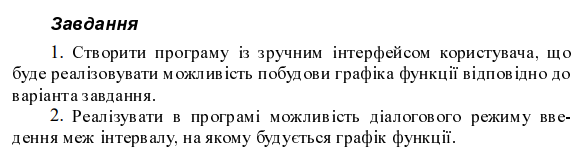
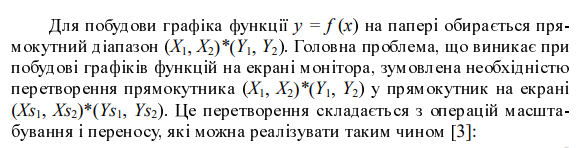
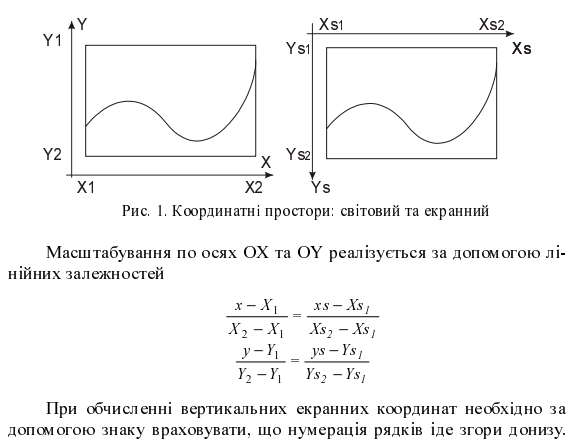
**Лабораторна робота №1**

**Побудова графіків функцій**



Короткі теоретичні відомості





**Рішення**

1. Текст програми

**Index.ts**

const tmp = {

func: '(0.1 + 0.3 \* x \* x \* x) / (5 + Math.sqrt(0.15 + x \* x \* x \* x))',

};

const app = new App(document.querySelector('.app'), createFunc(tmp.func));

const gui = new dat.GUI();

const guiFunc = gui.add(tmp, 'func');

function createFunc(str) {

return <(x, y?: number) => number>new Function('x, t', `

try {

return ${str};

} catch {

return 0;

}

`);

}

**App.ts**

export default class App {

private ctx: CanvasRenderingContext2D = null;

private $: IAppElements = {};

private \_animationFrameId = 0;

private \_size = new Vector(0, 0);

private \_coords = new Vector(0, 0);

private \_zoom = new Vector(55, 55);

private \_pointsInterval = 1;

constructor(private root: HTMLElement, private func: IFunc) {

this.init();

this.initEvents();

}

private init() {

this.getElements();

this.resize();

this.start();

}

private initEvents() {

const hammer = new Hammer(this.$.canvas);

hammer.get('pinch').set({ enable: true });

let startCoords;

hammer.on('pan panstart', ({ deltaX, deltaY, type }) => {

if (type === 'panstart') {

startCoords = this.\_coords.copy();

}

const delta = new Vector(-deltaX, deltaY).div(this.\_zoom);

this.\_coords = startCoords.copy().add(delta);

});

this.$.canvas.addEventListener('mousewheel', (e: MouseWheelEvent) => {

const zoom = this.\_zoom.copy();

zoom.y -= e.deltaY / 50;

if (e.shiftKey) {

zoom.x -= e.deltaX / 50;

} else {

zoom.x -= e.deltaY / 50;

}

this.setZoom(zoom);

});

window.addEventListener('resize', throttle(100, () => {

this.resize();

}));

window.addEventListener('load', () => {

this.resize();

});

}

public setZoom(zoom) {

zoom.x = Math.min(Math.max(zoom.x, 0.01), 1e10);

zoom.y = Math.min(Math.max(zoom.y, 0.01), 1e10);

this.\_zoom = zoom;

}

start() {

if (this.\_animationFrameId) return;

const app = this;

this.\_animationFrameId = requestAnimationFrame(function tik() {

app.clearCanvas();

app.tik();

requestAnimationFrame(tik);

});

}

stop() {

cancelAnimationFrame(this.\_animationFrameId);

this.\_animationFrameId = 0;

}

private tik() {

this.drawAllGrids();

this.renderFunc();

}

private renderFunc() {

const ctx = this.ctx;

ctx.save();

ctx.strokeStyle = 'red';

ctx.lineWidth = 1;

ctx.beginPath();

this.getPoints().forEach((point, i, points) => {

if (i != 0) {

const prev = points[i - 1];

if (Math.abs(point.y - prev.y) > this.\_pointsInterval \* 5) {

ctx.stroke();

ctx.beginPath();

}

}

const type = (i === 0) ? 'moveTo' : 'lineTo';

ctx[type](point.x, point.y);

});

ctx.stroke();

ctx.restore();

}

private getPoints() {

const interval = this.\_pointsInterval;

const steps = Math.ceil(this.\_size.x / interval) + 1;

const xs = new Array(steps).fill(0).map((\_, i) => i \* interval);

const ys = xs.map(x => this.xToWorld(x))

.map(x => this.func(x))

.map(y => this.yToView(y));

return xs.map((x, i) => new Vector(x, ys[i]));

