Uniwersytet Mikołaja Kopernika

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Adam Sosnowski

Implementacja i porównanie metod numerycznych stosowanych do realistycznych symulacji dynamiki zbioru punktów materialnych. Wizualizacja z użyciem grafiki 3D

Praca inżynierska

wykonana w Zakładzie Mechaniki Kwantowej

Opiekun: **dr Jacek Matulewski**

Toruń 2008

Dziękuję panu dr Jackowi Matulewskiemu za poświęcanie mi czasu oraz udzielenie wielu cennych wskazówek i pomoc, bez której ta praca nie powstałaby.

UMK zastrzega sobie prawo własności niniejszej pracy inżynierskiej w celu udostępnienia dla potrzeb działalności naukowo-badawczej lub dydaktycznej

Spis treści

Strona

1. Wstęp………………………………………………………………… 5

2. Algorytmy………………………………………………………………… 6

2.1. Równanie ruchu – przedstawienie problemu……………………………. 6

2.2. Metoda Eulera……………………………………………………………… 6

2.3. Metoda punktu środkowego - MidPoint………………………………… 7

2.4. Metoda Runge-Kutty czwartego rzędu………………………………….. 9

2.5. Reguła Stoermera………………………………………………………... 11

2.6. Algorytm Verleta…………………………………………………………... 13

3. Implementacja……………………………………………………………. 15

3.1. Wektor……………………………………………………………………… 15

3.1.1. Iloczyn skalarny………………………………………………………….. 16

3.1.2. Iloczyn wektorowy……………………………………………………….. 16

3.1.3. Długość wektora…………………………………………………………. 17

3.1.4. Normowanie……………………………………………………………… 17

3.1.5. Operatory arytmetyczne………………………………………………….. 17

3.2. Punkt materialny………………………………………………………… 19

3.2.1. Metody dostępu do informacji o cząstce……………………………. 19

3.2.2. Implementacja metod numerycznych klasie `PktMat`……………… 20

3.3. Zbiór punktów materialnych…………………………………………….. 22

3.3.1. Implementacja metod numerycznych numerycznych klasie `ZbPktMat`…………………………………………………………. 22

3.3.2. Klasy rozszerzające szablon ZbPktMat – implementacje przykładowych modeli………………………………………………. 27

3.3.2.1. Swobodne punkty………………………………………………. 27

3.3.2.2. Łańcuch oscylatorów…………………………………………... 28

3.3.2.3. Siatka…………………………………………………………… 30

3.3.2.4. Lina…………………………………………………………….. 33

3.3.2.5. Cząstki oddziałujące ze sobą potencjałem Morse'a……………. 36

4. Testy………………………………………………………………………. 40

4.1. Testy zbieżności………………………………………………………. 41

4.2. Testy czasu trwania obliczeń…………………………………………. 46

4.3. Podsumowanie……………………………………………………... 50

5. Podsumowanie……………………………………………………………. 51

6. Literatura…………………………………………………………………. 52

1. Wstęp

Zadaniem pierwszych komputerów było symulowanie trajektorii pocisków. Obecnie symulacje komputerowe dzięki dynamicznemu rozwojowi technologii komputerowych stały się ważnym narzędziem wielu dziedzin nauki takich jak fizyka, biologia czy chemia. Coraz szerzej wykorzystywane są też w szeroko rozumianej rozrywce. Atrakcyjność filmów i gier komputerowych zwiększa się np. przez realizm dodawany w formie odpowiedniego ruchu obiektów w tworzonych scenach, ich wzajemną interakcją. Sprowadza się to do modelowania odpowiednich zjawisk fizycznych.

Skuteczne zastosowanie symulacji poprzedzone musi być stworzeniem odpowiedniego modelu badanego środowiska. W swojej pracy stworzyłem kilka prostych modeli układów fizycznych w celu symulacji ich zachowania dla różnych warunków początkowych. Badanie dynamiki owych modeli możliwe jest oczywiście dzięki zastosowaniu metod numerycznych. Symulacja dynamiki przeprowadzona może być wieloma algorytmami. Wybór optymalnego jest jednym ze stawianych tej pracy zadań. Moja praca w głównej mierze składa się z opisu i porównania tych metod pod względem dokładności i złożoności czasowej. Podsumowaniem całej pracy jest aplikacja prezentująca praktyczne zastosowanie powyższych zagadnień. Jej atutem jest prezentacja wyników za pomocą grafiki 3D generowanej za pomocą biblioteki OpenGL [5, 6, 10, 12, 13, 14], tworząc trójwymiarowe animacje układów symulowanych w czasie rzeczywistym.

2. Algorytmy

2.1. Równanie ruchu – przedstawienie problemu

Równanie ruchu [1] dla pojedynczego punktu materialnego w jednym wymiarze o działającej na niego określonej sile wypadkowej przybiera prostą postać równania różniczkowego drugiego rzędu:

$$\frac{d^2x}{dt^2} = \frac{F_w}{m}, \tag{1}$$

gdzie:

F_w – siła wypadkowa,
x – położenie punktu na osi X,
m – masa punktu.

Najprostszym sposobem rozwiązania równania Newtona (1) opisującego przyspieszenie punktu materialnego jest przekształcenie go do układu dwóch równań sprzężonych pierwszego rzędu.

$$\frac{dx}{dt} = v \tag{2}$$

$$\frac{dv}{dt} = \frac{F_w}{m} \tag{3}$$

W przeprowadzanej symulacji rozwiązywanym zagadnieniem jest położenie punktu w zależności od czasu. Znane są położenie i prędkość w chwili początkowej, a także działające na cząstkę siły. Znając równania (2) i (3) można użyć którejś z opisanych niżej metod numerycznych do rozwiązania zadanego problemu i znalezienia prędkości oraz położenia punktu materialnego w każdej chwili czasu.

2.2. Metoda Eulera

W swej klasycznej postaci jest to jedna z najprostszych metod numerycznych [7, 15]. Służy ona przede wszystkim do rozwiązywania równań różniczkowych pierwszego stopnia. Zakładając znajomość pochodnej $x'(t)$ funkcji zmiennej rzeczywistej $x(t)$ i jej wartość początkową w punkcie t_0, można obliczyć wartość tej funkcji w punkcie $t_0+\Delta t$ i następnych. Algorytm odpowiada rozwiązaniu za pomocą rozwinięcia w szereg Taylora (4, 5):

$$x(t_0 + \Delta t) = x(t_0) + \frac{x'(t_0)}{1!} \cdot \Delta t^1 + \frac{x''(t_0)}{2!} \cdot \Delta t^2 + ... \quad (4)$$

$$x(t_0 + \Delta t) = \sum_{n=0}^{\infty} \frac{x^{(n)}(t_0)}{n!}\Big|_{t_0} \cdot \Delta t^n , \quad (5)$$

w którym zachowano tylko pierwsze pochodne:

$$x(t_0 + \Delta t) = x(t_0) + \frac{x'(t_0)}{1!} \cdot \Delta t^1 \quad (6)$$

Co jest równoważne:

$$x(t_0 + \Delta t) = x(t_0) + x'(t_0) \cdot \Delta t \quad (7)$$

Błąd metody dąży do zera dla Δt dążącego do zera. Obcięcie rozwiązania do wyrazu Δt w potędze pierwszej oznacza, że metoda ta ma dokładność jedynie pierwszego rzędu względem rozwinięcia w szereg Taylora. Błąd obcięcia wynosi zatem $O(\Delta t^2)$. To powoduje względnie szybką rozbieżność algorytmu.

Zastosowanie metody Eulera do rozwiązania równania Newtona (1) jest następujące:

1. W pierwszej kolejności obliczana jest prędkość na podstawie zadanego przyspieszenia:

$$v_1 = v_0 + dv \quad (8)$$

$$dv = \frac{F_w}{m} \cdot dt \quad (9)$$

2. Następnie korzystając z obliczonej prędkości wyznaczane jest położenie:

$$x_1 = x_0 + dx \quad (10)$$

$$dx = v_1 \cdot dt \quad (11)$$

2.3 Metoda punktu środkowego - MidPoint

Metoda ta jest szczególną postacią metody Rungego-Kutty [7, 9, 15]. Należy ona do klasy schematów dwustopniowych. Oznacza to, że wartość funkcji $f(x, t)$ musi być obliczana dwa razy dla każdego wyznaczenia wartości funkcji x w punkcie $t_0+\Delta t$. Idea metody MidPoint może być zapisana w trzech wyrażeniach:

$$k_1 = \Delta t \cdot f(x_0, t_0) \quad (12)$$

Powyższe równanie jest algorytmicznym zapisem metody Eulera. Na podstawie wartości początkowych i długości interwału czasowego obliczana jest wartość funkcji x w punkcie $t_0+\Delta t$ zapisana jako k_1.

$$k_2 = \Delta t \cdot f(x_0 + \frac{1}{2} \cdot k_1, t_0 + \frac{1}{2} \cdot \Delta t) \tag{13}$$

W (13) ponownie wykonywane jest całkowanie w celu znalezienia wartości f w punkcie odpowiadającym połowie k_1 wyznaczonej w kroku poprzednim, a następnie obliczana jest wielkość k_2. Końcowa wartość funkcji x w punkcie $t_0+\Delta t$ wyznaczona jest za pomocą rozwiązania z równania (13) i wartości początkowej.

$$x(t_0 + \Delta t) = x(t_0) + k_2 \tag{14}$$

Dokładność metody MidPoint jest drugiego rzędu. Uzyskuje się to dzięki podzieleniu przedziału całkowania na dwie części. Nazwa metody pochodzi stąd, że rozwiązanie wyliczane w (13) wykorzystuje punkt środkowy rozwiązania (12).

Metodę MidPoint można wykorzystać do rozwiązania równania Newtona w sposób opisany w poniższych punktach:

1. Na podstawie znajomości położenia, prędkości i przyspieszenia w chwili t_0 obliczane są współczynniki k_1:

$$k_{1v} = \Delta t \cdot \frac{F_w}{m} \tag{15}$$

$$k_{1x} = \Delta t \cdot v_0 \tag{16}$$

2. Wartość siły wypadkowej aktualizowana jest dla połowy rozwiązania otrzymanego w punkcie pierwszym, a następnie wyliczane są współczynniki k_2:

$$k_{2v} = \Delta t \cdot \frac{F_w}{m} \tag{17}$$

$$k_{2x} = \Delta t \cdot \frac{1}{2} \cdot k_{1v} \tag{18}$$

3. Ostatecznie wartości prędkości i położenia w punkcie $t_0+\Delta t$ dane są wzorami:

$$v_1 = v_0 + k_{2v} \tag{19}$$

$$x_1 = x_0 + k_{2x} \tag{20}$$

2.4. Metoda Runge-Kutty czwartego rzędu

Metoda Runge-Kutty czwartego rzędu (RK4) [7, 9, 15] jest rozszerzeniem metody punktu środkowego o kolejne dwa kroki. Aby obliczyć wartość funkcji x w punkcie $t_0+\Delta t$ należy obliczyć wartość funkcji $f(x, t)$ czterokrotnie. Kompletny algorytm RK4 jest podobny do metody MidPoint, która w istocie jest metodą Runge-Kutty drugiego rzędu, przy czym zwiększona zostaje ilość dodatkowych obliczeń. Schemat jest następujący:

$$k_1 = \Delta t \cdot f(x_0, t_0) \tag{21}$$

$$k_2 = \Delta t \cdot f(x_0 + \frac{1}{2} \cdot k_1, t_0 + \frac{1}{2} \cdot \Delta t) \tag{22}$$

Pierwsze dwa kroki są identyczne jak w metodzie punktu środkowego.

$$k_3 = \Delta t \cdot f(x_0 + \frac{1}{2} \cdot k_2, t_0 + \frac{1}{2} \cdot \Delta t) \tag{23}$$

W kolejnym kroku analogicznie do poprzedniego wyznaczana jest wartość funkcji f, z tą różnicą, że przedział całkowania określa połowa wartości obliczonej w kroku poprzednim k_2.

$$k_4 = \Delta t \cdot f(x_0 + k_3, t_0 + \Delta t) \tag{24}$$

Ostatni krok metody to znalezienie ostatniego współczynnika k_4 całkowaniem metodą prostokątów. Wartość funkcji x w punkcie $t_0+\Delta t$ obliczana jest na podstawie znajomości wartości początkowej $x(t_0)$ oraz wyznaczonych w czterech krokach współczynników k_1, k_2, k_3 i k_4:

$$x(t_0 + \Delta t) = x(t_0) + \frac{1}{6} \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) \tag{25}$$

Metoda RK4 przewyższa metodę MidPoint dwukrotnie pod względem dokładności. Oznacza to, że dla dwukrotnie większego interwału czasowego może uzyskać wyniki równie wiarygodne.

Równanie Newtona rozwiązuje się metodą RK4 zgodnie z następującym algorytmem:

1. Na podstawie znajomości położenia, prędkości i przyspieszenia w chwili t_0 obliczane są współczynniki k_1:

$$k_{1v} = \Delta t \cdot \frac{F_w}{m} \tag{26}$$

$$k_{1x} = \Delta t \cdot v_0 \tag{27}$$

2. Wartość siły wypadkowej aktualizowana jest dla połowy rozwiązania otrzymanego w punkcie pierwszym, a następnie wyliczane są współczynniki k_2:

$$k_{2v} = \Delta t \cdot \frac{F_w}{m} \tag{28}$$

$$k_{2x} = \Delta t \cdot \frac{1}{2} \cdot k_{1v} \tag{29}$$

3. Siła wypadkowa jest ponownie aktualizowana w punkcie wyznaczonym przez połowę rozwiązania z punktu drugiego:

$$k_{3v} = \Delta t \cdot \frac{F_w}{m} \tag{30}$$

$$k_{3x} = \Delta t \cdot \frac{1}{2} \cdot k_{2v} \tag{31}$$

4. Ostatnie współczynniki potrzebne do wyznaczenia prędkości i położenia w chwili następnej obliczane są po ostatniej aktualizacji siły wypadkowej w punkcie wyznaczonym przez rozwiązanie z punktu poprzedniego:

$$k_{4v} = \Delta t \cdot \frac{F_w}{m} \tag{32}$$

$$k_{4x} = \Delta t \cdot k_{3v} \tag{33}$$

5. Prędkość i położenie w chwili następnej obliczane są na podstawie prędkości i położenia początkowego i sumy współczynników zsumowanych z odpowiednimi wagami:

$$x_1 = x_0 + \frac{1}{6} \cdot (k_{1x} + 2 \cdot k_{2x} + 2 \cdot k_{3x} + k_{4x}) \tag{34}$$

$$v_1 = v_0 + \frac{1}{6} \cdot (k_{1v} + 2 \cdot k_{2v} + 2 \cdot k_{3v} + k_{4v}) \tag{35}$$

2.5. Reguła Stoermera

Reguła Stoermera [9] jest datowana na rok 1907. Jest popularnym sposobem dyskretyzacji równań różniczkowych drugiego rzędu przedstawionych jako układ dwóch równań pierwszego rzędu. Układ ten składa się z równań, w których pochodna nie występuje po ich prawej stronie.

Równanie:

$$x'' = f(t) \tag{36}$$

Wartości początkowe:

$$x(t_0) = x_0 \tag{37}$$

$$x'(t_0) = v_0 \tag{38}$$

W celu zwiększenia precyzji obliczeń interwał czasowy Δt dzielony jest na kilka lub kilkanaście kroków. Długość kroku definiowana jest następująco:

$$h = \frac{\Delta t}{m}, \tag{39}$$

gdzie:

h – długość kroku,

m – ilość kroków,

Δt – długość interwału czasowego dla metody.

