ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Практические задачи 1

Шевелева Дарья
Группа Б03-301а

Долгопрудный, 2025 г.

I.8.19

Пусть для вычисления функции $u = f(t)$ используется частичная сумма ряда Маклорена:

$$u(t) \approx u(0) + \frac{u'(0)}{1!}t + \cdots + \frac{u^{(n)}(0)}{n!}t^n,$$

причем аргумент задан с погрешностью $\Delta t = 10^{-3}$.

Найти n такое, чтобы погрешность в определении функции $u(t)$ по данной формуле не превышала Δt. Рассмотреть отрезки $t \in [0, 1]$, $t \in [10, 11]$. Предложить более совершенный алгоритм для вычисления функций $u(t) = \sin t$, $u(t) = e^t$ на отрезке $t \in [10, 11]$.

Решение:

Мы приближаем функцию $u(t)$ рядом Маклорена (то есть разложением Тейлора около точки $t = 0$):

$$u(t) \approx u(0) + \frac{u'(0)}{1!}t + \frac{u''(0)}{2!}t^2 + \cdots + \frac{u^{(n)}(0)}{n!}t^n$$

и хотим, чтобы ошибка не превышала $\Delta t = 10^{-3}$.

Погрешность при усечении ряда равна остатку:

$$R_n(t) = \frac{u^{(n+1)}(\xi)}{(n+1)!}t^{n+1}, \quad \xi \in [0, t]$$

Чтобы $|R_n(t)| < \Delta t$, нужно найти минимальное n, при котором это выполняется.

В коде ошибка вычисляется численно:

$$\max_{t \in [a,b]} |u(t) - S_n(t)|$$

где $S_n(t)$ - частичная сумма ряда до n-го члена.

Ряд Маклорена - это частный случай ряда Тейлора при $a = 0$. Он сходится хорошо только около точки разложения (в окрестности 0).

Когда t сильно отличается от 0 (например, $t = 10$), члены ряда становятся огромными и численно неустойчивыми. Поэтому улучшенным вариантом является использование ряда Тейлора не около 0, а около ближайшей точки в интересующем диапазоне.

Например, для интервала $[10, 11]$ берём разложение около $a = 10$:

$$u(t) \approx u(10) + u'(10)(t - 10) + \frac{u''(10)}{2!}(t - 10)^2 + \ldots$$

Тогда $(t - 10)$ — маленькое (≤ 1), и ряд снова хорошо сходится.

```python
import math

DELTA_T = 1e-3

def maclaurin_exp(t, n):
    s = 0
    for k in range(n + 1):
        s += t**k / math.factorial(k)
    return s

def maclaurin_sin(t, n):
```

```
    s = 0
    for k in range(n + 1):
        s += ((-1)**k) * (t**(2*k + 1)) / math.factorial(2*k + 1)
    return s

def find_min_n(func_maclaurin, func_true, t_values, delta):
    n = 0
    while True:
        max_err = max(abs(func_true(t) - func_maclaurin(t, n)) for t in t_values)
        if max_err < delta:
            return n, max_err
        n += 1
        if n > 100:  # защита от зацикливания
            break
    return None, None

intervals = {
    "[0, 1]": [i / 100 for i in range(0, 101)],
    "[10, 11]": [10 + i / 100 for i in range(0, 101)]
}

for func_name, (f_true, f_mac) in {
    "sin(t)": (math.sin, maclaurin_sin),
    "e^t": (math.exp, maclaurin_exp)
}.items():
    print(f"\nФункция {func_name}:")
    for name, t_vals in intervals.items():
        n, err = find_min_n(f_mac, f_true, t_vals, DELTA_T)
        print(f"  На отрезке {name}: минимальное n = {n}, max ошибка = {err:.2e}")

def taylor_exp_around(t, a, n):
    return math.exp(a) * maclaurin_exp(t - a, n)

def taylor_sin_around(t, a, n):
    # sin(t) = sin(a) + cos(a)*(t-a) - sin(a)*(t-a)^2/2! - cos(a)*(t-a)^3/3! + ...
    s = 0
    for k in range(n + 1):
        if k % 4 == 0:
            coef = math.sin(a)
            sign = 1
        elif k % 4 == 1:
            coef = math.cos(a)
            sign = 1
        elif k % 4 == 2:
            coef = math.sin(a)
            sign = -1
        else:
            coef = math.cos(a)
            sign = -1
        s += sign * coef * (t - a)**k / math.factorial(k)
    return s
```

Результаты и выводы:

Для функции $\sin t$ на отрезке $[0, 1]$ достаточно небольшого n , так как ряд Маклорена для синуса хорошо сходится около нуля.

