Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Stanisław Dawidowicz

Nr albumu: 1100699

Symulacje Monte Carlo ciekłych kryształów nematycznych dwuosiowych

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2018

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Serdeczne podziękowania i wyrazy wdzięczności składam panu dr. hab. Andrzejowi Kapanowskiemu, za ogromny wkład w przygotowanie podstaw teoretycznych dla projektu oraz nieustanne, twórcze wsparcie na każdym etapie pracy. Szczególną wdzięczność chciałbym wyrazić Panu Doktorowi za okazaną mi życzliwość i cierpliwość.

Streszczenie

Praca opisuje implementację zestawu narzędzi do przeprowadzania symulacji Monte Carlo ciekłych kryształów nematycznych termotropowych. Narzędzia te zostały zaimplementowane w języku Python. Ciekłe kryształy są to materiały, w których występują fazy pośrednie pomiędzy kryształem a izotropową cieczą. W ciekłych kryształach termotropowych przejścia fazowe są indukowane zmianami temperatury. Fazy nematyczne wykazują uporządkowanie orientacyjne anizotropowych molekuł przy braku uporządkowania ich środków mas. Symulacje komputerowe są pomocne w zrozumieniu właściwości ciekłych kryształów i w poszukiwaniu materiałów o pożądanych właściwościach.

W ramach projektu stworzono implementację kwaternionów, korzystającą z pakietu NumPy, oraz aplikację umożliwiającą uruchamianie symulacji dla modelu Isinga, modelu Lebwohla-Lashera i modelu molekuł dwuosiowych. We symulacjach Monte Carlo zastosowano algorytm Metropolisa-Hastingsa. Aplikacja pozwala również uzyskać podgląd stanu układu po przeprowadzonej symulacji i ułatwia analizę wyników, generując wykresy zależności temperaturowych wybranych wielkości fizycznych. Podgląd stanu układu zaimplementowano przy użyciu technologii OpenGL, której praktyczne podstawy zostały opisane w pracy.

W pracy przedstawiono sposób konfiguracji projektu na wybranych systemach operacyjnych. Opisane zostały również podstawowe przypadki użycia stworzonych narzędzi, z wykorzystaniem przykładów ilustrujących parametryzację oraz oczekiwane wyniki. Praca zawiera także opis struktury i implementacji aplikacji.

Symulator został wykorzystany do odtworzenia wyników znanych eksperymentów komputerowych opisanych w literaturze przedmiotowej, co pozwoliło na potwierdzenie poprawności stworzonego kodu. W pracy przedstawiono i omówiono wyniki przeprowadzonych eksperymentów.

Słowa kluczowe: ciekłe kryształy, symulacje Monte Carlo, model Isinga, model Lebwohla-Lashera, kwaterniony, Python, OpenGL

English title: Monte Carlo simulations of biaxial nematic liquid crystals

Abstract

This thesis details implementation of a Monte Carlo thermotropic nematic liquid crystal simulator toolchain. The toolchain was implemented using the Python programming language. Liquid crystals are materials with intermediate phases between a crystal and a liquid. These materials are thermotropic if phase transitions are triggered by the change in temperature. In nematic phases, the centres of gravity of the anisotropic molecules have a liquid-like order but there is some orientational order. Computer simulations are useful in determining the properties of liquid crystals and finding materials with the desired properties.

As a part of the project, a class implementing quaternions was created where the NumPy package was used. Additionally, a script was created that allows for running preconfigured simulations for the following theoretical models: the Ising model, the Lebwohl-Lasher model, and the biaxial molecule model. The Metropolis-Hastings algorithm was utilized in these simulations. The toolchain also contains a script allowing for viewing the system state after simulations, as well as a script for analyzing the final results. The script generates graphs with temperature dependence of selected physical variables. The system-state view was implemented using OpenGL. The basics of utilizing OpenGL in regards to this implementation have also been described in this thesis.

The thesis also contains instructions for configuring the simulator on selected operating systems, as well as instructions for specific use cases with examples provided. A detailed description of the toolchain's structure and implementation is also provided.

The toolchain was used to re-create famous computer experiments described in the subject literature. We confirmed known results and tested the correctness of our programs in this way. The details of those experiments are also included.

Keywords: liquid crystals, Monte Carlo simulations, Lebwohl-Lasher model, Ising model, quaternions, Python, OpenGL

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

Pierwotnym celem projektu było stworzenie biblioteki pythonowej implementującej kwaterniony oraz operacje matematyczne na kwaternionach. Projekt ten został rozszerzony i w docelowej postaci jest prototypem narzędzia do przeprowadzania komputerowych symulacji Monte Carlo ciekłych kryształów i wizualizacji wyników.

Tak więc, tematem niniejszej pracy są symulacje komputerowe ciekłych kryształów. Nazwa *ciekły kryształ* odnosi się do stanu materii o właściwościach pośrednich między cieczą a kryształem. Ciekłymi kryształami nazywa się także materiały występujące w takim stanie materii, w fazie ciekłokrystalicznej. Materiały te mają wiele zastosowań, z których najpopularniejsze to wyświetlacze LCD (ang. *Liquid Crystal Display*) [1]. Podstawowym celem pracy było stworzenie programu do przeprowadzania symulacji ciekłych kryształów nematycznych dwuosiowych oraz obrazowania ułożenia molekuł w próbce. Symulacje komputerowe pomagają lepiej zrozumieć właściwości istniejących materiałów, co jest przydatne w syntezie nowych materiałów o potrzebnych cechach. Ciekłe kryształy nematyczne dwuosiowe w teorii mogą być bardzo przydatne w przemyśle, ale w praktyce od wielu lat są problemy z otrzymaniem odpowiednich ciekłych kryształów termotropowych, tj. takich, w których przejścia fazowe są indukowane zmianami temperatury.

Ciekłe kryształy są zwykle zbudowane z molekuł organicznych, o kształcie anizotropowym. Molekuły tworzące fazę ciekłokrystaliczną modelujemy jako bryły sztywne o symetrii co najmniej D_{2h}, czyli bryły posiadające trzy prostopadłe do siebie osie 2-krotne. Bryły są również symetryczne względem inwersji, co generuje trzy odbicia względem płaszczyzn prostopadłych do osi 2-krotnych. Orientację bryły, a także obroty w przestrzeni trójwymiarowej, można opisywać za pomocą trzech kątów Eulera. Taki opis ma jednak co najmniej dwie wady w kontekście obliczeń komputerowych. Po pierwsze, przy wyznaczaniu położenia osi symetrii bryły, potrzebne są kosztowne wywołania funkcji trygonometrycznych. Po drugie, w pewnych konfiguracjach następuje zjawisko redukcji stopni swobody (ang. *gimbal lock*).

Z tego powodu podjęliśmy decyzję o wykorzystaniu kwaternionów w naszych obliczeniach [2]. Zdecydowaliśmy się wykorzystać implementację kwaternionów z pierwotnej wersji projektu, zamiast polegać na implementacjach z innych bibliotek. Kwaterniony zostały zastosowane w symulacjach ciekłych kryształów dwuosiowych oraz przy programowaniu aplikacji umożliwiającej wizualizację stanu macierzy.

Ze względu na duże znaczenie podglądu fizycznego stanu macierzy dla weryfikacji poprawności symulacji, zdecydowaliśmy się jako pierwszą zaimplementować część aplikacji odpowiedzialną za wizualizację wyników.

Następnym istotnym dla pracy etapem było zaznajomienie się z metodą

Monte Carlo, która jest szeroko stosowana w symulacjach w różnych działach fizyki. Z tego powodu, jako pierwsza została zaimplementowana symulacja modelu Isinga [3], który opisuje oddziaływujące spiny leżące w węzłach sieci 3D. Korzystaliśmy z programu napisanego w języku Python przez Kristiana Haule, dotyczącego spinów na sieci 2D [4]. Następnie zaimplementowano model Lebwohla-Lashera [5], który opisuje ciekłe kryształy nematyczne jednoosiowe. W ostatnim etapie projektu zaimplementowano model symulujący fazy ciekłokrystaliczne zawierające molekuły dwuosiowe.

Praca została zorganizowana w następujący sposób. Rozdział 1 opisuje cele oraz etapy pracy nad projektem. Rozdział 2 zawiera wprowadzenie do fizyki ciekłych kryształów. Rozdział 3 jest poświęcony matematyce kwaternionów. Rozdział 4 opisuje zastosowane technologie oraz motywy ich wyboru. Rozdział 5 zawiera wprowadzenie do technologii OpenGL. Rozdział 6 stanowi instrukcję poprawnej konfiguracji projektu, dla wybranych systemów operacyjnych. Rozdział 7 opisuje sposób korzystania z aplikacji. Rozdział 8 przedstawia implementację aplikacji przygotowanej w ramach projektu. Rozdział 9 zawiera opis eksperymentów przeprowadzonych za pomocą symulatora, dla sprawdzenia jego merytorycznej poprawności. Rozdział 10 jest podsumowaniem pracy. W dodatku A przeanalizowano model Isinga, a w dodatku B pokazano przykładowe obliczenia na kwaternionach.

2. Fizyka ciekłych kryształów

Materia zwykle znajduje się w jednym z trzech stanów skupienia: stałym, ciekłym lub gazowym. Ciekłe kryształy to materiały, które posiadają dodatkowe stany o właściwościach pośrednich między stanem ciekłym, a stanem stałym (tzw. mezofazy lub stany mezomorficzne). W tym rozdziale omówimy krótko cechy tych materiałów.

2.1. Wprowadzenie do ciekłych kryształów

Ciekłe kryształy na ogół zbudowane są z molekuł anizotopowych: wydłużonych jak pręty albo spłaszczonych jak dyski. W fazie ciekłej środki mas molekuł są ułożone przypadkowo, podobnie jak ich orientacje. W fazie stałej mamy uporządkowanie środków mas na sieci krystalicznej i uporządkowanie orientacji molekuł względem pewnego kierunku. W fazach pośrednich pojawia się uporządkowanie w pewnych stopniach swobody, ale pozostaje nieporządek w innych stopniach swobody. Obecnie znanych jest wiele różnych faz ciekłokrystalicznych. W niniejszej pracy zajmujemy się ciekłymi kryształami *nematycznymi*, w których środki mas molekuł są ułożone przypadkowo jak w cieczy, ale występuje uporządkowanie orientacyjne molekuł. Stopień uporządkowania orientacji molekuł opisujemy za pomocą parametrów porządku, które omówimy w dalszych rozdziałach.

2.2. Symulacje komputerowe ciekłych kryształów

Symulacje komputerowe pomagają zrozumieć zjawiska fizyczne zachodzące w układach atomowych i molekularnych. W przypadku ciekłych kryształów okazało się, że wiele faz może być modelowanych za pomocą twardych anizotropowych molekuł, które oddziaływują ze sobą jedynie w chwili zderzenia [6]. Następnym krokiem było uwzględnienie oddziaływania przyciągającego, które modelowano potencjałem typu Lennarda-Jonesa [7] lub w inny sposób.

Dwie najważniejsze metody stosowane w symulacjach to Monte Carlo (MC) i dynamika molekularna (MD). Metoda Monte Carlo wykorzystuje liczby przypadkowe do modelowania zachowania układu molekuł. Dynamika molekularna polega na rozwiązywaniu klasycznych równań ruchu Newtona dla układów wielociałowych i wyznaczaniu równowagowych i nierównowagowych właściwości układów.

2.3. Model Lebwohla-Lashera

W modelu Lebwohla-Lashera (LL) środki mas molekuł znajdują się w węzłach sieci prostej kubicznej (ang. *simple cubic lattice*) z periodycznymi warunkami brzegowymi [5]. Molekuły są jednoosiowe o symetrii $D_{\infty h}$, czyli mają oś symetrii obrotowej i płaszczyznę odbicia prostopadłą do tej osi. Oddziaływanie molekuł z ich najbliższymi sąsiadami, wyraża się wzorem

$$H = -\epsilon \sum_{ij} P_2(\cos\theta_{ij}), \tag{2.1}$$

gdzie θ_{ij} jest kątem pomiędzy osiami sąsiadujących molekuł i oraz j, ϵ jest miarą oddziaływania, $P_2(x) = (3x^2 - 1)/2$ jest drugim wielomianem Legendre'a. Temperaturę podaje się w jednostkach bezwymiarowych $T^* = k_B T/\epsilon$, gdzie k_B to stała Boltzmanna. Dla konfiguracji idealnie uporządkowanej dostajemy $H = -3\epsilon N$, gdzie N jest liczbą molekuł.

Symulacje MC dla modelu LL, przedstawione w pracy [5], przebiegały następująco. Węzły sieci ponumerowano sekwencyjnie. Ustalono konfigurację początkową układu. W fazie izotropowej była to konfiguracja przypadkowa, w fazie nematycznej wybierano konfigurację z idealnym uporządkowaniem. Wybrano pierwszą molekułę i ustalono dla niej przypadkową orientację. Porównywano nową energię układu E_n z energią E_p w poprzednim stanie. Jeżeli $E_n < E_p$, to zachowywano nową konfigurację. Jeżeli $E_n > E_p$, to nową konfigurację zachowywano z prawdopodobieństwem

$$\exp[-(E_n - E_p)/(k_B T)] \tag{2.2}$$

Tą samą procedurę zastosowano do kolejnych molekuł na sieci. Powtarzanie tej procedury prowadzi do stanu równowagowego. Z dala od przejścia fazowego używano sieci $10 \times 10 \times 10$ (średniowano po 2000 konfiguracjach układu), w pobliżu przejścia powiększono sieć do $20 \times 20 \times 20$ (średniowano po 8000 konfiguracjach). Warto zaznaczyć, że jeden cykl symulacji obejmuje przejście przez wszystkie molekuły w układzie. Właściwości termodynamiczne układu były określane poprzez średniowanie po konfiguracjach układu, które wygenerowano po dojściu do równowagi.

W modelu istnieje przejście fazowe pierwszego rodzaju z fazy izotropowej do fazy nematycznej jednoosiowej, którego temperatura w granicy termodynamicznej szacowana jest na $T^*_{NI} = 1.1255$. W oryginalnej pracy $T^*_{NI} = 1.124(7)$, skok parametru porządku $\langle P_2 \rangle = 0.33(4)$, ciepło przemiany 1.09ϵ.

Dużo pracy autorzy włożyli w badanie natury przejścia fazowego, którego obecność wskazywały nieciągłości na wykresie temperaturowym energii układu i parametru porządku. Po pierwsze, wykonano przybliżone obliczenia analityczne dla tego modelu. Po drugie, w punkcie przejścia wykonano histogram, który zliczał konfiguracje o danej wartości parametru porządku. Pojawiły się dwa piki, które potwierdzały współistnienie dwóch stanów, fazy izotropowej (parametr porządku bliski zeru) i fazy uporządkowanej orientacyjnie (parametr porządku powyżej 0.3).

Model LL zawiera cechy charakterystyczne przejścia fazowego NI, takie jak małe ciepło utajone (ang. *latent heat*), rozbieżność korelacji orientacyjnych przy przejściu fazowym (Fabbri, Zannoni, 1986) [8] [sieć $30 \times 30 \times 30$,

$T^*_{NI} = 1.1232(6)$]. Do symulacji modelu LL wykorzystywano też komputery wieloprocesorowe, co pozwoliło na badanie sieci $120 \times 120 \times 120$ [9]. Zaobserwowano przesunięcie piku ciepła właściwego w stronę wyższych temperatur wraz ze wzrostem pola zewnętrznego. Obecność bardzo słabego przejścia fazowego pierwszego rodzaju w modelu LL potwierdził Zhang *et al.* [sieć $28 \times 28 \times 28$, $T^*_{NI} = 1.1232(1)$, ciepło przejścia $0.20(4)\epsilon$] [10]. Priezjev i Pelcovits pokazali jak zmodyfikować klastrowy algorytm Wolffa dla układów spinowych (redukcja krytycznego spowolnienia), tak aby działał dla direktorów w ciekłych kryształach [11]. Greef i Lee badali fluktuacje w modelu LL w temperaturach tuż powyżej przejścia fazowego [sieć $30 \times 30 \times 30$] [12].

2.4. Przykładowe obliczenia dla modelu Lebwohla-Lashera

W fazie nematycznej jednoosiowej środki mas molekuł nie wykazują uporządkowania przestrzennego. Istnieje natomiast uporządkowanie orientacyjne molekuł, które mają tendencję do ułożenia równoległego względem osi wyznaczonej przez jednostkowy wektor (direktor) $\vec{N}$. Stany direktora $\vec{N}$ oraz $-\vec{N}$ są nierozróżnialne.

Stopień uporządkowania molekuł określa tensor symetryczny bezśladowy Q

$$Q_{\alpha\beta} = \frac{3}{2} x \left(N_\alpha N_\beta - \frac{1}{3}\delta_{\alpha\beta} \right), \tag{2.3}$$

gdzie x jest skalarnym parametrem porządku, należącym do przedziału $[0, 1]$. W fazie izotropowej $x = 0$. Wektor $\vec{N}$ jest wektorem własnym odpowiadającym największej wartości własnej x. Pozostałe dwie wartości własne wynoszą $-x/2$. W symulacjach komputerowych tensor Q w danej konfiguracji obliczamy następująco

$$Q^{nn}_{\alpha\beta} = \frac{3}{2N} \sum_{i=1}^{N} \left(n^i_\alpha n^i_\beta - \frac{1}{3}\delta_{\alpha\beta} \right), \tag{2.4}$$

gdzie $\vec{n}^i$ to jednostkowy wektor określający kierunek osi długiej molekuły numer i. Tensor $Q^{nn}_{\alpha\beta}$ diagonalizujemy i największą wartość własną przyjmujemy za skalarny parametr porządku x.

2.5. Model z molekułami dwuosiowymi

Przy opisie faz tworzonych przez molekuły dwuosiowe korzysta się z rzeczywistych funkcji specjalnych $F^{(j)}_{\mu\nu}(R)$ dopasowanych do symetrii D_{2h} [13], przy czym R oznacza orientację molekuły. Najważniejsze funkcje mają postać

$$F^{(0)}_{00}(R) = 1, \tag{2.5}$$

$$F^{(2)}_{00}(R) = \frac{1}{2}(-1 + 3n_z^2) = P_2(n_z), \tag{2.6}$$

$$F_{02}^{(2)}(R) = \frac{\sqrt{3}}{2}(-1 + n_z^2 + 2l_z^2) = \frac{\sqrt{3}}{3}[P_2(l_z) - P_2(m_z)], \tag{2.7}$$

$$F_{20}^{(2)}(R) = \frac{\sqrt{3}}{2}(-1 + n_z^2 + 2n_x^2) = \frac{\sqrt{3}}{3}[P_2(n_x) - P_2(n_y)], \tag{2.8}$$

$$F_{22}^{(2)}(R) = \frac{1}{2}(-3 + n_z^2 + 2l_z^2 + 2n_x^2 + 4l_x^2), \tag{2.9}$$

$$F_{22}^{(2)}(R) = \frac{1}{3}[P_2(l_x) + P_2(m_y) - P_2(m_x) - P_2(l_y)]. \tag{2.10}$$

Uporządkowanie w układzie opisuje się za pomocą czterech parametrów porządku $\langle F_{00}^{(2)}\rangle$, $\langle F_{02}^{(2)}\rangle$, $\langle F_{20}^{(2)}\rangle$, $\langle F_{22}^{(2)}\rangle$, przy czym najważniejszy jest pierwszy i ostatni, bo z rosnącym uporządkowaniem zmierzają do wartości jeden. W fazie izotropowej $\langle F_{\mu\nu}^{(2)}\rangle = 0$. W fazie nematycznej jednoosiowej $\langle F_{0\nu}^{(2)}\rangle \neq 0$. W fazie nematycznej dwuosiowej $\langle F_{\mu\nu}^{(2)}\rangle \neq 0$.

Oddziaływanie molekuł dwuosiowych ma postać

$$H = \sum_{ij} V(R_i, R_j), \tag{2.11}$$

$$V(R_i, R_j) = \sum_k \sum_{\mu\nu} v_{\mu\nu}^{(k)} F_{\mu\nu}^{(k)}(R_i^{-1} R_j), \tag{2.12}$$

gdzie warunek niezmienniczości oddziaływania ze względu na permutację molekuł $V(R_i, R_j) = V(R_j, R_i)$ implikuje $v_{\mu\nu}^{(k)} = v_{\nu\mu}^{(k)}$. Wyraz $v_{00}^{(0)}$ zwykle się zaniedbuje, a wykorzystuje się wyrazy z $k = 2$. Sumowanie we wzorze (2.11) przebiega po najbliższych sąsiadach. Zauważmy, że dla dowolnego obrotu R zachodzi $V(RR_i, RR_j) = V(R_i, R_j)$. Model Lebwohla-Lashera odtwarzamy dla $v_{00}^{(2)} = -\epsilon$, $v_{02}^{(2)} = v_{20}^{(2)} = v_{22}^{(2)} = 0$.

Prace teoretyczne przewidują na ogół istnienie czterech faz w układach z molekułami dwuosiowymi, w zależności od stopnia dwuosiowości molekuł. Jest to faza izotropowa I, faza nematyczna jednoosiowa dodatnia N_+ (występująca dla molekuł prętopodobnych) oraz faza nematyczna jednoosiowa ujemna N_- (występująca dla molekuł dyskopodobnych), faza nematyczna dwuosiowa N_B. Przejście $I - N_U$ jest pierwszego rodzaju (nieciągłe) i słabnie wraz ze wzrostem dwuosiowości molekuł, aż do punktu o największej dwuosiowości molekuł (punkt Landaua). W tym punkcie układ powinien mieć bezpośrednie ciągłe przejście $I - N_B$, które należy do klasy uniwersalności izotropowego modelu Heisenberga 3D z wymiarem parametru porządku $n = 5$ [14]. Przejście $N_U - N_B$ jest drugiego rodzaju (ciągłe).

W literaturze były badane różne zestawy współczynników $v_{\mu\nu}^{(2)}$ [15]. Dla uzyskania fazy nematycznej dwuosiowej niezbędny jest niezerowy współczynnik $v_{22}^{(2)}$. Jednym ze sposobów ustalenia wartości współczynników $v_{\mu\nu}^{(2)}$ jest założenie, że oddziaływanie $V(R_i, R_j)$ jest proporcjonalne do *objętości wykluczonej* dla molekuł w kształcie prostopadłościanów o wymiarach $L \times B \times W$ [16].

$$3v_{00}^{(0)} = 8LBW + 2(W + B)(W + L)(B + L), \tag{2.13}$$

$$3v_{00}^{(2)} = 6LBW + L(W^2 + B^2) - 2W(B^2 + L^2) - 2B(L^2 + W^2), \quad (2.14)$$

$$v_{02}^{(2)} = v_{20}^{(2)} = \frac{\sqrt{3}}{3}(B - W)(L^2 - WB), \quad (2.15)$$

$$v_{22}^{(2)} = -L(W - B)^2. \quad (2.16)$$

Molekuły jednoosiowe dostaje się dla $W = B$, wtedy $v_{02}^{(2)} = v_{20}^{(2)} = v_{22}^{(2)} = 0$. Największa dwuosiowość występuje dla $L^2 = WB$, gdzie $v_{02}^{(2)} = v_{20}^{(2)} = 0$, $v_{00}^{(2)} > 0$, $v_{22}^{(2)} < 0$ (wtedy układ przechodzi przez punkt Landaua). W podanej parametryzacji istnieje mapowanie pomiędzy układem prętopodobnym a dyskopodobnym.

Inna często stosowana parametryzacja oddziaływania ma postać [17], [18], [19]

$$V(R_i, R_j) = -\epsilon\{F_{00}^{(2)} + a[F_{02}^{(2)} + F_{20}^{(2)}] + a^2 F_{22}^{(2)}\}, \quad (2.17)$$

gdzie punkt Landaua dostaje się dla $a = 1/\sqrt{3}$, dla $a = 0$ odtwarza się model LL dla prętów, dla $a = \sqrt{2}$ mamy model LL dla dysków. Dla konfiguracji idealnie uporządkowanej dostajemy $H = -3\epsilon N(1 + a^2)$, gdzie N jest liczbą molekuł.

2.6. Przykładowe obliczenia dla modelu z molekułami dwuosiowymi

W fazie nematycznej dwuosiowej stopień uporządkowania molekuł określa tensor symetryczny bezśladowy Q [20]

$$Q_{\alpha\beta} = \frac{3}{2}x\left(N_\alpha N_\beta - \frac{1}{3}\delta_{\alpha\beta}\right) - \frac{1}{2}y\left(L_\alpha L_\beta - M_\alpha M_\beta\right), \quad (2.18)$$

gdzie $\vec{L}$, $\vec{M}$, $\vec{N}$ są trzema ortogonalnymi wektorami własnymi Q, odpowiadającymi wartościom własnym $-(x+y)/2$, $-(x-y)/2$, x. W fazie izotropowej $x = y = 0$, w fazie nematycznej jednoosiowej $x \neq 0$, $y = 0$, w fazie nematycznej dwusiowej $x \neq 0$, $y \neq 0$. Warto zauważyć następujące związki (inwarianty):

$$\mathrm{Tr}(Q) = Q_{\alpha\alpha} = 0, \quad (2.19)$$

$$\mathrm{Tr}(Q^2) = Q_{\alpha\beta}Q_{\alpha\beta} = \frac{3x^2 + y^2}{2}, \quad (2.20)$$

$$\mathrm{Tr}(Q^3) = Q_{\alpha\beta}Q_{\beta\gamma}Q_{\gamma\alpha} = \frac{3x(x^2 - y^2)}{4}. \quad (2.21)$$

Faza ma największą dwuosiowość dla niezmiennika $\mathrm{Tr}(Q^3) = 0$, czyli mamy dwa przypadki:

— Dla $x = y$, $Q_{\alpha\beta} = x(N_\alpha N_\beta - L_\alpha L_\beta)$.

— Dla $y = -x$, $Q_{\alpha\beta} = x(N_\alpha N_\beta - M_\alpha M_\beta)$.

W symulacjach komputerowych wyznacza się trzy tensory [21], [22], [17]

$$Q^{ll}_{\alpha\beta} = \frac{3}{2N}\sum_{i=1}^{N}\left(l^i_\alpha l^i_\beta - \frac{1}{3}\delta_{\alpha\beta}\right), \tag{2.22}$$

$$Q^{mm}_{\alpha\beta} = \frac{3}{2N}\sum_{i=1}^{N}\left(m^i_\alpha m^i_\beta - \frac{1}{3}\delta_{\alpha\beta}\right), \tag{2.23}$$

$$Q^{nn}_{\alpha\beta} = \frac{3}{2N}\sum_{i=1}^{N}\left(n^i_\alpha n^i_\beta - \frac{1}{3}\delta_{\alpha\beta}\right), \tag{2.24}$$

gdzie trójka wektorów jednostkowych ortogonalnych $(\vec{l^i}, \vec{m}^i, \vec{n}^i)$ wyznacza orientację molekuły numer i. Po jednoczesnym zdiagonalizowaniu trzech tensorów otrzymuje się związki na parametry porządku [17]

$$\langle F^{(2)}_{00}\rangle = Q^{nn}_{zz} = -Q^{ll}_{zz} - Q^{mm}_{zz}, \tag{2.25}$$

$$\sqrt{3}\langle F^{(2)}_{02}\rangle = Q^{ll}_{zz} - Q^{mm}_{zz} = -Q^{ll}_{xx} - Q^{ll}_{yy} + Q^{mm}_{xx} + Q^{mm}_{yy}, \tag{2.26}$$

$$\sqrt{3}\langle F^{(2)}_{20}\rangle = Q^{nn}_{xx} - Q^{nn}_{yy} = -Q^{ll}_{xx} - Q^{mm}_{xx} + Q^{ll}_{yy} + Q^{mm}_{yy}, \tag{2.27}$$

$$3\langle F^{(2)}_{22}\rangle = Q^{ll}_{xx} + Q^{mm}_{yy} - Q^{ll}_{yy} - Q^{mm}_{xx}. \tag{2.28}$$

Po diagonalizacji tensorów nie jest oczywiste, które kierunki należy przyjąć za x, y, z. Przyjmujemy procedurę opisana w pracy [17]. Parametr porządku $\langle F^{(2)}_{00}\rangle$ ma być nieujemny. Parametry porządku obliczane na dwa sposoby mają mieć taką samą wartość. Parametry porządku w nowej temperaturze powinny być bliskie wartościom z poprzedniej temperatury.

3. Matematyka kwaternionów

Materiał przedstawiony w tym rozdziale przekracza to, co było niezbedne dla potrzeb naszego projektu. Jednak pewne zagadnienia, np. opis obrotu bryły sztywnej za pomocą kwaternionu, nie są łatwo dostępne w literaturze naukowej, dlatego zebraliśmy je razem, aby mogły służyć w przyszłości jako poradnik praktycznych zastosowań kwaternionów.

3.1. Definicja kwaternionów

Zbiór kwaternionów $\mathbb{H}$ może być zidentyfikowany jako $\mathbb{R}^4$, czyli czterowymiarowa przestrzeń wektorowa nad ciałem liczb rzeczywistych $\mathbb{R}$ [2]. W zbiorze $\mathbb{H}$ mamy trzy operacje: dodawanie, mnożenie przez skalar i mnożenie kwaternionów. Pierwsze dwa działania są tożsame z działaniami w $\mathbb{R}^4$. Mnożenie kwaternionów wygodnie jest opisać używając elementów bazowych oznaczanych $\{1, i, j, k\}$, gdzie 1 jest elementem jednostkowym. Każdy element zbioru $\mathbb{H}$ można zapisać jako kombinację liniową

$$q = q_0 1 + q_1 i + q_2 j + q_3 k, \tag{3.1}$$

przy czym zwykle pomija się 1. Liczby q_0, q_1, q_2, q_3 są rzeczywiste. Mnożenie kwaternionów można odtworzyć z mnożenia elementów bazowych

$$i^2 = j^2 = k^2 = ijk = -1. \tag{3.2}$$

Mnożenie kwaternionów na ogół nie jest przemienne, np. $ij = k$, $ji = -k$. Natomiast mnożenie kwaternionu przez liczbę rzeczywistą jest przemienne.

Grupa kwaternionów. Zbiór ośmiu kwaternionów $\{1, -1, i, -i, j, -j, k, -k\}$ tworzy nieabelową grupę multiplikatywną rzędu 8, nazywaną *grupą kwaternionów* Q_8 [23]. Generatorami tej grupy są kwaterniony i oraz j. Grupa kwaternionów pojawia się w mechanice kwantowej przy opisie spinu elektronu (teoria Pauliego).

3.2. Część skalarna i część wektorowa

Kwaternion postaci $q_0 + 0i + 0j + 0k$ jest nazywany *rzeczywistym*. Kwaternion postaci $0 + q_1 i + q_2 j + q_3 k$ jest nazywany *czysto urojonym*, o ile $|q_1| + |q_2| + |q_3| > 0$. Dla dowolnego kwaternionu q, element q_0 jest nazywany *częścią skalarną*, a $\vec{q} = q_1 i + q_2 j + q_3 k$ *częścią wektorową*. W tej konwencji część wektorowa jest wektorem z $\mathbb{R}^3$.

Tabela 3.1. Tabelka mnożenia kwaternionów (tablica Cayleya).

1	-1	i	$-i$	j	$-j$	k	$-k$
-1	1	$-i$	i	$-j$	j	$-k$	k
i	$-i$	-1	1	k	$-k$	$-j$	j
$-i$	i	1	-1	$-k$	k	j	$-j$
j	$-j$	$-k$	k	-1	1	i	$-i$
$-j$	j	k	$-k$	1	-1	$-i$	i
k	$-k$	j	$-j$	$-i$	i	-1	1
$-k$	k	$-j$	j	i	$-i$	1	-1

Jeżeli kwaternion zapiszemy w postaci $q = (q_0, \vec{q})$, to działania na kwaternionach można zapisać następująco ($a \in \mathbb{R}$):

$$(p_0, \vec{p}) + (q_0, \vec{q}) = (p_0 + q_0, \vec{p} + \vec{q}), \tag{3.3}$$

$$a(p_0, \vec{p}) = (ap_0, a\vec{p}), \tag{3.4}$$

$$(p_0, \vec{p})(q_0, \vec{q}) = (p_0 q_0 - \vec{p} \cdot \vec{q}, p_0\vec{q} + q_0\vec{p} + \vec{p} \times \vec{q}), \tag{3.5}$$

gdzie wykorzystano iloczyn skalarny i wektorowy z $\mathbb{R}^3$. Dla dwóch czysto urojonych kwaternionów $p = (0, \vec{p})$ i $q = (0, \vec{q})$ możemy zapisać

$$(\vec{p} \cdot \vec{q}, 0) = \frac{1}{2}(\bar{p}q + \bar{q}p) = \frac{1}{2}(p\bar{q} + q\bar{p}), \tag{3.6}$$

$$(0, \vec{p} \times \vec{q}) = \frac{1}{2}(pq - \bar{q}\bar{p}). \tag{3.7}$$

3.3. Sprzężenie, norma, iloczyn skalarny

Sprzężenie kwaternionu $q = q_0 + q_1 i + q_2 j + q_3 k$ (ang. *conjugate of* q) to jest kwaternion $\bar{q} = q_0 - q_1 i - q_2 j - q_3 k$. Podwójne sprzężenie zwraca oryginalny kwaternion. Sprzężenie iloczynu kwaternionów wynosi $\overline{pq} = \bar{q}\bar{p}$ (zmiana kolejności). Okazuje się, że kwaternion sprzężony można otrzymać wykorzystując dodawanie i mnożenie

$$\bar{q} = \frac{-1}{2}(q + iqi + jqj + kqk). \tag{3.8}$$

Za pomocą sprzężenia definiuje się *normę kwaternionu*,

$$\|q\| = \sqrt{q\bar{q}} = \sqrt{\bar{q}q} = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}. \tag{3.9}$$

Norma jest multiplikatywna, czyli dla dowolnych dwóch kwaternionów p i q zachodzi związek

$$\|pq\| = \|p\|\|q\|. \tag{3.10}$$

Iloczyn skalarny dwóch kwaternionów p i q definiuje się jako

$$\langle p, q \rangle = p\bar{q}. \tag{3.11}$$

Widać, że normę kwaternionu można zapisać jako $\|q\| = \sqrt{\langle q, q \rangle}$.

Kwaternion odwrotny do kwaternionu $q = q_0 + q_1 i + q_2 j + q_3 k$ (ang. *reciprocal of* q) oznaczamy przez q^{-1}, przy czym spełnione są zależności $qq^{-1} = q^{-1}q = 1$,

$$q^{-1} = \frac{1}{\|q\|^2}\bar{q}. \tag{3.12}$$

Warto zauważyć, że dzielenie kwaternionu p przez kwaternion q nie jest jednoznacznie zdefiniowane, ponieważ są dwie nierównoważne możliwości wyboru: pq^{-1} lub $q^{-1}p$ [2].

3.4. Kwaternion jako macierz rzeczywista

Kwaternion q można zapisać jako macierz rzeczywistą 4×4

$$\begin{pmatrix} q_0 & q_1 & -q_3 & -q_2 \\ -q_1 & q_0 & -q_2 & q_3 \\ q_3 & q_2 & q_0 & q_1 \\ q_2 & -q_3 & -q_1 & q_0 \end{pmatrix}. \tag{3.13}$$

Wtedy działania na kwaternionach sprowadzają się do działań na macierzach 4×4.