Algorytm ten jest metodą wielopunktową, co oznacza, że nie startuje z jednego punktu. Wymagana ilość wartości początkowych uzyskiwana jest po wykonaniu pierwszej, startowej fazy algorytmu. Odpowiada ona rozwinięciu w szereg Taylora (5) wokół t do drugiego rzędu wielkości:

$$x(t_0 + \Delta t) = x(t_0) + x'(t_0) \cdot \Delta t^1 + \frac{1}{2} \cdot x''(t_0) \cdot \Delta t^2, \tag{40}$$

co po podstawieniu wartości h za wyraz Δt, zastosowaniu równań (36, 37, 38) oraz kilku prostych przekształceniach daje:

$$x_1 = x_0 + h \cdot [v_0 + \frac{1}{2} \cdot h \cdot f(t_0)] \tag{41}$$

Druga faza metody polega wykonaniu w pętli obliczeń wyznaczających wartość funkcji w chwili następnej $x(t+ \Delta t)$. Ilość iteracji pętli jest określona ilością kroków minus dwa(kroki od drugiego do przedostatniego). Każde wyliczenie jest poprzedzane aktualizacją wartości drugiej pochodnej danej funkcji $f(t_0+kh)$.

$$x_{k+1} - 2 \cdot x_k + x_{k-1} = h^2 \cdot f(t_0 + k \cdot h) \qquad k=1, \dots, m-1 \tag{42}$$

Po przekształceniu ze względu na x_{k+1}:

$$x_{k+1} = 2 \cdot x_k - x_{k-1} + h^2 \cdot f(t_0 + k \cdot h) \qquad k=1, \dots, m-1 \tag{43}$$

W trzeciej, ostatniej fazie, na podstawie wcześniej wyznaczonych wartości (43) obliczana jest wartość pierwszej pochodnej rozważnej funkcji. Wzór ten uzyskuje się z rozwinięcia w szereg Taylora (5) dla funkcji $x(t)$ do drugiego rzędu wartości (40) i przekształceniu otrzymanego równania ze względu na drugi wyraz tego rozwinięcia (44).

$$x'(t_0) = \frac{x(t) - x(t_0)}{\Delta t} + \frac{1}{2} \cdot x''(t_0) \cdot \Delta t \tag{44}$$

Co po zastosowaniu równań (36, 37, 38) oraz zależności (39) daje:

$$x'(t_0 + \Delta t) = \frac{(x_m - x_{m-1})}{h} + \frac{1}{2} \cdot h \cdot f(t_0 + \Delta t) \tag{45}$$

Zasada Stoermera w rozwiązaniu zagadnienia punktu materialnego przedstawia się następująco:

1. Krok czasowy metody dzielony jest na kilka części:

$$h = \frac{dt}{z}, \tag{46}$$

gdzie:

z – ilość kroków,

h – długość kroku.

2. Pierwszy krok oblicza kolejne położenie potrzebne do wystartowania generalnej części metody:

$$x_1 = x_0 + h \cdot (v_0 + \frac{1}{2} \cdot h \cdot \frac{F_w}{m}) \tag{47}$$

3. Główna część bazująca na obliczeniach z poprzedniego punktu:

$$x_{k+1} = 2 \cdot x_k - x_{k-1} + h^2 \cdot \frac{F_w}{m}, \qquad k=1, \ldots, z-1 \tag{48}$$

Dla każdego k aktualizowana jest wartość siły wypadkowej F_w.

4. Ostatnia czynność polega na wyliczeniu prędkości bazującym na znajomości dwóch ostatnich położeń i siły wypadkowej:

$$v = \frac{x_k - x_{k-1}}{h} + \frac{1}{2} \cdot h \cdot \frac{F_w}{m} \tag{49}$$

2.6. Algorytm Verleta

Algorytm Verleta [2, 8, 11, 15] można określić jako uproszczoną wersję metody Stoermera. Tak jak ona algorytm ten jest metodą wielopunktową, jednak w przeciwieństwie do pierwowzoru nie przewiduje samodzielnego obliczania wymaganej ilości wartości początkowych. W tym celu należy posłużyć się jedną z metod jednopunktowych, np. metodą Eulera.

Algorytm Verleta jest metodą rutynowo wykorzystywaną w symulacjach biofizycznych (dynamice molekularnej). Znana jest także pod nazwą jawnej metody różnic centralnych. Służy do numerycznego rozwiązywania układów równań różniczkowych zwyczajnych drugiego rzędu. Została opracowana w 1967 r. przez francuskiego fizyka Loupe'a Verleta.

Podstawowa idea algorytmu polega na dwóch rozwinięciach w szereg Taylora (5) wyrażenia $x(t)$ do trzeciego rzędu wielkości. Pierwsze równanie jest rozwinięciem „do przodu" w czasie – $t_0+\Delta t$:

$$x(t_0 + \Delta t) = x(t_0) + x'(t_0) \cdot \Delta t + \frac{1}{2} x''(t_0) \cdot \Delta t^2 + \frac{1}{6} \cdot x'''(t_0) + O(\Delta t^4), \tag{50}$$

a drugie „do tyłu" – $t_0-\Delta t$:

$$x(t_0 - \Delta t) = x(t_0) - x'(t_0) \cdot \Delta t + \frac{1}{2} x''(t_0) \cdot \Delta t^2 - \frac{1}{6} \cdot x'''(t_0) + O(\Delta t^4) \tag{51}$$

Dodając stronami powyższe wyrażenia uzyskuje się:

$$x(t_0 + \Delta t) + x(t_0 - \Delta t) = 2 \cdot x'(t_0) + x''(t_0) \cdot \Delta t^2 + O''(\Delta t^4) \tag{52}$$

Rozwiązując ze względu na $x(t+\Delta t)$, w celu znalezienia wartości funkcji w chwili następnej:

$$x(t+\Delta t) = 2 \cdot x(t) - x(t-\Delta t) + x''(t_0) \cdot \Delta t^2 + O''(\Delta t^4) \quad (53)$$

Błąd obcięcia w przypadku algorytmu Verleta rozwijanego wartościami Δt jest rzędu Δt^4, nawet, jeśli trzecie pochodne nie są jawne. Jego zaletą jest to, że jest łatwy do zaimplementowania, dokładny i stabilny. Problem z algorytmem Verleta jest taki, że pierwsze pochodne nie są generowane wprost. Jest to pożyteczne, kiedy nie są one potrzebne, bo nie zabierają czasu procesora potrzebnego do ich obliczania. Jeśli jednak ich znajomość jest konieczna, wyprowadza się je odejmując stronami rozwinięcia (50) i (51):

$$x(t+\Delta t) - x(t-\Delta t) = 2 \cdot x'(t_0) \cdot \Delta t \quad (54)$$

Po prostym przekształceniu wyrażenia (54) można uzyskać pochodną funkcji $x'(t)$:

$$x'(t_0) = \frac{1}{2 \cdot \Delta t} \cdot (x(t+\Delta t) - x(t-\Delta t)) \quad (55)$$

Wadą metody jest to, że znajomość wartości funkcji wyprzedza o jeden krok znajomość pochodnej tej funkcji. Wynika to ze sposobu jej obliczania opartym na znajomości wartości funkcji w kroku poprzednim i następnym. Błąd związany z wyznaczaniem pierwszej pochodnej jest rzędu Δt^2.

Można łatwo zauważyć, że wartość funkcji x w punkcie $t_0+\Delta t$ zarówno w Verlecie jak i Stoermerze jest wyrażona w taki sam sposób. Różnica polega na tym, że zasada Stoermera dzieli interwał czasowy na kilka mniejszych kroków i wykonuje obliczenia dla każdego z nich, za każdym razem uwzględniając zmiany drugiej pochodnej. Algorytm Verleta wykonuje tę czynność jednokrotnie, dla wartości drugiej pochodnej zadanej jako wartość początkowa.

Schemat zastosowania metody Verleta do rozwiązania równania Newtona składa się z dwóch części: inicjalizacyjnej, wykonanej np. metodą Eulera oraz właściwej, w której obliczenia wykonywane są już przy użyciu algorytmu Verleta.

1. Położenie wyznaczane na podstawie dwóch poprzednich położeń:

$$x_2 = 2 \cdot x_1 - x_0 + \frac{F_w}{m} \cdot dt^2 \quad (56)$$

2. Wyrażenie na prędkość w danej chwili opiera się na znajomości położenia w chwili poprzedniej i następnej:

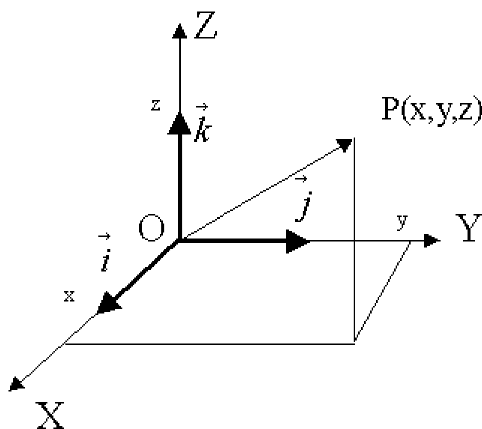
$$v_1 = \frac{x_2 - x_0}{2 \cdot dt} \quad (57)$$

3. Implementacja

Klasy zaprojektowane na potrzeby programu są szablonami C++ [4, 5]. Oznacza to, że definicja danej klasy jest przygotowana tylko raz, ale może być użyta dla różnych typów danych (w naszym przypadku dla liczb zmiennoprzecinkowych pojedynczej i podwójnej precyzji).

3.1 Wektor

Zjawiska fizyczne symulowane w programie w znacznej mierze opierają się na rachunku wektorowym. Z tego powodu przygotowanie zwartej struktury danych opisującej wektor jest bardzo wygodne. Wektor w przestrzeni trójwymiarowej, w układzie kartezjańskim reprezentowany jest przez trzy współrzędne: x, y i z.



Rys. 1. Wektor w kartezjańskim układzie współrzędnych

Konstruktor obiektu przyjmuje jako argumenty wywołania trzy liczby typu określonego przez parametr szablonu, na podstawie którego kompilator tworzy klasę opisującą wektor. Są to wartości inicjujące współrzędne tworzonego w ten sposób wektora:

```
Wektor<float> x(1.0f, 2.0f, 3.0f);
```

Alternatywny konstruktor przyjmuje także trójelementową tablicę tych samych wartości. Kolejne elementy oznaczają odpowiednio składowe: x, y, z. Powyższy zapis jest zatem równoważny z:

```
float tablica[] = {1.0f, 2.0f, 3.0f};
Wektor<float> x(tablica);
```

Konstruktor domyślny tworzy obiekt o składowych zerowych.

Składowe wektora można modyfikować wykorzystując funkcję składową klasy `Wektor`: `Set`. Jej argumenty są takie jak w przypadku konstruktora obiektu: trzy liczby typu określonego przez parametr szablonu:

```
x.Set(5.0f, 6.0f, 7.0f);
```

lub trójelementowa tablica takich liczb:

```
float tablica[] = {5.0f, 6.0f, 7.0f};
x.Set(tablica);
```

Odczytu składowych wektora można dokonać na dwa sposoby. Pierwszy: poprzez funkcję składową `Get` jako argument wywołania przyjmującą tablicę trójelementową, do której zostaną zwrócone współrzędne wektora.

```
float tablica[3];
x.Get(tablica);
```

Drugi sposób to użycie `operatora[]` zwracającego poszczególne składowe w zależności od wywołania funkcji. Argument 1 oznacza składową x, analogicznie 2 – y i 3 – z.

```
skladowa_x = x[1];
skladowa_y = x[2];
skladowa_z = x[3];
```

3.1.1 Iloczyn skalarny

W trójwymiarowej przestrzeni euklidesowej iloczyn skalarny dwóch wektorów: **a**=(a_x, a_y, a_z) i **b**=(b_x, b_y, b_z) oznaczany jest przez: **a**·**b** i definiowany jako suma iloczynów składowych obu wektorów.

$$\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z \qquad (58)$$

Wynikiem iloczynu skalarnego jest liczba. W programie iloczyn skalarny dwóch wektorów można obliczyć korzystając z funkcji `ScalarProduct`.

```
Wektor<float> a(1.0f, 2.0f, 3.0f), b(3.0f, 2.0f, 1.0f);
float iloczyn = ScalarProduct(a, b);
```

3.1.2. Iloczyn wektorowy

Iloczyn wektorowy dwóch wektorów zapisywany jest jako **a**×**b**. Daje on w wyniku wektor. Wzór na wyrażenie iloczynu wektorowego poprzez składowe obu wektorów można zapisać w postaci quasi wyznacznika macierzy:

$$\vec{a}\times\vec{b}=\begin{vmatrix}\hat{x} & \hat{y} & \hat{z}\\ a_x & a_y & a_z\\ b_x & b_y & b_z\end{vmatrix}=(a_y\cdot b_z-a_z\cdot b_y)\cdot\hat{x}+(a_z\cdot b_x-a_x\cdot b_z)\cdot\hat{y}+(a_x\cdot b_y-a_y\cdot b_x)\cdot\hat{z} \quad (59)$$

Iloczyn wektorowy można obliczyć za pomocą funkcji `VectorProduct` zaimplementowanej, tak jak iloczyn skalarny, dla obiektów klasy `Wektor`.

```
Wektor<float> a(1.0f, 2.0f, 3.0f), b(3.0f, 2.0f, 1.0f);
Wektor<float> iloczyn = VectorProduct(a, b);
```

3.1.3. Długość wektora

Długość wektora w przestrzeni zdefiniowana jest jako pierwiastek kwadratowy z iloczynu skalarnego wektora z samym sobą. Normę z wektora zapisuje się następująco:

$$|\vec{a}|=\sqrt{(\vec{a}\cdot\vec{a})}=\sqrt{(a_x^2+a_y^2+a_z^2)} \quad (60)$$

W programie do obliczania długości wektora służy funkcja składowa `Length` zdefiniowana w klasie `Wektor`.

```
Wektor<float> x(1.0f, 2.0f, 3.0f);
float dlugosc = x.Lenght();
```

3.1.4. Normowanie

W programie często występuje konieczność użycia wersora (wektor o jednostkowej długości) wskazującego kierunek w przestrzeni. W takich przypadkach stosuje się operację normowania. Zachowuje ona zwrot i kierunek wektora, a jego długość sprowadza do jednostki. Wektor jednostkowy definiowany jest następująco:

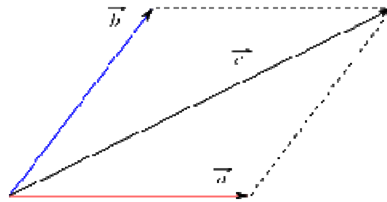
$$\hat{a}=\frac{\vec{a}}{|\vec{a}|} \quad (61)$$

Wektor normalny równy jest ilorazowi danego wektora i jego długości. Użycie normowania w programie możliwe jest dzięki funkcji składowej `Unit`:

```
Wektor<float> x(1.0f, 2.0f, 3.0f);
Wektor<float> jednostkowy = x.Unit();
```

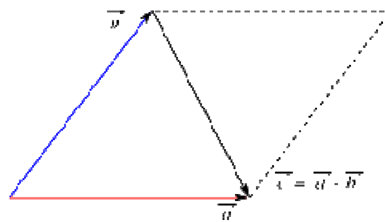
3.1.5. Operatory arytmetyczne

Poza powyższymi operacjami klasa `Wektor` udostępnia także operatory arytmetyczne takie jak suma i różnica dwóch wektorów. Wektor powstały w wyniku operacji dodawania równa się sumie poszczególnych składowych.



Rys. 2. Suma dwóch wektorów w dwóch wymiarach. Składowa z równa jest zeru

Analogicznie jest w przypadku różnicy.



Rys. 3. Różnica dwóch wektorów. Składowa z jest równa zeru

Dodatkowo zaimplementowane są operatory mnożenia i dzielenia wektora przez skalar. Odpowiadają one odpowiednio jego wydłużeniu i skróceniu odpowiednią ilość razy. Każda składowa jest mnożona lub dzielona przez liczbę. Użycie powyższych operatorów przedstawia się następująco:

```
Wektor<float> a(1.0f, 2.0f, 3.0f), b(3.0f, 2.0f, 1.0f);
Wektor<float> suma = a + b;
Wektor<float> roznica = a -b;
Wektor<float> powiekszony = 3 * a;
Wektor<float> pomniejszony = a / 3;
```

Ponadto w klasie `Wektor` istnieje w specyficzny sposób przeciążony operator mnożenia działający na dwa wektory. Został napisany na potrzeby implementacji metod numerycznych opisanych w poprzednim rozdziale. Wynikiem jest wektor, którego składowe odpowiadają iloczynowi poszczególnych składowych mnożonych wektorów. Dla macierzy operator taki nazywa się iloczynem Hadamarda znany także pod nazwą iloczynu Schura lub po prostu iloczynu po współrzędnych

$$(A \bullet B)_{ij} = a_{ij} \cdot b_{ij} \,. \tag{62}$$

Jego użycie pokazane jest poniżej.

```
Wektor<float> nowy_wektor = a * b;
```

3.2. Punkt materialny

Podstawowy element, z którego budowane będą opisane niżej modele trójwymiarowych układów fizycznych to punkt materialny. Do opisu cząstki potrzebne są informacje takie jak: masa, prędkość, czy położenie. Ruch takiej cząstki może być ograniczony przez pewne więzy. Kolejną informacją przydatną do opisu punktu materialnego jest określenie czy cząstka oddziałuje z innymi, sąsiednimi. Szablon klasy `PktMat` (punkt materialny) zawiera wszystkie te dane oraz metody służące do ich ustawiania i pobierania informacji o reprezentowanej cząstce.

