ГУАП

КАФЕДРА № 43

ОТЧЕТ  
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| профессор |  |  |  | Ю.А. Скобцов |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №8 |
|  |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4134 |  |  |  | Д. В. Самарин |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**Цель работы:**

Разработка эволюционного алгоритма оценки стоимости программных проектов.

Графическое отображение результатов.

**Вариант 14.**

№ варианта – 4

Тип эволюционного алгоритма - ГА

Кодирование решения – Веществ. Вектор

Фитнесс-функция (тип ошибки) – ED

Оператор кроссовера - арифметич.

Оператор мутации - арифметич.

Оператор репродукции – турнир

# Задание:

1. Разобраться в теоретическом описании математического метода оценки стоимости

программного проекта – модели СОСОМО.

2. Из приведенной выше табл. 8.1 (или табл. 8.2) экспериментальных данных

(программных проектов НАСА) отобрать из 18 проектов в качестве обучающего

множества 13 (40) проектов.

3. В соответствии с вариантом лабораторной работы, заданного табл. 8.3 определить тип

используемого эволюционного алгоритма (генетический или роевой алгоритм,

генетическое программирование), кодирование потенциального решения, вид ошибки в

целевой функции, вид генетических операторов кроссовера, мутации и репродукции

4. Отработать алгоритм решения задачи с помощью заданного метода на обучающем

множестве.

5. Разработать программу на языке Python, включающую в себя реализацию

пользовательского интерфейса в виде диалогового меню, реализацию алгоритма

решения поставленной задачи заданным методом.

6. Протестировать разработанную программу: вычислить заданный тип ошибки на

тестовом множестве – оставшихся 5 (из 18) проектов табл. 8.1 (или табл. 8.2).

5. Выполнить вывод полученного решения в виде текста и графиков.

