**Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования**

**"Уфимский университет науки и технологий"**

**Кафедра** Высокопроизводительных вычислительных технологий и систем

**Дисциплина:** Математическое моделирование.

**Отчет по лабораторной работе № 3**

**Тема:** «Исследование динамики одномерной цепочки частиц с различными потенциалами межчастичного взаимодействия»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа ПМ-457 | Фамилия И.О. | Подпись | Дата | Оценка |
| Студент | Акмурзин М.Э. |  |  |  |
| Принял | Лукащук В.О. |  |  |  |

**Уфа 2025**

**Цель работы:** получить навык моделирования динамики системы многих частиц методами молекулярной динамики на примере задачи распространения возмущений в одномерной цепочке частиц одинаковой массы, связанных нелинейным потенциалом взаимодействия Ферми-Паста-Улама.

**Теоретическая часть**

Рассматривается цепочка частиц одинаковой массы, находящаяся в начальный момент времени в состоянии равновесия (расстояния между частицами одинаковы). В качестве граничных условий используется условие периодичности (или условной замкнутости) цепочки. В качестве потенциала меж частичного взаимодействия рассматривается нелинейный степенной потенциал типа Ферми–Паста–Улама (FPU-потенциал)

|  |  |
| --- | --- |
|  | (1) |

При данный потенциал принято называть FPU-α, при – FPU-β.

Уравнение движения *i*-ой частицы будут иметь вид

|  |  |
| --- | --- |
|  | (2) |

Правая часть этого уравнения представляет собой сумму двух сил, действующих на *i*-ю частицу со стороны соседних (*i*–1)-ой и (*i*+1)-ой частиц. Величина – смещение *i*-ой частицы от положения равновесия.

Для интегрирования уравнений движения вида (1) используются два численных алгоритма

*Алгоритм Верле в скоростной форме.* Основные шаги алгоритма на каждом временном шаге:

Здесь и – скорость и ускорение *i*-ой частицы, соответственно. В начальный момент времени значения и известны и по ним рассчитываются значения .

*Симплектический алгоритм типа Верле в скоростной форме.* Основные действия алгоритма на каждом временном шаге:

Здесь параметр определяется из условия минимальности значения коэффициента при в оценке погрешности метода и равен

**Задание**

1. Определить вид правой части уравнения (1) для потенциала (2).
2. На языке программирования Си (Си++) выполнить программную реализацию приведенных численных алгоритмов Верле и симплектического типа Верле.
3. Провести сравнение точности алгоритмов посредством сравнения величины полной энергии системы (гамильтона)

после N временных шагов. Количество частиц n, параметры потенциала взаимодействия, а также значение N задаются преподавателем.

1. Для FPU-α потенциала взаимодействия определить диапазон значений в котором в пространстве скоростей существуют односолитонные решения при условии, что возмущение задаются в центре цепочки в виде приложениях заданных импульсов к двум соседним частицам, равных по величине и противоположных по направлению (так что суммарный импульс системы равен нулю). Выполнить анимационную визуализацию динамики цепочки.
2. Для FPU-β потенциала взаимодействия определить диапазоны значений в которых в пространстве скоростей существуют одно- , двух- и трехсолитонные решения. Возмущения задаются аналогично п. 4). Выполнить анимационную визуализацию цепочки.

**Выполнение работы**

Работа выполнена согласно варианту №2

Начальные условия: при t = 0

Начальное смещение: а = 1, Масса каждой частицы равна 1.

**Уравнение движения.**

Уравнение движения для потенциала будет иметь вид:

|  |
| --- |
|  |

**Сравнение точности алгоритма посредством сравнения величины полной энергии системы (гамильтониана).**

При t = 0:

Начальная скорость всех точек равна нулю, тогда при :

H(0) = 0,0108 +0,000432 + 0,00005832.

Для того, чтобы проверить выполнение закона сохранения энергии для обоих алгоритмов возьмем .