Utworzenie obiektu danej klasy wymaga podania jego położenia, prędkości masy i określenia czy na punkt nałożone są więzy uniemożliwiające mu swobodne poruszanie się. Klasa `PktMat` do reprezentacji pierwszych dwóch z powyżej przedstawionych danych używa obiektów wcześniej opisanej klasy `Wektor`. Masa jest liczbą typu określonego przez parametr szablonu, a więzy są zmienną logiczną (`false` oznacza punkt o ruchu nieograniczonym, `true` – z nałożonymi więzami):

```
Wektor<float> polozenie(1.0f, 2.0f, 3.0f);
Wektor<float> predkosc(3.0f, 2.0f, 1.0f);
float masa = 5.0f;
bool wiezy = true;
PktMat<float> czastka(polozenie, predkosc, masa, wiezy);
```

Jest to równoważne zapisowi:

```
PktMat<float> czastka(1.0f, 2.0f, 3.0f, 3.0f, 2.0f, 1.0f,
                      5.0f, true);
```

A także:

```
float tablica1[] = {1.0f, 2.0f, 3.0f};
float tablica2[] = {3.0f, 2.0f, 1.0f};
PktMat<float> czastka(tablica1, tablica2, 5.0f, true);
```

Domyślny konstruktor tworzy cząstkę znajdującą się w początku układu współrzędnych, spoczywającą, o jednostkowej masie i bez żadnych więzów.

3.2.1. Metody dostępu do informacji o cząstce

Zmiana parametrów opisujących punkt materialny możliwa jest przez użycie funkcji składowej `Set`. Jej pierwszy argument jest stałą identyfikującą parametr punktu materialnego, który chcemy zmodyfikować. Globalne stałe identyfikujące zmieniane parametry to: `CURR_POS`, który oznacza aktualne położenie, `PREV_POS` – położenie w

chwili poprzedniej, VEL – prędkość cząstki, MASS – jej masę, CONS – informacja o więzach. Drugim argumentem jest nowa wartość przypisywana modyfikowanemu parametrowi.

```
PktMat<float> czastka;
czastka.Set(CURR_POS, 1.0f, 2.0f, 3.0f);
czastka.Set(PREV_POS, 0.0f, 1.0f, 2.0f);
czastka.Set(MASS, 2.0f);
czastka.Set(CONS, false);
```

W przypadku jednoczesnego ustalania położenia i prędkości, tak jak w konstruktorze obiektu, zamiast trzech liczb można użyć trójelementowej tablicy lub instancji klasy Wektor.

Informację o stanie poszczególnych parametrów można uzyskać korzystając z funkcji składowej Get. Podobnie jak w przypadku funkcji Set jej pierwszym parametrem jest stała identyfikująca parametr, którego wartość ma być pobrana. Położenie i prędkość mogą zostać zwrócone jako wektory lub tablice współrzędnych.

```
Wektor<float> polozenie = czastka.Get(CURR_POS);

float tablica[3];
czastka.Get(VEL, tablica);
```

Do pozyskania masy lub informacji o więzach należy użyć innych funkcji. Są nimi GetMass zwracająca masę molekuły i IsHanged informująca czy jest ona nieruchoma względem układu odniesienia.

```
float masa = czastka.GetMass();
bool wiezy = czastka.IsHanged();
```

3.2.2. Implementacja metod numerycznych w klasie punktu materialnego

W klasie PktMat zostały zaimplementowane dwie proste metody numeryczne. Są to metoda Eulera i Verleta. Ich opis znajduje się w rozdziale drugim. Algorytmy te zostały wykorzystane w funkcjach: StepForwardEuler i StepForwardVerlet. Obliczają one położenie cząstki, po upływie czasu podawanego jako argument funkcji. Dodatkowo jako argument wywołania funkcja przyjmuje wektor przyspieszenia działającego na punkt materialny.

```
Wektor<float> przyspieszenie(1.0f, 1.0f, 1.0f);
czastka.StepForwardEuler(przyspieszenie, 0.1f);
czastka.StepForwardVerlet(przyspieszenie, 0.1f);
```

Poniżej znajdują się listingi przedstawiające poszczególne metody numeryczne. Listing 1 prezentuje metodę StepForwardEuler odpowiedzialną za realizację kroku czasowego w metodzie Eulera.

Listing 1. Metoda realizująca krok czasowy w metodzie Eulera

```
template<class type>
void PktMat<type>::StepForwardEuler(const Wektor<type> &a, type dt)
{
    prev_pos = curr_pos;
    Wektor<type> tmp = velocity + a * dt;
    velocity = tmp;
    tmp = curr_pos + velocity * dt;
    curr_pos = tmp;
}
```

W pierwszym etapie działania funkcji w zmiennej przechowującej poprzednie położenie cząstki zapisywana jest bieżąca pozycja (w chwili wywołania metody). Następnie korzystając ze wzoru (8) na podstawie prędkości początkowej, przyspieszenia i kroku czasowego obliczana jest wartość prędkości. Następnie analogicznie, na podstawie położenia początkowego, obliczonej w poprzedniej linii kodu prędkości i interwału czasowego uaktualniana jest wartość położenia zgodnie ze wzorem (7).

Metoda widoczna w listingu 2 to implementacja algorytmu Verleta. Jest to algorytm trójpunktowym, dlatego w pierwszym kroku symulacji zmuszeni jesteśmy do korzystania ze zwykłego algorytmy Eulera.

Listing 2. Metoda realizująca krok czasowy w metodzie Verleta

```
template<class type>
void PktMat<type>::StepForwardVerlet(const Wektor<type> &a, type dt)
{
    type dwa = 2.0;
    if(curr_pos == prev_pos)
        StepForwardEuler(a, dt);
    else
    {
        Wektor<type> tmp = (curr_pos*dwa) - prev_pos + (a*dt*dt);
        prev_pos = curr_pos;
        curr_pos = tmp;
        velocity = curr_pos - prev_pos * (1/(2*dt));
    }
}
```

Informacja o poprzednim położeniu (`prev_pos`) punktu, którą w metodzie `StepForwardEuler` zapisywaliśmy w polach klasy `PktMat` przygotowana została właśnie na potrzeby powyższej implementacji metody Verleta. Jeśli wynik porównania położenia poprzedniego i aktualnego jest prawdziwy, wywoływana jest funkcja odpowiadająca metodzie Eulera. W ten sposób obliczane są potrzebne wartości startowe metody Verleta. Kolejne instrukcje implementują wzory (53) i (55).

Powyższe metody są algorytmami jednokrokowymi. Znajomość przyspieszenia działającego na cząstkę w chwili wywołania funkcji jest wystarczająca do wyznaczenia położenia punktu w chwili następnej: $x(t_0+\Delta t)$. Pozostałe metody opisane w rozdziale drugim: Rungego-Kutty drugiego i czwartego rzędu oraz Stoermera, są metodami wielokrokowymi.

Do obliczenia kolejnego położenia wymagają kilkukrotnego wyznaczania przyspieszenia działającego na daną cząstkę. Z tego powodu implementacje tych metod znajdują się pozostałych klasie `ZbPktMat`. Dokładniejsze wyjaśnienie znajduje się w części 3.3.1.

3.3. Zbiór punktów materialnych

`ZbPktMat` (zbiór punktów materialnych) jest kolejnym szablonem klasy abstrakcyjnej. Oznacza to, że nie można utworzyć obiektu będącego instancją konkretyzacji tej klasy. Jej zadaniem jest stworzenie fundamentu dla klas pochodnych implementujących konkretne symulowane modele. Definiuje ona zbiór punktów materialnych reprezentowanych przez obiekty klasy `PktMat`. Przechowuje także informacje o przyspieszeniach i siłach działających na każdą z cząstek z tego zbioru. Dodatkową informacją przechowywaną w klasie jest nazwa aktualnie używanego algorytmu numerycznego, za pomocą którego wyznaczane jest położenie molekuły wszystkich punktów następnej kolejnej chwili czasu.

Metody dostępu do informacji opisujących każdą z cząstek zbioru są takie same jak w przypadku obiektów klasy `PktMat`. Jedyną różnicą w wywołaniu tych funkcji jest ich dodatkowy argument będący liczbą całkowitą, określający numer cząstki w zbiorze. Dla przykładowej klasy dziedziczącej `przyklad` składającej się z pięciu punktów materialnych użycie tych funkcji może być następujące:

```
przyklad<float> x(RK4);
x.Set(VEL, 2, 1.0f, 1.0f, 1.0f);
wektor<float> pozycja4czastki = x.Get(CURR_POS, 3);
bool wiezy4czastki = IsHanged(3);//cząstki są numerowane od zera
```

3.3.1. Implementacja metod numerycznych w klasie `ZbPktMat`

W klasie zaimplementowane zostały wszystkie metody numeryczne opisane w rozdziale drugim. Należą do nich algorytmy: Eulera, Verleta, Stoermera, Rungego-Kutty drugiego i czwartego rzędu. Podczas tworzenia obiektu klasy dziedziczącej obowiązkowym argumentem konstruktora jest stała identyfikująca metodę używaną do rozwiązywania równań ruchu układu. Metodę tę wybiera się podając jako argument jeden z elementów typu wyliczeniowego: `EULER`, `VERLET`, `STOERM`, `RK2`, `RK4`. Po utworzeniu obiektu można zmienić metodę przydzieloną układowi podczas konstrukcji. Służy do tego funkcja `Set` wywoływana z argumentami `ALG`, oznaczającym zmianę algorytmu i elementem powyżej wymienionego typu określającym dany algorytm.

```
x.Set(ALG, RK4);
```

Powoduje to jednak znaczny skok rozbieżności obliczeń, a w rezultacie załamanie się symulacji. Funkcja `StepForward` dokonująca obliczeń wywoływana jest tylko z jednym argumentem: krokiem czasowym metody określonej w powyższy sposób, np.

```
x.StepForward(0.1f);
```

Implementacja algorytmów Eulera i Verleta w klasie zbioru punktów materialnych sprowadza się do wywołania odpowiednich metod w obiektach każdego punktu materialnego zbioru wchodzącego w skład modelu (listing 3).

Listing 3. Funkcje realizujące krok czasowy metodami Eulera i Verleta dla wszystkich cząstek ze zbioru

```
template<class type>
void ZbPktMat<type>::StepForwardEuler(type dt)
{
    CalculateForces();
    for(int i=0; i<size; i++)
        collection[i].StepForwardEuler(accelerations[i], dt);
}

template<class type>
void ZbPktMat<type>::StepForwardVerlet(type dt)
{
    CalculateForces();
    for(int i=0; i<size; i++)
        collection[i].StepForwardVerlet(accelerations[i], dt);
}
```

Pozostałe metody nie mogły być zaimplementowane w ten sposób. Ograniczenie polegało na tym, że przyspieszenie działające na cząstkę nie mogło być podane jako argument wywołania funkcji metody numerycznej. Pozostałe trzy metody numeryczne wymagają więcej niż jednego wyznaczenia sił działających na punkty materialne. Zadanie to realizowane jest przy pomocy funkcji `CalculateForces`, która musi być zaimplementowana w każdej klasie pochodnej. Innym rozwiązaniem mogłoby być zamieszczenie wszystkich metod numerycznych w klasie `PktMat`, tak jak w przypadku algorytmów Eulera i Verleta, a funkcja obliczająca przyspieszenia przekazywana była by przez wskaźnik. Rodzi to jednak problem z aktualizacją tablicy przyspieszeń będącą częścią klasy `ZbPktMat`. Spowodowałoby to zazębienie się klas `PktMat` i `ZbPktMat`, podczas gdy założenie wstępne polegało na utworzeniu zbioru punktów z elementów klasy `PktMat` i zachowaniu pewnej hierarchii(plików programu i klas). Użyte rozwiązanie zachowuje przejrzystą, czytelną i łatwą do zrozumienia strukturę programu jak i prostą implementację metod numerycznych. Funkcja dla niektórych algorytmów jest wywoływana więcej niż raz w ciągu kroku czasowego. Kod poszczególnych metod został zamieszczony poniżej.

Listing 4. Funkcja realizująca krok czasowy metodą Stoermera

```
template<class type>
void ZbPktMat<type>::StepForwardStoerm(type dt)
{
    int substeps = 10;
    type h = dt / substeps;
    type halfh = 0.5 * h;
    type two = 2.0;
    CalculateForces();
    for(int i = 0; i < Count(); i++)
        {
            Wektor<type> nowPosition = Get(CURR_POS, i);
            Set(PREV_POS, i, nowPosition);
            Wektor<type> tmp = nowPosition + h * (Get(VEL, i) + halfh
* accelerations[i]);
            Set(CURR_POS, i, tmp);
        }
    CalculateForces();
    for(int j = 1; j < substeps - 1; j++)
        {
            for(int i = 0; i < Count(); i++)
                {
                    Wektor<type> nowPosition = Get(CURR_POS, i);
                    Wektor<type> tmp = two * nowPosition -
Get(PREV_POS, i) + h * h * accelerations[i];
                    Set(PREV_POS, i, nowPosition);
                    Set(CURR_POS, i, tmp);
                }
            CalculateForces();
        }
    for(int i =0; i <Count(); i++)
        {
            Wektor<type> nowPosition = Get(CURR_POS, i);
            Wektor<type> tmp = two * nowPosition - Get(PREV_POS, i) +
h * h * accelerations[i];
            Set(PREV_POS, i, nowPosition);
            Set(CURR_POS, i, tmp);
            tmp = (Get(CURR_POS, i) - Get(PREV_POS, i)) / h + halfh *
accelerations[i];
            Set(VEL, i, tmp);
        }
}
```

Funkcja `CalculateForces`, obliczająca siły wypadkowe działające na poszczególne cząstki układu jest wywoływana na początku metody oraz po każdej zmianie położeń w trakcie jej działania, tj. po każdym przebiegu pętli odpowiadającym kolejnym krokom metody (implementacja wzorów 43 i 45).

Listing 4. Funkcja realizująca krok czasowy metodą Rungego-Kutty drugiego rzędu

```
template<class type>
void ZbPktMat<type>::StepForwardRK2(type dt)
{
    Wektor<type>* prev_vel = new Wektor<type>[Count()];
    type half = 0.5;
    CalculateForces();
    for(int i = 0; i < Count(); i++)//k1
        {
            Wektor<type> PositionNow = Get(CURR_POS, i);
            Set(PREV_POS, i, PositionNow);
            Wektor<type> VelocityNow = Get(VEL, i);
```

```
                prev_vel[i] = VelocityNow;

                Wektor<type> k1 = dt * VelocityNow;
                Wektor<type> tmp = PositionNow + half * k1;
                Set(CURR_POS, i, tmp);

                k1 = dt * accelerations[i];
                tmp = VelocityNow + half * k1;
                Set(VEL, i, tmp);
            }
        CalculateForces();
        for(int i = 0; i < Count(); i++)
            {
                Wektor<type> k2 = dt * Get(VEL, i);
                Wektor<type> tmp = Get(PREV_POS, i) + k2;
                Set(CURR_POS, i, tmp);

                k2= dt * accelerations[i];
                tmp = prev_vel[i] + k2;
                Set(VEL, i, tmp);
            }
        delete [] prev_vel;
    }
```

W implementacji algorytmu Runge-Kutty drugiego rzędu (listing 4) `CalculateForces` wywoływana jest dwa razy. Służy, zgodnie z algorytmem opisanym w rozdziale drugim, do obliczania współczynników wyznaczających wartości położenia i prędkości po upływie określonego czasu. Dynamicznie utworzona tablica `prev_vel` jest odpowiednikiem wartości `prev_pos` z klasy `PktMat`, służy bowiem do przechowywania prędkości cząstek z poprzedniej fazy obliczeń metody.

