = \frac{\rho(k)}{-ik\epsilon_0} \quad (28)$$

Elementy tablicy k można otrzymać w następujący sposób:

$$k_n = 2\pi n / L \quad (29)$$

Algorytm pozwalający otrzymać równowagowe pole elektrostatyczne w spojrzeniu wysokopoziomowym jest więc relatywnie prosty:

- Używamy algorytmu FFT na gęstości prądu zebranej z cząstek na siatkę
- Dzielimy wynikową tablicę przez $-ik\epsilon_0$, używając do tego wbudowanego typu `complex`

3. OPIS ALGORYTMÓW PIC

obsługiwanego przez Numpy jak i zwykłego Pythona

- Używamy odwrotnej transformaty Fouriera, by otrzymać $E_x(x)$.

Jedynym problemem jest element $k_n = 0$, utrudniający dzielenie. Istnieją dwie możliwości:

- wymuszenie $k_0 = k_1/\text{const}$, gdzie const jest dostatecznie duże (np. 10^6)
- ustawienie $E(k_0) = 0$, co jest jednoznaczne z wyzerowaniem średniego ładunku w obszarze symulacji. Można to interpretować jako ustawienie nieruchomego tła neutralizującego, co jest wykorzystywane w okresowych symulacjach elektrostatycznych.

3.5 Ewolucja czasowa pola elektromagnetycznego

Ewolucja pola elektromagnetycznego opisana jest poprzez równania Maxwella. Jak pokazują Buneman i Villasenor, numerycznie można zastosować dwa główne podejścia: [16]

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 każdej komórce siatki.
2. wykorzystać równania na rotację pola (prawa Ampera i Faradaya), opisujące ewolucję czasową pól. Jak łatwo pokazać (Buneman [16]), 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:

1. algorytm ewolucji pola staje się trywialny obliczeniowo, zwłaszcza w 1D — ogranicza się bowiem do elementarnych operacji lokalnego dodawania i mnożenia.
2. 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 schemacie ewolucji czasowej wykorzystanym w programie wychodzimy z rotacyjnych równań Maxwella 4, 3.

Jako że symulacja zakłada symetrię układu wzdłuż osi propagacji lasera x , możemy przyjąć $\frac{\partial}{\partial y} = \frac{\partial}{\partial z} = 0$. Jednocześnie z prawa Gaussa dla pola magnetycznego wynika $B_x = 0$. Stąd równania dla pola elektrycznego podłużnego oraz pola elektromagnetycznego poprzecznego rozłączają się i można je rozwiązywać praktycznie oddzielnie.

3.5.1 Pole podłużne

$$(\nabla \times \vec{B})_x = 0 = \mu_0 \left(j_x + \varepsilon_0 \frac{\partial E_x}{\partial t} \right) \quad (30)$$

Składowa podłużna pola jest zatem obliczana poprzez wyrażenie

$$\frac{\partial E_x}{\partial t} = -\frac{j_x}{\varepsilon_0} \quad (31)$$

a jej dyskretny odpowiednik to:

$$E_i^{n+1} = E_i^n - \frac{\Delta t}{\varepsilon_0} j_{x,i}^{n+1/2} \quad (32)$$

Z tego powodu bardzo istotnym dla dokładności i stabilności algorytmu staje się sposób depozycji ładunku — należy pilnować, aby był robiony w sposób który spełnia zachowanie ładunku. Inaczej koniecznym staje się aplikowanie tak zwanej poprawki Borysa[?], aby upewnić się, że warunek z równań Gaussa 5,6 jest wciąż spełniony.

Należy zwrócić uwagę, iż w tej wersji pole elektrostatyczne E_x w danej komórce jest aktualizowane jedynie na podstawie informacji w tej właśnie komórce.

3.5.2 Pole poprzeczne

Wyprowadzenie dla pola poprzecznego przedstawimy, za [3], dla składowych E_y oraz B_z . Wyprowadzenie dla E_z , B_y jest analogiczne i przedstawimy jedynie jego wyniki.

Sumując równania 4 oraz przemnożone przez prędkość światła jako prędkość propagacji fali elektromagnetycznej 3 dla składowej \hat{y} :

$$\frac{\partial}{\partial t} (E_y + cB_z) = -j_y/\varepsilon_0 - c \frac{\partial}{\partial x} (E_y + cB_z) \quad (33)$$

$$\frac{\partial}{\partial t} (E_y - cB_z) = -j_y/\varepsilon_0 + c \frac{\partial}{\partial x} (E_y - cB_z) \quad (34)$$

Przekształcając:

$$\left(\frac{\partial}{\partial t} + c \frac{\partial}{\partial x} \right) (E_y + cB_z) = -j_y/\varepsilon_0 \quad (35)$$

$$\left(\frac{\partial}{\partial t} - c \frac{\partial}{\partial x} \right) (E_y - cB_z) = -j_y/\varepsilon_0 \quad (36)$$

Możemy więc uzyskać wielkości $F^+ = E_y + cB_z$ oraz $F^- = E_y - cB_z$ reprezentujące liniowo spolaryzowane fale elektromagnetyczne, które propagują się po siatce z prądem j_y jako wyrazem źródłowym.

Wyniki dla E_z , B_y zrealizowane poprzez wprowadzenie wielkości $G^\pm = E_z \pm cB_y$ przedstawiamy

3. OPIS ALGORYTMÓW PIC

za [3]:

$$\left(\frac{\partial}{\partial t} + c\frac{\partial}{\partial x}\right)(E_z + cB_y) = -j_z/\varepsilon_0 \quad (37)$$

$$\left(\frac{\partial}{\partial t} - c\frac{\partial}{\partial x}\right)(E_z - cB_y) = -j_z/\varepsilon_0 \quad (38)$$

Finalny schemat jest superpozycją rozwiązań dla F^\pm i G^\pm .

Poprzez dobór kroku czasowego symulacji $\Delta t = \Delta x/c$ uzyskujemy trywialną numeryczną propagację wielkości F^\pm [?]:

$$F_{i+1}^{+,n+1} = F_i^{+,n} - \Delta t j_{y_{i+1/2}}^{n+1/2}/\varepsilon_0 \quad (39)$$

Wielkości z minusem propagują się w lewo, zaś wielkości G^\pm wykorzystują prąd j_z .

Odzyskanie fizycznych pól elektrycznych i magnetycznych jest równie trywialne, na przykład

$$E_y = (F^+ + F^-)/2 \quad (40)$$

3.6 Warunki początkowe dla cząstek

W celu dobrania warunków początkowych wykorzystuje się algorytm opisany w [3]. Jego działanie można łatwo zilustrować na przykładzie początkowej funkcji gęstości cząstek zadanej dowolną funkcją wyrażoną wzorem analitycznym.

Używając funkcji dystrybucji w jednym wymiarze, zależnej jedynie od położenia i znormalizowanej do liczby cząstek N , można wykonać całkowanie kumulative na siatce gęstszej niż liczba cząstek na wybranym przedziale, po czym umieścić cząstki w miejscach, gdzie obliczona dystrybuanta funkcji przybiera kolejne większe całkowite wartości.

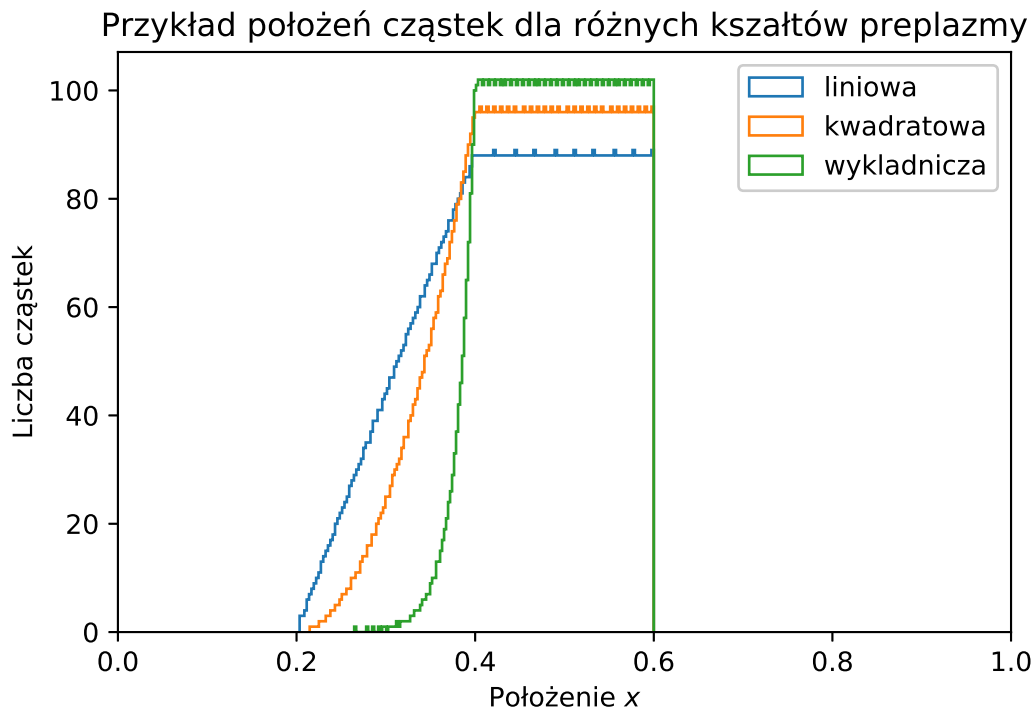
Zaimplementowany algorytm jest w stanie przyjąć dowolną funkcję analityczną przybierającą wartości z zakresu $(0, 1)$ i znormalizować ją tak, aby $\int_0^L f(x)dx = N$. W praktyce wykorzystuje się wartość normalizacji marginalnie większą niż N , mianowicie $N + 0.1$, co pozwala na uniknięcie problemów ze skończoną dokładnością obliczeń na liczbach zmiennoprzecinkowych oraz rzutowaniem na liczby całkowite.

W bieżącej wersji symulacji zorientowanej pod symulację układu tarczy wodorowej istnieją trzy wbudowane rodzaje ustawień początkowych cząstek. Wszystkie trzy składają się z części o jednorodnym rozkładzie gęstości oraz ustawionej przed nią "preplazmy" mającej rozkład gęstości przyrastający z gęstością odpowiednio liniowo, kwadratowo i wykładniczo ($\exp 10x - 10$).

W bieżącej wersji kodu prędkości cząstek inicjalizowane są jako zera. Jest to uzasadnione niewielkim znaczeniem początkowej prędkości cząstek w porównaniu do efektów, jaki ma na nich impuls laserowy.

3.6.1 Warunki brzegowe

Program implementuje dwa rodzaje warunków brzegowych dla cząstek:



Rysunek 5: Zaimplementowane w symulacji kształty preplazmy.

- okresowy, do sytuacji o symetrii translacyjnej (cząstka opuszczająca rejon symulacji z prawej strony wraca do niego z lewej strony),
- nieokresowy, do skończonych układów bez symetrii translacyjnej (cząstki opuszczające rejon symulacji zostają usunięte i nie są brane dalej pod uwagę).