Таблица 1. Отклонения энергии системы от начального значения

|  |  |  |  |
| --- | --- | --- | --- |
| N\_t | Алгоритм Верле в скоростной форме | Симплектический алгоритм типа Верле в скоростной форме | Энергия системы |
|  | 4.84181e-07 | 9.23559e-09 | 0.01129032 |
|  | 5.2204e-07 | 9.71273e-09 | 0.01129032 |
|  | 4.74439e-07 | 1.42734e-07 | 0.01129032 |

Симплектический алгоритм типа Верле в скоростной форме является более точным, чем алгоритм Верле в скоростной форме, однако у него же уходит больше времени на вычисления. Оба алгоритма накапливают погрешность в ходе вычислений.

**Поиск солитонов с использованием симплектического алгоритма типа Верле в скоростной форме.**

Далее найдем одно-, двух- и трехсолитонные решения системы при различных параметрах .

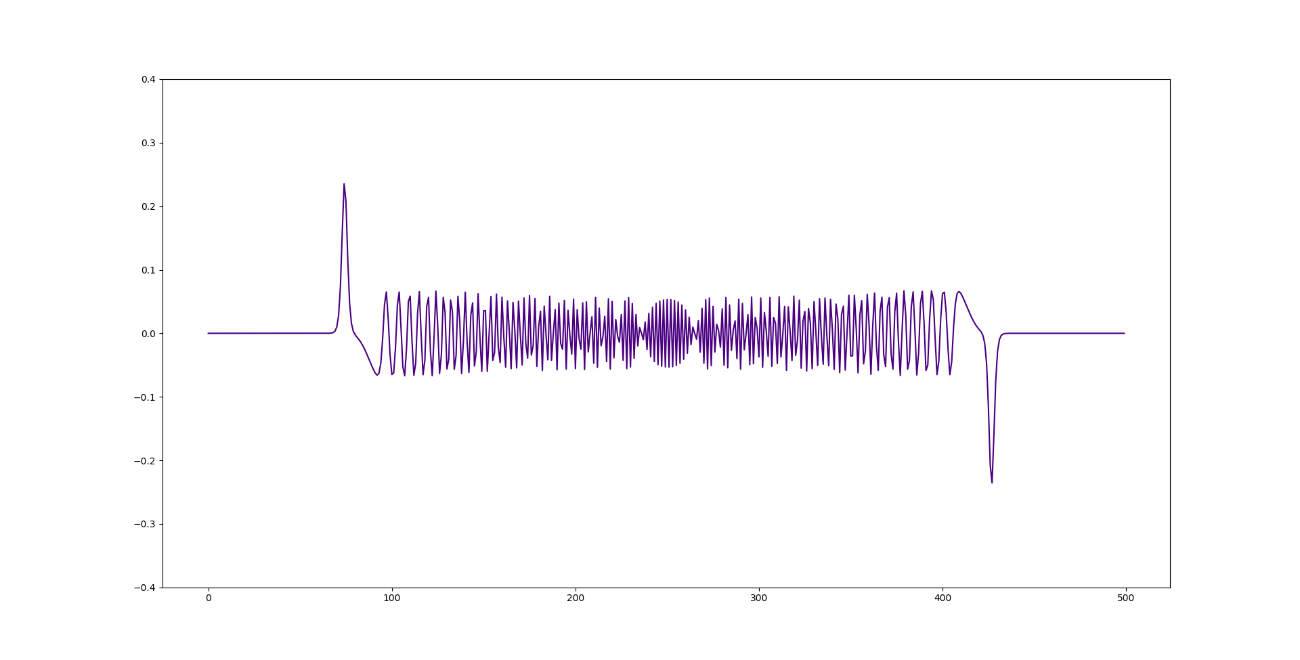


Рисунок 1 – Односолитонное решение при (FPU- потенциал).