Listing 5. Funkcja realizująca krok czasowy metodą Rungego-Kutty czwartego rzędu

```
    template<class type>
    void ZbPktMat<type>::StepForwardRK4(type dt)
    {
        type half = 0.5;
        type two = 2.0;
        type one_by_six = 1.0 / 6.0;
        Wektor<type>* sum_kv = new Wektor<type>[Count()];
        Wektor<type>* sum_kr = new Wektor<type>[Count()];
        Wektor<type>* prev_vel = new Wektor<type>[Count()];
        CalculateForces();
        for(int i = 0; i < Count(); i++)
            {
                Wektor<type> tmp = Get(CURR_POS, i);
                Set(PREV_POS, i, tmp);
                tmp = Get(VEL, i);
                prev_vel[i] = tmp;

                Wektor<type> k1 = dt * Get(VEL, i);
                sum_kr[i] = k1;
                tmp = Get(PREV_POS, i) + half * k1;
                Set(CURR_POS, i, tmp);

                k1 = dt * accelerations[i];
                sum_kv[i] = k1;
                tmp = prev_vel[i] + half * k1;
```

```
                Set(VEL, i, tmp);
            }
        CalculateForces(); //dla yn + k1/2
        for(int i =0; i < Count(); i++)
             {
                 Wektor<type> k2 = dt * Get(VEL, i);
                 sum_kr[i] += (two * k2);
                 Wektor<type> tmp = Get(PREV_POS, i) + half * k2;
                 Set(CURR_POS, i, tmp);

                 k2 = dt * accelerations[i];
                 sum_kv[i] += (two * k2);
                 tmp = prev_vel[i] + half * k2;
                 Set(VEL, i, tmp);
             }
        CalculateForces();//oblicza f(yn + k2/2)
        for(int i =0; i < Count(); i++)
            {
                Wektor<type> k3 = dt * Get(VEL, i);//h * f(yn + k2/2)
                sum_kr[i] += (two * k3);
                Wektor<type> tmp = Get(PREV_POS, i) + k3;
                Set(CURR_POS, i, tmp);

                k3 = dt * accelerations[i];
                sum_kv[i] += (two * k3);
                tmp = prev_vel[i] + k3;
                Set(VEL, i, tmp);
            }

        CalculateForces();
        for(int i = 0; i < Count(); i++)
            {
                Wektor<type> k4 = dt * Get(VEL, i);
                sum_kr[i] += k4;
                Wektor<type> tmp = Get(PREV_POS, i) + one_by_six *
    sum_kr[i];
                Set(CURR_POS, i, tmp);

                k4 = dt * accelerations[i];
                sum_kv[i] += k4;
                tmp = prev_vel[i] + one_by_six * sum_kv[i];
                Set(VEL, i, tmp);
            }
        delete [] sum_kv;
        delete [] sum_kr;
        delete [] prev_vel;
    }
```

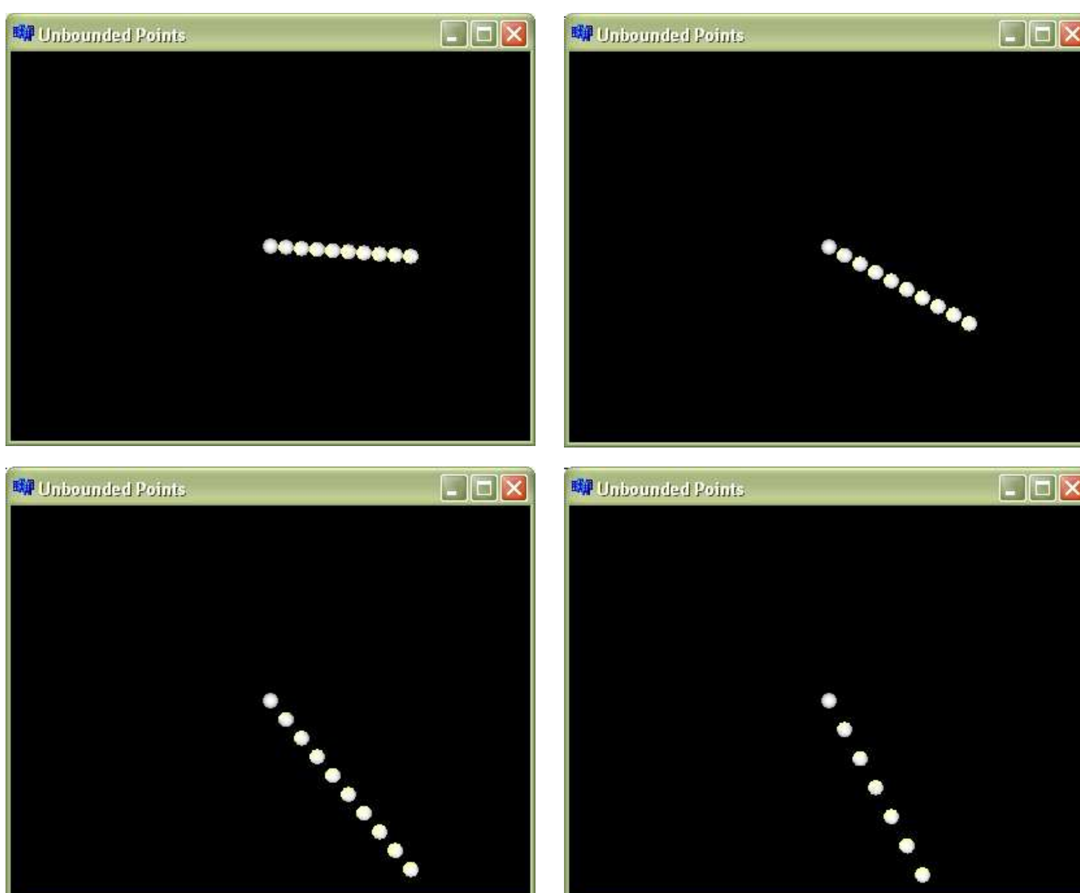
W przypadku metody Runge-Kutty czwartego rzędu liczba wywołań funkcji `CalculateForces` rośnie do czterech, analogicznie do metody MidPoint. Dynamicznie utworzone na początku funkcji tablice `sum_kr` i `sum_kv` służą do przechowywania sum poszczególnych współczynników k dla położenia i prędkości (zob. wzór 25).

3.3.2. Klasy rozszerzające szablon `ZbPktMat` – implementacje przykładowych modeli

W ramach projektu zostało utworzonych kilka klas pochodnych względem `ZbPktMat` (ich opis znajduje się poniżej). Projekt jest otwarty i umożliwia tworzenie własnych klas implementujących różnego typu układy fizyczne.

3.3.2.1. Swobodne punkty

`UnboundedPionts` (swobodne punkty) jest najprostszą klasą dziedziczącą z klasy `ZbPktMat`. Modelowany przez nią układ składa się z określonej w momencie tworzenia obiektu ilości punktów materialnych o jednostkowej masie ułożonych w rzędzie wzdłuż osi x.



Rys. 4. Punkty materialne z działającymi na nie siłami proporcjonalnymi do ich położenia w szeregu

Klasa ta pozwoliła na przeprowadzenie podstawowych testów implementacji metod numerycznych i ich ocenę za pomocą wizualizacji przygotowanej w OpenGL. Symulacja pokazuje naturalnie, że cząstki o większym przyspieszeniu spadają szybciej. Kod klasy realizujący powyższy układ przedstawiony jest na listingu 6.

Listing 6. Konstruktor klasy `UnboundedPoints` i funkcja `CalculateForces` tej klasy

```
template<class type>
UnboundedPoints<type>::UnboundedPoints(algorithms alg, int size) :
ZbPktMat<type>(size)
{
    Set(ALG, alg);
    for(int i=1; i<Count(); i++)
    {
        Set(CURR_POS, i, i, 0.0, 0.0);
        Set(PREV_POS, i, i, 0.0, 0.0);
        Set(VEL, i, 0.0, 0.0, 0.0);
    }
}

template<class type>
void UnboundedPoints<type>::CalculateForces()
{
    for(int i = 0; i < Count(); i++)
        accelerations[i].Set(0.0, -0.1*i, 0.0);
}
```

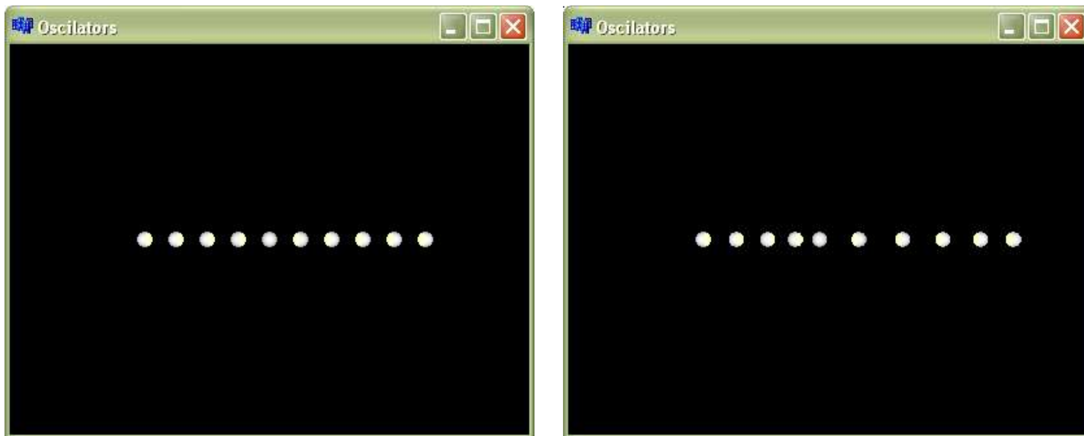
Każda cząstka ma stałe przyspieszenie proporcjonalne do numeru jej indeksu w zbiorze punktów materialnych.

3.3.2.2. Łańcuch oscylatorów

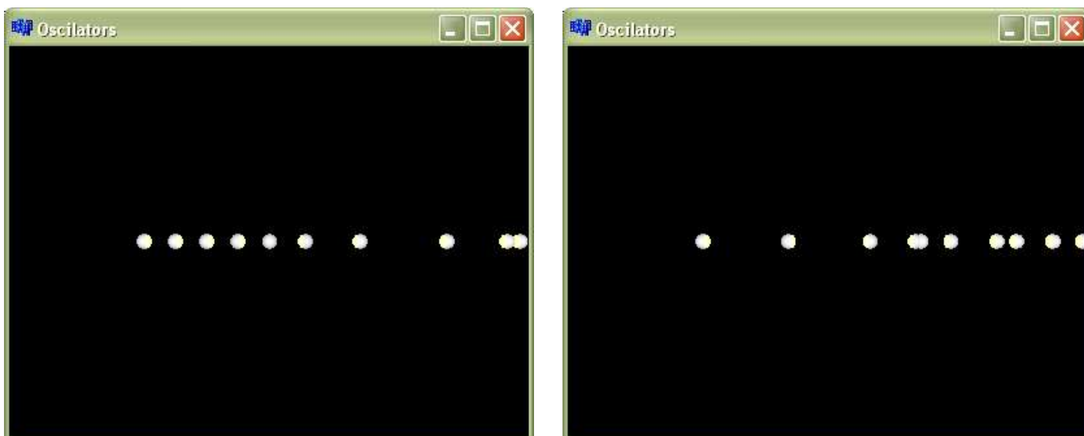
Kolejną klasą rozszerzającą `ZbPktMat` jest klasa `Oscilators`. Podobnie, jak w poprzednim modelu cząstki ułożone są w rzędzie wzdłuż osi x. Jednak tym razem każda cząstka układu oddziałuje z dwoma sąsiednimi siłą harmoniczną, tj. wprost proporcjonalną do wychylenia z położenia równowagi Δx o współczynniku proporcjonalności zadanym przez stałą sprężystości k:

$$F = -k \cdot \Delta x \tag{63}$$

Na poniższych rysunkach przedstawione zostały wyniki symulacji, w której ostatniej cząstce nadawana została początkowa prędkość 2.0. Zmieniana była stała sprężystości. W pierwszym przypadku równa 1.0, a w drugim 0.1.



Rys. 5. Łańcuch oscylatorów w położeniu równowagi i w ruchu. Stała sprężystości wynosi 1.0



Rys. 6. Łańcuch oscylatorów w ruchu. Stała sprężystości 0.1

Zmiana stałej sprężystości na mniejszą powoduje rozluźnienie się wiązań między punktami. W konsekwencji „później" reagują one na zmianę pozycji swoich sąsiadów. Cały łańcuch oscylatorów rozciąga się i ściska w większej skali niż w przypadku dużej stałej sprężystości.

Konstruktor obiektu klasy `Oscilators` jest podobny do konstruktora klasy `UnboundedPoints`:

Listing 7. Konstruktor klasy modelującej zbiór oscylatorów

```
template<class type>
Oscilators<type>::Oscilators(algorithms alg, int size, type inc, type
elst, type x, type y, type z) : ZbPktMat<type>(size)
{
    Set(ALG, alg);
    increment = inc;
    elasticity = elst;
    for(int i=0; i<Count(); i++)
    {
        Set(CURR_POS, i, increment*i+x, y, z);
        Set(PREV_POS, i, increment*i+x, y, z);
        Set(VEL, i, 0.0, 0.0, 0.0);
        Set(MASS, i, 1.0);
    }
    Set(CONS, 0, true); }
```

Przyjmuje jednak kilka nowych argumentów. Można określić odległości między oscylującymi punktami (domyślnie wartość 2.0), zdefiniować stałą sprężystości (domyślnie 1.0) i początkowe położenie pierwszej cząstki modelu (domyślnie środek układu współrzędnych). Rzeczywistą różnicę stanowi jednak funkcja składowa wyznaczająca siły działające na cząstki (listing 8).

Listing 8. Funkcja wyznaczająca siły działające na cząstki w klasie `Oscilators`

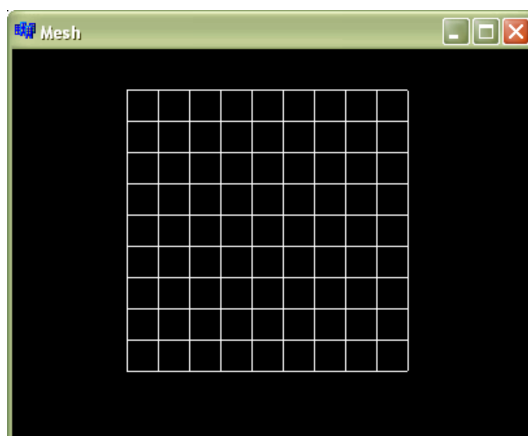
```
template<class type>
void Oscilators<type>::CalculateForces()
{
    //const type attenuationConstant=0.1;
    const type springConstant = elasticity;
    const type equilibriumPoint = increment;
    for(int i=0;i<Count();i++)
        {
            if (IsHanged(i)) continue;
            forces[i].Set(0.0, 0.0, 0.0);
            Wektor<type> tmp = Get(CURR_POS, i);
            if (i>0)//do lewego
                {
                    Wektor<type> distance = tmp - Get(CURR_POS, i-1);
                    forces[i] += -distance.Unit() *
(distance.Length() - equilibriumPoint) * springConstant;
                }
            if (i<Count()-1)//do prawego
                {
                    Wektor<type> distance = tmp - Get(CURR_POS, i+1);
                    forces[i] += -distance.Unit() *
(distance.Length() - equilibriumPoint) * springConstant;
                }
            //forces[i] += Get(VEL, i) * -attenuationConstant;
            accelerations[i] = forces[i] / GetMass(i);
        }
}
```

Obliczana w tej metodzie siła jest wypadkową oddziaływania z dwoma sąsiadami z lewej i prawej strony, oczywiście o ile tacy istnieją. Możemy nałożyć na poszczególne cząstki więzy – należy polu klasy `PktMat` o nazwie `constraints` nadać wartość `true` – wówczas siły działające na tą cząstkę nie są wyznaczane. Zakomentowany w listingu 8 fragment kodu odpowiedzialny jest za tłumienie drgań.