Przygotowane są również analogiczne wrappery algorytmów obliczeniowych depozycji i interpolacji.

Dla pól możliwe jest dołączenie do symulacji obiektu reprezentującego Laser, generującego falę elektromagnetyczną o wybranej polaryzacji na lewej krawędzi obszaru symulacji.

4. OPIS I TREŚĆ KODU PYTHONPIC

4 Opis i treść kodu PythonPIC

Cały kod programu w celu ułatwienia jego użycia i reprodukowalności wyników tworzony był i jest dostępny² na platformie GitHub.

Program ma obiektową strukturę zewnętrzną, którą w celu łatwości zrozumienia jego działania nakrywa wewnętrzna warstwa składająca się głównie z n-wymiarowych tablic `numpy.ndarray` oraz zwektoryzowanych operacji na nich.

Część symulacyjna kodu składa się z kilku prostych koncepcyjnie elementów:

4.1 Grid — siatka Eulera

Klasa reprezentująca dyskretną siatkę Eulera, na której dokonywane są obliczenia dotyczące pól elektromagnetycznych oraz gęstości ładunku i prądu. Zawiera dane historyczne dotyczące wielkości na siatce i linki do plików `.hdf5` przechowujących te wielkości. Posiada dwie subklasy do praktycznych zastosowań w różnego rodzaju symulacjach: `PeriodicGrid` oraz `NonperiodicGrid`.

Zawiera następujące parametry:

- x_i — tablicę położenia lewych krawędzi komórek siatki
- N_G — liczbę komórek siatki
- T — sumaryczny czas trwania symulacji
- 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.
- $E_{i,j}, B_{i,j}$ — tablice pól elektrycznego i magnetycznego na siatce.
- c, ε_0 — stałe fizyczne — prędkość światła oraz przenikalność elektryczną próżni.
- Δt — krok czasowy symulacji, obliczony jako $\Delta t = \Delta x / c$.
- N_T — liczbę iteracji czasowych symulacji.
- BC — Boundary Condition, obiekt zawierający informacje dotyczące warunku brzegowego zadanego na pole elektryczne i magnetyczne. W przypadku symulacji laserowej jest to instancja klasy `Laser` zawierająca informacje o długości fali, kształcie obwiedni i polaryzacji impulsu. W każdej iteracji BC poprzez metodę `apply` aktualizuje wartości na lewym brzegu siatki obliczeniowej.

Istotne metody klasy `Grid`, o których należy wspomnieć, to:

- `apply_bc` — aktualizuje krańcowe wartości tablic E, B w oparciu o podany warunek brzegowy (BC).
- `gather_current` — zbiera ładunek z cząstek na siatkę
- `gather_charge` — zbiera prąd z cząstek na siatkę
- `init_solve` — uruchamia początkową, spektralną iterację obliczającą równowagowe pole elektrostatyczne na podstawie gęstości ładunku z cząstek
- `solve` — uruchamia iterację ewolucji czasowej pola elektromagnetycznego na podstawie prądów

²<https://github.com/StanczakDominik/PythonPIC>

4. OPIS I TREŚĆ KODU PYTHONPIC

w symulacji

- `field_function` — interpoluje pola elektryczne i magnetyczne do danych położzeń, używane do obliczenia pól w cząstkach
- `save_field_values` — zapisuje dane dotyczące wartości historycznych na siatce w danej iteracji. Należy zaznaczyć, że wielkości na komórkach granicznych nie są zapisywane.
- `postprocess` — przetwarza zebrane w czasie symulacji minimalne informacje dotyczące pól, prądów i ładunków na informacje dotyczące energii

4.2 Species — cząstki

Klasa reprezentująca pewną grupę makrocząstek o wspólnych cechach, takich jak ładunek bądź masa. Przykładowo, w symulacji oddziaływania lasera z tarczą wodorową jedną grupą są protony, zaś drugą — elektrony. Do zainicjalizowania wymaga instancji `Grid`, z której pobiera informacje takie jak stałe fizyczne c , ε_0 , liczbę iteracji czasowych N_T i czas trwania iteracji Δt .

Zawiera skalary:

- N — liczba makrocząstek
- q — ładunek cząstki
- m — masa cząstki
- `scaling` — liczba rzeczywistych cząstek, jakie reprezentuje sobą makrocząstka. Jej sumaryczny ładunek wynosi $q \cdot \text{scaling}$, masa $m \cdot \text{scaling}$.
- `N_alive` — liczba cząstek obecnie aktywnych w symulacji. Zmniejsza się w miarę usuwania cząstek przez warunki brzegowe.

Poza skalarami zawiera tablice rozmiaru N :

- jednowymiarowych położzeń makrocząstek x^n , zapisywanych w iteracjach $n, n+1, n+2 \dots$
- trójwymiarowych prędkości makrocząstek $\vec{v}^{n+\frac{1}{2}}$, zapisywanych w iteracjach $n+\frac{1}{2}, n+\frac{3}{2}, n+\frac{5}{2} \dots$
- stanu makrocząstek (flagi oznaczające cząstki aktywne bądź usunięte z obszaru symulacji)

Poza tym, zawiera też informacje dotyczące zbierania danych diagnostycznych dla cząstek, niepotrzebnych bezpośrednio w czasie symulacji:

- `name` — słowny identyfikator grupy cząstek, dla potrzeb legend wykresów
- N_T — liczbę iteracji czasowych w symulacji
- N_T^s — zmniejszoną liczbę iteracji, w których następuje pełne zapisanie położzeń i prędkości cząstek. Dane te są wykorzystywane do tworzenia diagramów fazowych cząstek.
- odpowiadające poprzednio wymienionym tablice rozmiaru (N_T^s, N) , $(N_T^s, N, 3)$.
- jedną tablicę rozmiaru (N_T, N_G) dotyczącą zebranych podczas depozycji ładunku informacji diagnostycznych o przestrzennej gęstości cząstek.
- trzy tablice rozmiaru (N_T) dotyczącą średnich prędkości, średnich kwadratów prędkości i odchyłeń standardowych prędkości.

Jeżeli liczba makrocząstek lub iteracji przekracza pewną stałą (rzędu 10^4), dane zapisywane są jedynie dla co n -tej cząstki, gdzie n jest najniższą liczbą całkowitą która pozwala na zmniejszenie tablic poniżej tej stałej.

4. OPIS I TREŚĆ KODU PYTHONPIC

Warto wspomnieć o następujących metodach klasy `Species`:

- `apply_bc` — aplikuje warunki brzegowe. W symulacjach okresowych jest to przywrócenie cząstek do rejonu symulacji ($x := x\%L$), w symulacjach nieokresowych usunięcie ich.
- `position_push` — przemieszcza cząstki w przestrzeni.
- `velocity_push` — interpoluje pola do pozycji cząstek i aktualizuje ich prędkości na podstawie integratora Borysa.
- `save_particle_values` — zapisuje wartości danych historycznych cząstek.
- `distribute_uniformly` — rozmieszcza cząstki równomiernie na danym przedziale
- `distribute_nonuniformly` — rozmieszcza cząstki według zadanej funkcji rozkładu gęstości

Poza tym `Species` zawiera również rozmaite metody służące inicjalizacji prędkości i położen przy pomocy zaburzeń sinusoidalnych bądź losowych.

4.3 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

Przygotowanie warunków początkowych do danej symulacji polega na utworzeniu nowej klasy dziedziczącej po `Simulation`, która przygotowuje siatkę, cząstki i warunki brzegowe zgodnie z założeniami eksperymentu i wywołuje konstruktor `Simulation`. Należy również przeciążyć metodę `grid_species_initialization`, która przygotowuje warunki początkowe. Domyślna wersja tej metody wykonuje pierwszą, początkową iterację równań ruchu przesuwającą prędkość o $-\Delta t/2$ w tył. To zaś pozwala na zachowanie symplektyczności integratora, co pomaga zachować energię w symulacji. W tym momencie jest też tworzony plik `.hdf5`, do którego zapisywane są dane gromadzone w czasie symulacji.

Aby uruchomić symulację, należy wywołać jedną z metod:

- `run` — podstawowy cykl obliczeń, używany do pomiarów wydajności programu
- `test_run` — obliczenia oraz obróbka danych na potrzeby analizy, głównie stosowana w testach
- `lazy_run` — `test_run` z zapisem do pliku oraz wczytaniem z pliku `.hdf5`, jeżeli początkowe warunki oraz wersja kodu zgadzają się. W przeciwnym razie symulacja zostaje uruchomiona na nowo.

4.4 Pliki pomocnicze

Poza powyższymi program jest podzielony na pliki z implementacjami algorytmów numerycznych, co ułatwia kompilację JIT, pozwala na zwiększenie niezależności testów oraz

4. OPIS I TREŚĆ KODU PYTHONPIC

zwiększa modularność kodu.

- `BoundaryCondition` — implementacja warunków brzegowych
 - `charge_deposition` — depozycja ładunku na siatkę
 - `current_deposition` — depozycja prądu podłużnego i poprzecznego na siatkę
 - `density_profiles` — profile przestrzenne kształtów preplazmy oraz algorytmy rozkładające cząstki według tych rozkładów.
 - `FieldSolver` — spektralne i rotacyjne algorytmy rozwiązywania równań Maxwella na siatce Eulera.
 - `particle_push` — implementacja integratorów równań ruchu typu leapfrog oraz Borysa.
 - `field_interpolation` — interpolacja pól elektromagnetycznych z położenia na siatce do cząstek.
- Katalog `visualization` zawiera wizualizacje danych.
- `animation` — tworzy animacje dla celów analizy danych
 - `time_snapshots` — implementacje wykresów zależnych od czasu, ilustrujących stan symulacji w danej iteracji.
 - `static_plots` — tworzy statyczne wykresy dla celów analizy danych
 - `plotting` — zawiera ustawienia dotyczące analizy danych

Przygotowane konfiguracje istniejących symulacji są zawarte w katalogu `configs`, zaś algorytmiczne testy jednostkowe są zawarte w katalogu `tests`.

5. WYKORZYSTANE OPROGRAMOWANIE I BIBLIOTEKI PYTHONA

5 Wykorzystane oprogramowanie i biblioteki Pythona

Pomijamy tutaj bibliotekę standardową jako wbudowaną w sam język, skupiając się na przydatnych bibliotekach zewnętrznych.

5.1 Numpy

`numpy`[15] to biblioteka umożliwiająca wykonywanie złożonych obliczeń na n-wymiarowych macierzach bądź tablicach, utworzona w celu umożliwienia zastąpienia operacjami wektorowymi iteracji po tablicach, powszechnie stosowanych w metodach numerycznych i będących znanym słabym punktem Pythona.

Pod zewnętrzną powłoką zawiera odwołania do znanych, wypróbowanych i sprawdzonych w nauce modułów LAPACK, BLAS napisanych w szybkich, niskopoziomowych językach C oraz FORTRAN. Jest to *de facto* standard większości obliczeń numerycznych w Pythonie.

Należy zauważyć, że operacje matematyczne w wersji `numpy` zawartej w popularnej dystrybucji Anaconda są automatycznie zrównoleglane tam, gdzie pozwala na to niezależność obliczeń dzięki dołączonym bibliotekom Intel Math Kernel Library.[14]

Numpy jest oprogramowaniem otwartym, udostępnianym na licencji BSD.