3.5. Funkcje zmiennej kwaternionowej

$$\exp(q) = \sum_{n=0}^{\infty} \frac{q^n}{n!} = \exp(q_0)\left(\cos\|\vec{q}\| + \frac{\vec{q}}{\|\vec{q}\|}\sin\|\vec{q}\|\right). \tag{3.14}$$

Zwróćmy uwagę, że zależność $\exp(p+q) = \exp(p)\exp(q)$ na ogół nie jest spełniona w zbiorze kwaternionów. Szczególne przypadki, kiedy ten związek jest prawdziwy, to np. $p = p_0$ lub $q = q_0$.

Formalny wzór na logarytm z kwaternionu jest następujący

$$\ln(q) = \ln\|q\| + \frac{\vec{q}}{\|\vec{q}\|}\arccos\frac{q_0}{\|q\|}. \tag{3.15}$$

W praktyce dla tej funkcji pojawiają się problemy z cięciami, analogiczne do przypadku liczb zespolonych, dlatego zdecydowaliśmy się nie implementaować tej funkcji dla kwaternionów.

Definicja funkcji trygonometrycznych opiera się na wzorze Eulera.

$$\cos(q) = \frac{\exp(+iq) + \exp(-iq)}{2}. \tag{3.16}$$

$$\sin(q) = \frac{\exp(+iq) - \exp(-iq)}{2i}. \tag{3.17}$$

$$\cosh(q) = \frac{\exp(+q) + \exp(-q)}{2} = \cos(iq). \tag{3.18}$$

$$\sinh(q) = \frac{\exp(+q) - \exp(-q)}{2} = -i\sin(iq). \tag{3.19}$$

Pierwiastki kwadratowe z minus jeden. Rozważmy równanie $q^2 = -1$ w zbiorze kwaternionów. dla $q = q_0 + q_1 i + q_2 j + q_3 k$ otrzymujemy cztery równania

$$q_0^2 - q_1^2 - q_2^2 - q_3^2 = -1, \tag{3.20}$$

$$2q_0 q_1 = 0, \tag{3.21}$$

$$2q_0 q_2 = 0, \tag{3.22}$$

$$2q_0 q_3 = 0. \tag{3.23}$$

Jeżeli q_0 byłoby niezerowe, to w konsekwencji dostalibyśmy sprzeczność: $q_1 = q_2 = q_3 = 0$, $q_0^2 = -1$. Dlatego rozwiązaniem jest zbiór kwaternionów czysto urojonych z normą równą jeden, $q_0 = 0$, $q_1^2 + q_2^2 + q_3^2 = 1$. Inaczej mówiąc jest to sfera jednostkowa w $\mathbb{R}^3$ [2].

3.6. Kwaterniony i obroty

Kwaterniony jednostkowe (ang. *unit quaternions*) są to kwaterniony z normą równą jeden. Niech $u = 0 + u_1 i + u_2 j + u_3 k$ będzie jednostkowym kwaternionem czysto urojonym, natomiast $v = 0 + v_1 i + v_2 j + v_3 k$ będzie zwykłym kwaternionem czysto urojonym. Wtedy

$$q = (q_0, \vec{q}) = \exp((\theta/2)u) = \cos(\theta/2) + u\sin(\theta/2), \tag{3.24}$$

$$q = (\cos(\theta/2), \vec{u}\sin(\theta/2)),\ \|q\| = 1, \tag{3.25}$$

$$qvq^{-1} = (0, \vec{v}\cos\theta + (\vec{u}\cdot\vec{v})\vec{u}(1-\cos\theta) + (\vec{u}\times\vec{v})\sin\theta), \tag{3.26}$$

$$qvq^{-1} = (0, \vec{v} + 2\vec{q}\times(\vec{q}\times\vec{v} + q_0\vec{v})). \tag{3.27}$$

Część urojona wyrażenia qvq^{-1} odpowiada wektorowi $\vec{v}$ obróconenu o kąt θ wokół osi obrotu zdefiniowanej przez wektor jednostkowy $\vec{u}$. Wzór 3.27 pozwala wykonać obrót wektora $\vec{v}$ z użyciem 15 mnożeń i 15 dodawań (lub 18 mnożeń i 12 dodawań, jeżeli czynnik 2 jest zrealizowany jako mnożenie) [24].

Jeżeli p i q są jednostkowymi kwaternionami, wtedy obrót odpowiadający pq wynosi

$$(pq)v(pq)^{-1} = pqvq^{-1}p^{-1} = p(qvq^{-1})p^{-1}, \tag{3.28}$$

co odpowiada wykonaniu najpierw obrotu q, a potem obrotu p. Nieprzemienność mnożenia kwaternionów odzwierciedla nieprzemienność składania obrotów. Obrót przeciwny do q to q^{-1}, co dla kwaternionów jednostkowych daje kwaternion sprzężony $\bar{q}$. Obrót q^2 to obrót wokół tej samej osi co q, ale o kąt dwa razy większy niż obrót q.

Przykład: Rozważmy obrót o kąt $\theta = 2\pi/3$ wokół osi $u = (i + j + k)/\sqrt{3}$ [24]. Odpowiedni kwaternion jednostkowy to

$$q = \cos(\pi/3) + u\sin(\pi/3) = \frac{1 + i + j + k}{2}. \tag{3.29}$$

$$qvq^{-1} = q(v_1 i + v_2 j + v_3 k)\bar{q} = v_3 i + v_1 j + v_2 k. \tag{3.30}$$

Zgodnie z oczekiwaniami, obrót o kąt $2\pi/3$ wokół długiej przekątnej sześcianu daje cykliczną permutację krawędzi sześcianu. Ortogonalna macierz obrotu $\mathcal{R}$ odpowiadająca kwaternionowi q ma postać

$$\mathcal{R} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}. \tag{3.31}$$

3.7. Obrót ciała sztywnego

Obrót wektora $\vec{v}$ można dokonać za pomocą ortogonalnej macierzy obrotu $\mathcal{R}$ [25]. Obrócony wektor to $\mathcal{R}\vec{v}$. Warunek ortogonalności ma postać $\mathcal{R}^T\mathcal{R} = \mathcal{I}$, gdzie $\mathcal{R}^T$ to macierz transponowana, a $\mathcal{I}$ to macierz jednostkowa.

Zauważmy matematyczną tożsamość prawdziwą dla każdych wektorów $\vec{\omega}$ i $\vec{v}$:

$$\vec{\omega} \times \vec{v} = \begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix} \vec{v} = \mathcal{W}(\vec{\omega})\vec{v}. \tag{3.32}$$

Jeżeli macierz obrotu $\mathcal{R}$ ma w różnych chwilach czasu pokazywać bieżącą orientację ciała względem zewnętrznego układu współrzędnych, to $\mathcal{R}$ musi być funkcją czasu. Okazuje się, że zmianę macierzy $\mathcal{R}(t)$ podczas obrotu ciała z prędkością kątową $\vec{\omega}$ można wyrazić wzorem

$$\frac{d\mathcal{R}}{dt} = \mathcal{W}(\vec{\omega})\mathcal{R}(t). \tag{3.33}$$

Orientację ciała możemy opisywać za pomocą kwaternionu $q(t)$ zależnego od czasu. Zmieniony kwaternion w czasie Δt możemy zapisać następująco [26]

$$q(t + \Delta t) = \left(\cos\left(\frac{\omega\Delta t}{2}\right), \sin\left(\frac{\omega\Delta t}{2}\right)\frac{\vec{\omega}}{\omega}\right) q(t), \tag{3.34}$$

$$q(t + \Delta t) \approx \left(1, \frac{\vec{\omega}\Delta t}{2}\right) q(t), \tag{3.35}$$

$$\frac{dq}{dt} = \lim_{\Delta t \to 0} \frac{q(t+\Delta t) - q(t)}{\Delta t} = \left(0, \frac{\vec{\omega}}{2}\right) q(t). \tag{3.36}$$

Mając dany kwaternion q możemy wyznaczyć odpowiadającą mu ortogonalną macierz obrotu $\mathcal{R}$ [24]

$$\mathcal{R} = \frac{1}{\|q\|^2} \begin{pmatrix} 2(q_0^2+q_1^2)-1 & 2(q_1q_2-q_0q_3) & 2(q_0q_2+q_1q_3) \\ 2(q_0q_3+q_1q_2) & 2(q_0^2+q_2^2)-1 & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_0q_1+q_2q_3) & 2(q_0^2+q_3^2)-1 \end{pmatrix}. \tag{3.37}$$

W naszej implementacji kwaternionów ten wzór przekształcono do postaci:

$$\mathcal{R} = \frac{1}{\|q\|^2} \begin{pmatrix} (q_0^2+q_1^2-q_3^2-q_4^2) & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 2(q_1q_2+q_0q_3) & (q_0^2-q_1^2+q_2^2-q_3^2) & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & (q_0^2-q_1^2-q_2^2+q_3^2) \end{pmatrix}. \tag{3.38}$$

3.8. Losowanie orientacji

W symulacjach MC pojawia się potrzeba losowania orientacji molekuły, czyli losowania odpowiedniego kwaternionu jednostkowego. Powstaje pytanie jak zapewnić odpowiedni rozkład losowanych orientacji. W naszym podejściu wyszliśmy od grupy obrotów, opisywanej trzema kątami Eulera. Po grupie można całkować z miarą

$$dR = d\phi d\theta \sin\theta d\phi = d\phi d(-\cos\theta) d\phi, \tag{3.39}$$

gdzie $\phi, \psi \in [0, 2\pi]$, $\theta \in [0, \pi]$, $\cos\theta \in [-1, 1]$. Te związki wykorzystano przy tworzeniu funkcji random_quat_biax() z modułu qtools.

Listing 3.1. Orientacja przypadkowa molekuły jako kwaternion.

```
def random_quat_biax():
    """Return a random rotation quat for biaxial molecules."""
    phi = random.uniform(0, 2*math.pi)
    ct = random.uniform(-1, 1)
    theta = math.acos(ct)    # mozna tez -ct
    psi = random.uniform(0, 2*math.pi)
    quat = Quat.create_from_axis_rotation([0, 0, 1], phi)
    quat *= Quat.create_from_axis_rotation([0, 1, 0], theta)
    quat *= Quat.create_from_axis_rotation([0, 0, 1], psi)
    return quat
```

Interfejs kwaternionów z modułu Quaternion zaprezentowano w dodatku B.

4. Zastosowane technologie

W tym rozdziale przedstawimy technologie zastosowane przy tworzeniu aplikacji oraz podamy motywacje dla ich wyboru.

4.1. Python 3

Wybór języka Python [52] do przeprowadzenia symulacji był podyktowany kilkoma względami.

Przede wszystkim, Python pozwala na szybkie tworzenie prototypów aplikacji, co jest istotne wtedy, gdy nie ma się doświadczenia w danej dziedzinie i trzeba dopiero ustalić optymalny przebieg przetwarzania danych, wąskie gardła systemu, itp.

Po drugie, Python dostarcza wielu pakietów wspomagających obliczenia numeryczne, przez co użycie Pythona staje się coraz bardziej popularne w wielu dziedzinach, takich jak np. bioinformatyka, *data science*, uczenie maszynowe.

Ponadto, Python ułatwia łączenie komponentów napisanych w różnych językach, co daje możliwość przepisania w przyszłości wybranych fragmentów aplikacji na inne bardziej wydajne języki programowania, bez konieczności ponownej implementacji całego projektu.

Początkowo planowaliśmy zaimplementować aplikację używając Pythona 2. Jednakże, okazało się, że biblioteki pythonowe obsługujące grafikę 3D bardzo wolno działają dla Pythona 2. Z tego powodu zdecydowaliśmy się użyć Pythona 3, który ponadto szybciej obsługiwał symulacje.

4.2. NumPy

W projekcie wykorzystaliśmy bibliotekę NumPy, która dostarcza wiele użytecznych funkcji matematycznych oraz strukturę danych reprezentującą wielowymiarowe tablice wartości [53]. Ułatwia ona implementację obliczeń na kwaternionach i macierzach, pozytywnie wpływa na szybkość i dokładność obliczeń. Wiele bibliotek matematycznych korzysta z NumPy, w tym również wykorzystane w tym projekcie biblioteki pyrr i PyOpenGL. Biblioteki NumPy użyliśmy do implementacji kwaternionów i grafiki 3D.

4.3. pyrr

Kolejną ważną biblioteką w projekcie jest biblioteka pyrr, która dostarcza implementację matematyki wektorowej oraz operacji na macierzach [54], wy-

korzystując do tego bibliotekę NumPy. Została użyta do zaprogramowania fragmentów aplikacji implementujących grafikę 3D.

4.4. PyOpenGL

Istnieje kilka silników oraz bibliotek graficznych obsługujących grafikę 3D w języku Python, jak np. vPython czy Panda3D. Zdecydowaliśmy się użyć pythonowej implementacji OpenGL (PyOpenGL) [55], z kilku powodów.

OpenGL jest bardzo rozbudowaną technologią, ale grafika potrzebna dla tego projektu była na tyle nieskomplikowana, że użycie niskopoziomowej technologii takiej jak OpenGL nie stanowiło problemu.

Ze względu na popularność tej technologii, zarówno oficjalna dokumentacja jest bardzo rozbudowana [31, 32], jak również dostępne są liczne nieoficjalne materiały na jej temat.

Dodatkowo, OpenGL pozwala na bezpośrednią kontrolę programistyczną nad procesem tworzenia grafiki i chcieliśmy nabrać większego doświadczenia w stosowaniu tej technologii.

4.5. pyglet

Biblioteka pyglet dostarcza narzędzi ułatwiających tworzenie interaktywnych okien, inicjalizację kontekstu OpenGL oraz rysowanie grafiki 2D [56], co jest przydatne do tworzenia GUI (Graficznego Interfejsu Użytkownika).

4.6. pickle

Biblioteka pickle dostarcza proste w użyciu narzędzia pozwalające na serializację i deserializację, tj. zapisywanie i odczyt, obiektów pythonowych w postaci plików binarnych [57]. Użycie biblioteki pickle znacznie ułatwia zapis i odczyt dowolnych obiektów bądź struktur danych w Pythonie, ponadto zapis binarny pozwala na stworzenie mniejszych plików i szybsze operacje zapisu lub odczytu. Biblioteki pickle użyto do zapisywania i odczytu stanów macierzy po skończonej symulacji, jak również do zapisu i odczytu prostych modeli 3D używanych do wyświetlania podglądu macierzy.

Warto zauważyć, że użycie pickle ma pewne wady, które odkryliśmy podczas rozwoju projektu. Przede wszystkim, jeśli zapisujemy zdefiniowane przez nas klasy, użycie pickle może utrudnić refaktoryzację programu. Jeśli zmienimy interfejs, nazwę klasy bądź nazwy atrybutów, pliki zapisane przed zmianami mogą stać się niekompatybilne z obecną wersją programu. Można obejść to ograniczenie, zostawiając w projekcie wcześniejszą wersję implementacji danej klasy i pisząc odpowiedni skrypt konwertujący stare pliki na nową wersję klasy.

5. Podstawy OpenGL

Ten rozdział jest poświęcony ogólnemu opisowi technologii OpenGL i jej zastosowania w kontekście opisywanego projektu. OpenGL jest niezwykle rozbudowaną technologią, dlatego poniższy opis będzie uproszczony oraz ograniczony do elementów technologii zastosowanych w projekcie. W oficjalnej dokumentacji OpenGL [31] pojawiają się terminy, które nie są tam zdefiniowane, dlatego ich wyjaśnienia będą oparte o nieoficjalne źródła.

Fragmenty kodu z tego rozdziału stanowią przykłady mające na celu pokazanie pewnych elementów OpenGL oraz algorytmów zastosowanych w programie. Dla zachowania czytelności przykładów, fragmenty te są często przerobione lub uproszczone w stosunku do rzeczywistego kodu źródłowego aplikacji.

5.1. OpenGL

OpenGL to specyfikacja otwartego i uniwersalnego API do tworzenia grafiki [36, 34]. Biblioteki implementujące OpenGL pośredniczą w komunikacji pomiędzy programem aplikacji a sterownikiem karty graficznej. W opisywanym projekcie zastosowano bibliotekę PyOpenGL [55].

OpenGL stosuje architekturę klient-serwer [35]. Aplikacje korzystające z OpenGL, takie jak nasza, to aplikacje klienckie używające dostarczonych przez OpenGL funkcji do wysyłania poleceń serwerowi. Implementacja programu serwera zależy od danego środowiska, przede wszystkim od zainstalowanej na nim karty graficznej lub innego sprzętu. Serwer OpenGL zazwyczaj działa na tym samym komputerze co aplikacja kliencka.

Serwer OpenGL zachowuje się jak maszyna stanów skończonych [35, 37]. Aplikacja kliencka uruchamia tę maszynę w jakimś domyślnym stanie, zależnym od implementacji serwera i może następnie modyfikować ten stan poprzez uruchamianie funkcji dostarczonych przez API OpenGL. Załóżmy dla przykładu, że aplikacja kliencka chce narysować dwa identyczne obiekty, każdy w innej pozycji wewnątrz okienka. Aplikacja najpierw przesyła serwerowi dane wspólne dla obu obiektów, później wysyła dane dotyczące pozycji pierwszego obiektu i polecenie rysowania, następnie wysyła dane dotyczące pozycji drugiego obiektu i ponownie polecenie rysowania. Wszystkie dane oraz ustawienia, które były takie same dla obu obiektów, nie musiały być ponownie wysłane. Serwer pamięta wszystkie dane, które mu przesłaliśmy, dopóki nie nadpiszemy ich nowymi danymi lub nie każemy ich usunąć.

5.2. Warstwa graficzna symulatora

W naszej aplikacji zastosowano bardzo minimalistyczne podejście do obsługi grafiki 3D. Okienko aplikacji zawiera scenę, a scena przedstawia stan molekuł wewnątrz sieci prostej kubicznej po zakończonej symulacji. Każda molekuła jest reprezentowana przez model 3D, obrazujący spin molekuły (model Isinga) lub orientację molekuły (model Lebwohla-Lashera, model molekuł dwuosiowych). Więcej informacji o modelach 3D w rozdziale 5.3.

Przed narysowaniem molekuły trzeba przekazać serwerowi dane o modelu. Jak już wspomniano w rozdziale 5.1, jeśli molekuły są reprezentowane przez ten sam model 3D, to danych modelu nie musimy przesyłać ponownie dla każdej molekuły. Więcej o tym w jaki sposób przesyłamy te dane w rozdziale 5.4.

Każda z tych molekuł będzie miała własną pozycję i orientację wewnątrz okienka. Aby ustalić ich pozycję na ekranie, program będzie musiał przeprowadzić szereg przekształceń geometrycznych opisanych specjalnymi macierzami: przekształcenia związane z tym, jaką pozycję w przestrzeni sceny zajmuje dana molekuła, z jakiej perspektywy patrzy na scenę obserwator i w końcu, przekształcenie związane z rzutowaniem obrazu z perspektywy obserwatora na powierzchnię okienka. Więcej na temat wspomnianych przekształceń w rozdziale 5.5.

Po przesłaniu do serwera danych i wywołaniu funkcji odpowiedzialnej za rysowanie, resztę pracy przejmuje potok graficzny [38]. Większość potoku graficznego jest wykonywana po stronie serwera, zwykle bezpośrednio na karcie graficznej i z poziomu aplikacji nie mamy do niego bezpośredniego dostępu. Ważny wyjątek od tej reguły stanowią shadery, odpowiadające za programowalne elementy potoku [37, 38]. Więcej na temat potoku graficznego i shaderów w rozdziale 5.6.

5.3. Modele

Model 3D zawiera dane pozwalające na odtworzenie w komputerze geometrii i topologii trójwymiarowej bryły. Modele 3D można podzielić na dwie ogólne kategorie: modelujące wyłącznie powierzchnię bryły (ang. *shell/boundary model*) oraz modelujące zarówno powierzchnię jak i wnętrze bryły (ang. *solid model*) [41]. Modeli typu 'solid' używa się m.in. w symulacjach medycznych, zaawansowanych modelach fizycznych oraz programach pozwalających na tworzenie nowych modeli 3D poprzez cyfrowe "rzeźbienie" początkowej bryły. Modeli powierzchniowych używa się wtedy, kiedy wystarczy sama reprezentacja graficzna danego obiektu. Ponieważ nasz symulator używa modeli 3D wyłącznie do podglądu stanu macierzy, a nie do obliczeń podczas symulacji, model powierzchniowy molekuł całkowicie wystarczy. Jest on również wydajniejszy i łatwiejszy w użyciu.

Istnieje kilka sposobów reprezentowania modeli powierzchniowych w komputerze, najprostszym i najbardziej powszechnym z nich jest użycie tzw. siatki wielokątów (ang. *polygonal mesh*) [42, 43]. Siatki wielokątów składają się z wierzchołków połączonych krawędziami w wielokątne ścianki. Każdy

wielokąt można podzielić na trójkąty, a wiele operacji z zakresu geometrii obliczeniowej jest łatwiejszych do przeprowadzenia na trójkątach. Z tych powodów bardzo wydajną i elastyczną reprezentacją modelu powierzchniowego jest siatka trójkątów.

Istnieje wiele sposobów na opisanie struktury i właściwości siatki trójkątów w komputerze. Dosyć typowym podejściem jest stworzenie zbioru wierzchołków oraz zbioru informacji na temat tego w jaki sposób wierzchołki są połączone. Każdy wierzchołek w zbiorze ma zestaw zdefiniowanych liczbowo atrybutów (np. pozycję w przestrzeni, wektor normalny, współrzędne tekstury, etc.). Modele 3D używają własnego (lokalnego) układu współrzędnych do opisywania położeń wierzchołków [44, 45]. W przypadku modeli zastosowanych w projekcie, początek układu współrzędnych pokrywa się ze środkiem ciężkości ciała.

Na potrzeby tego projektu, do reprezentacji modelu 3D, użyta została struktura zoptymalizowana pod kątem prostej i wydajnej współpracy z biblioteką PyOpenGL. Struktura jest złożona z dwóch tabel będących obiektami array dostarczonymi przez bibliotekę NumPy. Pierwsza tabela zawiera dane wierzchołków, druga dane, na podstawie których OpenGL połączy wierzchołki w ściany figury. Dane z obu tabel są przesyłane do odpowiednich buforów.

5.4. Bufory

Bufory to obiekty tworzone przez OpenGL, pozwalające na szybkie przesyłanie dużej liczby danych do pamięci karty graficznej [46]. Każdy bufor przechowuje dane w postaci niesformatowanego bloku bajtów (w dokumentacji jest to nazwane *tablicą niesformatowanej pamięci*) oraz informacje na temat tego, jak ten blok powinien być interpretowany.

Bufory tworzymy przy pomocy funkcji glGenBuffers(n) [31], gdzie n jest liczbą buforów do utworzenia. Funkcja zwraca nam numer identyfikacyjny buforu bądź listę numerów ID, jeśli $n > 1$. Dokumentacja OpenGl nazywa zwracaną wartość *nazwą buforu*.

Aby użyć buforu trzeba go najpierw powiązać z odpowiednim targetem przy pomocy funkcji glBindBuffer(target, **buffer**) [31]. Target jest określeniem przeznaczenia buforu, tj. określa w jaki sposób OpenGL użyje danych zawartych w buforze [46]. Parametry funkcji:
— target - (typ GLenum) target, do którego bufor ma być podłączony,
— **buffer** - (typ int) *nazwa buforu*.
W trakcie działania programu można zmieniać, który z buforów jest podpięty pod dany target. Wygodnie jest myśleć o zmiennej target jako definiującej "rodzaj" bufora. W naszej aplikacji zastosowano dwa "rodzaje" buforów: GL_ARRAY_BUFFER przechowuje dane o wierzchołkach, natomiast GL_ELEMENT_ARRAY_BUFFER zawiera dane, na podstawie których wierzchołki z GL_ARRAY_BUFFER są łączone w ścianki figury.

Po podpięciu buforów do odpowiednich targetów, można załadować do nich dane przy pomocy funkcji glBufferData(target, size, data, usage) [31]. Parametry funkcji:

— target - (GLenum) target, do którego dane zostaną załadowane,
— size - (int) rozmiar buforowanych danych podany jako ilość bajtów,
— data - wskaźnik do obiektu array (numpy.array) przechowującego dane do załadowania,
— usage - (GLenum) sugerowana strategia dostępu do pamięci buforu.

Zmienna usage, podobnie jak target, przyjmuje pewne predefiniowane wartości stałe. Nazwy tych wartości składają się z dwóch wyrazów [46]. Pierwszy definiuje to jak często spodziewamy się modyfikować zawartość bufora:
— STATIC - nigdy albo bardzo rzadko,
— DYNAMIC - w miarę często,
— STREAM - po każdym albo prawie każdym użyciu danych.

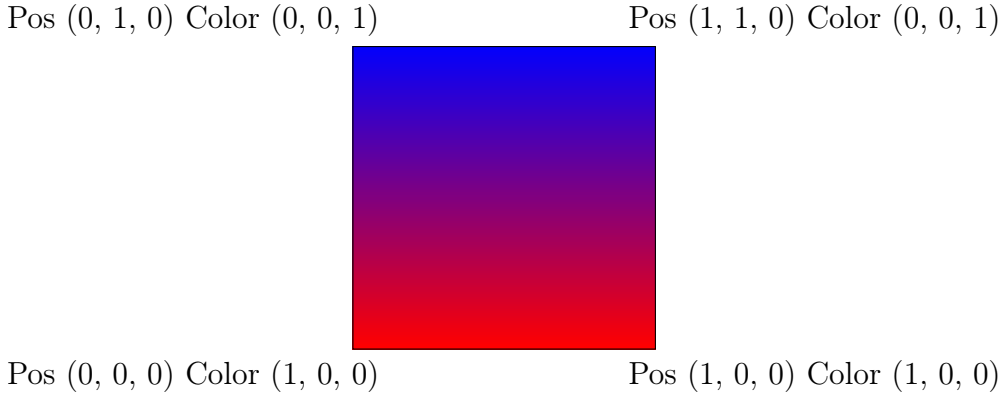
Drugi wyraz definiuje spodziewany podział praw do odczytu i zapisu danych pomiędzy aplikacją kliencką a serwerem OpenGL:
— DRAW - aplikacja kliencka przesyła dane do bufora, a serwer z niego czyta,
— READ - serwer zapisuje dane do bufora, a aplikacja kliencka z niego czyta,
— COPY - bufor do użytku przez oprogramowanie działające na serwerze, aplikacja kliencka ani w nim nic nie zapisuje, ani z niego nie czyta.

Na przykład, usage o wartości GL_DYNAMIC_DRAW (takiej jaką stosujemy w obu naszych buforach), sugeruje serwerowi OpenGL, że dane będą zapisywane w buforze przez aplikację kliencką i będą co jakiś czas nadpisywane.

5.4.1. Bufor wierzchołków

Bufor wierzchołków (GL_ARRAY_BUFFER) [47] zawiera wszystkie dane dotyczące wierzchołków, z których składa się dana figura. W naszej aplikacji, dane pojedynczego wierzchołka składają się z informacji o jego pozycji oraz informacji o jego kolorze. Pozycja wierzchołka jest zakodowana przy użyciu trzech liczb zmiennoprzecinkowych, jako jego współrzędne x, y, z. Kolor wierzchołka jest również zakodowany przy użyciu trzech liczb zmiennoprzecinkowych, odpowiadających wartościom red, green i blue w formacie rgb. Kolor ściany jest ustalany na podstawie kolorów wierzchołków, z których się składa.

Załóżmy, że chcemy narysować kwadrat przedstawiony na rysunku 5.1.



Rysunek 5.1. Kwadrat (pozycje wierzchołków i kolory).

Potrzebujemy załadować do bufora dane wierzchołków, które powinny wyglądać w następujący sposób:

```
[0.0, 0.0, 0.0,  1.0, 0.0, 0.0,
 1.0, 0.0, 0.0,  1.0, 0.0, 0.0,
 1.0, 1.0, 0.0,  0.0, 0.0, 1.0,
 0.0, 1.0, 0.0,  0.0, 0.0, 1.0]
```

Zanim OpenGL będzie mógł cokolwiek narysować, musi wiedzieć jak zinterpretować, które dane w tabeli odpowiadają pozycji, a które kolorowi. Poniższy kod ilustruje w jaki sposób przekazujemy te informacje.

Listing 5.1. Definiowanie wskaźników atrybutów.

```
# position
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
    float_size * 6, ctypes.c_void_p(0))
glEnableVertexAttribArray(0)

# colors
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
    float_size * 6, ctypes.c_void_p(float_size * 3))

glEnableVertexAttribArray(1)
```

Funkcja glEnableVertexAttribArray(index) [31] informuje OpenGL, że atrybut o danym indeksie będzie używany w shaderze. Natomiast funkcja glVertexAttrib–Pointer() [31] definiuje, które bajty z bufora mają zostać przypisane do danego atrybutu.

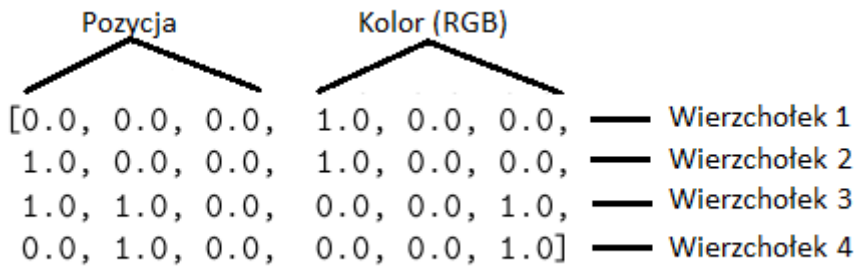
glVertexAttribPointer(index, size, **type**, normalized, stride, pointer)

— index - (int) indeks atrybutu, do którego należy przesłać dane,
— size - (int) z ilu wartości składa się atrybut,
— **type** - (GLenum) typ zmiennej opisującej wartości atrybutu (np. GL_FLOAT dla liczb zmiennoprzecinkowych),
— normalized - (bool) czy dane w buforze powinny być znormalizowane przed użyciem,
— stride - (int) w naszym przypadku, ile bajtów zajmują dane pojedynczego wierzchołka,
— pointer - (pointer do wartości int) ile bajtów zajmują wartości poprzednich atrybutów.

W powyższym przykładzie atrybut color jest definiowany poprzez wywołanie funkcji glVertexAttribPointer() z następującymi parametrami:

— index=1 - w programie shadera atrybut color ma indeks 1,
— size=3 - atrybut color składa się z trzech liczb,
— **type**=GL_FLOAT - są to liczby zmiennoprzecinkowe,
— normalized=GL_FALSE - nie stosujemy mechanizmu normalizacji,
— stride=float_size $*$ 6 - dane jednego wierzchołka są opisywane łącznie przez 6 liczb zmiennoprzecinkowych, więc razem zajmują one 6 razy float_size bajtów,
— pointer=ctypes.c_void_p(float_size $*$ 3) - wartości poprzednich atrybutów zajmują razem 3 razy float_size bajtów.

Po wykonaniu powyższej konfiguracji, dane w buforze wierzchołków będą interpretowane w następujący sposób:



Rysunek 5.2. Interpretacja danych z bufora.

5.4.2. Bufor indeksów

Bufor indeksów (GL_ELEMENT_ARRAY_BUFFER) [47] zawiera dane, na podstawie których program łączy wierzchołki bryły w ścianki. OpenGL stosuje różne tryby interpretacji zawartości bufora indeksów [31]. Najczęściej stosowanym trybem interpretacji jest GL_TRIANGLES, w którym pierwsze trzy liczby w buforze to indeksy trzech wierzchołków, które tworzą trójkątną ściankę, następne trzy liczby odpowiadają indeksom trzech wierzchołków następnego trójkąta, itd. W ten sposób program interpretuje wierzchołki jako bryłę stworzoną z trójkątnych ścianek. Istnieją też inne sposoby interpretacji bufora indeksów, jak GL_QUADS - 4 kolejne indeksy to indeksy wierzchołków tworzących czworokąt, GL_TRIANGLE_FAN - wierzchołki tworzą wachlarz trójkątów, itd. Te tryby pozwalają na zmniejszenie ilości przesyłanych danych.

Można też interpretować indeksy w buforze jako końce linii (GL_LINES), jeśli chcemy narysować siatkę krawędzi zamiast bryły złożonej ze ścian lub też jako zbiór punktów (GL_POINTS), jeśli chcemy narysować same punkty.

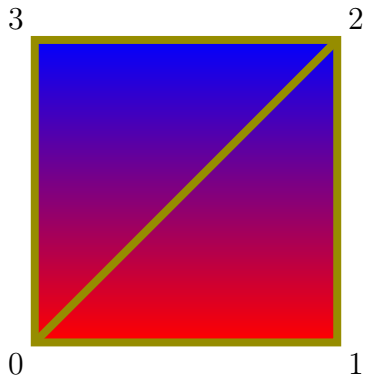
Załóżmy dla przykładu, że chcemy narysować kwadrat korzystając z pierwszych czterech wierzchołków w buforze wierzchołków w trybie GL_TRIANGLES. Przesyłamy do bufora indeksów następujące dane: [0, 1, 2, 0, 2, 3]. Następnie wywołujemy funkcję rysującą.

```
glDrawElements(GL_TRIANGLES, len(index_data),GL_UNSIGNED_INT, None)
```

W wyniku otrzymujemy kwadrat na rysunku 5.3. Granice trójkątów są wyróżnione na rysunku w celach demonstracyjnych. Jeśli chcielibyśmy narysować je w trybie GL_LINES, bufor indeksów miałby następującą zawartość: [0, 1, 1, 2, 2, 3, 3, 0, 0, 2].

Aby narysować ten sam kwadrat w trybie GL_QUADS bufor indeksów musiałby mieć następującą zawartość: [0, 1, 2, 3]. Czasami takie podejście się opłaca, np. jeśli zysk z wysyłania mniejszej ilości danych do karty jest istotny. W naszym programie tryb GL_TRIANGLES jest używany uniwersalnie, ze względu na prostotę tego rozwiązania.

Podsumowując, aplikacja rysującą musi:



Rysunek 5.3. Kwadrat (indeksy i podział na ścianki).

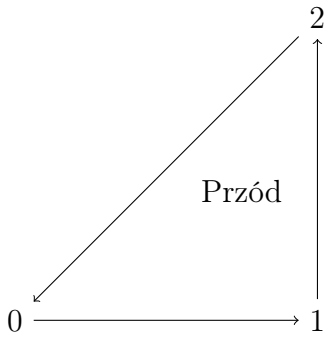
— załadować odpowiednie dane do obu buforów,
— zdefiniować podział danych w buforze wierzchołków na atrybuty,
— wywołać odpowiednią funkcję rysującą, w naszym przypadku glDrawElements().

Po wykonaniu powyższych kroków, pracę przejmuje potok graficzny i działające na serwerze GL shadery.