3.3.2.3. Siatka

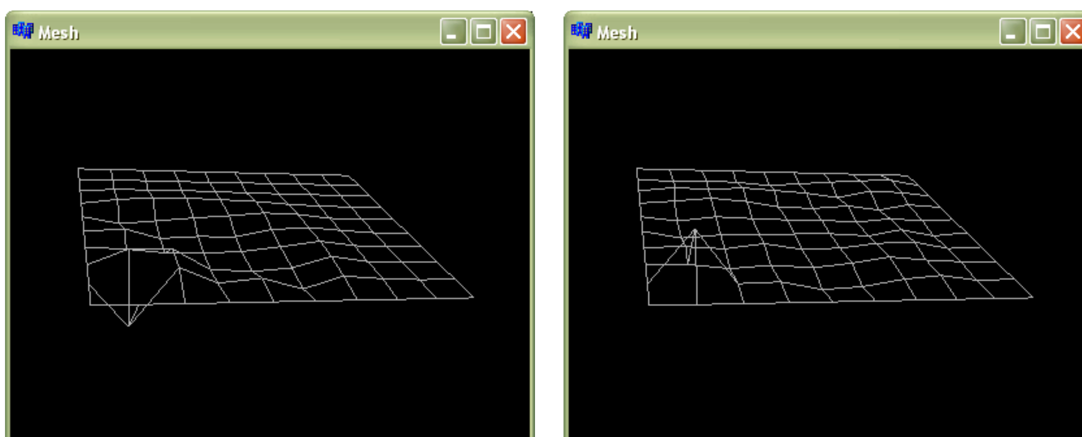
Klasą podobną do łańcucha oscylatorów jest klasa `Mesh` – implementuje dwuwymiarową siatkę drgających punktów. Konstruując obiekt poza wyborem algorytmu można zmienić domyślną wartość 10 cząstek w jednym rzędzie kwadratowej siatki, a także odległość między punktami. Cząstki oddziałują z punktami sąsiednimi zarówno z lewej i prawej strony jak i góry, dołu, a także znajdujących się po przekątnych. Oddziaływanie to jest

takie samo jak w przypadku opisanym wcześniej tj. cząstki związane są siłami harmonicznymi.



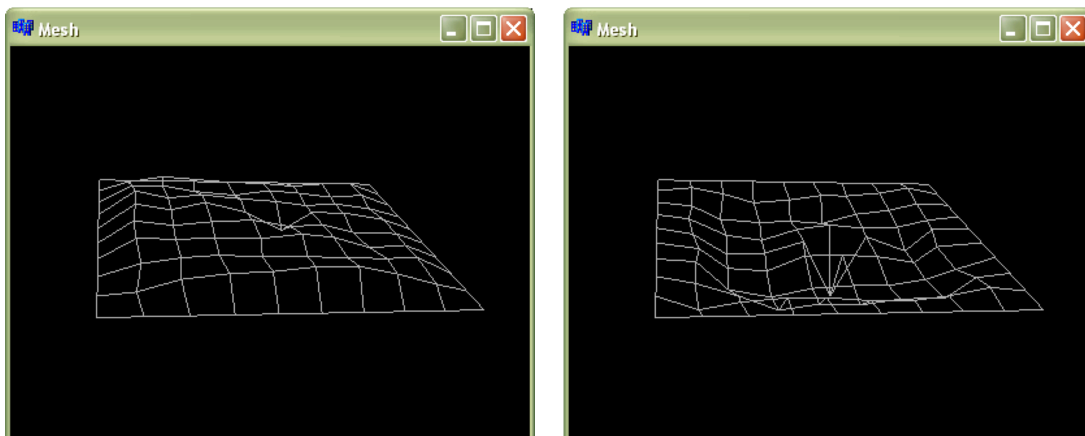
Rys. 7. Siatka nieruchoma w rzucie z góry

Zaburzenie w siatce generowane jest przez nadanie prędkości początkowej jednemu lub kilku punktom, z których składa się model. W przypadkach pokazanych na poniższych rysunkach prędkość nadawana poszczególnym punktom wynosiła 1.0 i skierowana pionowo w górę.



Rys. 8. Zaburzenie rozchodzące się od lewego rogu. Stała sprężystości równa 1.0

Zmiana stałej sprężystości w sile, z jaką oddziałują z sobą cząstki układu powoduje inną prędkość rozchodzenia się zaburzenia w siatce, a także amplitudę drgań. Zmniejszenie tej wartości o połowę powoduje znaczne zmiany w przebiegu symulacji.



Rys. 9. Zaburzenie rozchodzące się z centralnego punktu modelu. Stała sprężystości wynosi 0.5

Konstruktor działa podobnie jak w klasach opisanych poprzednio. Określany jest zatem algorytm numeryczny, wymiar siatki, współczynnik sprężystości oraz położenie pierwszego punktu układu. Konstruktor z podanej wcześniej ilości punktów formuje siatkę w kształcie kwadratu.

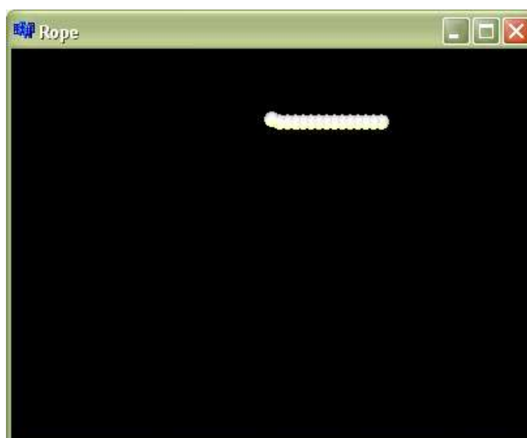
Listing 9. Konstruktor tworzący siatkę dwuwymiarową oscylatorów

```
template<class type>
Mesh<type>::Mesh(algorithms alg, int dim, type inc, type elst, type
x, type y, type z) : ZbPktMat<type>(dim*dim)
{
    Set(ALG, alg);
    dimension = dim;
    increment = inc;
    elasticity = elst;
    int index = 0;
    for(int i=0; i<dimension; i++)
        for(int j=0; j<dimension; j++)
            {
                Set(CURR_POS, index, increment*j+x, 0.0+y, -
increment*i+z);
                Set(PREV_POS, index, increment*j+x, 0.0+y, -
increment*i+z);
                Set(VEL, index, 0.0, 0.0, 0.0);
                Set(MASS, index, 1);
                if(i==0 || i==(dimension-1) || j==0 || j==(dimension-
1))
                Set(CONS, index, true);
                index++;
            }
}
```

Siła działająca na cząstkę, podobnie jak w przypadku oscylatorów, wyznaczana jest dla wszystkich punktów poza tymi, na które nałożone są więzy położenia. Różnica w stosunku do oscylatorów jest taka, że oddziaływanie liczone jest ze wszystkimi punktami bezpośrednio powiązanymi z daną cząstką, a nie tylko dwoma sąsiadami.

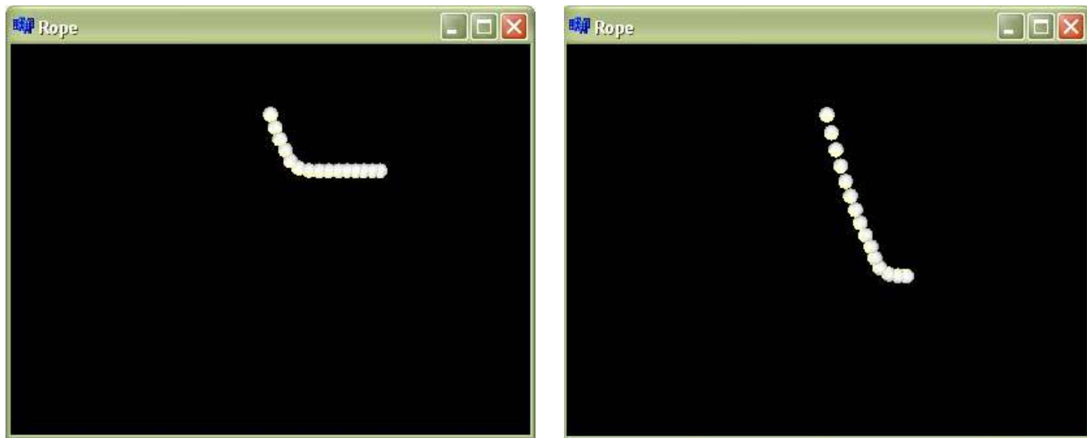
3.3.2.4. Lina

Klasa `Rope` (Lina) jest modelem składającym się z punktów materialnych połączonych ze sobą siłami sprężystymi. Na wszystkie cząstki poza pierwszą, która jest punktem zaczepienia liny, poza siłą wypadkową pochodzącą od sąsiadów działa także siła grawitacji skierowana równolegle do wartości ujemnych osi *y*. Podobnie jak w poprzednich klasach rozszerzających klasę `ZbPktMat` za pomocą argumentów konstruktora zmienić można ilość punktów, z których zbudowany jest układ (domyślnie 10) włączając w to punkt zaczepienia.



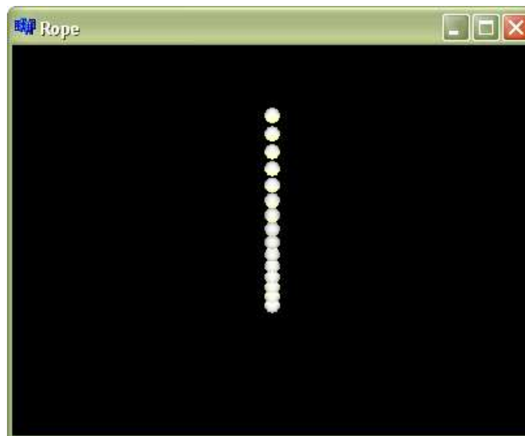
Rys. 10. Początkowe położenie liny

Początek symulacji to lina ułożona poziomo (rys. 10). Wszystkie cząstki znajdują się w położeniach równowagi wobec swoich sąsiadów. Masa każdej cząstki wynosi 0.1, a odległość między nimi to 0.5. Stała sprężystości *k* oddziaływania cząstek na siebie jest równa 200, a stała tłumienności drgań sprężystych 0.3. Na wszystkie punkty liny działa siła grawitacji, a właściwie przyspieszenie ziemskie równe 9.81. Podczas opadania liny widać jak grawitacja działa na wszystkie cząstki, a także jak powiązanie z punktem zaczepienia zakrzywia tor opadających punktów składających się na linę (por. rys. 11). Widać drgania spowodowane siłami harmonicznymi.



Rys. 11. Fazy opadania liny

Po stłumieniu oscylacji napięcie liny (odległości pomiędzy poszczególnymi punktami) zależy od odległości od punktu zaczepienia. Punkty blisko punktu zaczepienia są bardziej oddalone ze względu na ciężar punktów „podczepionych do nich" (rys. 12). Lina „rozciąga się" pod własnym ciężarem.



Rys. 12. Wisząca lina

Konstruktor klasy `Rope` jest podobny do tych, z opisanych wyżej klas. Jego argumenty są identyczny, jak w przypadku klasy implementującej model sprzężonych oscylatorów (listing 10).

Listing 10. Konstruktor liny

```
template<class type>
Rope<type>::Rope(algorithms alg, int size, type x, type y, type z) :
ZbPktMat<type>(size)
{
    Set(ALG, alg);
    increment = 0.5;
        type dx = increment;
    for(int i=0; i<Count(); i++)
    {
        Set(CURR_POS, i, x+dx*i, y, z);
```

```
            Set(PREV_POS, i, x+dx*i, y, z);
            Set(VEL, i, 0.0, 0.0, 0.0);
            Set(MASS, i, 0.3);
        }
        Set(CONS, 0, true);
    }
```

Również metoda `CalculateForces` jest podobna do tej, z klasy oscylatorów, z tym, że oprócz sił sprężystych i tłumienia dodana została siła grawitacji (listing 11).

Listing 11. Metoda obliczająca siły działające na każdą cząstkę liny

```
template<class type>
void Rope<type>::CalculateForces()
{
    //stale fizyczne
    const type springConstant=200.0;
    const type attenuationConstant=0.3;
    const type equilibriumPoint=increment;
    const Wektor<type> gravitationalConstant(0.0, -9.81, 0.0);
    //obliczanie sil
    for(int i=0;i<Count();i++)
        {
            if (IsHanged(i)) continue;
            forces[i].Set(0.0, 0.0, 0.0);
            Wektor<type> distance(0.0, 0.0, 0.0);
            Wektor<type> tmp = Get(CURR_POS, i);
            if (i > 0)//oddzialywanie z lewym sasiadem
                {
                    distance = tmp - Get(CURR_POS, i-1);
                    forces[i] += -distance.Unit() *
(distance.Length() - equilibriumPoint) * GetMass(i) * springConstant;
                }
            if (i < Count() - 1)//oddzialywanie z prawym sasiadem
                {
                    distance = tmp - Get(CURR_POS, i+1);
                    forces[i] += -distance.Unit() *
(distance.Length() - equilibriumPoint) * GetMass(i) * springConstant;
                }
            forces[i] += Get(VEL, i) * -attenuationConstant;
            accelerations[i] = forces[i] / GetMass(i);
            accelerations[i] += gravitationalConstant;
        }
}
```

Lina jest pierwszym modelem, w którym przygotowany przeze mnie kod został użyty do nietrywialnego obiektu mającego symulować rzeczywisty układ. Efekt jest bardzo zadowalający – wynik symulacji dobrze oddaje ruch prawdziwej liny w ośrodku. Bez trudu można powyższą klasę zmodyfikować tak, aby modelowała ruch włosów lub trawy – to standardowe już problemy symulowane na przykład przy produkcji filmów animowanych komputerowo. Po raz pierwszy szerokiej publiczności podobny efekt pokazany został bodajże w filmach Shrek 2 i Epoka Lodowcowa 2, czyli całkiem niedawno. Oczywiście tam

symulowany był nie jeden włos, a tysiące. Jednak przy produkcji filmów symulacja nie odbywa się w czasie rzeczywistym – jedna klatka może być renderowana godzinami.

3.3.2.5. Cząstki oddziałujące na siebie potencjałem Morse'a.

W ostatnim zagadnieniu uczyniłem ukłon w stronę dynamiki molekularnej, która była jedną z inspiracji do przygotowania omawianego projektu. Klasa `Morse` implementuje model, w którym cząstki oddziałują na siebie zgodnie z potencjałem Morse'a [15, 16] – standardowy model oddziaływania cząstek odpychających się, gdy są bardzo blisko i przyciągających się, gdy są dalej. Oddziaływanie dąży do zera przy odległości zwiększającej się do nieskończoności. Potencjał Morse'a zadany jest wzorem:

$$V(x) = D \cdot \{1 - \exp[-b \cdot (x - x_0)]^2\} - 1, \tag{64}$$

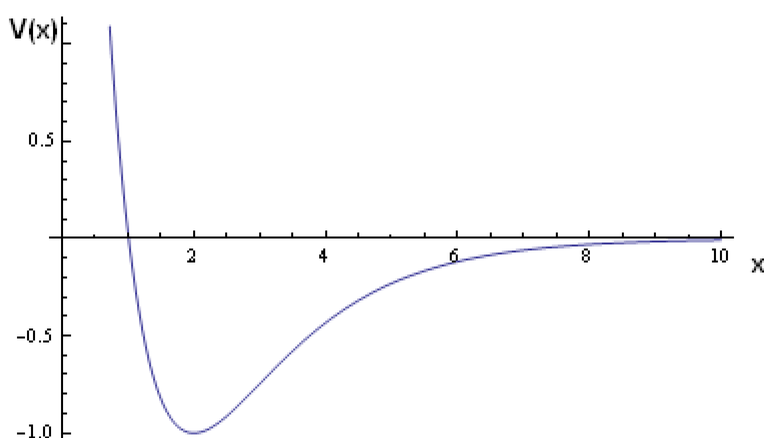
zatem siła dana jest przez:

$$F(x) = -V'(x) = 2 \cdot b \cdot D \cdot \exp[-b \cdot (x - x_0)] \cdot \{-1 \cdot \exp[-b \cdot (x - x_0)]\}, \tag{65}$$

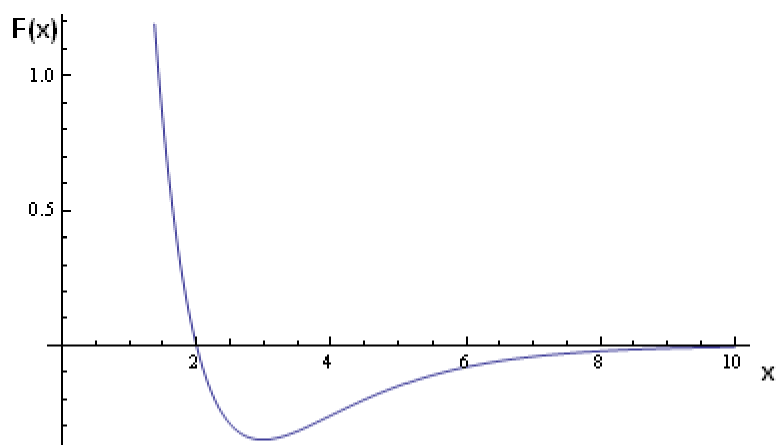
gdzie: D – oznacza głębokość studni potencjału, x – odległość między cząstkami, x_0 – punkt równowagi, a b – jest zdefiniowane jako:

$$b = \sqrt{\frac{k}{2 \cdot D}}, \tag{66}$$

gdzie k – stała siłowa wiązania.