5.2 scipy

Kolejną podstawową biblioteką w numerycznym Pythonie jest `scipy`[10], biblioteka zawierająca wydajne gotowe implementacje wielu powszechnych algorytmów numerycznych służących między innymi całkowaniu, optymalizacji funkcji rzeczywistych, uczeniu maszynowemu, algebrze liniowej czy transformatom Fouriera.

5.3 Numba

`numba` to biblioteka służąca do kompilacji just-in-time wysokopoziomowego kodu Pythona do kodu niskopoziomowego przy pierwszym uruchomieniu programu. W wielu przypadkach pozwala na osiągnięcie kodem napisanym w czystym Pythonie wydajności marginalnie niższej bądź nawet równej do analogicznego programu w C bądź Fortranie. [11]

Jednocześnie należy zaznaczyć prostotę jej użycia. W wielu przypadkach wystarczy dodać do funkcji dekorator `@numba.jit`:

Listing 1: Przykład zastosowania kompilacji just-in-time z biblioteki Numba wykorzystujący metody pomiaru czasu środowiska Jupyter.

```
1 import numba
2 import numpy
3
4 def sqrt(x):
5     return x**0.5
```

5. WYKORZYSTANE OPROGRAMOWANIE I BIBLIOTEKI PYTHONA

```
6
7 sqrt_jitted = numba.njit(sqrt)
8
9 @numba.njit
10 def sqrt_jitted_as_decorator(x):
11     return x**0.5
12
13 x = numpy.arange(100000)
14
15 %%timeit sqrt(x)
16 # 49.4 ms +- 8.77 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
17
18 %%timeit sqrt_jitted(x)
19 # 4.66 ms +- 70.3 us per loop (mean +- std. dev. of 7 runs, 1 loop each)
20
21 %%timeit sqrt_jitted_as_decorator(x)
22 # 4.38 ms +- 50.8 us per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

Istniejącym od niedawna kodem symulacyjnym implementującym tą metodę jest FBPIC[?].

5.4 HDF5

HDF5 jest wysokowydajnym formatem plików służącym przechowywaniu danych liczbowych w drzewiastej, skompresowanej strukturze danych, razem z równoległym, wielowątkowym zapisem tych danych. W Pythonie implementuje go biblioteka h5py[6].

W bieżącej pracy wykorzystuje się go do przechowywania danych liczbowych dotyczących przebiegu symulacji, pozwalających na ich dalsze przetwarzanie i analizę poprzez wizualizację.

5.5 matplotlib

Do wizualizacji danych z symulacji (oraz tworzenia schematów w sekcji teoretycznej niniejszej pracy) użyto własnoręcznie napisanych skryptów w uniwersalnej bibliotece graficznej matplotlib[9]. matplotlib zapewnia wsparcie zarówno dla grafik statycznych w różnych układach współrzędnych (w tym 3D), jak również dla dynamicznie generowanych animacji przedstawiających przebiegi czasowe symulacji.

5.6 pytest

Przy pracy nad kodem użyto frameworku testowego `pytest` [1]. Tworzenie testów jest trywialne:

Listing 2: Podstawowy przykład testu w pliku `func.py`

```
1 def f(x):
```

5. WYKORZYSTANE OPROGRAMOWANIE I BIBLIOTEKI PYTHONA

```
2     return 3*x
3
4
5 def test_passing():
6     assert f(4) == 12
7
8
9 def test_failing():
10    assert f(4) == 13
```

Uruchamianie zaś:

```
>>> pytest func.py
=== FAILURES ===
___failing ___
```

```
def test_failing():
>     assert f(4) == 13
E     assert 12 == 13
E     + where 12 = f(4)
```

```
func.py:10: AssertionError
=== 1 failed , 1 passed in 0.10 seconds ===
```

Należy zaznaczyć, że w symulacjach numerycznych, gdzie błędne działanie programu nie objawia się zazwyczaj formalnym błędem, 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 symulacyjny powinien je wykorzystywać, najlepiej do weryfikacji każdej części algorytmu z osobna.

Dobrym przykładem skutecznego testu jednostkowego jest porównanie energii kinetycznej elektronu o znanej prędkości z wartością tablicową, zawarte jako test w pliku `pythonpic/tests/test_species.py`.

`py.test` jest oprogramowaniem otwartym, dostępnym na licencji MIT.

5.7 Travis CI

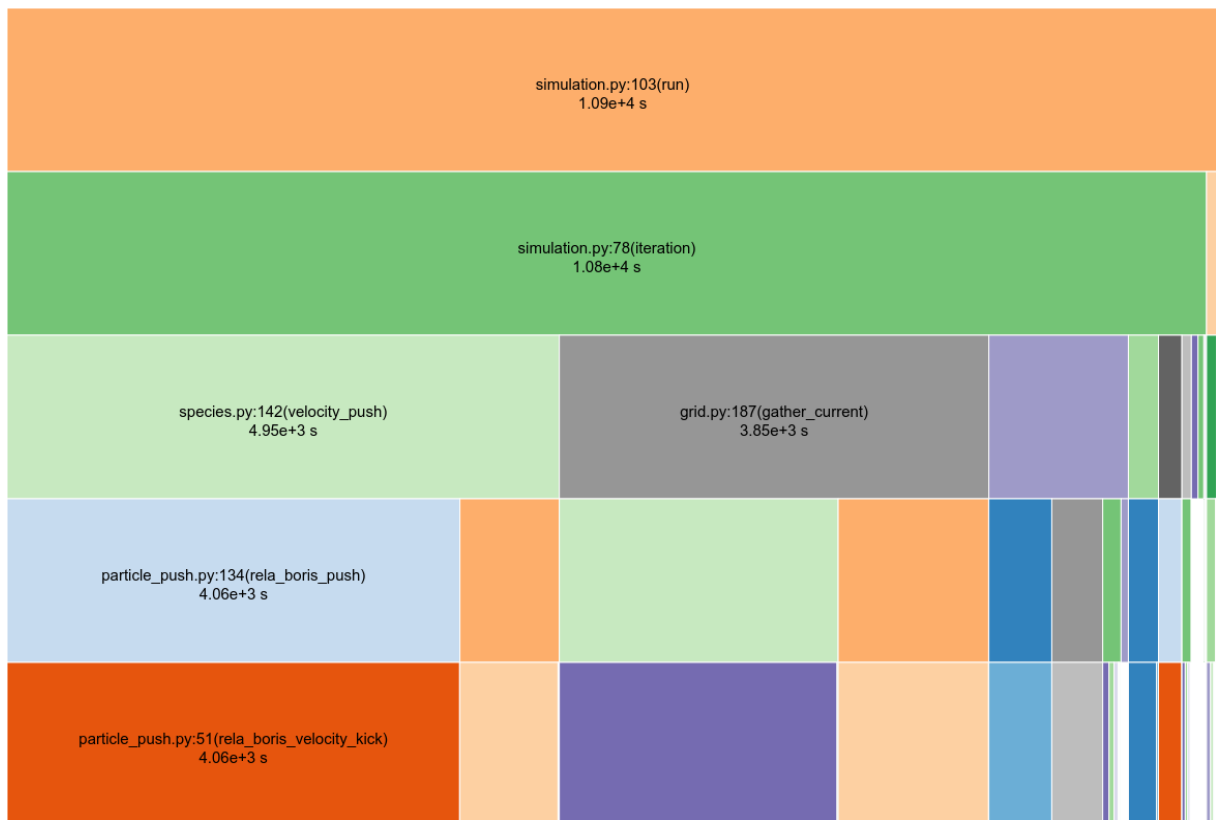
Nieocenionym narzędziem w pracy nad kodem był system ciągłej integracji (*continuous integration*) Travis CI [2] dostępny za darmo dla projektów open-source. Travis pobiera aktualne wersje kodu przy każdej aktualizacji wersji dostępnej na serwerze GitHub i uruchamia testy, zwracając komunikat o ewentualnym niepowodzeniu i pozwalając na jednoczesne uruchamianie bieżących, intensywnych symulacji przy jednoczesnym weryfikowaniu w chmurze poprawności

5. WYKORZYSTANE OPROGRAMOWANIE I BIBLIOTEKI PYTHONA

działania lżejszych, acz wciąż intensywnych symulacji testowych i testów algorytmicznych.

5.8 snakeviz

W optymalizacji przydatny okazał się program `snakeviz`[7] dostępny na GitHubie i pozwalający na wizualizację wyników z profilowania symulacji. Pozwala w wygodny sposób zbadać, które fragmenty kodu najbardziej spowalniają symulację, które są najlepszymi kandydatami do optymalizacji, oraz jak skuteczne (bądź nieskuteczne) okazują się próby polepszenia ich wydajności. Działanie programu ilustruje rysunek 6.



Rysunek 6: Wizualizacja szybkości działania poszczególnych fragmentów kodu wygenerowana programem `snakeviz`.

5.9 cProfile

Uruchamianie kodu w celu zmierzenia wydajności całkowitej polegało na uruchomieniu skryptu `make benchmark`, który:

1. Czyści zawartość folderu `data_analysis`, aby przeprowadzić obliczenia od początku bez wczytywania poprzednich danych
2. Uruchamia środowisko Anaconda zawierające bardziej zoptymalizowane od zwyczajnych wersje biblioteki Numpy

5. WYKORZYSTANE OPROGRAMOWANIE I BIBLIOTEKI PYTHONA

3. Uruchamia skrypt `fulllaser.py` w trybie `cProfile` i zapisuje dane do pliku `benchmark.prof`

