**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»**

Інститут **ІКНІ**

Кафедра **ПЗ**



**ЗВІТ**

До лабораторної роботи №2

**На тему: «**Програмування кривої Безьє»

**З дисципліни: «**Комʼютерна графіка»

**Лектор:**

доц. каф. ПЗ

Левус Є.В.

**Виконав:**

ст. гр. ПЗ-24

Войтинський Д.О.

**Прийняв:**

доц. каф. ПЗ

Горечко О.М.

«\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2025р.

∑ = \_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

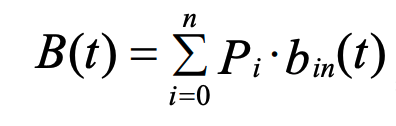
Львів – 2025

**Тема роботи:** Програмування кривої Безьє

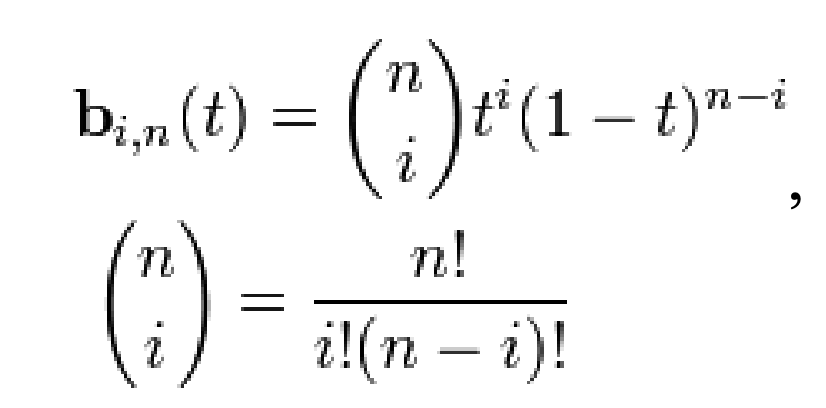
**Мета роботи:** Навчитися програмувати алгоритми побудови кривої Безьє

**Теоретичні відомості**

Крива Безьє - параметрична крива задається виразом:

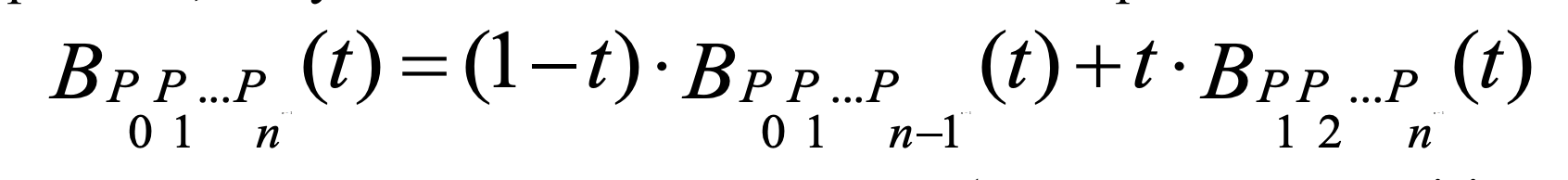


де параметр t [0;1], n-степінь полінома, який характеризується n+1 контрольними точками (вершинами), Рі (хі , уі) – і-та контрольна точка, і - порядковий номер точки (вершини), - базисні функції кривої Безьє, названі також поліномами Бернштейна, які визначаються виразами:

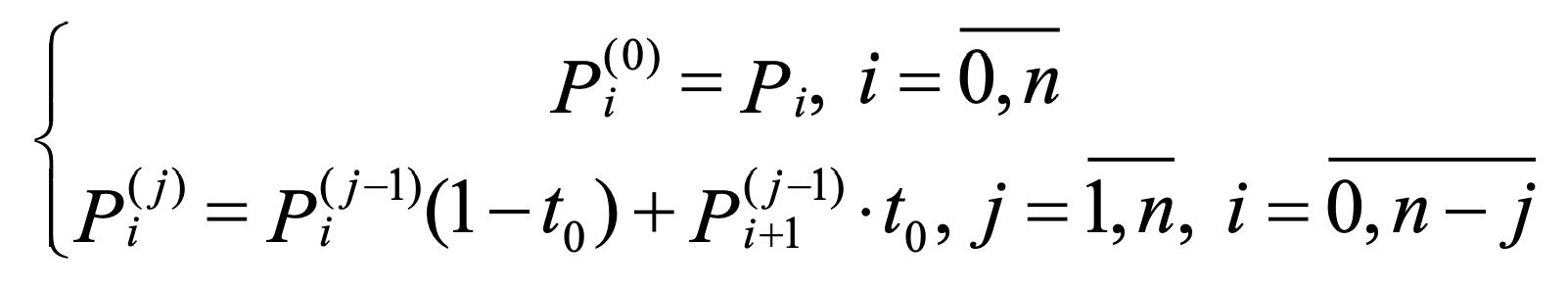


Крім параметричної, використовують рекурсивну і матричну формули для обчислення точок кривої Безьє.

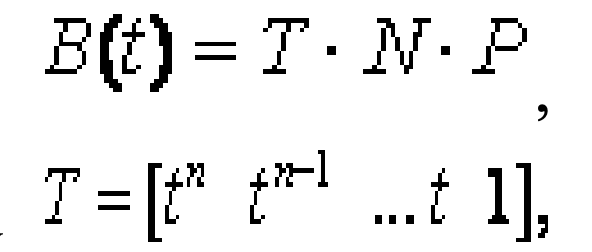
За рекурсивною формулою де Кастельє крива Безьє визначається двома іншими кривими, побудованими на підмножинах контрольних точок:

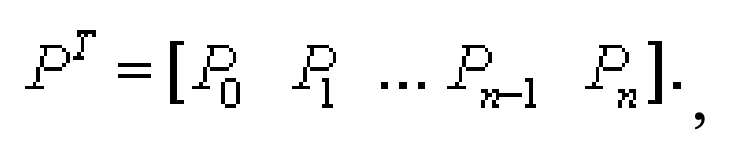


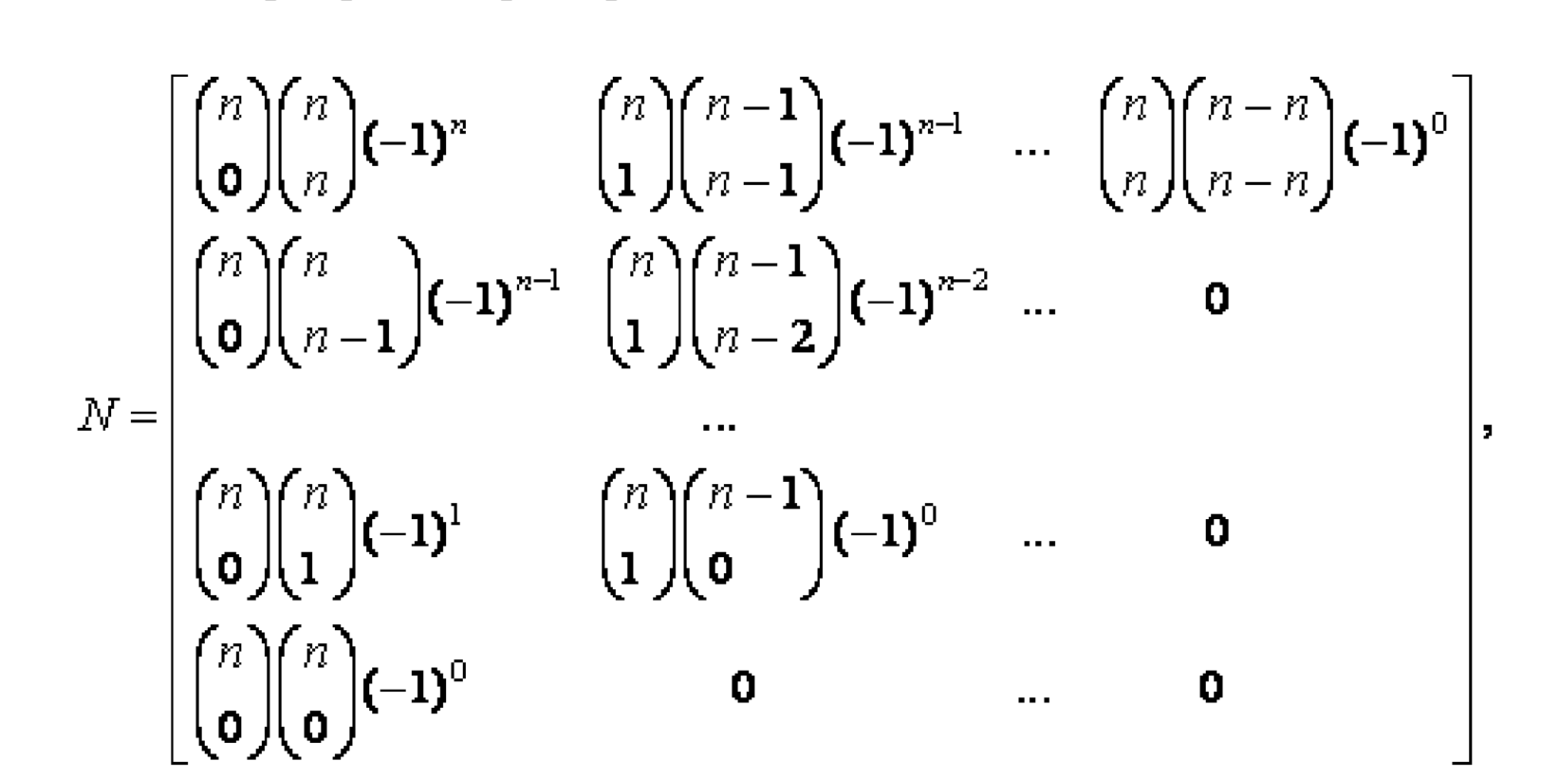
У свою чергу кожна з двох кривих також будується на основі інших двох кривих, побудованих на відповідних підмножинах точок, і т.д. Отримаємо такі рекурсивні формули для знаходження точки кривої Безьє при значенні параметру t0:



Для представлення кривої Безьє використовують вираз:



Р - вектор вершин характеристичної ламаної, 



**Приклад розрахунку точки:**

Для кубічної кривої Безьє використовується наступна формула:

B(t)=(1−t)^3 \* P0​+3t(1−t)^2 \* P1​+3t2(1−t) \* P2​+t3 \* P3​,0<t<1

де:

* P₀, P₁, P₂, P₃ — задані контрольні точки,
* t — параметр, що змінюється від 0 до 1.

Приклад контрольних точок:

* P₀ = (0, 0)
* P₁ = (1, 2)
* P₂ = (3, 3)
* P₃ = (4, 0)

Обчислимо дві точки на кривій для значень t = 0.3 та t = 0.7.

Розрахунок для t = 0.3

1. Обчислення коефіцієнтів:
   * (1 - t)³ = (0.7)³ ≈ 0.343
   * 3t(1 - t)² = 3 × 0.3 × (0.7)² ≈ 0.441
   * 3t²(1 - t) = 3 × (0.3)² × 0.7 ≈ 0.189
   * t³ = (0.3)³ ≈ 0.027
2. Обчислення координат точки B(0.3):
   * x = 0.343·0 + 0.441·1 + 0.189·3 + 0.027·4 ≈ 1.116
   * y = 0.343·0 + 0.441·2 + 0.189·3 + 0.027·0 ≈ 1.449

Отже, точка B(0.3) ≈ (1.116, 1.449).

Розрахунок для t = 0.7

1. Обчислення коефіцієнтів:
   * (1 - t)³ = (0.3)³ ≈ 0.027
   * 3t(1 - t)² = 3 × 0.7 × (0.3)² ≈ 0.189
   * 3t²(1 - t) = 3 × (0.7)² × 0.3 ≈ 0.441
   * t³ = (0.7)³ ≈ 0.343
2. Обчислення координат точки B(0.7):
   * x = 0.027·0 + 0.189·1 + 0.441·3 + 0.343·4 ≈ 2.884
   * y = 0.027·0 + 0.189·2 + 0.441·3 + 0.343·0 ≈ 1.701

Отже, точка B(0.7) ≈ (2.884, 1.701).

**Завдання**

Створити редактор кривої Безьє, який має такий функціонал:

* введення і редагування вершин характеристичної ламаної,
* побудова кривої за параметричною формулою,
* додавання нової точки для характеристичної ламаної,
* контроль коректності введених даних,
* виведення необхідних підказок, повідомлень,
* виконання *індивідуального варіанту.*

**Варіант 2:**

Візуалізувати криву Безьє за матричною формулою; намалювати характеристичну ламану одним кольором, а криву – іншим; обчислити координати точок з кроком на заданому проміжку параметру t (крок, проміжок вводиться користувачем); сформувати у файлі матрицю коефіцієнтів для матричного представлення кривої.

**Код програми**

**scripts.js**

// ============================================

// CANVAS SETUP AND INITIALIZATION

// ============================================

const canvas = document.getElementById('canvas');

const ctx = canvas.getContext('2d');

// Set actual size in memory (scaled to account for extra pixel density)

const dpr = window.devicePixelRatio || 1;

canvas.width = canvas.offsetWidth \* dpr;

canvas.height = canvas.offsetHeight \* dpr;

// Normalize coordinate system to use CSS pixels

ctx.scale(dpr, dpr);

// Global state variables

let scale = 40; // pixels per unit

let offsetX = canvas.width / (2 \* dpr);

let offsetY = canvas.height / (2 \* dpr);

let animationId = null;

let isAnimating = false;

let bezierCurves = [];

let points = [];

let currentBezierCurve = null;

let bezierMode = false;

let activeCurveIndex = -1;

let autoUpdateCurve = true;

// ============================================

// COORDINATE SYSTEM UTILITIES

// ============================================

// Convert graph coordinates to canvas coordinates

function graphToCanvas(x, y) {

return {

x: offsetX + x \* scale,

y: offsetY - y \* scale

};

}

// Convert canvas coordinates to graph coordinates

function canvasToGraph(x, y) {

return {

x: (x - offsetX) / scale,

y: (offsetY - y) / scale

};

}

// Calculate appropriate decimal places based on scale

function calculateDecimalPlaces(scale) {

if (scale >= 100) return 0;

if (scale >= 40) return 1;

if (scale >= 20) return 2;

if (scale >= 10) return 3;

return 4;

}

// Calculate appropriate step for displaying numbers based on scale

function calculateNumberStep(scale) {

if (scale < 5) return 10;

if (scale < 10) return 5;

if (scale < 20) return 2;

return 1;

}

// ============================================

// MATHEMATICAL FUNCTIONS FOR BEZIER CURVES

// ============================================

// Calculate factorial of n

function factorial(n) {

if (n <= 1) return 1;

return n \* factorial(n - 1);

}

// Calculate binomial coefficient (n choose i)

function binomial(n, i) {

return factorial(n) / factorial(i) / factorial(n - i);

}

// Calculate Bernstein polynomial

function BerstainPolynomial(n, i, t) {

return binomial(n, i) \* Math.pow(t, i) \* Math.pow(1 - t, n - i);

}

// Calculate Bezier point using parametric method

function calculateBezierPoint(controlPoints, t) {

const n = controlPoints.length - 1;

let x = 0;

let y = 0;

for (let i = 0; i <= n; i++) {

const b = BerstainPolynomial(n, i, t);

x += controlPoints[i].x \* b;

y += controlPoints[i].y \* b;

}

return { x, y };