Wyk. 1. Wykres potencjału Morse'a. Dane do wykresu: D=1, k=1, x_0=2, b=0.7



Wyk. 2. Wykres siły. Dane jak w wyk. 1

Początkowo cząstki w modelu ułożone są w sześcian. Po chwili siły, z jakimi na siebie działają formują je w pulsującą kulę. Powstały twór można próbować bombardować cząstką i testować czy da się w ten sposób wybić jedną z molekuł układu. Sytuacja taka symulowana jest za pomocą obiektu klasy `Morse`. Na listingu 12 przedstawiony jest konstruktor klasy modelującej powyżej opisany układ.

Listing 12. Konstruktor klasy `Morse`.

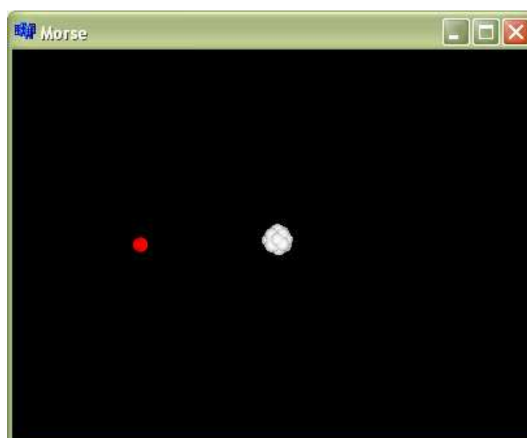
```
template<class type>
Morse<type>::Morse(algorithms alg, type D, type k, type x0) :
ZbPktMat<type>(28)
{
    Set(ALG, alg);
    DepthWellConstant = D;
    BondForceConstant = k;
    EquilibriumBondConstant = x0;
    type inc = 0.5;
    int index = 0;
    for(int k = 0; k < 3; k++)
        {
            for(int i = 0; i < 3; i++)
                for(int j = 0; j < 3; j++)
                    {
                        Set(CURR_POS, index, inc*j, inc*i, inc*k);
                        Set(PREV_POS, index, inc*j, inc*i, inc*k);
                        Set(MASS, index, 1.0);
                        index++;
                    }
        }//cube
    //bullet
    Set(CURR_POS, Count()-1, -10, 0.0, 0.0);
    Set(PREV_POS, Count()-1, -10, 0.0, 0.0);
    Set(MASS, Count()-1, 1.0);
}
```

Argumentami konstruktora poza określeniem metody numerycznej, są takie wielkości jak głębokość studni potencjału (domyślnie ustawiona na wartość 1.0), stała wiązania międzycząsteczkowego (domyślnie 0.5) oraz odległość między punktami, dla której potencjał

przyjmuje wartość minimalną (domyślnie 0.5). Siła działająca na każdą z cząstek układu obliczana jest na podstawie wzoru (65) przez funkcję z listingu 13.

Listing 13. Funkcja obliczająca siły działające na cząstki w klasie `Morse`

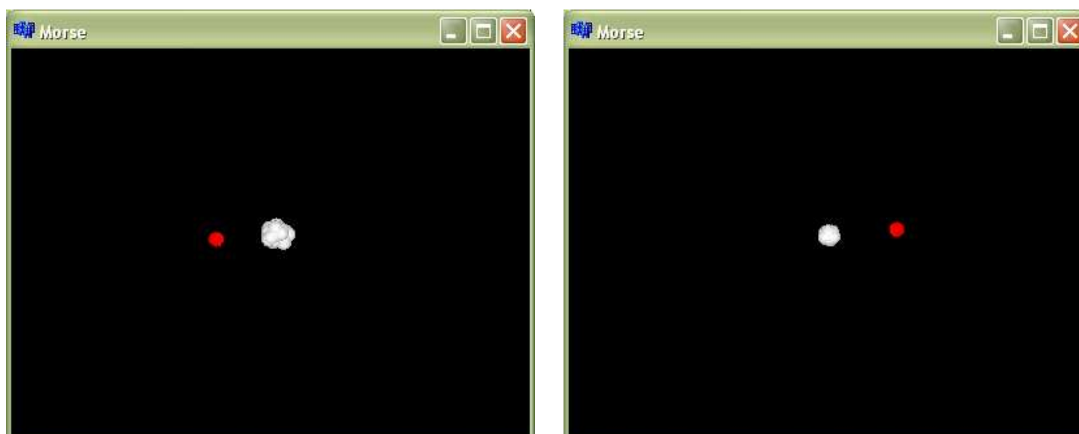
```
template<class type>
void Morse<type>::CalculateForces()
{
    const type D = DepthWellConstant;
    const type b = sqrt(BondForceConstant / (2.0*DepthWellConstant));
    const type r0 = EquilibriumBondConstant;
    const type dwa = 2;
    for(int i = 0; i < Count(); i++)
        {
            if(IsHanged(i)) accelerations[i].Set(0.0, 0.0, 0.0);
            else
                forces[i].Set(0.0, 0.0, 0.0);
                for(int j = 0; j < Count(); j++)
                    {

                        if( i == j) continue;
                        Wektor<type> tmp=Get(CURR_POS, i)-
Get(CURR_POS, j);
                        type distance = tmp.Length();
                        type wyr1 = exp(-b*(distance-r0));
                        type wyr2 = -1 + exp(-b*(distance-r0));
                        forces[i] += tmp.Unit()*dwa*b*D*wyr1*wyr2;
                    }
            accelerations[i] = forces[i] / GetMass(i);
        }
}
```



Rys. 13. Cząstki oddziałujące na siebie potencjałem Morse'a bombardowane pojedynczą cząstką

Każda symulacja rozpoczyna się takim samym stanem układu. Dwadzieścia siedem cząstek ułożonych początkowo w sześcian (po trzy cząstki w linii tworzącej krawędź figury) i oddalonych od siebie o 0.3 jednostki długości, jest bombardowane przez molekułę znajdującą się po lewej stronie w odległości 10 jednostek długości. Masy wszystkich punktów są jednostkowe. Badana jest reakcja układu na uderzenie punktem o różnych prędkościach początkowych. Podczas symulacji udało się zauważyć tylko dwa zachowania układu.

Pierwsze polegało na wciągnięciu pocisku do układu, a drugie na przepuszczeniu go z spowolnieniem lub przyspieszeniem. Wszystkie zachowania zachodziły dla takich samych domyślnych warunków brzegowych modelu, a zmieniana była tylko prędkość cząsteczki bombardującej.

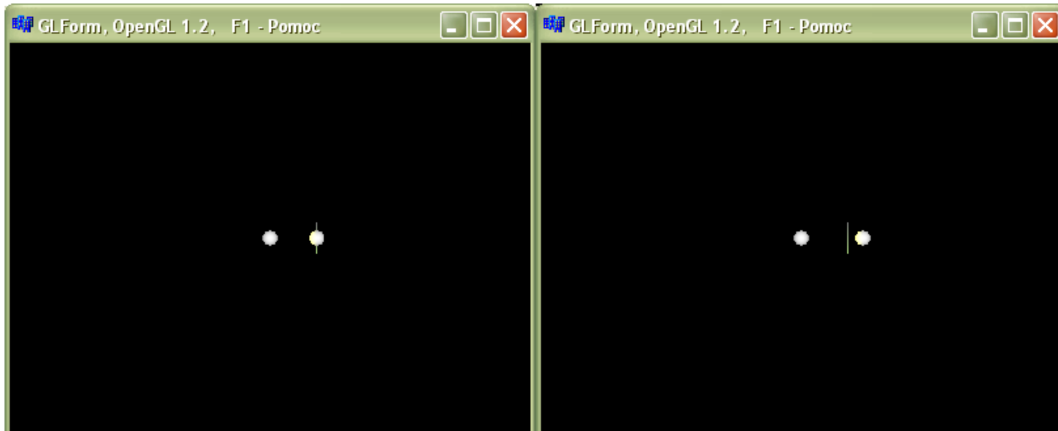


Rys. 14. Przyspieszenie cząstki i odchylenie jej toru ruchu

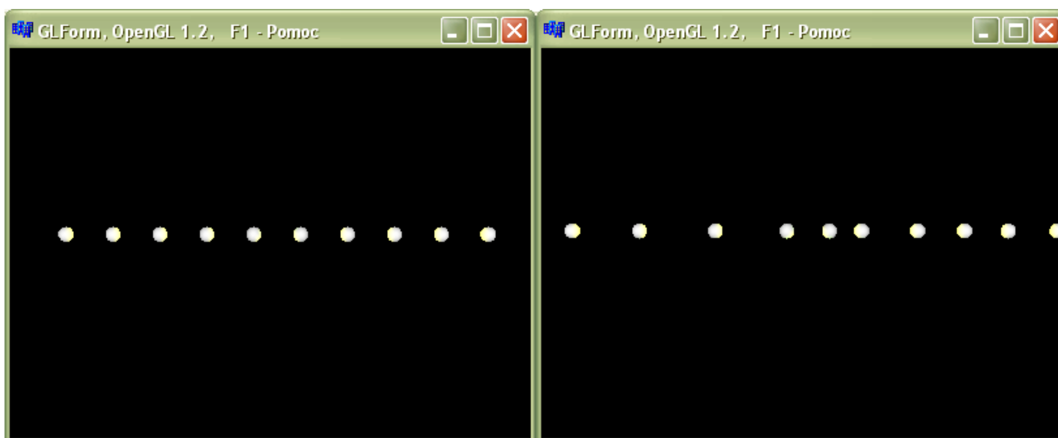
Dla małej prędkości pocisku - 0.5 wraz z jego zbliżaniem się do układu jest on coraz bardziej przyspieszany przez oddziaływanie z pozostałymi cząstkami, następnie przebija się przez kulę i ucieka. Zjawisko to zachodzi także dla dużych prędkości (np. 2.0), ale wtedy przyspieszanie pocisku nie jest dostrzegalne jak w powyższym przypadku. Dla niektórych prędkości np. 1.0 pocisk po przebiciu układu zostaje wyhamowany i zawrócony. Tworzy z pulsującą kulą układ pewnego oscylatora. Zostaje ściągany do środka obiektu, prędkość nadana tym sposobem pozwala na przetunelowanie na drugą stronę i sytuacja się powtarza. Bombardująca cząstka przenika przez układ z lewej strony na prawą i odwrotnie. Im dalej jest od niego prędkość staje się coraz mniejsza, aż do zerowej a następnie zawraca.

4. Testy

Metody numeryczne opisane w rozdziale drugim zostały przetestowane pod względem szybkości i dokładności. Pierwszym modelem użytym w testach były dwa punkty materialne oddziaływujące na siebie siłami harmonicznymi (sprzężone oscylatory). Drugim modelem było dziesięć takich punktów ustawionych w rzędzie (rys. 18 i 19).



Rys. 18. Dwa punkty materialne połączone siłami sprężystymi. Położenie równowagi zaznaczone jest pionową kreską



Rys. 19. Dziesięć punktów oddziaływujących na swoich sąsiadów siłami harmonicznymi (po lewej w położeniu równowagi, po prawej w ruchu)

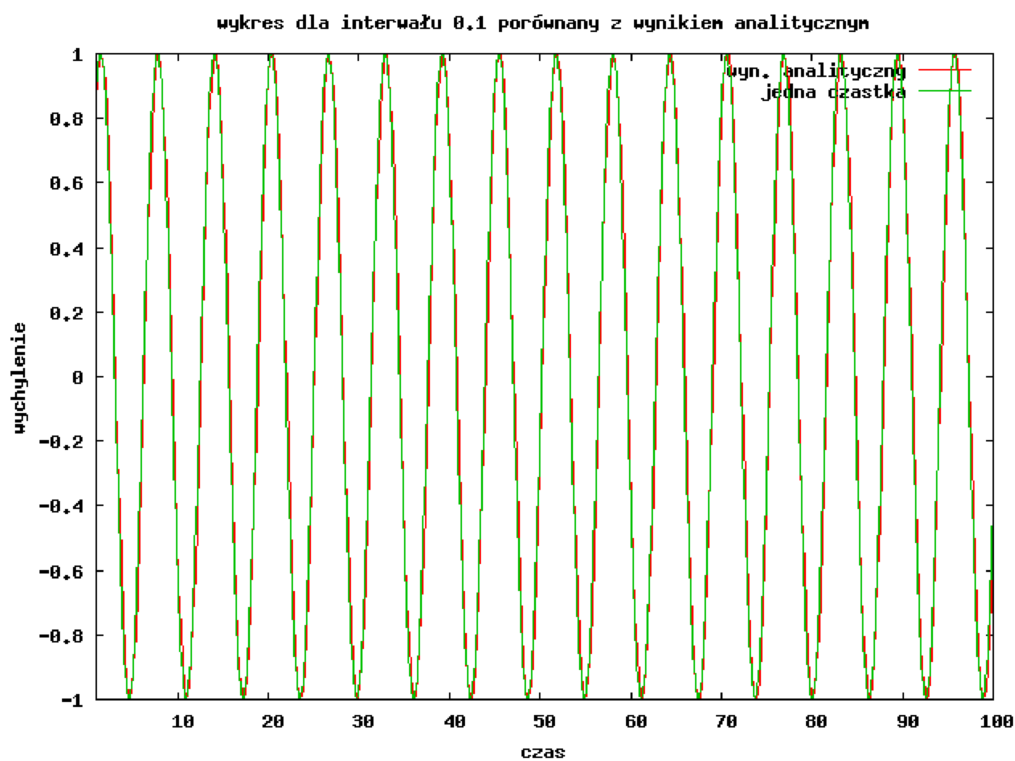
Wszystkie testy rozpoczynały się od wprawienia w ruch ostatniego punktu poprzez nadanie mu prędkości początkowej. W próbie każdego algorytmu wykonane zostało 1000 kroków czasowych o tej samej długości. Obserwowanymi wielkościami były rzeczywisty czas trwania obliczeń (ang. *wall time*) oraz położenie każdego punktu w funkcji czasu (wychylenie z położenia równowagi). Wykresy tych ostatnich były porównywane w celu oceny zbieżności numerycznej algorytmów.