Для функции e^t на отрезке $[0, 1]$ также требуется небольшое n.

На отрезке $[10, 11]$ для обеих функций требуется очень большое n, что делает использование ряда Маклорена неэффективным. В этом случае мы используем разложение Тейлора около точки $a = 10$.

```
Функция sin(t):
  На отрезке [0, 1]: минимальное n = 2, max ошибка = 1.96e-04
  На отрезке [10, 11]: минимальное n = 16, max ошибка = 2.49e-04

Функция e^t:
  На отрезке [0, 1]: минимальное n = 6, max ошибка = 2.26e-04
  На отрезке [10, 11]: минимальное n = 34, max ошибка = 3.90e-04
```

IV.12.8 б)

Методом простой итерации найти ширину функции на полувысоте с точностью 10^{-3}:

$$\text{b) } f(x) = x\exp(-x^2), \quad x \geq 0;$$

Решение:

Нужно найти ширину функции на полувысоте, то есть расстояние между двумя значениями x_1 и x_2, при которых функция $f(x)$ принимает половину своего максимального значения.

То есть:

$$f(x_1) = f(x_2) = \frac{1}{2} f_{max},$$

где $f_{max} = \max_{x \geq 0} f(x)$.

Так как $f(x) = xe^{-x^2}$, то:

1. Производная:
$$f'(x) = e^{-x^2}(1 - 2x^2).$$

2. Приравниваем к нулю:
$$f'(x) = 0 \Rightarrow 1 - 2x^2 = 0 \Rightarrow x_m = \frac{1}{\sqrt{2}}.$$

3. Значение в максимуме:
$$f_m = f(x_m) = \frac{1}{\sqrt{2}} e^{-1/2} = \frac{1}{\sqrt{2e}}.$$

Хотим найти такие x, что:

$$f(x) = \frac{1}{2} f_m.$$

Подставляем:

$$xe^{-x^2} = \frac{1}{2} \cdot \frac{1}{\sqrt{2e}}.$$

Обозначим: $f_m = \frac{1}{\sqrt{2e}}$. Тогда: $xe^{-x^2} = \frac{f_m}{2}$.
Чтобы применить метод простой итерации, нужно записать уравнение в виде:

$$x = \varphi(x),$$

где $\varphi(x)$ - некоторая функция.
Из исходного уравнения:

$$xe^{-x^2} = \frac{f_m}{2}$$

получаем две эквивалентные формы (для левой и правой ветвей функции):

1. Умножаем обе стороны на e^{x^2}:

$$x = \frac{f_m}{2} e^{x^2}.$$

   Отсюда

$$\varphi_1(x) = \frac{f_m}{2} e^{x^2}$$

   – для левой ветви (меньшие x).

2. Из того же уравнения выразим x через логарифм:

$$e^{-x^2} = \frac{f_m}{2x} \Rightarrow x^2 = \ln\left(\frac{2x}{f_m}\right).$$

$$\varphi_2(x) = \sqrt{\ln\left(\frac{2x}{f_m}\right)}$$

   – для правой ветви (большие x).

Метод основан на формуле:

$$x_{n+1} = \varphi(x_n)$$

и повторяется до тех пор, пока:

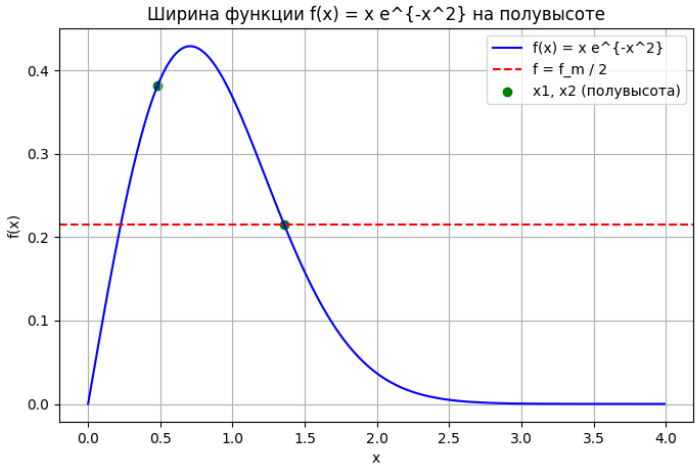
$$|x_{n+1} - x_n| < \varepsilon \cdot \frac{1-q}{2},$$