}

// Get Bezier coefficient matrix for matrix method

function getBezierMatrix(n) {

const matrix = Array(n + 1).fill().map(() => Array(n + 1).fill(0));

for (let i = 0; i <= n; i++) {

for (let j = 0; j <= i; j++) {

let coefficient = Math.pow(-1, i - j) \* binomial(n, i) \* binomial(i, j);

matrix[j][n - i] = coefficient;

}

}

return matrix;

}

// Matrix multiplication: vector \* matrix

function multiplyVectorMatrix(vector, matrix) {

const result = Array(matrix[0].length).fill(0);

for (let j = 0; j < matrix[0].length; j++) {

for (let i = 0; i < vector.length; i++) {

result[j] += vector[i] \* matrix[i][j];

}

}

return result;

}

// Matrix multiplication: vector \* control points

function multiplyVectorPoints(vector, points) {

let x = 0, y = 0;

for (let i = 0; i < vector.length; i++) {

x += vector[i] \* points[i].x;

y += vector[i] \* points[i].y;

}

return { x, y };

}

// Calculate Bezier point using the matrix method

function calculateBezierPointMatrix(controlPoints, t) {

const n = controlPoints.length - 1;

// Create the parameter vector [t^n, t^(n-1), ..., t, 1]

const paramVector = Array(n + 1).fill(0);

for (let i = 0; i <= n; i++) {

paramVector[i] = Math.pow(t, n - i);

}

// Get the Bezier basis matrix

const bezierMatrix = getBezierMatrix(n);

// Calculate T \* M

const coefficients = multiplyVectorMatrix(paramVector, bezierMatrix);

// Calculate (T \* M) \* P

return multiplyVectorPoints(coefficients, controlPoints);

}

// Calculate Bezier curve points using either parametric or matrix method

function calculateBezierCurve(controlPoints, numPoints, method = 'parametric', tMin = 0, tMax = 1) {

const curve = [];

for (let i = 0; i <= numPoints; i++) {

const t = tMin + (i / numPoints) \* (tMax - tMin);

let point;

if (method === 'matrix') {

console.log('Using matrix method');

point = calculateBezierPointMatrix(controlPoints, t);

} else {

point = calculateBezierPoint(controlPoints, t);

}

curve.push(point);

}

return curve;

}

// ============================================

// DRAWING FUNCTIONS

// ============================================

// Draw arrow head for axes

function drawArrow(x, y, dirX, dirY) {

const arrowSize = 15;

ctx.beginPath();

ctx.moveTo(x, y);

// Calculate the points for the arrow head based on direction

if (Math.abs(dirX) > Math.abs(dirY)) {

// Horizontal arrow (X-axis)

const xOffset = dirX \* arrowSize;

ctx.lineTo(x - xOffset, y - arrowSize/2);

ctx.lineTo(x - xOffset, y + arrowSize/2);

} else {

// Vertical arrow (Y-axis)

const yOffset = dirY \* arrowSize;

ctx.lineTo(x - arrowSize/2, y - yOffset);

ctx.lineTo(x + arrowSize/2, y - yOffset);

}

ctx.closePath();

ctx.fill();

}

// Draw a single Bezier curve with control points and polyline

function drawBezierCurve(curve) {

if (!curve || curve.points.length < 2) return;

const controlPoints = curve.points;

// Draw control point polyline

ctx.beginPath();

ctx.strokeStyle = curve.polylineColor || 'gray';

ctx.lineWidth = 1;

// Convert first point from graph to canvas coordinates

const firstPoint = graphToCanvas(controlPoints[0].x, controlPoints[0].y);

ctx.moveTo(firstPoint.x, firstPoint.y);

// Draw lines connecting control points

for (let i = 1; i < controlPoints.length; i++) {

const canvasPoint = graphToCanvas(controlPoints[i].x, controlPoints[i].y);

ctx.lineTo(canvasPoint.x, canvasPoint.y);

}

ctx.stroke();

// Calculate the actual Bezier curve

const tStep = curve.tStep || 0.01;

const numPoints = Math.floor((curve.tMax - curve.tMin) / tStep);

const method = curve.method || 'parametric';

const curvePoints = calculateBezierCurve(

controlPoints,

numPoints,

method,

curve.tMin,

curve.tMax

);

// Draw the Bezier curve

ctx.beginPath();

ctx.strokeStyle = curve.curveColor || 'blue';

ctx.lineWidth = 2;

// Convert first point from graph to canvas coordinates

const firstCurvePoint = graphToCanvas(curvePoints[0].x, curvePoints[0].y);

ctx.moveTo(firstCurvePoint.x, firstCurvePoint.y);

// Draw the curve

for (let i = 1; i < curvePoints.length; i++) {

const canvasPoint = graphToCanvas(curvePoints[i].x, curvePoints[i].y);

ctx.lineTo(canvasPoint.x, canvasPoint.y);

}

ctx.stroke();

// Draw control points

ctx.fillStyle = 'red';

for (let i = 0; i < controlPoints.length; i++) {

const canvasPoint = graphToCanvas(controlPoints[i].x, controlPoints[i].y);

ctx.beginPath();

ctx.arc(canvasPoint.x, canvasPoint.y, 5, 0, 2 \* Math.PI);

ctx.fill();

}

}

// Draw all Bezier curves

function drawBezierCurves() {

bezierCurves.forEach(curve => {

drawBezierCurve(curve);

});

// Draw current points if any

if (points.length > 0) {

ctx.fillStyle = 'red';

points.forEach(point => {

const canvasPoint = graphToCanvas(point.x, point.y);

ctx.beginPath();

ctx.arc(canvasPoint.x, canvasPoint.y, 5, 0, 2 \* Math.PI);

ctx.fill();

});

}

}

// Draw coordinate grid with axes and numbers

function drawGrid() {

ctx.clearRect(0, 0, canvas.width/dpr, canvas.height/dpr);

// Draw grid lines

ctx.strokeStyle = '#e0e0e0';

ctx.lineWidth = 1;

// Vertical lines

for (let x = offsetX % scale; x < canvas.width/dpr; x += scale) {

ctx.beginPath();

ctx.moveTo(x, 0);

ctx.lineTo(x, canvas.height/dpr);

ctx.stroke();

}

// Horizontal lines

for (let y = offsetY % scale; y < canvas.height/dpr; y += scale) {

ctx.beginPath();

ctx.moveTo(0, y);

ctx.lineTo(canvas.width/dpr, y);

ctx.stroke();

}

// Draw axes

ctx.strokeStyle = '#000';

ctx.lineWidth = 2;

// X-axis

ctx.beginPath();

ctx.moveTo(0, offsetY);

ctx.lineTo(canvas.width/dpr, offsetY);

ctx.stroke();

// Y-axis

ctx.beginPath();

ctx.moveTo(offsetX, 0);

ctx.lineTo(offsetX, canvas.height/dpr);

ctx.stroke();

// Draw numbers

ctx.fillStyle = '#000';

ctx.font = '12px Arial';

ctx.textAlign = 'center';

ctx.textBaseline = 'middle';

// Calculate decimal places based on scale

const decimalPlaces = calculateDecimalPlaces(scale);

// Calculate step for numbers to avoid overcrowding

const numberStep = calculateNumberStep(scale);

// X-axis numbers

for (let x = Math.ceil(-offsetX / scale) \* scale; x < (canvas.width/dpr - offsetX); x += scale \* numberStep) {

if (Math.abs(x) < 0.001) continue; // Skip very small values near zero

const xPos = offsetX + x;

const value = (x / scale).toFixed(decimalPlaces);

// Remove trailing zeros and decimal point if unnecessary

const displayValue = value.replace(/\.?0+$/, '');

ctx.fillText(displayValue, xPos, offsetY + 20);

