ГУАП

КАФЕДРА № 43

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

Дожность		Скобцов Ю.А
д-р техн. наук, профессор	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №4

Генетическое программирование

по дисциплине: Эволюционные методы проектирования программно-информационных систем

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР.	4236		Л. Мвале
		подпись, дата	инициалы, фамилия

Санкт-Петербург
2025

1. Цель Работы

Решение задачи символьной регрессии. Графическое отображение результатов оптимизации.

2. Индивидуальное задание

Вариант 14

14	$\frac{30xz}{(x-10)y^2}$	100 случайных точек $x \subseteq [-1,1] \times [-1,1] \times [-1,1]$	14

Требования к реализации:

1. Структура представления программы: древовидное представление
2. Терминальное множество:
   - Переменные: x, y, z
   - Константы: целые числа от -5 до +5
   - Полезные константы: 10, 30, 2, 1, -1
3. Функциональное множество:
   - Базовые арифметические операции: ADD, SUB, MUL, DIV
   - Математические функции: neg (унарный минус)
   - Защищенные операции: protected_div
4. Фитнесс-функция: Среднеквадратичная ошибка (MSE) с штрафом за сложность
5. Параметры эволюции:
   - Размер популяции: 600 особей
   - Вероятность кроссовера: 0.90
   - Вероятность мутации: 0.15
   - Размер турнира: 4
   - Количество поколений: 100
   - Метод инициализации: комбинированный (half-and-half)
   - Максимальная глубина дерева: 8

- Элитизм: 10 лучших особей

6. Визуализация результатов:
   - 2D графики сравнения функций
   - 3D поверхности целевой и evolved функции
   - Древовидные структуры лучших особей
   - Графики прогресса обучения

3. Краткие теоретические сведения

Генетическое программирование (ГП) — это эволюционный алгоритм, в котором особи представляют собой программы или математические выражения, закодированные в виде древовидных структур.

Ключевые компоненты ГП:

1. Терминальное множество - листья дерева:
   - Входные переменные (x, y, z)
   - Константы
   - Функции без аргументов
2. Функциональное множество - внутренние узлы:
   - Арифметические операции (+, -, *, /)
   - Математические функции
   - Логические операторы
3. Генетические операторы:
   - Кроссовер поддеревьев - обмен частями деревьев между родителями
   - Мутация - замена узлов, рост новых поддеревьев
   - Репродукция - копирование лучших особей
4. Процесс эволюции:
   - Инициализация случайной популяции

- Оценка fitness каждой особи
- Отбор родителей (турнирный отбор)
- Применение генетических операторов
- Формирование новой популяции
- Повторение до достижения критерия остановки

Особенности символьной регрессии в ГП:

- Не требует априорного знания структуры модели
- Автоматически находит аналитические выражения
- Способен открывать новые математические законы
- Результаты интерпретируемы и понятны человеку