# Программа, ход работы

|  |
| --- |
| Inter.py  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  # Загрузка данных  data = pd.DataFrame({      "Номер проекта": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18],      "L": [90.2, 46.2, 46.5, 54.5, 31.1, 67.5, 12.8, 10.5, 21.5, 3.1, 4.2, 7.8, 2.1, 5.0, 78.6, 9.7, 12.5, 100.8],      "Me": [30.0, 20.0, 19.0, 20.0, 35.0, 29.0, 26.0, 34.0, 31.0, 26.0, 19.0, 31.0, 28.0, 29.0, 35.0, 27.0, 27.0, 34.0],      "Ef": [115.8, 96.0, 79.0, 909.8, 39.6, 98.4, 18.9, 10.3, 28.5, 7.0, 9.0, 7.3, 5.0, 8.4, 98.7, 15.6, 23.9, 138.3]  })  # Разделение данных на обучающее и тестовое множества, а - задает базовый уровень трудозатрат для проекта данного типа. b - Определяет степень зависимости трудозатрат от размера проекта  train\_data = data.iloc[:13]  test\_data = data.iloc[13:]  # Диалоговое меню  print("=== Параметры алгоритма ===")  try:      population\_size = int(input("Введите размер популяции (по умолчанию 700): ") or 700)      generations = int(input("Введите количество поколений (по умолчанию 100): ") or 100)      mutation\_rate = float(input("Введите вероятность мутации (по умолчанию 0.5): ") or 0.5)  except ValueError:      print("Ошибка ввода. Используются значения по умолчанию.")      population\_size = 700      generations = 100      mutation\_rate = 0.5  # Параметры COCOMO для различных типов ПО позволяет COCOMO учитывать реальную сложность проекта и точнее оценивать ресурсы, необходимые для его выполнения  cocomo\_params = {      "1": ("организационное", (2.4, 1.05)),      "2": ("базовое", (2.5, 1.2)),      "3": ("усложнённое", (2.8, 1.35))  }  print("\n=== Выбор типа программного обеспечения ===")  print("1: Организационное")  print("2: Базовое")  print("3: Усложнённое")  software\_choice = input("Выберите тип ПО (1/2/3): ") or "2"  software\_type, (a, b) = cocomo\_params.get(software\_choice, cocomo\_params["2"])  print(f"Выбран тип ПО: {software\_type} (a = {a}, b = {b})")  # Инициализация популяции  def initialize\_population():      population = []      for \_ in range(population\_size):          random\_a = np.random.uniform(2, 3)  # Пределы для a          random\_b = np.random.uniform(0.5, 1)  # Пределы для b          population.append([random\_a, random\_b])      return np.array(population)  # Оценка приспособленности  def fitness(individual, data\_subset):      predictions = individual[0] \* (data\_subset["L"] \*\* individual[1])      error = np.sqrt(np.sum((predictions - data\_subset["Ef"]) \*\* 2))      max\_error = np.max(data\_subset["Ef"])      probability\_error = error / max\_error      return -probability\_error  # Оператор кроссовера  def arithmetic\_crossover(parent1, parent2):      alpha = np.random.rand()      child1 = alpha \* parent1 + (1 - alpha) \* parent2      child2 = alpha \* parent2 + (1 - alpha) \* parent1      return child1, child2  # Оператор мутации  def aggressive\_mutation(child):      if np.random.rand() < mutation\_rate:          mutation\_amount = np.random.uniform(-0.1, 0.1, size=child.shape)  # Более агрессивная мутация          child += mutation\_amount      return child  # Оператор турнира для селекции  def tournament\_selection(population, fitness\_values, tournament\_size=3):      selected\_indices = np.random.choice(range(population.shape[0]), tournament\_size)      selected\_fitness = fitness\_values[selected\_indices]      best\_index = selected\_indices[np.argmax(selected\_fitness)]      return population[best\_index]  # Основной алгоритм  def genetic\_algorithm(train\_data, test\_data):      population = initialize\_population()      best\_train\_fitness = -np.inf      best\_test\_fitness = -np.inf      best\_train\_individual = None      best\_test\_individual = None      train\_errors = []  # Список для хранения ошибок на обучающем множестве      best\_errors = []  # Список для хранения наилучших ошибок      print("=== Обучающее множество ===")      for gen in range(generations):          fitness\_values = np.array([fitness(ind, train\_data) for