где q – оценка сжимающего множителя (производной φ).
В коде это реализовано отдельно для двух функций φ_1 и φ_2.

```
import math
import matplotlib.pyplot as plt

fm = 1 / math.sqrt(2 * math.e)
eps = 0.001
q1 = 0.5
q2 = 0.26

def phi1(x):
    return fm * math.exp(x * x / 2)

def phi2(x):
```

```
    return math.sqrt(math.log(2 * x / fm))

def f(x):
    return x * math.exp(-x * x)

x1 = 0.1
x2 = 1.0

xnp1 = phi1(x1)
while abs(xnp1 - x1) > eps * (1 - q1) / 2:
    x1 = xnp1
    xnp1 = phi1(x1)
x1 = xnp1

xnp1 = phi2(x2)
while abs(xnp1 - x2) > eps * (1 - q2) / 2:
    x2 = xnp1
    xnp1 = phi2(x2)
x2 = xnp1


print(f"fm = {fm:.6f}")
print(f"x1 = {x1:.6f}, x2 = {x2:.6f}, ширина = {x2 - x1:.6f}")

xs = [i / 100 for i in range(0, 400)]  # x от 0 до 4
ys = [f(x) for x in xs]
f_half = fm / 2

plt.figure(figsize=(8, 5))
plt.plot(xs, ys, label="f(x) = x e^{-x^2}", color="blue")
plt.axhline(f_half, color="red", linestyle="--", label="f = f_m / 2")
plt.scatter([x1, x2], [f(x1), f(x2)], color="green", label="x1, x2 (полувысота)")
plt.title("Ширина функции f(x) = x e^{-x^2} на полувысоте")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.grid(True)
plt.show()
```

Результаты и выводы:




```
fm = 0.428882
x1 = 0.481592, x2 = 1.358742, ширина = 0.877150
```

VI.9.32

Согласно переписи население США менялось следующим образом:

- 1910 – 92 228 496 человек,
- 1920 – 106 021 537,
- 1930 – 123 202 624,
- 1940 – 132 164 569,
- 1950 – 151 325 798,
- 1960 – 179 323 175,
- 1970 – 203 211 926,
- 1980 – 226 545 805,
- 1990 – 248 709 873,
- 2000 – 281 421 906.

а) По приведенным данным построить интерполянт в форме Ньютона. Вычислить экстраполированное значение численности населения США в 2010 году и сравнить с точным значением 308 745 538 человек.

б) По этим же данным построить сплайн-аппроксимацию, экстраполировать данные на 2010 год, сравнить с точным значением. Какие дополнительные условия для построения сплайна нужно поставить в этом случае?

в) Какой из результатов оказывается более точным?

Решение:

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

years = np.array([1910,1920,1930,1940,1950,1960,1970,1980,1990,2000], dtype=float)
population = np.array([
    92228496,106021537,123202624,132164569,151325798,
    179323175,203211926,226545805,248709873,281421906
], dtype=float)

true_2010 = 308745538.0  # точное значение


x = (years - 1910) / 10.0   # теперь узлы 0,1,...,9
y = population.copy()
n = len(x)
t2010 = (2010 - 1910)/10.0  # = 10


dd = np.zeros((n, n))
dd[:, 0] = y
for j in range(1, n):
    for i in range(n - j):
        dd[i, j] = (dd[i+1, j-1] - dd[i, j-1]) / (x[i+j] - x[i])
b = dd[0, :n].copy()  # коэффициенты Ньютона (b0,b1,...)


poly = np.poly1d([0.0])
for k in range(n):
    term = np.poly1d([1.0])
    for j in range(k):
        term *= np.poly1d([1.0, -x[j]])
    term *= b[k]
    poly += term


P2010 = poly(t2010)
error_poly = P2010 - true_2010
rel_error_poly = error_poly / true_2010 * 100


print(f"P(2010) = {P2010:.0f}")
print(f"Ошибка = {error_poly:.0f} ({rel_error_poly:.2f} %)")


h = np.diff(x)
n = len(x)