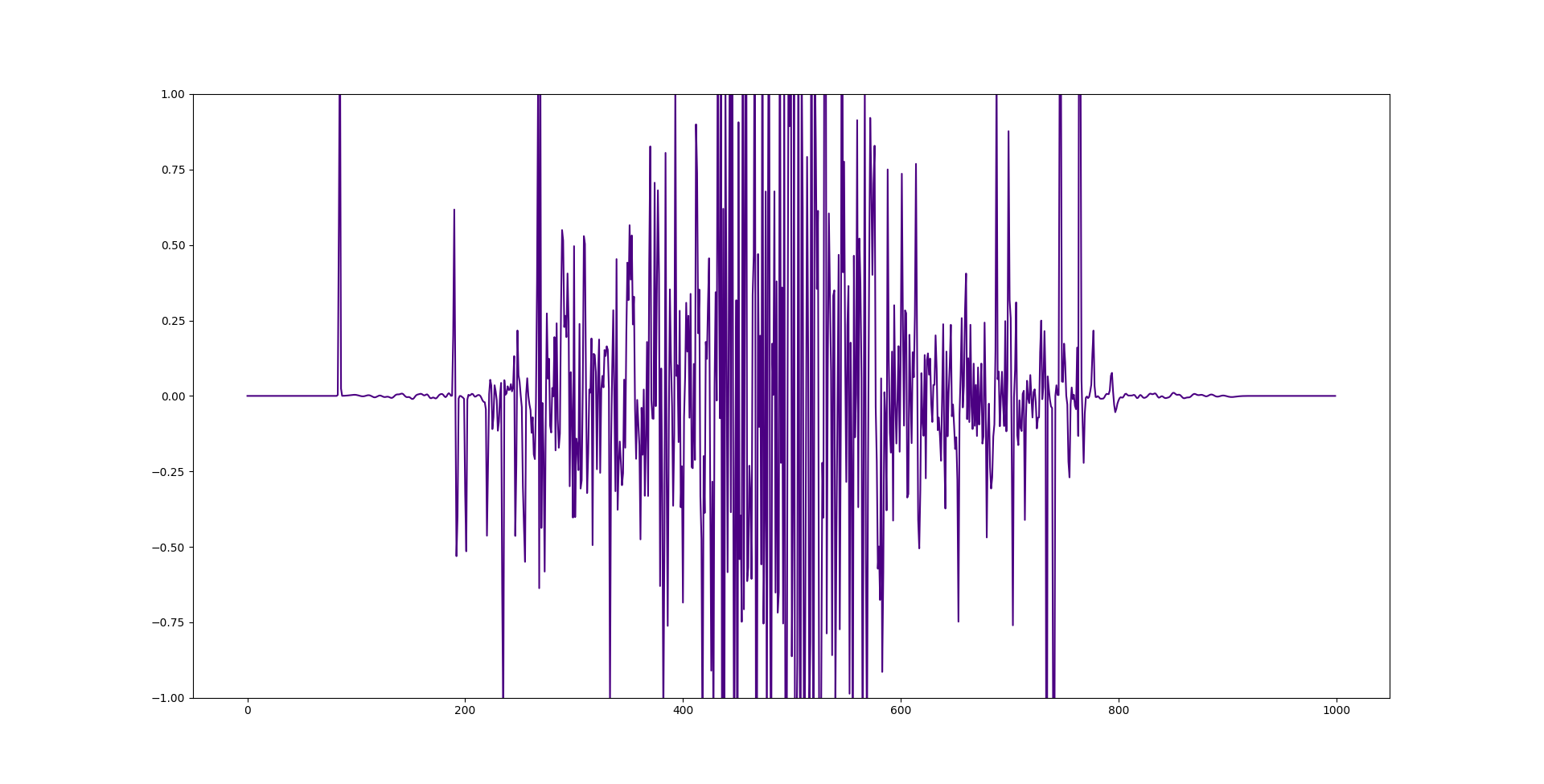


Рисунок 2 – Односолитонное решение при (FPU- потенциал).