```
python -m cProfile -o benchmark.prof fulllaser.py
```

Następnie zapisane dane są wizualizowane programem `snakeviz`.

Za wskaźnik efektów optymalizacji przyjęto całkowity czas trwania symulacji oraz ułamek tego czasu spędzony w metodzie `iteration` klasy `Simulation`.

5.10 IPython, %timeit

Do optymalizacji niewielkich fragmentów kodu wykorzystano środowisko Jupyter (dawniej IPython) Notebook[?]. Użycie “magicznych” komend `%timeit` pozwala na automatyczne profilowanie wydajności niewielkich fragmentów kodu. Przykładem zastosowania tej komendy jest listing 5.3.

6 Część weryfikacyjna

Niniejsza analiza przeprowadzona została na “finalnej” w chwili pisania niniejszej pracy wersji programu. W repozytorium Git na GitHubie jest to commit “placeholder”

uzupełnić commita

identyfikowany również jako wersja 1.0.

6.1 Przypadki testowe

Kod przetestowano w dwojaki sposób. Pierwszym z nich są testy jednostkowe. Automatyczne testy jednostkowe uruchamiane po każdej wymiernej zmianie kodu pozwalają kontrolować działanie programu znacznie ułatwiają zapobieganie błędom.

Poszczególne algorytmy podlegały testom przy użyciu ogólnodostępnego pakietu `pytest` 5.6 i w większości były uruchamiane na platformie TravisCI.

6.1.1 Testy algorytmiczne

Testy algorytmiczne 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.

- Gather — `test_current_deposition`, `test_charge_deposition`
 - Depozycja prądu lub ładunku z pojedynczej cząstki na niewielką siatkę i porównanie z wynikiem analitycznym[?]
 - Depozycja prądu lub ładunku z dwóch pojedynczych cząstek na niewielką siatkę i porównanie z sumą prądów lub ładunków dla obu pojedynczych cząstek
 - Depozycja prądu z dużej ilości równomiernie rozłożonych cząstek — weryfikacja średniej wartości prądu
 - Depozycja prądu lub ładunku z cząstek blisko granic i porównanie z zamierzonymi warunkami brzegowymi
 - Depozycja prądu dla cząstki poruszającej się powyżej prędkości światła — procedura depozycji zwraca błąd
- Solve — `test_FieldSolver`, `test_LongitudinalSolver`
 - Test solvera spektralnego poprzez rozwiązanie równania Poissona dla sinusoidalnej, deltowej i rosnącej liniowo dystrybucji ładunku. Weryfikacja dokładności rozwiązania i energii pola
 - Test solvera rotacyjnego poprzez porównanie z rezultatami analitycznymi dla pojedynczych iteracji

6. WERYFIKACJA

- Scatter — test_field_interpolation — porównanie interpolowanego pola z przypadkiem analitycznym dla kilku prostych rozkładów pola
- Push — test_pusher
 - Ruch w jednorodnym polu elektrycznym wzdłuż osi układu, w wersji nierelatywistycznej i relatywistycznej. Prędkość cząstki rośnie jednostajnie w reżimie nierelatywistycznym.
 - Relatywistyczny i nierelatywistyczny ruch kołowy z zachowaniem energii w jednorodnym polu magnetycznym w kierunku osi \hat{z} .
 - Relatywistyczny ruch w potencjale oscylatora harmonicznego. Ruch cząstki jest porównany z trajektorią obliczoną analitycznie w przypadku relatywistycznym.
 - Zachowanie energii ruchu cząstki bez pola elektrycznego przy wysoce relatywistycznej prędkości, w kierunku ruchu oraz z prędkościami w trzech wymiarach.
 - Test dryfu $\vec{E} \times \vec{B}$ — cząstka w skrzyżowanych polach elektrycznym i magnetycznym uzyskuje stabilną prędkość E/B .
 - Test warunków brzegowych dla symulacji okresowych — zachowanie liczby cząstek przechodzących przez granicę obszaru symulacji.
 - Test warunków brzegowych dla symulacji okresowych — spadek liczby cząstek przechodzących przez granicę obszaru symulacji do zera.

6.2 Testy symulacyjne — przypadki elektrostatyczne

Testy symulacyjne polegały na uruchomieniu niewielkiej symulacji testowej z różnymi warunkami brzegowymi i ilościowym, automatycznym bądź jakościowym, wizualnym zweryfikowaniu dynamiki zjawisk w niej zachodzących.

Zastosowano kod do symulacji kilku znanych problemów w fizyce plazmy:

6.2.1 oscylacje zimnej plazmy

Jest to efektywnie elektrostatyczna fala stojąca. Symulacja zaczyna z ujemnymi cząstkami o zerowej prędkości początkowej, rozłożonymi w okresowym pudełku symulacyjnym równomiernie z nałożonym na nie sinusoidalnym zaburzeniem:

$$x = x_0 + x_1 \tag{41}$$

$$x_0 = L * n/N \tag{42}$$

$$x_1 = A \sin(kx_0) = A \sin(2\pi n x_0/L) \tag{43}$$

Określenie “zimna plazma” bierze się z nietermalnego, deltowego rozkładu prędkości cząstek — jest to faktycznie strumień cząstek o stałej (w tym szczególnym przypadku zerowej) prędkości.

Sumaryczny ładunek w obszarze symulacji jest ustawiony na zero w pierwszym kroku algorytmu rozwiązywania pola elektrycznego poprzez wyzerowanie zerowej składowej fourierowskiej gęstości ładunku, co jest jednoznaczne z przyjęciem nieskończenie masywnych i nieruchomych jonów dodatnich dokładnie neutralizujących gęstość ładunku elektronów.

Sytuacja ta pozwala na obserwację oscylacji cząstek wokół ich stabilnych położenia równowagi. W przestrzeni fazowej x, V_x cząstki zataczają efektywnie elipsy, co pozwala wnioskować że ruch ten jest z dobrym przybliżeniem harmoniczny. Oczywiście, nie jest to do końca oscylacja harmoniczna z powodu odchylenia pola interpolowanego z Eulerowskiej siatki od generowanego faktycznym potencjałem $\sim x^2$.

Jest to spełnione jedynie dla niewielkich odchylenia; dla $A \rightarrow dx$ obserwuje się nieliniowy reżim oscylacji,

Symulacja ta jest wykorzystywana do weryfikacji podstawowych warunków, jakie powinna spełniać symulacja elektrostatyczna — na przykład długofalowe zachowanie energii, liczby cząstek (w układzie okresowym cząstki nie powinny znikać).

Liniowy reżim obserwacji jest zaznaczony na rysunku 7, zaś nieliniowy ??.

6.2.2 niestabilność dwóch strumieni

W tym przypadku symulacja również zawiera zimną plazmę, lecz tym razem są to dwa strumienie ujemnych cząstek o stałych, przeciwnych sobie prędkościach v_0 oraz $-v_0$.

Dla niewielkich prędkości początkowych strumieni obserwuje się liniowy reżim oscylacji cząstek — oba strumienie pozostają stabilne. Obserwuje się niewielkie oscylacje oraz grupowanie się cząstek w rejony koherentnej większej gęstości wewnątrz strumienia (opisany przez Birdsalla i Langdona *bunching*).

Dla dużych prędkości obserwuje się nieliniowe zachowanie cząstek w przestrzeni fazowej. Oscylacje prędkości cząstek przybierają rząd wielkości porównywalny z początkową różnicą prędkości strumieni. Strumienie zaczynają się mieszać ze sobą nawzajem, zaś cały układ się termalizuje. Energia kinetyczna uporządkowanego ruchu strumieni zamienia się w energię potencjalną pola równowagowego oraz termalną energię kinetyczną, co sprawia, że średnia prędkość obu strumieni ulega zmniejszeniu.[3]

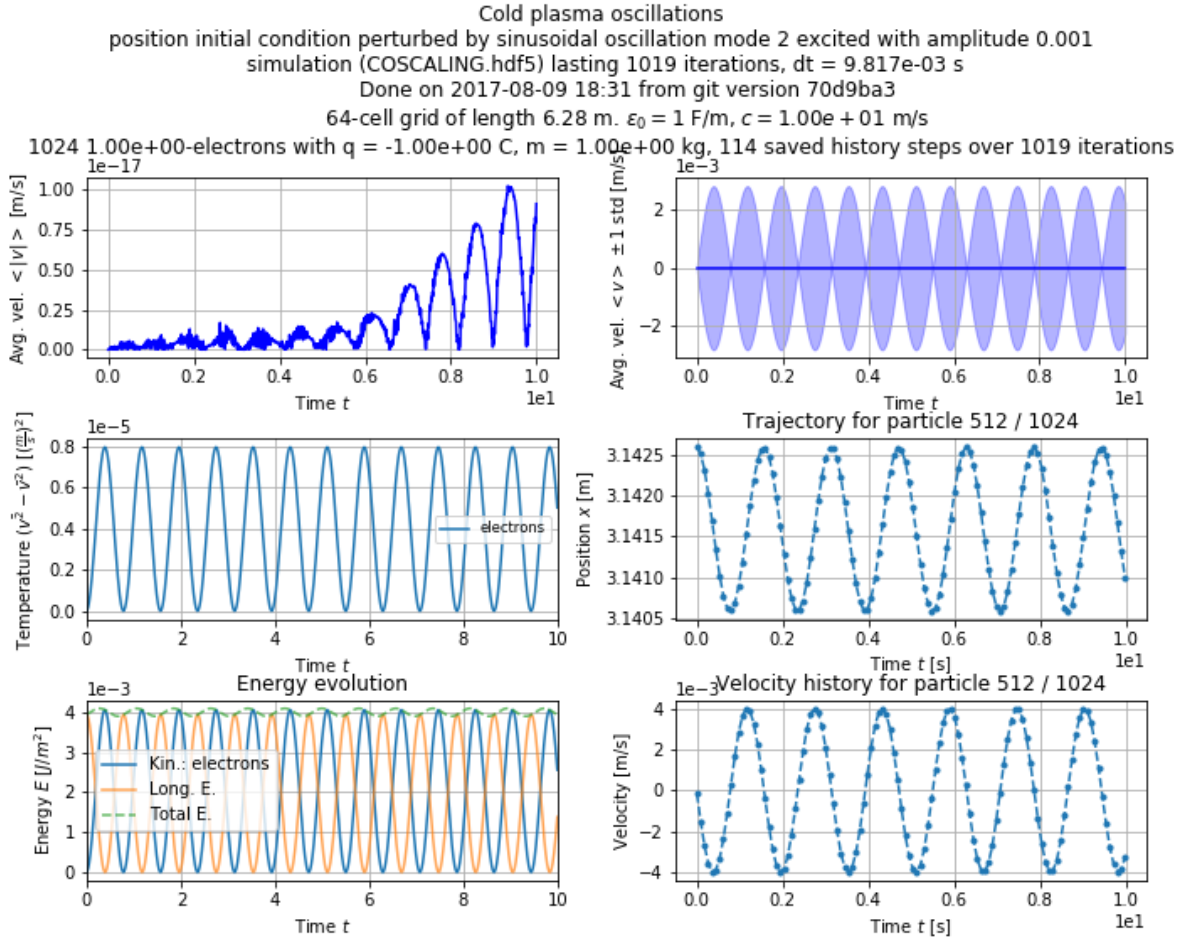
To oraz stabilność symulacji o warunkach początkowych w obu reżimach jest obiektem automatycznych testów sprawdzających poprawność symulacji.

6.3 Testowe symulacje elektromagnetyczne — propagacja fali

W tym przypadku symulacja nie zawiera plazmy, a badana jest jedynie propagacja fali elektromagnetycznej w obszarze symulacji dla różnych charakterystyk czasowych.

Testy jednostkowe obejmują zachowanie energii weryfikowane poprzez zgodność energii pola elektromagnetycznego w symulacji ze strumieniem Poyntinga generowanym przez warunek brzegowy.

6. WERYFIKACJA



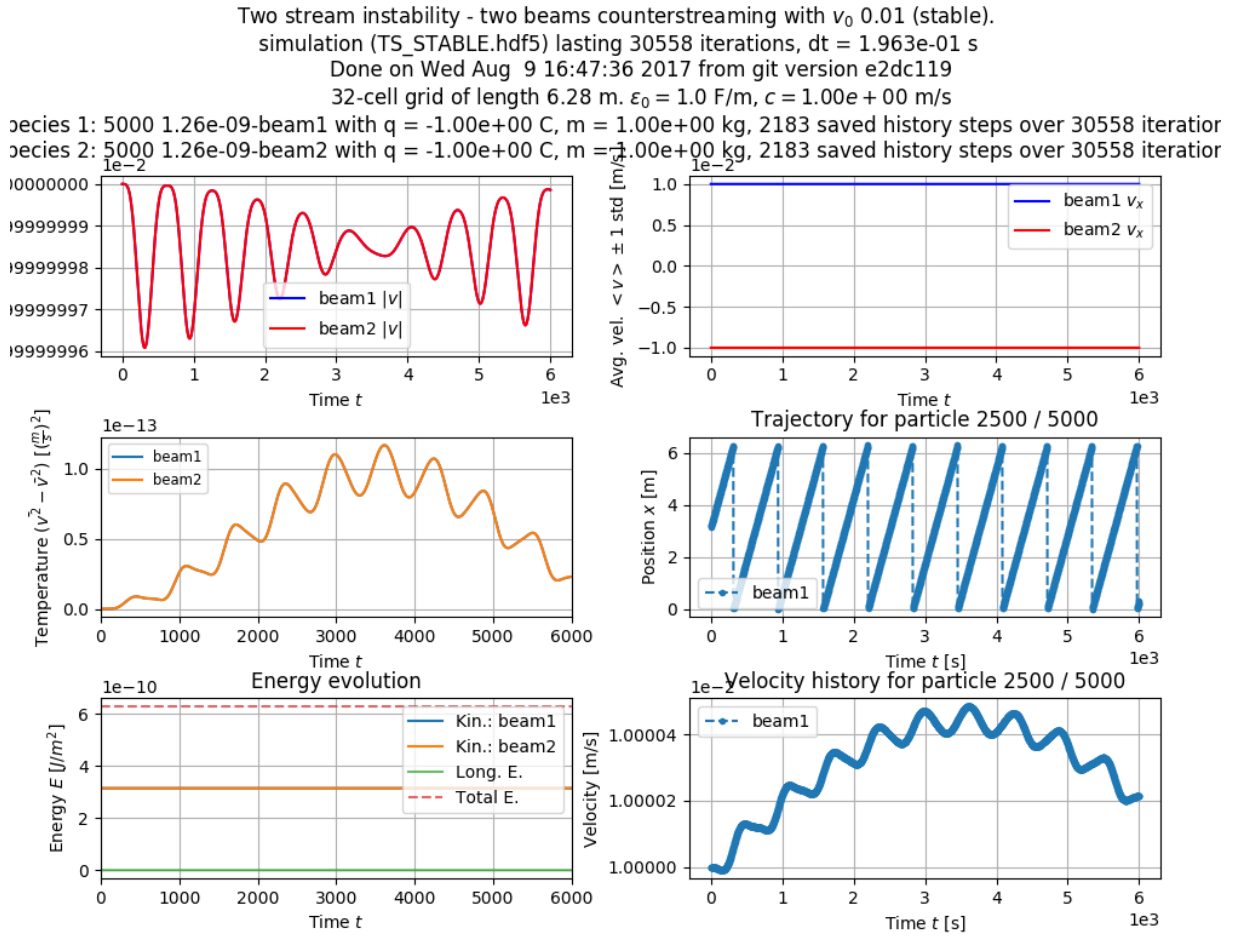
Rysunek 7: Oscylacje zimnej plazmy w reżimie liniowym. Cząstki wykonują prawie harmoniczne oscylacje wokół swoich położeń równowagi, zaś energia w układzie jest wymieniana między energią kinetyczną cząstek a energią pola elektrostatycznego. Jako wyjście programu, wykresy z *PythonPIC* w bieżącym rozdziale zamieszczane są w języku angielskim.

6.4 Symulacja elektromagnetyczna — oddziaływanie wiązki laserowej z tarczą wodorową

Jest to sytuacja, która od początku rozwoju programu miała być głównym celem istnienia kodu. Szerzej została opisana w 2.3.

Jako warunki początkowe przyjęto plazmę rozbitą na dwie części — *preplazmę* o narastającej funkcji rozkładu gęstości oraz plazmę właściwą o stałej gęstości. Funkcja gęstości jest generowana automatycznie poprzez metodę opisaną w [3] i jest normalizowana do danego poziomu maksymalnej gęstości w obszarze plazmy właściwej przy zadanej liczbie makrocząstek.