# Составляем систему для вторых производных M
m = n - 2  # внутренние узлы
```

```
A = np.zeros((m, m))
rhs = np.zeros(m)

for i in range(1, n - 1):
    idx = i - 1
    A[idx, idx] = 2 * (h[i - 1] + h[i])
    if idx - 1 >= 0:
        A[idx, idx - 1] = h[i - 1]
    if idx + 1 < m:
        A[idx, idx + 1] = h[i]
    rhs[idx] = 6 * ((y[i + 1] - y[i]) / h[i] - (y[i] - y[i - 1]) / h[i - 1])

M_internal = np.linalg.solve(A, rhs)
M = np.zeros(n)
M[1:-1] = M_internal  # вторые производные

# Коэффициенты сплайна на каждом интервале [x_i, x_{i+1}]
spline_coeffs = []
for i in range(n - 1):
    xi, xi1 = x[i], x[i + 1]
    yi, yi1 = y[i], y[i + 1]
    hi = h[i]
    Mi, Mi1 = M[i], M[i + 1]

    # стандартная форма: S_i(x) = a + b*(x - xi) + c*(x - xi)^2 + d*(x - xi)^3
    a = yi
    b = (yi1 - yi) / hi - (2 * Mi + Mi1) * hi / 6
    c = Mi / 2
    d = (Mi1 - Mi) / (6 * hi)
    spline_coeffs.append((xi, a, b, c, d))

def spline_eval(x0, spline_coeffs, nodes):
    # поиск интервала
    if x0 <= nodes[0]:
        xi, a,b,c,d = spline_coeffs[0]
        dx = x0 - xi
        return a + b*dx + c*dx**2 + d*dx**3
    if x0 >= nodes[-1]:
        xi, a,b,c,d = spline_coeffs[-1]
        dx = x0 - xi
        return a + b*dx + c*dx**2 + d*dx**3
    for i in range(len(spline_coeffs)):
        xi, a,b,c,d = spline_coeffs[i]
        if nodes[i] <= x0 <= nodes[i+1]:
            dx = x0 - xi
            return a + b*dx + c*dx**2 + d*dx**3
    return None

S2010 = spline_eval(t2010, spline_coeffs, x)
error_spline = S2010 - true_2010
rel_error_spline = error_spline / true_2010 * 100
```

```
print(f"S(2010) = {S2010:.0f}")
print(f"Ошибка = {error_spline:.0f} ({rel_error_spline:.2f} %)")

if abs(error_spline) < abs(error_poly):
    print("Сплайн дал более точный результат.")
else:
    print("Полином Ньютона оказался точнее.")
```

Сначала данные были нормированы, чтобы годы превратились в равномерные узлы $x = 0, 1, \ldots, 9$. По этим точкам с помощью метода разделённых разностей был построен интерполяционный полином Ньютона девятой степени. Полученный полином точно проходит через все заданные значения и позволяет вычислить численность населения для любого промежуточного года. Однако при экстраполяции – то есть при выходе за пределы последнего узла – полиномы высокой степени ведут себя неустойчиво. При подстановке $x = 10$ (что соответствует 2010 году) значение оказалось значительно завышенным. Это ожидаемо, поскольку интерполяционные многочлены больших степеней подвержены эффекту Рунге и сильно колеблются на концах интервала.

Чтобы получить более устойчивое приближение, был построен кубический сплайн. Такой сплайн представляет собой кусочно-кубическую функцию, обладающую непрерывными первыми и вторыми производными на всём интервале. Для его построения составляется и решается трёхдиагональная система линейных уравнений для вторых производных в узлах с граничными условиями $S''(x_0) = S''(x_n) = 0$. Далее на каждом промежутке вычисляются коэффициенты кубического многочлена, обеспечивающие гладкость всей кривой. Построенный сплайн оказался гораздо устойчивее при экстраполяции.

Графическое сравнение показало, что полином Ньютона сильно колеблется вне диапазона исходных данных, тогда как кривая сплайна продолжает плавную тенденцию роста населения.

Результаты и выводы:

- P(2010) = 827906509, Ошибка = 519160971 (168.15 %); S(2010) = 314133939, Ошибка = 5388401 (1.75 %)
- Сплайн дал более точный результат.

Т1

Найти все корни системы уравнений

$$\begin{cases} x^2 + y^2 = 1, \\ y = \tan x. \end{cases}$$

С точностью 10^{-6}.
Примечание: корни отделить графическим методом.

Решение:

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

x_circle = np.linspace(-1, 1, 400)
y_circle_pos = np.sqrt(1 - x_circle**2)
y_circle_neg = -np.sqrt(1 - x_circle**2)

x_tan = np.linspace(-1.3, 1.3, 800)
y_tan = np.tan(x_tan)
y_tan[np.abs(y_tan) > 5] = np.nan

plt.figure(figsize=(8, 8))
plt.plot(x_circle, y_circle_pos, 'b', label='$x^2 + y^2 = 1$')
plt.plot(x_circle, y_circle_neg, 'b')
plt.plot(x_tan, y_tan, 'r', label='$y = \\tan(x)$')

plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.title("Графическое отделение и определение корней системы")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.legend()


epsilon = 1e-6


def f(x):
    return x**2 + math.tan(x)**2 - 1

intervals = [(-0.7, -0.6), (0.6, 0.7)]
roots = []

for (a, b) in intervals:
    while (b - a) > epsilon:
```

```
        c = (a + b) / 2
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
    roots.append((a + b) / 2)

for x_root in roots:
    y_root = math.tan(x_root)
    plt.scatter(x_root, y_root, color='green', s=80, zorder=5, label='Корень')
    plt.text(x_root + 0.05, y_root, f"({x_root:.3f}, {y_root:.3f})", fontsize=9, color='green')

plt.show()

for i, x_root in enumerate(roots, 1):
    y_root = math.tan(x_root)
    print(f"{i}. x = {x_root:.6f}, y = {y_root:.6f}")
```