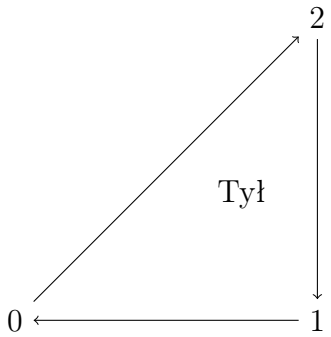
5.4.3. Kolejność indeksów

Domyślnie kolejność indeksów wyznacza, która strona ściany jest przodem, a która tyłem [48]. To, jakie znaczenia ma przód czy tył ściany zależy od reszty kodu w aplikacji. Dobrą praktyką jest zachowanie odpowiedniej kolejności, żeby ściany bryły zawsze przodem na zewnątrz. Pozwala to uniknąć problemów w dalszym rozwoju aplikacji.

Indeksy ściany oglądanej od przodu powinny układać się w kolejności przeciwnej do kierunku ruchu wskazówek zegara [48].



index_data = [0, 1, 2]

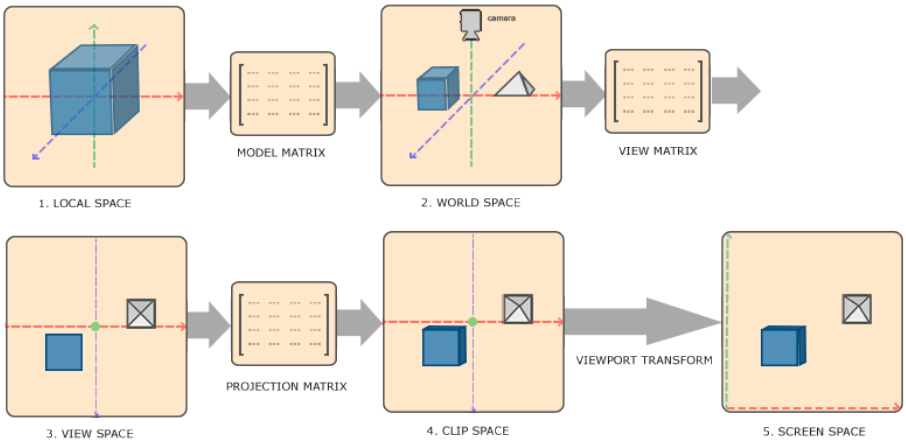


index_data = [0, 2, 1]

5.5. Transformacje geometryczne i macierze

Jak już wspomniano w rozdziale 5.3, modele 3D przechowują informacje o pozycjach wierzchołków względem jakiegoś arbitralnie wybranego punktu. W celu zbudowania z tych modeli sceny i wyświetlenia jej na ekranie,

należy przeprowadzić szereg geometrycznych przekształceń, przedstawionych na rysunku 5.4.



Rysunek 5.4. Proces przekształceń geometrycznych, od przestrzeni modelu do obrazu na ekranie. Ilustracja zaczerpnięta z *learnopengl.com* [44]

.

Przed wyjaśnieniem poszczególnych systemów koordynat warto zaznaczyć, że przejścia pomiędzy większością z nich są wyrażone przy pomocy macierzy 4×4, co wymaga od nas zastosowania 4-wymiarowych wektorów. Dowolny wektor 3D o współrzędnych $[x, y, z]$, można zapisać w postaci $V = [x * w, y * w, z * w, w]$, gdzie $x, y, z, w \in \mathbb{R}$. Domyślnie $w = 1.0$.

Wyjaśnienia zostały napisane w oparciu o artykuły zaczerpnięte z OpenGL Wiki [32], *learnopengl.com* [44], oraz o własne doświadczenie autora pracy.

1. Przestrzeń lokalna (ang. *local space*)

   Zwana również przestrzenią modelu, opisuje położenie wierzchołków-modelu względem pewnego arbitralnie wybranego punktu. Wygodne jest ustawienie tego punktu jako środka figury.

   Przestrzeń lokalna jest wygodną przestrzenią do definiowania oraz modyfikacji kształtu modelu.Wynikiem takich operacji jest zmiana położeń wierzchołków modelu względem siebie.
2. Przestrzeń globalna (ang. *global space*)

   Zwana również przestrzenią sceny, opisuje położenie wierzchołków figury względem pozostałych obiektów wewnątrz danej sceny 3D.

   Przekształcenie opisujące przejście pomiędzy systemem koordynat przestrzeni lokalnej a systemem koordynat przestrzeni globalnej jest wyrażone za pomocą tzw. macierzy modelu (ang. *model matrix*). Macierz modelu opisuje złożenie rotacji modelu oraz jego translacji względem początku globalnego układu współrzędnych. Innymi słowy, macierz modelu opisuje orientację oraz położenie modelu wewnątrz sceny.

   Dla uniknięcia kolizji nazw z modelami 3D, macierz modelu jest często nazywana w programach macierzą transformacji (ang. *transform matrix*).

   Przestrzeń globalna jest wygodną przestrzenią do wykonywania operacji związanych z poruszaniem oraz rotacją obiektów wewnątrz sceny.

Każdy obiekt w scenie ma swoją własną macierz modelu. Operacje przeprowadzone w przestrzeni globalnej prowadzą do modyfikacji tej macierzy dla danego obiektu.

3. Przestrzeń widoku (ang. *view space*)

   Zwana również przestrzenią kamery, opisuje położenie obiektów (czy też ich wierzchołków), względem obserwatora zwanego często kamerą. Macierz widoku (ang. *view matrix*) opisuje przejście z globalnego układu współrzędnych do układu współrzędnych kamery.

   Manipulacje macierzą widoku pozwalają na oglądanie tej samej sceny z różnych punktów widzenia.

4. Przestrzeń przycinania (ang. *clip space*)

   Pośrednia przestrzeń, obsługiwana automatycznie przez OpenGL. Upraszcza m.in. rzutowanie perspektywiczne, jak również 'obcinanie' elementów sceny znajdujących się poza polem widzenia kamery. Obcinanie jest procesem optymalizującym działanie potoku graficznego (o potoku graficznym w rozdziale 5.6)

   Przejście z przestrzeni widoku do przestrzeni przycinania jest wyrażone za pomocą macierzy rzutowania. Macierz rzutowania określa przede wszystkim jaki zakres pozycji z przestrzeni kamery będzie widoczny wewnątrz okienka. Opisuje też przekształcenia związane z samym rzutowaniem. Jeśli zastosowano rzutowanie ortogonalne, to współrzędne wierzchołków będą miały współrzędne w postaci $V = [x, y, z, w]$, gdzie $w = 1.0$. Jeśli natomiast zastosowano rzutowanie perspektywiczne, współrzędna w może być różna od 1.0.

   Przestrzeń przycinania definiuje, które wierzchołki są wewnątrz tzw. obszaru widoczności (ang. *viewing volume*), na podstawie poniższego układu nierówności [49]. Wierzchołki, których współrzędne spełniają ten układ, są wewnątrz obszaru widoczności.

$$\begin{aligned} -w \leqslant x \leqslant w \\ -w \leqslant y \leqslant w \\ -w \leqslant z \leqslant w \end{aligned} \tag{5.1}$$

5. Przestrzeń viewportu (na rysunku "screen space")

   W naszym przypadku tożsama z systemem współrzędnych wewnątrz okienka. Transformacja współrzędnych z przestrzeni przycinania do przestrzeni okna jest obsługiwana automatycznie przez OpenGL.

5.5.1. Wykorzystanie macierzy

Wspomniane powyżej przekształcenia wierzchołków, za wyjątkiem viewport transform, są typowo wykonywane po stronie serwera OpenGL przez tzw. shader wierzchołków (więcej o działaniu shadera wierzchołków w rozdziale 5.6). Aby shader mógł wykonać stosowne obliczenia, należy mu wysłać dane z macierzy do specjalnej zmiennej typu 'uniform mat4'. Słowo kluczowe 'uniform' oznacza, że w przeciwieństwie do atrybutów, których wartości są potencjalnie różne dla każdego przetwarzanego wierzchołka, dane zawarte w 'uniform' będą takie same dla wszystkich wierzchołków.

Danych 'uniform' nie wysyłamy przy pomocy buforów, zamiast tego wykorzystujemy parę specjalnych funkcji glGetUniformLocation(program, name) oraz glUniformMatrix4fv(location, count, transpose, value).

Funkcja glGetUniformLocation(program, name) [31] zwraca wskaźnik logiczny (tj. unikatowy numer ID) zmiennej o podanej nazwie wewnątrz podanego programu shadera.

Parametry glGetUniformLocation(program, name):

— program - (int) wskaźnik logiczny do programu, zawierającego szukaną zmienną
— name - (string) nazwa szukanej zmiennej

Funkcja glUniformMatrix4fv(location, count, transpose, value) [31] służy do przesłania macierzy 4×4 składającej się z liczb zmiennoprzecinkowych do podanego adresu logicznego.

Parametry glUniformMatrix4fv(location, count, transpose, value):

— location - (int) wskaźnik logiczny do zmiennej, do której chcemy przesłać dane,
— count - (int) ilość macierzy, które przesyłamy, w naszym przypadku 1. Jeśli zmienna docelowa byłaby tablicą macierzy, można by przesłać jej więcej macierzy,
— transpose - (bool) jeśli prawda, macierz jest automatycznie transponowana przed wysłaniem,
— value - wskaźnik do macierzy lub tablicy macierzy.

Podczas rysowania dowolnego modelu, ważne jest aby współrzędne wierzchołków zostały przekształcone do przestrzeni przycinania. Nie jest natomiast konieczne, żeby to przekształcenie rozbijać na etapy, dlatego można wysłać dwie macierze albo nawet jedną macierz, zamiast wszystkich trzech - Model, View, oraz Projection. W naszym projekcie przekazywana jest jedna macierz ModelViewProjection opisująca złożenie wszystkich trzech przekształceń, ModelViewProjection = Model $*$ View $*$ Projection.

Listing 5.2. Przekazywanie macierzy MVP.

```
mvp_loc = glGetUniformLocation(program, "mvp")

glUniformMatrix4fv(mvp_loc, 1, GL_FALSE, mvp_matrix)
```

Zawartość macierzy jest generowana przy użyciu funkcji dostarczonych przez bibliotekę pyrr.

Jak już wspomniano, macierz modelu (w programie nazywana macierzą transformacji) jest wyliczana na podstawie pozycji oraz orientacji danego obiektu wewnątrz sceny.

Listing 5.3. Wyliczanie zawartości macierzy modelu (tranfsorm).

```
rotation_matrix =
    matrix44.create_from_quaternion(self.orientation)
translation_matrix =
    matrix44.create_from_translation(self.position)

self.transform_matrix = matrix44.multiply(
    rotation_matrix, translation_matrix)
```

Natomiast metody do tworzenia macierzy widoku oraz projekcji są dostarczone bezpośrednio przez bibliotekę pyrr.

Listing 5.4. Wyliczanie zawartość macierzy view oraz projection.

```
view = matrix44.create_look_at(
    camera_pos, target_pos, camera_up)

projection = matrix44.create_perspective_projection_matrix(
    45.0, aspect_ratio, 0.1, 100.0)
```

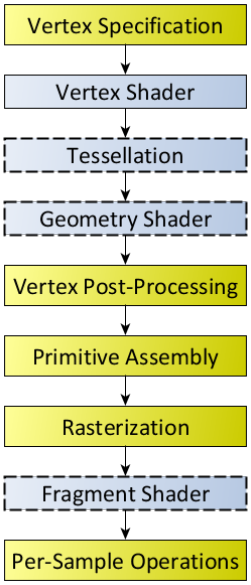
Następnie podczas rysowania wszystkie macierze zostają odpowiednio przemnożone przez siebie i wysłane do programu shadera jako macierz MVP (ModelViewProjection).

Listing 5.5. Wyliczanie macierzy MVP.

```
vp_matrix = matrix44.multiply(view, projection)

for particle in particle_list:
    mvp_matrix = matrix44.multiply(
        particle.get_transform(), vp_matrix)
    glUniformMatrix4fv(mvp_loc, 1, GL_FALSE, mvp_matrix)
    particle.draw()
```

5.6. Potok graficzny i shadery

Potok graficzny opisuje proces renderowania grafiki przez OpenGL [38], tj. tworzenia dwuwymiarowego obrazu na ekranie, na podstawie danych opisujących trójwymiarową scenę [39]. Składa się on z szeregu etapów, część z nich jest opcjonalna. Z wyjątkiem pierwszego etapu, wszystkie etapy odbywają się po stronie serwera OpenGL.



Rysunek 5.5. Standardowy potok renderowania OpenGL. Programowalne etapy są zaznaczone na niebiesko, opcjonalne przerywaną linią. Ilustracja zaczerpnięta z artykułu *Rendering Pipeline Overview* z OpenGL Wiki [38]

.

Nasz projekt korzysta przede wszystkim z podstawowych etapów: Vertex Specification, VertexShader, Vertex Post-Processing, Primitive Assembly, Rasterization, Fragment Shader, Per-Sample Operations.

Opis poszczególnych etapów potoku graficznego został napisany na podstawie artykułu *Rendering Pipeline Overview* z OpenGL Wiki [38] oraz artykułów dotyczących poszczególnych etapów wyżej wymienionego potoku.

1. Vertex Specification

   Aplikacja kliencka przygotowuje i wysyła uporządkowaną listę wierzchołków wraz z pozostałymi danymi (jak np. macierz ModelViewProjection) do potoku.
2. Vertex Shader

   Każdy wierzchołek jest osobno przetwarzany przez program shadera wierzchołków. Atrybuty wierzchołków trafiają do zmiennych w tym programie. Pozostałe dane trafiają do programu shadera w postaci zmiennych typu 'uniform', dane 'uniform' są takie same dla każdego przetwarzanego wierzchołka. Program wykonuje obliczenia i zwraca zdefiniowany przez programistę zestaw danych wyjściowych oraz pewne predefiniowane przez OpenGL dane wyjściowe, w tym zawsze ostateczną pozycję wierzchołka (zmienna gl_position) w tzw. przestrzeni przycinania (ang. *clip space*).

   Współrzędne x, y, z w gl_position muszą leżeć wewnątrz określonego zakresu wartości, aby wierzchołek był widoczny wewnątrz okienka (patrz wzór 5.1 w podrozdziale 5.5).
3. Tesselation

   Opcjonalny krok wykonywany tylko jeśli shader teselacji został zdefiniowany.

4. Geometry Shader

   Opcjonalny krok, wykonywany tylko jeśli shader geometrii został zdefiniowany.

5. Vertex Post-Processing

   W tym etapie może zostać wykonanych kilka różnych operacji w zależności od tego, które z wcześniejszych opcjonalnych etapów zostały wykonane.

   Jedną z operacji, które zostaną wykonane niezależnie od opcjonalnych etapów, jest przycinanie (ang. *clipping*). Sposób działania przycinania zależy od tego co aktualnie rysujemy. Jeśli rysujemy punkty, to punkty leżące poza widocznym zakresem (patrz punkt 2), zostaną odrzucone. Jeśli rysujemy linie, to linie leżące całkowicie poza widocznym zakresem zostaną odrzucone, a linie leżące częściowo poza zakresem zostaną skrócone, tak aby się w nim mieściły. Jeśli rysujemy trójkąty (na tym etapie każda ściana jest reprezentowana przez jeden lub więcej trójkątów), trójkąty leżące poza zakresem zostają odrzucone, natomiast te leżące częściowo poza zakresem zostają rozbite na mniejsze trójkąty pokrywające część trójkąta leżącą wewnątrz zakresu. Celem przycinania jest optymalizacja, poprzez odrzucenie prymitywów (punktów, linii, bądź trójkątów) leżących poza widocznym obszarem sceny, aby nie marnować na nie mocy obliczniowej w późniejszych etapach potoku.

   Ostatnim krokiem wykonywanym podczas tego etapu jest transformacja z przestrzeni przycinania do przestrzeni viewportu, tj. do systemu koordynat wewnątrz docelowego okna.

6. Rasterization

   Wszystkie prymitywy są rozbijane na tzw. fragmenty. Fragment reprezentuje niewielki kawałek prymitywu (w naszym przypadku trójkątnej ścianki). To na ile fragmentów jest rozbity prymityw zależy od wielu czynników, ale przede wszystkim od ilości pikseli na ekranie, które pokrywają się z jego powierzchnią. Prymityw zostanie rozbity na co najmniej jeden fragment na każdy taki piksel [40].

   Dane fragmentu są wyliczane poprzez interpolację danych wierzchołków, z których składa się prymityw, do którego fragment należy.

7. Fragment Shader

   Program shadera operuje na każdym fragmencie, w sposób analogiczny do shadera wierzchołków. Na podstawie danych wejściowych ostatecznie ustala pozycję oraz kolor danego fragmentu.

8. Pre-Sample Operations

   W tym etapie ostatecznie ustala się kolor pikseli na ekranie, na podstawie tego, które okienko ma do danego piksela dostęp, tj. które okno jest w tym punkcie monitora na wierzchu. Jeśli okno ma do danego piksela dostęp, to jego kolor jest wyznaczany na podstawie mieszania kolorów fragmentów znajdujących się pod tym pikselem. Załóżmy, że fragment ma nieprzezroczysty kolor, w takim wypadku kolor piksela to po prostu kolor fragmentu, który był pod nim najbliżej. Jeśli kolory fragmentów były częściowo przezroczyste, to kolory fragmentów są odpowiednio mieszane.

5.6.1. Shader wierzchołków

W naszym programie zastosowano najprostsze możliwe shadery, gdyż bardziej zaawansowana grafika nie była potrzebna. Shader wierzchołków jest odpowiedzialny wyłącznie za wyznaczanie pozycji wierzchołka w przestrzeni przycinania oraz przekazanie tej pozycji wraz z kolorem wierzchołka odpowiednio do wbudowanych zmiennych gl_position i v_Color.

Wartość gl_position jest wyliczana poprzez mnożenie wysłanej do shadera macierzy MVP przez wektor pozycji wierzchołka (patrz rozdział 5.5). Wartość koloru jest przekazywana bezpośrednio z atrybutu color do zmiennej v_color, przy okazji konwertowana z formatu RGB do formatu RGBA.

Listing 5.6. Shader wierzchołków.

```
#version 330
in layout(location = 0) vec3 position;
in layout(location = 1) vec3 color;

uniform mat4 mvp;

out vec4 v_color;

void main()
{
gl_Position = mvp * vec4(position, 1.0f);
v_color = vec4(color, 1.0);
}
```

5.6.2. Shader fragmentów

Shader fragmentów przypisuje kolor wyliczony podczas rasteryzacji (ang. *rasterization*) do zmiennej wbudowanej gl_FragColor, w ten sposób ostatecznie ustalając kolor fragmentu. Teoretycznie, uruchamianie shadera fragmentów jest opcjonalnym krokiem w potoku graficznym, jednakże bez działającego shadera fragmentów, fragmenty nie zostaną pokolorowane, co jest niezbędne przy wyświetlaniu grafiki.

Listing 5.7. Shader fragmentów.

```
varying vec4 v_color;

void main()
{
gl_FragColor = v_color;
}
```

5.7. Generacja modeli 3D

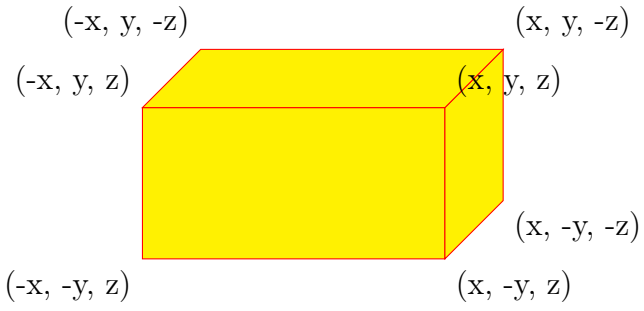
W naszym projekcie potrzebne są modele 3D reprezentujące spiny (plus i minus) w modelu Isinga oraz molekuły w pozostałych modelach: sferocylindry dla modelu Lebwohla-Lashera i sferopłytki dla modelu molekuł dwuosiowych.

5.7.1. Spiny jako prostopadłościany

Spin w dół może być reprezentowany jako znak minus. W przestrzeni 3D znak minus wygląda jak prostopadłościan. Natomiast spin w górę najprościej przedstawić jako znak plus, czyli dwa prostopadłościany złożone na krzyż. W celu wygenerowania prostopadłościanu potrzebne są jego rozmiary wzdłuż trzech osi oraz - w przypadku naszego programu - jego kolor.

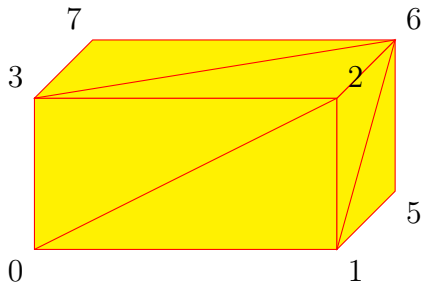
Listing 5.8. Wyznaczanie współrzędnych wierzchołków w prostopadłościanie.

```
size_x = float(size_x)
size_y = float(size_y)
size_z = float(size_z)

x = size_x / 2
y = size_y / 2
z = size_z / 2

r, g, b = color

vertex_data = [
-x, -y,  z,  r, g, b, # vertex 0
 x, -y,  z,  r, g, b, # vertex 1
 x,  y,  z,  r, g, b, # vertex 2
-x,  y,  z,  r, g, b, # vertex 3

-x, -y, -z,  r, g, b, # vertex 4
 x, -y, -z,  r, g, b, # vertex 5
 x,  y, -z,  r, g, b, # vertex 6
-x,  y, -z,  r, g, b  # vertex 7
]
```



Rysunek 5.6. Współrzędne wierzchołków prostopadłościanu. Dla zachowania czytelności, wierzchołek 4 jest przesłonięty.

Po ustaleniu parametrów wierzchołków, wystarczy wygenerować tabele indeksów. Jak już wspomniano w podrozdziale 5.4.2, w naszym programie używany jest sposób indeksowania GL_TRIANGLES, co oznacza, że każde trzy kolejne indeksy wskazują na wierzchołki tworzące jeden trójkąt. Każdą z sześciu ścian prostopadłościanu można łatwo podzielić na dwa trójkąty.



Rysunek 5.7. Podział ścian prostopadłościanu na trójkąty. Dla zachowania czytelności, wierzchołek 4 jest przesłonięty.

Wiedząc, które wierzchołki tworzą daną ścianę, można wyznaczyć stałą kolejność indeksów w buforze.

Listing 5.9. Kolejność indeksów prostopadłościanu w buforze.

```
index_data = [
0, 1, 2,  0, 2, 3,
4, 5, 1,  4, 1, 0,
4, 7, 6,  4, 6, 5,
1, 5, 6,  1, 6, 2,
0, 3, 7,  0, 7, 4,
3, 2, 6,  3, 6, 7
]
```

5.7.2. Łączenie modeli

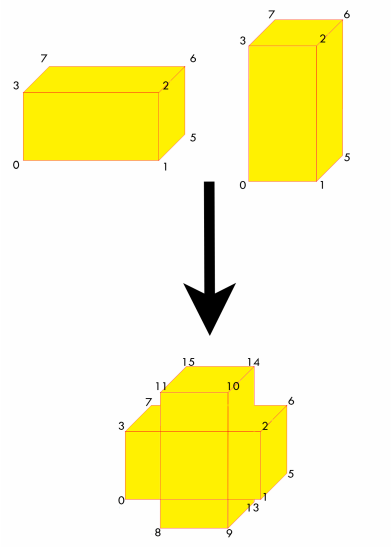
W naszym programie, łączenie modeli jest używane tylko do generowania modelu znaku plus, ale może mieć też inne potencjalne zastosowania w dalszym rozwoju projektu. Łączenie modeli jest dosyć proste, zakładając że dane wierzchołków oraz indeksów w obu modelach mają tą samą strukturę, tj. dane o wierzchołkach mają tą samą kolejność atrybutów i dane indeksów są dostosowane do tego samego sposobu interpretacji, np. obie tabele indeksów mają byćm interpretowane jako GL_TRIANGLES.

Zakładając, że powyższe warunki są spełnione i dane wierzchołków nie muszą być dodatkowo obrobione (np. modele nie muszą być obrócone ani przesunięte względem siebie przed ich złączeniem), łączenie dwóch modeli sprowadza się do kilku prostych kroków.

1. Oblicz z ilu wierzchołków składa się pierwszy model.
2. Stwórz nową tabelę wierzchołków, zawierającą wierzchołki z obu modeli (wszystkie wierzchołki modelu 1 w ich początkowej kolejności, potem wszystkie wierzchołki z modelu 2 w ich początkowej kolejności).
3. Powiększ wszystkie wartości w tabeli indeksów modelu 2 o liczbę wierzchołków w modelu 1.
4. Stwórz nową tabelę indeksów, zawierającą wszystkie indeksy z tabel dla obu modeli.
5. Zwróć nowy model zawierający nową tabelę wierzchołków i nową tablę indeksów.

Listing 5.10. Łączenie modeli 3D.

```
offset = int(model_1.vertex_data.size / 6)
vertex_data = numpy.concatenate(
    (model_1.vertex_data, model_2.vertex_data))
index_data = numpy.add(model_2.index_data, offset)
index_data = numpy.concatenate(
    (model_1.index_data, index_data))
return SimpleModel(SimpleModelData(vertex_data, index_data))
```



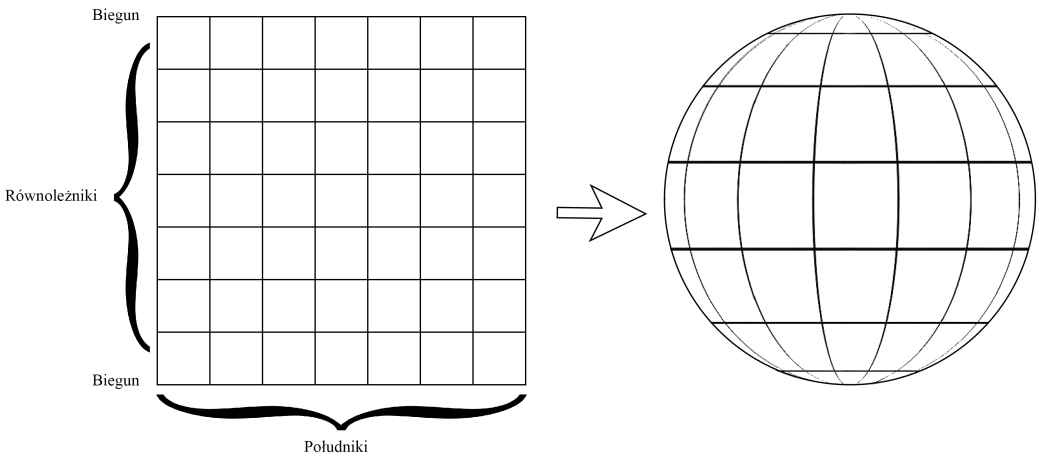
Rysunek 5.8. Łączenie modeli 3D.

5.7.3. Sfery

Funkcja generująca sfery o danym promieniu i kolorze została zaimplementowana, ale w obecnej wersji programu nie jest używana. Generowanie modelu sfery jest jednak kluczową (oraz najtrudniejszą) częścią problemu generowania sferocylindrów i sferopłytek, dlatego zasługuje na własny podrozdział.

Problem generowania sfery, tak jak i problem generowania innych modeli 3D, można zredukować do podziału figury na trójkątne ścianki. Jest kilka sposobów na zrobienie tego dla sfery. Zdecydowaliśmy się zaadaptować sposób zaczerpnięty z tutorialu *Leaning WebGL* [33], głównie ze względu na jego względną intuicyjność oraz fakt, że już kiedyś z niego korzystaliśmy.

Wygodnie jest myśleć o ścianach, które tworzą powierzchnię sfery, jako o siatce kwadratów, którą zakrzywiono w kształt sfery.



Rysunek 5.9. Sfera jako siatka kwadratów.

Linie wyznaczające krawędzie tych kwadratów odpowiadają południkom i równoleżnikom na globusie. Punkty przecięcia tych linii będą naszymi wierzchołkami. W naszej metodzie dzielimy sferę na 20 równoleżników i 20 południków. Iterujemy po wszystkich kombinacjach południków i równoleżników, aby wyznaczyć ich punkty przecięcia (nasze wierzchołki). Punkty leżące na południku 0° pokrywają się z punktami leżącymi na południku 360°, podobnie wszystkie punkty leżące na równoleżniku 0° oraz 180° zbiegają się w jednym z dwóch punktów (biegunów). Tworzymy w ten sposób kilka nadmiarowych punktów, ale taka metoda upraszcza łączenie punktów w ściany wyznaczające powierzchnię sfery.

Listing 5.11. Iterowanie po równoleżnikach i południkach w sferze.

```
steps = 20

for latNumber in range(steps + 1):
    for longNumber in range(steps + 1):
        ...
```

Kombinacja południka oraz równoleżnika, na którym leży dany punkt wyznacza nam jego współrzędne sferyczne, tj. kąty ϕ oraz θ, określające jego położenie.

Na podstawie współrzędnych sferycznych punktu, można obliczyć jego współrzędne kartezjańskie, za pomocą wzoru [50]:

$$\begin{aligned} x &= r\cos\theta\cos\phi, \\ y &= r\cos\theta\sin\phi, \\ z &= r\sin\theta. \end{aligned} \tag{5.2}$$

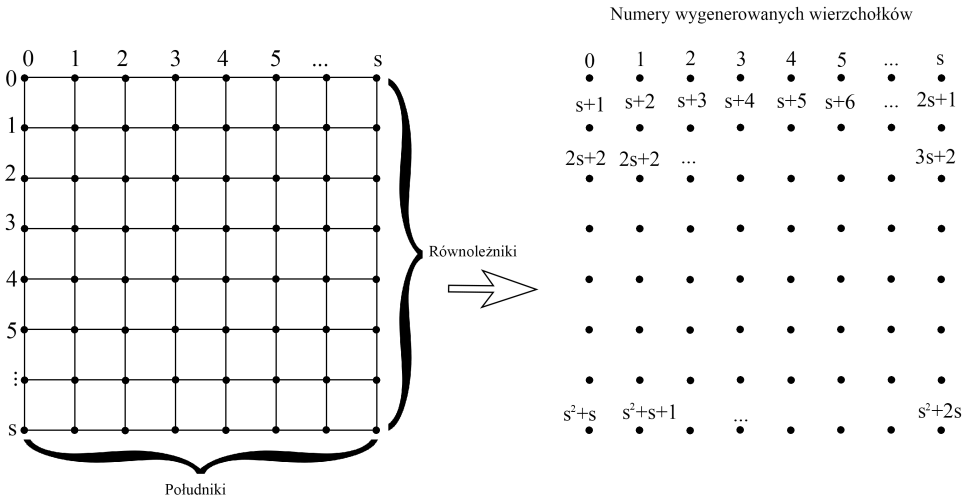
W efekcie otrzymujemy następujący algorytm generowania wierzchołków na powierzchni sfery.

Listing 5.12. Generowanie wierzchołków sfery.

```
r, g, b = color
```

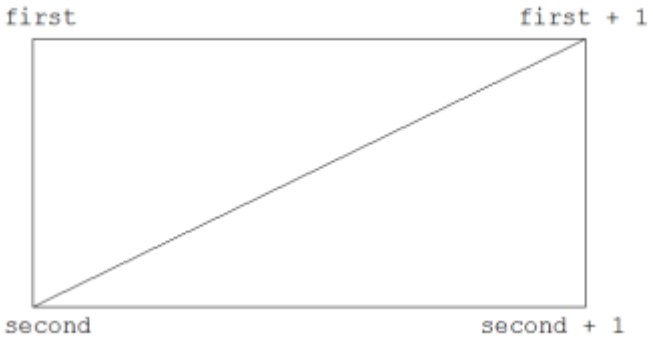
```
vertex_data = []

steps = 20
angle_step = np.pi / steps

for latNumber in range(steps + 1):
    theta = latNumber * angle_step
    sinTheta = np.sin(theta)
    cosTheta = np.cos(theta)
    for longNumber in range(steps + 1):
        phi = longNumber * 2 * angle_step
        sinPhi = np.sin(phi)
        cosPhi = np.cos(phi)

        x = cosPhi * sinTheta
        y = cosTheta
        z = sinPhi * sinTheta

        vertex_data.extend((x * radius, y * radius, z * radius,
        r, g, b))
```

Po wygenerowaniu wszystkich potrzebnych wierzchołków, należy wyznaczyć indeksy, aby OpenGL wiedział jak ma połączyć te wierzchołki w ściany. Poniższa ilustracja pokazuje w jakiej kolejności są ułożone wierzchołki w naszej tablicy wierzchołków.



Rysunek 5.10. Kolejność wierzchołków w buforze wierzchołków dla modelu sfery, gdzie s to liczba południków oraz równoleżników.

Pamiętajmy, że każdy kwadrat należy rozbić na dwa trójkąty. Znając kolejność wierzchołków w buforze, wyznaczamy indeksy wierzchołków tworzących kolejne trójkąty w następujący sposób.



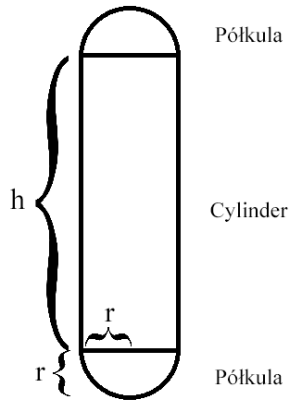
Rysunek 5.11. Wyznaczanie indeksów wierzchołków trójkątów tworzących sferę. Ilustracja zaczerpnięta z tutorialu *Learn WebGL* [33]

Listing 5.13. Wyznaczanie indeksów wierzchołków sfery.

```
for latNumber in range(steps):
    for longNumber in range(steps):
        first = (latNumber * (steps + 1)) + longNumber
        second = first + steps + 1

        index_data.append(first)
        index_data.append(second)
        index_data.append(first + 1)

        index_data.append(second)
        index_data.append(second + 1)
        index_data.append(first + 1)
```

5.7.4. Sferocylindry

Molekuły jednoosiowe są w naszej aplikacji wizualizowane jako sferocylindry, tj. cylindry z półkulami w miejscu podstaw.



Rysunek 5.12. Geometria sferocylindra (r - promień h - wysokość cylindra).

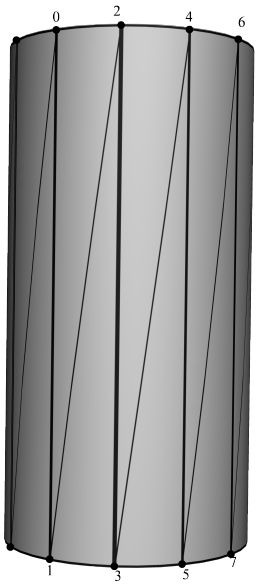
Metoda generująca cylinder przyjmuje jako parametry promień i wysokość cylindra oraz kolory dla części cylindrycznej i półkulistych. Zaczynamy od wygenerowania cylindra korzystając ze zmodyfikowanego wzoru na przejście z układu współrzędnych biegunowych do układu współrzędnych kartezjańskich [51].

$$\begin{aligned} x &= r\cos\phi, \\ z &= r\sin\phi. \end{aligned} \tag{5.3}$$

Dla każdego kąta ϕ z zakresu $[0°, 360°]$ generujemy dwa wierzchołki: jeden na wysokości $y = h/2$, drugi na wysokości $y = -h/2$. Od razu generujemy też indeksy tworzące ścianki wyznaczone przez dwa wierzchołki, które właśnie wygenerowaliśmy i dwa, które wygenerujemy w następnym kroku. Po wyjściu z pętli, do tablicy wierzchołków dodajemy ostatnie 2 wierzchołki (dla $\phi = 360°$).
Przykład:

1. Generujemy wierzchołki 0, 1, dodajemy je do tablicy wierzchołków, a do tablicy indeksów dodajemy [0, 1, 2, 1, 3, 2], tj tworzymy prostokąt z wierzchołkach 0, 1, 2, 3.
2. Generujemy wierzchołki 2, 3, tworzymy prostokąt o wierzchołkach 2, 3, 4, 5.
   ...
s. Generujemy wierzchołki $s-1$, s, tworzymy prostokąt o wierzchołkach $s-1$, s, $s+1$, $s+2$.

