ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

**Московский институт электроники и математики им. А.Н. Тихонова**

Кащенко Станислав Дмитриевич

**Численное решение дифференциальных уравнений для колебаний нелинейного маятника при помощи метода Эйлера**

Программа для численного моделирования эволюции маятника.

Данная программа визуализирует фазовый портрет и эволюции во времени колебательного математического маятника.

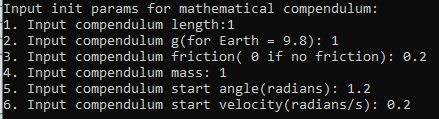


Рис. 1 – Ввод начальных условий через консоль

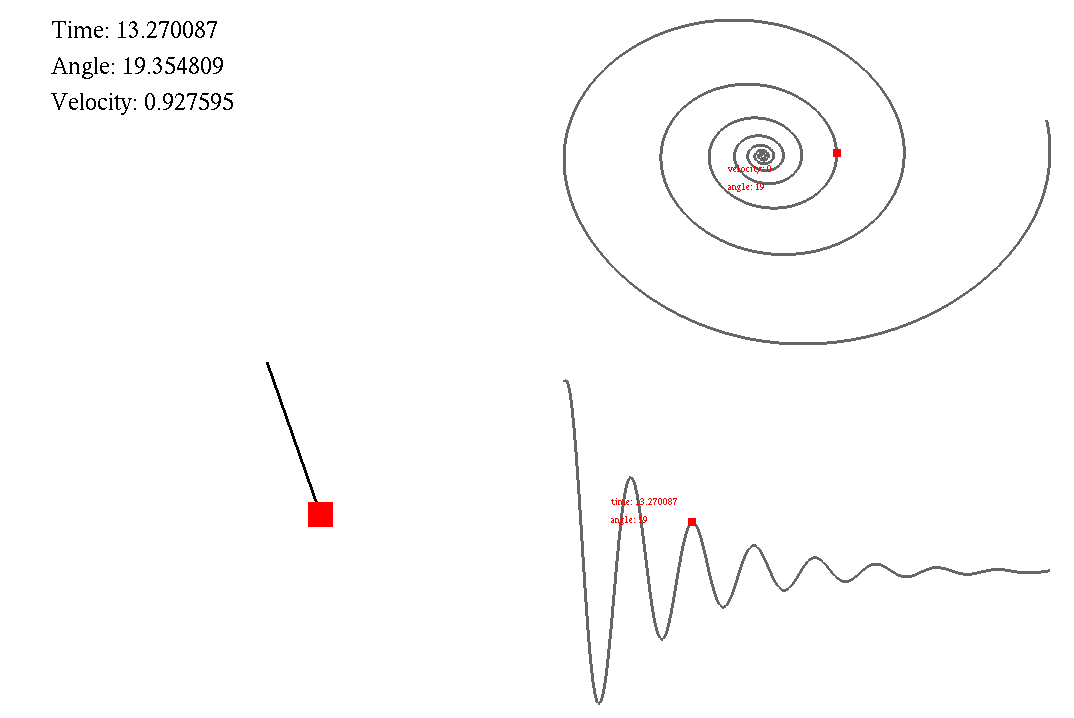


Рис. 2 – Пример работы программы

Данная программа решает уравнение , которое является нелинейным и имеет сложное аналитическое решение. Обычно такие уравнения проще решить численно, когда при более простых приближениях используют линейное уравнение вида . В данном уравнении *x* – угол отклонения от состояния покоя маятника, – круговая частота колебаний. В дальнейшем уравнение было дополнено еще одним членом, отвечающий за диссипацию энергии в системе.

Используемые внешние графические библиотеки.

Данная программа использует OpenGL legacy и библиотеку GLUT для расширения возможностей работы с текстом.

OpenGL (Open Graphics Library) — спецификация, определяющая платформонезависимый (независимый от языка программирования) программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику.

OpenGL Utility Toolkit (GLUT) — библиотека утилит для приложений под OpenGL, которая в основном отвечает за системный уровень операций ввода-вывода при работе с операционной системой. Из функций можно привести следующие: создание окна, управление окном, мониторинг за вводом с клавиатуры и событий мыши. Она также включает функции для рисования ряда геометрических примитивов: куб, сфера, чайник. GLUT даже включает возможность создания несложных всплывающих меню.