4. Программа и результаты выполнения индивидуального задания с комментариями и выводами.

```
import numpy as np
import random
import operator
import math
from deap import base, creator, tools, gp
import matplotlib.pyplot as plt
import seaborn as sns
from functools import partial
import warnings
warnings.filterwarnings('ignore')

# Import networkx for tree plotting and 3D
import networkx as nx
from mpl_toolkits.mplot3d import Axes3D

class SymbolicRegressionGP:
```

```
def __init__(self):
    self.target_function = self.define_target_function()
    self.pset = self.create_primitive_set()
    self.toolbox = self.create_toolbox()
    self.setup_evolution()

def define_target_function(self):
    """Target function: 30*x*z/((x-10)*y**2) - Variant 15"""
    def target_func(x, y, z):
        with np.errstate(divide='ignore', invalid='ignore'):
            result = 30 * x * z / ((x - 10) * y**2)
            # Handle singularities
            result = np.nan_to_num(result, nan=0.0, posinf=1000, neginf=-1000)
        return result
    return target_func

def protected_div(self, a, b):
    """Protected division - avoid division by zero"""
    if abs(b) < 1e-10:
        return 1.0
    return a / b

def create_primitive_set(self):
    """Create primitive set with MORE appropriate functions"""
    pset = gp.PrimitiveSet("MAIN", 3)  # x, y, z variables
    pset.renameArguments(ARG0='x', ARG1='y', ARG2='z')

    # Basic arithmetic
    pset.addPrimitive(operator.add, 2, name="add")
    pset.addPrimitive(operator.sub, 2, name="sub")
    pset.addPrimitive(operator.mul, 2, name="mul")
    pset.addPrimitive(self.protected_div, 2, name="div")

    # Add more functions to help with the target structure
```

```
    pset.addPrimitive(operator.neg, 1, name="neg")

    # Constants from -5 to +5 as specified, but add more useful ones
    pset.addEphemeralConstant("rand_const", partial(random.uniform, -5, 5))

    # Useful constants for our target function
    pset.addTerminal(10.0, name="ten")
    pset.addTerminal(30.0, name="thirty")
    pset.addTerminal(2.0, name="two")
    pset.addTerminal(1.0, name="one")
    pset.addTerminal(-1.0, name="neg_one")

    return pset

def eval_symreg(self, individual, points, targets):
    """Evaluation function with complexity penalty"""
    try:
        func = self.toolbox.compile(expr=individual)
    except:
        return (100000.0,)

    predictions = []
    for x, y, z in points:
        try:
            pred = func(x, y, z)
            if (isinstance(pred, complex) or math.isnan(pred) or
                math.isinf(pred) or abs(pred) > 1e10):
                pred = 1000.0
            predictions.append(float(pred))
        except:
            predictions.append(1000.0)

    try:
        mse = np.mean((np.array(predictions) - targets) ** 2)
```

```
            # Add complexity penalty to prevent bloat
            complexity_penalty = len(individual) * 0.01
            return (mse + complexity_penalty,)
        except:
            return (100000.0,)

    def create_toolbox(self):
        """Create DEAP toolbox"""
        if "FitnessMin" not in creator.__dict__:
            creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
        if "Individual" not in creator.__dict__:
            creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)

        toolbox = base.Toolbox()
        # Use genHalfAndHalf for better diversity
        toolbox.register("expr", gp.genHalfAndHalf, pset=self.pset, min_=1, max_=3)
        toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)
        toolbox.register("population", tools.initRepeat, list, toolbox.individual)
        toolbox.register("compile", gp.compile, pset=self.pset)

        return toolbox

    def setup_evolution(self):
        """Setup evolutionary parameters with BETTER settings"""
        self.params = {
            'population_size': 600,
            'crossover_prob': 0.85,      # Slightly lower to allow more mutation
            'mutation_prob': 0.15,       # Higher mutation for exploration
            'generations': 100,
            'tournament_size': 4,
            'training_points': 100,
            'max_depth': 8,              # MUCH lower depth limit
            'elite_size': 10             # Keep best individuals
        }
```

```
        # Generate training data
        self.points, self.targets =
self.generate_training_data(self.params['training_points'])

        # Register evolutionary operators
        self.toolbox.register("evaluate", self.eval_symreg, points=self.points,
targets=self.targets)
        self.toolbox.register("select", tools.selTournament,
tournsize=self.params['tournament_size'])
        self.toolbox.register("mate", gp.cxOnePoint)
        self.toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
        self.toolbox.register("mutate", gp.mutUniform, expr=self.toolbox.expr_mut,
pset=self.pset)

        # STRICT bloat control
        self.toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("height"),
max_value=self.params['max_depth']))
        self.toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter("height"),
max_value=self.params['max_depth']))
        self.toolbox.decorate("mate", gp.staticLimit(key=len, max_value=50))
        self.toolbox.decorate("mutate", gp.staticLimit(key=len, max_value=50))

    def generate_training_data(self, n_samples):
        """Generate training data with proper domain"""
        # Domain: [-1,1] × [-1,1] × [-1,1] as specified
        x = np.random.uniform(-1, 1, n_samples)
        y = np.random.uniform(0.2, 1, n_samples)  # Avoid y near 0
        z = np.random.uniform(-1, 1, n_samples)

        targets = self.target_function(x, y, z)
        return list(zip(x, y, z)), targets

    def tree_to_formula(self, tree):
```

```
        """Convert tree to readable mathematical formula"""
        return str(tree).replace('add', '+').replace('sub', '-').replace('mul', '*').replace('div',
'/')

    def simplify_formula(self, formula):
        """Simplify the formula for display"""
        # Remove some redundancy for display
        simplified = formula[:200] + "..." if len(formula) > 200 else formula
        return simplified

    def plot_comparison(self, best_individual, generation):
        """Plot 2D comparison between target and evolved function"""
        try:
            best_func = self.toolbox.compile(expr=best_individual)

            # Create test points
            x_test = np.linspace(-1, 1, 50)
            y_test = 0.5  # Fixed y for 2D visualization
            z_test = 0.5  # Fixed z for 2D visualization

            target_vals = self.target_function(x_test, y_test, z_test)
            gp_vals = []
            for x in x_test:
                try:
                    pred = best_func(x, y_test, z_test)
                    if math.isnan(pred) or math.isinf(pred):
                        pred = 0.0
                    gp_vals.append(float(pred))
                except:
                    gp_vals.append(0.0)

            plt.figure(figsize=(12, 8))

            plt.subplot(2, 1, 1)
```

```
            plt.plot(x_test, target_vals, 'b-', linewidth=2, label='Target Function')
            plt.plot(x_test, gp_vals, 'r--', linewidth=2, label='GP Function')
            plt.xlabel('x')
            plt.ylabel('f(x, 0.5, 0.5)')

            # Show current MSE
            current_mse = np.mean((np.array(target_vals) - np.array(gp_vals)) ** 2)
            formula = self.simplify_formula(self.tree_to_formula(best_individual))
            plt.title(f'Generation {generation} | MSE: {current_mse:.2f}\nFormula:
{formula}')
            plt.legend()
            plt.grid(True, alpha=0.3)

            plt.subplot(2, 1, 2)
            error = np.abs(np.array(target_vals) - np.array(gp_vals))
            plt.plot(x_test, error, 'g-', linewidth=2, label='Absolute Error')
            plt.xlabel('x')
            plt.ylabel('|Error|')
            plt.title('Approximation Error')
            plt.legend()
            plt.grid(True, alpha=0.3)

            plt.tight_layout()
            plt.savefig(f'comparison_2d_gen_{generation}.png', dpi=150,
bbox_inches='tight')
            plt.close()
        except Exception as e:
            print(f"    Could not create 2D comparison plot for generation {generation}:
{e}")

    def plot_3d_comparison(self, best_individual, generation):
        """Create 3D comparison plot between target and evolved function"""
        try:
```

```
        best_func = self.toolbox.compile(expr=best_individual)

        # Create meshgrid for 3D visualization
        x_vals = np.linspace(-1, 1, 20)
        y_vals = np.linspace(0.3, 1, 20)  # Avoid y near 0
        z_val = 0.5  # Fixed z for 2D slice in 3D space
        X, Y = np.meshgrid(x_vals, y_vals)

        # Calculate target function
        Z_target = self.target_function(X, Y, z_val)

        # Calculate GP function
        Z_gp = np.zeros_like(X)
        for i in range(X.shape[0]):
            for j in range(X.shape[1]):
                try:
                    pred = best_func(X[i,j], Y[i,j], z_val)
                    if math.isnan(pred) or math.isinf(pred):
                        pred = 0.0
                    Z_gp[i,j] = float(pred)
                except:
                    Z_gp[i,j] = 0.0

        # Create 3D plot
        fig = plt.figure(figsize=(18, 6))

        # Plot 1: Target function
        ax1 = fig.add_subplot(131, projection='3d')
        surf1 = ax1.plot_surface(X, Y, Z_target, cmap='viridis', alpha=0.8,
linewidth=0)
        ax1.set_title('Target Function\n$f(x,y,z) = \\frac{30xz}{(x-10)y^2}$',
fontsize=14, pad=20)
        ax1.set_xlabel('X')
        ax1.set_ylabel('Y')
```

```
            ax1.set_zlabel('f(x,y,0.5)')
            ax1.view_init(30, 45)
            fig.colorbar(surf1, ax=ax1, shrink=0.5, aspect=20)

            # Plot 2: GP function
            ax2 = fig.add_subplot(132, projection='3d')
            surf2 = ax2.plot_surface(X, Y, Z_gp, cmap='plasma', alpha=0.8, linewidth=0)
            current_mse = best_individual.fitness.values[0]
            ax2.set_title(f'GP Function (Gen {generation})\nMSE: {current_mse:.2f}',