}

/\*\*

\* Convert viewport coords to world coords

\* @param coords viport coordinates

\*/

private toWorld(coords: Vector) {

return coords.copy()

.sub(this.\_size.copy().div(new Vector(2, 2)))

.mul(new Vector(1, -1))

.div(this.\_zoom)

.add(this.\_coords);

}

/\*\*

\* Convert world coords to viewport coords

\* @param coords world coordinates

\*/

private toView(coords: Vector) {

return coords.copy()

.sub(this.\_coords)

.mul(this.\_zoom)

.mul(new Vector(1, -1))

.add(this.\_size.copy().div(new Vector(2, 2)));

}

private xToWorld(x: number) { return this.toWorld(new Vector(x, 0)).x; }

private yToWorld(y: number) { return this.toWorld(new Vector(0, y)).y; }

private xToView(x: number) { return this.toView(new Vector(x, 0)).x; }

private yToView(y: number) { return this.toView(new Vector(0, y)).y; }

private drawAllGrids() {

const ctx = this.ctx;

const z = Math.min(this.\_zoom.x, this.\_zoom.y);

ctx.save();

ctx.strokeStyle = 'rgba(255,255,255,.2)';

if (z >= 9) {

ctx.lineWidth = 1;

this.drawGrid(new Vector(1, 1));

}

if (z >= 3) {

ctx.lineWidth = 1;

this.drawGrid(new Vector(10, 10));

}

if (z >= 0.4) {

ctx.lineWidth = 2;

this.drawGrid(new Vector(50, 50));

} else if (z >= 0.08) {

ctx.lineWidth = 1;

this.drawGrid(new Vector(500, 500));

}

ctx.lineWidth = (z >= 0.4) ? 3 : 1;

this.drawGrid(new Vector(10000, 10000));

ctx.restore();

}

private drawGrid(interval: Vector) {

const ctx = this.ctx;

const z = this.\_zoom.copy();

const step = interval.copy().mul(z);

const numberOfSteps = this.\_size.copy().div(step);

const g0 = this.toWorld(new Vector(-1, -1));

const worldStart = g0.copy().sub(g0.mod(interval));

const start = this.toView(worldStart);

for (let i = 0; i < numberOfSteps.x; i++) {

const x = Math.round(start.x + step.x \* i);

ctx.beginPath();

ctx.moveTo(x, 0);

ctx.lineTo(x, this.\_size.y);

ctx.stroke();

}

for (let i = 0; i < numberOfSteps.y; i++) {

const y = Math.round(start.y + step.y \* i);

ctx.beginPath();

ctx.moveTo(0, y);

ctx.lineTo(this.\_size.x, y);

ctx.stroke();

}

}

private resize() {

this.updateMetrics();

this.updateCanvasSize();

}

private updateMetrics() {

this.\_size.x = this.root.offsetWidth;

this.\_size.y = this.root.offsetHeight;

}

private updateCanvasSize() {

this.$.canvas.width = this.\_size.x;

this.$.canvas.height = this.\_size.y;

}

private clearCanvas() {

this.ctx.clearRect(0, 0, this.\_size.x, this.\_size.y);

}

private getElements() {

this.$.canvas = this.root.querySelector('.app\_\_canvas');

this.ctx = this.$.canvas.getContext('2d');

}

public setFunc(func: IFunc) {

this.func = func;

}

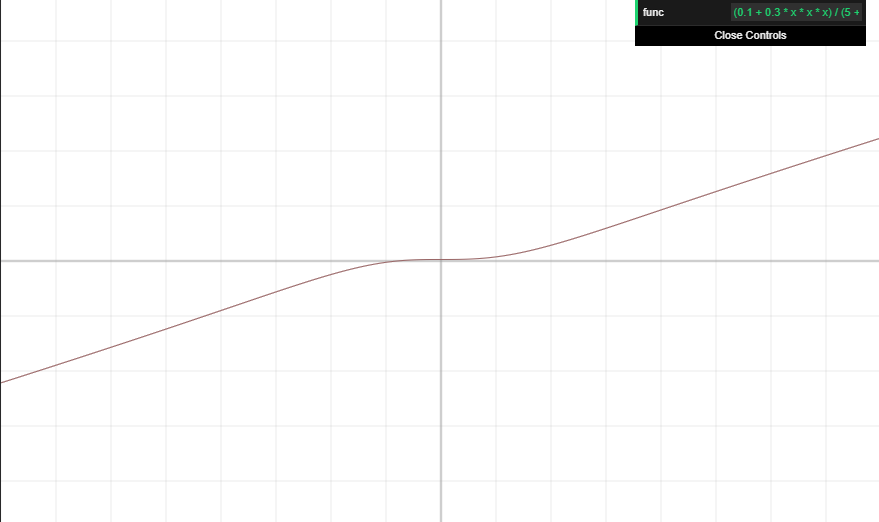
set coords(coords: Vector) {

this.\_coords = coords.copy();

}

}

2. Результат виконання програми



**Висновок:** на цій лабораторній роботі оволодів навичками побудови графіків функцій.