}

// Y-axis numbers

for (let y = Math.ceil(-offsetY / scale) \* scale; y < (canvas.height/dpr - offsetY); y += scale \* numberStep) {

if (Math.abs(y) < 0.001) continue; // Skip very small values near zero

const yPos = offsetY + y;

const value = (-y / scale).toFixed(decimalPlaces);

// Remove trailing zeros and decimal point if unnecessary

const displayValue = value.replace(/\.?0+$/, '');

ctx.fillText(displayValue, offsetX - 20, yPos);

}

// Draw origin label

ctx.fillText('0', offsetX - 10, offsetY + 20);

// Draw arrow heads

drawArrow(canvas.width/dpr - 10, offsetY, 1, 0); // X-axis arrow

drawArrow(offsetX, 10, 0, -1); // Y-axis arrow

// X and Y labels

ctx.fillText('X', canvas.width/dpr - 10, offsetY - 20);

ctx.fillText('Y', offsetX + 20, 10);

// Draw all Bezier curves

drawBezierCurves();

}

// ============================================

// CANVAS INTERACTION HANDLERS

// ============================================

// Handle dragging of the coordinate plane

function handleDrag(e) {

if (isAnimating) return;

const rect = canvas.getBoundingClientRect();

const startX = e.clientX - rect.left;

const startY = e.clientY - rect.top;

const startOffsetX = offsetX;

const startOffsetY = offsetY;

function move(e) {

const x = e.clientX - rect.left;

const y = e.clientY - rect.top;

offsetX = startOffsetX + (x - startX);

offsetY = startOffsetY + (y - startY);

drawGrid();

}

function stopDrag() {

document.removeEventListener('mousemove', move);

document.removeEventListener('mouseup', stopDrag);

}

document.addEventListener('mousemove', move);

document.addEventListener('mouseup', stopDrag);

}

// Handle zooming with mouse wheel

function handleZoom(e) {

e.preventDefault();

// Get mouse position in graph coordinates before zoom

const rect = canvas.getBoundingClientRect();

const mouseX = e.clientX - rect.left;

const mouseY = e.clientY - rect.top;

const graphPosBeforeZoom = canvasToGraph(mouseX, mouseY);

// Apply zoom factor

const zoomFactor = e.deltaY > 0 ? 0.9 : 1.1; // Zoom out or in

scale \*= zoomFactor;

// Limit minimum and maximum zoom level

scale = Math.min(Math.max(scale, 10), 200);

// Adjust offset to keep the point under cursor at same position

const graphPosAfterZoom = canvasToGraph(mouseX, mouseY);

offsetX += (graphPosAfterZoom.x - graphPosBeforeZoom.x) \* scale;

offsetY -= (graphPosAfterZoom.y - graphPosBeforeZoom.y) \* scale;

drawGrid();

}

// Animate the coordinate plane

function animate() {

// Example animation - rotate the coordinate system

offsetX += 1;

offsetY = canvas.height / (2 \* dpr) + Math.sin(Date.now() / 1000) \* 50;

drawGrid();

if (isAnimating) {

animationId = requestAnimationFrame(animate);

}

}

// ============================================

// UI EVENT HANDLERS AND UTILITY FUNCTIONS

// ============================================

// Attach basic canvas event listeners

canvas.addEventListener('mousedown', handleDrag);

canvas.addEventListener('wheel', handleZoom);

// Reset button handler

document.getElementById('reset').addEventListener('click', function() {

scale = 40;

offsetX = canvas.width / (2 \* dpr);

offsetY = canvas.height / (2 \* dpr);

drawGrid();

});

// Initial draw when the page loads

window.addEventListener('load', function() {

drawGrid();

});

// Main UI initialization when DOM is ready

document.addEventListener('DOMContentLoaded', function() {

const pointXInput = document.getElementById('pointX');

const pointYInput = document.getElementById('pointY');

const addPointButton = document.getElementById('addPoint');

const pointsList = document.getElementById('pointsList');

const drawCurveButton = document.getElementById('drawCurve');

const messageDiv = document.getElementById('message');

// Initialize points array

points = [];

// Function to update points list in UI

function updatePointsList() {

pointsList.innerHTML = '';

points.forEach((point, index) => {

const li = document.createElement('li');

li.textContent = `Point ${index+1}: (${point.x.toFixed(2)}, ${point.y.toFixed(2)})`;

// Create edit button

const editButton = document.createElement('button');

editButton.textContent = 'Edit';

editButton.className = 'edit-point';

editButton.addEventListener('click', () => {

// Create input fields for editing

li.textContent = '';

const xInput = document.createElement('input');

xInput.type = 'number';

xInput.value = point.x;

xInput.step = '0.1';

xInput.style.width = '60px';

const yInput = document.createElement('input');

yInput.type = 'number';

yInput.value = point.y;

yInput.step = '0.1';

yInput.style.width = '60px';

const saveButton = document.createElement('button');

saveButton.textContent = 'Save';

saveButton.className = 'edit-point';

saveButton.addEventListener('click', () => {

const newX = parseFloat(xInput.value);

const newY = parseFloat(yInput.value);

if (!isNaN(newX) && !isNaN(newY)) {

// Update point coordinates

points[index].x = newX;

points[index].y = newY;

// Update UI

updatePointsList();

// Update curve if auto-update is enabled

if (autoUpdateCurve && activeCurveIndex !== -1) {

bezierCurves[activeCurveIndex].points = [...points];

drawGrid();

} else {

drawGrid(); // Just redraw to show updated point

}

messageDiv.textContent = `Point ${index+1} updated to (${newX.toFixed(2)}, ${newY.toFixed(2)})`;

messageDiv.className = 'message-success';

} else {

messageDiv.textContent = 'Please enter valid coordinates';

messageDiv.className = 'message-error';

}

});

const cancelButton = document.createElement('button');

cancelButton.textContent = 'Cancel';

cancelButton.className = 'delete-point';

cancelButton.addEventListener('click', () => {

updatePointsList(); // Reset list to normal view

});

// Add elements to the list item

li.appendChild(document.createTextNode('X: '));

li.appendChild(xInput);

li.appendChild(document.createTextNode(' Y: '));

li.appendChild(yInput);

li.appendChild(saveButton);

li.appendChild(cancelButton);

});

const deleteButton = document.createElement('button');

deleteButton.textContent = 'Delete';

deleteButton.className = 'delete-point';

deleteButton.addEventListener('click', () => {

points.splice(index, 1);

updatePointsList();

if (points.length >= 2 && autoUpdateCurve && activeCurveIndex !== -1) {

// Update current curve with remaining points

bezierCurves[activeCurveIndex].points = [...points];

drawGrid();

} else if (points.length < 2 && activeCurveIndex !== -1) {

// Not enough points for a curve, remove the active curve

bezierCurves.splice(activeCurveIndex, 1);

activeCurveIndex = -1;

drawGrid();

}

// Show message

messageDiv.textContent = `Point deleted. ${points.length} point(s) remaining.`;

messageDiv.className = 'message-info';

});

// Add buttons to the list item

li.appendChild(editButton);

li.appendChild(deleteButton);

pointsList.appendChild(li);

});

}

// Function to create or update the active Bezier curve

function updateActiveCurve() {

if (points.length < 2) return;

// Default parameters

const tMin = parseFloat(document.getElementById('tMin').value) || 0;

const tMax = parseFloat(document.getElementById('tMax').value) || 1;

const tStep = parseFloat(document.getElementById('tStep').value) || 0.01;

const polylineColor = document.getElementById('polylineColor').value;

const curveColor = document.getElementById('curveColor').value;

const method = document.getElementById('drawingMethod').value;

if (activeCurveIndex === -1) {

// Create a new curve

const newCurve = {

points: [...points],

tMin,

tMax,

tStep,

polylineColor,

curveColor,

method

};

bezierCurves.push(newCurve);

activeCurveIndex = bezierCurves.length - 1;