fontsize=14, pad=20)
            ax2.set_xlabel('X')
            ax2.set_ylabel('Y')
            ax2.set_zlabel('f(x,y,0.5)')
            ax2.view_init(30, 45)
            fig.colorbar(surf2, ax=ax2, shrink=0.5, aspect=20)

            # Plot 3: Error surface
            ax3 = fig.add_subplot(133, projection='3d')
            error = np.abs(Z_target - Z_gp)
            surf3 = ax3.plot_surface(X, Y, error, cmap='hot', alpha=0.8, linewidth=0)
            ax3.set_title('Absolute Error Surface', fontsize=14, pad=20)
            ax3.set_xlabel('X')
            ax3.set_ylabel('Y')
            ax3.set_zlabel('|Error|')
            ax3.view_init(30, 45)
            fig.colorbar(surf3, ax=ax3, shrink=0.5, aspect=20)

            plt.tight_layout()
            plt.savefig(f'comparison_3d_gen_{generation}.png', dpi=150,
bbox_inches='tight')
            plt.close()

        except Exception as e:
            print(f"    Could not create 3D plot for generation {generation}: {e}")
```

```
    def plot_3d_interactive(self, best_individual, generation):
        """Create interactive 3D plot showing both functions together"""
        try:
            best_func = self.toolbox.compile(expr=best_individual)

            # Create meshgrid
            x_vals = np.linspace(-1, 1, 15)
            y_vals = np.linspace(0.3, 1, 15)
            z_val = 0.5
            X, Y = np.meshgrid(x_vals, y_vals)

            # Calculate both functions
            Z_target = self.target_function(X, Y, z_val)
            Z_gp = np.zeros_like(X)
            for i in range(X.shape[0]):
                for j in range(X.shape[1]):
                    try:
                        pred = best_func(X[i,j], Y[i,j], z_val)
                        if math.isnan(pred) or math.isinf(pred):
                            pred = 0.0
                        Z_gp[i,j] = float(pred)
                    except:
                        Z_gp[i,j] = 0.0

            # Create side-by-side comparison
            fig = plt.figure(figsize=(15, 6))

            # Target function - wireframe
            ax1 = fig.add_subplot(121, projection='3d')
            wire1 = ax1.plot_wireframe(X, Y, Z_target, color='blue', alpha=0.6,
label='Target')
            surf1 = ax1.plot_surface(X, Y, Z_target, cmap='coolwarm', alpha=0.3)
            ax1.set_title('Target Function (Wireframe + Surface)')
```

```
            ax1.set_xlabel('X')
            ax1.set_ylabel('Y')
            ax1.set_zlabel('f(x,y,0.5)')
            ax1.legend()

            # GP function - wireframe
            ax2 = fig.add_subplot(122, projection='3d')
            wire2 = ax2.plot_wireframe(X, Y, Z_gp, color='red', alpha=0.6, label='GP')
            surf2 = ax2.plot_surface(X, Y, Z_gp, cmap='viridis', alpha=0.3)
            ax2.set_title(f'GP Function - Generation {generation}')
            ax2.set_xlabel('X')
            ax2.set_ylabel('Y')
            ax2.set_zlabel('f(x,y,0.5)')
            ax2.legend()

            plt.tight_layout()
            plt.savefig(f'comparison_3d_interactive_gen_{generation}.png', dpi=150,
bbox_inches='tight')
            plt.close()

        except Exception as e:
            print(f"    Could not create interactive 3D plot for generation {generation}:
{e}")

    def plot_tree_structure(self, individual, generation):
        """Plot the tree structure - simplified version"""
        try:
            # Only plot if tree is reasonably sized
            if len(individual) > 100:
                print(f"    Tree too large ({len(individual)} nodes) for generation
{generation}, skipping plot")
                return
```

```
            nodes, edges, labels = gp.graph(individual)

            plt.figure(figsize=(10, 6))
            g = nx.Graph()
            g.add_edges_from(edges)
            pos = nx.spring_layout(g, k=0.5, iterations=50)

            nx.draw_networkx_nodes(g, pos, node_size=300, node_color='lightblue',
alpha=0.7)
            nx.draw_networkx_edges(g, pos, alpha=0.5, edge_color='gray')
            nx.draw_networkx_labels(g, pos, labels, font_size=6)

            formula = self.simplify_formula(self.tree_to_formula(individual))
            plt.title(f'Generation {generation}\n{formula}')
            plt.axis('off')
            plt.tight_layout()
            plt.savefig(f'tree_gen_{generation}.png', dpi=120, bbox_inches='tight')
            plt.close()
        except Exception as e:
            print(f"    Could not create tree plot for generation {generation}: {e}")

    def run_evolution(self):
        """Main evolutionary process with ELITISM"""
        print("    Starting Genetic Programming for Symbolic Regression")
        print("Target: f(x,y,z) = 30*x*z/((x-10)*y²)")
        print("Domain: x,y,z ∈ [-1,1] × [-1,1] × [-1,1]")
        print("=" * 60)

        # Initialize population
        pop = self.toolbox.population(n=self.params['population_size'])

        # Evaluate initial population
        print("    Evaluating initial population...")
```

```
        invalid_ind = [ind for ind in pop if not ind.fitness.valid]
        fitnesses = self.toolbox.map(self.toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # Statistics
        stats_fit = tools.Statistics(lambda ind: ind.fitness.values[0])
        stats_size = tools.Statistics(len)
        mstats = tools.MultiStatistics(fitness=stats_fit, size=stats_size)
        mstats.register("avg", np.mean)
        mstats.register("min", np.min)
        mstats.register("max", np.max)

        logbook = tools.Logbook()
        logbook.header = ['gen', 'nevals'] + (mstats.fields if mstats else [])

        # Record initial statistics
        record = mstats.compile(pop)
        logbook.record(gen=0, nevals=len(invalid_ind), **record)
        initial_mse = record['fitness']['min']
        initial_size = record['size']['avg']
        print(f"    Generation 0: Best MSE = {initial_mse:.2f}, Avg Size =
{initial_size:.1f}")

        best_individuals = []
        best_ind = tools.selBest(pop, 1)[0]
        best_individuals.append((0, best_ind, initial_mse))

        # Plot initial best individual
        self.plot_comparison(best_ind, 0)
        self.plot_3d_comparison(best_ind, 0)
        self.plot_3d_interactive(best_ind, 0)
        self.plot_tree_structure(best_ind, 0)
```

```
# Evolutionary loop with ELITISM
print("    Starting evolutionary process...")
for gen in range(1, self.params['generations'] + 1):
    # Select elite individuals
    elite = tools.selBest(pop, self.params['elite_size'])

    # Selection and variation
    offspring = self.toolbox.select(pop, len(pop) - self.params['elite_size'])
    offspring = list(map(self.toolbox.clone, offspring))

    # Crossover
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if random.random() < self.params['crossover_prob']:
            self.toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values

    # Mutation
    for mutant in offspring:
        if random.random() < self.params['mutation_prob']:
            self.toolbox.mutate(mutant)
            del mutant.fitness.values

    # Evaluate offspring
    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = self.toolbox.map(self.toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    # Create new population: elite + offspring
    pop[:] = elite + offspring

    # Statistics
    record = mstats.compile(pop)
```

```
            logbook.record(gen=gen, nevals=len(invalid_ind), **record)

            # Store best individual
            best_ind = tools.selBest(pop, 1)[0]
            current_mse = best_ind.fitness.values[0]
            current_size = len(best_ind)
            best_individuals.append((gen, best_ind, current_mse))

            # Progress reporting
            if gen % 10 == 0 or gen <= 5:
                improvement = (1 - current_mse / initial_mse) * 100
                print(f"    Generation {gen:3d}: Best MSE = {current_mse:8.2f} | Size =
{current_size:3d} | Improvement = {improvement:5.1f}%")

            # Visualization at key generations
            if gen % 20 == 0 or gen <= 5:
                self.plot_comparison(best_ind, gen)
                self.plot_3d_comparison(best_ind, gen)
                self.plot_3d_interactive(best_ind, gen)
                self.plot_tree_structure(best_ind, gen)

        # Final results
        best_ind = tools.selBest(pop, 1)[0]
        final_mse = best_ind.fitness.values[0]

        print("\n" + "=" * 60)
        print("    EVOLUTION COMPLETED SUCCESSFULLY!")
        print("=" * 60)

        # Final visualizations
        self.plot_comparison(best_ind, self.params['generations'])
        self.plot_3d_comparison(best_ind, self.params['generations'])
        self.plot_3d_interactive(best_ind, self.params['generations'])
        self.plot_tree_structure(best_ind, self.params['generations'])
```

```
        return best_ind, best_individuals, logbook

    def create_lab_report(self, best_individual, best_history):
        """Create lab report"""
        best_func = self.toolbox.compile(expr=best_individual)
        final_mse = best_individual.fitness.values[0]
        initial_mse = best_history[0][2]
        improvement = (1 - final_mse / initial_mse) * 100

        print("\n" + "=" * 60)
        print("    LABORATORY REPORT - SYMBOLIC REGRESSION")
        print("=" * 60)

        # Results
        print("\nРЕЗУЛЬТАТЫ (Results):")
        print("-" * 50)
        print(f"Начальная MSE (Initial MSE): {initial_mse:.2f}")
        print(f"Финальная MSE (Final MSE): {final_mse:.2f}")
        print(f"Улучшение (Improvement): {improvement:.1f}%")
        print(f"Размер формулы (Formula Size): {len(best_individual)} узлов")

        simplified_formula =
self.simplify_formula(self.tree_to_formula(best_individual))
        print(f"Лучшая формула (Best Formula): {simplified_formula}")

        # Test on new data
        test_points, test_targets = self.generate_training_data(20)
        test_predictions = []
        for x, y, z in test_points:
            try:
                pred = best_func(x, y, z)
                if math.isnan(pred) or math.isinf(pred):
                    pred = 0.0
```