Po wyjściu z pętli dodajemy wierzchołki $s+1$, $s+2$.



Rysunek 5.13. Kolejność tworzenia wierzchołków i podział cylindra na ścianki.

Listing 5.14. Generowanie wierzchołków i indeksów cylindra.

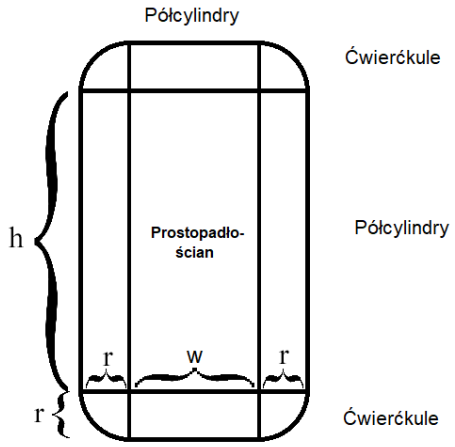
```
half_height = height / 2.0
steps = 20
```

```
angle_step = 2 * np.pi / steps

for i in range(steps):
    angle = i * angle_step
    x = radius * np.cos(angle)
    z = radius * np.sin(angle)

    vertex_data.extend((x, half_height, z))
    vertex_data.extend(tube_color)
    vertex_data.extend((x, -half_height, z))
    vertex_data.extend(tube_color)

    index = 2 * i

    index_data.append(index)
    index_data.append(index + 1)
    index_data.append(index + 2)

    index_data.append(index + 1)
    index_data.append(index + 3)
    index_data.append(index + 2)

vertex_data.extend((radius, half_height, 0))
vertex_data.extend(tube_color)
vertex_data.extend((radius, -half_height, 0))
vertex_data.extend(tube_color)
```

Po wygenerowaniu wierzchołków oraz indeksów cylindra, generujemy wierzchołki i indeksy dla obu półkul, najpierw górnej, potem dolnej. Wykorzystujemy zmodyfikowany algorytm opisany w rozdziale 5.7.3. Przy generowaniu wierzchołków półkul bierzemy pod uwagę przesunięcie punktów wzdłuż osi y o $h/2$ dla górnej półkuli oraz $-h/2$ dla dolnej półkuli. Przy generowaniu indeksów dla obu półkul, uwzględniamy ilość wierzchołków z jakich składają się uprzednio wygenerowane części figury.

5.7.5. Sferopłytki

Sferopłytek używamy do wizualizacji molekuł dwuosiowych. Metoda generująca model sferopłytki przyjmuje jako parametry: promień (ćwierćsfer oraz półcylindrów), wysokość i szerokość prostopadłościanu.



Rysunek 5.14. Geometria sferopłytki. Wymiary prostopadłościanu $w \times h \times 2r$. Wymiary sferopłytki $(w + 2r) \times (h + 2r) \times 2r$

Na początek generujemy dwie widoczne ściany prostopadłościanu.

Listing 5.15. Generowanie płaskich ścian sferopłytki.

```
# plate vertexes
vertex_data = [-half_width,  half_height, radius, r, g, b,
half_width,  half_height, radius, r, g, b,
half_width, -half_height, radius, r, g, b,
-half_width, -half_height, radius, r, g, b,

-half_width,  half_height, -radius, r, g, b,
half_width,  half_height, -radius, r, g, b,
half_width, -half_height, -radius, r, g, b,
-half_width, -half_height, -radius, r, g, b]

# plate indexes
index_data = [0, 1, 2, 0, 2, 3,
4, 5, 6, 4, 6, 7]
```

Następnie generujemy odpowiednio półcylindry i ćwierćkule, używając lekko zmodyfikowanych algorytmów opisanych we wcześniejszych rozdziałach.

6. Konfiguracja projektu

W celu skonfigurowania programu należy zainstalować wymagane programy oraz dodać wymagane pakiety do zainstalowanego interpretera Pythona. Projekt może funkcjonować w dowolnym środowisku wspierającym Pythona 3. Poniżej opisano konfigurację projektu w środowiskach Windows i Linux.

Wymagane programy:
— Python 3, wersja 3.3 lub nowsza
— python3-pip (w dalszej części pracy będzie nazywany pip), zwykle instaluje się razem z Pythonem 3

Wymagane pakiety:
— pyglet - tworzenie okienka do wyświetlania molekuł
— pickleshare - zapisywanie i wczytywanie plików modeli
— PyOpenGL - OpenGL dla Pythona
— PyOpenGL-accelerate - optymalizacja OpenGL dla Pythona, poprawia działanie części graficznej aplikacji
— numpy - tablice, operacje na tablicach, zaawansowane funkcje matematyczne
— pyrr - operacje na macierzach w aplikacji graficznej (wykorzystuje numpy)
— matplotlib - rysowanie wykresów
— python3-tk - backend dla wykresów z matplotlib
— freeglut3-dev - zainstalowanie tego pakietu rozwiązuje niektóre problemy z instalacją PyOpenGL w niektórych systemach

Możliwe jest, że w niektórych systemach oraz w pewnych przypadkach użycia, projekt będzie mógł działać poprawnie bez uprzedniego zainstalowania niektórych z powyższych pakietów. Jednak w celu zapewnienia stabilnej pracy programu w dowolnym środowisku, sugerujemy zainstalowanie wszystkich wyżej wymieniowych pakietów.

Niezależnie od środowiska, w którym uruchomimy nasz program, należy go najpierw pobrać z repozytorium lub innego źródła i w miarę potrzeby rozpakować. Program nie zawiera instalatora.

6.1. Instalacja w systemie Windows

W systemie Windows najłatwiej zainstalować wymagane biblioteki poprzez środowisko PyCharm.

1. Pobierz instalator Python 3 ze strony `https://www.python.org/` i uruchom go.

2. Znajdź w internecie instalator PyCharm i uruchom go.
3. Uruchom PyCharm i otwórz w nim nasz projekt.
4. Skonfiguruj interpreter oraz dodaj wymagane moduły do projektu:
   a) przejdź do okna
      File -> Settings -> Project -> Project Interpreter,
   b) kliknij koło zębate znajdujące się po prawej stronie pola 'Project interpreter' i wybierz 'Add Local',
   c) znajdź plik 'python.exe' (wewnątrz folderu instalacyjnego Pythona z kroku 1) i dodaj go,
   d) kliknij ikonkę '+' znajdującą się poniżej ikonki koła zębatego,
   e) wyszukaj potrzebny moduł po nazwie i kliknij 'Install Package'.

Alternatywnie, bez potrzeby instalacji PyCharm, można dodać wymagane pakiety do interpretera Python 3 za pomocą poniższego polecenia:

```
py -3 -m pip install NazwaPakietu
```

Biblioteka PyOpenGL ma pewne zależności systemowe, których pip czasem nie może samodzielnie zainstalować. W takim wypadku, komunikat błędu będzie zawierać wskazówki dotyczące tego, co trzeba jeszcze doinstalować. Po instalacji wskazanych zależności, należy ponownie spróbować zainstalować PyOpenGL.

W przypadku, w którym inne biblioteki nie będą się instalowały za pomocą pip, należy wyszukać w Internecie instalator dla danej biblioteki. Jeśli taki nie istnieje, można też wyszukać plik źródłowy biblioteki (mają one najczęściej format .tar.gz lub .whl) odpowiedni dla używanej wersji Windows oraz Pythona. Plik ten należy ściągnąć i zainstalować przy pomocy polecenia:

```
py -3 -m pip install SciezkaDoPliku
```

Jeśli żaden z powyższych sposobów nie rozwiązuje problemu, należy samodzielnie wyszukać rozwiązania na podstawie komunikatu błędu.

6.2. Instalacja w systemie Linux

Uruchomienie poniższej serii poleceń w konsoli powinno zainstalować nam wszystkie wymagane programy i biblioteki. Dla bezpieczeństwa zalecamy uruchamianie tych poleceń pojedynczo i rozwiązywanie wszelkich problemów z instalacją, jak tylko się pojawią.

— sudo apt-get install python3
— sudo apt-get install python3-pip
— sudo pip3 install pyglet
— sudo pip3 install pickleshare
— sudo pip3 install PyOpenGL
— sudo pip3 install PyOpenGL-accelerate
— sudo pip3 install pyrr
— sudo pip3 install numpy
— sudo -H pip3 install matplotlib
— sudo apt-get install python3-tk

— sudo apt-get install freeglut3-dev

Jeśli będzie problem z instalacją którejś z bibliotek, spowodowany brakiem praw dostępu, dodanie flagi -H do polecenia sudo może ten problem rozwiązać.

Jeśli pip nie może znaleźć odpowiedniej dystrybucji pakietu dla danego środowiska, należy ją ręcznie wyszukać przez Google, pobrać i zainstalować przy pomocy polecenia:

```
sudo apt-get install SciezkaDoPliku
```

lub

```
sudo pip3 install SciezkaDoPliku
```

w miarę potrzeby dodając flagę -H.

Jeśli żaden z powyższych sposobów nie rozwiązuje problemu, należy samodzielnie wyszukać rozwiązanie na podstawie komunikatu błędu.

6.3. Generowanie modeli 3D

Po instalacji pakietów należy uruchomić skrypt ModelGenerator.py. Skrypt ten tworzy pliki zawierające dane modeli 3D, z których będą korzystać skrypty graficzne.

7. Przypadki użycia aplikacji

Aplikacja została zrealizowana jako zestaw modułów i skryptów odpowiedzialnych za uruchamianie symulacji i podglądanie wyników. Przykłady wywołania skryptów wykorzystują składnię z konsoli bashowej. Skrypty można jednak uruchomić w dowolnym środowisku wspierającym Pythona 3.3 (bądź nowszego), o ile zainstalowano odpowiednie pakiety.

Przypadki użycia naszej aplikacji można przypisać do trzech ogólnych kategorii:

1. Przeprowadzanie symulacji.
2. Podglądanie stanu macierzy.
3. Analiza wyników.

7.1. Przeprowadzanie symulacji

Przeprowadzenie symulacji jest podstawowym przypadkiem użycia aplikacji. Podgląd stanu macierzy oraz analiza wyników są możliwe dopiero po wykonaniu symulacji. Aby przeprowadzić symulację, należy uruchomić skrypt run_simulations.py.

Skrypt przyjmuje dowolną liczbę parametrów, z których każdy powinien być ścieżką do pliku konfiguracyjnego. Dla każdego parametru, skrypt uruchamia serię symulacji skonfigurowaną wewnątrz pliku.

Plik konfiguracyjny może być dowolnym plikiem pythonowym (rozszerzenie .py), zawierającym słownik o nazwie configuration. Wszystkie ustawienia dotyczące symulacji są pobierane z tego słownika. Opis wszystkich parametrów z pliku konfiguracyjnego znajduje się w tabeli 7.1.

Tabela 7.1. Opis parametrów w pliku konfiguracyjnym.

Klucz	Wartość wymagana	Typ zmiennej / możliwe wartości	Opis
lattice_size	TAK*	(int, int, int)	Rozmiar macierzy (x, y, z)
temperature_list	TAK*	dowolna iterowalna kolekcja wartości float	Lista temperatur. Dla każdej temperatury z listy uruchamiana jest osobna symulacja, wykorzystująca stan końcowy z poprzedniej symulacji jako swój stan początkowy. Stan początkowy pierwszej symulacji jest określany przez 'init_type'.
simulation_type	TAK*	'ising', 'uniax', 'biax'	Typ symulacji do przeprowadzenia: 'ising' - model Isinga, 'uniax' - model Lebowhla-Lashera, 'biax' - model molekuł dwuosiowych.
zannoni_parameter (alias 'a')	tylko dla symulacji 'biax'	float	Parametr dwuosiowości molekuły: $a = 0 \Rightarrow$ idealnie jednoosiowa, $0 < a < 1/\sqrt{3} \Rightarrow$ dwuosiowa prętopodobna, $a = 1/\sqrt{3} \Rightarrow$ idealnie dwuosiowa, $1/\sqrt{3} < a < \sqrt{2} \Rightarrow$ dwuosiowa dyskopodobna, $a = \sqrt{2} \Rightarrow$ dysk.
init_type	NIE	'random', 'organized', 'checkered', 'load'	Sposób inicjalizacji początkowego stanu macierzy. Domyślnie 'random'. 'random' - przypadkowa macierz 'organized' - macierz uporządkowana 'checkered' - w kratkę, tylko dla M. Isinga 'load' - wczytaj z pliku
load_from	tylko jeśli 'init_type' to 'load'	string	Ścieżka do pliku, z którego ładujemy początkową konfigurację.
production_cycles	NIE**	int	Ilość cykli produkcyjnych, domyślnie 0.
warmup_cycles	NIE**	int	Ilość cykli warmupowych, domyślnie 0.
input_dir	NIE	string	Ścieżka do folderu zawierającego wyniki poprzedniej symulacji. Używając wyników poprzedniej symulacji można skrócić/pominąć etap warmup.
output_dir	NIE	string	Ścieżka do folderu do zapisu wyników. Domyślnie wyniki są zapisywane w tym samym folderze co plik konfiguracyjny.
warmup_data_output	NIE	boolean	Jeśli True, dane zbierane podczas etapu warmup są zapisywane do osobnego pliku.
production_data_output	NIE	boolean	Jeśli True, dane zbierane podczas etapu produkcyjnego są zapisywane do osobnego pliku.
repeats	NIE	int	Liczba powtórzeń. Domyślnie 1. Jeśli repeats > 1 seria symulacji przeprowadzana jest repeats razy, a wyniki zapisywane są do ponumerowanych podfolderów.

*Parametr nie jest wymagany, jeśli parametr 'input_dir' został zdefiniowany.
W takim przypadku parametr jest dedukowany na podstawie zawartości podanego folderu, a jego wartość podana w pliku konfiguracyjnym jest ignorowana.
**Jedną z tych dwóch wartości trzeba zdefiniować.

Listing 7.1. Przykładowe wywołanie skryptu run_simulations.py.

```
./run_simulations.py someDirectory/example_config.py
```

Listing 7.2. Przykładowa zawartość pliku konfiguracyjnego.

```
import numpy as np

configuration = {
'simulation_type': 'uniax',
'init_type': 'random',
'lattice_size': (10, 10, 10),
'temperature_list': np.round(np.linspace(3, 0.1, 30), decimals=1),
'production_cycles': 1000,
'warmup_cycles': 1000,
'input_dir': '',
'output_dir': 'Figures3/uniax/uniax_10x10x10_2',
'warmup_data_output': True,
'production_data_output': True,
'repeats': 1,
}
```

7.1.1. Ogólny schemat działania

Zestaw symulacji można uruchomić używając pojedynczego pliku konfiguracyjnego:

```
./run_simulations.py config.py
```

lub kilku plików konfiguracyjnych, np.:

```
./run_simulations.py config1.py config2.py config3.py
```

W przypadku użycia kilku plików konfiguracyjnych, skrypt najpierw wykonuje serię symulacji zdefiniowaną w pierwszym pliku, po jej zakończeniu wykonuje tę opisaną w drugim pliku, potem w trzecim, itd.

Skrypt ładuje dane z pliku konfiguracyjnego, a następnie dla każdej wartości z 'temperature_list' (patrz tabela 7.1) uruchamia odpowiednią symulację.

Seria symulacji zwraca wyniki w formie plików. Wszystkie pliki generowane przez symulacje są zapisywane w folderze podanym w 'output_dir' (lub domyślnie w tym samym folderze, co plik konfiguracyjny).

Jeśli folder 'output_dir' jest inny niż ten, w którym znajduje się plik konfiguracyjny, plik konfiguracyjny jest kopiowany do 'output_dir' pod nazwą *config.py*. Aplikacja automatycznie tworzy wszystkie wymagane foldery, jeśli nie istnieją.

7.1.2. Generowane pliki

Pliki, generowane przez symulacje w naszym programie, można podzielić na dwie kategorie: pliki macierzowe (rozszerzenia .sng, .unx, .bnx), oraz pliki zawierające dane liczbowe (rozszerzenie .data.csv). Rodzaj pliku można rozpoznać po nazwie. Stosowane są następujące schematy nazw.

1. Plik macierzy:
   Lattice_[X]x[Y]x[Z]_[T].[rozszerzenie]
   (np. *Lattice_10x10x10_0.2.unx*)
2. Plik warmupowy:
   [simulation_type]_warmup_[X]x[Y]x[Z]_[T].data.csv
   (np. *uniax_warmup_10x10x10_0.2.data.csv*)
3. Plik produkcyjny:
   [simulation_type]_production_[X]x[Y]x[Z]_[T].data.csv
   (np. *uniax_production_10x10x10_0.2.data.csv*)
4. Plik wynikowy:
   [simulation_type]_[X]x[Y]x[Z].data.csv
   (np. *uniax_10x10x10.data.csv*)

gdzie:
— 'simulation_type' - wartość z pliku konfiguracyjnego, wyznaczająca typ symulacji,
— X, Y, Z - rozmiar macierzy wzdłuż osi X, Y, Z,
— T - temperatura.

Program zakłada, że docelowy folder (najgłębszy folder wymieniony w ścieżce 'output_dir') będzie nazwany według schematu:
[simulation_type]_[X]x[Y]x[Z]_[num] (np. *ising_10x10x10_1*),
gdzie *num* to pewien arbitralnie przypisany numer symulacji.

Jeśli nazwa folderu docelowego nie jest zbudowana według spodziewanego schematu, aplikacja generuje odpowiedni podfolder wewnątrz 'output_dir'. W takim przypadku, jako *num* przyjmowany jest czas uniksowy z chwili wygenerowania folderu, a 'output_dir' jest nadpisywany ścieżką do wygenerowanego folderu.

7.1.2.1. Zawartość pliku macierzowego

Pliki macierzowe są plikami binarnymi. Zawierają stan macierzy w postaci tabeli numpy.ndarray spakowanej do pliku przy pomocy biblioteki pickle. Zapisywanie oraz odczytywanie danych z tych plików odbywa się za pośrednictwem metod dostarczonych przez tą bibliotekę.

7.1.2.2. Zawartość plików z danymi liczbowymi

Pliki .data.csv zawierają dane w formacie tekstowym. Są podzielone na wiersze, przy czym każdy wiersz zawiera zestaw danych zależny od typu symulacji (model Isinga, Lebwohla-Lashera, molekuły dwuosiowe) i rodzaju pliku (warmupowy, produkcyjny, wynikowy).

Pliki warmupowy oraz produkcyjny zawierają odpowiednio dane zbierane podczas etapów warmup i production. Dane zawarte w tych plikach pozwalają prześledzić poprawność przebiegu tych etapów.

Pliki wynikowe zawierają wyniki wszystkich symulacji z zestawu. Pozwalają prześledzić zmiany stanu próbki w zależności od temperatury.

Do opisu danych zawartych w każdym z tych plików będziemy używali poniższych oznaczeń:
— it - numer iteracji (numer_cyklu * rozmiar_cyklu)
— e - gęstość energii
— m - magnetyzacja

— ord - parametr porządku dla molekuł jednoosiowych
— cv - ciepło właściwe
— chi - podatność magnetyczna w modelu Isinga lub podatność parametru porządku na zmiany w modelu Lebwohla-Lashera
— F200 - parametr porządku $\langle F_{00}^{(2)} \rangle$
— F202 - parametr porządku $\langle F_{02}^{(2)} \rangle$
— F220 - parametr porządku $\langle F_{20}^{(2)} \rangle$
— F222 - parametr porządku $\langle F_{22}^{(2)} \rangle$
— t - temperatura

Pliki warmupowe: Plik warmup jest generowany dla każdej symulacji w zestawie, pod warunkiem, że parametr 'warmup_data_output' został ustawiony jako True w pliku konfiguracyjnym oraz parametr 'wamup_cycles' jest większy niż 0.

Każdy wiersz pliku warmupowego zawiera następujące dane:
— dla modelu Isinga
[it] [e] [m]
— dla modelu Lebwohla-Lashera
[it] [e] [ord]
— dla modelu molekuł dwuosiowych
[it] [e] [F200] [F202] [F220] [F222]

Pliki produkcyjne: Plik produkcyjny jest generowany dla każdej symulacji w zestawie, pod warunkiem, że parametr 'production_data_output' został ustawiony jako True w pliku konfiguracyjnym oraz parametr 'production_cycles' jest większy niż 0.

Każdy wiersz pliku produkcyjnego zawiera następujące dane:
— dla modelu Isinga
[it] [e] [m] [e^2] [m^2]
— dla modelu Lebwohla-Lashera
[it] [e] [ord]
— dla modelu molekuł dwuosiowych
[it] [e] [e^2] [F200] [F202] [F220] [F222]

Pliki wynikowe: Pojedynczy plik wynikowy jest generowany dla całego zestawu symulacji, pod warunkiem, że parametr 'production_cycles' jest większy od 0.

Każdy wiersz pliku wynikowego zawiera następujące dane:
— dla modelu Isinga
[t] [e] [m] [cv] [chi]
— dla modelu Lebwohla-Lashera
[t] [e] [ord] [cv] [chi]
— dla modelu molekuł dwuosiowych
[t] [e] [F200] [F202] [F220] [F222] [cv]

7.1.3. Przykładowe symulacje

1. Prosta symulacja warmupowa dla modelu Isinga.

Chcemy przeprowadzić serię warmupów dla symulacji modelu Isinga w macierzy o rozmiarze 10x10x10. Plik konfiguracyjny znajduje się w folderze docelowym *Figures/ising_10x10x10_1*. Dane z przebiegu etapu warmup zostaną zapisane do pliku, aby upewnić się, że warmup przebiegł prawidłowo.
Ponieważ zaczynamy od wysokiej temperatury, tryb inicjalizacji 'random' jest najbardziej odpowiedni.
Parametry zestawu symulacji:
— model Isinga,
— rozmiar macierzy: 10x10x10,
— zakres temperatur: [9.0, 8.9, 8.8, ... 0.1],
— tryb inicjalizacji: 'random',
— ilość cykli warmup: 200,
— ilość cykli produkcyjnych: 0,
— zapis danych warmupowych włączony.
Plik konfiguracyjny *Figures/ising_10x10x10_1/config.py*:

```
import numpy as np

configuration = {
'simulation_type': 'ising',
'lattice_size': (10, 10, 10),
'temperature_list': np.round(np.linspace(9, 0.1, 90),
decimals=1),
'init_type': 'random',
'warmup_cycles': 200,
'warmup_data_output': True,
}
```

Wywołanie:

```
./run_simulations.py Figures/ising_10x10x10_1/config.py
```

2. Symulacja produkcyjna wykorzystująca wyniki symulacji z punktu 1.
Chcemy wykorzystać wyniki z symulacji 1 (zawarte w folderze *Figures/ising_10x10x10_1/*), aby pominąć etap warmup w symulacji produkcyjnej i w ten sposób przyśpieszyć wykonanie. Chcemy też zdefiniować folder *Figures/ising_10x10x10_2*, jako folder do zapisu danych. W celu upewnienia się, że etap produkcyjny przebiegł prawidłowo, chcemy zapisać dane z tego przebiegu.
Parametry zestawu symulacji:
— model Isinga,
— folder z danymi z poprzedniej symulacji: *Figures/ising_10x10x10_1/*,
— folder do zapisu danych: *Figures/ising_10x10x10_2*,
— zapis danych produkcyjnych włączony.
Plik konfiguracyjny *config.py*:

```
configuration = {
'input_dir': 'Figures/ising_10x10x10_1/',
'output_dir': 'Figures/ising_10x10x10_2/',
'production_cycles': 200,
'production_data_output': True,
}
```

Wywołanie:

```
./run_simulations.py config.py
```

3. Symulacja wielokrotnie powtarzająca ten sam eksperyment, w celu zbadania przypadkowości przebiegu etapu warmup.
Metody Monte Carlo są z natury przypadkowe, chcemy sprawdzić jaki wpływ na przebieg etapu warmup ma ta przypadkowość. W tym celu warto przeprowadzić ten sam eksperyment wielokrotnie.
Parametry zestawu symulacji:
— model Isinga,
— rozmiar macierzy: 10x10x10,
— zakres temperatur: [9.0, 8.9, 8.8, ... 0.1],
— ilość cykli warmupowych: 300,
— zapis danych warmupowych włączony,
— folder do zapisu danych: *Figures/ising_10x10x10_3*,
— ilość powtórzeń: 4.
Plik konfiguracyjny *config.py*:

```
import numpy as np

configuration = {
'simulation_type': 'ising',
'init_type': 'random',
'lattice_size': (10, 10, 10),
'temperature_list': np.round(np.linspace(9, 0.1, 90),
decimals=1),
'warmup_cycles': 300,
'warmup_data_output': True,
'repeats': 4,
}
```

Wywołanie:

```
./run_simulations.py config.py
```

4. Symulacja Lebwohla-Lashera dla dużej macierzy, z zapisem danych ze wszystkich etapów.
Chcemy przeprowadzić symulację Lebwohla-Lashera dla macierzy 20x20x20, w celu sprawdzenia, czy symulacja dobrze działa dla większych rozmiarów macierzy. Zdecydowaliśmy się zapisać dane ze wszystkich etapów.
Parametry zestawu symulacji:
— model Lebwohla-Lashera,
— rozmiar macierzy: 20x20x20,
— zakres temperatur: [3.0, 2.9, 2.8, ... 0.1],
— ilość cykli warmupowych: 200,
— ilość cykli produkcyjnych: 200,
— zapis danych warmupowych włączony,
— zapis danych produkcyjnych włączony,
— folder do zapisu danych: *Figures/uniax_20x20x20_1*.
Plik konfiguracyjny *config.py*:

```
import numpy as np
```

```
configuration = {
'simulation_type': 'uniax',
'init_type': 'random',
'lattice_size': (20, 20, 20),
'temperature_list': np.round(np.linspace(3, 0.1, 30),
decimals=1),
'warmup_cycles': 200,
'production_cycles': 200,
'warmup_data_output': True,
'production_data_output': True,
'output_dir': 'Figures/uniax_20x20x20_1',
}
```

5. Kontynuowanie zestawu symulacji przerwanego przez awarię.
   Załóżmy, że z powodu awarii sprzętu zestaw symulacji z poprzedniego przykładu został przerwany w trakcie wykonania. Ponieważ symulacja zapisuje wyniki na bieżąco, wystarczy wczytać stan końcowy z ostatniej wykonanej symulacji oraz wykonać symulacje dla pozostałych temperatur.
   Załóżmy, że ostatni zestaw symulacji był dla modelu Lebwohla-Lashera, macierzy 20x20x20, zakresu temperatur [3.0, ... 0.1], a ostatnia prawidłowo zakończona symulacja została wykonana dla temperatury 0.3. Wystarczy wczytać stan macierzy z pliku *Lattice_20x20x20_0.3.unx* i przeprowadzić brakujące dwie symulacje dla temperatur 0.2 oraz 0.1.
   Plik konfiguracyjny *config.py*:

```
configuration = {
'simulation_type': 'uniax',
'init_type': 'load',
'load_from': 'Figures/uniax_20x20x20_2/Lattice_20x20x20_0.3.unx',
'lattice_size': (20, 20, 20),
'temperature_list': [0.2, 0.1],
'production_cycles': 200,
'warmup_cycles': 200,
'warmup_data_output': True,
'production_data_output': True,
'output_dir': 'Figures/uniax_20x20x20_2',
}
```

6. Symulacje modelu molekuł dwuosiowych.
   Chcemy przeprowadzić dwa zestawy symulacji modelu molekuł dwuosiowych, każdy dla innej wartości parametru dwuosiowości. Nie można tego skonfigurować pojedynczym plikiem konfiguracyjnym, każdy z dwóch zestawów trzeba skonfigurować osobno.
   Symulację chcemy przeprowadzić dla zakresu temperatur od 0.05 do 2.0. Ponieważ zaczynamy od niskich temperatur, ma sens ustawienie początkowego stanu macierzy na stan uporządkowany.
   Parametry obu zestawów symulacji:
   — model molekuł dwuosiowych,
   — rozmiar macierzy: 10x10x10,
   — tryb inicjalizacji: 'organized',
   — zakres temperatur: [0.05, 0.1, 0.15, ... 2.0],

— parametr dwuosiowości: 0.55 lub $1/\sqrt{3}$,
— ilość cykli warmupowych: 1000,
— ilość cykli produkcyjnych: 1000,
— folder do zapisu danych: *Figures3/biax/biax_a55* lub *Figures3/biax/biax_a577*.

Plik *config1.py*:

```
configuration = {
'simulation_type': 'biax',
'zannoni_parameter': 0.55,
'init_type': 'organized',
'lattice_size': (10, 10, 10),
'temperature_list': np.round(np.arange(0.05, 2.01, 0.05),
decimals=2),
'production_cycles': 1000,
'warmup_cycles': 1000,
'output_dir': 'Figures/biax/biax_a55',
}
```

Plik *config2.py*:

```
configuration = {
'simulation_type': 'biax',
'zannoni_parameter': 1 / math.sqrt(3),
'init_type': 'organized',
'lattice_size': (10, 10, 10),
'temperature_list': np.round(np.arange(0.05, 2.01, 0.05),
decimals=2),
'production_cycles': 1000,
'warmup_cycles': 1000,
'output_dir': 'Figures/biax/biax_a577',
}
```

Wywołanie:

```
./run_simulations.py config1.py config2.py
```

7.2. Podglądanie stanu macierzy

Po wykonaniu symulacji można podejrzeć stan macierzy. Takie podglądy pomagają w sprawdzeniu poprawności działania symulacji, jak również zlokalizowaniu domen uporządkowanych wewnątrz próbki.

Projekt dostarcza dwa sposoby podglądania stanu próbki: podgląd podzielony na warstwy (skrypt show_layers.py), oraz podgląd całości próbki (skrypt show_lattice.py).

Oba skrypty działają w podobny sposób. Skrypt przyjmuje jako argument ścieżkę do pliku przechowującego stan macierzy, np.
Figures/uniax_20x20x20_2/Lattice_20x20x20_0.3.unx,
albo do folderu zawierającego wyniki symulacji, np.
Figures/uniax_20x20x20_2/.

W przypadku, w którym argumentem jest plik macierzy, skrypt wyświetli podgląd stanu macierzy zapisanego w tym pliku. Jeśli argumentem jest folder, skrypt wyświetli po kolei wszystkie zawarte w nim macierze. Ponieważ

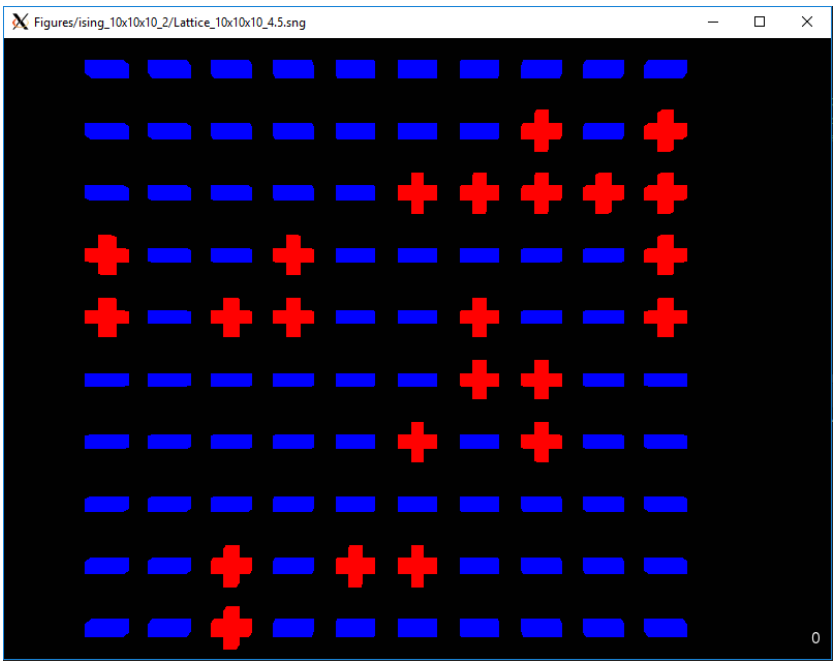
program wyświetla tylko jeden podgląd na raz, aby wyświetlić następny należy zamknąć otwarte przez program okno, wówczas program automatycznie otworzy następne.

7.2.1. Podgląd warstw

Podgląd warstw uzyskujemy za pomocą skryptu show_layers.py. Sposób w jaki należy wyświetlić dane jest dostosowany do rodzaju pliku, na podstawie jego rozszerzenia.

Pliki .sng zawierają macierze modelu Isinga.

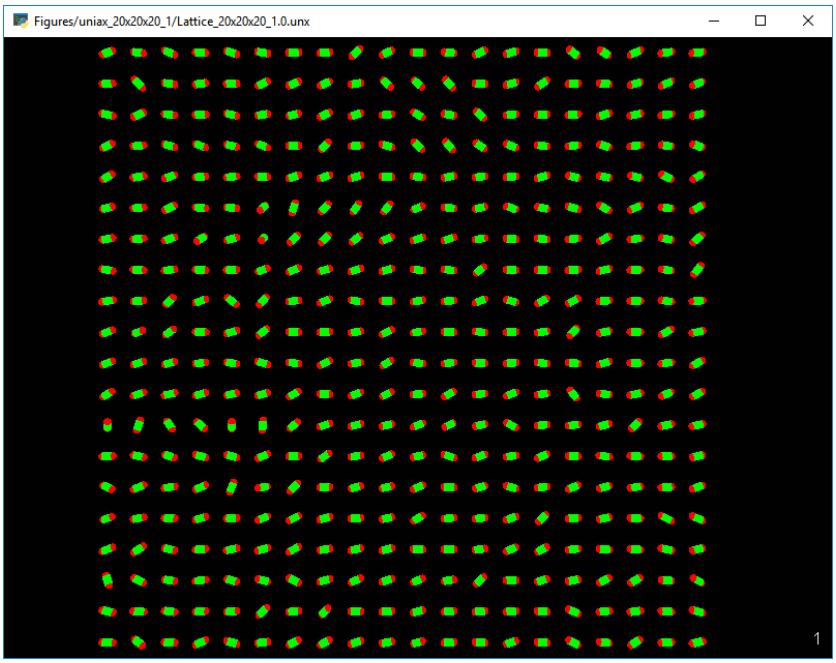
```
./show_layers.py Figures/ising_10x10x10_2/Lattice_10x10x10_4.5.sng
```



Rysunek 7.1. Podgląd warstw w modelu Isinga.

Pliki .unx zawierają macierze modelu Lebwohla-Lashera.