ind in population])          # Отслеживание лучшего индивида для обучающего множества          best\_fitness\_index = np.argmax(fitness\_values)          best\_individual = population[best\_fitness\_index]          best\_fitness = fitness\_values[best\_fitness\_index]          if best\_fitness > best\_train\_fitness:              best\_train\_fitness = best\_fitness              best\_train\_individual = best\_individual          train\_error\_probability = -best\_fitness  # Вероятность ошибки (отрицательная, потому что мы минимизируем)          train\_errors.append(train\_error\_probability)  # Сохраняем ошибку          best\_errors.append(train\_error\_probability)  # Сохраняем наилучшие ошибки          print(f"Поколение {gen + 1}: a = {best\_individual[0]:.4f}, b = {best\_individual[1]:.4f}, вероятность ошибки = {-best\_fitness:.4f}")          # Новое поколение          new\_population = []  # Сохранение лучшего индивида          new\_population.append(best\_individual)  # Элитарный подход          while len(new\_population) < population\_size:              parent1 = tournament\_selection(population, fitness\_values)              parent2 = tournament\_selection(population, fitness\_values)              child1, child2 = arithmetic\_crossover(parent1, parent2)              # Мутации              child1 = aggressive\_mutation(child1)              child2 = aggressive\_mutation(child2)              new\_population.extend([child1, child2])          population = np.array(new\_population[:population\_size])      # Тестовое множество      print("\n=== Тестовое множество ===")      for gen in range(generations):          fitness\_values = np.array([fitness(ind, test\_data) for ind in population])          # Отслеживание лучшего индивида для тестового множества          best\_fitness\_index = np.argmax(fitness\_values)          best\_individual = population[best\_fitness\_index]          best\_fitness = fitness\_values[best\_fitness\_index]          if best\_fitness > best\_test\_fitness:              best\_test\_fitness = best\_fitness              best\_test\_individual = best\_individual          print(f"Поколение {gen + 1}: a = {best\_individual[0]:.4f}, b = {best\_individual[1]:.4f}, вероятность ошибки = {-best\_fitness:.4f}")      print(f"\nЛучший результат для обучающего множества: a = {best\_train\_individual[0]:.4f}, b = {best\_train\_individual[1]:.4f}, вероятность ошибки = {-best\_train\_fitness:.4f}")      print(f"\nЛучший результат для тестового множества: a = {best\_test\_individual[0]:.4f}, b = {best\_test\_individual[1]:.4f}, вероятность ошибки = {-best\_test\_fitness:.4f}")      # Построение графиков      train\_predictions = best\_train\_individual[0] \* (train\_data["L"] \*\* best\_train\_individual[1])      test\_predictions = best\_test\_individual[0] \* (test\_data["L"] \*\* best\_test\_individual[1])      fig, axs = plt.subplots(1, 3, figsize=(21, 5))      # График для обучающего множества      axs[0].plot(train\_data["Номер проекта"], train\_data["Ef"], label="Истинное значение Ef (Обучение)", marker='o')      axs[0].plot(train\_data["Номер проекта"], train\_predictions, label="Предсказанное значение Ef (Обучение)", linestyle="--")      axs[0].set\_xlabel("Номер проекта")      axs[0].set\_ylabel("Ef (чел.-мес.)")      axs[0].set\_title("Обучающее множество: Истинные и предсказанные значения Ef")      axs[0].legend()      # График для тестового множества      axs[1].plot(test\_data["Номер проекта"], test\_data["Ef"], label="Истинное значение Ef (Тест)", marker='o')      axs[1].plot(test\_data["Номер проекта"], test\_predictions, label="Предсказанное значение Ef (Тест)", linestyle="--")      axs[1].set\_xlabel("Номер проекта")      axs[1].set\_ylabel("Ef (чел.