Opisane powyżej testy zostały wykonane dla dwóch typów danych zmiennoprzecinkowych: pojedynczej i podwójnej precyzji (`float` i `double`). Konkretny

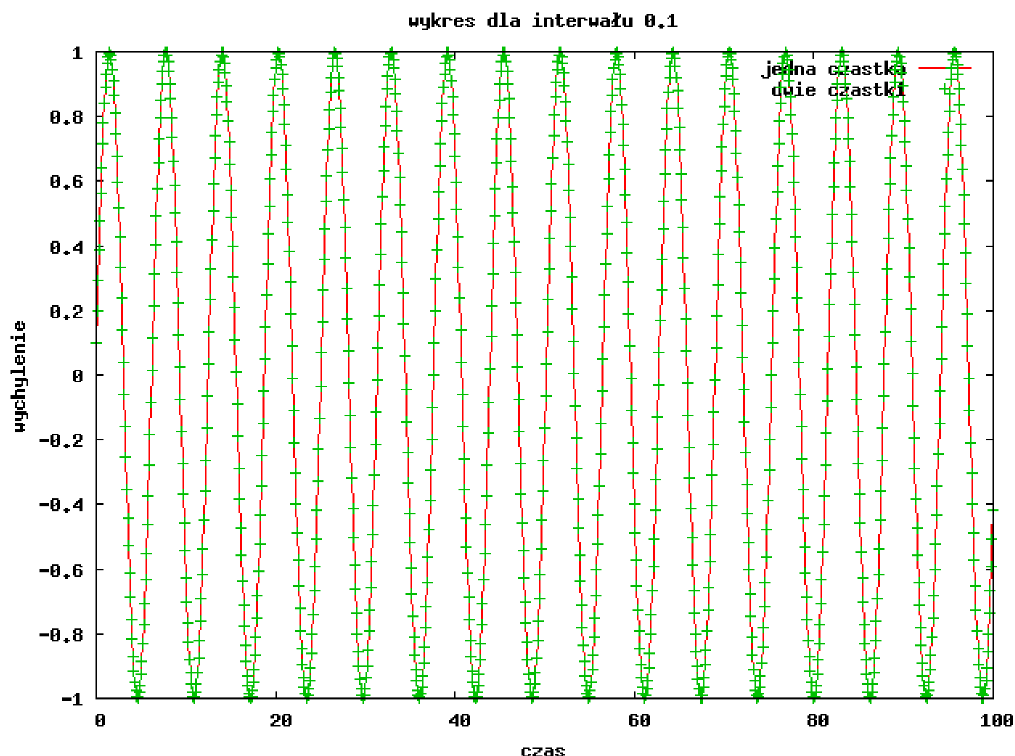
typ danych używanych był konsekwentnie w całym kodzie do reprezentacji wszystkich zmiennych niezbędnych do przeprowadzenia symulacji takich jak położenia punktów, prędkości, przyspieszenia, masy, siły, współczynniki sprężystości, tłumienia, itp. Jest oczywistym, że obliczenia są dokładniejsze w przypadku typu `double`, a szybsze w przypadku `float`. Celem tego testu było jednak sprawdzenie, czy w przypadku symulacji, których głównym celem jest prezentacja wyników w przestrzeni 3D, obliczenia pojedynczej precyzji są zadowalające (biblioteka OpenGL korzysta jedynie z typu `float`) oraz czy różnica prędkości jest rzeczywiście istotnie różna.

4.1. Testy zbieżności

Pierwsze testy wykonane zostały dla jednego oscylatora z działającą na niego siłą harmoniczną. W modelu wykorzystano dwa punkty materialne, ale położenie pierwszego z nich zostało na stałe związane z początkiem układu współrzędnych. Parametry drugiego punktu to: masa m=1.0, stała sprężystości k=1.0. W ten sposób wynik symulacji można porównać z wynikiem analitycznym (zadanym funkcją $\cos(\omega t)$, gdzie ω równa się pierwiastkowi z ilorazu stałej sprężystości k i masy m, w tym wypadku ω wynosi 1) i względem niego ocenić dokładność obliczeń.. Po nadaniu mu prędkości początkowej o wartości +1.0 (kierunek osi X, zwrot w prawo) zaczynał oscylować względem początku układu współrzędnych. Na rysunku 3 pokazano położenie oscylującego punktu w funkcji czasu oraz wynik analityczny. Na rysunku 4 porównane jest wychylenie jednej drgającej cząstki z układem cząstki oddziałującej z drugą, nieruchomą. Łatwo można zauważyć, że wykresy te pokrywają się. Takie same rezultaty otrzymane są trzema różnymi sposobami.



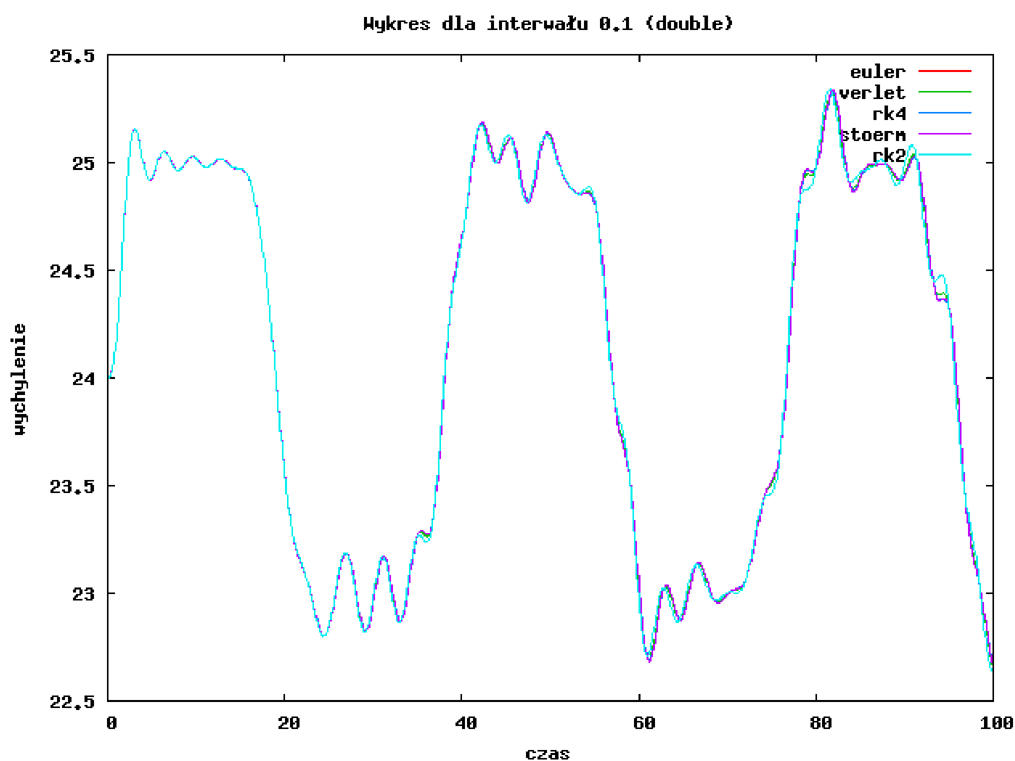
Wyk. 3. Wykres wychylenia w funkcji czasu dla: jednej cząstki w polu sił – zielony, wynik analityczny obliczony na podstawie danych symulacji – czerwony



Wyk. 4. Wykres wychylenia w funkcji czasu cząstki oddziałującej z drugą- nieruchomą, porównany z pojedynczą oscylującą cząstką

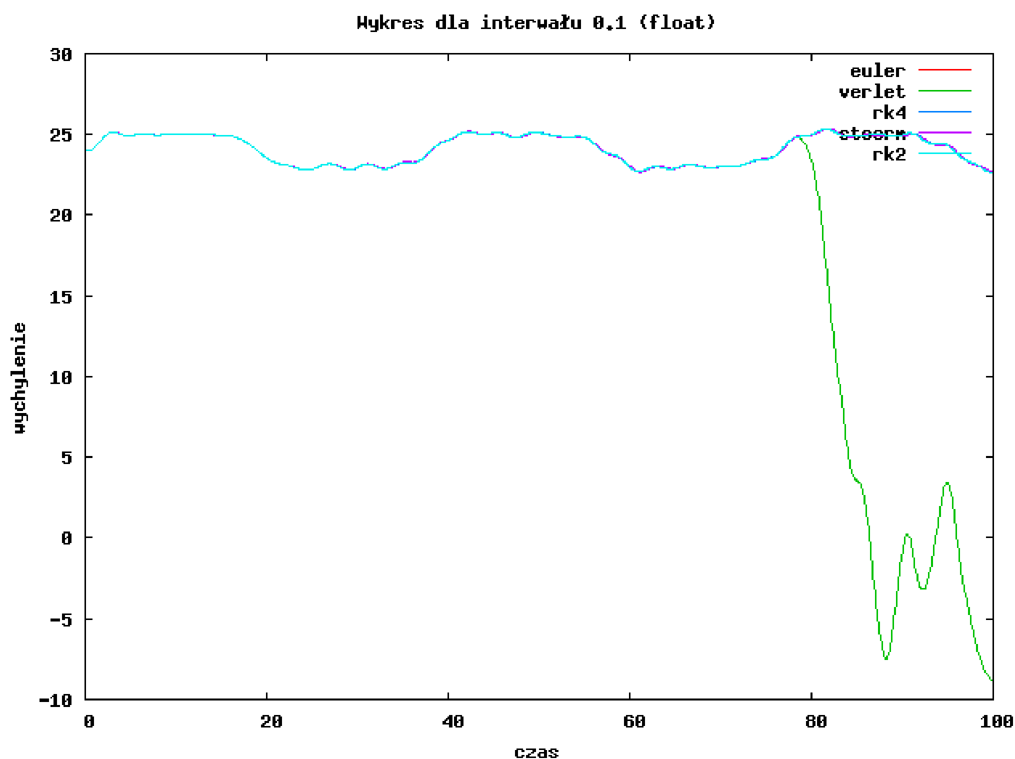
Dalsze testy zbieżności polegały na obserwacji wychylenia cząstek z położenia równowagi w zależności od czasu (w modelu dziesięciu oscylatorów). Dokładność każdej z metod mierzona była dla dwóch typów danych: `float` i `double`, a także dla różnych

interwałów czasowych. Kroki czasowe należały do przedziału od 0.1 do 1.0 i zmieniały się co 0.1. Początkowa prędkość w każdej próbie była taka sama jak w teście dla jednego punktu i wynosiła 1.0.



Wyk. 5. Wykres zależności wychylenia z położenia równowagi dla dziesiątego, ostatniego z oscylujących punktów. `double` obliczenia wykonane w podwójnej precyzji

Dla kroku czasowego 0.1 wszystkie metody zachowywały się niemal identycznie. Jest to widoczne na rysunku 21. Dla typu `float` metoda Verleta wykazuje jednak pewną osobliwość. Po wykonaniu około ośmiuset kroków wyniki zaczęły gwałtownie się rozbiegać. (rys. 22).



Wyk. 6. Nietypowe zachowanie metody Verleta dla danych reprezentowanych za pomocą typu `float`

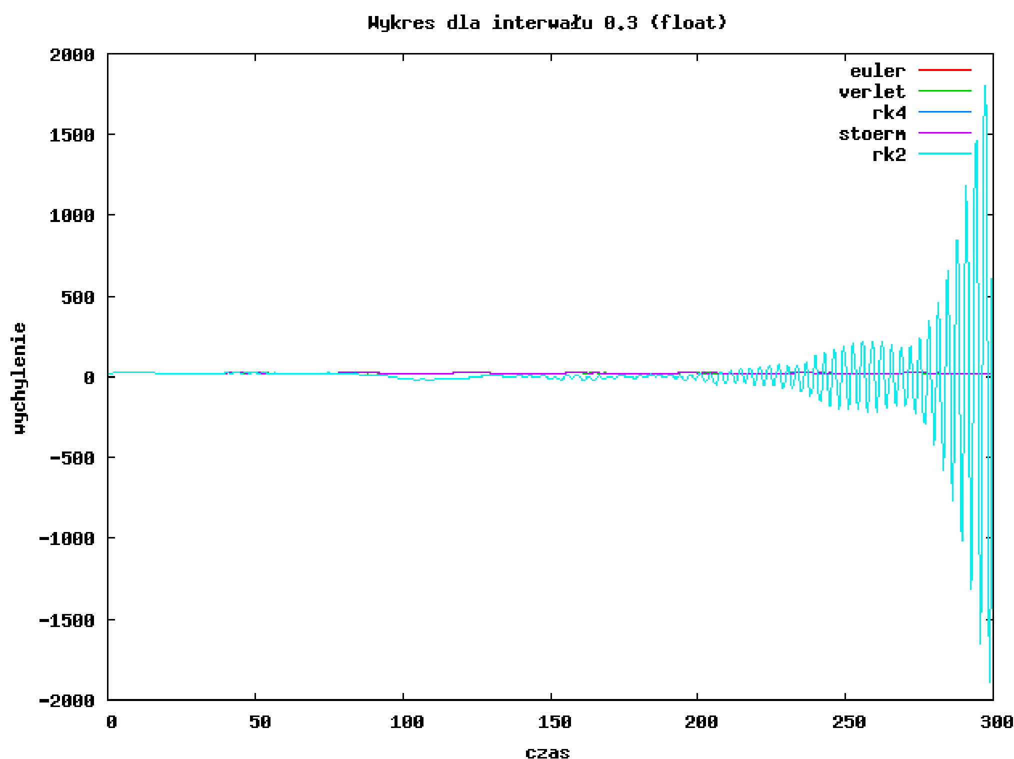
Zjawiska takiego nie zaobserwowano przy obliczeniach prowadzonych w podwójnej precyzji lub dla nieco innych warunków początkowych (inna prędkość początkowa nadawana ostatniej cząstce). Wyniki otrzymane metodą Verleta i Eulera różnią się cyframi na czwartym miejscu po przecinku. Podobną zbieżność wykazują wobec siebie metody Runge-Kutty czwartego rzędu i Stoermera. Trochę większą różnicę zauważyć można porównując metody powyższe z metodą MidPoint. W tym przypadku różnica w stosunku do powyższych algorytmów jest rzędu jednej setnej.

Tab. 1. Położenia dziesiątej cząstki w 680 kroku dla typu `float` i interwału czasowego równego 0.1

Metoda	**Położenie**
Euler	25.7906
Verlet	25.7907
RK4	25.7862
Stoerm	25.7906
RK2	25.8029

Kolejne testy wykonane były dla stopniowo zwiększanego kroku czasowego. Metoda punktu środkowego wykazywała coraz większą rozbieżność. Dla kroku czasowego trzykrotnie większego od pierwszego nastąpiło całkowite załamanie się zbieżności tego

algorytmu (rys. 23). Nie jest on brany pod uwagę przy testach z większymi wartościami kroku czasowego.



Wyk. 7. Rozbieżność metody Runge-Kutty II rzędu przy kroku czasowym wielkości 0.3. Wykres wychylenia w zależności od czasu dla dziesiątego oscylatora

Metody Eulera i Verleta dla większych kroków czasowych zachowują się podobnie jak w pierwszym teście. Wyniki zwracane przez nie różnią się między sobą nieznacznie. Tak jak w powyższym przypadku różnica ta jest zauważalna na czwartym miejscu po przecinku. Zmienia się jednak ogólny charakter działania tych metod w stosunku do pozostałych biorących udział w porównaniu. Znaczący rozłam w symulacji następuje dla kroku 1.0 tj. dziesięciokrotnie większego od kroku podstawowego w tych testach. W tym przypadku dla metody Verleta tak duży krok powoduje rozbieżność na tyle dużą, że położenie punktów staje się większe od zakresu liczb zmiennopozycyjnych (zarówno dla typu `float` jak i `double`). Powoduje to zgłoszenie błędu i zakończenie programu. Co ciekawe, dla tego kroku metoda Eulera nadal działa, choć jej wyniki także są całkowicie niewiarygodne.

Podobnie do powyższych zachowywały się algorytmy Stoermera i RK4. Ich wyniki były nadal bardzo zbliżone do siebie. Wraz ze wzrostem wielkości kroku czasowego rezultaty otrzymywane metodą Stoermera coraz bardziej oscylowały wokół wyników RK4. Począwszy od interwału wielkości 0.6 pierwszy z powyższych algorytmów po dłuższej symulacji zaczynał się rozbiegać. Zwiększanie interwału powodowało zmniejszanie się czasu

poprawnego działania reguły Stoermera. Metoda RK4 dawała wyniki podobne do rzeczywistych nawet przy kroku wielkości 1.4.

Tab. 2. Maksymalny krok dla zagadnienia jednego i dziesięciu drgających punktów materialnych, przy którym dana metoda zachowuje zbieżność

Metoda	**Maksymalny krok**	
	Jeden punkt	**Dziesięć punktów**
Euler	1.8	0.9
MidPoint	0.3	0.2
RK4	1.9	1.4
Stoerm	0.6	0.6
Verlet	1.8	0.9

Tabela 2 pokazuje, dla jakich maksymalnych długości kroku czasowego poszczególne metody zachowują zbieżność. Skomplikowanie zagadnienia skraca tę wielkość dla większości algorytmów. Reguła Stoermera zachowuje się w tym przypadku wyjątkowo. Zarówno dla jednopunktowego jak i układu złożonego z dziesięciu cząstek maksymalny krok czasowy nadal zachowujący zbieżność metody jest taki sam.