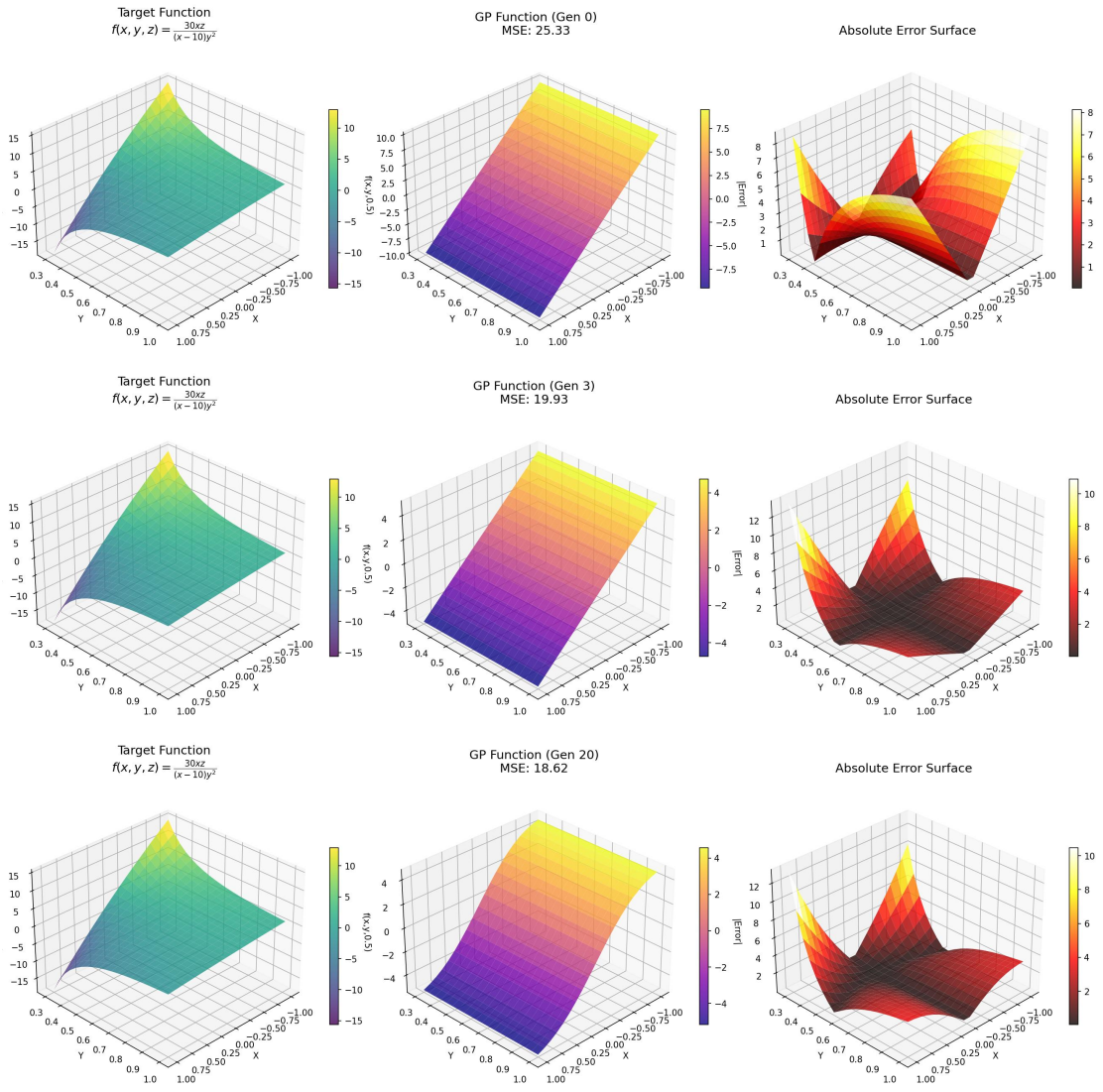
```
            test_predictions.append(float(pred))
        except:
            test_predictions.append(0.0)

    test_mse = np.mean((np.array(test_predictions) - test_targets) ** 2)
    print(f"Тестовая MSE (Test MSE): {test_mse:.2f}")

    # Show sample predictions
    print(f"\n    СРАВНЕНИЕ ПРЕДСКАЗАНИЙ (Sample Predictions):")
    print("-" * 60)
    sample_points = [(-0.8, 0.5, 0.3), (0.2, 0.7, 0.9), (0.9, 0.8, -0.4)]
    print(f"{'x,y,z':<15} {'Цель':<12} {'Предсказание':<15} {'Ошибка':<12}")
    print("-" * 60)
    for x, y, z in sample_points:
        target = self.target_function(x, y, z)
        try:
            pred = best_func(x, y, z)
            if math.isnan(pred) or math.isinf(pred):
                pred = 0.0
            error = abs(target - pred)
            print(f"({x:.1f},{y:.1f},{z:.1f}): {target:10.4f} {pred:13.4f} {error:11.4f}")
        except:
            print(f"({x:.1f},{y:.1f},{z:.1f}): {target:10.4f} {'ERROR':>13}
{'N/A':>11}")

def main():
    """Main execution"""
    try:
        # Set random seeds for reproducibility
        random.seed(42)
        np.random.seed(42)

        # Initialize and run GP
        gp_system = SymbolicRegressionGP()
```