Początkowe prędkości cząstek przyjęto jako zerowe, zaś początkowe położenia elektronów są identyczne z położeniami początkowymi protonów. Jest to prosta w inicjalizacji, inherentnie równowagowa sytuacja (gwarantująca nie tylko kwaziobojętność elektryczną plazmy, ale również



Rysunek 8: Stabilny reżim dla układu dwóch strumieni. Widoczne są jedynie cykliczny ruch jednostajny w obszarze symulacji i niewielkie zmiany prędkości cząstki.

całkowitą obojętność początkowego układu) do czasu przybycia lasera. Należy również zauważyć, że przy zakładanych energiach lasera jakakolwiek termalna dynamika układu przed interakcją zostanie szybko wymazana. Jest to więc akceptowalne przybliżenie.

Za intensywność lasera przyjęto wielkości 10^{21} , 10^{22} , 10^{23} W/m^2 , zaś za jego długość fali $1.064 \mu\text{m}$ (jest to laser Nd:YAG). Przeprowadzono badania w polaryzacjach liniowej (pole elektryczne w kierunku osi \hat{y} — nie zaobserwowano znaczących różnic w symulacji, w której pole elektryczne skierowano w kierunku osi \hat{z}) oraz kołowej wiązki laserowej.

Długość obszaru symulacji to około $10.6 \mu\text{m}$, podzielone na 1378 komórek siatki.

Dla mocy lasera do 10^{22} J/m^2 przy polaryzacji liniowej laser zostaje w większości wytłumiony i odbity przez plazmę. Obserwuje się generację niewielkich oscylacji wysokiej częstotliwości emitowanych w kierunku $+\hat{x}$. Widać to zwłaszcza w przypadku polaryzacji kołowej.

Dla wszystkich przypadków od 10^{22} J/m^2 obserwuje się spłaszczenie preplazmy w kierunku plazmy głównej.

6. WERYFIKACJA

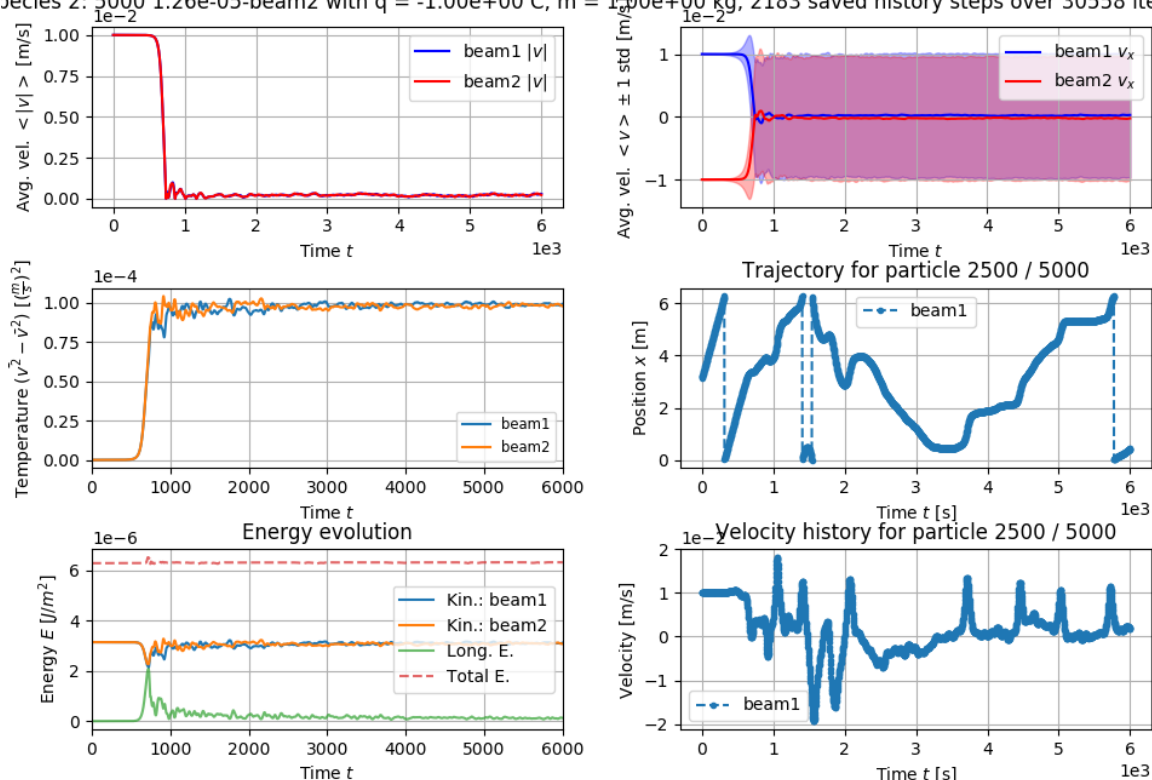
Two stream instability - two beams counterstreaming with v_0 0.01 (unstable).

simulation (TS_UNSTABLE.hdf5) lasting 30558 iterations, $dt = 1.963e-01$ s

Done on Wed Aug 9 16:47:36 2017 from git version e2dc119

32-cell grid of length 6.28 m. $\epsilon_0 = 1.0$ F/m, $c = 1.00e+00$ m/s

Species 1: 5000 $1.26e-05$ -beam1 with $q = -1.00e+00$ C, $m = 1.00e+00$ kg, 2183 saved history steps over 30558 iterations
Species 2: 5000 $1.26e-05$ -beam2 with $q = -1.00e+00$ C, $m = 1.00e+00$ kg, 2183 saved history steps over 30558 iterations



Rysunek 9: Niestabilny reżim dla układu dwóch strumieni. Trajektoria cząstki staje się chaotyczna.

Dla mocy lasera 10^{23} J/m^2 przy polaryzacji liniowej obserwuje się znaczące przejście wiązki laserowej przez warstwę plazmy oraz silne zniszczenie początkowego rozkładu przestrzennego plazmy, z przejściem do termalizacji. Gęstości obu rodzajów cząstek uzyskują rozkład normalny.

Wizualizacje symulacji tarczy wodorowej przedstawiono na rysunkach 10–15.

tu jeszcze kombinuję z najsensowniejszym wrzuceniem snapshotów dobrze oddających dynamikę układu, najchętniej bym wkleił animację ale w pracy do druku się jeszcze nie da

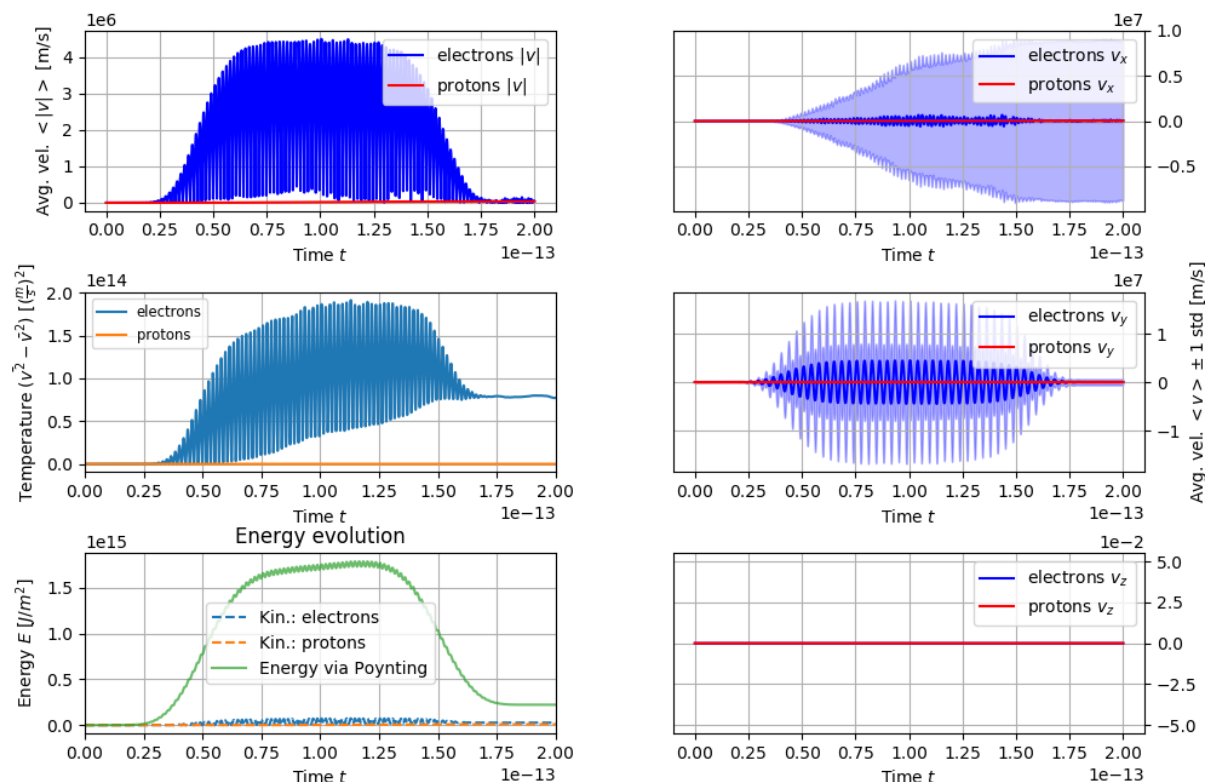
6.5 Parametry uruchomienia

Profilowanie przeprowadzono na komputerze o parametrach:

- Intel Core 4 Quad 6600 3.0GHz
- 4GB RAM
- SSD firmy Crucial
- Arch Linux

6. WERYFIKACJA

Hydrogen shield-laser interaction simulation (75000_1378_run_21_Ey.hdf5) lasting 7755 iterations, $dt = 2.579e-17$ s
 Done on Sat Jul 29 13:10:04 2017 from git version 40d9c15
 1378-cell grid of length 0.00 m. $\epsilon_0 = 8.854187817e-12$ F/m, $c = 3.00e+08$ m/s
 species 1: 75000 $9.85e+24$ -electrons with $q = -1.60e-19$ C, $m = 9.11e-31$ kg, 647 saved history steps over 7755 iterations
 species 2: 75000 $9.85e+24$ -protons with $q = 1.60e-19$ C, $m = 1.67e-27$ kg, 647 saved history steps over 7755 iterations



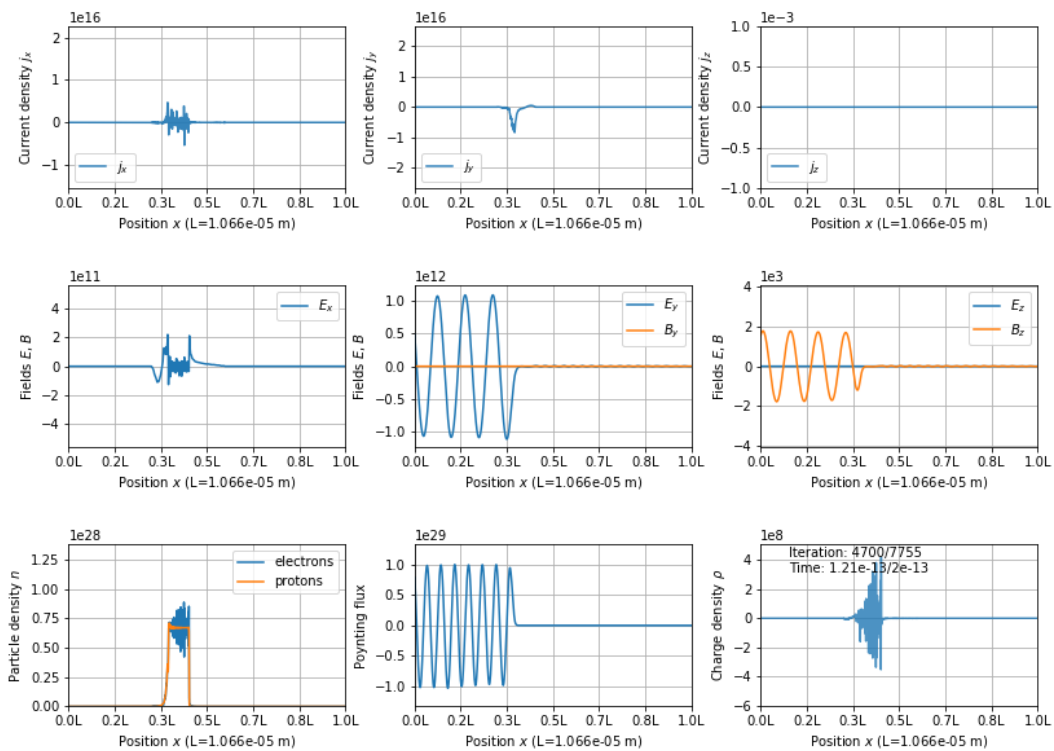
Rysunek 10: Intensywność wiązki laserowej 10^{21} J/m^2 , polaryzacja liniowa.

Wykorzystując następujące wersje oprogramowania i bibliotek obliczeniowych:

- Python 3.6.1
- Numpy 1.13.1
- Numba 0.34.0
- h5py 2.7.0
- SciPy 0.19.1

6. WERYFIKACJA

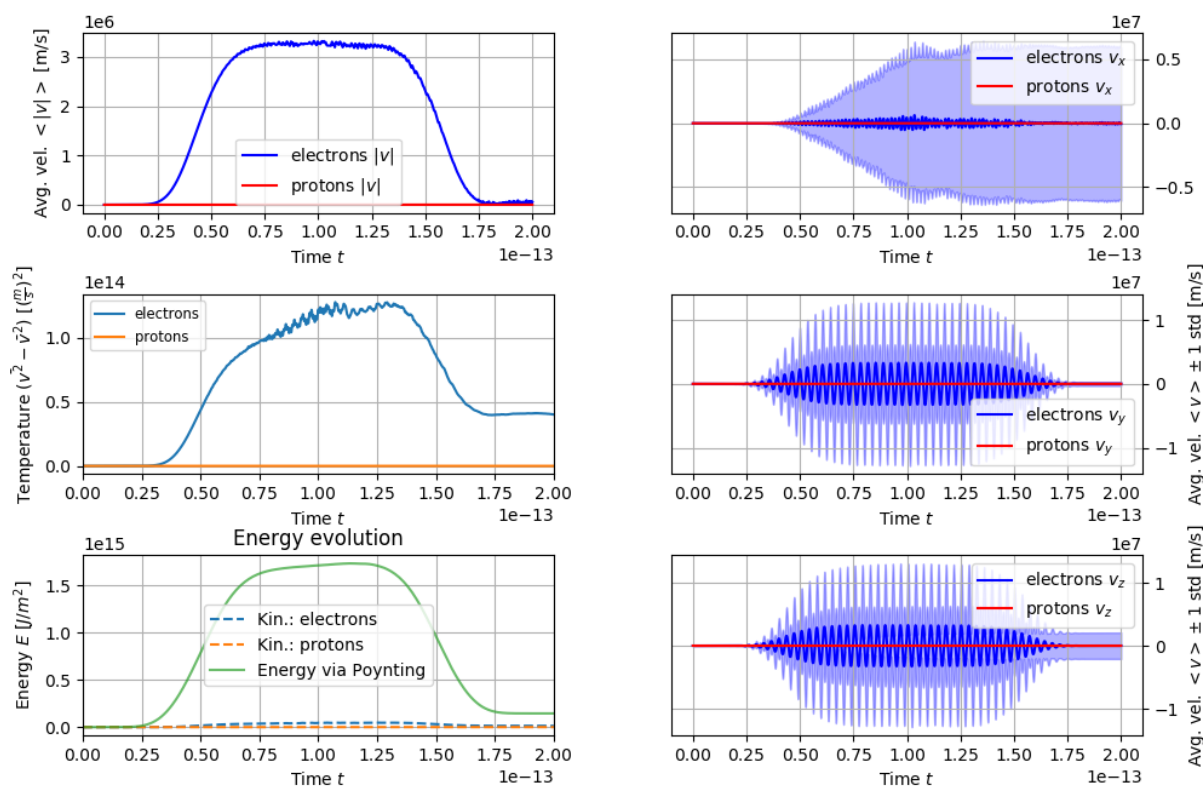
Hydrogen shield-laser interaction simulation (75000_1378_run_21_Ey.hdf5) lasting 7755 iterations, $dt = 2.579e-17$ s
 Done on Sat Jul 29 13:10:04 2017 from git version 40d9c15
 1378-cell grid of length 0.00 m. $\epsilon_0 = 8.854187817e-12$ F/m, $c = 3.00e+08$ m/s