} else {

// Update existing curve

bezierCurves[activeCurveIndex].points = [...points];

// Keep other parameters unless explicitly changed

}

// Redraw everything

drawGrid();

}

// Format matrix for display in console

function formatMatrix(matrix) {

if (!matrix || !matrix.length) return "Empty matrix";

// Format each row to have consistent decimal places

return matrix.map(row =>

row.map(val => val.toFixed(2)).join('\t')

).join('\n');

}

// Display a matrix on the console with proper formatting

function displayMatrix(matrix, title) {

console.log(`=== ${title} ===`);

console.log(formatMatrix(matrix));

console.log("==================");

}

// Add a point from inputs

addPointButton.addEventListener('click', function() {

const x = parseFloat(pointXInput.value);

const y = parseFloat(pointYInput.value);

if (!isNaN(x) && !isNaN(y)) {

points.push({ x, y });

updatePointsList();

if (autoUpdateCurve) {

updateActiveCurve();

} else {

drawGrid(); // Just redraw to show the new point

}

// Clear inputs

pointXInput.value = '';

pointYInput.value = '';

// Show message

messageDiv.textContent = `Point (${x.toFixed(2)}, ${y.toFixed(2)}) added`;

messageDiv.className = 'message-success';

} else {

messageDiv.textContent = 'Please enter valid coordinates';

messageDiv.className = 'message-error';

}

});

// Allow adding points by clicking on the canvas

canvas.addEventListener('click', function(e) {

if (bezierMode) {

const rect = canvas.getBoundingClientRect();

const x = e.clientX - rect.left;

const y = e.clientY - rect.top;

// Convert to graph coordinates

const graphPoint = canvasToGraph(x, y);

points.push(graphPoint);

updatePointsList();

if (autoUpdateCurve) {

updateActiveCurve();

} else {

drawGrid(); // Just redraw to show the new point

}

messageDiv.textContent = `Point (${graphPoint.x.toFixed(2)}, ${graphPoint.y.toFixed(2)}) added`;

messageDiv.className = 'message-success';

}

});

// Draw Bezier curve (now works as "finalize curve" with custom parameters)

drawCurveButton.addEventListener('click', function() {

if (points.length < 2) {

messageDiv.textContent = 'Need at least 2 points to draw a curve';

messageDiv.className = 'message-error';

return;

}

const tMin = parseFloat(document.getElementById('tMin').value) || 0;

const tMax = parseFloat(document.getElementById('tMax').value) || 1;

const tStep = parseFloat(document.getElementById('tStep').value) || 0.01;

if (tStep > 1 || tStep < 0) {

alert('Step must be between 0 and 1');

return;

}

if (tMax < tMin) {

alert('tMax must be greater than tMin');

return;

}

const polylineColor = document.getElementById('polylineColor').value;

const curveColor = document.getElementById('curveColor').value;

const method = document.getElementById('drawingMethod').value;

if (activeCurveIndex !== -1) {

// Update existing curve with new parameters

const curve = bezierCurves[activeCurveIndex];

curve.tMin = tMin;

curve.tMax = tMax;

curve.tStep = tStep;

curve.polylineColor = polylineColor;

curve.curveColor = curveColor;

curve.method = method;

} else {

// Create a new curve

const newCurve = {

points: [...points],

tMin,

tMax,

tStep,

polylineColor,

curveColor,

method

};

bezierCurves.push(newCurve);

activeCurveIndex = bezierCurves.length - 1;

}

// Finalize the current curve and prepare for a new one

updatePointsList();

activeCurveIndex = -1;

// Redraw everything

drawGrid();

messageDiv.textContent = 'Curve finalized with custom parameters';

messageDiv.className = 'message-success';

});

// Clear all curves

document.getElementById('clearAllCurves').addEventListener('click', function() {

bezierCurves = [];

points = []; // Also clear points when clearing all curves

activeCurveIndex = -1; // Reset active curve index

updatePointsList();

drawGrid(); // Redraw the grid without any curves

messageDiv.textContent = 'All curves cleared';

messageDiv.className = 'message-info';

});

// Toggle bezier mode

document.getElementById('toggleBezierMode').addEventListener('click', function() {

bezierMode = !bezierMode;

this.textContent = bezierMode ? 'Exit Point Mode' : 'Enter Point Mode';

messageDiv.textContent = bezierMode ? 'Click on the canvas to add points' : 'Point mode disabled';

messageDiv.className = 'message-info';

});

// New function to start a new curve

document.getElementById('newCurve').addEventListener('click', function() {

if (points.length >= 2 && activeCurveIndex !== -1) {

// Finalize the current curve

bezierCurves[activeCurveIndex].points = [...points];

}

// Reset for a new curve

points = [];

activeCurveIndex = -1;

updatePointsList();

drawGrid();

messageDiv.textContent = 'Started new curve';

messageDiv.className = 'message-info';

});

// Save matrix (now displays the Bezier basis coefficient matrix)

document.getElementById('saveMatrix').addEventListener('click', function() {

if (bezierCurves.length === 0) {

messageDiv.textContent = 'No curves to save';

messageDiv.className = 'message-error';

return;

}

// Display coefficient matrices for each curve

bezierCurves.forEach((curve, index) => {

const n = curve.points.length - 1;

const bezierMatrix = getBezierMatrix(n);

// Display matrix in console

displayMatrix(bezierMatrix, `Bezier Coefficient Matrix (n=${n}) for Curve #${index + 1}`);

// Log points for reference

console.log(`Curve #${index + 1} Control Points:`, curve.points);

if (curve.method === 'matrix') {

// Also display a sample calculation for t=0.5 to verify

const t = 0.5;

const paramVector = Array(n + 1).fill(0).map((\_, i) => Math.pow(t, n - i));

console.log(`Parameter Vector for t=${t}:`, paramVector);

const coefficients = multiplyVectorMatrix(paramVector, bezierMatrix);

console.log(`Resulting Coefficients:`, coefficients);

const point = calculateBezierPointMatrix(curve.points, t);

console.log(`Resulting Point at t=${t}:`, point);

}

});

// Original curve data output

const output = bezierCurves.map(curve => {

return {

points: curve.points,

parameters: {

tMin: curve.tMin,

tMax: curve.tMax,

tStep: curve.tStep,

method: curve.method

},

colors: {

polyline: curve.polylineColor,

curve: curve.curveColor

}

};

});

console.log('Saved curves data:', output);

messageDiv.textContent = 'Bezier matrix coefficients saved to console (press F12 to view)';

messageDiv.className = 'message-success';

});

// Add a toggle for auto-update mode

document.getElementById('toggleAutoUpdate').addEventListener('click', function() {

autoUpdateCurve = !autoUpdateCurve;

this.textContent = autoUpdateCurve ? 'Disable Auto Update' : 'Enable Auto Update';

messageDiv.textContent = autoUpdateCurve ? 'Curves will update automatically' : 'Manual curve updates enabled';

messageDiv.className = 'message-info';

});

// Add change event listener for drawing method

document.getElementById('drawingMethod').addEventListener('change', function() {

if (autoUpdateCurve && activeCurveIndex !== -1 && points.length >= 2) {

bezierCurves[activeCurveIndex].method = this.value;

drawGrid();

messageDiv.textContent = `Drawing method changed to ${this.value}`;

messageDiv.className = 'message-info';

}

});

});