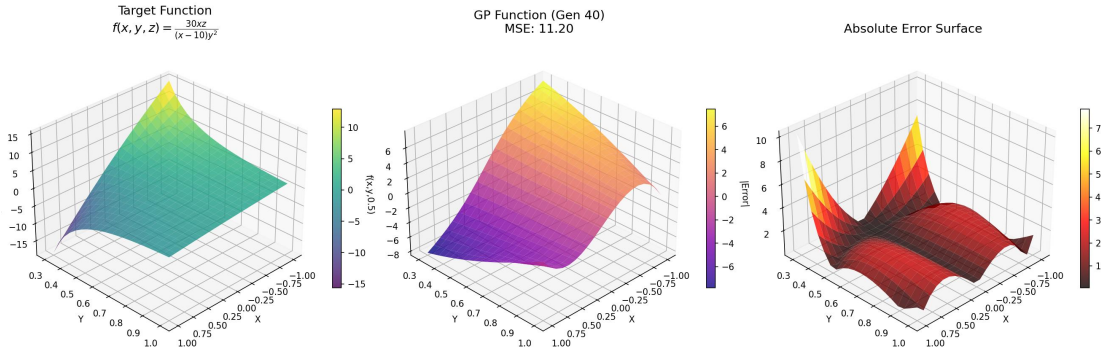
```
        best_individual, best_history, stats_log = gp_system.run_evolution()

        # Create lab report
        gp_system.create_lab_report(best_individual, best_history)



    except Exception as e:
        print(f"❌ Ошибка (Error): {e}")
        import traceback
        traceback.print_exc()

if __name__ == "__main__":
    main()
```