Реализация

За основу реализации взят численный метод Эйлера для решения дифференциальных уравнений второго порядка.

Структура, отвечающая за стартовые условия маятника.

struct compendulum {

double L; - длина маятника

double g; - ускорение свободного падения

double angle0; - стартовый угол

double velocity; - стартовая скорость

double m; - масса маятника

double friction; - коэффициент трения

};

double \*\*calculate(compendulum cmp)– вычисление траектории маятника с помощью численного метода.

По вычисленному массиву точек: время, угол, скорость, - производится визуализация графиков и упрощенной модели маятника.

void make\_phaseGraph(double \*\*calcs, int time, double spacing) – отрисовка фазового портрета.

void make\_timeGraph(double \*\*calcs, int time, double spacing) – отрисовка графика угла от времени.

void draw\_cmp(double \*\*calcs, compendulum cmp, int time, double spacing) – отрисовка маятника.

Для отрисовки текста была использована библиотека GLUT.

void outText(float x, float y, string str) – за основу взята функция с параметрами:

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_10, str[i]);

void outText24(float x, float y, string str) - за основу взята функция с параметрами:

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_24, str[i]);

Другие вспомогательные функции:

outInfoCmp(double \*\*calcs, int time) - отрисовка информации о текущем угле, скорости и времени.

double get\_maxPhi(double \*\*calc); - нахождение максимального угла

double get\_maxPPhi(double \*\*calc); - нахождение максимальной скорости

double get\_minPhi(double \*\*calc); - нахождение минимального угла

double get\_minPPhi(double \*\*calc); - нахождение минимальной скорости

double get\_maxTime(double \*\*calcs); - нахождение максимального времени

double to\_rad(double angle); - перевод градусов в радианы

Листинг программы:

#include <GLFW/glfw3.h>

#include <iostream>

#include <math.h>

#include <glut.h>

#include <string>

#define PI 3.141592653589

#define N 1000

using namespace std;

struct compendulum {

double L;

double g;

double angle0;

double velocity;

double m;

double friction;

};

double \*\*calculate(compendulum cmp);

double get\_maxPhi(double \*\*calc);

double get\_maxPPhi(double \*\*calc);

double get\_minPhi(double \*\*calc);

double get\_minPPhi(double \*\*calc);

double get\_maxTime(double \*\*calcs);

double to\_rad(double angle);

double round\_up(double value, int decimal\_places);

void make\_phaseGraph(double \*\*calcs, int time, double spacing);

void make\_timeGraph(double \*\*calcs, int time, double spacing);

void draw\_cmp(double \*\*calcs, compendulum cmp, int time, double spacing);

double accuracy\_period(double \*\*calcs, compendulum cmp);

void outText(float x, float y, string string) {

//const char \*c;

glRasterPos2f(x, y);

for (int i = 0; i < string.length(); i++) {

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_10, string[i]);

}

}

void outText24(float x, float y, string string) {

//const char \*c;

glRasterPos2f(x, y);

for (int i = 0; i < string.length(); i++) {

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_24, string[i]);

}

}

void outInfoCmp(double \*\*calcs, int time) {

double x = -0.9;

double y = 0.9;

glColor3d(0.0, 0.0, 0.0);

outText24(x, y, "Time: " + to\_string(calcs[time][0]));

outText24(x, y - 0.1, "Angle: " + to\_string(calcs[time][1] \* 180 / PI));

outText24(x, y - 0.2, "Velocity: " + to\_string(calcs[time][2] \* 180 / PI));

}

int main(void)

{

compendulum my\_cmp;

cout << "Input init params for mathematical compendulum:\n1. Input compendulum length:";

cin >> my\_cmp.L;

cout << "2. Input compendulum g(for Earth = 9.8): ";

cin >> my\_cmp.g;

cout << "3. Input compendulum friction( 0 if no friction): ";

cin >> my\_cmp.friction;

cout << "4. Input compendulum mass: ";

cin >> my\_cmp.m;

cout << "5. Input compendulum start angle(radians): ";

cin >> my\_cmp.angle0;

cout << "6. Input compendulum start velocity(radians/s): ";

cin >> my\_cmp.velocity;