4.2. Testy dotyczące czasu trwania obliczeń

Wynik porównania szybkości działania poszczególnych metod można przedstawić w postaci listy ułożonej w kolejności rosnących czasów trwania obliczeń. Ranking algorytmów przedstawia się następująco:

Tab. 3. Czasy trwania obliczeń dla podstawowego kroku czasowego – 0.1

	Metoda	**Czasy trwania obliczeń**	
		`float`	`double`
1	Euler	25.1	31.1
2	Verleta	29.7	31.4
3	MidPoint	60.8	63.8
4	RK4	139.1	157.6
5	Stoerm	321.6	328.5

Czasy podane są w milisekundach. Wartości te zostały otrzymane w wyniku uśrednienia z dziesięciu pomiarów. Każdy pomiar to zmierzony czas trwania obliczeń tysiąca kroków.

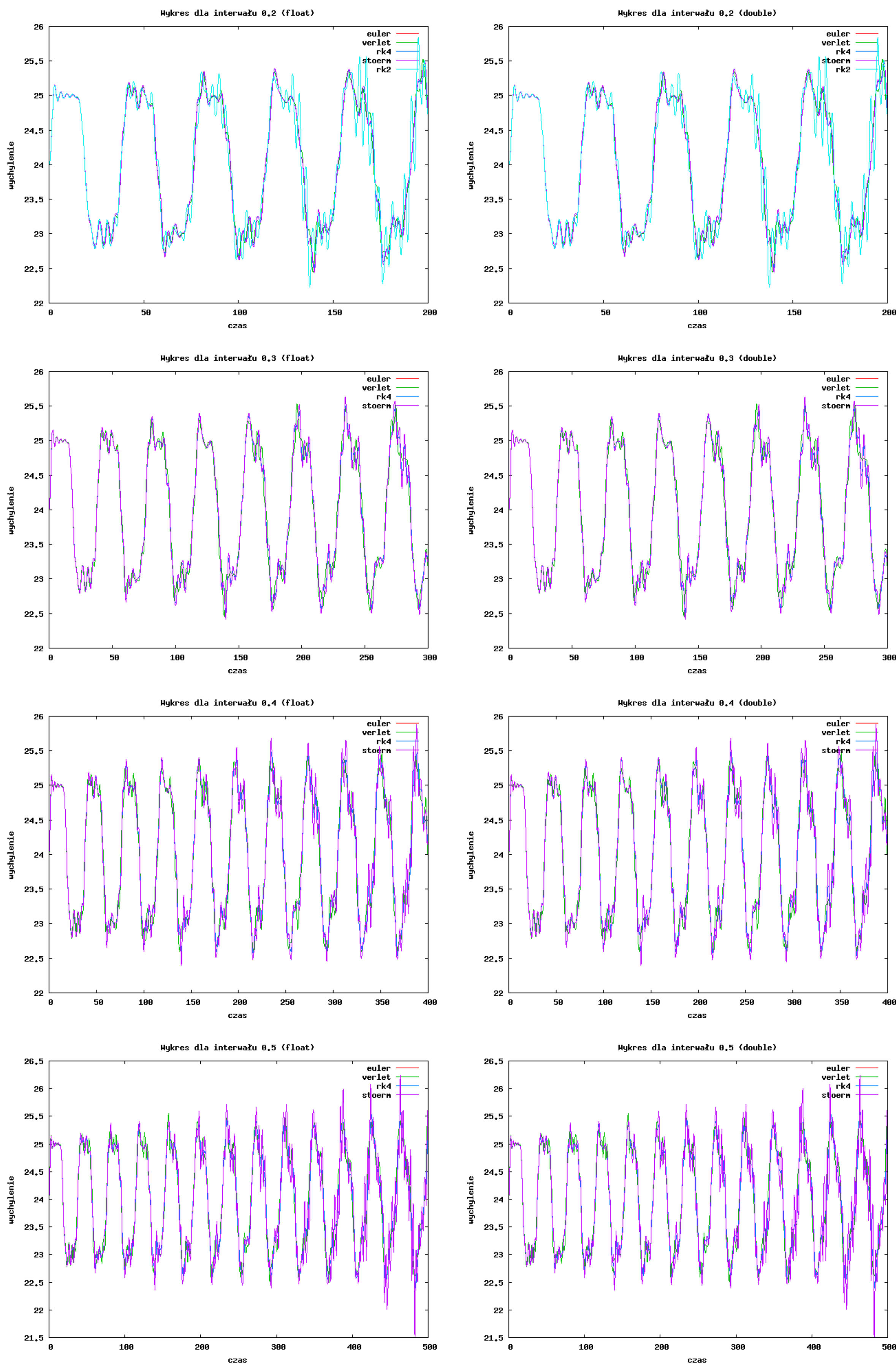
Można zauważyć, że różnice czasów pomiędzy typem `double` a `float` były niewielkie, rzędu kilku milisekund. Reguła Stoermera testowana była w wersji, w której wykonywane było dziesięć uaktualnień wartości drugiej pochodnej funkcji położenia.

Łatwo można zauważyć, że czasy trwania obliczeń są determinowane przez to, ile razy w ciągu kroku czasowego uaktualniana jest wartość drugiej pochodnej (obliczana jest siła wypadkowa działająca na każdą cząstkę w danej chwili). Metody Eulera i Verleta wymagające tylko jednego wyznaczenia sił wypadkowych obliczeń działają najszybciej. Ich szybkości są bardzo zbliżone. Metoda Runge-Kutty II lub inaczej punktu środkowego (MidPoint), obliczająca drugą pochodną dwukrotnie, ma czas dwukrotnie większy od metod z pozycji pierwszej i drugiej. Analogicznie jest w przypadku algorytmów Rungego-Kutty czwartego rzędu (RK4) i Stoermera. Wynik reguły Stoermera w badanej wersji jest dziesięciokrotnie większy od czasu Eulera i Verleta. Metoda ta oblicza drugą pochodną dziesięć razy. RK4 wymaga czterokrotnej aktualizacji wartości sił wypadkowych i zajmuje to czas około czterokrotnie większy od metod najszybszych. Wartość ta jest nieco większa od wyniku pomnożenia czasu Eulera lub Verleta przez liczbę cztery.

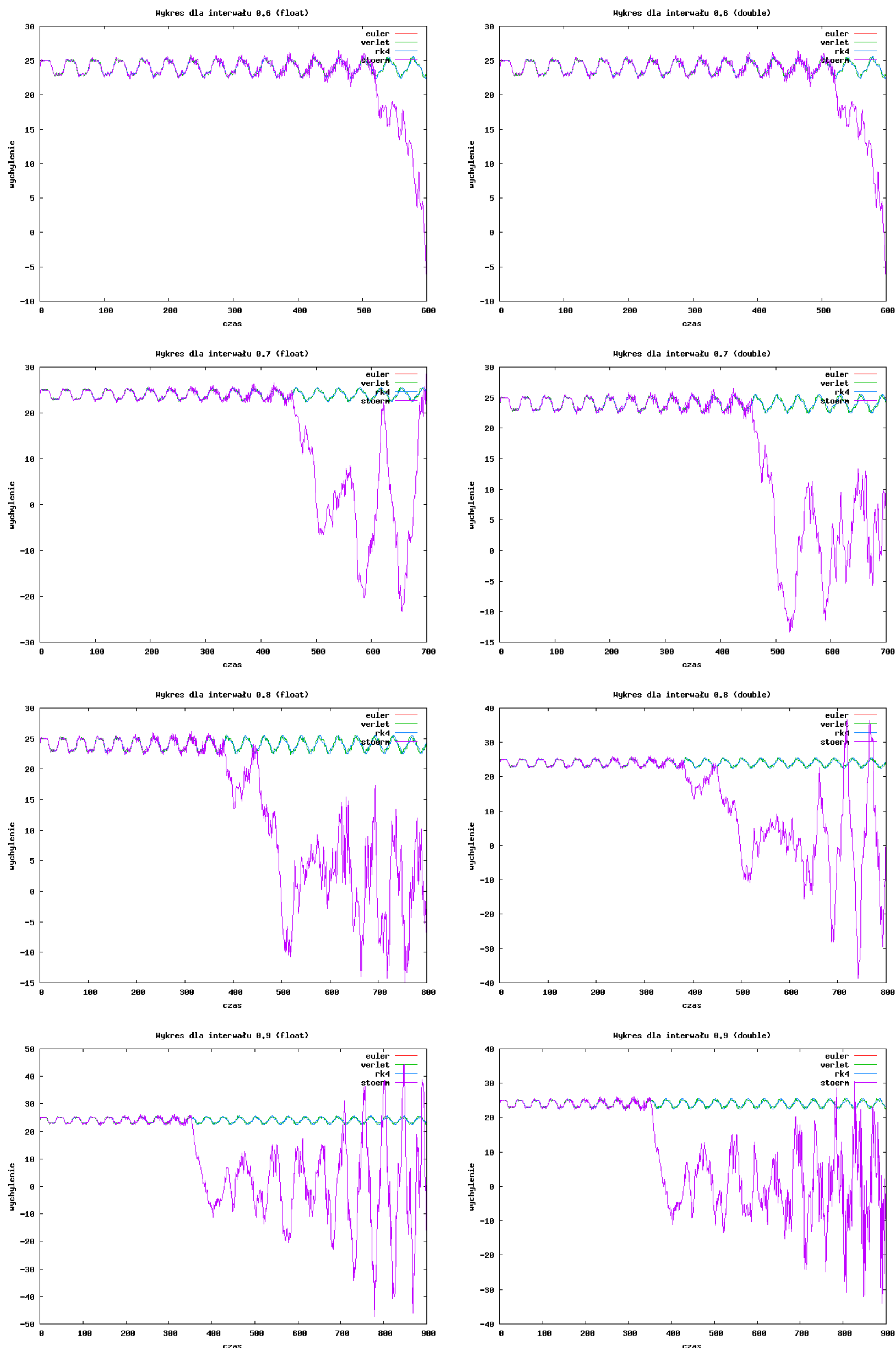
Przy porównywaniu szybkości algorytmów numerycznych ważna jest nie tyle różnica czasu trwania obliczeń z tym samym krokiem czasowym, co czas trwania obliczeń przy maksymalnym kroku czasowym, dla którego dana metoda daje wiarygodne wyniki. W przypadku z podstawowego testu zbieżności, dla interwału czasowego równego 0.1 i 1000 kroków, w rzeczywistości uzyskuje się 100sekund symulacji (jednostka interwału czasowego = sekunda, w tym wypadku 0.1s) . Znając maksymalne długości interwałów, dla których poszczególne metody zachowują zbieżność (tab. 2) można porównać ich efektywne czasy trwania. Zestawienie tych wartości znajduje się w tabeli 4.

Tab. 4. Czasy trwania obliczeń maksymalnego efektywny czas symulacji dla maksymalnego kroku czasowego każdej metody

Metoda		**Czasy trwania obliczeń**		**Efektywny czas symulacji**
		`float`	**`double`**	
1	Euler	25	32.7	900
2	Verleta	31.3	31.4	900
3	MidPoint	62.7	64.1	200
4	RK4	140.6	154.4	1400
5	Storm	311.2	328.2	600



Wyk. 8. Wykresy wychyleń dziesiątej cząstki w funkcji czasu obliczane każdą z metod dla różnych kroków czasowych (od 0.2 do 0.5 od góry) oraz dla różnych typów danych(`float` po lewej, `double` po prawej)



Wyk. 9. Wykresy wychyleń dziesiątej cząstki w funkcji czasu dla interwałów czasowych od 0.6 do 0.9 (od góry), dla typu `float`(po lewej) i `double`(po prawej).

4.3. Podsumowanie

Biorąc pod uwagę zarówno szybkość działania, jak i zbieżność metod można wybrać odpowiednią do rozwiązania zadanego problemu. W zagadnieniu drgających punktów opisanym powyżej najlepiej prezentował się algorytm Runge-Kutty czwartego rzędu. Jest on zbieżny nawet przy dużych krokach czasowych, a przy tym obliczenia tej metody zabierają stosunkowo mało czasu. Jego efektywność jest zatem bardzo duża. Pomimo swej prostoty całkiem dobrze zachowują się metody Eulera i Verleta. Zachowują zbieżność nawet przy dużych krokach czasowych (jednak mniejszych niż w przypadku RK4), a przy tym są bardzo szybkie. Zdecydowanie najgorzej pod wszystkimi względami prezentują się metoda MidPoint i reguła Stoermera. Pierwsza z nich mimo szybkości działania jest rozbieżna już dla niewielkich kroków czasowych. Druga natomiast dla niewielkich interwałów daje dobre wyniki, jednak obliczenia zajmują jej zdecydowanie najwięcej czasu (dwukrotnie więcej niż najwolniejsza z pozostałych metod, a aż dziesięciokrotnie wolniejsza od najszybszej z nich).

5. Podsumowanie

Kluczem do dobrze przeprowadzonej symulacji komputerowej jest właściwe określenie modelu układu reprezentującego badane zagadnienie oraz dobranie odpowiedniej do rozwiązania danego problemu metody numerycznej. W wielu przypadkach, w których nadmierna dokładność nie jest potrzebna, a pożądana jest szybkość działania (dobrym przykładem są gry komputerowe) warto zrezygnować ze skomplikowanego, wprawdzie dokładnego, ale za to bardzo powolnego algorytmu na rzecz prostej i szybkiej, a w nieskomplikowanych układach także dokładnej i stabilnej metody Eulera lub Verleta. Jeśli jednak prezentacja skomplikowanego układu w czasie rzeczywistym z dużą częstotliwością odświeżania animacji nie jest konieczna, a liczy się precyzja obliczeń (tu przykładem są obliczenia dynamiki molekularnej) lepiej skorzystać z metody Rungego-Kutty czwartego rzędu. Warto też zwrócić uwagę na wrażliwość zarówno układu fizycznego, jak i metody numerycznej na warunki brzegowe symulacji. Nie wszystkie algorytmy (a właściwie tylko RK4) nadawały się do przeprowadzenia symulacji liny.

Ważnym aspektem jest wybór typu danych do reprezentacji wielkości opisujących parametry symulowanych układów. Przeprowadzone testy dowodzą, że w przypadku prostych zadań (takich jak przedstawione w pracy) typ `float` ma wystarczającą dokładność, a dodatkowo jest typem używanym przez bibliotekę graficzną (nie jest potrzebne rzutowanie na inny typ danych) prezentującą otrzymane wyniki w łatwy do interpretacji sposób.

6. Literatura

[1] Dawid Halliday, Robert Resnick, Jearl Walter, *Podstawy fizyki tom.1*, Wydawnictwo naukowe PWN, Warszawa 2005.

[2] Dieter W. Heerman, *Podstawy symulacji komputerowych w fizyce*, Wydawnictwa Naukowo-Techniczne 1997.

[3] Jerzy Grębosz, *Symfonia C++ standard*, Wydawnictwo Edition 2000, Kraków 2006.

[4] Jerzy Grębosz, *Pasja C++*, Wydawnictwo Edition 2000, Kraków 2006.

[5] Jacek Matulewski, *C++ Builder 2006. 222 gotowe rozwiązania*, Helion, Gliwice 2006.

[6] Jacek Matulewski, OpenGL(C++ Builder), skrypt, Toruń 2007.

[7] Maciej Matyka, *Symulacje komputerowe w fizyce*, Helion, Gliwice 2002.

[8] Tao Pang, *Metody obliczeniowe w fizyce*, Wydawnictwo naukowe PWN Warszawa 2001.

[9] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes in C*, Cambridge University Press, New York 1995.

[10] Richard S. Wright Jr., Beniamin Lipchak, *OpenGL. Księga eksperta*, Helion, Gliwice 2005.

[11] http://www.fisica.uniud.it/~ercolessi/md/md/node21.html - podstrona strony Wydziału Fizyki Uniwersytetu w Udine poświęcona algorytmowi Verleta

[12] http://www.nehe.gamedev.net – internetowy kurs grafiki OpenGL.

[13] http://www.opengl.org/ - strona internetowa projektu OpenGL

[14] http://aklimx.sppieniezno.pl/nehepl/ - przetłumaczone na język polski lekcje z pozycji [12]

[15] http://www.wikipedia.org – strona wolnej encyklopedii internetowej.

[16] http://cat.middlebury.edu/~chem/chemistry/class/physical/quantum/help/morse/morse.html - strona z informacjami na temat potencjału Morse'a