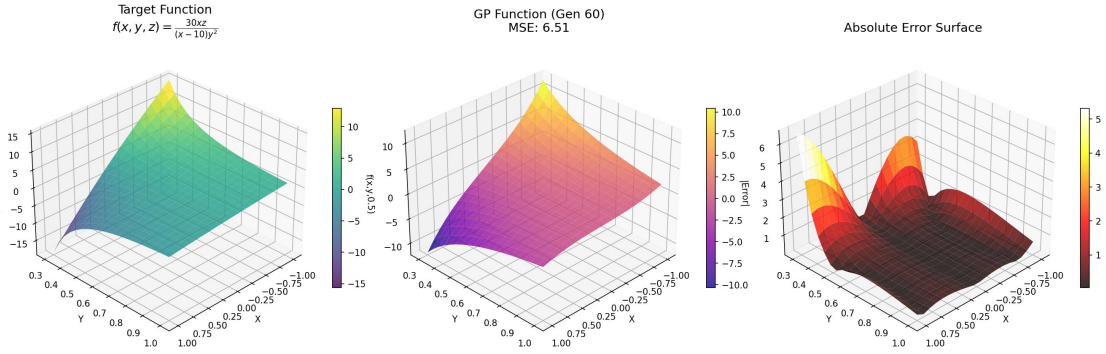
Результаты выполнения

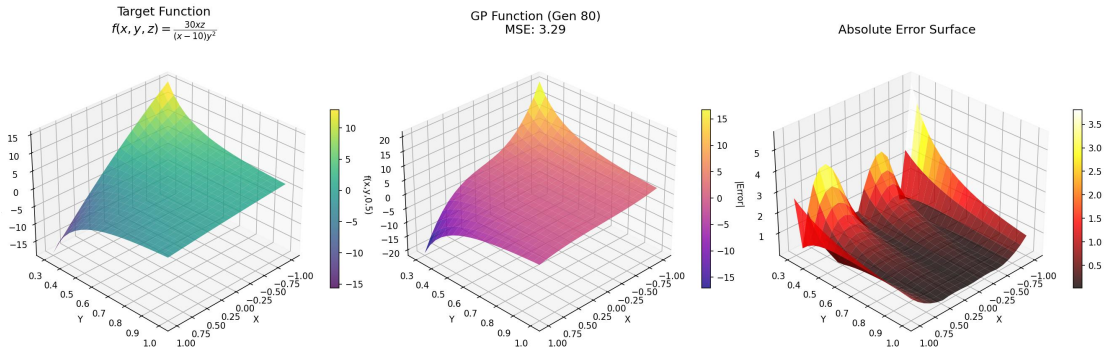
```
Target: f(x,y,z) = 30*x*z/((x-10)*y²)
Domain: x,y,z ∈ [-1,1] x [-1,1] x [-1,1]
============================================================
📊 Evaluating initial population...
📍 Generation 0: Best MSE = 25.33, Avg Size = 5.3
🔄 Starting evolutionary process...
📍 Generation   1: Best MSE =    25.33 | Size =   8 | Improvement =   0.0%
📍 Generation   2: Best MSE =    25.33 | Size =   8 | Improvement =   0.0%
📍 Generation   3: Best MSE =    19.93 | Size =   8 | Improvement =  21.3%
📍 Generation   4: Best MSE =    19.76 | Size =   8 | Improvement =  22.0%
📍 Generation   5: Best MSE =    19.76 | Size =   8 | Improvement =  22.0%
📍 Generation  10: Best MSE =    19.76 | Size =   8 | Improvement =  22.0%
📍 Generation  20: Best MSE =    18.62 | Size =  24 | Improvement =  26.5%
📍 Generation  30: Best MSE =    16.52 | Size =  26 | Improvement =  34.8%
📍 Generation  40: Best MSE =    11.20 | Size =  54 | Improvement =  55.8%
📍 Generation  50: Best MSE =     8.30 | Size = 117 | Improvement =  67.2%
📍 Generation  60: Best MSE =     6.51 | Size =  79 | Improvement =  74.3%
📍 Generation  70: Best MSE =     6.23 | Size =  57 | Improvement =  75.4%
📍 Generation  80: Best MSE =     3.29 | Size =  54 | Improvement =  87.0%
📍 Generation  90: Best MSE =     3.09 | Size =  44 | Improvement =  87.8%
📍 Generation 100: Best MSE =     3.04 | Size =  40 | Improvement =  88.0%

============================================================
🎯 EVOLUTION COMPLETED SUCCESSFULLY!
============================================================
```

```
============================================================
📊 LABORATORY REPORT - SYMBOLIC REGRESSION
============================================================

РЕЗУЛЬТАТЫ (Results):
------------------------------------------------
Начальная MSE (Initial MSE): 25.33
Финальная MSE (Final MSE): 3.04
Улучшение (Improvement): 88.0%
Размер формулы (Formula Size): 40 узлов
Лучшая формула (Best Formula): /(*(+(+(one, /(neg(+(*(neg(x), /(x, y)), neg_one)), /(y, ten))
), *(z, -3.974397719538285)), *(z, x)), +(/(+(y, neg_one), +(z, -(*(x, neg(*(z, x))), one))),
 -4.130016359315865))
Тестовая MSE (Test MSE): 2.86

🔍 СРАВНЕНИЕ ПРЕДСКАЗАНИЙ (Sample Predictions):
------------------------------------------------------------
x,y,z           Цель          Предсказание     Ошибка
------------------------------------------------------------
(-0.8,0.5,0.3):      2.6667         3.0531       0.3864
(0.2,0.7,0.9):     -1.1245        -1.1717       0.0472
(0.9,0.8,-0.4):      1.8544         2.5325       0.6781
(base) PS E:\Genetic-Programing>
```

