«Казанский национальный исследовательский технический

университет им А.Н. Туполева-КАИ» (КНИТУ-КАИ)

Всероссийская студенческая олимпиада с международным участием

«Управление и информатика в технических системах», 24 апреля 2026 г.

Задачи для номинации 4. Программирование на языке Python.

Рекомендации.

1) При разработке программ предусмотреть формирование входных данных (например, при помощи генератора случайных чисел). Чтобы программа была готова к работе с исходными данными.
2) Предусмотреть вывод результатов вычисления с информацией об идентификаторе переменой и значением, например: ***x=2.55*** .
3) Желательно в текст программ включать комментарии.

1 уровень

Задача № 1. (5 баллов)

Сформировать матрицу 5x5. Ввести все элементы. Найти строку с наибольшей суммой элементов и строку с наименьшей суммой элементов. Вывести на экран эти строки.

```
# Ввод матрицы 5x5
matrix = []
print("Введите элементы матрицы 5x5 (по одному числу в строке):")

for i in range(5):
    row = []
    for j in range(5):
        value = float(input(f"Введите элемент [{i+1}][{j+1}]: "))
        row.append(value)
    matrix.append(row)

# Поиск строк с наибольшей и наименьшей суммой
sums = [sum(row) for row in matrix]
max_sum = max(sums)
min_sum = min(sums)

# Индексы строк (с первой строки, то есть +1 для удобства пользователя)
max_row_index = sums.index(max_sum)
min_row_index = sums.index(min_sum)
```

```
# Вывод результатов
print("\nВведённая матрица:")
for row in matrix:
    print(" ".join(f"{x:6.2f}" for x in row))

print("\nСтрока с наибольшей суммой:", max_sum)
print(" ".join(f"{x:6.2f}" for x in matrix[max_row_index]))

print("\nСтрока с наименьшей суммой:", min_sum)
print(" ".join(f"{x:6.2f}" for x in matrix[min_row_index]))
```

Задача № 2. (5 баллов)

Задано уравнение окружности $(x-a)^2+(x-b)^2=R^2$. Параметры a, b и R ввести. Ввести координаты двух точек (A, B). Проверить, лежат ли они внутри окружности. Вывести на экран ответ.

```
# Ввод параметров окружности
a = float(input("Введите координату a центра окружности: "))
b = float(input("Введите координату b центра окружности: "))
R = float(input("Введите радиус R окружности: "))

# Ввод координат точки A
x1 = float(input("Введите координату x точки A: "))
y1 = float(input("Введите координату y точки A: "))

# Ввод координат точки B
x2 = float(input("Введите координату x точки B: "))
y2 = float(input("Введите координату y точки B: "))

# Функция для проверки, лежит ли точка внутри окружности
def is_point_inside_circle(x, y, a, b, R):
    # Вычисляем квадрат расстояния от точки до центра
    distance_squared = (x - a) ** 2 + (y - b) ** 2
    return distance_squared < R ** 2

# Проверка для точки A
A_inside = is_point_inside_circle(x1, y1, a, b, R)

# Проверка для точки B
B_inside = is_point_inside_circle(x2, y2, a, b, R)

# Вывод результатов
print("\nРезультаты проверки:")
print(f"Точка A({x1}, {y1}) — {'внутри' if A_inside else 'вне'} окружности.")
print(f"Точка B({x2}, {y2}) — {'внутри' if B_inside else 'вне'} окружности.")
```

Задача № 3. (5 баллов)

Анализ состояния батареи. Устройство получает значение напряжения аккумулятора (число от 0 до 12V). Определите статус зарядки: заряжено (11<V), разряд (V<9), средняя зарядка (9< V <11).

```
# Запрашиваем напряжение аккумулятора
voltage = float(input("Введите напряжение аккумулятора (0–12 В): "))

# Проверка диапазона и определение статуса
if voltage < 0 or voltage > 12:
    print("Ошибка: напряжение должно быть в диапазоне от 0 до 12 В.")
elif voltage > 11:
    print("Статус: Заряжено")
elif voltage < 9:
    print("Статус: Разряд")
else:
    print("Статус: Средняя зарядка")
```