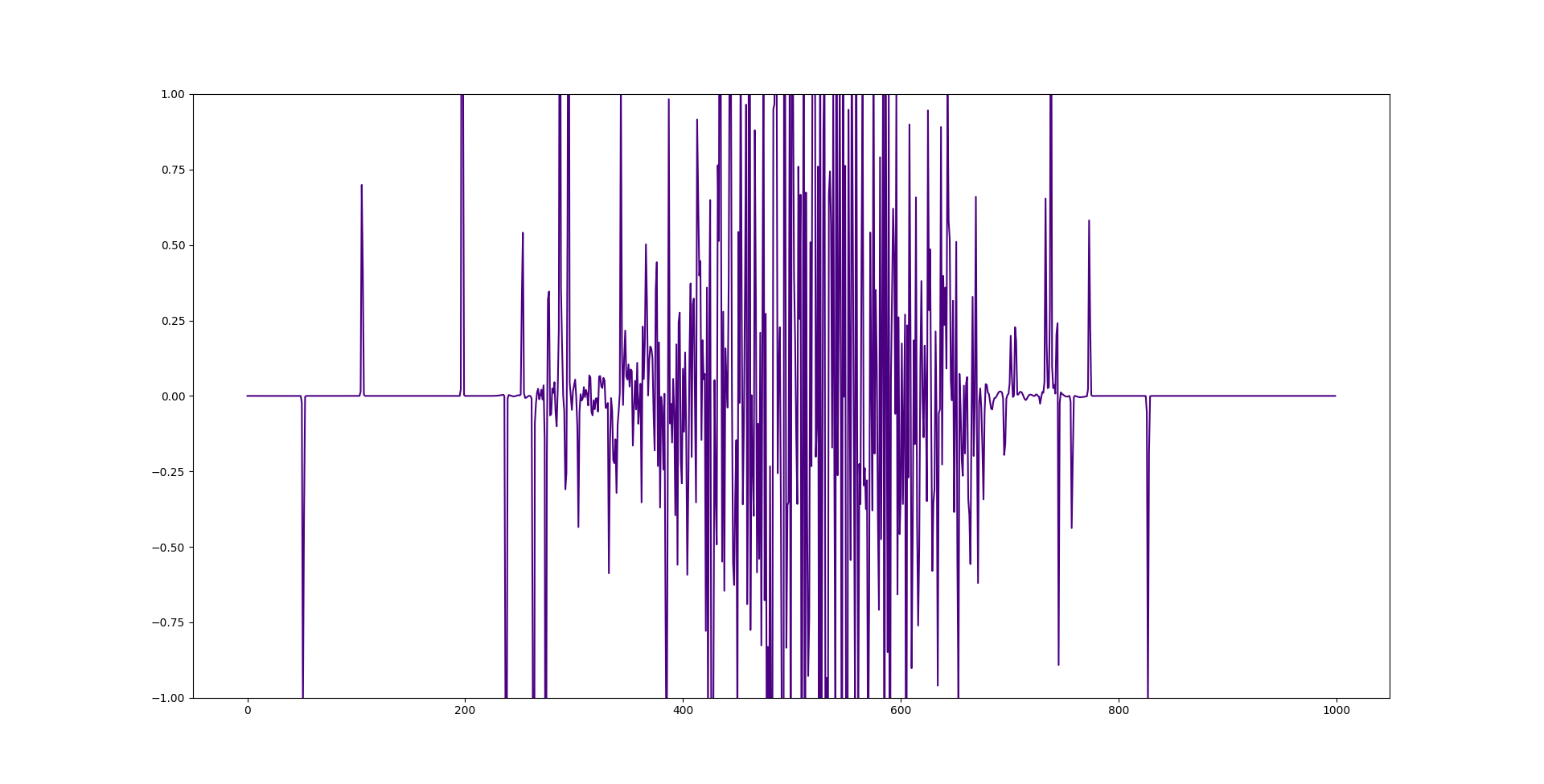


Рисунок 3 – Двухсолитонное решение при (FPU- потенциал).