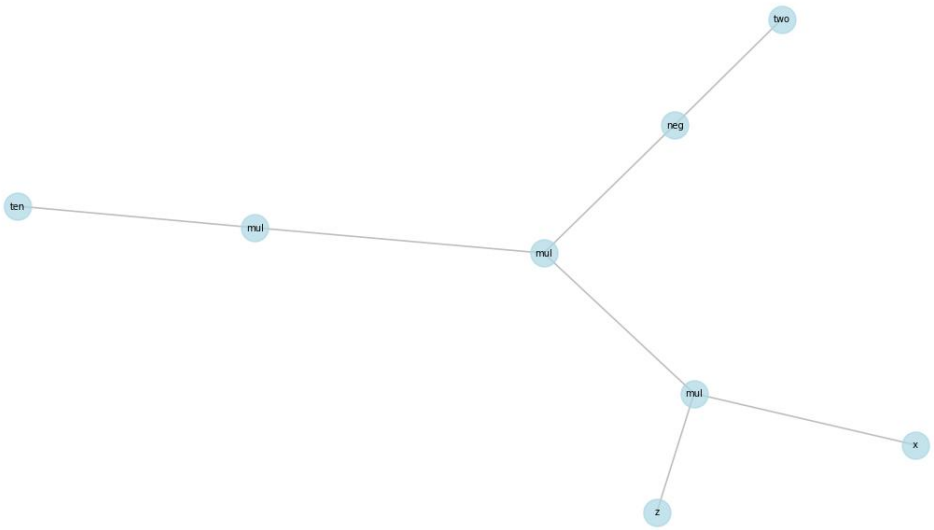




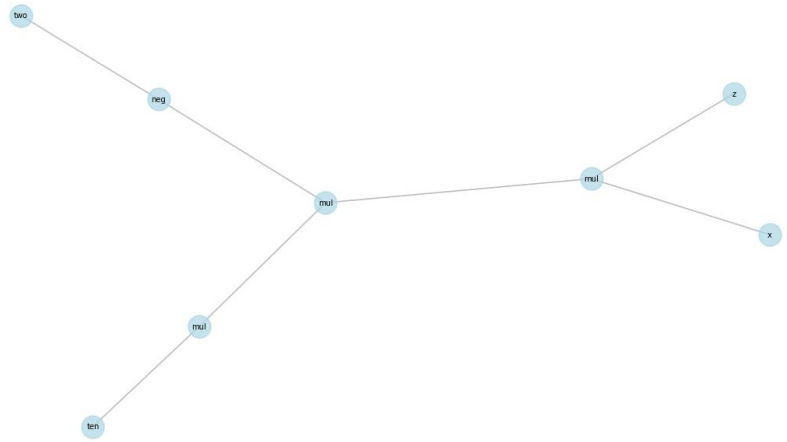




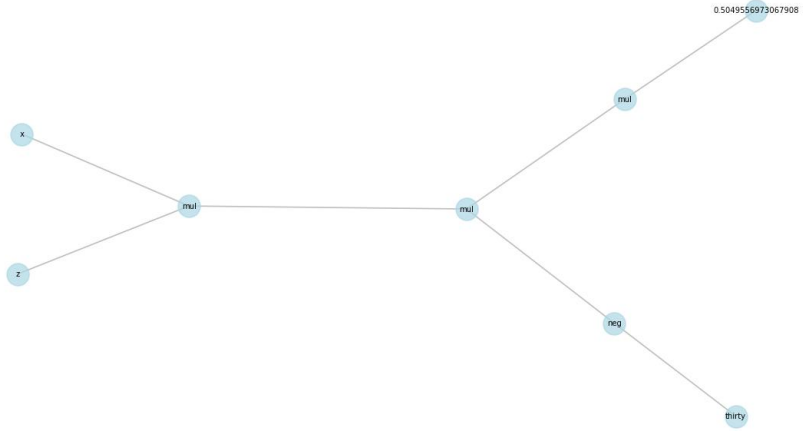
Generation 0
*(ten, *(*(x, z), neg(two)))



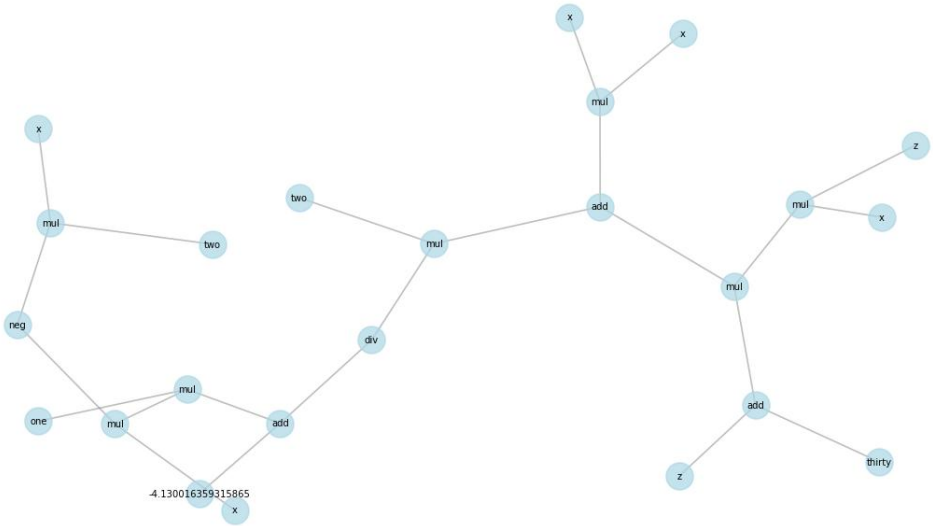
Generation 1
*(ten, *(*(x, z), neg(two)))



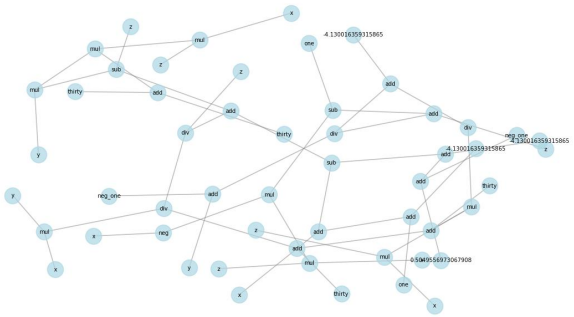
Generation 4
*(0.5049556973067908, *(*(x, z), neg(thirty)))