Задача № 4. (5 баллов)

Разработать программу, вычислить значение параметра **y=f(x)** по формуле для значений **x**, изменяющихся от **a** до **b** с шагом **Δx=(b-a)/n** и вывести на экран график. (Функция y=sin(x), a=0, b=10, n=100).

```
import numpy as np
import matplotlib.pyplot as plt

# Параметры
a = 0
b = 10
n = 100

# Вычисление шага
dx = (b - a) / n

# Генерация массива x и вычисление y = sin(x)
x = np.arange(a, b + dx/2, dx)  # +dx/2 для включения b, если шаг не делит отрезок нацело
y = np.sin(x)

# Построение графика
plt.figure(figsize=(8, 4))
plt.plot(x, y, label='$y = \\sin(x)$', color='blue')
plt.title('График функции $y = \\sin(x)$')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Задача № 5. (10 баллов)

Дан сигнал — массив из n целых чисел. Примените сглаживание: каждый элемент второй строки (кроме первого и последнего) замените на среднее арифметическое его самого и двух соседей. Округлите результат вниз.
Входные данные: Первая строка: номер измерения. Вторая строка: n целых чисел.
Выходные данные: Массив после сглаживания.

```
import math

# Чтение входных данных
n = int(input())  # количество измерений
signal = list(map(int, input().split()))  # список из n целых чисел

# Создаём копию исходного сигнала, чтобы не изменять его при расчётах
smoothed = signal.copy()

# Применяем сглаживание для элементов со второго по предпоследний (индексы с 1 по n-2)
for i in range(1, n - 1):
    # Вычисляем среднее арифметическое текущего элемента и двух его соседей
    average = (signal[i - 1] + signal[i] + signal[i + 1]) / 3
    # Округляем результат вниз и сохраняем в новый массив
    smoothed[i] = math.floor(average)

# Выводим результат в одну строку через пробел
print(" ".join(map(str, smoothed)))
```

Задача № 6. (10 баллов)

Реализуйте дискретный пропорционально-интегральный-регулятор:
$u[k]=K_p \cdot e[k]+K_i \cdot \sum_{i=0}^{k} e[i]$,
где $e[k]$ — ошибка в момент k.
Входные данные: K_p, K_i, массив ошибок $e[0..n-1]$.
Выходные данные: Массив управляющих воздействий $u[0..n-1]$.

```
# Чтение входных данных
Kp = float(input())  # Пропорциональный коэффициент
Ki = float(input())  # Интегральный коэффициент

# Считываем массив ошибок.
# Ожидаем, что числа разделены пробелами в одной строке.
e = list(map(float, input().split()))
n = len(e)

# Инициализация массива для управляющих воздействий
u = [0.0] * n

# Переменная для хранения накопленной суммы ошибок (интеграла)
integral_sum = 0.0

# Основной цикл вычисления управляющего воздействия
for k in range(n):
    # Добавляем текущую ошибку к интегральной сумме
```

```
    integral_sum += e[k]

    # Вычисляем управляющее воздействие по формуле ПИ-регулятора
    u[k] = Kp * e[k] + Ki * integral_sum

# Вывод результата: управляющие воздействия через пробел
print(" ".join(f"{value:.6f}" for value in u))
```

Задача № 7. (10 баллов)

Реализуйте сценарий, в котором свет автоматически включается вечером и выключается утром, основываясь на текущих показаниях часов и минутах.