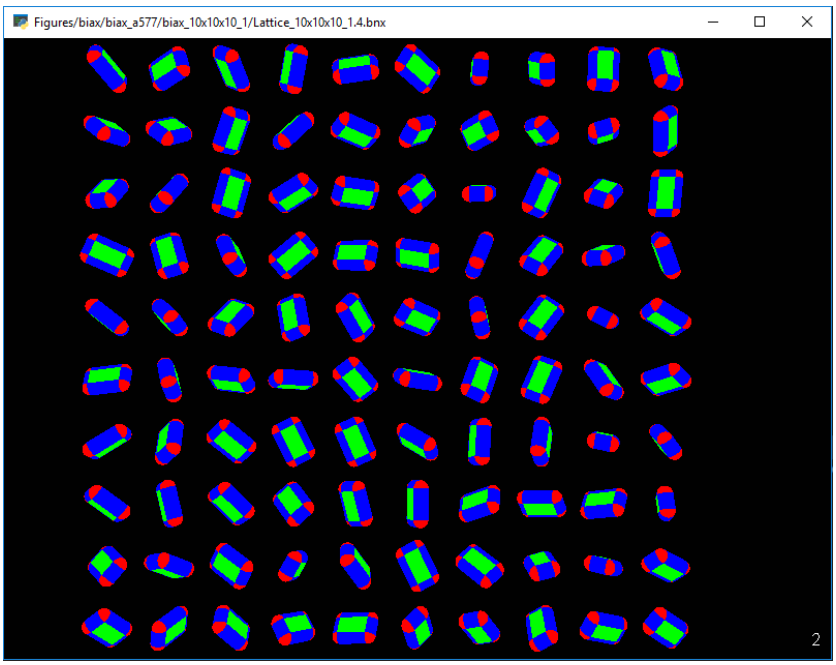
```
./show_layers.py Figures/uniax_20x20x20_1/Lattice_20x20x20_1.0.unx
```



Rysunek 7.2. Podgląd warstw w modelu Lebwohla-Lashera.

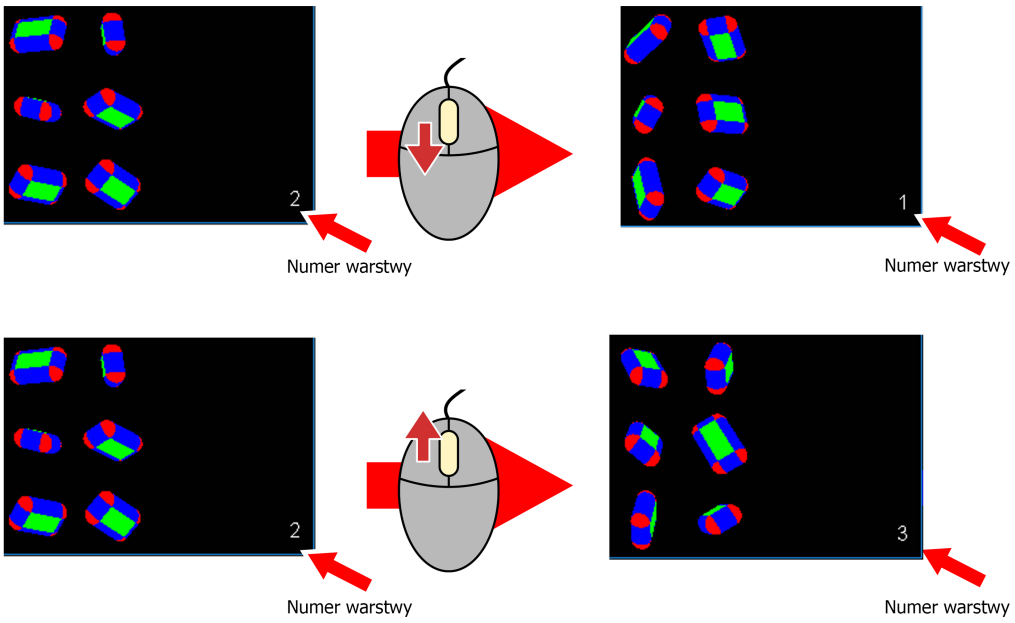
Pliki .bnx zawierają macierze modelu molekuł dwuosiowych.

```
./show_layers.py Figures/biax/biax_a577/biax_10x10x10_1/
Lattice_10x10x10_1.4.bnx
```



Rysunek 7.3. Podgląd warstw w modelu molekuł dwuosiowych.

Podczas podglądu, można przełączać się pomiędzy warstwami za pomocą kółka myszy.



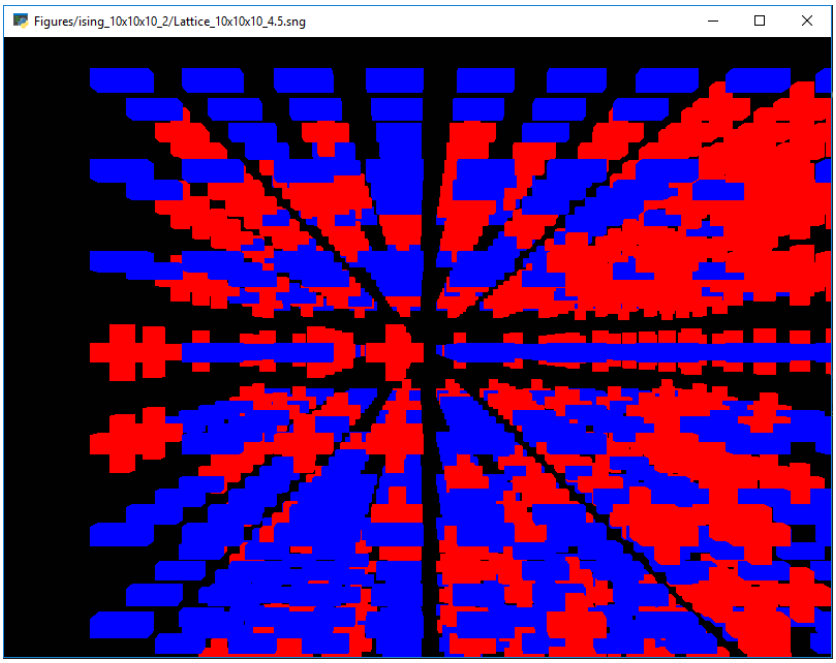
Rysunek 7.4. Przełączanie podglądu warstwy.

7.2.2. Podgląd całej próbki

Podgląd całej próbki uzyskujemy za pomocą skryptu show_lattice.py. Podobnie jak w przypadku skryptu show_layers.py, sposób wyświetlania danych jest dostosowany do rodzaju pliku.

Pliki .sng zawierają macierze modelu Isinga.

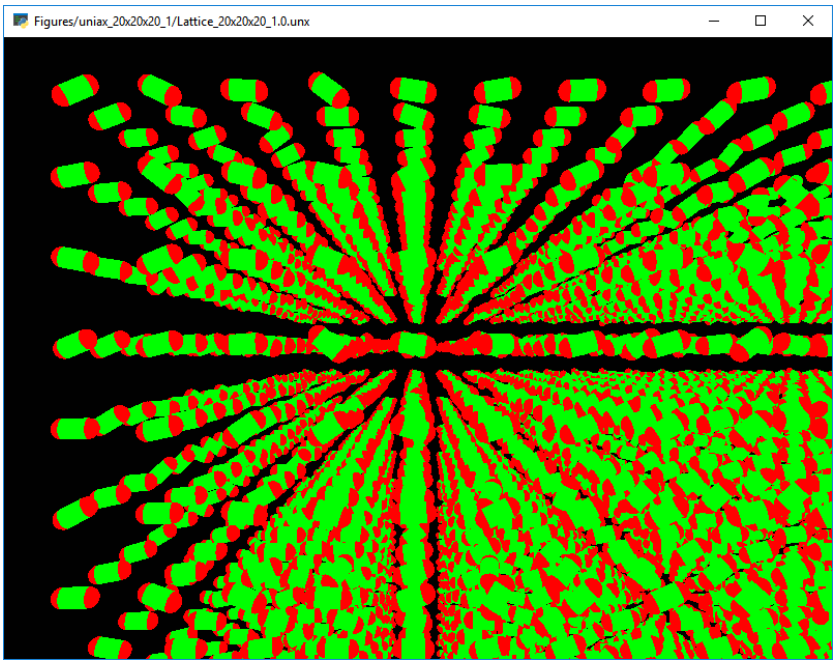
```
./show_lattice.py Figures/ising_10x10x10_2/Lattice_10x10x10_4.5.sng
```



Rysunek 7.5. Podgląd macierzy modelu Isinga.

Pliki .unx zawierają macierze modelu Lebwohla-Lashera.

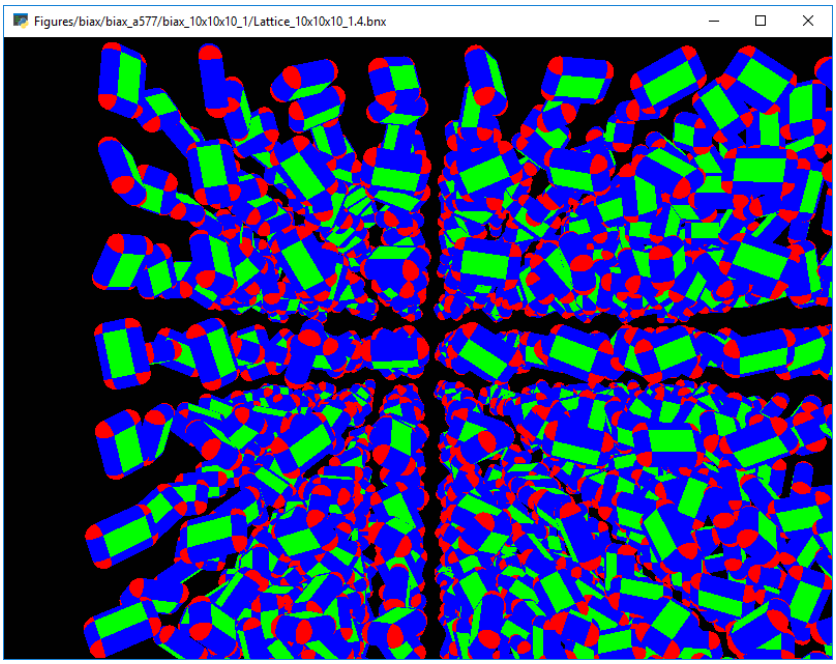
```
./show_lattice.py Figures/uniax_20x20x20_1/Lattice_20x20x20_1.0.unx
```



Rysunek 7.6. Podgląd macierzy modelu Lebwohla-Lashera.

Pliki .bnx zawierają macierze modelu molekuł dwuosiowych.

```
./show_lattice.py Figures/biax/biax_a577/biax_10x10x10_1/
Lattice_10x10x10_1.4.bnx
```



Rysunek 7.7. Podgląd macierzy modelu molekuł dwuosiowych.

Podczas podglądu można dowolnie sterować kamerą przy pomocy klawiatury i myszki.

Klawisze W/S/A/D odpowiednio przesuwają kamerę w górę, dół, lewo i prawo. Wymienione kierunki są wyznaczone zależnie od aktualnej orientacji kamery.

Kółko myszy pozwala na poruszanie kamery w przód i w tył.

Klawisze strzałek "orbitują" kamerę względem punktu arbitralnie wyznaczonego na środku widoku. Orbitująca kamera okrąża punkt odpowiednio wokół własnych osi X i Y i ustawia się, aby na ten punkt spoglądać. "Orbitowanie" można kontrolować również ruszając myszką przytrzymując prawy przycisk myszy.

Lewy przycisk myszy pozwala na obracanie kamery wokół własnych osi X i Y, za pomocą myszki.

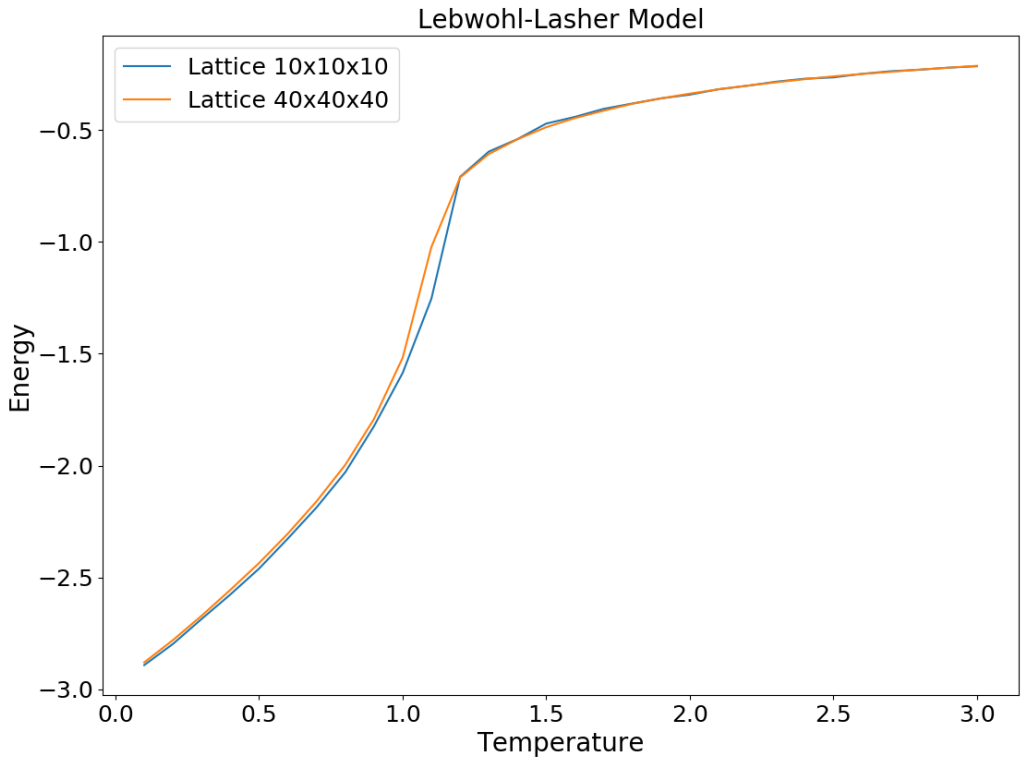
Przytrzymanie prawego i lewego przycisku myszy jednocześnie, pozwala na obrót kamery wokół własnej osi Z.

7.3. Analiza wyników

Ostatnią funkcjonalnością dostarczaną przez nasz symulator jest automatyczne generowanie wykresów na podstawie plików wynikowych zwracanych przez symulację.

Opracowano dwa skrypty do generowania wykresów: compare_figures.py oraz show_pyplot_figures.py.

Skrypt compare_figures.py jako parametry przyjmuje dwie ścieżki do plików wynikowych i wyświetla zestaw wykresów porównujących wyniki z obu symulacji. W obecnej wersji programu, skrypt compare_figures.py nie jest kompatybilny z plikami warmup oraz production.



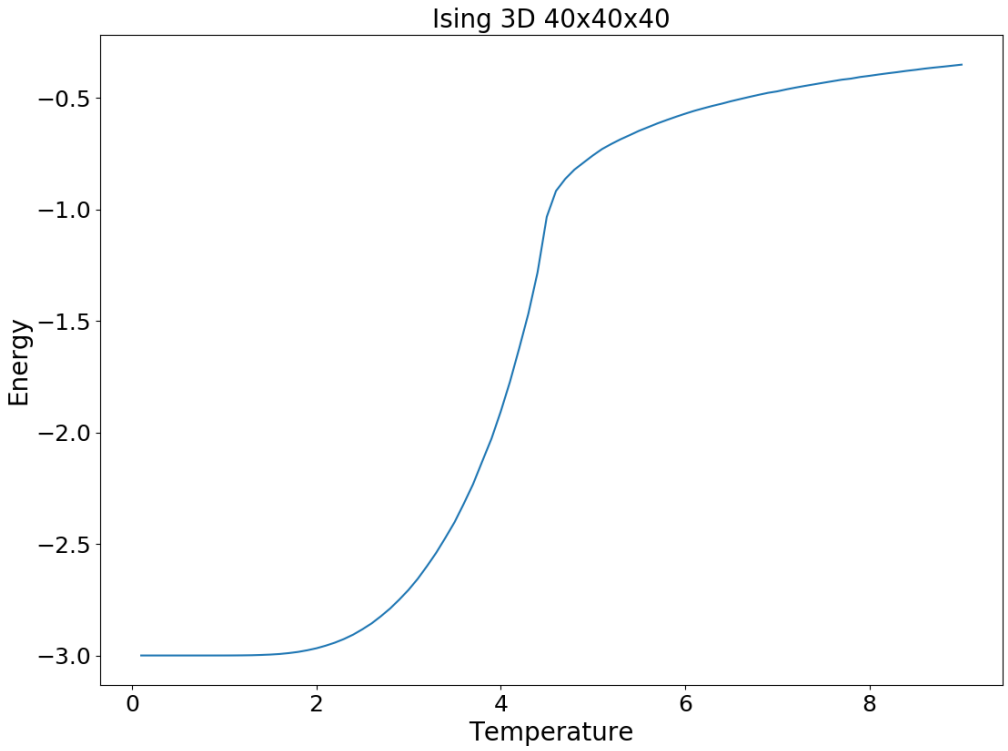
Rysunek 7.8. Wykres energii porównujący wyniki symulacji Lebwohla-Lashera dla dwóch różnych rozmiarów macierzy. Wyniki z większej macierzy są bardziej dokładne, przy założeniu, że wykonano odpowiednią ilość cykli warmup oraz production.

Podobnie jak w przypadku skryptów do podglądu stanu macierzy, skrypty generujące wykresy ładują następny wykres po zamknięciu aktualnie otwartego okna.

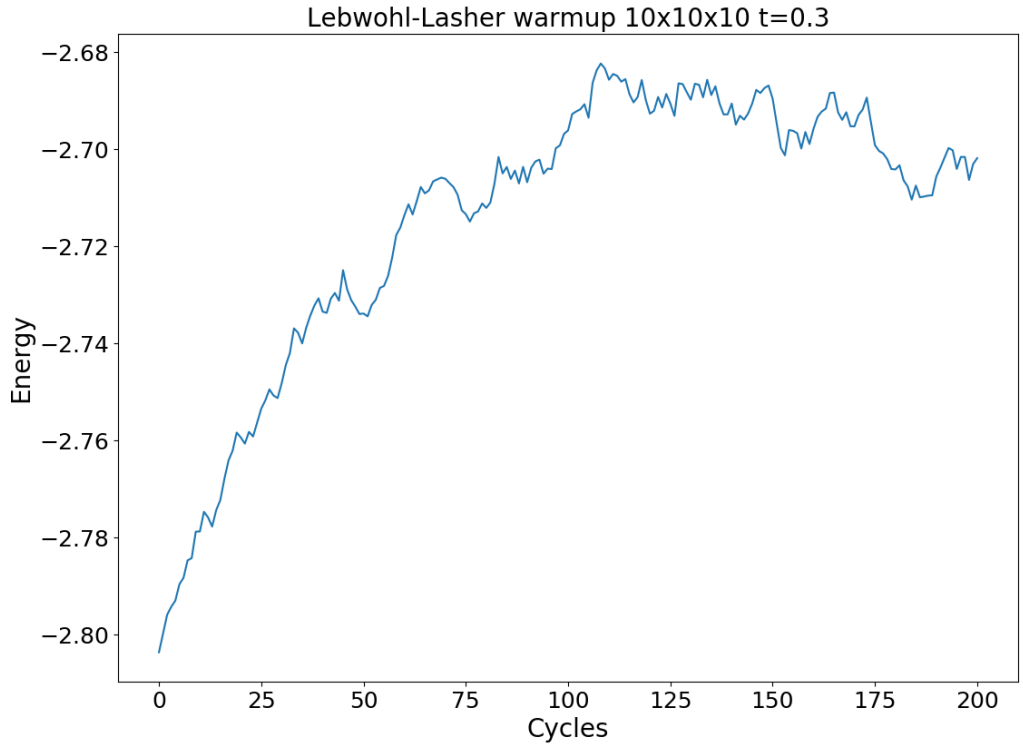
Skrypt show_pyplot_figures.py przyjmuje dowolną ilość argumentów, każdy z nich może być ścieżką do pliku o formacie .data.csv lub ścieżką do folderu.

Dla każdej ścieżki do pliku, skrypt wyświetla zestaw wykresów odpowiednich do danych zawartych wewnątrz pliku. To jakie dane program spodziewa się znaleźć wewnątrz pliku zależy od jego nazwy, tj. nazwy symulacji zawartej w nazwie pliku (ising, uniax, biax) oraz tego czy nazwa pliku zawiera słowa warmup bądź production. Przykłady na wykresach 7.9, 7.10 i 7.11.

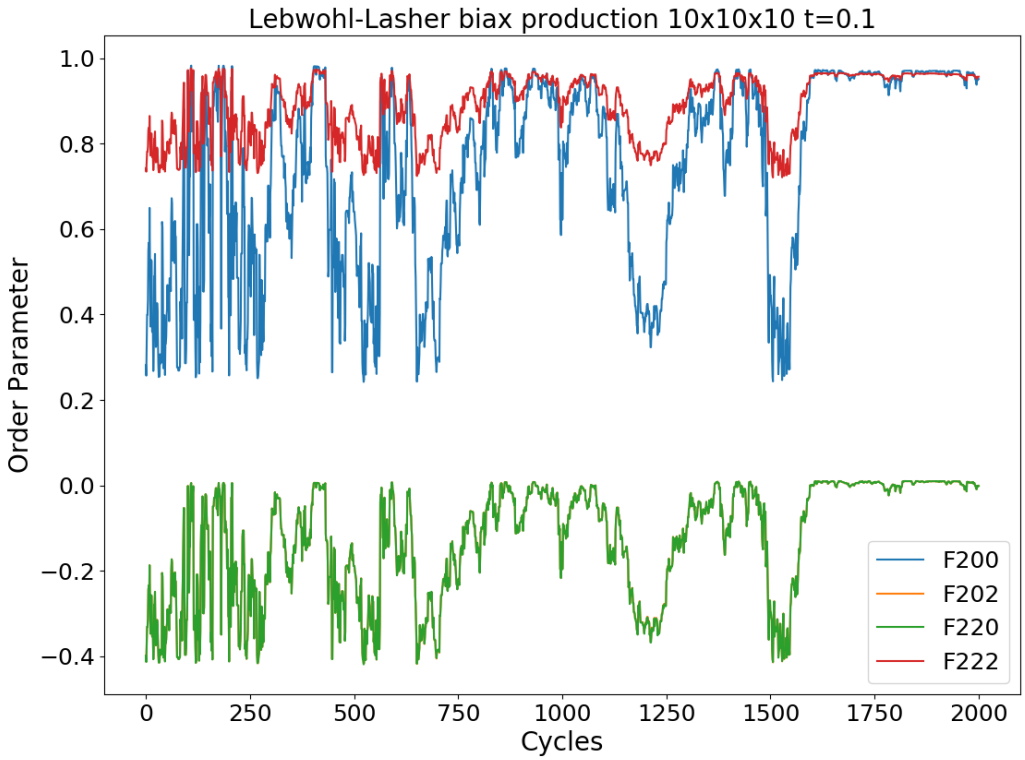
Dla każdej ścieżki do folderu, program wyświetla odpowiedni zestaw wykresów dla każdego pliku .data.csv zawartego wewnątrz folderu.



Rysunek 7.9. Wykres energii dla pliku *Ising3D_40x40x40.data.csv*.



Rysunek 7.10. Wykres energii dla pliku *uniax_warmup_10x10x10_0.3.data.csv*. Jeśli wartości nie oscylują (ani nie utrzymują stałej wartości) przed końcem wykresu, jest to wskazówka, że należy wydłużyć czas warmup.

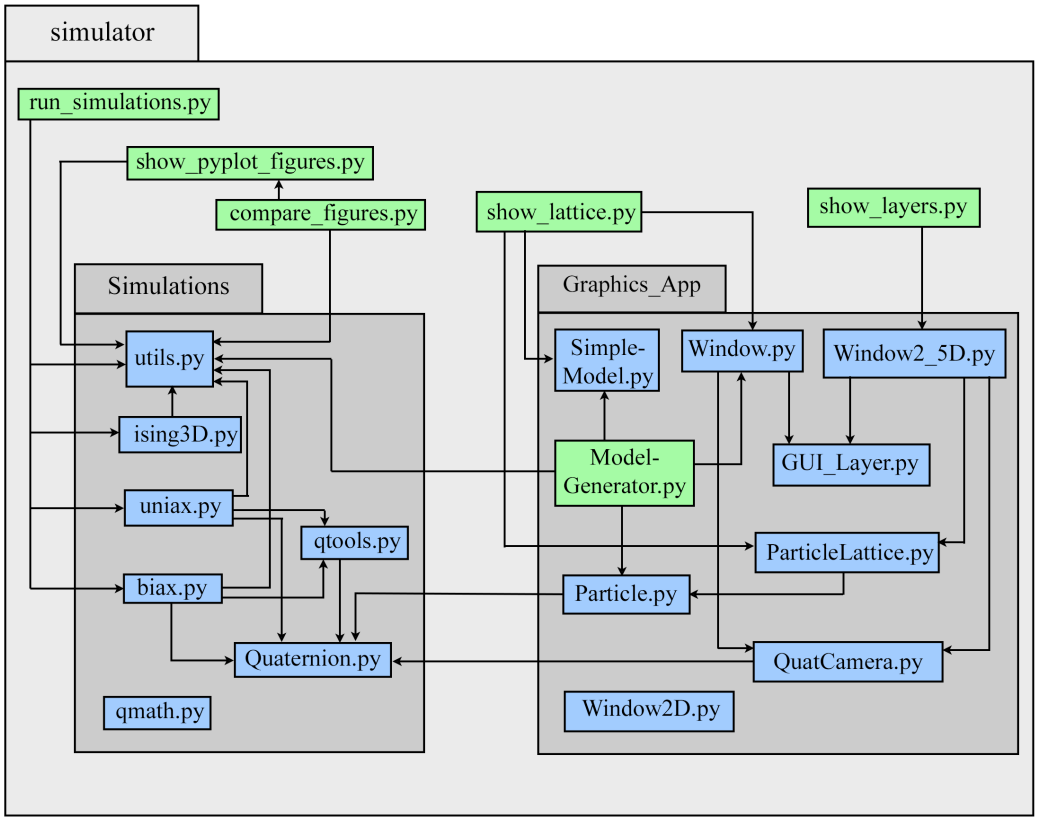


Rysunek 7.11. Wykres parametrów porządku dla pliku *biax_production_10x10x10_0.1.data.csv*. Wszystkie wartości powinny oscylować od początku wykresu. Jeśli tak nie jest, to oznacza, że należy wydłużyć czas warmup.

8. Implementacja aplikacji

Pierwotnym celem projektu było stworzenie biblioteki pythonowej implementującej kwaterniony. Projekt ten został rozszerzony i w docelowej postaci jest prototypem narzędzia do przeprowadzania symulacji Monte Carlo ciekłych kryształów i wizualizacji wyników.

W celu zrealizowania projektu zdecydowaliśmy się skupić na zidentyfikowaniu oraz zaimplementowaniu kluczowych funkcjonalności w jak najprostszej formie.



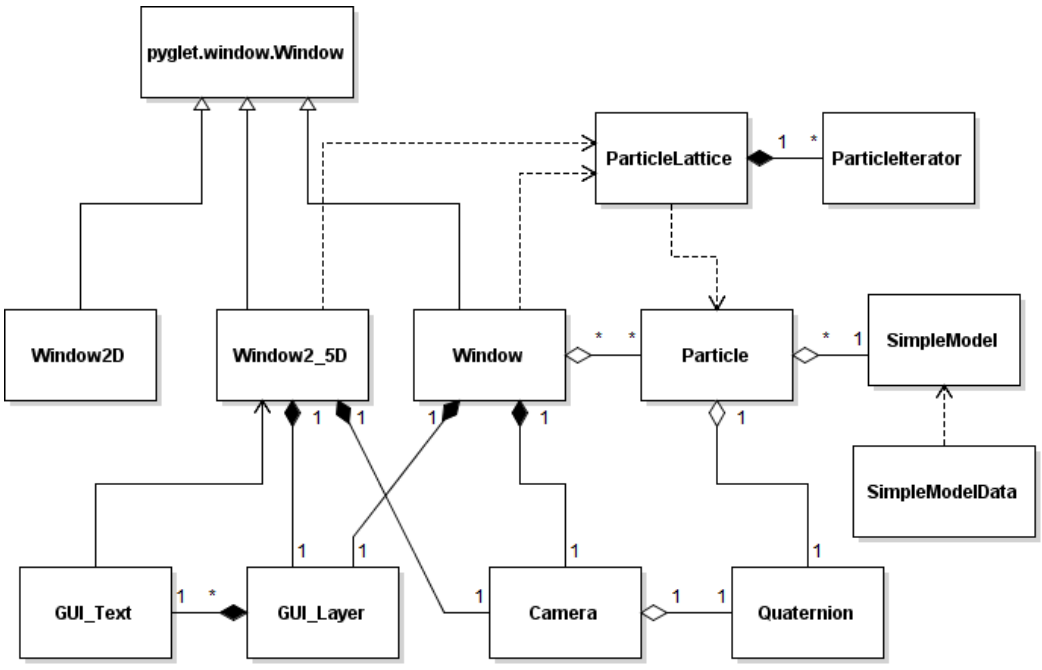
Rysunek 8.1. Diagram pakietów. Strzałki pokazują importy pomiędzy modułami zawartymi w projekcie. Importy z pakietów poza projektem zostały pominięte. Pliki skryptowe zaznaczono kolorem zielonym.

Aplikacja składa się z dwóch modułów oraz zestawu skryptów korzystających ze wspomnianych modułów. Moduł Simulations odpowiada za obliczenia związane z symulacjami oraz za zapis wyników. Moduł Graphics_App jest odpowiedzialny za wyświetlanie podglądu stanu macierzy.

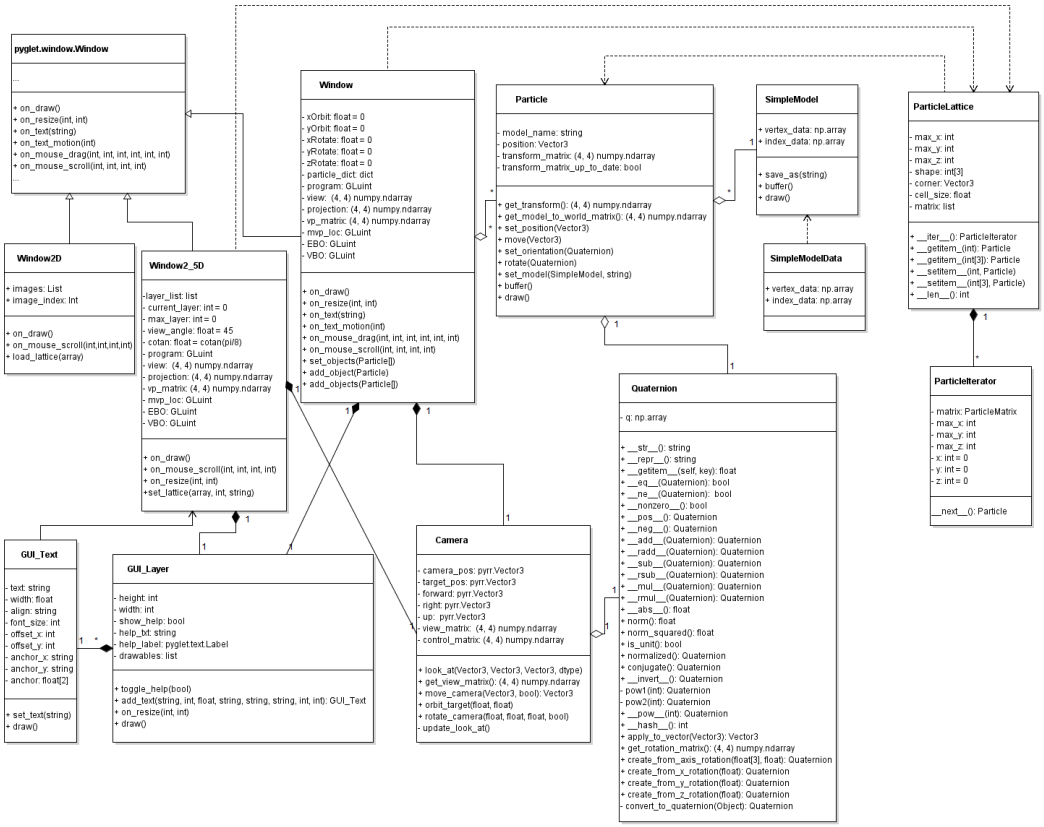
Ponieważ kwestia podglądu stanu macierzy była istotna dla weryfikacji poprawności działania symulacji (podgląd uporządkowania orientacyjnego molekuł dla różnych temperatur, sprawdzenie poprawności algorytmu losowania orientacji molekuł, etc.), jako pierwszą zaimplementowano cześć graficzną aplikacji.

8.1. Moduł Graphics_App

Moduł Graphics_App zawiera moduły definiujące klasy, skrypty oraz funkcje używane do wyświetlania interaktywnych graficznych reprezentacji stanu macierzy. Związki pomiędzy klasami w aplikacji przedstawiono na diagramach 8.2 i 8.3. Klasa Quaternion jako jedyna pochodzi z modułu Simulations, pozostałe zdefiniowano w module Graphics_App. Diagram 8.3 zwiera pełne opisy klas, natomiast diagram 8.2 został uproszczony dla czytelniejszego zobrazowania zależności między klasami.



Rysunek 8.2. Uproszczony diagram klas w aplikacji.



Rysunek 8.3. Diagram klas w aplikacji.

8.1.1. ModelGenerator

Moduł *Graphics_App.ModelGenerator* zawiera metody pozwalające na wygenerowanie modeli 3D wykorzystywanych przez skrypty show_lattice.py oraz show_layers.py. Opis algorytmów służących do tworzenia modeli 3D znajduje się w rozdziale 5.7.

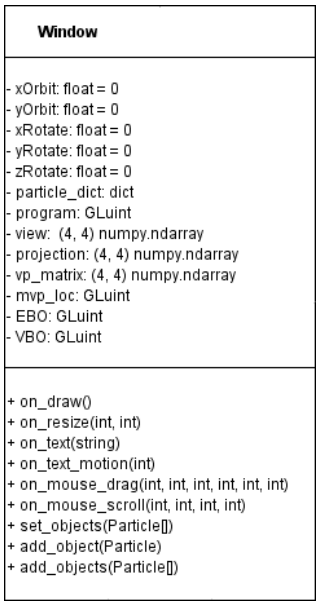
Metody z *ModelGenerator* zwracają utworzone modele w postaci obiektów klasy SimpleModel (z modułu *Graphics_App.SimpleModel*).

Plik *ModelGenerator.py* można też uruchomić jako skrypt. Po uruchomieniu skrypt generuje wszystkie modele używane obecnie w programie (plus, minus, sferocylinder i sferopłytkę), wyświetla ich podgląd na ekranie i zapisuje dane modeli w folderze *simulator/models*.

W celu wyświetlenia podglądu modeli na ekranie, skrypt importuje klasy Window (z *Graphics_App.Window*) oraz Particle (z *Graphics_App.Particle*). Klasa Window pozwala na stworzenie prostego okienka do wyświetlania grafiki 3D, klasa Particle pozwala na przekazanie do klasy Window obiektów zawierających dane o swoim modelu 3D, położeniu i orientacji.