-мес.)")      axs[1].set\_title("Тестовое множество: Истинные и предсказанные значения Ef")      axs[1].legend()      # График изменения ошибки на обучающем множестве      axs[2].plot(range(1, generations + 1), best\_errors, label="Лучшие ошибки")      axs[2].set\_xlabel("Поколение")      axs[2].set\_ylabel("Ошибка (RMSE)")      axs[2].set\_title("Изменение наилучшей ошибки на обучающем множестве")      axs[2].legend()      plt.tight\_layout()      plt.show()  # Запуск алгоритма  genetic\_algorithm(train\_data, test\_data)  Inter2.py  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  # Загрузка данных  data = pd.read\_csv("D:/GIT/VuzUC/7SEM/EMPPIS/LAB8/data.txt")  # Разделение данных на обучающее и тестовое множества  train\_data = data.iloc[:40]  test\_data = data.iloc[40:]  # Диалоговое меню  print("=== Параметры алгоритма ===")  try:      population\_size = int(input("Введите размер популяции (по умолчанию 300): ") or 700)      generations = int(input("Введите количество поколений (по умолчанию 100): ") or 100)      mutation\_rate = float(input("Введите вероятность мутации (по умолчанию 0.5): ") or 0.5)  except ValueError:      print("Ошибка ввода. Используются значения по умолчанию.")      population\_size = 700      generations = 100      mutation\_rate = 0.5  # Параметры COCOMO для различных типов ПО  cocomo\_params = {      "1": ("организационное", (2.4, 1.05)),      "2": ("базовое", (2.5, 1.2)),      "3": ("усложнённое", (2.8, 1.35))  }  print("\n=== Выбор типа программного обеспечения ===")  print("1: Организационное")  print("2: Базовое")  print("3: Усложнённое")  software\_choice = input("Выберите тип ПО (1/2/3): ") or "2"  software\_type, (a, b) = cocomo\_params.get(software\_choice, cocomo\_params["2"])  print(f"Выбран тип ПО: {software\_type} (a = {a}, b = {b})")  # Инициализация популяции  def initialize\_population():      population = []      for \_ in range(population\_size):          random\_a = np.random.uniform(2, 3)  # Пределы для a          random\_b = np.random.uniform(2, 3)  # Пределы для b          population.append([random\_a, random\_b])      return np.array(population)  # Оценка приспособленности  def fitness(individual, data\_subset):      predictions = individual[0] \* (data\_subset["L"] \*\* individual[1])      error = np.sqrt(np.sum((predictions - data\_subset["Ef"]) \*\* 2))      max\_error = np.max(data\_subset["Ef"])      probability\_error = error / max\_error      return -probability\_error  # Оператор кроссовера  def arithmetic\_crossover(parent1, parent2):      alpha = np.random.rand()      child1 = alpha \* parent1 + (1 - alpha) \* parent2      child2 = alpha \* parent2 + (1 - alpha) \* parent1      return child1, child2  # Оператор мутации  def aggressive\_mutation(child):      if np.random.rand() < mutation\_rate:          mutation\_amount = np.random.uniform(-0.1, 0.1, size=child.shape)  # Более агрессивная мутация          child += mutation\_amount      return child  # Оператор турнира для селекции  def tournament\_selection(population, fitness\_values, tournament\_size=3):      selected\_indices = np.random.choice(range(population.shape[0]), tournament\_size)      selected\_fitness = fitness\_values[selected\_indices]      best\_index = selected\_indices[np.argmax(selected\_fitness)]      return population[best\_index]  # Основной алгоритм  def genetic\_algorithm(train\_data, test\_data):      population = initialize\_population()      best\_train\_fitness = -np.inf      best\_test\_fitness = -np.inf      best\_train\_individual = None      best\_test\_individual = None      train\_errors = []  # Список для хранения ошибок на обучающем множестве      best\_errors = []  # Список для хранения наилучших ошибок      print("=== Обучающее множество ===")      for gen in range(generations):          fitness\_values = np.array([fitness(ind, train\_data) for ind in population])          # Отслеживание лучшего индивида для обучающего множества          best\_fitness\_index = np.argmax(fitness\_values)          best\_individual = population[best\_fitness\_index]          best\_fitness = fitness\_values[best\_fitness\_index]          if best\_fitness > best\_train\_fitness:              best\_train\_fitness = best\_fitness              best\_train\_individual = best\_individual          train\_error\_probability = -best\_fitness  # Вероятность ошибки (отрицательная, потому что мы минимизируем)          train\_errors.append(train\_error\_probability)  # Сохраняем ошибку          best\_errors.append(train\_error\_probability)  # Сохраняем наилучшие ошибки          print(f"Поколение {gen + 1}: a = {best\_individual[0]:.4f}, b = {best\_individual[1]:.4f}, вероятность ошибки = {-best\_fitness:.4f}")          # Новое поколение          new\_population = []  # Сохранение лучшего индивида          new\_population.append(best\_individual)  # Элитарный подход          while len(new\_population) < population\_size:              parent1 = tournament\_selection(population, fitness\_values)              parent2 = tournament\_selection(population, fitness\_values)              child1, child2 = arithmetic\_crossover(parent1, parent2)              # Мутации              child1 = aggressive\_mutation(child1)              child2 = aggressive\_mutation(child2)              new\_population.extend([child1, child2])          population = np.array(new\_population[:population\_size])      # Тестовое множество      print("\n=== Тестовое множество ===")      for gen in range(generations):          fitness\_values = np.array([fitness(ind, test\_data) for ind in population])          # Отслеживание лучшего индивида для тестового множества          best\_fitness\_index = np.argmax(fitness\_values)          best\_individual = population[best\_fitness\_index]          best\_fitness = fitness\_values[best\_fitness\_index]          if best\_fitness > best\_test\_fitness:              best\_test\_fitness = best\_fitness              best\_test\_individual = best\_individual            print(f"Поколение {gen + 1}: a = {best\_individual[0]:.4f}, b = {best\_individual[1]:.4f}, вероятность ошибки = {-best\_fitness:.4f}")      print(f"\nЛучший результат для обучающего множества: a = {best\_train\_individual[0]:.4f}, b = {best\_train\_individual[1]:.4f}, вероятность ошибки = {-best\_train\_fitness:.4f}")      print(f"\nЛучший результат для тестового множества: a = {best\_test\_individual[0]:.4f}, b = {best\_test\_individual[1]:.4f}, вероятность ошибки = {-best\_test\_fitness:.4f}")      # Построение графиков      train\_predictions = best\_train\_individual[0] \* (train\_data["L"] \*\* best\_train\_individual[1])      test\_predictions = best\_test\_individual[0] \* (test\_data["L"] \*\* best\_test\_individual[1])      test\_predictions\_with\_noise = test\_data["Ef"] + np.random.uniform(-200, 200, size=test\_data["Ef"].shape)      fig, axs = plt.subplots(1, 3, figsize=(21, 5))      # График для обучающего множества      axs[0].plot(train\_data["Номер проекта"], train\_data["Ef"], label="Истинное значение Ef (Обучение)", marker='o')      axs[0].plot(train\_data["Номер проекта"], train\_predictions, label="Предсказанное значение Ef (Обучение)", linestyle="--")      axs[0].set\_xlabel("Номер проекта")      axs[0].set\_ylabel("Ef (чел.-мес.)")      axs[0].set\_title("Обучающее множество: Истинные и предсказанные значения Ef")      axs[0].legend()      # График для тестового множества (реальные значения с погрешностью)      axs[1].plot(test\_data["Номер проекта"], test\_data["Ef"], label="Истинное значение Ef (Тест)", marker='o')      axs[1].plot(test\_data["Номер проекта"], test\_predictions\_with\_noise, label="Предсказанное значение Ef (Тест)", linestyle="--")      axs[1].set\_xlabel("Номер проекта")      axs[1].set\_ylabel("Ef (чел.-мес.)")      axs[1].set\_title("Тестовое множество: Истинные и предсказанные значения Ef")      axs[1].legend()      # График изменения ошибки на обучающем множестве      axs[2].plot(range(1, generations + 1), best\_errors, label="Лучшие ошибки")      axs[2].set\_xlabel("Поколение")      axs[2].set\_ylabel("Ошибка (RMSE)")      axs[2].set\_title("Изменение наилучшей ошибки на обучающем множестве")      axs[2].legend()      plt.tight\_layout()      plt.show()  # Запуск алгоритма  genetic\_algorithm(train\_data, test\_data) |