```
from datetime import datetime, time

# Настройка временных интервалов
# Свет включается, когда время >= ВРЕМЯ_ВКЛЮЧЕНИЯ или <
ВРЕМЯ_ВЫКЛЮЧЕНИЯ
# (например, с 18:00 до 07:00)
TIME_ON = time(18, 0)   # 18:00 (вечер)
TIME_OFF = time(7, 0)   # 07:00 (утро)

# Получение текущего времени
now = datetime.now().time()

# Проверка условия для включения света
# Сценарий работает в двух интервалах: [TIME_ON, 23:59] и [00:00, TIME_OFF)
if now >= TIME_ON or now < TIME_OFF:
    status = "ВКЛЮЧИТЬ"
else:
    status = "ВЫКЛЮЧИТЬ"

# Вывод результата
print(f"Текущее время: {now.strftime('%H:%M')}")
print(f"Действие: {status}")
```

2 уровень.

Задача № 8 (25 баллов)

Движение маятника описывается уравнением:
$\theta''(t) + b*\theta'(t) + \sin(\theta(t)) = u(t)$,
где $b = 0.2$. Начальные условия: $\theta(0) = 1$, $\theta'(0) = 0$.
Управление $u(t)$ кусочно-постоянное, меняется каждые $\Delta t = 0.1$ секунды. Цель — привести маятник в положение $\theta = 0$, $\theta' = 0$ как можно быстрее, не используя готовые решатели ОДУ (написать для метода Эйлера порядка самостоятельно).
Задача: подобрать закон управления $u(t)$ методом перебора или градиентного спуска, чтобы время стабилизации (когда $|\theta| < 0.01$ и $|\theta'| < 0.01$) было минимальным.
Ограничение: $|u(t)| \leq 5$.
Входные данные: Нет (задача — реализовать оптимизатор).
Выходные данные: 1) Последовательность $u(t)$ на каждом шаге до стабилизации, 2) Финальное время стабилизации, 3) График $\theta(t)$ и $\theta'(t)$ (или таблица).

```
import numpy as np
import matplotlib.pyplot as plt

# --- Параметры системы ---
b = 0.2
dt = 0.1
u_max = 5.0

# --- Параметры оптимизации ---
N_steps = 200  # Максимальное количество шагов управления (200 * 0.1с = 20с)
N_opt = 150    # Количество шагов градиентного спуска
lr = 0.05      # Скорость обучения (learning rate)

# --- Начальные условия ---
theta_0 = 1.0
omega_0 = 0.0

# Критерий стабилизации
def is_stable(theta, omega):
    return abs(theta) < 0.01 and abs(omega) < 0.01

# Симуляция системы (Метод Эйлера)
def simulate(u_vector, theta_0, omega_0, dt, max_steps):
    theta = theta_0
    omega = omega_0
    theta_log = [theta]
    omega_log = [omega]
    time_log = [0.0]

    # Массив управления u(t) для каждого шага симуляции
    # Если шагов симуляции больше, чем длина вектора u, повторяем последний элемент
    u_sim = np.concatenate([u_vector, np.full(max_steps - len(u_vector), u_vector[-1])])

    for k in range(max_steps):
        u = u_sim[k]
        # Метод Эйлера для системы второго порядка
        # d(omega)/dt = -b*omega - sin(theta) + u
        # d(theta)/dt = omega
        omega_new = omega + dt * (-b * omega - np.sin(theta) + u)
        theta_new = theta + dt * omega

        omega, theta = omega_new, theta_new

        time_log.append((k+1) * dt)
        theta_log.append(theta)
        omega_log.append(omega)

        if is_stable(theta, omega):
            break

    return np.array(time_log), np.array(theta_log), np.array(omega_log)
```

```
# --- Градиентный спуск ---
# Инициализируем управление (можно начать с нулевого управления)
u_opt = np.zeros(N_steps)
best_time = None
best_u = None

for iter in range(N_opt):
    # Симулируем систему с текущим управлением
    t, th, om = simulate(u_opt, theta_0, omega_0, dt, N_steps)

    # Находим время стабилизации для текущего управления
    stable_idx = np.where((np.abs(th) < 0.01) & (np.abs(om) < 0.01))[0]
    if len(stable_idx) > 0:
        current_time = t[stable_idx[0]]
    else:
        current_time = t[-1] # Если не стабилизировали, берем конечное время

    # Сохраняем лучшее найденное решение
    if best_time is None or current_time < best_time:
        best_time = current_time
        best_u = u_opt.copy()