W celu zapisu danych o modelach, skrypt wykorzystuje: metodę make_sure_dir_exists(), zaimportowaną z *Simulations.utils* do upewnienia się, że docelowy folder istnieje, oraz metodę save_as() z klasy Particle do zapisu danych o modelu do pliku.

8.1.2. Window



Rysunek 8.4. Klasa Window.

Moduł *Graphics_App.Window* zawiera klasę Window (dziedziczącą po klasie Window z modułu *pyglet.window*) przeznaczoną do wyświetlania prostej grafiki 3D.

Klasa Window korzysta:
— z kamery zaimportowanej z modułu *Graphics_App.QuatCamera*,
— z prostego GUI zaimportowanego z modułu *Graphics_App.GUI_Layer*,
— z klas i metod do obsługi wektorów i macierzy z pakietu *pyrr*,
— z narzędzi do budowy i obsługi okna z pakietu *pyglet*,
— z funkcji OpenGL z pakietu *PyOpenGL*,
— z definicji klasy Iterable z pakietu *collections*.

Klasa Window zawiera metody umożliwiające tworzenie okienka, wyświetlanie sceny, dodawanie obiektów do wyświetlanej sceny oraz sterowanie kamerą. Klasa została zaprojektowana z myślą o szerszym zastosowaniu, niż to jakie ma ona w obecnej wersji projektu, np. potrafi obsługiwać dynamiczne sceny ze zmienną ilością obiektów, korzystających z wielu różnych modeli 3D jednocześnie. W efekcie klasa ta nie jest idealnie zoptymalizowana pod jej użycie w naszym projekcie, ale jest łatwo rozszerzalna do użycia w przyszłych wersjach naszego projektu oraz w innych projektach.

Klasa Window wymaga, aby obiekty dodawane do wyświetlanej przez nią sceny posiadały:
— atrybut model_name, Window używa tego atrybutu do grupowania razem obiektów wykorzystujących ten sam model 3D w celach optymalizacji,
— metody **buffer**() oraz draw(), odpowiednio do buforowania modelu i rysowania obiektu na ekranie,
— metodę get_transform() zwracającą macierz opisującą położenie oraz orientację obiektu.

Konstruktor tworzy okno, obiekt klasy QuatCamera reprezentujący kamerę, oraz obiekt klasy GUI_Layer reprezentujący warstwę GUI. Inicjalizuje też serwer OpenGL wraz ze wszystkimi potrzebnymi zmiennymi do wyświetlenia sceny. Kompiluje też używane shadery, aby zdefiniować OpenGL'owy program używany do wyświetlania grafiki (patrz rozdział 5.6). W dalszej części opisu ten program będzie nazywany programem rysującym.

Listing 8.1. Konstruktor klasy Window.

```
__init__(self, width=1200, height=600, title='Preview',
    camera_pos=Vector3([2.0, 1.5, -10.0]),
    target_pos=Vector3([2.0, 1.5, -6.0]))
```

Metody on_text() oraz on_text_motion() obsługują zdarzenia związane z wciśnięciem klawiszy na klawiaturze, odpowiednio liter bądź klawiszy strzałek. Metody on_mouse_drag() i on_mouse_scroll() obsługują zdarzenia związane z ruchem myszy oraz kółka myszy. Wszystkie te metody są używane do sterowania ruchem kamerą (więcej o sterowaniu kamerą w rozdziale 7.2).

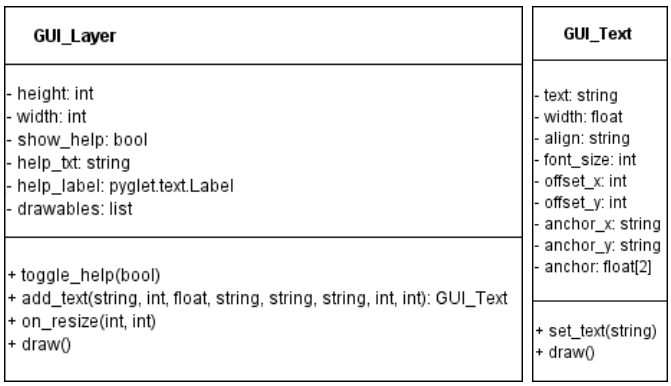
Metody on_text() i on_mouse_scroll() służą do sterowania położeniem kamery i wywołują metodę move_camera() na obiekcie kamery. Metody on_text_motion() oraz on_mouse_drag() służą do sterowania orientacją kamery, w tym celu zapisują dane o obrocie kamery do zmiennych, które zostaną wykorzystane do ustawienia jej orientacji przy następnym wywołaniu funkcji on_draw().

Metoda on_resize() obsługuje zdarzenie związane ze zmianą rozmiaru okna. Metoda ta dostosowuje rozmiar odpowiednich viewportów oraz wywołuje metodę on_resize() na obiekcie warstwy GUI.

Metoda draw() obsługuje zdarzenie związane z rysowaniem sceny, pyglet automatycznie wywołuje tę metodę w miarę potrzeby (np. po wywołaniu innej funkcji na oknie). W ten sposób, program aktualizuje widok wewnątrz okna wtedy i tylko wtedy, gdy ten widok miał szansę się zmienić.

Metoda oczyszcza scenę, ustawia program rysujący jako aktywny program, podpina odpowiednie bufory i w miarę potrzeby wykonuje obroty kamerą. Potem oblicza macierz ViewProjection i rysuje po kolei wszystkie obiekty w scenie, korzystając z ich macierzy Model do obliczenia ostatecznej macierzy MVP dla każdego obiektu. Obiekty są pogrupowane według używanych w nich modeli 3D, tak więc przed narysowaniem każdej grupy obiektów ich model zostaje zbuforowany. Takie podejście pozwala zminimalizować spowolnienie aplikacji przy rysowaniu scen wykorzystujących więcej niż jeden model 3D, takich jak np. podgląd stanu macierzy w modelu Isinga. Po narysowaniu wszystkich obiektów, program wywołuje metodę draw() na obiekcie klasy GUI_Layer.

8.1.3. GUI_Layer



Rysunek 8.5. Klasy GUI_Layer i GUI_Text.

Moduł *Graphics_App.GUI_Layer* zawiera klasy GUI_Layer oraz GUI_Text. Obiekty klasy GUI_Layer reprezentują warstwę GUI w aplikacji. Obiekty

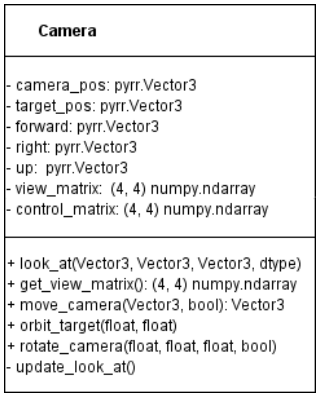
klasy GUI_Text to proste obiekty tekstowe, które automatycznie dostosowują swoje położenie na ekranie wraz ze zmianą rozmiaru warstwy GUI_Layer.

Klasy z modułu *Graphics_App.GUI_Layer* korzystają z metod i klas dostarczonych przez pakiety *pyglet* oraz *PyOpenGL*.

Moduł pyglet dostarcza metody znacząco ułatwiające implementację GUI. Niestety, dokumentacja pygleta nie wspomina o tym, że metody te do poprawnego działania wymagają użycia domyślnego programu rysującego OpenGL oraz jego domyślnych buforów. Metoda draw() obiektu GUI_Layer przywraca domyślny program, bufory oraz tablice atrybutów OpenGL przed rozpoczęciem rysowania.

W obecnej wersji programu, GUI_Layer obsługuje wyłącznie proste elementy tekstowe, ponieważ wyłącznie takie były potrzebne. Jednakże, dodanie nowych rodzajów elementów GUI do warstwy graficznej to tylko kwestia napisania nowych klas reprezentujących te obiekty.

8.1.4. QuatCamera



Rysunek 8.6. Klasa QuatCamera.

Moduł QuatCamera zawiera klasę implementującą tzw. kamerę, tj. obiekt używany do obliczania macierzy widoku na podstawie pozycji oraz orientacji wirtualnej kamery. Wiele implementacji kamer jest ograniczonych do konkretnego rodzaju ruchu, np. orbitowania wokół wyznaczonego punktu, w ograniczonym zakresie kątów widzenia lub tzw. ruchu oraz obrotu swobodnego wokół własnych osi. Przy okazji tego projektu chcieliśmy stworzyć ogólną implementację kamery, którą można by wykorzystać do uzyskania dowolnego rodzaju kontroli nad ruchem kamery.

Nasza implementacja kamery używa kwaternionów do wykonywania obrotów, a orientację kamery zapisuje w postaci trzech prostopadłych wektorów up, forward i right.

Metoda look_at() to standardowa metoda w wielu implementacjach kamer oraz modułach do obsługi macierzy 4×4. Jako parametry przyjmuje położenie kamery, punkt na który kamera ma "patrzeć" (target), oraz kierunek, który z perspektywy kamery ma być górą. Metoda odpowiednio ustawia kamerę i zwraca uzyskaną macierz widoku.

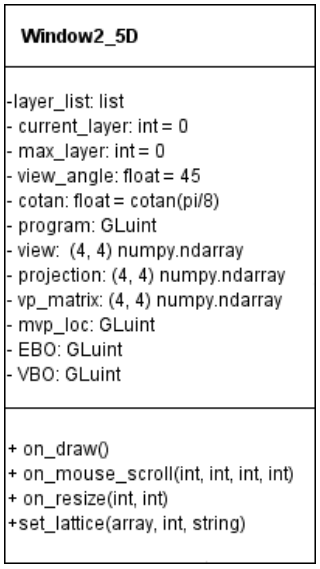
Metoda get_view_matrix() zwraca macierz widoku.

Metoda move_camera() przesuwa kamerę wraz z punktem, na który patrzy o określony wektor, wzdłuż własnych bądź globalnych osi, w zależności od wartości parametru use_global_coordinates. Metoda zwraca pozycję kamery po przesunięciu.

Metoda rotate_camera() obraca kamerę o podane kąty wokół własnych bądź globalnych osi, w zależności od wartości parametru use_world_axes.

Metoda orbit_target() pozwala na "orbitowanie" kamery wokół aktualnego punktu target. Orbitująca kamera okrąża punkt i automatycznie ustawia się tak, by na niego spoglądać.

8.1.5. Window2_5D



Rysunek 8.7. Klasa Window2_5D.

Moduł Graphics_App.Window2_5D zawiera klasę Window2_5D. Podobnie jak klasa Window, dziedziczy ona po klasie *pyglet.window.Window*. Klasa Window2_5D została stworzona z myślą o skrypcie *show_layers.py*. Pozwala na wyświetlenie podglądu macierzy molekuł podzielonego na warstwy (więcej o podglądzie warstw w rozdziale 7.2.1). W przeciwieństwie do okna Window, okno Window2_5D nie pozwala na ruch kamerą.

Klasa Window2_5D korzysta:

— z kamery z modułu *Graphics_App.QuatCamera*,
— z prostego GUI z modułu *Graphics_App.GUI_Layer*,
— z klas Particle oraz ParticleLattice z modułu *Graphics_App.ParticleLattice*,
— z klasy SimpleModel z modułu *Graphics_App.SimpleModel*,
— z klas i metod do obsługi wektorów oraz macierzy z pakietu *pyrr*,
— z narzędzi do budowy i obsługi okna z pakietu *pyglet*,
— z funkcji OpenGL z pakietu *PyOpenGL*.

Konstruktor klasy Window2_5D działa niemal identycznie jak konstruktor klasy Window, z tym że dodaje do warstwy GUI dodatkowy element tekstowy, pokazujący numer warstwy, którą obecnie wyświetlamy.

Metody on_resize() and on_mouse_scroll() też działają podobnie do tych w klasie Window, z tym że on_mouse_scroll() jest używany do przełączania pomiędzy podglądem różnych warstw, zamiast do sterowania kamerą.

Metoda set_lattice() przyjmuje jako argument tablicę numpy.ndarray, reprezentującą układ spinów bądź orientacji molekuł w próbce. Przyjmuje też rozmiar komórki dla docelowej sieci kubicznej oraz argument opisujący typ danych w tablicy.

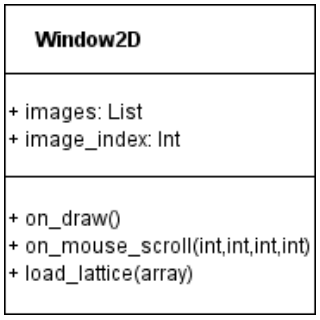
Na podstawie podanych informacji, metoda automatycznie rozmieszcza molekuły wewnątrz okienka, ustala jakich modeli 3D użyć do ich reprezentacji, dzieli sieć kubiczną na warstwy i ustala odpowiednią pozycję kamery, aby mogła mieć widok na całą warstwę jednocześnie.

Metoda on_draw() również działa podobnie jak w klasie Window, z tą różnicą, że tylko jedna warstwa macierzy jest wyświetlana. To, która warstwa macierzy jest aktualnie wyświetlana, jest kontrolowane przez funkcję on_mouse_scroll().

8.1.6. Window2D

Klasa Window2D z modułu *Graphics_App. Window2D* to klasa implementująca proste okienko wyświetlające podgląd stanu macierzy tworząc

do tego zestaw obrazów 2D reprezentujących stan poszczególnych warstw wewnątrz macierzy.



Rysunek 8.8. Klasa Window2D.

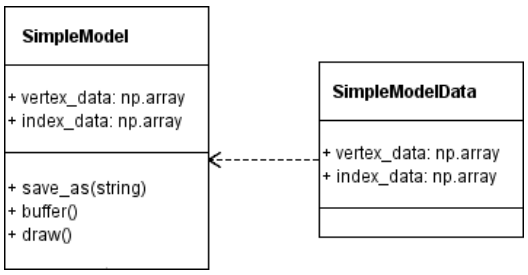
Klasa wykorzystuje klasy oraz metody zaimportowane z modułów *PIL.Image* oraz *pyglet* do tworzenia i wyświetlania grafiki 2D.

W aktualnej wersji programu klasa Window2D jest w stanie wyświetlać tylko podgląd macierzy modelu Isinga i nie jest wykorzystywana przez żaden skrypt. Obecnie podgląd warstw macierzy jest realizowany za pomocą klasy Window2_5D, wykorzystującej trójwymiarowe modele do reprezentacji stanu molekuł wewnątrz macierzy.

Oryginalnie planowaliśmy wykorzystać grafikę 2D i reprezentować stany molekuł za pomocą kolorów. Niestety, takie rozwiązanie okazało się trudne do zrealizowania dla modeli Lebwohla-Lashera oraz molekuł dwuosiowych. Problem tkwi w znalezieniu metody dobrze odwzorowującej przestrzeń orientacji molekuł na przestrzeń kolorów. Jest to trudne przy jednoczesnym zachowaniu właściwości symetrii orientacji molekuł i ciągłości. Z symetrii molekuł wynika, że orientacje o przeciwnym zwrocie są tożsame, zatem powinny być odwzorowane na ten sam kolor. Funkcja odwzorowująca powinna być ciągła, tj. każda para zbliżonych wartości z dziedziny powinna odpowiadać podobnym kolorom, bez przypadków granicznych tj. nieciągłości, dla których zbliżone orientacje odpowiadają kontrastującym kolorom.

Klasa Window2D została zachowana w projekcie jako prototyp, razem z prototypem funkcji z modułu *Simulations.utils* mapującej orientację molekuły jednoosiowej na kolor rgb. Skrypt *show_layer.py* również zawiera prototypy funkcji korzystających z Window2D.

8.1.7. SimpleModel



Rysunek 8.9. Klasy SimpleModel i SimpleModelData.

Moduł *Graphics_App.SimpleModel* zawiera dwie klasy: SimpleModel i SimpleModelData.

Klasa SimpleModelData to uproszczona wersja klasy SimpleModel, pozbawiona metod, przeznaczona do inicjalizacji obiektów klasy SimpleModel oraz zapisu danych modelu do pliku.

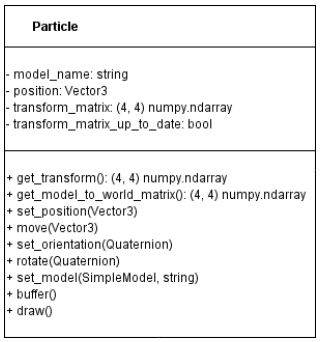
Klasa SimpleModel reprezentuje prosty model 3D w aplikacji. Przechowuje dane wierzchołków oraz indeksów modelu (patrz rozdziały 5.4.1 oraz 5.4.2) w dwóch tablicach typu numpy.array.

Metoda save_as() jako parametr przyjmuje ścieżkę do pliku. Metoda tworzy wyznaczony plik, a następnie zapisuje do niego dane modelu przy pomocy metody pickle.dump().

Metoda **buffer**() automatycznie buforuje dane do podpiętych pod program rysujący buforów oraz ustawia odpowiednie wskaźniki atrybutów.

Metoda draw() wywołuje odpowiednią dla modelu metodę rysującą.

8.1.8. Particle



Rysunek 8.10. Klasa Particle.

Moduł *Graphics_App.Particle* zawiera klasę Particle, reprezentującą swobodną molekułę do wyświetlenia na ekranie. Moduł importuje narzędzia do obsługi macierzy 4×4 i wektorów trójwymiarowych z pakietu *pyrr* oraz klasę Quaternion z modułu *Graphics_App.Quaternion*.

Obiekt Particle zawiera referencję do obiektu reprezentującego obiekt 3D (może to być SimpleModel bądź inna klasa implementująca metody **buffer**() i draw()). Przechowuje również nazwę modeli 3D (klasa Window używa tej informacji do grupowania molekuł), informację o pozycji molekuły (może to być pyrr.Vector3 albo numpy.array o długości 3) oraz aktualną orientację molekuły w postaci kwaternionu (klasa Quaternion).

Metoda get_transform_matrix() zwraca macierz modelu molekuły (patrz rozdział 5.5). Metoda get_model_to_world_matrix() jest aliasem tej metody.

Metody set_position(), move(), set_orientation() i rotate() pozwalają zmieniać odpowiednio pozycję oraz orientację molekuły. Podobnie set_model() pozwala zmieniać przypisany do molekuły obiekt modelu.

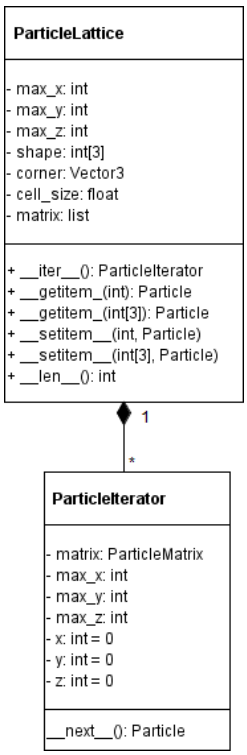
Metody **buffer**() oraz draw() wywołują odpowiadające metody na obiekcie modelu podpiętym do molekuły.

8.1.9. ParticleLattice

Początkowo planowano, aby część graficzna i część wykonująca symulacje Monte Carlo były bardziej ze sobą związane, co pozwoliłoby np. na podglądanie przebiegu symulacji na żywo lub po jej zakończeniu. Ostatecznie uznano, że takie rozwiązania nie mają większego sensu.

Pierwotnie, klasa ParticleLattice miała być strukturą danych używaną wspólnie przez obie części programu, takie rozwiązanie okazało się jednak niepotrzebne. Obecnie klasa ParticleLattice spełnia dwie role. Przede wszystkim, automatyzuje proces przypisywania molekułom pozycji w przestrzeni na podstawie ich indeksów w macierzy. Ponadto, pozwala rozwiązać problem, który pojawiał się przy próbie użycia tabel numpy.ndarray do przechowywania

obiektów Particle. Mianowicie, przy odczytywaniu zawartości tabeli program nie wykrywał poprawnie klasy przechowywanego obiektu.



Rysunek 8.11. Klasa ParticleLattice.

Klasa ParticleLattice używa zagnieżdżonych list do przechowywania macierzy obiektów Particle. Udostępnia metody pozwalające na odczytywanie oraz modyfikowanie zawartości macierzy, używając zarówno pojedynczego indeksu (kolejność elementu w macierzy), jak i zestawu 3 indeksów (numer kolumny, numer wiersza, numer warstwy). Udostępnia też metodę __iter__() pozwalającą na iterowanie w macierzy za pomocą pętli for.

Klasa ParticleIterator to specjalny iterator do klasy ParticleLattice zwracany przez metodę __iter__().

Moduł *Graphics_App.ParticleLattice* definiuje również funkcję default_position_function(), używaną przez konstruktor klasy ParticleLattice do ustalania pozycji w przestrzeni molekuł wewnątrz macierzy. Jako opcjonalny parametr w konstruktorze ParticleLattice można podać inną funkcję w celu uzyskania innego rozmieszczenia przestrzennego molekuł. Funkcja ta powinna przyjmować taki sam zestaw argumentów jak funkcja domyślna.

Argumenty funkcji default_position_function():
— pozycja jednego z kątów macierzy (ref_point),
— odległość sąsiednich molekuł między sobą (cell_size),
— indeksy molekuły (x, y, z).

8.2. Moduł Simulations

Moduł Simulations zawiera klasę Quaternion, definiuje też metody pozwalające na przeprowadzanie symulacji oraz zapisywanie danych do plików.

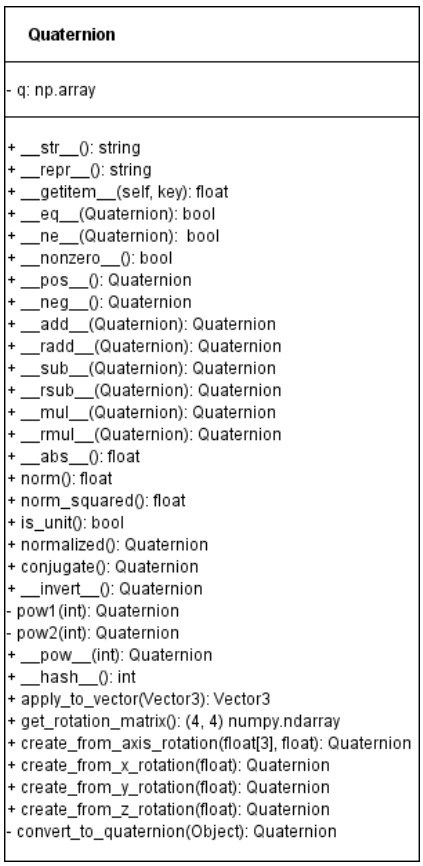
8.2.1. Quaternion

Moduł *Simulations.Quaternion* definiuje klasę Quaternion służącą do reprezentacji kwaternionu. Klasa przechowuje wartość kwaternionu w atrybucie __q, będącym wskaźnikiem do tabeli numpy.array. Pierwsza wartość w tabeli odpowiada części skalarnej kwaternionu, a pozostałe trzy odpowiadają części wektorowej.

Nasza implementacja kwaternionów zapewnia ich niezmienność oraz haszowalność. Metoda haszująca __hash__() wykorzystuje wbudowany mechanizm haszowania krotek w języku Python.

Metody implementujące podstawowe operacje na kwaternionach zostały opisane w tabeli 8.1. Wszystkie metody zwracające kwaternion zwracają nową instancję klasy.

Metoda __pow__(), opisana w tabeli, przyjmuje jako argument wyłącznie liczby całkowite, gdyż dla liczb rzeczywistych pojawiają się problemy z ciągłością. Stosowany jest algorytm potęgowania binarnego.



Rysunek 8.12. Klasa Quaternion.

Poza metodami wymienionymi w tabeli, klasa Quaternion implementuje również inne metody.

Metoda norm() jest aliasem metody __abs__(), zwraca normę kwaternionu.

Metoda norm_squared() zwraca normę podniesioną do drugiej potęgi.

Metoda is_unit() sprawdza czy kwaternion jest kwaternionem jednostkowym.

Metoda normalized() zwraca kwaternion jednostkowy odpowiadający kwaternionowi, na którym została wywołana.

Metoda apply_to_vector() przyjmuje jako parametr wektor wyrażony jako dowolna kolekcja składająca się z co najmniej 3 liczb rzeczywistych (np. krotka, lista, tablica numpy.array). Zwraca wektor obrócony przez kwaternion, w postaci 3 elementowej tablicy numpy.array.

Metoda get_ratation_matrix() zwraca macierz obrotu odpowiadającą kwaternionowi, w postaci tablicy 4×4 numpy.ndarray. Do obliczenia macierzy wykorzystywany jest wzór 3.38.

Metoda statyczna create_from_axis_rotation() przyjmuje jako parametry oś obrotu, wyrażoną jako dowolna kolekcja składająca się z przynajmniej 3 liczb rzeczywistych oraz kąt obrotu podany w radianach. Zwraca nowy kwaternion opisujący obrót wokół podanej osi, o podany kąt.

Metoda create_from_x_rotation() przyjmuje jako parametr kąt obrotu podany w radianach. Zwraca nowy kwaternion opisujący obrót wokół osi x o podany kąt.

Metoda create_from_y_rotation() przyjmuje jako parametr kąt obrotu podany w radianach. Zwraca nowy kwaternion opisujący obrót wokół osi y o podany kąt.

Metoda create_from_z_rotation() przyjmuje jako parametr kąt obrotu podany w radianach. Zwraca nowy kwaternion opisujący obrót wokół osi z o podany kąt.

Tabela 8.1. Podstawowe operatory kwaternionów. p i q są kwaternionami, n jest liczbą całkowitą nieujemną, a, b, c, d to liczby (**int**, **float**), z to liczba zespolona (**complex**).

Metoda	Znaczenie	Operacja
__init__	tworzenie kwaternionu	q = Quat(a, b, c, d)
__str__	tekstowa reprezentacja kwaternionu	**print** q
__repr__	tekstowa reprezentacja kwaternionu	**print repr**(q)
__getitem_	odczyt składowych kwaternionu	a, b, c, d = q a = q[0] b, c, d = q[1:]
__add__	dodawanie kwaternionów	p + q, p + a, p + z
__radd__	dodawanie kwaternionów	a + q, z + q
__sub__	odejmowanie kwaternionów	p − q, p − a, p − z
__rsub__	odejmowanie kwaternionów	a − q, z − q
__mul__	mnożenie kwaternionów	p ∗ q, p ∗ a, p ∗ z
__rmul__	mnożenie kwaternionów	a ∗ q, z ∗ q
__pos__	operator jednoargumentowy	+q
__neg__	operator jednoargumentowy	−q
__eq__	porównywanie kwaternionów	p == q
__ne__	porównywanie kwaternionów	p != q
__bool__	porównywanie kwaternionów	p != 0, **bool**(p)
__pow__	potęgowanie kwaternionu	q ** n, **pow**(q, n)
__abs__	wartość bezwzględna	**abs**(q)
__invert__	kwaternion odwrotny	~q
conjugate	sprzężenie kwaternionu	q.conjugate()

8.2.2. qtools

Moduł *Simulations.qtools* definiuje dwie dodatkowe funkcje używające kwaternionów.

Funkcja random_quat_uniax() zwraca przypadkową orientację dla molekuły jednoosiowej, natomiast random_quat_biax() zwraca przypadkową orientację dla molekuły dwuosiowej.

8.2.3. qmath

Moduł *Simulations.qmath* implementuje funkcje opisane w rozdziale 3.5. Każda z funkcji zwraca nową instancję kwaternionu.

8.2.4. utils

Moduł *Simulations.utils* definiuje funkcje obsługujące zapisywanie i odczytywanie informacji z plików. Definiuje też wczesny prototyp funkcji przeliczającej kwaterniony na kolory (patrz rozdział 8.1.6).

Funkcja make_sure_dir_exists() jako parametr przyjmuje ścieżkę do folderu i sprawdza czy folder istnieje. Jeśli dany folder nie istnieje, funkcja tworzy wszystkie brakujące foldery w ścieżce. Na przykład, dla ścieżki `'C:`

`/document/directory'`, jeśli folder `'C:/document'` nie istnieje, funkcja tworzy `'C:/document'` a później `'C:/document/directory'`.

Funkcja save_columns_to_file() jako parametry przyjmuje ścieżkę do pliku oraz listę list. Funkcja tworzy dany plik i zapisuje dane z każdej listy do osobnej kolumny wewnątrz pliku.

Funkcja append_line_to_file() jako parametr przyjmuje ścieżkę do istniejącego pliku oraz listę wartości. Funkcja dodaje nową linijkę do pliku i zapisuje każdą wartość z listy w osobnej kolumnie.

Funkcja read_columns_from_file() jako parametr przyjmuje ścieżkę do pliku tekstowego. Funkcja odczytuje dane z pliku i zwraca krotkę zawierającą listy danych z każdej kolumny wewnątrz pliku.

8.2.5. Symulacje

Moduły *Simulations.ising3D*, *Simulations.uniax*, oraz *Simulations.biax* zawierają zestawy funkcji pozwalających na przeprowadzenie odpowiednio symulacji modelu Isinga, modelu Lebwohl-Lashera lub modelu molekuł dwuosiowych.

We wszystkich trzech symulacjach macierz molekuł jest reprezentowana przez tablicę numpy.array o wymiarach $x \times y \times z$, gdzie x, y, z to wymiary macierzy. W przypadku modelu Isinga tablica zawiera liczby całkowite opisujące spin danej molekuły: 1 dla spinu w górę, -1 dla spinu w dół. W przypadku pozostałych dwóch modeli, tablica zawiera kwaterniony (reprezentowane przez obiekty klasy Quaternion), opisujące orientacje molekuł.

W przebiegu każdej ze wspomnianych symulacji możemy wyróżnić 3 etapy: inicjalizację macierzy, etap warmup i etap produkcyjny.

Podczas inicjalizacji macierzy tworzona jest tabela reprezentująca macierz i dobierane są początkowe wartości zawarte w tabeli.

Podczas etapu warmup używamy metod Monte Carlo w celu doprowadzenia macierzy do stanu równowagi termodynamicznej dla danej temperatury. Etap warmup jest podzielony na cykle. Ilość cykli warmup jest arbitralnie wybierana przez użytkownika. Każdy cykl jest podzielony na iteracje, liczba iteracji w cyklu odpowiada liczbie molekuł w macierzy. W każdej iteracji losowana jest jedna molekuła z macierzy, dla tej molekuły losowana jest nowa orientacja (lub w przypadku modelu Isinga, odwracany jest jej spin), potem obliczana jest zmiana energii układu wywołana zmianą stanu molekuły. Jeśli nowa energia układu jest mniejsza od poprzedniej, zmiana orientacji (lub spinu) jest akceptowana. Jeśli nowa energia układu jest większa, prawdopodobieństwo zaakceptowania zmiany jest opisane wzorem 2.2. Pod koniec każdego cyklu, obecna energia układu oraz inne parametry mierzone w danej symulacji, zostają zapisane do odpowiednich list. Dane z tych list pomagają prześledzić przebieg etapu warmup i dobrać odpowiednią ilość cykli warmup dla przyszłych symulacji.

Funkcje obsługujące etap warmup przyjmują jako parametry obiekt macierzy, temperaturę, dla której ma zostać przeprowadzona symulacja, ilość cykli przeznaczonych na etap warmup, oraz opcjonalnie ścieżkę do folderu, w jakim powinien zostać umieszczony plik wynikowy (save_directory). Jeśli parametr save_directory został określony, funkcja zapisuje dane zebrane pod

koniec każdego cyklu w pliku wynikowym. To, jakie dane są zbierane pod koniec każdej symulacji zostało opisane w rozdziale 7.1.2. Funkcja zwraca jako wynik dane zebrane pod koniec ostatniego cyklu.

Podczas etapu produkcyjnego, program zachowuje się podobnie jak podczas etapu warmup, z tą różnicą, że pod koniec każdego cyklu zbieranych jest więcej danych na temat stanu macierzy. Po zakończeniu ostatniego cyklu obliczana jest średnia z danych z każdej listy, te średnie posłużą do wyliczenia ostatecznych wyników symulacji.

Funkcje obsługujące etap produkcyjny przyjmują jako parametry obiekt macierzy, temperaturę, dla której ma zostać przeprowadzona symulacja, ilośc cykli przeznaczonych na etap warmup, ilośc cykli przeznaczonych na etap produkcyjny. Funkcja przyjmuje również opcjonalne zmienne warmup_save_directory oraz production_save_directory, na podstawie których decyduje czy i gdzie ma zapisać dane z obu etapów. Jeśli ilość cykli przeznaczonych na warmup jest większa od zera, funkcja (produkcyjna) uruchamia funkcję warmup i pobiera z niej początkowe dane. W przeciwnym wypadku funkcja oblicza początkowe dane na podstawie stanu macierzy.

8.2.5.1. Symulacja modelu Isinga

Moduł *Simulations.ising3D* definiuje funkcje umożliwiające przeprowadzenie symulacji dla trójwymiarowego modelu Isinga.

Funkcje ising_random_spins(), ising_organized_spins() i ising_checkered_spins() obsługują etap inicjalizacji macierzy.

Funkcja ising_random_spins() jako parametry przyjmuje wymiary macierzy i zwraca macierz o przypadkowo wylosowanych spinach.

Funkcja ising_organized_spins() przyjmuje również opcjonalny parametr spin_value o domyślnej wartości 1. Zwraca macierz, w której wszystkie spiny to spin_value.

Funkcja ising_checkered_spins() przyjmuje te same parametry co funkcja ising_random_spins() i zwraca macierz, w której kolejne spiny są na zmianę w górę (1) i w dół (-1).

Funkcja ising3D_warmup() obsługuje etap warmup. Przyjmuje parametry opisane w rozdziale 8.2.5. Oprócz tego przyjmuje opcjonalny parametr prob_list będący listą prawdopodobieństw przyjęcia nowej konfiguracji molekuł, na podstawie zmiany energii spowodowanej przez zmianę spinu. Jeśli parametr ten nie został podany, funkcja automatycznie generuje własną listę prawdopodobieństw. Ta lista jest używana później w programie podczas podejmowania decyzji czy przyjąć, czy odrzucić nowy spin. Funkcja zwraca jako wynik energię i magnetyzację układu spinów opisanego przez macierz.

Funkcja ising3D_simulation() obsługuje etap produkcyjny. Przyjmuje parametry opisane w rozdziale 8.2.5 oraz opcjonalny parametr prob_list opisany powyżej. Funkcja zwraca jako wynik gęstość energii, magnetyzację, ciepło właściwe, oraz podatność magnetyczną układu spinów opisanego przez macierz.

Funkcja gen_prob_list() zwraca wspomnianą wcześniej listę prawdopodobieństw, wygenerowaną dla podanej temperatury.

Funkcje calc_energy() i calc_magnetyzation() zwracają odpowiednio energię i magnetyzację układu spinów opisanego przez podaną macierz.

8.2.5.2. Symulacja modelu Lebwohla-Lashera

Moduł *Simulations.uniax* definiuje funkcje umożliwiające przeprowadzenie symulacji dla modelu Lebwohla-Lashera.

Funkcje uniax_random_spins() i uniax_organized_spins() obsługują etap inicjalizacji macierzy.

Funkcja uniax_random_spins() jako parametry przyjmuje wymiary macierzy i zwraca macierz zawierającą kwaterniony opisujące przypadkowo wylosowane orientacje dla molekuł jednoosiowych.