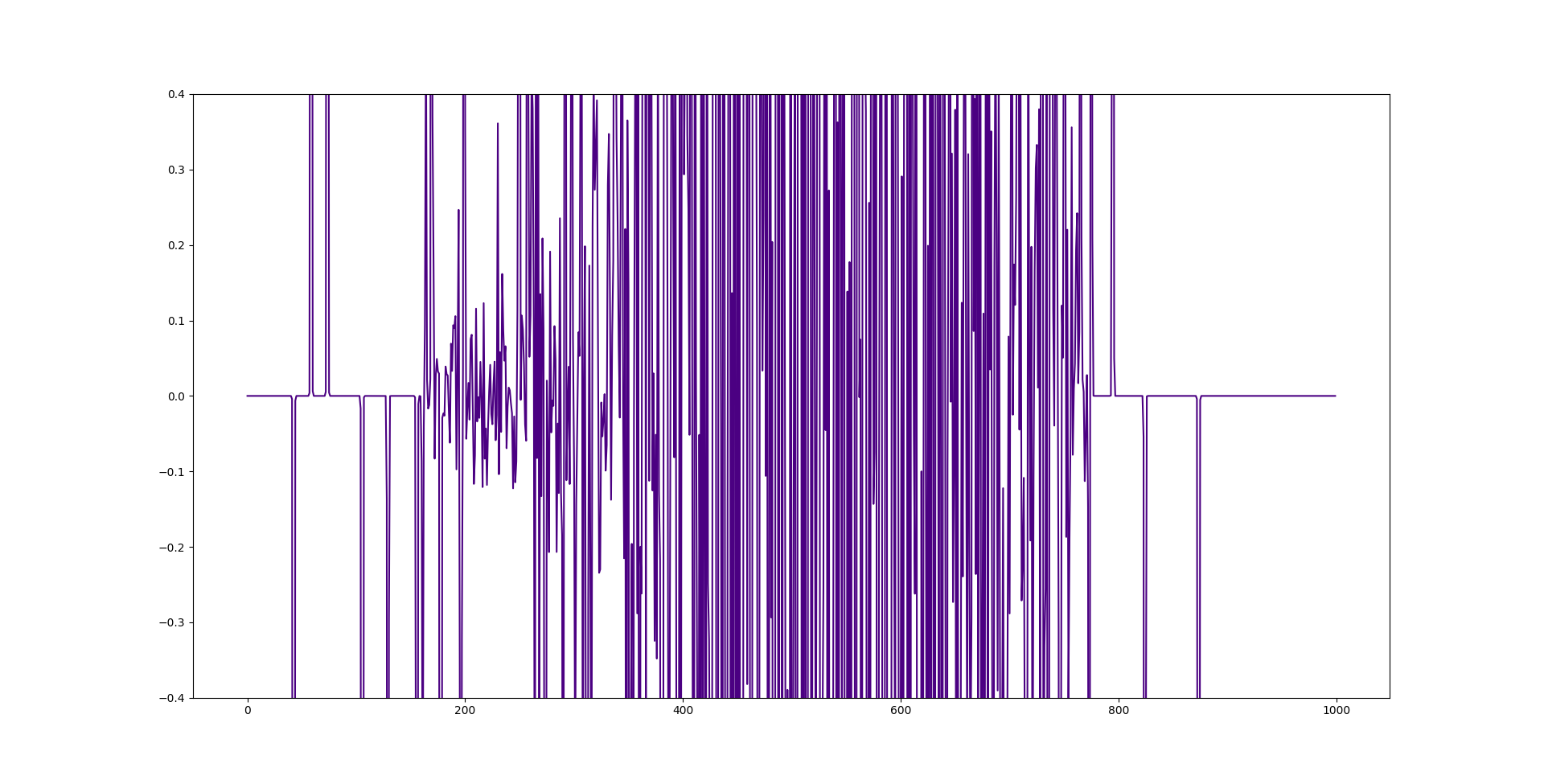


Рисунок 4 – Трехсолитонное решение при (FPU- потенциал).

**Вывод**

В ходе лабораторной работы были получены навыки моделирования динамики системы многих частиц методами молекулярной динамики на примере задачи распространения возмущений в одномерной цепочке частиц одинаковой массы, связанных нелинейным потенциалом взаимодействия типа Ферми-Паста-Улама. Было проведено сравнение алгоритмов интегрирования движения частиц и сделан вывод о точности алгоритмов. Симплектический алгоритм типа Верле в скоростной форме точнее алгоритма Верле в скоростной форме на два порядка, но дольше работает. Визуализирована динамика частиц и найдены такие параметры, при которых возникают одно, двух- и трехсолитонные решения.