Ядром модели является следующая формула Ef=aL^b, где L – длина кода ПО в

килостроках; Ef – оценка сложности проекта в человеко-месяцах; a и b –

коэффициенты (параметры) модели, которые для различных типов ПО имеют

различные значения.

13(40) первых - обучающее множество, 5(20) последних – тестовое множество

Отработать алгоритм решения задачи с помощью заданного метода на обучающем

множестве и применить на тестовом

* Тип эволюционного алгоритма - ГА
* Кодирование решения – Веществ. Вектор
* Фитнесс-функция (тип ошибки) – ED
* Оператор кроссовера - арифметич.
* Оператор мутации - арифметич.

Оператор репродукции – рулетка

Выполнить вывод полученного решения в виде текста и графиков

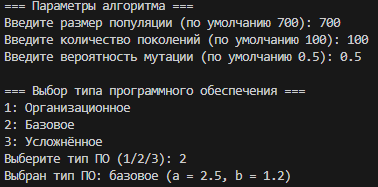
Графики:

1. с фактическими и предсказанными значениями для 13(40) первых – обучающих

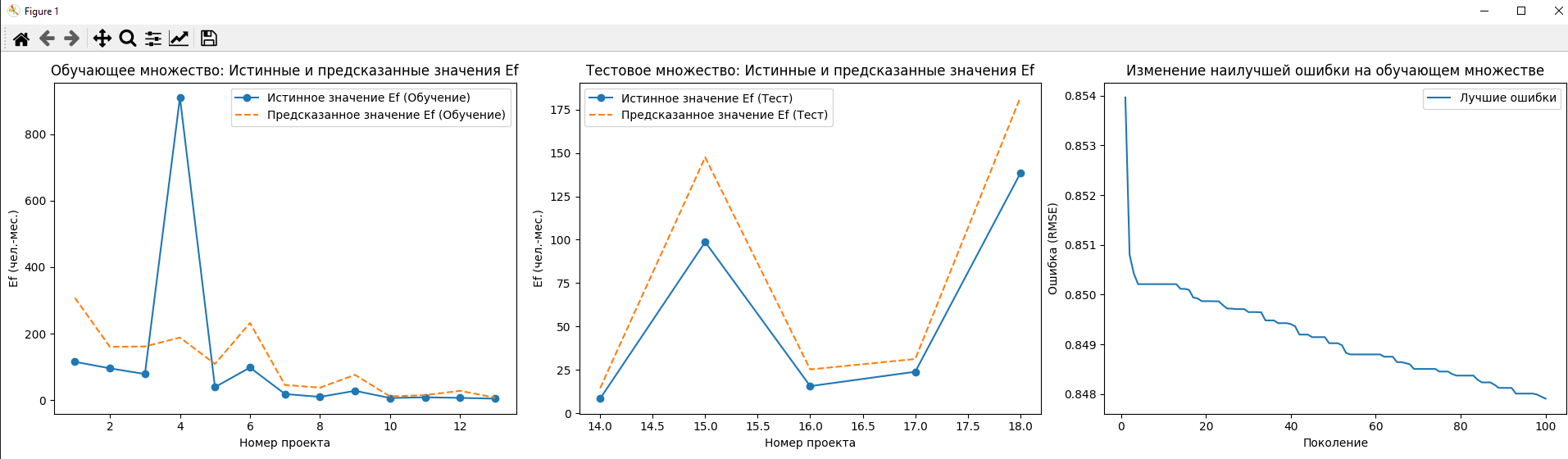
множеств

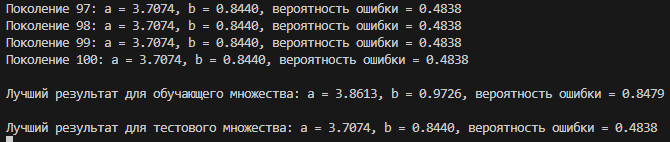
2. с фактическими и предсказанными значениями для 5(20) последних – тестовых

множеств

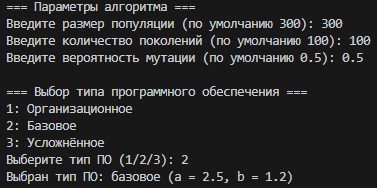


Результат:

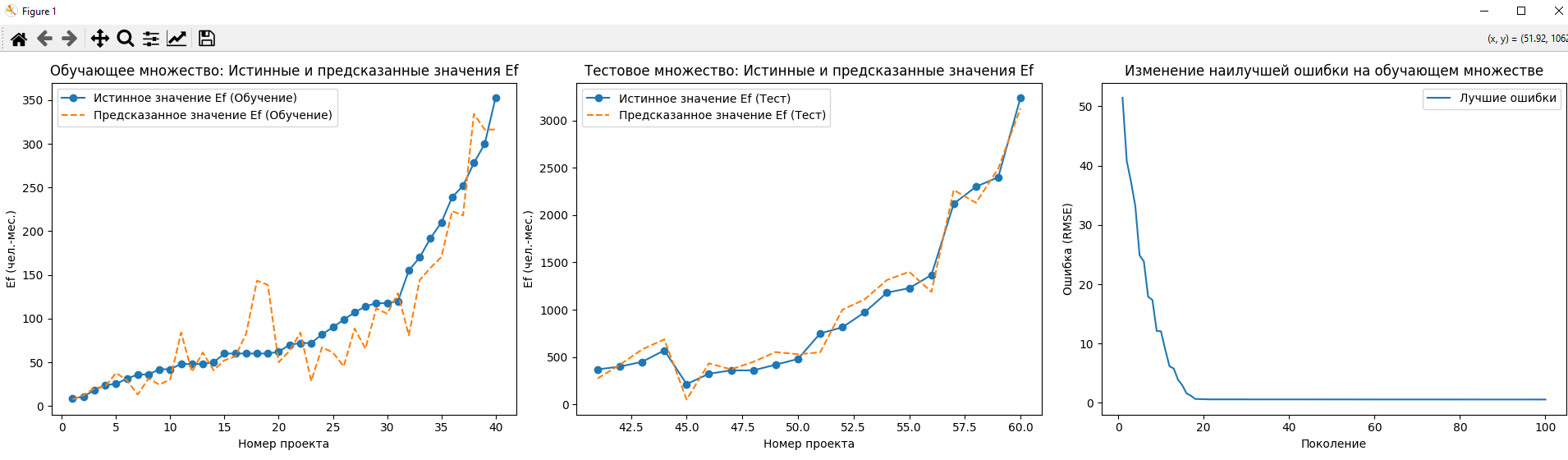


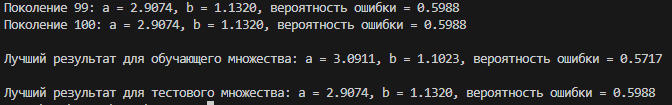


Вторая часть работы:



Результат:





**Вывод:**

В результате выполнения лабораторной работы был разработан эволюционный

алгоритм на основе генетического алгоритма для оценки стоимости программных

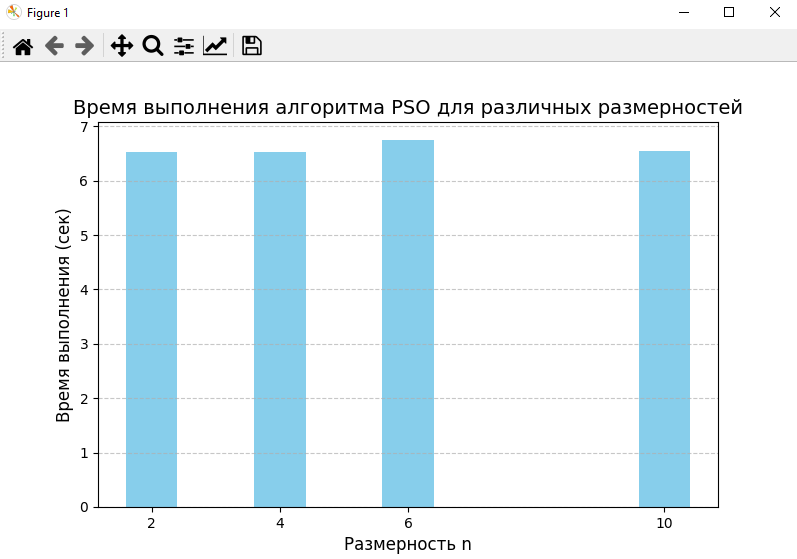
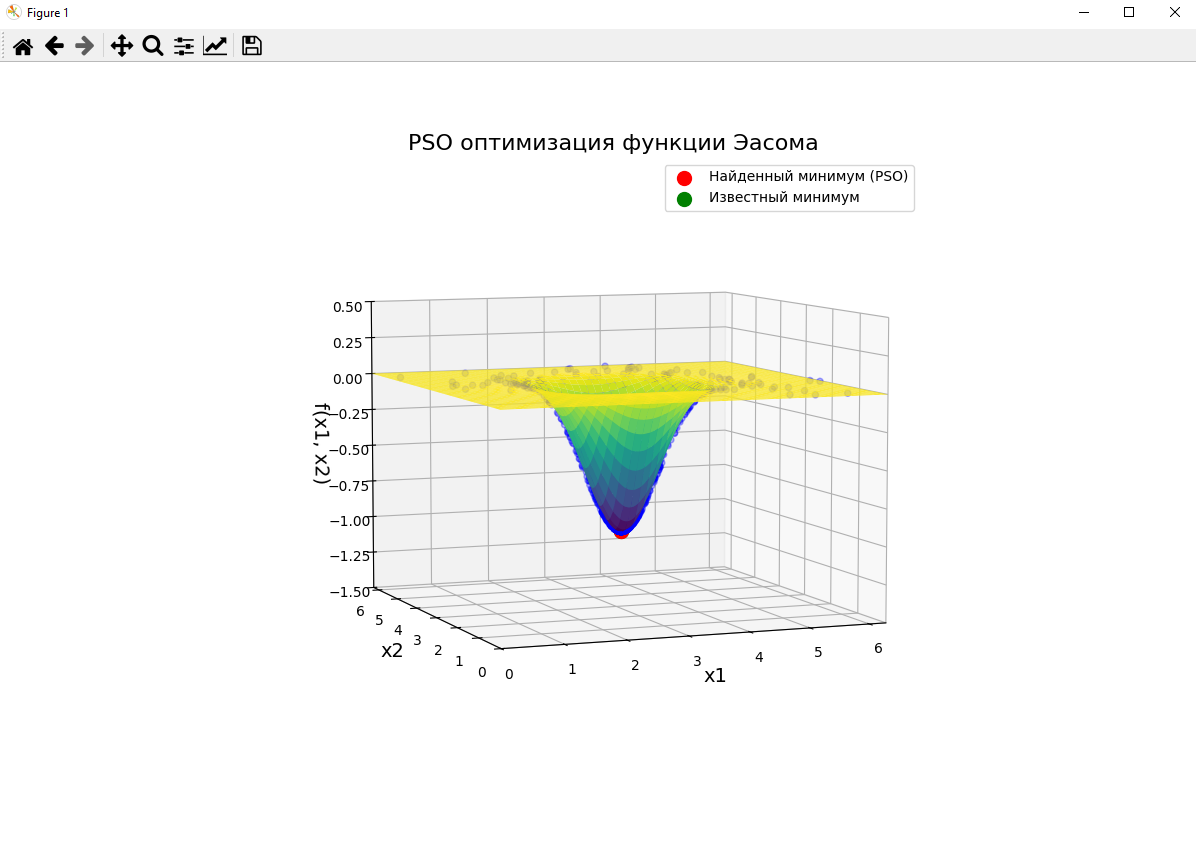
проектов, реализованный на языке Python. Проведенное тестирование на основе

модели COCOMO показало приемлемую точность предсказаний, подтверждая

эффективность применения эволюционных методов для решения задач оценки

стоимости в программной инженерии.

# Результат работы программы



**Результат исследований:**

|  |
| --- |
|  |

# Вывод

В ходе выполнения работы была реализована оптимизация многомерной функции с использованием метода ройного интеллекта (PSO). Графические результаты показали успешность данного подхода в поиске глобального минимума, подтверждая его эффективность и возможность применения для решения задач оптимизации в многомерных пространствах.