Funkcja uniax_organized_spins() jako parametry przyjmuje wymiary macierzy oraz opcjonalny parametr orientation o domyślnej wartości Quaternion(1). Zwraca macierz, w której wszystkie orientacje mają wartość orientation.

Funkcja uniax_warmup() obsługuje etap warmup. Przyjmuje wyłącznie parametry opisane w rozdziale 8.2.5. Jako wynik zwraca energię, parametr porządku oraz tensor porządku układu molekuł opisanego przez macierz.

Funkcja uniax_simulation() obsługuje etap produkcyjny. Przyjmuje wyłącznie parametry opisane w rozdziale 8.2.5. Jako wyniki zwraca gęstość energii, parametr porządku, ciepło właściwe oraz podatność parametru porządku na zmiany dla układu molekuł opisanego przez macierz.

Funkcja calc_one_spin() jako parametr przyjmuje orientację pojedynczej molekuły i zwraca tensor porządku obliczony dla tej jednej molekuły. Funkcja calc_order_tensor() jako parametr przyjmuje obiekt macierzy, oblicza i zwraca średnią z wyników funkcji calc_one_spin() dla wszystkich orientacji zawartych w macierzy. Średnia ta stanowi tensor porządku dla całej macierzy.

Funkcja calc_energy() oblicza i zwraca energię dla podanej macierzy. Wykorzystuje do tego funkcję interaction(), która dla podanych orientacji dwóch sąsiednich molekuł oblicza energię oddziaływania między nimi. Ta funkcja jest wykorzystywana również przy obliczaniu zmiany energii spowodowanej zmianą orientacji molekuły podczas symulacji.

8.2.5.3. Symulacja modelu molekuł dwuosiowych

Moduł *Simulations.biax* definiuje funkcje umożliwiające przeprowadzenie symulacji dla modelu molekuł dwuosiowych.

Funkcje biax_random_spins() i biax_organized_spins() działają podobnie do odpowiednio uniax_random_spins() i uniax_organized_spins(), z tą różnicą, że biax_random_spins() używa bardziej złożonej funkcji losującej spin niż jej odpowiednik z modułu *Simulations.uniax*.

Funkcja biax_warmup() obsługuje etap warmup. Przyjmuje parametry opisane w rozdziale 8.2.5 oraz parametr zannoni_parameter opisujący stopień dwuosiowości molekuły. Ten parametr został wcześniej opisany w tej pracy jako parametr a we wzorze 2.17. Funkcja zwraca jako wyniki wartości obliczone dla układu molekuł opisanego daną macierzą:

— energię,
— trzy tensory porządku: Q^{ll}, Q^{mm}, Q^{nn},
— cztery parametry porządku: $\langle F_{00}^{(2)} \rangle$, $\langle F_{02}^{(2)} \rangle$, $\langle F_{20}^{(2)} \rangle$, $\langle F_{22}^{(2)} \rangle$.

Funkcja biax_simulation() obsługuje etap produkcyjny. Przyjmuje parametry opisane w rozdziale 8.2.5 oraz parametr zannoni_parameter. Jako wyniki zwraca:

— gęstość energii,
— parametry porządku: $\langle F_{00}^{(2)} \rangle$, $\langle F_{02}^{(2)} \rangle$, $\langle F_{20}^{(2)} \rangle$, $\langle F_{22}^{(2)} \rangle$,
— ciepło właściwe.

Funkcje calc_energy() oraz interaction() działają podobnie do ich odpowiedników z modułu *Simulation.uniax*, z tym że przyjmują również zannoni_parameter jako parametr i wykonują bardziej złożone obliczenia.

Uporządkowanie układu molekuł w modelu molekuł dwuosiowych jest opisywane przez trzy tensory Q^{ll}, Q^{mm}, Q^{nn}. Każdy z tych tensorów ma własną funkcję obliczającą go dla pojedynczej molekuły, jak i dla całej macierzy:
— Q^{ll}: calc_one_spin_ll(), calc_order_tensor_ll(),
— Q^{mm}: calc_one_spin_mm(), calc_order_tensor_mm(),
— Q^{nn}: calc_one_spin_nn(), calc_order_tensor_nn().
Funkcje te działają analogicznie do opisanych wcześniej funkcji z modułu *Simulations.uniax*, czyli calc_one_spin() i calc_order_tensor().

Na podstawie obliczonych w ten sposób tensorów funkcja calc_Fjmn() oblicza parametry porządku $\langle F_{00}^{(2)} \rangle$, $\langle F_{02}^{(2)} \rangle$, $\langle F_{20}^{(2)} \rangle$, $\langle F_{22}^{(2)} \rangle$. Wzory, na podstawie których obliczane są wspomniane tensory oraz parametry porządku zostały opisane w rozdziale 2.6.

8.3. Skrypty

Bardziej dokładny opis działania skryptów pod kątem sposobu uruchomienia, przyjmowanych argumentów, oraz wyników działania poszczególnych skryptów znajduje się w rozdziale 7. W tym rozdziale uzupełnimy opis skryptów o pewne szczegóły implementacyjne, które nie były konieczne do opisu sposobu korzystania ze skryptów. Wyjaśnimy też pewne decyzje projektowe dotyczące implementacji danych skryptów.

8.3.1. Uwagi ogólne

Skrypty w tym projekcie korzystają z tzw. importów względnych (ang. *relative imports*). W Pythonie 3 importy względne często nie działają poprawnie i mogą być zależne od tego z jakiego folderu użytkownik wywołał dany skrypt. W celu obejścia tego problemu można tymczasowo dodać ścieżkę do naszego modułu do listy sys.path. Przykład poniżej.

Listing 8.2. Dodawanie modułu Graphics_App do sys.path.

```
source_root = os.path.abspath(os.path.dirname(__file__))
graph_app_module_path =
    os.path.abspath(os.path.join(source_root, 'Graphics_App'))
project_root =
    os.path.abspath(os.path.join(source_root, '..'))
print(graph_app_module_path)
sys.path.insert(0, graph_app_module_path)
```

8.3.2. Skrypt run_simulations.py

Skrypt jako parametr wywołania przyjmuje listę ścieżek do plików konfiguracyjnych o rozszerzeniu .py. Użycie plików pythonowych jako plików konfi-

guracyjnych ma kilka zalet. Przede wszystkim, pozwala nam załadować dane z pliku nie używając do tego parsera, a użycie parsera wiąże się z pewnymi trudnościami i ograniczeniami. Taki parser należałoby albo napisać, albo znaleźć moduł pythonowy implementujący parser najbardziej odpowiadający naszym potrzebom. Po drugie, bylibyśmy ograniczeni co do struktur danych wspieranych przez format pliku konfiguracyjnego oraz możliwości wybranego parsera. Użycie dynamicznie ładowanych plików pythonowych pozwala nam na skorzystanie z dowolnej struktury danych wspieranej przez Pythona bądź dowolną z jego bibliotek. Pozwala nam też na proceduralne generowanie danych, np. tworzenie listy temperatur za pomocą metody numpy.linspace().

Dla każdej ścieżki podanej jako parametr wywołania, skrypt dynamicznie ładuje podany plik jako moduł, w poniższy sposób.

Listing 8.3. run_simulations.py: dynamiczne ładowanie modułów

```
spec = importlib.util.spec_from_file_location("config", path)
config = importlib.util.module_from_spec(spec)
spec.loader.exec_module(config)
```

Po wywołaniu powyższych funkcji many dostęp do zawartości wczytanego modułu za pośrednictwem zmiennej config. Możemy na przykład, pobrać z modułu zmienną zwierającą słownik definiujący konfigurację.

```
configuration = config.configuration
```

Po pobraniu konfiguracji, program sprawdza wartości odpowiednich słów kluczowych i na podstawie tego czy wartość input_dir została zdefiniowana w konfiguracji, uruchomia odpowiednio run_sims() lub run_sims_based_on_dir().

8.3.2.1. Opis działania funkcji run_sims()

Funkcja run_sims() przyjmuje jako parametry wywołania:
— typ symulacji (simulation_type),
— sposób inicjalizacji (init_type),
— rozmiar macierzy podany jako krotka lub lista trzech int (lattice_size),
— listę temperatur (temperature_list),
— liczbę cykli produkcyjnych (production_cycles),
— liczbę cykli warmupowych (warmup_cycles),
— folder do zapisu wyników (out_put_dir),
— opcjonalne flagi warmup_output oraz production_output informujące program o tym czy ma zapisywać dane z przebiegu etapów warmup oraz production (więcej o zapisywanych danych w rozdziale 7.1.2),
— opcjonalny parametr load_from, wymagany jeśli simulation_type ma wartość 'load',
— opcjonalny parametr zannoni_parameter, wymagany jeśli simulation_type ma wartość 'biax'.

Na początek program dobiera metodę do inicjalizacji macierzy na podstawie wartości parametrów simulation_type oraz init_type. Na przykład, dla typu symulacji 'biax' oraz sposobu inicjalizacji 'random' program używa funkcji biax_random_spins() z pakietu *Simulations.biax* do zainicjalizowania macierzy. Jeśli parametr init_type ma wartość 'load' program wczytuje początkowy

stan macierzy z pliku wskazanego przez parametr load_from będący ścieżką do pliku. Używa do tego funkcji load() z pakietu *pickle*.

Na podstawie parametru simulation_type program dobiera też funkcję obsługującą symulację oraz rozszerzenie plików do zapisu stanów macierzy w każdej symulacji. Na przykład, dla symulacji typu 'biax': biax_simulation() z pakietu *Simulations.biax* oraz rozszerzenie .bnx.

Po inicjalizacji macierzy oraz dobraniu typu symulacji, dla każdej temperatury z listy temperature_list, program uruchamia odpowiednią symulację wykorzystując obecny stan macierzy. Stan macierzy nie jest ponownie inicjalizowany pomiędzy symulacjami. Po zakończeniu każdej symulacji program dopisuje uzyskane wyniki do pliku wynikowego (patrz rozdział 7.1.2).

8.3.2.2. Opis działania funkcji run_sims_based_on_dir()

Funkcja run_sims_based_on_dir() przyjmuje jako parametrywywołania:
— folder używany do pobrania początkowych stanów macierzy (input_dir),
— folder do zapisu wyników (out_put_dir),
— liczbę cykli produkcyjnych (production_cycles),
— liczbę cykli warmupowych (warmup_cycles),
— opcjonalne flagi warmup_output i production_output informujące program o tym czy ma zapisywać dane z przebiegu etapów warmup oraz production (więcej o zapisywanych danych w rozdziale 7.1.2),
— opcjonalny parametr zannoni_parameter, wymagany jeśli simulation_type ma wartość 'biax'.

Na początek, program pobiera informacje na temat rodzaju symulacji i rozmiaru macierzy z nazwy folderu wyznaczonego przez parametr input_dir.

Następnie, dla każdego pliku macierzy (tj. o odpowiednim dla danej symulacji rozszerzeniu) zawartego w folderze input_dir, program pobiera z niego stan macierzy. Na podstawie nazwy wspomnianego pliku wyznacza również temperaturę symulacji. Dla każdej pobranej w ten sposób macierzy przeprowadza skonfigurowaną symulację i dopisuje zwrócone wyniki do pliku wynikowego (patrz rozdział 7.1.2).

8.3.3. Skrypt show_pyplot_figures.py

Skrypt show_pyplot_figures.py jako parametr wywołania przyjmuje listę ścieżek. Każda z tych ścieżek może prowadzić do pliku o rozszerzeniu .data.csv bądź do folderu zawierającego takie pliki.

Funkcja show_figures() dla podanej ścieżki sprawdza czy prowadzi ona do pliku, czy do folderu. Jeśli ścieżka prowadzi do pliku, zostaje przekazana jako parametr do funkcji show_figure_from_data_file(). Jeśli ścieżka prowadzi do folderu, funkcja wyszukuje wszystkie pliki o rozszerzeniu .data.csv i każdy z nich przekazuje do funkcji show_figure_from_data_file().

Funkcja show_figure_from_data_file() na podstawie nazwy pliku ustala jaki typ danych ona zawiera i odpowiednio wywołuje jedną z dziewięciu funkcji rysujących:
— ising_warmup_from_data_file(),
— ising_production_from_data_file(),
— ising_figures_from_data_file(),

— uniax_warmup_from_data_file(),
— uniax_production_from_data_file(),
— uniax_figures_from_data_file(),
— biax_warmup_from_data_file(),
— biax_production_from_data_file(),
— biax_figures_from_data_file().

Wszystkie powyższe funkcje wykorzystują funkcję figure_from_data(). Wywołuje ona serię funkcji z modułu matplotlib.pyplot pozwalającą wyświetlić wykres obrazujący jedną zależność. Parametrami funkcji figure_from_data() są:
— lista argumentów x (x_data),
— lista wartości y odpowiadająca danym wartościom x (y_data),
— tytuł wykresu (title),
— etykieta osi x (x_label),
— etykieta osi y (y_label).

Funkcja multiline_figure() jest wykorzystywana przez funkcje rysujące wykresy 'biax', tj. wykresy dla symulacji molekuł dwuosiowych oraz przez funkcje ze skryptu compare_figures.py. Pozwala na przedstawienie wielu zależności na jednym wykresie. W naszych skryptach jest wykorzystywana do porównywania wyników symulacji dla różnych rozmiarów macierzy na jednym wykresie oraz do wyświetlania zależności wszystkich czterech parametrów porządku w modelu 'biax' od temperatury na wspólnym wykresie. Jako argumenty przyjmuje listę krotek oraz zbiór nazwanych argumentów (kwargs). Każda krotka z listy zawiera w sobie dwie listy oraz jeden ciąg znaków. Listy zawarte w krotce odpowiadające argumentom x_data i y_data z funkcji figure_from_data(). Ciąg znaków zawarty w krotce posłuży do oznaczenia w legendzie wykresu zależności przedstawionych na wykresie. Nazwane argumenty są odpowiednikami parametrów title , x_label oraz y_label z funkcji figure_from_data().

8.3.4. Skrypt compare_figures.py

Skrypt compare_figures.py nie wymaga dalszego opisu poza tym co zostało napisane w rozdziale 7.3.

8.3.5. Skrypt show_lattice.py

Skrypt show_lattice.py jako parametr wywołania przyjmuje listę ścieżek. Każda ścieżka może prowadzić do pliku zawierającego macierz, tj. pliku o rozszerzeniu .sng, .unx, .bnx, bądź do folderu zawierające takie pliki.

Funkcja show_lattice() przyjmuje jako argument ścieżkę oraz zmienną opisującą tryb wyświetlania. W obecnej wersji programu jedynym używanym trybem wyświetlania jest tryb 3D. Obsługę trybu 2D porzucono z powodów, które opisano w rozdziale 8.1.6. Funkcja sprawdza czy podana ścieżka prowadzi do pliku, czy do folderu. W przypadku pliku sprawdza jego rozszerzenie, ładuje macierz za pomocą funkcji pickle.load() i wywołuje odpowiednią funkcję obsługującą macierz z pliku o danym rozszerzeniu: display_ising() dla .sng, display_uniax() dla .unx, display_biax() dla .bnx. W przypadku folderu,

wyszukuje w nim pliki o odpowiednich rozszerzeniach i wywołuje na nich wspomniane wyżej funkcje.

Każda z opisanych wyżej funkcji ładuje odpowiednie dla danej symulacji modele 3D z plików wygenerowanych wcześniej przez skrypt Graphics_App.ModelGenerator.py (patrz rozdział 8.1.1). Następnie, wykorzystując załadowany model oraz dane z macierzy, inicjuje obiekt klasy ParticleLattice i przekazuje go do obiektu klasy Window.

Listing 8.4. Funkcja display_biax().

```
def display_biax(L, title='preview'):
    biax_model = SimpleModel(pickle.load(open(os.path.join(
        source_root, 'models/SpheroPlate.smd'), 'rb')))
    max_x, max_y, max_z = L.shape
    particle = Particle(biax_model, 'SpheroCylinder')
    particle_lattice = ParticleLattice(
        max_x, max_y, max_z, particle=particle)

    for x in range(max_x):
        for y in range(max_y):
            for z in range(max_z):
                particle_lattice[x,y,z].set_orientation(
                                        L[x, y, z])

    window = Window(800, 600, title=title)
    window.set_objects(particle_lattice)
    pyglet.app.run()

    window.close()
    del window
```

8.3.6. Skrypt show_layers.py

Skrypt show_layers.py przyjmuje te same parametry co show_lattice.py.

Dla każdej podanej ścieżki funkcja main() sprawdza czy jest to ścieżka do pliku, czy do folderu. Jeśli ścieżka prowadzi do pliku, zostaje przekazana jako argument do funkcji show_layers(). Jeśli to ścieżka do folderu, to każda ścieżka do pliku macierzy zawartego w tym folderze zostaje przekazana jako argument do show_layers().

Funkcja show_layers() ładuje macierz zawartą w pliku i przekazuje ją jako argument do metody set_lattice() obiektu klasy Window2_5D.

Listing 8.5. Funkcja show_layers().

```
def show_layers(path):
    if path.endswith(".sng"):
        window = Window2_5D(title=path)
        with open(path, 'rb') as f:
            lattice = pickle.load(f)
        window.set_lattice(lattice, type="ising")
        pyglet.app.run()
    elif path.endswith(".unx"):
        window = Window2_5D(title=path)
        with open(path, 'rb') as f:
            lattice = pickle.load(f)
```

```
        window.set_lattice(lattice, type="uniax")
        pyglet.app.run()
    elif path.endswith(".bnx"):
        window = Window2_5D(title=path)
        with open(path, 'rb') as f:
            lattice = pickle.load(f)
        window.set_lattice(lattice, type="biax")
        pyglet.app.run()
```

9. Przeprowadzone eksperymenty

Rozdział opisuje eksperymenty komputerowe przeprowadzone za pomocą symulatora.

9.1. Czas warmup

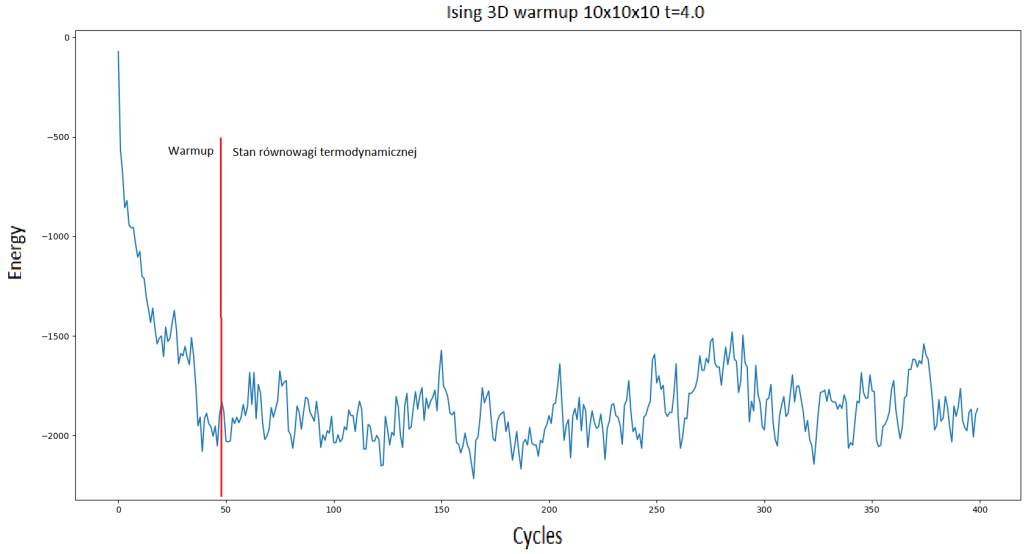
Dużą część eksperymentów stanowiły obserwacje zależności pomiędzy ilością cykli w etapie wamup (potrzebnych do uzyskania oscylacji danych) a rozmiarem macierzy i temperaturą próbki. Każdy cykl składa się z n iteracji, gdzie n to ilość molekuł w macierzy. Ilość tych cykli nazywamy w dalszej części pracy *czasem warmup*, a stan, w którym dane pomiarowe oscylują nazywamy *stanem równowagi termodynamicznej.*

Eksperymenty te były przeprowadzone głównie na modelu Isinga, przede wszystkim ze względu na szybkość obliczeń i fakt, iż symulacje Isinga były pierwszymi w pełni zaimplementowanymi symulacjami w naszym projekcie.

Nie są znane ogólne metody wyznaczania czasu warmup. Czas warmup w naszym projekcie był wyznaczany eksperymentalnie. W celu dobrania tego czasu, analizowane były wartości energii i magnetyzacji próbki podczas etapu warmup (rysunek 9.1).

Zauważono następujące zależności:

1. Ilość cykli potrzebnych do uzyskania stanu oscylacji danych waha się przypadkowo z eksperymentu na eksperyment (rysunek 9.2).
2. Użycie większej macierzy wydłuża średni czas warmup, ale wahania czasu warmup pomiędzy eksperymentami są mniejsze (rysunek 9.3).
3. W zakresie temperatur (1.5, 4.0] im większa temperatura, tym dłuższy czas warmup, przy czym ta zależność jest mniej widoczna dla większych rozmiarów macierzy (np. 20x20x20, 30x30x30).
4. Dla niskich temperatur (ok 1.0 lub mniej), inicjowanie symulacji macierzą uporządkowaną skraca czas warmup, dla wysokich temperatur (ok. 4.0 lub więcej) lepiej inicjować macierzą o nieuporządkowanych (losowych) spinach.
5. Dla niskich temperatur czas warmup może być bardzo długi. Wynika to z faktu, że za pomocą metod losowych (Monte Carlo) mało prawdopodobne jest uzyskanie odpowiednio uporządkowanej macierzy.
6. Dla macierzy 10x10x10 czas warmup jest najczęściej mniejszy od 100 a prawie zawsze mniejszy od 200 cykli.
7. Dla macierzy 20x20x20 czas warmup jest najczęściej pomiędzy 100 a 200 cykli, a prawie zawsze mniejszy od 300.
8. Dla macierzy 30x30x30 czas warmup jest najczęściej pomiędzy 300 a 400.



Rysunek 9.1. Wykres zależności energii próbki od ilości cykli podczas etapu warmup. Próbka osiąga stan równowagi termodynamicznej po ok. 48 cyklach warmup.

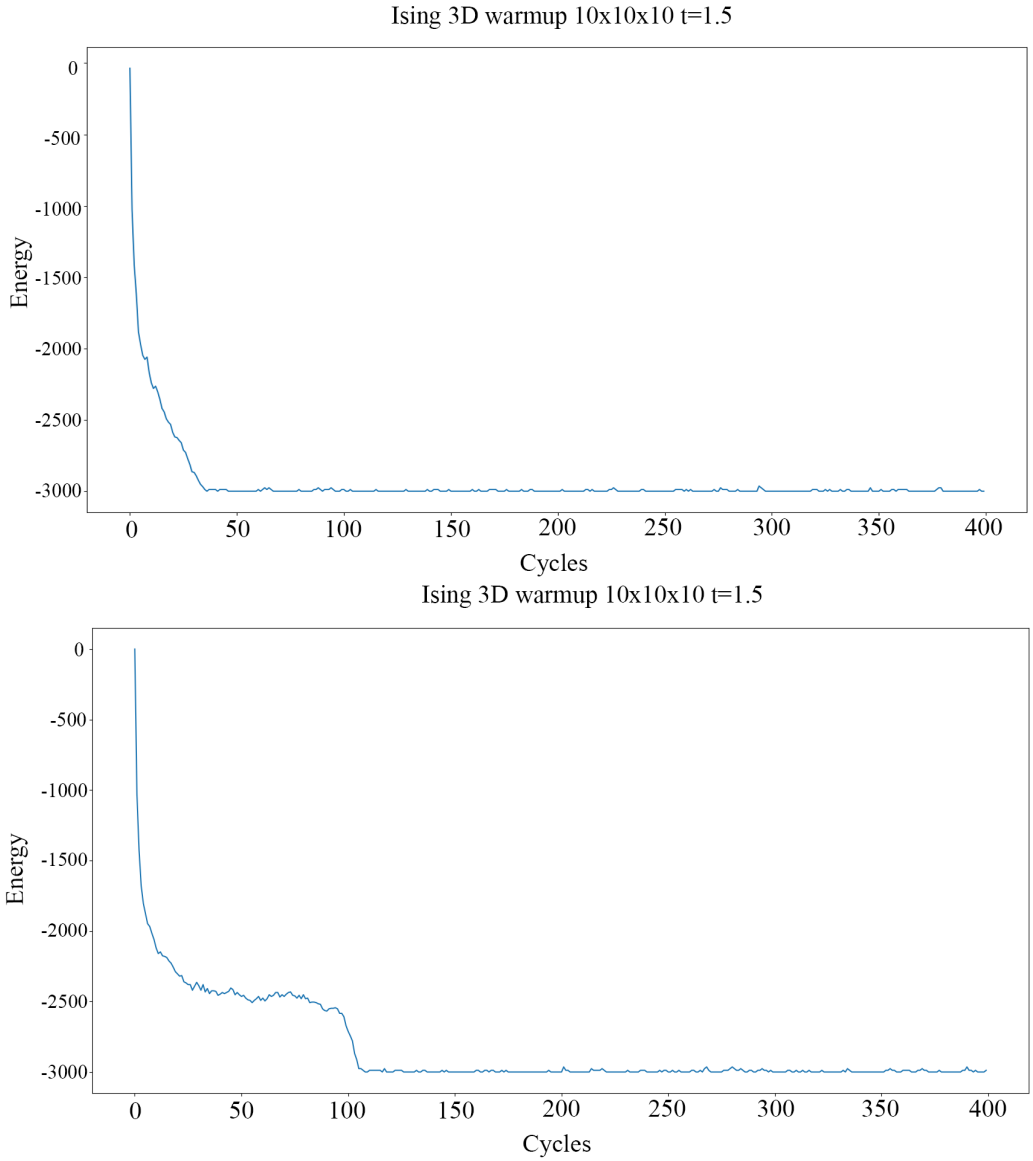
9.2. Zależności temperaturowe w modelu Isinga

Zestaw wykresów przedstawiony na rysunku 9.4 ilustruje zależności temperaturowe w modelu Isinga, wyznaczone na podstawie symulacji dla dwóch różnych rozmiarów macierzy: 10x10x10 (pomarańczowa linia) oraz 40x40x40 (niebieska linia).

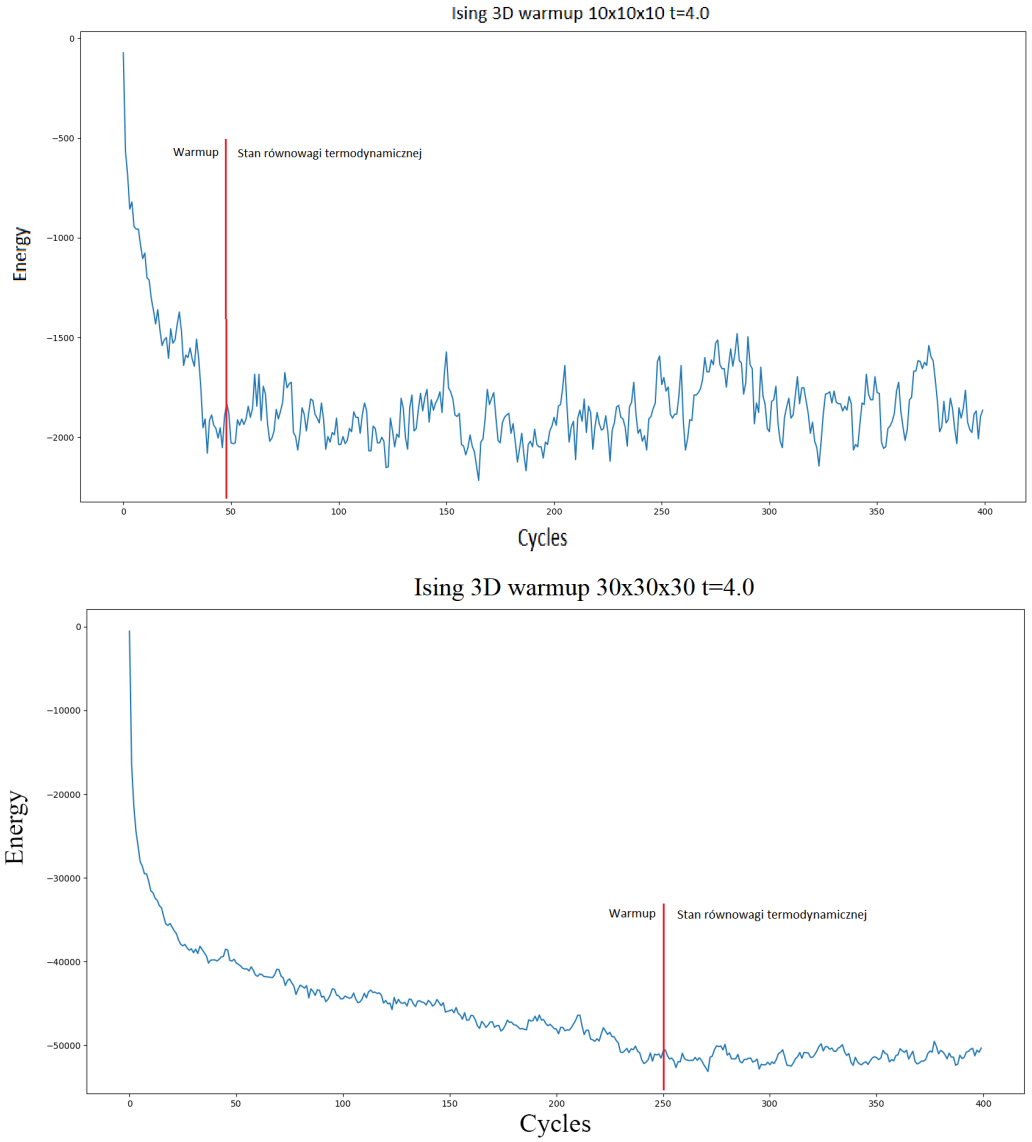
Wspomniane wykresy uzyskano na podstawie symulacji o poniższych parametrach:

— typ symulacji: model Isinga,
— zbiór temperatur [9.0, 8.9, ..., 0.1],
— tryb inicjalizacji: 'random',
— rozmiar macierzy: 10x10x10 lub 40x40x40,
— liczba cykli warmup: 400,
— liczba cykli production: 400.

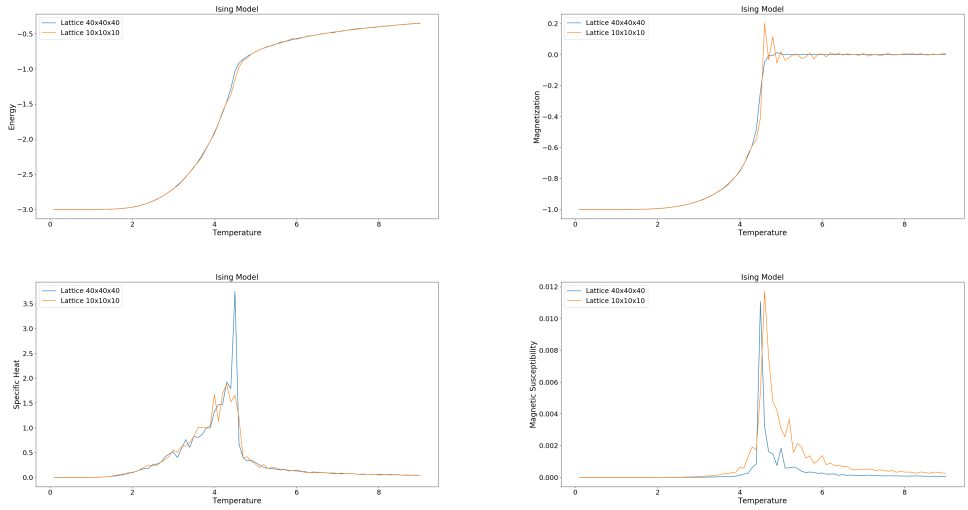
Wspomniane wykresy pozwalają nam prześledzić właściwości termodynamiczne(gęstość energii, ciepło właściwe) oraz ferromagnetyczne (magnetyzacja, podatność magnetyczna) układu w różnych temperaturach oraz zaobserwować temperaturę, w której dochodzi do przejścia fazowego. Na wykresach przejście fazowe można rozpoznać po gwałtownej zmianie w wartościach energii i magnetyzacji, oraz po gwałtownym wzroście wartości ciepła właściwego i podatności magnetycznej. Z naszych wykresów wynika, że do przejścia fazowego w modelu Isinga dochodzi w temperaturze ok. 4.5.



Rysunek 9.2. Przebieg etapów warmup dwóch symulacji m. Isinga dla macierzy 10x10x10 o temperaturze 1.5. Pierwsza symulacja potrzebowała tylko ok. 40 cykli na dojście do stanu równowagi. Druga potrzebowała ponad 100.



Rysunek 9.3. Warmup dla macierzy 30x30x30. Większa macierz potrzebuje więcej cykli na osiągnięcie stanu równowagi termodynamicznej.



Rysunek 9.4. Zależności temperaturowe w modelu Isinga. Wykresy przedstawiają porównanie wyników z symulacji przeprowadzonych dla macierzy o wymiarach 10x10x10 oraz dla macierzy o wymiarach 40x40x40.

9.3. Zależności temperaturowe w modelu Lebwohla-Lashera

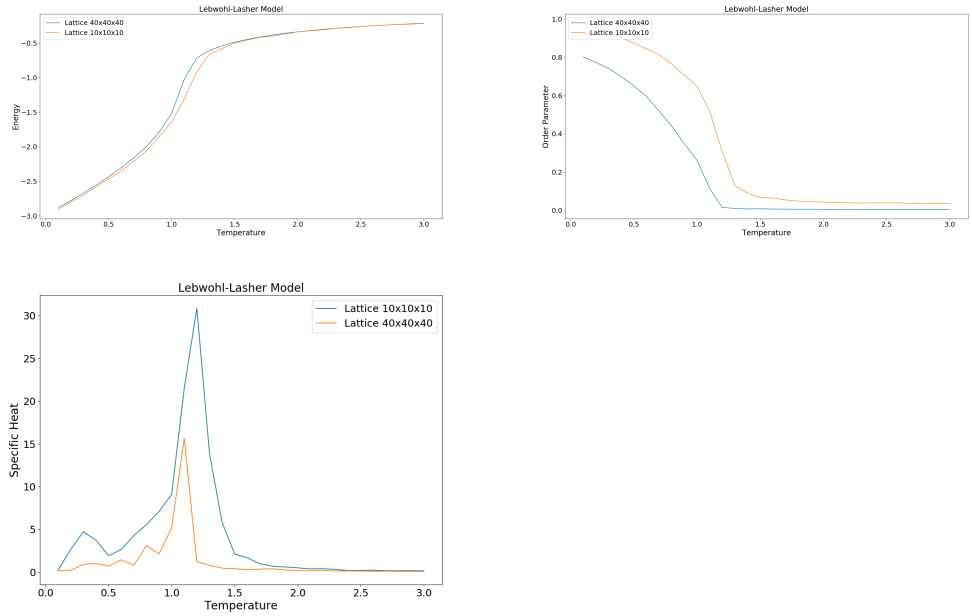
Zestaw wykresów przedstawiony na rysunku 9.5 ilustruje zależności temperaturowe w modelu Lebwohla-Lashera, wyznaczone na podstawie symulacji dla dwóch różnych rozmiarów macierzy: 10x10x10 (pomarańczowa linia) oraz 40x40x40 (niebieska linia).