**Приложение**

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

// Проверка выполнения закона сохранения энергии

//const int n = 500; // Число частиц

//const int N\_time = 1e6; // Число слоев по времени

//const double alpha = 1, beta = 1.;

//const double tau = 0.01, m = 1.0;

//const double xi = 0.1931833275037836;

//double q0 = 0.06;

// 1 солитон alpha

//const int n = 500; // Число частиц

//const int N\_time = 1e5; // Число слоев по времени

//const double alpha = 0.7, beta = 0.;

//const double tau = 0.01, m = 1.0;

//const double xi = 0.1931833275037836;

//double q0 = 0.5;

// 1 солитон beta

//const int n = 1000; // Число частиц

//const int N\_time = 1e5; // Число слоев по времени

//const double alpha = 0., beta = 50;

//const double tau = 0.01, m = 1.0;

//const double xi = 0.1931833275037836;

//double q0 = 0.9;

// 2 солитона beta

//const int n = 1000; // Число частиц

//const int N\_time = 1e5; // Число слоев по времени

//const double alpha = 0., beta = 64;

//const double tau = 0.01, m = 1.0;

//const double xi = 0.1931833275037836;

//double q0 = 0.9;

// 3 солитона beta

//const int n = 1000; // Число частиц

//const int N\_time = 1e5; // Число слоев по времени

//const double alpha = 0., beta = 200;

//const double tau = 0.01, m = 1.0;

//const double xi = 0.1931833275037836;

//double q0 = 0.9;

std::vector<double> GradV(std::vector<double>& q)

{

std::vector<double> GradV\_vector(q.size());

for (int i = 1; i < n - 1; i++)

{

GradV\_vector[i] = (q[i + 1] - 2 \* q[i] + q[i - 1]) +

alpha \* (pow(q[i + 1] - q[i], 2) - pow(q[i] - q[i - 1], 2)) +

beta \* (pow(q[i + 1] - q[i], 3) - pow(q[i] - q[i - 1], 3));

}

return GradV\_vector;

}

void Verle(std::vector<double>& q, std::vector<double>& v, std::vector<double>& a)

{

std::ofstream f\_out;

f\_out.open("data//Verle.dat");

clock\_t t\_solv\_end = clock();

for (int i = 0; i < N\_time; i++)

{

clock\_t t = clock();

if (i % 100 == 0)

{

for (int j = 0; j < n; j++)

{

f\_out << v[j] << " ";

}

f\_out << "\n";

}

for (int j = 1; j < n - 1; j++)

{

q[j] = q[j] + v[j] \* tau + 0.5 \* a[j] \* tau \* tau;

v[j] = v[j] + 0.5 \* a[j] \* tau;

}

q[0] = q[n - 2];

v[0] = v[n - 2];

q[n - 1] = q[1];

v[n - 1] = v[1];

std::vector<double> GradV\_vector = GradV(q);

for (int j = 1; j < n - 1; j++)

{

a[j] = 1. / m \* GradV\_vector[j];

v[j] = v[j] + 0.5 \* a[j] \* tau;

}

t = clock() - t;

#ifndef NDEBUG

std::cout << "Time for one iteration" << t << " clicks (" << ((double)t) / CLOCKS\_PER\_SEC << " seconds).\n";

std::cout << "End calculation for t = " << (i \* 100) / (N\_time) << "%\n";

#endif // NDEBUG

if (i % 100 == 0) std::cout << "End calculation Verle for t = " << (i \* 100) / (N\_time) << "%\n";

}

t\_solv\_end = clock() - t\_solv\_end;

std::cout << "Time Solve Verle: " << t\_solv\_end << " clicks (" << ((double)t\_solv\_end) / CLOCKS\_PER\_SEC << " seconds).\n";

for (int j = 0; j < n; j++)

{

f\_out << v[j] << " ";

}

f\_out << "\n";

f\_out.close();

}

void simplexVerle(std::vector<double>& q, std::vector<double>& v, std::vector<double>& a)

{

std::ofstream f\_out;

f\_out.open("data//simplexVerle.dat");

clock\_t t\_solv\_end = clock();

for (int i = 0; i < N\_time; i++)

{

for (int j = 0; j < n; j++)

{

f\_out << v[j] << " ";

}

f\_out << "\n";

clock\_t t = clock();

for (int j = 0; j < n; j++)

{

q[j] = q[j] + v[j] \* xi \* tau;

}

std::vector<double> GradV\_vector = GradV(q);

for (int j = 1; j < n - 1; j++)

{

a[j] = 1. / m \* GradV\_vector[j];

v[j] = v[j] + 0.5 \* a[j] \* tau;