/\*my\_cmp.L = 1;

my\_cmp.g = 9.8;

my\_cmp.friction = 0.5;

my\_cmp.m = 1;

my\_cmp.angle0 = 1.01\*PI;

my\_cmp.velocity = 0.5;\*/

double \*\*calcs = calculate(my\_cmp);

cout << endl << accuracy\_period(calcs, my\_cmp)<<endl;

GLFWwindow\* window;

if (!glfwInit())

return -1;

window = glfwCreateWindow(1080, 720, "Hello World", NULL, NULL);

glfwSetWindowSizeLimits(window, 1080, 720, 1080, 720);

if (!window)

{

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);

int time = 0;

double spacing = 0.9;

while (!glfwWindowShouldClose(window))

{

/\* Render here \*/

glClearColor(1.0, 1.0, 1.0, 0.0);

glClear(GL\_COLOR\_BUFFER\_BIT);

make\_phaseGraph(calcs, time, spacing);

make\_timeGraph(calcs, time, spacing);

draw\_cmp(calcs, my\_cmp, time, 0.6);

outInfoCmp(calcs, time);

time++;

time = time % N;

/\* Swap front and back buffers \*/

glfwSwapBuffers(window);

/\* Poll for and process events \*/

glfwPollEvents();

}

glfwTerminate();

return 0;

}

double \*\*calculate(compendulum cmp) {

double T = 2 \* PI\*sqrt(cmp.L / cmp.g);

double dt = (double)T / N \* 8;

double \*\*calculations = new double\*[N];

for (int i = 0; i < N; i++) {

calculations[i] = new double[3];

}

calculations[0][0] = 0; // time column

calculations[0][1] = cmp.angle0; // angle column

calculations[0][2] = cmp.velocity; // deriv angle column

for (int i = 1; i < N; i++) {

calculations[i][0] = calculations[i - 1][0] + dt;

calculations[i][2] = calculations[i - 1][2] - (cmp.g / cmp.L \* sin(calculations[i - 1][1]) + cmp.friction / cmp.m \*calculations[i - 1][2])\* dt; // ???

calculations[i][1] = calculations[i][2] \* dt + calculations[i - 1][1];

}

return calculations;

}

double accuracy\_period(double \*\*calcs, compendulum cmp) {

double T\_analytics = 2 \* PI\*sqrt(cmp.L / cmp.g);

double T\_calculated = 0;

double time1\_max = 0, time2\_max = 0;

int i = 1;

while ( (time2\_max == 0) ) {

if (calcs[i - 1][1]<calcs[i][1] && calcs[i][1]>calcs[i + 1][1]) {

if (time1\_max == 0) {

time1\_max = calcs[i][0];

}

else {

time2\_max = calcs[i][0];

}

}

i++;

}

return T\_analytics - (time2\_max - time1\_max);

}

void make\_phaseGraph(double \*\*calcs, int time, double spacing) {

double max\_phi = get\_maxPhi(calcs);

double max\_p\_phi = get\_maxPPhi(calcs);

double min\_phi = get\_minPhi(calcs);

double min\_p\_phi = get\_minPPhi(calcs);

double x = 0;

double y = 0;

double x\_factor = (1 - 0) / (max\_phi - min\_phi)\*spacing;

double y\_factor = (1 - 0) / (max\_p\_phi - min\_p\_phi)\*spacing;

double x\_transfer = -(max\_phi + min\_phi) / 2 \* x\_factor + 0.5;

double y\_transfer = -(max\_p\_phi + min\_p\_phi) / 2 \* y\_factor + 0.5;

glLineWidth(3);

glBegin(GL\_LINE\_STRIP); {

glColor3d(0.4, 0.4, 0.4);

for (int i = 0; i < N; i++) {

x = calcs[i][1] \* x\_factor + x\_transfer;

y = calcs[i][2] \* y\_factor + y\_transfer;

glVertex2d(x, y);

}

}

glEnd();

glPointSize(8);

glBegin(GL\_POINTS); {

glColor3d(1.0, 0.0, 0.0);

x = calcs[time][1] \* x\_factor + x\_transfer;

y = calcs[time][2] \* y\_factor + y\_transfer;

glVertex2d(x, y);