Wspomniane wykresy uzyskano na podstawie symulacji o poniższych parametrach:

— typ symulacji: model Lebwohla-Lashera,
— zbiór temperatur [3.0, 2.9, ..., 0.1],
— tryb inicjalizacji: 'random',
— rozmiar macierzy: 10x10x10 lub 40x40x40,
— liczba cykli warmup: 200,
— liczba cykli production: 200.

Podobnie jak w przypadku wykresów dla modelu Isinga, wykresy te pozwalają prześledzić właściwości ciała w zależności od temperatury. W tym przypadku są to zależności termodynamiczne (gęstość energii, ciepło właściwe), oraz właściwości ciekło-krystaliczne (stopień uporządkowania molekuł, wyznaczający fazę). Temperaturę przejścia fazowego można rozpoznać na wykresach na podstawie gwałtownej zmiany w gęstości energii oraz ciepła właściwego ciała. Porównując tą wartość z wykresem wartości parametru porządku można wyznaczyć wartość graniczną pomiędzy fazą izomorficzną a nematyczną.

Na podstawie symulacji, możemy ustalić, że przejście fazowe zachodzi w temperaturze ok. 1.2, wartość parametru porządku gwałtownie wzrasta wraz ze spadkiem temperatury od zera do wartości bliskich jeden. Skok pa-



Rysunek 9.5. Zależności temperaturowe w modelu Lebwohla-Lashera. Wykresy przedstawiają porównanie wyników z symulacji przeprowadzonych dla macierzy o wymiarach 10x10x10 oraz dla macierzy o wymiarach 40x40x40.

rametru porządku praktycznie nie jest widoczny ze względu na skończone rozmiary układu. Dokładana analiza przejścia fazowego wymaga większych układów, dłuższych symulacji, a także innych narzędzi znanych z literatury.

9.4. Zależności temperaturowe w modelu molekuł dwuosiowych

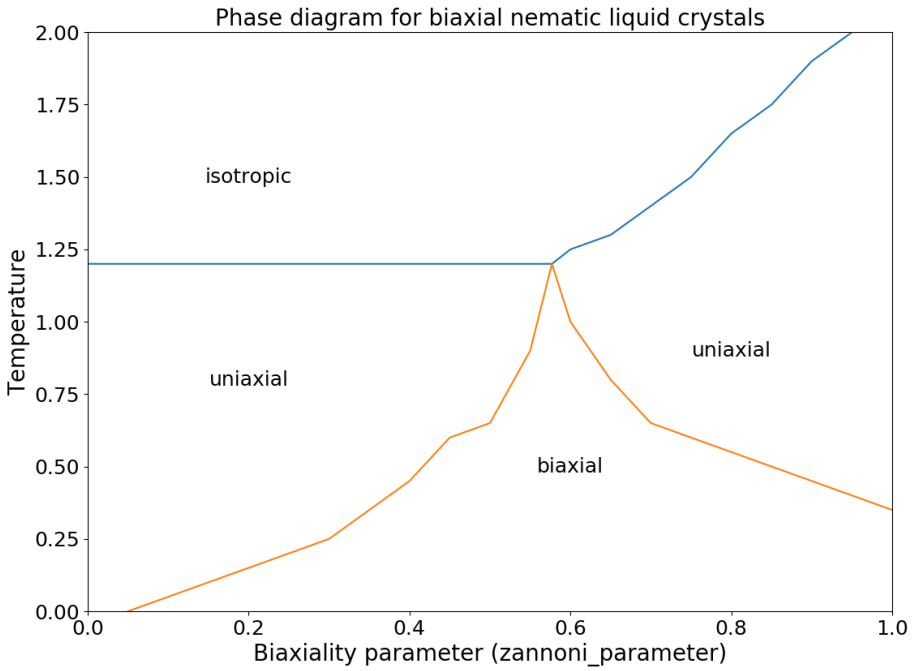
Model molekuł dwuosiowych był najbardziej wymagającym modelem pod względem obliczeniowym rozważanym w naszej pracy. W porównaniu z modelem Lebwohla-Lashera, ten model ma ogólniejsze oddziaływanie między molekułami, a co za tym idzie pojawia się możliwość powstawania większej liczby faz o różnych symetriach i parametrach porządku.

Do obliczeń wybraliśmy oddziaływanie dane wzorem 2.17, ponieważ za pomocą jednego parametru a ('zannoni_parameter') można wygodnie zmieniać właściwości układu. Rysunek 9.6 przedstawia diagram fazowy modelu uzyskany za pomocą naszej aplikacji. Odtwarza on dokładnie wyniki uzyskane w pracy [17]. Parametr a opisujący układ był zmieniany z krokiem 0.05. Symulacie MC dla molekuł dwuosiowych były prowadzone dla układu $10 \times 10 \times 10$. Dla każdej temperatury (skok 0.1) oszacowano minimalny warmup. Zaobserwowano, że zwiększony warmup jest potrzebny w otoczeniu przejść fazowych i dla niskich temperatur. W tych samych obszarach mamy doczynienia z dużymi fluktuacjami w układzie. We wszystkich symulacjach energia układu rośnie w sposób ciągły ze wzrostem temperatury. Przejście $I - N_U$ widać jako skok nachylenia wykresu energii. Przejście $N_U - N_B$ cza-

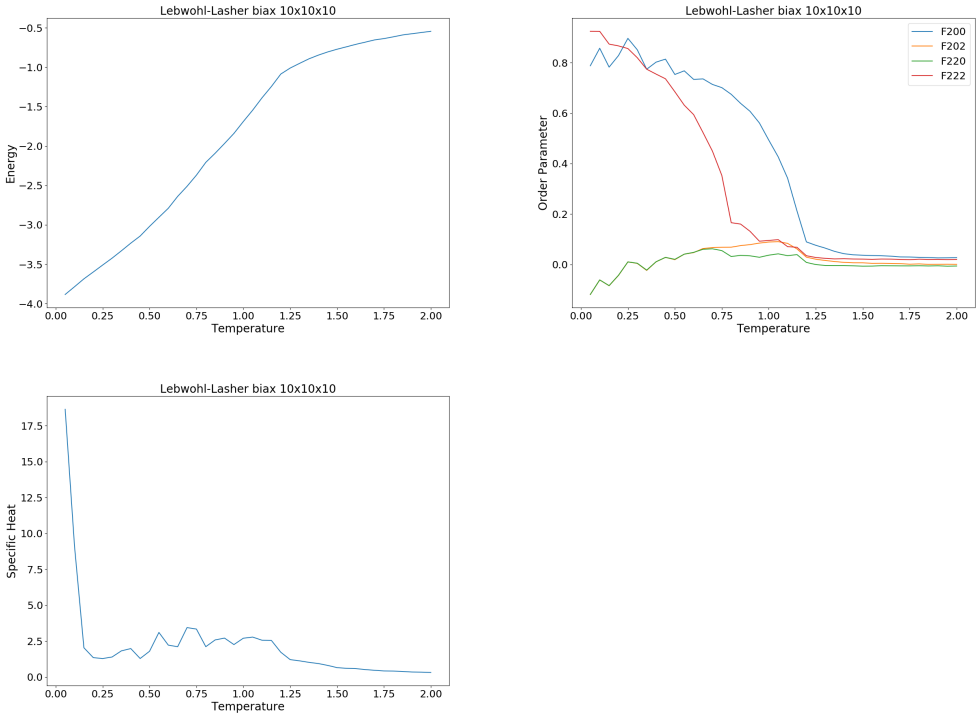
sem zaznacza się jako zafalowanie wykresu energii. Bardziej szczegółowe dane przedstawimy dla dwóch konkretnych układów.

Pierwszy układ z $a = 0.55$ (molekuły prętopodobne) posiada następującą sekwencję faz przy obniżaniu temperatury: $I - N_U - N_B$ (rysunek 9.7).
Parametry symulacji:
— model molekuł dwuosiowych,
— parametr dwuosiowości: 0.55,
— rozmiar macierzy: 10x10x10,
— tryb inicjalizacji: 'organized',
— zakres temperatur: [0.05, 0.1, 0.15, ... 2.0],
— ilość cykli warmupowych: 1000,
— ilość cykli produkcyjnych: 2000.
Przejście fazowe $I - N_U$ dla $T/\epsilon = 1.2$ jest potwierdzone przez wzrost parametru porządku $\langle F_{00}^{(2)} \rangle$ i zauważalny wzrost ciepła właściwego. Z wykresu nie da się wiarygodnie ocenić skoku parametru porządku przy przejściu. Przejście fazowe ciągłe $N_U - N_B$ dla $T/\epsilon = 0.9$ jest potwierdzone przez wzrost parametru porządku $\langle F_{22}^{(2)} \rangle$. Zbyt mały rozmiar układu powoduje duże fluktuacje ciepła właściwego, niefizyczny jest też skok ciepła właściwego poniżej temperatury $T/\epsilon = 0.2$.

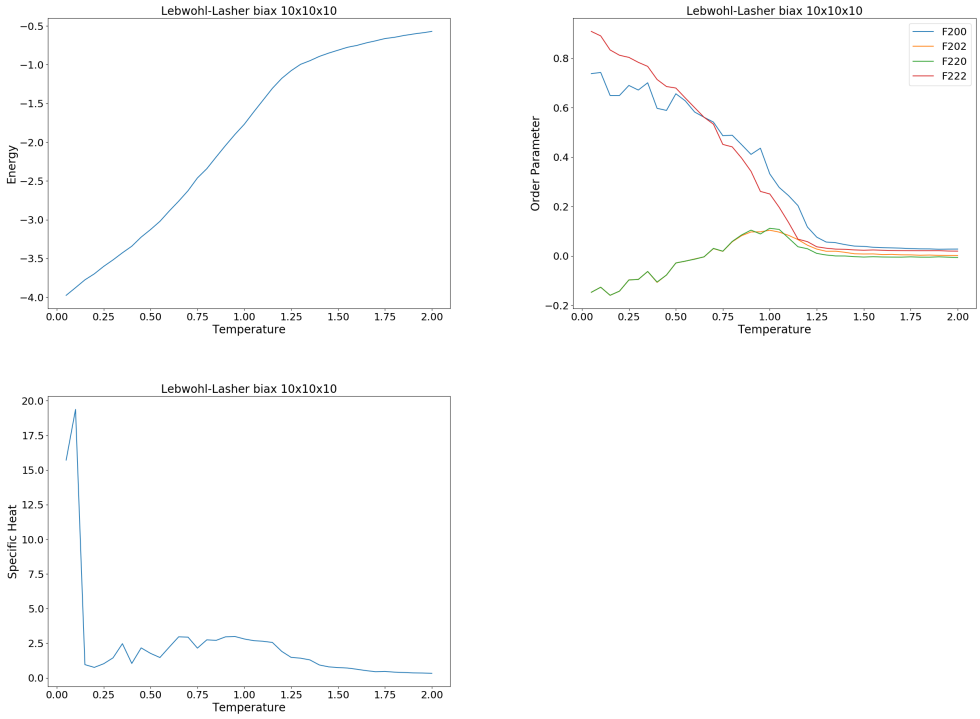
Drugi układ z $a = 1/\sqrt{3}$ (molekuły najbardziej dwuosiowe) posiada bezpośrednie przejście fazowe ciągłe $I - N_B$ (rysunek 9.8).
Parametry symulacji:
— model molekuł dwuosiowych,
— parametr dwuosiowości: $1/\sqrt{3}$,
— rozmiar macierzy: 10x10x10,
— tryb inicjalizacji: 'organized',
— zakres temperatur: [0.05, 0.1, 0.15, ... 2.0],
— ilość cykli warmupowych: 1000,
— ilość cykli produkcyjnych: 2000.
Bezpośrednie przejście fazowe $I - N_B$ dla $T/\epsilon = 1.2$ objawia się jednoczesnym wzrostem parametrów porządku $\langle F_{00}^{(2)} \rangle$ i $\langle F_{22}^{(2)} \rangle$. Nie widać też skoku w nachyleniu wykresu energii. Ciekawy jest wzrost ciepła właściwego zaczynający się w temperaturze dużo powyżej właściwego punktu przejścia fazowego. Sugeruje to istnienie efektów zapowiadających przejście fazowe. Wzrost ciepła właściwego dla temperatur poniżej 0.2 jest niefizyczny.



Rysunek 9.6. Diagram fazowy dla ciekłych kryształów nematycznych dwuosiowych. Obecne są cztery fazy nematyczne: izotropowa, jednoosiowa prętopodobna (z lewej), jednoosiowa dyskopodobna (z prawej), dwuosiowa.



Rysunek 9.7. Zależności temperaturowe w modelu molekuł dwuosiowych dla $a = 0.55$, gdzie a to parametr dwuosiowości.



Rysunek 9.8. Zależności temperaturowe w modelu molekuł dwuosiowych dla $a = 1/\sqrt{3}$, gdzie a to parametr dwuosiowości.

10. Podsumowanie

W wyniku realizacji projektu udało się stworzyć prototyp narzędzia do przeprowadzania symulacji Monte Carlo ciekłych kryształów nematycznych termotropowych. W ramach projektu stworzono implementację kwaternionów (moduł Quaternion), korzystającą z pakietu numpy, oraz aplikację umożliwiającą uruchamianie symulacji dla modeli Isinga, modelu Lebwohla-Lashera oraz modelu molekuł dwuosiowych. Aplikacja pozwala również uzyskać podgląd stanu układu po przeprowadzonej symulacji oraz ułatwia analizę wyników, generując wykresy zależności temperaturowych wybranych wielkości fizycznych.

Kwaterniony posłużyły do opisu orientacji molekuł, modelowanych jako bryły sztywne dwuosiowe. Dla takich brył wymagany jest pełny opis wykorzystujący trzy stopnie swobody, a nie tylko dwa, jak dla punktów na sferze. Podczas symulacji Monte Carlo z każdym węzłem sieci był związany kwaternion, a oddziaływanie między najbliższymi sąsiadami zostało zapisane poprzez kwaterniony. Uniknięto w ten sposób obliczania funkcji trygonometrycznych, pojawiających się przy opisie z kątami Eulera.

Schemat przetwarzania w symulacjach MC zaczerpnięto z programu w języku Python symulującego model Isinga 2D [4], który uogólniono najpierw na model Isinga 3D, potem na model Lebwohla-Laszera (ciekłe kryształy nematyczne jednoosiowe), a na końcu na model ciekłych kryształów nematycznych dwuosiowych. Zauważmy, że w modelu Isinga spiny na sieci mogły przybierać dwie dyskretne wartości, natomiast w modelach ciekłych kryształów mieliśmy do czynienia ze zmiennymi ciągłymi (kwaterniony).

Konfiguracje orientacji molekuł uzyskane w wyniku symulacji MC można obrazować za pomocą aplikacji specjalnie przygotowanej w ramach projektu. Można oglądać wszystkie molekuły w próbce lub pojedyncze warstwy. Warto podkreślić, że molekuły są obrazowane jako bryły sztywne o odpowiedniej symetrii, co automatycznie zapewnia ten sam widok nawet dla różnych orientacji, ale powiązanych ze sobą przez symetrię.

Stworzony kod pozwolił na odtworzenie wyników znanych z literatury (diagram fazowy, zależności temperaturowe), co potwierdziło poprawność merytoryczną programu. Sprawdzono wydajność kodu napisanego w języku Python i potwierdzono użyteczność tego języka w szybkim tworzeniu działających aplikacji w nowych obszarach zastosowań. Z drugiej strony, fizyka często pokazuje naukowcom, że ciekawe obszary badań wymagają obliczeń na granicy możliwości sprzętu i oprogramowania (np. otoczenie przejścia fazowego). Wtedy działający kod pythonowy może pomóc w otrzymaniu wydajniejszej wersji programu napisanej przykładowo w C++, kiedy kwestie architektury programu, przepływu strumienia danych, itp. są już rozpoznane.

A. Model Isinga

W celu nabrania doświadczenia z symulacjami Monte Carlo rozważyliśmy model Isinga [3], który opisuje układ N spinów $\sigma = \{\sigma_i\}$ zlokalizowanych na węzłach sieci d-wymiarowej, $\sigma_i \in \{+1, -1\}$. Energię układu w zewnętrznym polu magnetycznym B można przedstawić w postaci hamiltonianu

$$H(\sigma) = -\sum_{ij} J_{ij}\sigma_i\sigma_j - B\sum_k \sigma_k, \tag{A.1}$$

gdzie całka wymiany J_{ij} jest dodatnia dla oddziaływań ferromagnetycznych, a sumowanie jest w różnych parach molekuł i oraz j. Zakładamy oddziaływanie między najbliższymi sąsiadami, brak pola magnetycznego $B = 0$, oraz jednakowe oddziaływania $J_{ij} = J$. Interesują nas właściwości układu dla dużej liczby spinów. Przykładowo chcemy wiedzieć dla jakiej temperatury następuje przejście fazowe, jak duże są domeny ze spinami ustawionymi w jednym kierunku, ile wynoszą wykładniki krytyczne (ang. *critical exponents*) [27].

Jeżeli $F(\sigma)$ jest pewną funkcją określoną dla układu, to średnią wartość funkcji określamy jako

$$\langle F\rangle = \frac{1}{Z(T)}\sum_\sigma F(\sigma)\exp\left[-\beta H(\sigma)\right], \tag{A.2}$$

gdzie suma przebiega we wszystkich konfiguracjach układu, $1/\beta = k_BT$. Funkcja $Z(T)$ jest sumą statystyczną ($\langle 1\rangle = 1$)

$$Z(T) = \sum_\sigma \exp\left[-\beta H(\sigma)\right]. \tag{A.3}$$

Pochodna w temperaturze wartości średniej $\langle F\rangle$ wynosi

$$\frac{\partial\langle F\rangle}{\partial T} = k_B\beta^2\left[\langle FH\rangle - \langle F\rangle\langle H\rangle\right]. \tag{A.4}$$

Prawdopodobieństwo konfiguracji σ dane jest przez rozkład Boltzmanna

$$P(\sigma) = \exp\left[-\beta H(\sigma)\right]/Z(T). \tag{A.5}$$

A.1. Energia wewnętrzna

Energia wewnętrzna układu wynosi

$$U(T) = \langle H\rangle = -\frac{\partial \ln Z}{\partial\beta}. \tag{A.6}$$

Pojemność cieplna układu ma postać

$$C_V(T) = \frac{\partial U(T)}{\partial T} = k_B\beta^2[\langle H^2\rangle - \langle H\rangle^2]. \tag{A.7}$$

W praktyce wygodniejsze są wielkości intensywne określone na jeden spin, gęstość energii $u(T) = U(T)/N$, ciepło właściwe $c_V(T) = C_V(T)/N$.

A.2. Magnetyzacja

Niech $M(\sigma) = \sum_k \sigma_k$ oznacza magnetyzację układu. Magnetyzacja na spin jest określona jako $m(\sigma) = M(\sigma)/N$, a średnia wartość $M(\sigma)$ wynosi

$$\langle M\rangle = \frac{1}{\beta}\frac{\partial \ln Z}{\partial B}. \tag{A.8}$$

Podatność magnetyczna wynosi

$$\chi = \frac{\partial\langle M\rangle}{\partial B} = \beta[\langle M^2\rangle - \langle M\rangle^2]. \tag{A.9}$$

A.3. Rozwiązania modelu

Model jednowymiarowy $d = 1$ został ściśle rozwiązany przez Isinga w roku 1925. Dla $B = 0$ dostajemy $m = 0$, czyli nie ma ferromagnetyzmu, nie ma przejścia fazowego.

Model dwuwymiarowy $d = 2$ (bez pola magnetycznego) rozwiązał analitycznie Onsager w roku 1944. Temperatura krytyczna wynosi

$$k_BT_c/J = \frac{2}{\ln(1+\sqrt{2})} \approx 2.269185314213022. \tag{A.10}$$

Istnienie przejścia fazowego w tym modelu udowodnił Peierls w roku 1936. W granicy termodynamicznej dla $T < T_c$ magnetyzacja wynosi

$$m = [1 - \sinh^{-4}(2\beta J)]^{1/8}. \tag{A.11}$$

Model trójwymiarowy $d = 3$ i opis punktu krytycznego wykazuje podobieństwo z konforemną teorią pola (CFT). Temperatura krytyczna wynosi około $k_BT_c/J = 4.5116(1)$ [28], [29].

A.4. Symulacje komputerowe modelu Isinga

Symulacje komputerowe z natury rzeczy dotyczą układów skończonych. W takich układach nie może pojawić się przejście fazowe, które obserwuje się w granicy termodynamicznej (dla układów nieskończonych). Z drugiej strony, zmierzanie do granicy termodynamicznej jest tak szybkie, że nawet dla sieci skończonych rozmiarów obserwuje się oznaki przejścia fazowego, choć rozmyte.

W symulacjach zwykle przyjmuje się periodyczne warunki brzegowe (ang. *periodic boundary conditions*), jeżeli badane są własności w objętości fazy (ang. *bulk properies*). Jeżeli interesują nas zjawiska na granicy fazy, to pozostawiamy wyróżnioną warstwę graniczną.

Każdy spin w układzie może przyjmować dwie wartości, więc możliwych jest 2^N stanów układu. Dla dużej liczby spinów N nie jest możliwe w praktyce wykonanie sumy we wszystkich stanach, dlatego korzysta się z symulacji MC. Bardzo często korzysta się z algorytmu Metropolisa-Hastingsa, który jest metodą statystyczną typu MCMC (ang. *Markov chain Monte Carlo*) [30]. Celem algorytmu Metropolisa-Hastingsa jest wygenerowanie kolekcji stanów zgodnie z pewnym rozkładem prawdopodobieństwa. Aby to osiągnąć, algorytm używa procesu Markowa, który asymptotycznie osiąga jednoznaczny rozkład stacjonarny.

Na początku symulacji jest faza rozgrzewki (ang. *burn-in*, *warm-up*), gdzie jeszcze jest widoczna zależność od punktu startowego. Łańcuch Markowa zwykle jest ergodyczny i ta zależność zanika. Zwykle fazę rozgrzewki usuwa się z próbki. Potem łańcuch Markowa osiąga fazę stacjonarną, która nas interesuje najbardziej.

W symulacjach modelu Isinga dopuszcza się w jednym kroku zmianę jednego spinu na sieci (ang. *single-spin-flip dynamics*), ten spin jest losowany ze wszystkich spinów na sieci. Z danego stanu możemy przejść do dowolnego innego przez kolejną zmianę pojedyńczych spinów [3]. Po zmianie pojedyńczego spinu łatwo można obliczyć zmianę energii układu ΔH, ponieważ zależy ona jedynie od stanu najbliższych sąsiadów zmienionego spinu. Jeżeli $\Delta H < 0$, to nowy stan jest zawsze akceptowany. Jeżeli $\Delta H > 0$, to nowy stan jest akceptowany z prawdopodobieństwem $\exp(-\beta \Delta H)$. Ta procedura jest zgodna z warunkiem równowagi szczegółowej (ang. *detailed balance*).

Wyniki symulacji w niskich temperaturach zależą od wyboru konfiguracji startowej. Jeżeli startujemy od stanu idealnie uporządkowanego, to taki stan się utrzymuje i wyniki są zgodne z obliczeniami analitycznymi. Jeżeli startujemy od konfiguracji przypadkowej, to w układzie pojawiają się metastabilne domeny z jednakowo ułożonymi spinami.

B. Sesja interaktywna z kwaternionami

W celu przybliżenia możliwości modułu Quaternion pokażemy przykładową sesję interaktywną.

```
>>> from Quaternion import Quaternion
>>> i = Quaternion(0, 1)
>>> j = Quaternion(0, 0, 1)
>>> k = Quaternion(0, 0, 0, 1)
>>> i*j*k
Quaternion(-1.0, 0.0, 0.0, 0.0)
>>> i*i == -1, j*j == -1, k*k == -1
(True, True, True)
>>> i*j == -j*i, i*k == -k*i, j*k == -k*j
(True, True, True)
>>> i.is_unit(), j.is_unit(), k.is_unit()
(True, True, True)
>>> (i*j*k).is_unit()
True
>>> q = Quaternion(1, 3, 5, 7)
>>> q.is_unit()
False
>>> abs(q)
9.1651513899116797
>>>q.norm()
9.1651513899116797
>>> q.norm_squared()
84.0
>>> q.conjugate()
Quaternion(1.0, -3.0, -5.0, -7.0)
>>> ~q                          # kwaternion odwrotny
Quaternion(0.011904761904761904, -0.03571428571428571,
-0.05952380952380952, -0.08333333333333333)
>>> q ** 123
Quaternion(-1.6996939559171654e+118, -4.6193362528945275e+117,
-7.69889375482421e+117, -1.0778451256753893e+118)
>>> pow(q, -5)
Quaternion(8.03802966674918e-06, -4.349947515989442e-06,
-7.249912526649072e-06, -1.0149877537308697e-05)
>>> str(q)
'1.0 + 3.0i + 5.0j + 7.0k'
>>> repr(q)
'Quaternion(1.0, 3.0, 5.0, 7.0)'
>>> q[0]                        # część skalarna
1.0
>>> q[1:]                        # część wektorowa
array([ 3.,  5.,  7.])
>>> from numpy import pi
>>> z_rot = Quaternion.create_from_z_rotation(pi/2)
>>> z_rot
```

```
Quaternion(0.7071067811865476, 0.0, 0.0, 0.7071067811865475)
>>> z_rot.apply_to_vector([1., 0., 0.])
array([  2.22044605e-16,   1.00000000e+00,   0.00000000e+00])
>>> z_rot = Quaternion.create_from_z_rotation(pi)
>>> z_rot
Quaternion(6.123233995736766e-17, 0.0, 0.0, 1.0)
>>> z_rot.apply_to_vector([1., 0., 0.])
array([ -1.00000000e+00,   1.22464680e-16,   0.00000000e+00])
>>> z_rot.get_rotation_matrix()
array([[ -1.00000000e+00,  -1.22464680e-16,   0.00000000e+00,
0.00000000e+00],
[  1.22464680e-16,  -1.00000000e+00,   0.00000000e+00,
0.00000000e+00],
[  0.00000000e+00,   0.00000000e+00,   1.00000000e+00,
0.00000000e+00],
[  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
1.00000000e+00]])
```

Bibliografia

[1] J. Żmija, J. Zieliński, J. Parka, E. Nowinowski-Kruszelnicki, *Displeje ciekłokrystaliczne. Fizyka, technologia, zastosowanie*, Wydawnictwo Naukowe PWN, Warszawa 1993.

[2] Wikipedia, Quaternion, 2018, `https://en.wikipedia.org/wiki/Quaternion`.

[3] Wikipedia, Ising model, 2018, `https://en.wikipedia.org/wiki/Ising_model`.

[4] Kristian Haule, *Monte Carlo simulation of the 2D Ising model*, Rurgers University, 2017, `http://www.physics.rutgers.edu/~haule/681/src_MC/python_codes/ising.py`.

[5] P. A. Lebwohl, G. Lasher, *Nematic-Liquid-Crystal Order - A Monte Carlo Calculation*, Phys. Rev. A 6, 426 (1972).

[6] M. P. Allen and D. J. Tildesley, *Computer Simulations of Liquids*, Clarendon Press, 1989.

[7] Wikipedia, Lennard-Jones potential, 2018, `https://en.wikipedia.org/wiki/Lennard-Jones_potential`.

[8] U. Fabbri, C. Zannoni, *A MC investigation of the Lebwohl-Lasher lattice model in the vicinity of its orientational phase transition*, Mol. Phys. 58, 763 (1986).

[9] S. Boschi, M. Brunelli, C. Chiccoli, P. Pasini, C. Zannoni, *Liquid Crystal Lattice Models on Quadrics Supercomputers*, Int. J. Mod. Phys. C 08, 547 (1997).

[10] Z. Zhang, O. G. Mouritsen, and M. J. Zuckerman, *Weak First-Order Orientational Transition in the Lebwohl-Lasher Model for Liquid Crystals*, Phys. Rev. Lett. 69, 2803 (1992).

[11] N. V. Priezjev and Robert A. Pelcovits, *Cluster Monte Carlo simulations of the nematic-isotropic transition*. Phys. Rev. E 63, 062702 (2001).

[12] C. W. Greeff and Michael A. Lee, *Pretransitional fluctuations of the Lebwohl-Lasher model of a nematic liquid crystal*, Phys. Rev. E 49, 3225 (1994).

[13] A. Kapanowski, *Statistical theory of elastic constants of biaxial nematic liquid crystals*, Phys. Rev. E 55, 7090-7104 (1997).

[14] N. Ghoshal, S. Shabnam, S. DasGupta, and S. K. Roy, *Monte Carlo investigation of critical properties of the Landau point of a biaxial liquid-crystal system*, Phys. Rev. E 93, 052701 (2016).

[15] L. Longa, P. Grzybowski, S. Romano, and E. Virga, *Minimal coupling model of the biaxial nematic phase*, Phys. Rev. E 71, 051714 (2005).

[16] J. P. Straley, *Ordered phases of a liquid of biaxial particles*, Phys. Rev. A 10, 1881 (1974).

[17] F. Biscarini, C. Chiccoli, P. Pasini, F. Semeria, and C. Zannoni, *Phase Diagram and Orientational Order in a Biaxial Lattice Model: A Monte Carlo Study*, Phys. Rev. Lett. 75, 1803 (1995).

[18] C. Chiccoli, P. Pasini, F. Semeria, and C. Zannoni, *A detailed Monte Carlo*

investigation of the tricritical region of a biaxial liquid crystal system, Int. J. Mod. Phys. C 10, 469 (1999).
[19] R. Berardi, L. Muccioli, S. Orlandi, M. Ricci, and C. Zannoni, *Computer simulations of biaxial nematics*, J. Phys.: Condens. Matter 20, 463101 (2008).
[20] E. F. Gramsbergen, L. Longa, W. H. de Jeu, *Landau theory of the nematic-isotropic phase transition*, Phys. Rep. 135, 195 (1986).
[21] M. P. Allen, *Computer simulation of a biaxial liquid crystal*, Liq. Cryst. 8, 499 (1990).
[22] P. J. Camp and M. P. Allen, *Phase diagram of the hard biaxial ellipsoid fluid*, J. Chem. Phys. 106, 6681 (1997).
[23] Wikipedia, Quaternion group, 2018, `https://en.wikipedia.org/wiki/Quaternion_group`.
[24] Wikipedia, Quaternions and spatial rotation, 2018, `https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation`.
[25] David M. Bourg, *Fizyka dla programistów gier*, O'Reilly, Helion, Gliwice 2003.
[26] Andrzej Kądzielawa, *Programowanie symulacji fizyki w czasie rzeczywistym*, Materiały dydaktyczne, Uniwersytet Jagielloński, Kraków 2016.
[27] Wikipedia, Ising critical exponents, 2018, `https://en.wikipedia.org/wiki/Ising_critical_exponents`.
[28] Michael N. Barber, R. B. Pearson, Doug Toussaint, *Finite-size scaling in the three-dimensional Ising model*, Phys. Rev. B 32, 1720-1730 (1985).
[29] A. F. Sonsin, M. R. Cortes, D. R. Nunes, J. V. Gomes, R. S. Costa, *Computational Analysis of 3D Ising Model Using Metropolis Algorithms*, Journal of Physics: Conference Series 630, 012057 (2015).
[30] Wikipedia, Metropolis-Hastings algorithm, 2018, `https://en.wikipedia.org/wiki/Metropolis-Hastings_algorithm`.
[31] Khronos Group, OpenGL 4.5 Reference Pages, 2018, `https://www.khronos.org/registry/OpenGL-Refpages/gl4/`.
[32] Khronos Group, OpenGL Wiki, 2018, `https://www.khronos.org/opengl/wiki/Main_Page`.
[33] WebGL Lesson 11 – spheres, rotation matrices, and mouse events, 2009, `http://learningwebgl.com/blog/?p=1253URL`.
[34] Wikipedia, OpenGL, 2018, `https://en.wikipedia.org/wiki/OpenGL`.
[35] Wikipedia, OpenGL (wersja polska artykułu), 2018, `https://pl.wikipedia.org/wiki/OpenGL`.
[36] OpenGL Wiki, FAQ, 2018, `https://www.khronos.org/opengl/wiki/FAQ#What_is_OpenGL.3F`.
[37] OpenGL Wiki, OpenGL Concepts, 2017, `https://www.khronos.org/opengl/wiki/Portal:OpenGL_Concepts`.
[38] OpenGL Wiki, Rendering Pipeline Overview, 2017, `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`.
[39] Wikipedia, Rendering_(computer_graphics), 2018, `https://en.wikipedia.org/wiki/Rendering_(computer_graphics)`.
[40] OpenGL Wiki, Fragment, 2017, `https://www.khronos.org/opengl/wiki/Fragment`.
[41] Wikipedia, 3D Modeling, 2018, `https://en.wikipedia.org/wiki/3D_modeling`.
[42] Wikipedia, Polygon Mesh, 2018, `https://en.wikipedia.org/wiki/Polygon_mesh`.
[43] Dibya Chakravorty, 8 Most Common 3D File Formats of 2018, 2018,

https://all3dp.com/3d-file-format-3d-files-3d-printer-3d-cad-vrml-stl-obj/#Top.
[44] learnopengl.com, Coordinate Systems, 2014, https://learnopengl.com/Getting-started/Coordinate-Systems.
[45] NPTEL, kurs: Computer Aided Design and Manufacturing I, 2009, https://nptel.ac.in/courses/Webcourse-contents/IIT-Delhi/Computer%20Aided%20Design%20&%20ManufacturingI/mod2/01.htm.
[46] OpenGL Wiki, Buffer Object, 2018, https://www.khronos.org/opengl/wiki/Buffer_Object.
[47] OpenGL Wiki, Vertex_Specification, 2018, https://www.khronos.org/opengl/wiki/Vertex_Specification#Vertex_Buffer_Object
[48] OpenGL Wiki, Face Culling, 2016, https://www.khronos.org/opengl/wiki/Face_Culling.
[49] OpenGL Wiki, Vertex Post-Processing, Clipping, 2017, https://www.khronos.org/opengl/wiki/Vertex_Post-Processing#Clipping.
[50] Wikipedia, Układ współrzędnych sferycznych, 2018 https://pl.wikipedia.org/wiki/Uk%C5%82ad_wsp%C3%B3%C5%82rz%C4%99dnych_sferycznych.
[51] Wikipedia, Układ współrzędnych biegunowych, 2018 https://pl.wikipedia.org/wiki/Uk%C5%82ad_wsp%C3%B3%C5%82rz%C4%99dnych_biegunowych.
[52] Python Programming Language - Official Website, https://www.python.org/.
[53] NumPy Reference - Numpy v1.15 Manual, 2018, https://docs.scipy.org/doc/numpy/reference/.
[54] Pyrr Maths Library - Pyrr 0.9.2 documentation, 2018, https://pyrr.readthedocs.io/en/latest/.
[55] PyOpenGL - The Python OpenGL Binding, http://pyopengl.sourceforge.net/.
[56] pyglet Documentation - pyglet 1.3.2, https://pyglet.readthedocs.io/en/pyglet-1.3-maintenance/.
[57] 12.1. pickle — Python object serialization - Python 3.6.6 documentation, https://docs.python.org/3.6/library/pickle.html.