 75000 $9.85e+24$ -electrons with $q = -1.60e-19$ C, $m = 9.11e-31$ kg, 647 saved history steps over 7755 iterations
 75000 $9.85e+24$ -protons with $q = 1.60e-19$ C, $m = 1.67e-27$ kg, 647 saved history steps over 7755 iterations



Rysunek 11: Intensywność wiązki laserowej $10^{21} J/m^2$, polaryzacja liniowa. Rzut na iterację 4700/7755.

6. WERYFIKACJA

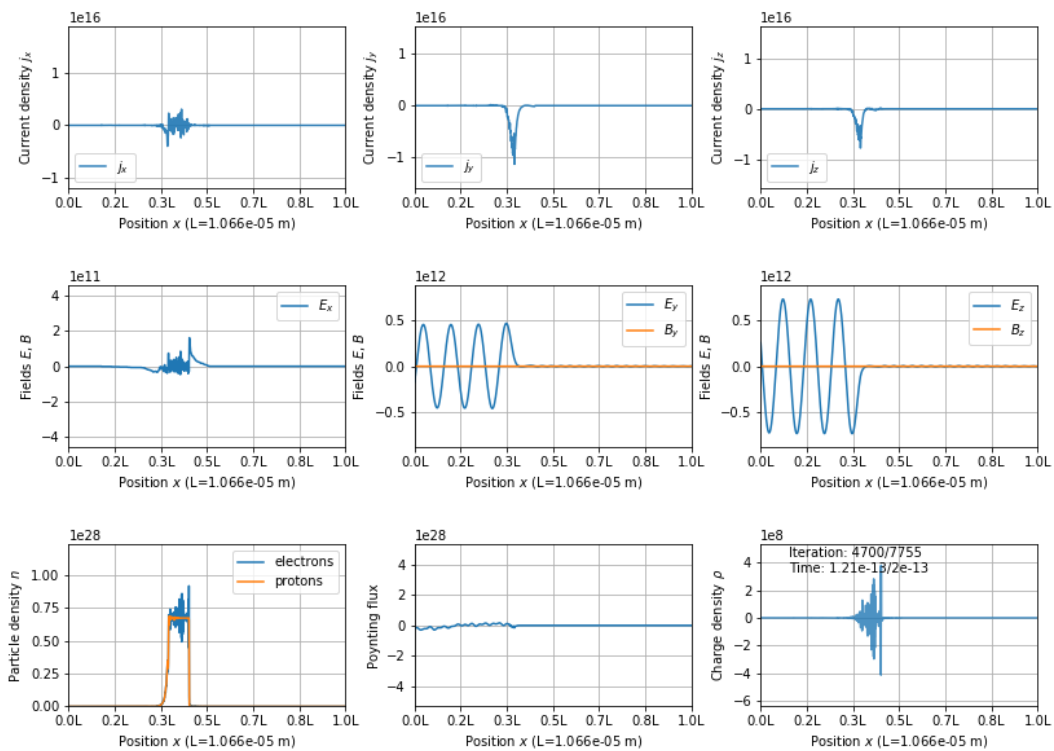
hydrogen shield-laser interaction simulation (75000_1378_run_21_Circular.hdf5) lasting 7755 iterations, $dt = 2.579e-17$
 Done on Sat Jul 29 13:10:04 2017 from git version 40d9c15
 1378-cell grid of length 0.00 m. $\epsilon_0 = 8.854187817e-12$ F/m, $c = 3.00e+08$ m/s
 Species 1: 75000 $9.85e+24$ -electrons with $q = -1.60e-19$ C, $m = 9.11e-31$ kg, 647 saved history steps over 7755 iterations
 Species 2: 75000 $9.85e+24$ -protons with $q = 1.60e-19$ C, $m = 1.67e-27$ kg, 647 saved history steps over 7755 iterations



Rysunek 12: Intensywność wiązki laserowej $10^{21} J/m^2$, polaryzacja kołowa.

6. WERYFIKACJA

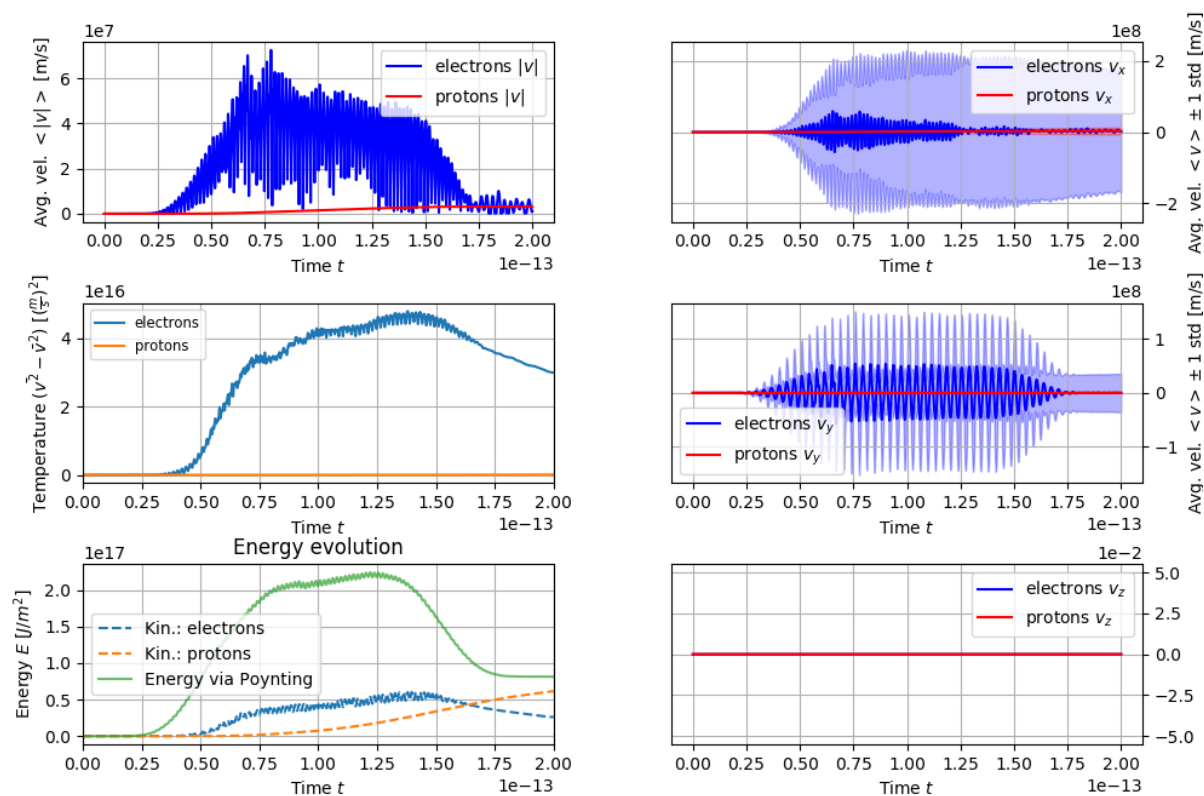
Hydrogen shield-laser interaction simulation (75000_1378_run_21_Circular.hdf5) lasting 7755 iterations, $dt = 2.579e-17$ s
 Done on Sat Jul 29 13:10:04 2017 from git version 40d9c15
 1378-cell grid of length 0.00 m. $\epsilon_0 = 8.854187817e-12$ F/m, $c = 3.00e+08$ m/s
 75000 $9.85e+24$ -electrons with $q = -1.60e-19$ C, $m = 9.11e-31$ kg, 647 saved history steps over 7755 iterations
 75000 $9.85e+24$ -protons with $q = 1.60e-19$ C, $m = 1.67e-27$ kg, 647 saved history steps over 7755 iterations



Rysunek 13: Intensywność wiązki laserowej $10^{21} J/m^2$, polaryzacja kołowa. Rzut na iterację 4700/7755.

6. WERYFIKACJA

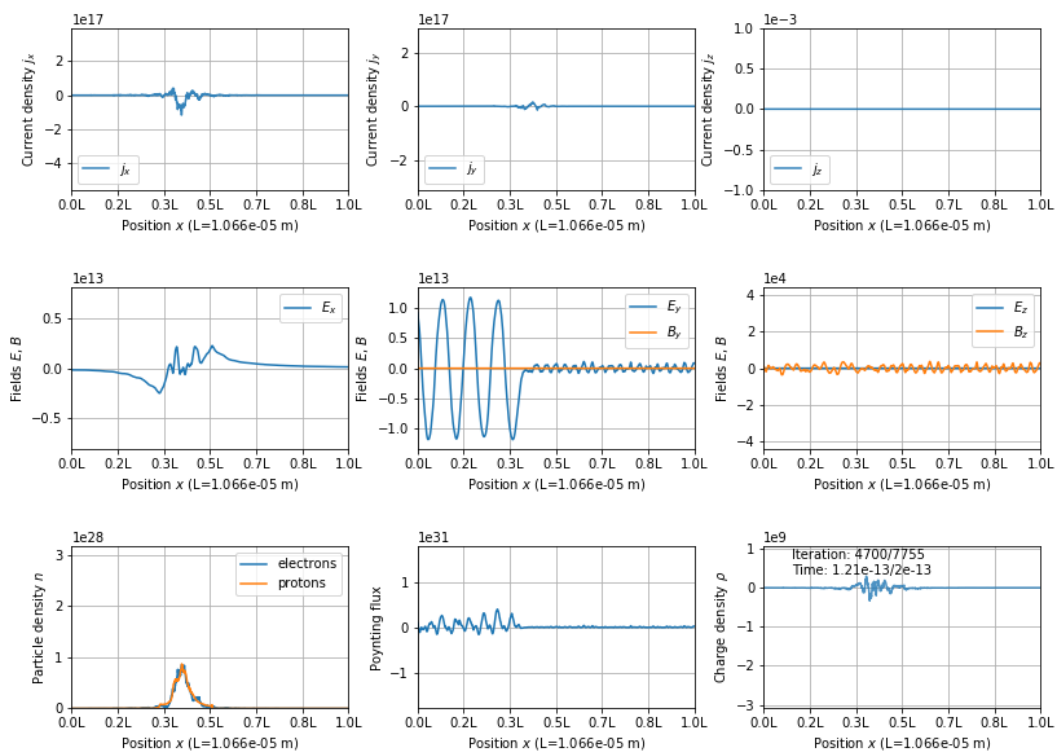
Hydrogen shield-laser interaction simulation (75000_1378_run_23_Ey.hdf5) lasting 7755 iterations, $dt = 2.579e-17$ s
 Done on Sat Jul 29 13:10:04 2017 from git version 40d9c15
 1378-cell grid of length 0.00 m. $\epsilon_0 = 8.854187817e-12$ F/m, $c = 3.00e+08$ m/s
 Species 1: 75000 $9.85e+24$ -electrons with $q = -1.60e-19$ C, $m = 9.11e-31$ kg, 647 saved history steps over 7755 iterations
 Species 2: 75000 $9.85e+24$ -protons with $q = 1.60e-19$ C, $m = 1.67e-27$ kg, 647 saved history steps over 7755 iterations



Rysunek 14: Intensywność wiązki laserowej $10^{23} J/m^2$, polaryzacja liniowa.

6. WERYFIKACJA

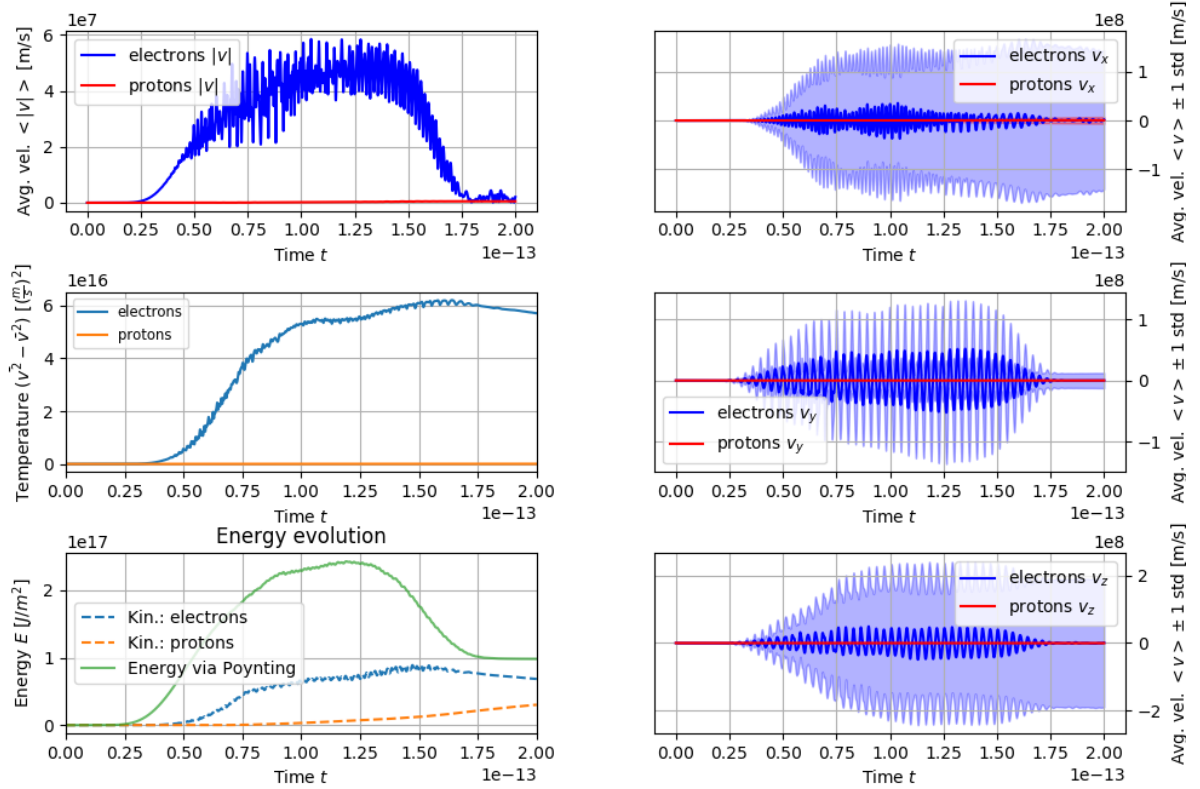
Hydrogen shield-laser interaction simulation (75000_1378_run_23_Ey.hdf5) lasting 7755 iterations, dt = 2.579e-17 s
 Done on Sat Jul 29 13:10:04 2017 from git version 40d9c15
 1378-cell grid of length 0.00 m. $\epsilon_0 = 8.854187817e-12$ F/m, $c = 3.00e+08$ m/s
 75000 9.85e+24-electrons with $q = -1.60e-19$ C, $m = 9.11e-31$ kg, 647 saved history steps over 7755 iterations
 75000 9.85e+24-protons with $q = 1.60e-19$ C, $m = 1.67e-27$ kg, 647 saved history steps over 7755 iterations



Rysunek 15: Intensywność wiązki laserowej $10^{23} J/m^2$, polaryzacja liniowa. Rzut na iterację 4700/7755.

6. WERYFIKACJA

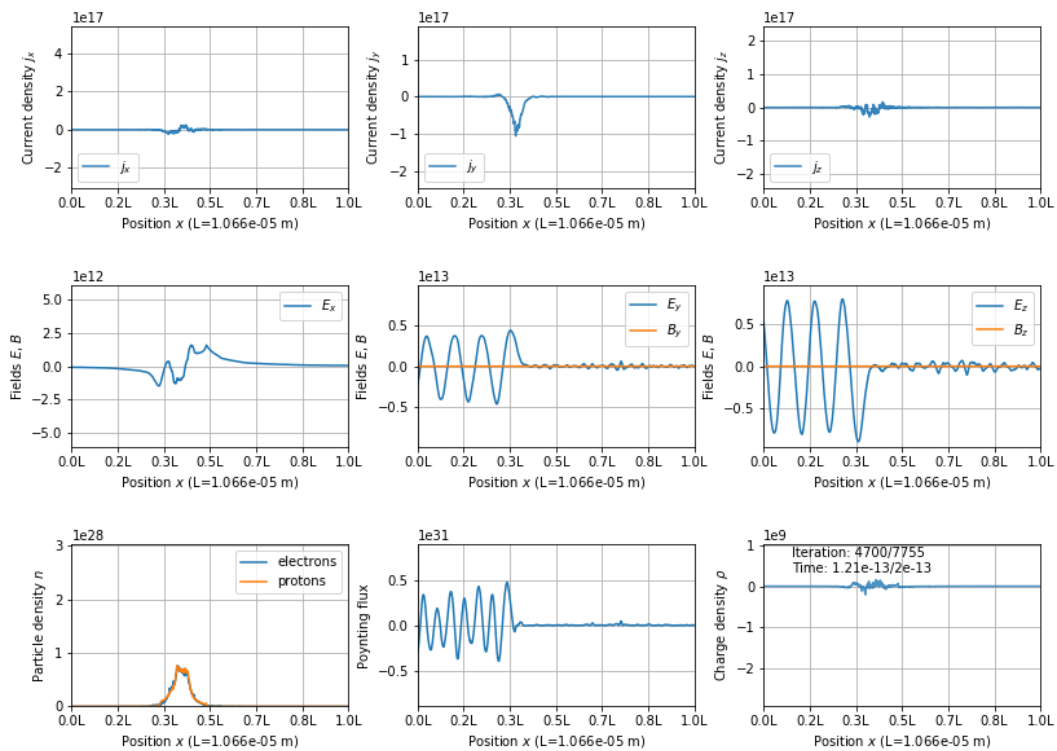
hydrogen shield-laser interaction simulation (75000_1378_run_23_Circular.hdf5) lasting 7755 iterations, $dt = 2.579e-17$
 Done on Sat Jul 29 13:10:04 2017 from git version 40d9c15
 1378-cell grid of length 0.00 m. $\epsilon_0 = 8.854187817e-12$ F/m, $c = 3.00e+08$ m/s
 species 1: 75000 $9.85e+24$ -electrons with $q = -1.60e-19$ C, $m = 9.11e-31$ kg, 647 saved history steps over 7755 iterations
 species 2: 75000 $9.85e+24$ -protons with $q = 1.60e-19$ C, $m = 1.67e-27$ kg, 647 saved history steps over 7755 iterations



Rysunek 16: Intensywność wiązki laserowej $10^{23} J/m^2$, polaryzacja kołowa.

6. WERYFIKACJA

Hydrogen shield-laser interaction simulation (75000_1378_run_23_Circular.hdf5) lasting 7755 iterations, $dt = 2.579e-17$ s
 Done on Sat Jul 29 13:10:04 2017 from git version 40d9c15
 1378-cell grid of length 0.00 m. $\epsilon_0 = 8.854187817e-12$ F/m, $c = 3.00e+08$ m/s
 75000 $9.85e+24$ -electrons with $q = -1.60e-19$ C, $m = 9.11e-31$ kg, 647 saved history steps over 7755 iterations
 75000 $9.85e+24$ -protons with $q = 1.60e-19$ C, $m = 1.67e-27$ kg, 647 saved history steps over 7755 iterations