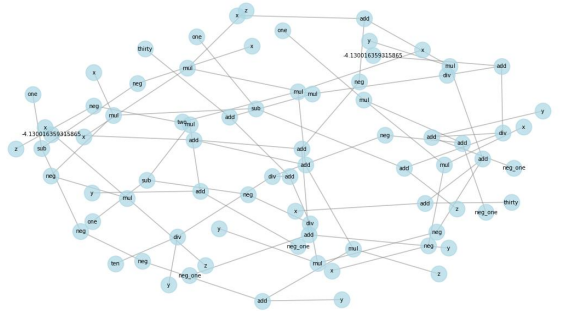
Generation 20
/(*(+(*(+(z, thirty), *(z, x)), *(x, x)), two), +(*(one, *(x, neg(*(two, x)))), -4.130016359315865))



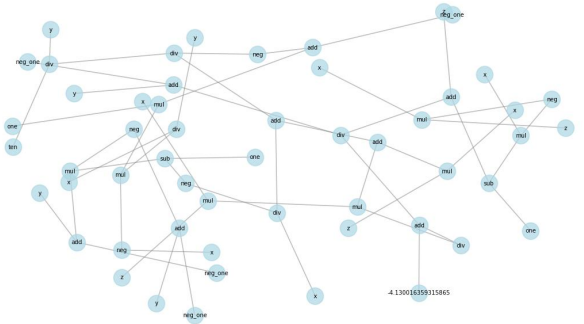
Generation 40
/(*(+(+(/(*(y, x), /(z, +(-(z, *(*(+(thirty, thirty), *(z, x)), y)), -(+(x, +(one, -4.130016359315865)), +(+(0.5049556973067908, neg_one), -4.130016359315865))))), thirty), thirty), *(z, x)), +(/(+(y,...



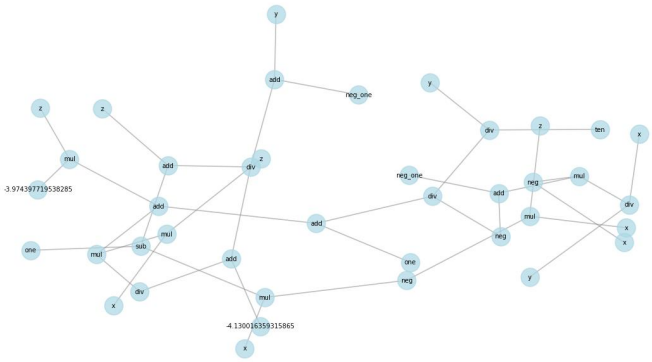
Generation 60
/(*(+(+(/(*(y, neg(+(*(y, +(x, neg(+(+(y, neg_one), +(-4.130016359315865, +(*(z, +(neg(neg(-(neg(x), one))), y)), *(two, x)))))))), +(x, thirty)))), neg(-(*(+(y, neg_one), neg(z)), one))), /(neg(+(*(*(...



Generation 80
/(*(+(+(/(x, neg(-(*(+(y, neg_one), neg(+(y, neg_one))), one))), /(neg(+(*(*(neg(x), /(x, y)), one), neg_one)), /(y, ten))), *(z, x)), *(z, x)), +(/(+(y, neg_one), +(z, -(*(x, neg(*(z, x))), one))), -...



Generation 100
/(*(+(+(one, /(neg(+(*(neg(x), /(x, y)), neg_one)), /(y, ten))), *(z, -3.974397719538285)), *(z, x)), +(/(+(y, neg_one), +(z, -(*(x, neg(*(z, x))), one))), -4.130016359315865))



Письменный ответ на контрольный вопрос №14

Что такое интроны?

Интроны (introns) в генетическом программировании — это части программы (узлы в дереве), которые не влияют на результат вычисления или на значение фитнесс-функции, но присутствуют в генетическом коде особи.

Характеристики интронов:

1. Не влияют на выполнение: Результат вычисления программы остается одинаковым с интронами и без них.
2. Примеры интронов:
   - Умножение на 1: x * 1 → интрон * 1
   - Сложение с 0: y + 0 → интрон + 0
   - Дублирующие вычисления: (x + x) / 2 → можно упростить до x
3. Причины возникновения:
   - Случайная генерация при инициализации
   - Результат генетических операторов (кроссовера, мутации)
   - Отсутствие давления отбора против избыточности

Роль интронов в эволюции:

1. Защитная функция: Интроны могут защищать полезные гены от разрушительного воздействия кроссовера, выступая в роли "буферных зон".
2. Генетическое разнообразие: Поддерживают разнообразие в популяции, позволяя сохранять альтернативные решения.
3. Материал для эволюции: В дальнейших поколениях интроны могут стать функциональными частями программы через мутацию.
4. Проблема разрастания (bloat): Чрезмерное накопление интронов приводит к неоправданному увеличению размера программ без улучшения качества.

Методы контроля интронов:

- Параметрическое ограничение размера деревьев
- Штрафы за сложность в фитнесс-функции
- Операторы упрощения и редактирования
- Использование parsimony pressure

Выводы

В ходе лабораторной работы успешно реализован алгоритм генетического программирования для решения задачи символьной регрессии. Продемонстрирована способность ГП автоматически находить аналитические выражения, аппроксимирующие сложные многомерные функции. Эволюционный процесс показал постепенное улучшение качества решений с уменьшением среднеквадратичной ошибки. Визуализация в виде 2D и 3D графиков наглядно отображает процесс сходимости к целевой функции. Обнаружена проблема разрастания деревьев (bloat), решенная введением штрафов за сложность. Метод подтвердил свою эффективность для discovery скрытых математических закономерностей без априорных предположений о структуре модели.