**Результат виконання роботи**

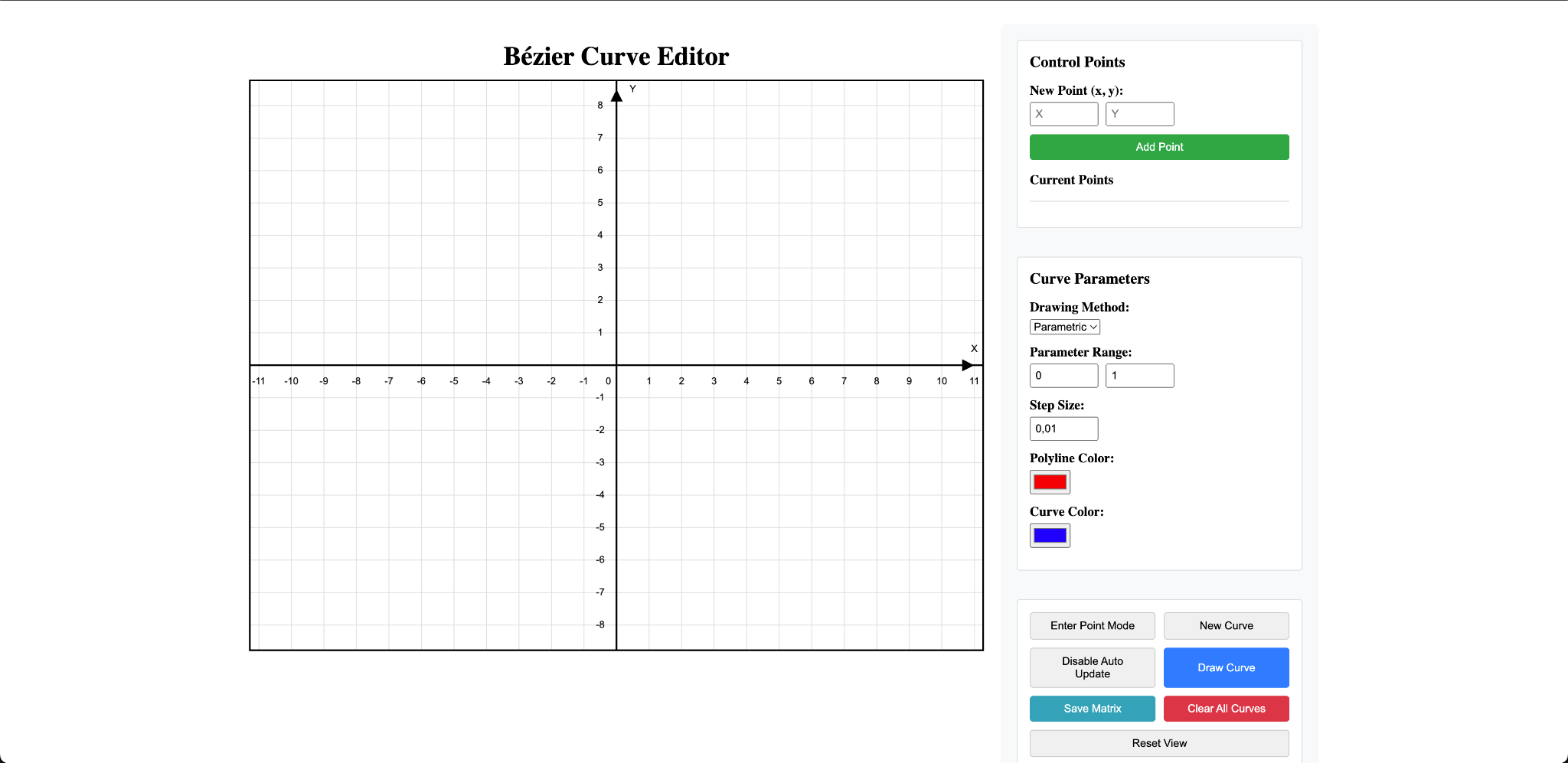
****

Рис.1. Загальний вигляд програми

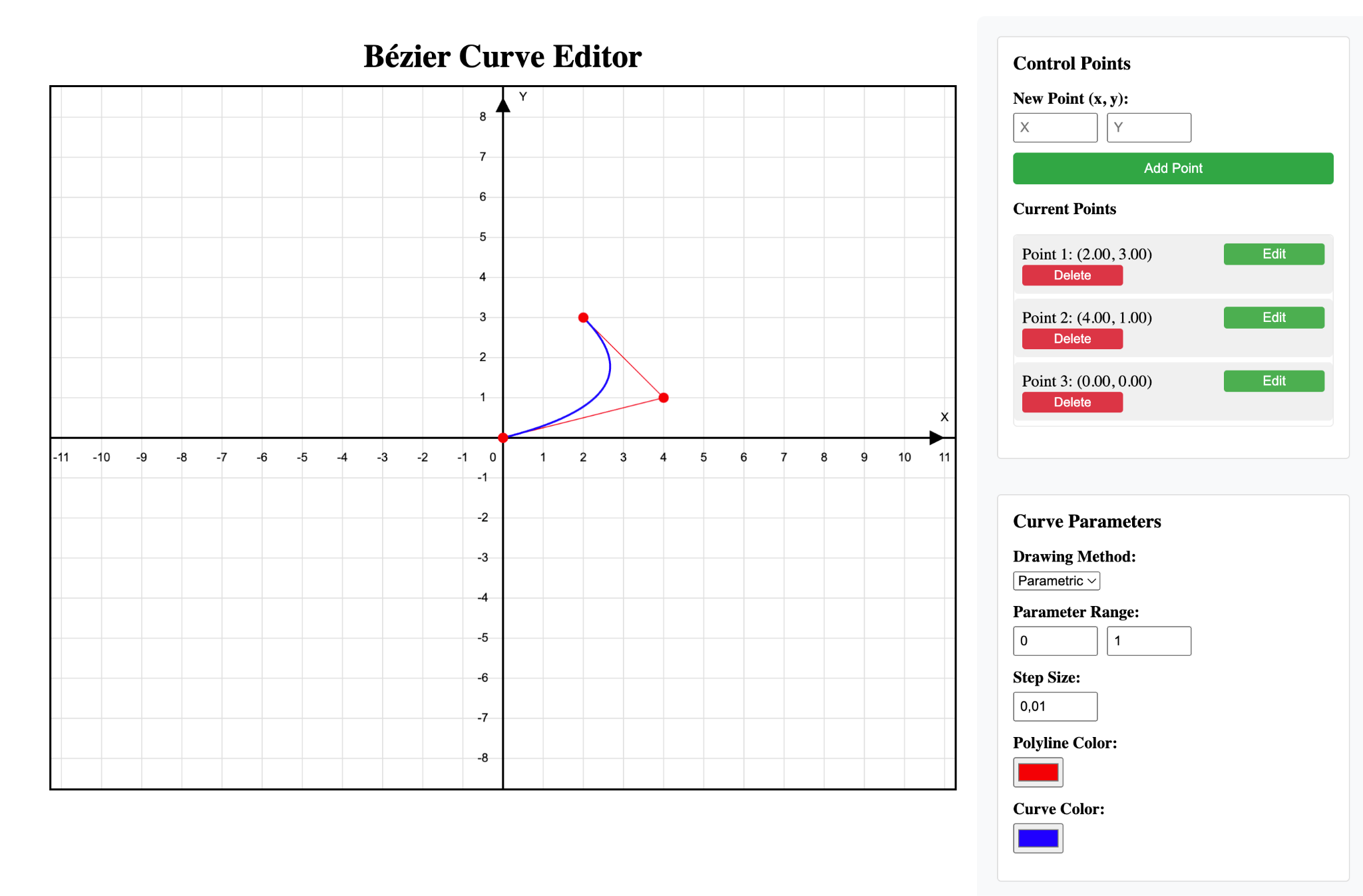
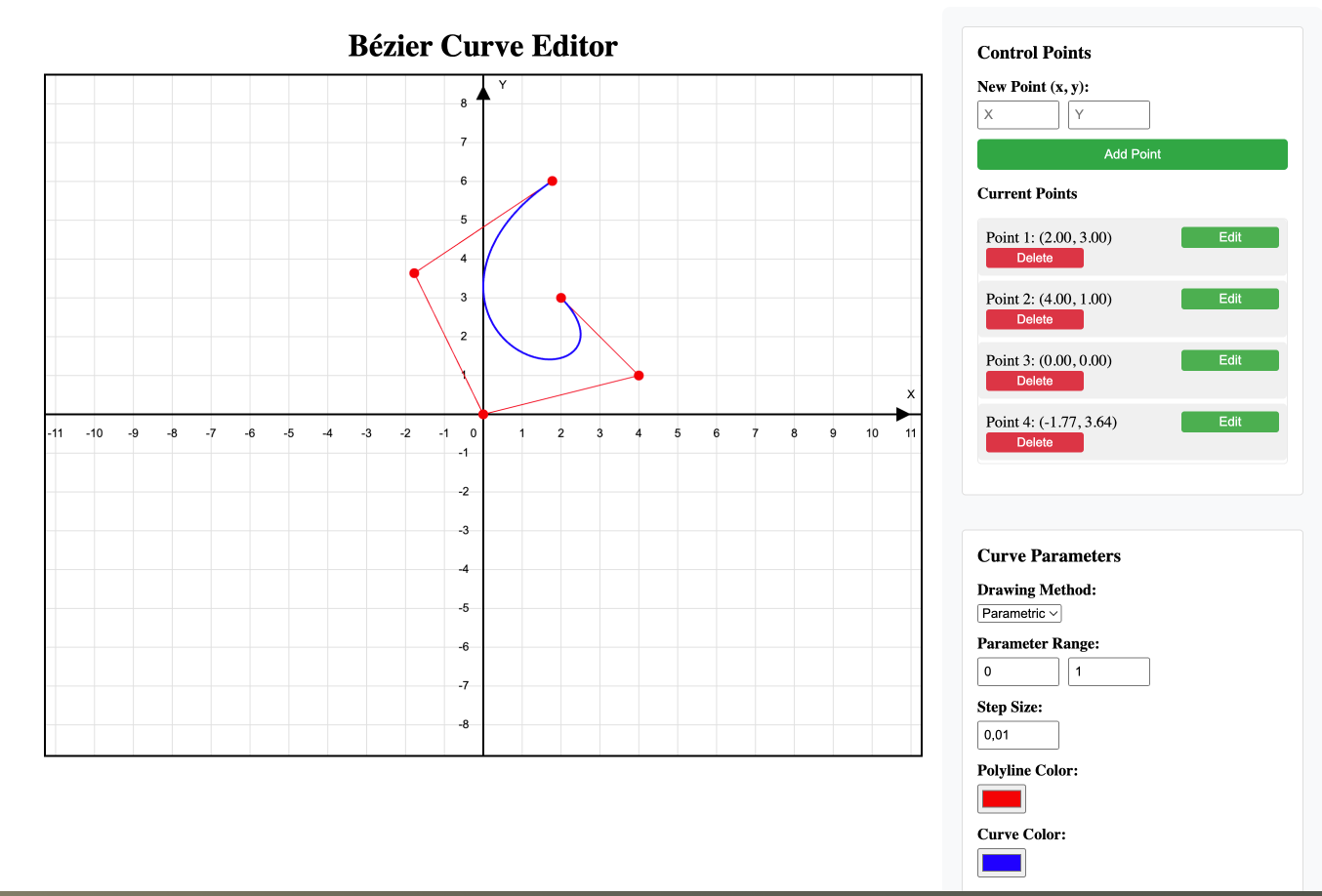