}

for (int j = 0; j < n; j++)

{

q[j] = q[j] + v[j] \* (1. - 2. \* xi) \* tau;

}

GradV\_vector = GradV(q);

for (int j = 1; j < n - 1; j++)

{

a[j] = 1. / m \* GradV\_vector[j];

v[j] = v[j] + 0.5 \* a[j] \* tau;

q[j] = q[j] + v[j] \* xi \* tau;

}

q[0] = q[n - 2];

v[0] = v[n - 2];

q[n - 1] = q[1];

v[n - 1] = v[1];

t = clock() - t;

#ifndef NDEBUG

std::cout << "Time for one iteration" << t << " clicks (" << ((double)t) / CLOCKS\_PER\_SEC << " seconds).\n";

std::cout << "End calculation for t = " << (i \* 100) / (N\_time) << "%\n";

#endif // NDEBUG

if (i % 100 == 0) std::cout << "End calculation simplexVerle for t = " << (i \* 100) / (N\_time) << "%\n";

}

t\_solv\_end = clock() - t\_solv\_end;

std::cout << "Time Solve simplexVerle: " << t\_solv\_end << " clicks (" << ((double)t\_solv\_end) / CLOCKS\_PER\_SEC << " seconds).\n";

for (int j = 0; j < n; j++)

{

f\_out << v[j] << " ";

}

f\_out << "\n";

f\_out.close();

}

double V(double r)

{

return (r \* r / 2. + alpha \* r \* r \* r / 3. + beta \* r \* r \* r \* r / 4.);

}

double Gamiltonian(std::vector<double>& v, std::vector<double>& q)

{

double P = 0.0;

// Левая точка выколота

for (int i = 1; i < n - 1; i++)

{

P += m \* v[i] \* v[i] / 2. + V(q[i + 1] - q[i]);

}

return P;

}

int main()

{

std::vector<double> q(n, 0), v(n, 0), a(n, 0);

std::vector<double> GradV\_vector;

// Начальные условия

q[n / 2 + 1] = q0;

q[n / 2] = -q0;

//v[n / 2] = 1.;

//v[n / 2 + 1] = -v0;

GradV\_vector = GradV(q);

for (int j = 1; j < n - 1; j++)

{

a[j] = 1. / m \* GradV\_vector[j];

}

double start = Gamiltonian(v, q);

std::cout << "first H = " << start << "\n";

Verle(q, v, a);

double end = Gamiltonian(v, q);

std::cout << "after Verle H = " << end << "\n";

std::cout << "Verle уrror: " << abs(start - end) << "\n";

////Начальные условия

//for (int i = 0; i < n; i++)

//{

// q[i] = v[i] = a[i] = 0.0;

//}

//q[n / 2 + 1] = q0;

//q[n / 2] = -q0;

//GradV\_vector = GradV(q);

//for (int j = 1; j < n - 1; j++)

//{

// a[j] = 1. / m \* GradV\_vector[j];

//}

//double start = Gamiltonian(v, q);

//std::cout << "first H = " << start << "\n";

//simplexVerle(q, v, a);

//double end = Gamiltonian(v, q);

//std::cout << "after simplexVerle H = " << end << "\n";

//std::cout << "simplexVerle уrror: " << abs(start - end) << "\n";

return 0;

}

**graph.py**

from matplotlib import pyplot as plt

import matplotlib.animation as animation

import numpy as np

speed = 1

PATH = ["D:\\Study\StudMaterials\\Математическое моделирование\\7 семестр\\Лабораторная работа №3\\data\\Verle.dat",

        "D:\\Study\StudMaterials\\Математическое моделирование\\7 семестр\\Лабораторная работа №3\\data\\simplexVerle.dat"]

velocity = np.loadtxt(PATH[0], unpack=False, dtype=float)

def animate(i):

    ax.clear()

    ax.set\_ylim(-3, 3)

    ax.plot(velocity[i \* speed % len(velocity)], color='#4B0082')

fig, ax = plt.subplots()

print(np.min(velocity)\*1.05, np.max(velocity)\*1.05)

ani = animation.FuncAnimation(fig, animate, interval=10)

plt.show()