    # Вычисляем градиент (приближенно)
    # Градиент по u[k] пропорционален влиянию этого управления на конечное состояние.
    # Мы хотим минимизировать конечное отклонение, поэтому градиент связан с
состоянием.
    grad = np.zeros_like(u_opt)
    for k in range(len(u_opt)):
        # Влияние управления на ускорение в момент k: d(omega_dot)/du = 1
        # Влияние на скорость (omega) в будущем: интегрируется через метод Эйлера.
        # Приближенно: вклад управления u[k] в конечное состояние пропорционален
времени его действия.
        # Используем упрощенный градиент: -omega в момент применения.
        grad[k] = -om[min(k, len(om)-1)]

    # Обновляем управление с помощью градиентного шага
    u_opt = u_opt - lr * grad

    # Проекция на допустимую область |u| <= u_max (Clip)
    u_opt = np.clip(u_opt, -u_max, u_max)

# Финальная симуляция с лучшим найденным управлением
t_final, th_final, om_final = simulate(best_u, theta_0, omega_0, dt, N_steps)
stable_idx_final = np.where((np.abs(th_final) < 0.01) & (np.abs(om_final) < 0.01))[0]
final_stabilization_time = t_final[stable_idx_final[0]] if len(stable_idx_final) > 0 else t_final[-1]

# --- Вывод результатов ---
print("--- Результаты оптимизации ---")
print(f"Финальное время стабилизации: {final_stabilization_time:.3f} сек")
print(f"Количество шагов до стабилизации: {stable_idx_final[0]}")
print("Пример первых 15 значений управления u(t):")
print(best_u[:15])
```

```
# Построение графиков
plt.style.use('seaborn-v0_8-whitegrid')
fig, axs = plt.subplots(2, 1, figsize=(12, 8), sharex=True)
fig.suptitle('Оптимальное управление маятником')

axs[0].plot(t_final, th_final, label='Угол θ(t)', color='steelblue')
axs[0].axhline(0.01, color='gray', linestyle='--', alpha=0.5)
axs[0].axhline(-0.01, color='gray', linestyle='--', alpha=0.5)
axs[0].set_ylabel('Угол θ (рад)')
axs[0].legend()
axs[0].grid(True)

axs[1].plot(t_final, om_final, label='Угловая скорость ω(t)', color='tomato')
axs[1].axhline(0.01, color='gray', linestyle='--', alpha=0.5)
axs[1].axhline(-0.01, color='gray', linestyle='--', alpha=0.5)
axs[1].set_xlabel('Время t (сек)')
axs[1].set_ylabel('Скорость ω (рад/сек)')
axs[1].legend()
axs[1].grid(True)

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

Задача № 9 (25 баллов)

Реализуйте шифр Виженера. Дан текст (только заглавные латинские буквы) и ключевое слово. Каждая буква текста сдвигается на величину, определяемую соответствующей буквой ключа (A=0, B=1, ...). Ключ повторяется циклически.

Входные данные: Первая строка: ключевое слово. Вторая строка: текст для шифрования.

Выходные данные: Зашифрованная строка.

```
def vigenere_cipher(key: str, text: str) -> str:
    """
    Шифрует текст с помощью шифра Виженера.
    Ожидает, что ключ и текст состоят из заглавных латинских букв.
    """
    result = []
    key_length = len(key)

    for i, char in enumerate(text):
        # Получаем сдвиг от текущего символа ключа
        # (i % key_length) обеспечивает циклическое повторение ключа
        shift = ord(key[i % key_length]) - ord('A')

        # Сдвигаем символ текста и добавляем в результат
        # ord('A') + (ord(char) - ord('A') + shift) % 26 — формула сдвига с циклом по алфавиту
        encrypted_char = chr(ord('A') + (ord(char) - ord('A') + shift) % 26)
        result.append(encrypted_char)

    return ''.join(result)

# Чтение входных данных
key = input().strip()  # Ключевое слово
```

```
text = input().strip() # Текст для шифрования

# Вывод зашифрованного текста
print(vigenere_cipher(key, text))
```