Рис.2. Крива другого порядку

Рис.3. Додано точки для кривої четвертого порядку

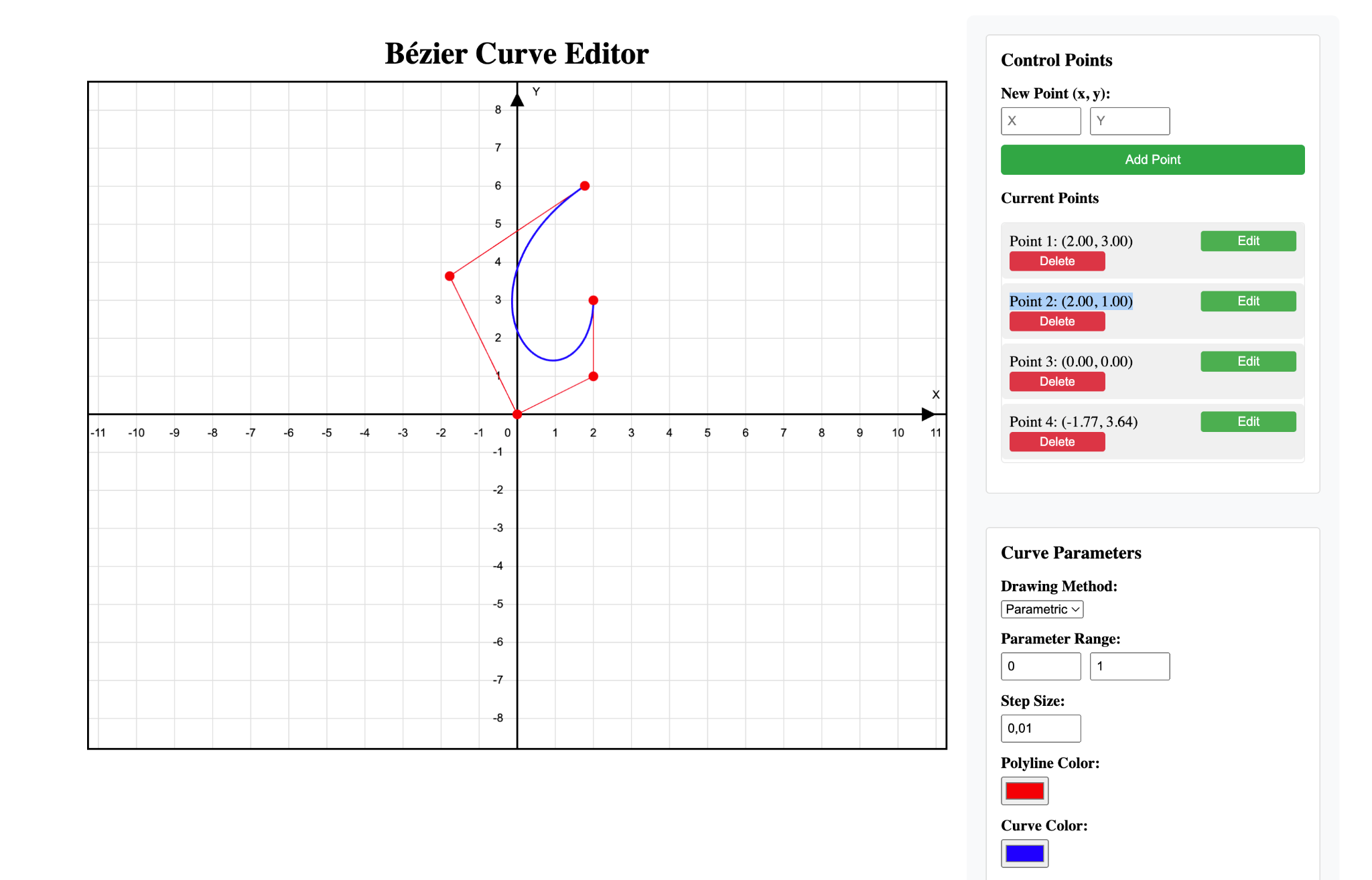


Рис.4. Зміна другої точки, відповідно і кривої

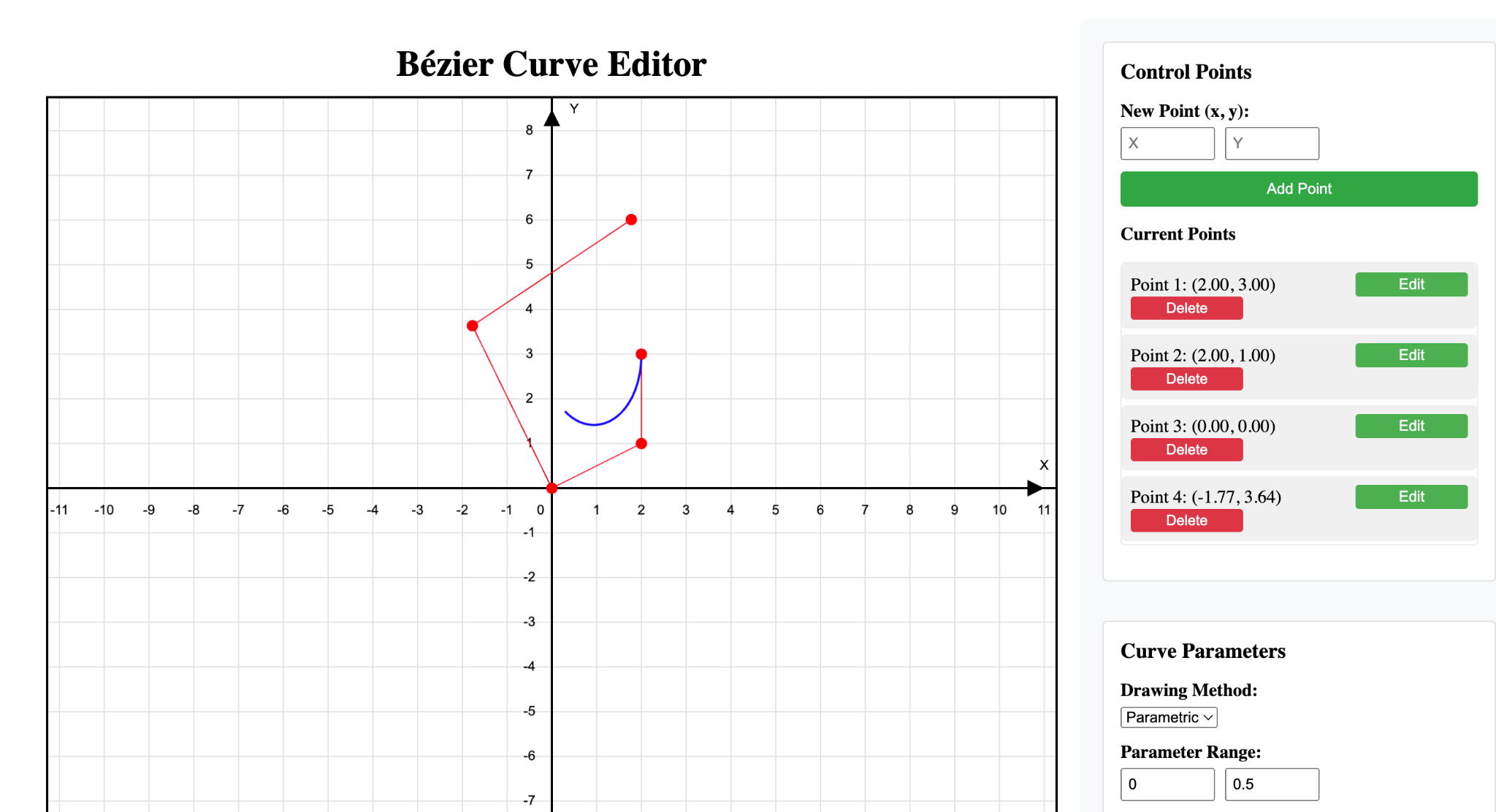


Рис.5. Перемалювання кривої з параметром 0<t<0.5

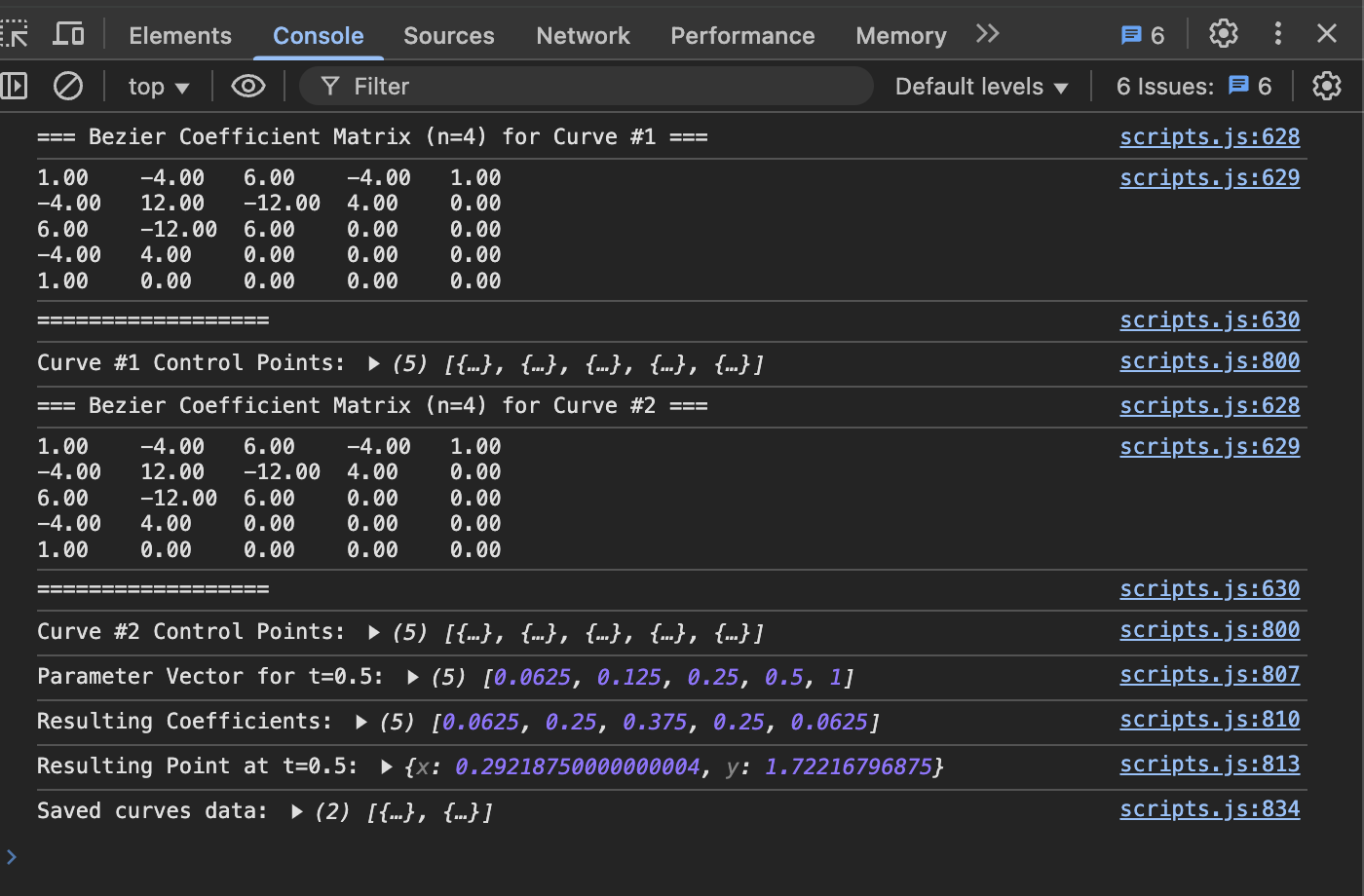


Рис.6. Сформована матриця коефіцієнтів для матриці четвертого порядку

**Висновок**

На цій лабораторній я навчився програмувати алгоритми побудови кривої Безьє, мною було розроблено інтерактивну програму для побудови кривої Безьє параметричним та матричним методом та реалізовано можливість редагування кривої та додавання нових точок.