}

glEnd();

outText(x - 0.2, y - 0.1, "angle: " + to\_string((int)to\_rad(calcs[time][1]) % 360));

outText(x - 0.2, y - 0.05, "velocity: " + to\_string((int)to\_rad(calcs[time][2]) % 360));

}

void make\_timeGraph(double \*\*calcs, int time, double spacing) {

//@todo std::string trimmedString = std::to\_string(doubleVal).substr(0, std::to\_string(doubleVal).find(".") + precisionVal + 1);

double max\_phi = get\_maxPhi(calcs);

double min\_phi = get\_minPhi(calcs);

double max\_time = get\_maxTime(calcs);

double x = 0;

double y = 0;

double x\_factor = (1 - 0) / (max\_time - 0)\*spacing;

double y\_factor = (1 - 0) / (max\_phi - min\_phi)\*spacing;

double x\_transfer = -(max\_time + 0) / 2 \* x\_factor + 0.5;

double y\_transfer = -(max\_phi + min\_phi) / 2 \* y\_factor - 0.5;

glLineWidth(3);

glBegin(GL\_LINE\_STRIP); {

glColor3d(0.4, 0.4, 0.4);

for (int i = 0; i < N; i++) {

x = calcs[i][0] \* x\_factor + x\_transfer;

y = calcs[i][1] \* y\_factor + y\_transfer;

glVertex2d(x, y);

}

}

glEnd();

glPointSize(8);

glBegin(GL\_POINTS); {

glColor3d(1.0, 0.0, 0.0);

x = calcs[time][0] \* x\_factor + x\_transfer;

y = calcs[time][1] \* y\_factor + y\_transfer;

glVertex2d(x, y);

}

glEnd();

outText(x - 0.15, y + 0.05, "time: " + to\_string(calcs[time][0]));

outText(x - 0.15, y, "angle: " + to\_string((int)to\_rad(calcs[time][1]) % 360));

}

double round\_up(double value, int decimal\_places) {

const double multiplier = std::pow(10.0, decimal\_places);

return std::ceil(value \* multiplier) / multiplier;

}

double get\_maxTime(double \*\*calcs) {

return calcs[N - 1][0];

}

double get\_maxPhi(double \*\*calc) {

double max = calc[0][1];

for (int i = 0; i < N; i++) {

if (calc[i][1] > max) {

max = calc[i][1];

}

}

return max;

}

double get\_maxPPhi(double \*\*calc) {

double max = calc[0][2];

for (int i = 0; i < N; i++) {

if (calc[i][2] > max) {

max = calc[i][2];

}

}

return max;

}

double get\_minPhi(double \*\*calc) {

double min = calc[0][1];

for (int i = 0; i < N; i++) {

if (calc[i][1] < min) {

min = calc[i][1];

}

}

return min;

}

double get\_minPPhi(double \*\*calc) {

double min = calc[0][2];

for (int i = 0; i < N; i++) {

if (calc[i][2] < min) {

min = calc[i][2];

}

}

return min;

}

double to\_rad(double angle) {

return angle \* 180 / PI;

}

void draw\_cmp(double \*\*calcs, compendulum cmp, int time, double spacing) {

//@MAKE TESTS and dopisat

double y = 0, x = 0;

double x\_factor = 0.5 \* spacing;

double y\_factor = 0.5 \* spacing \* 1080 / 720;

double x\_transfer = -0.5;

double y\_transfer = 0.0;

glBegin(GL\_LINES); {

glColor3d(0.0, 0.0, 0.0);

x = cmp.L \* sin(calcs[time][1]) / cmp.L \* x\_factor + x\_transfer;

y = -cmp.L \* cos(calcs[time][1]) / cmp.L \* y\_factor + y\_transfer;

//cout << sqrt( pow(x-x\_transfer,2) + pow(y-y\_transfer,2))<<endl;

glVertex2d(x\_transfer, y\_transfer);

glVertex2d(x, y);

}glEnd();

glPointSize(25);

glBegin(GL\_POINTS); {

glColor3d(1.0, 0.0, 0.0);

x = cmp.L \* sin(calcs[time][1]) / cmp.L \* x\_factor + x\_transfer;

y = -cmp.L \* cos(calcs[time][1]) / cmp.L \* y\_factor + y\_transfer;

glVertex2d(x, y);

}

glEnd();

}