Rysunek 17: Intensywność wiązki laserowej $10^{23} J/m^2$, polaryzacja kołowa. Rzut na iterację 4700/7755.

7 Wnioski

W niniejszej pracy opisano proces tworzenia kodu symulacyjnego implementującego algorytm particle-in-cell. W rozdziale 2 wyjaśniono zasadność modelowania plazmy symulacjami komputerowymi oraz zasadność przyjętego przybliżenia modelowania plazmy dyskretnymi cząstkami. Poszczególne elementy realizacji algorytmu particle-in-cell w bieżącym kodzie wyjaśniono w rozdziale 3.

W rozdziale 4 opisano ekosystem Pythona, jego potencjał dla nauki i symulacji numerycznej, a także strukturę i hierarchię samego kodu, który jest przedmiotem bieżącej pracy.

Z dobrą zgodnością z zakładanym modelem udało się wykonać symulację zachowania plazmy pobudzonej wysokoenergetycznym impulsem laserowym.

7.1 Potencjalne kierunki dalszej pracy

Program w obecnej postaci działa w jednym wymiarze, w reżimie relatywistycznym i jako taki może służyć do prostego modelowania oddziaływania plazmy z wiązkami laserowymi bądź do modelowania niestabilności w symulacjach elektrostatycznych.

Oczywistym kierunkiem rozwoju dla kodu jest dodanie obsługi ruchu cząstek w wielu wymiarach. Wymagałoby to niestety znacznej zmiany modelu rozwiązywania równań ewolucji czasowej pola elektromagnetycznego. Warunki początkowe można zaadaptować z pola elektrostatycznego, aczkolwiek gdy układ straci symetrię $\frac{\partial}{\partial y} = \frac{\partial}{\partial z} = 0$, będzie konieczne również rozwiązywanie prawa Gaussa dla magnetyzmu 6.

Kolejnym potencjalnym krokiem byłoby wykorzystanie mocy obliczeniowej kart graficznych. Potencjalnym kandydatem do tego celu byłoby wykorzystanie popularnej biblioteki PyCUDA. Alternatywnym sposobem zrównoleglenia kodu byłoby wykorzystanie paradygmatu OpenMP/MPI, do czego może posłużyć biblioteka MPI4PY.

Symulacja nie uwzględnia również kolizji. Możliwe jest dodanie ich na przykład poprzez sprzężenie algorytmu z implementacją DSMC (Direct Simulation Monte Carlo) [?].

W bieżącej symulacji użyto prostego modelu pojedynczo zjonizowanych atomów wodoru bez możliwości syntezy w neutralne cząstki wodoru. Istnieje możliwość wybrania atomów innych pierwiastków, lecz w chwili obecnej kod zupełnie nie uwzględnia możliwości dalszej jonizacji tych cząstek.

LITERATURA

Literatura

- [1] pytest. <https://docs.pytest.org/en/latest/>. Dostęp: 2017-08-13. 37
- [2] Travis CI. <https://travis-ci.org/>. Accessed: 2017-08-13. 38
- [3] C.K. Birdsall and A.B. Langdon. *Plasma Physics via Computer Simulation*. Series in Plasma Physics and Fluid Dynamics. Taylor & Francis, 2004. 22, 27, 29, 30, 43, 44
- [4] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, U. Schramm, J. Schuchart, and R. Widera. Radiative signatures of the relativistic kelvin-helmholtz instability. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 5:1–5:12, New York, NY, USA, 2013. ACM. 13
- [5] F.F. Chen. *Introduction to Plasma Physics and Controlled Fusion*. Number v. 1 in Introduction to Plasma Physics and Controlled Fusion. Springer, 1984. 19
- [6] Andrew Collette. *Python and HDF5*. O'Reilly, 2013. 37
- [7] Matt Davis. Snakeviz. <https://github.com/jiffyclub/snakeviz/>, 2016. 39
- [8] Freddie Witherden, Michael Klemm, and Peter Vincent. PyFR - EuroSciPy 2015. 14
- [9] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, May 2007. 37
- [10] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2017-08-13]. 36
- [11] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. 36
- [12] M.E.J. Newman. *Computational Physics*. Createspace Independent Pub, 2012. 21
- [13] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. 14
- [14] TODO. Todo. 36
- [15] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. 36
- [16] J. Villasenor and O. Buneman. Rigorous charge conservation for local electromagnetic field solvers. *Computer Physics Communications*, 69:306–316, March 1992. 23, 24, 28

- [17] F.D. Witherden, A.M. Farrington, and P.E. Vincent. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028 – 3040, 2014. 14