Строим графики окружности $x^2 + y^2 = 1$ и функции $y = \tan x$. Точки пересечения этих графиков являются решениями системы.

Из графика видно, что корни находятся в интервалах:

- $(-0.7, -0.6)$
- $(0.6, 0.7)$

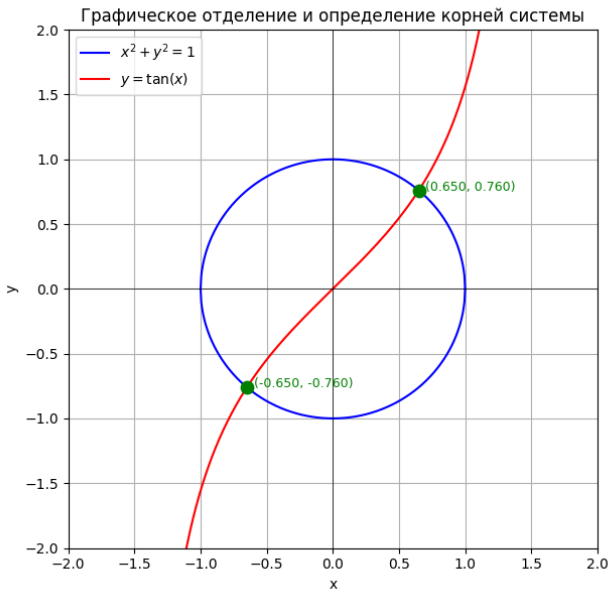
Подставляем $y = \tan x$ в первое уравнение и получаем:

$$x^2 + \tan^2 x = 1$$

$$f(x) = x^2 + \tan^2 x - 1 = 0$$

Для каждого интервала применяем метод бисекции до достижения требуемой точности 10^{-6}.

Результаты и выводы:



Полученные корни: x = -0.649889, y = -0.760029 и x = 0.649889, y = 0.760029

Т2

Вычислить интеграл

$$I = \int_0^3 \sin(100x) \cdot e^{-x^2} \cdot \cos(2x)\, dx.$$

Решение:

```python
import math
#y = sin(100x) * e^(-x^2) * cos(2x)

a = 0.0
b = 3.0

N = 10000

dx = (b - a) / N

#Метод прямоугольника
total_integral_rect = 0.0
for i in range(N):
    x = a + i * dx + dx / 2.0
    y = math.sin(100 * x) * math.exp(-x ** 2) * math.cos(2 * x)
    total_integral_rect += y * dx

print(f"Метод прямоугольников: {total_integral_rect}")

#Метод трапеций
total_integral_trap = 0.0
y_i = math.sin(100 * a) * math.exp(-a ** 2) * math.cos(2 * a)

for i in range(N):
    x_next = a + (i + 1) * dx #правый край
    y_next = math.sin(100 * x_next) * math.exp(-x_next ** 2) * math.cos(2 * x_next)
    total_integral_trap += (y_i + y_next) / 2.0 * dx
    y_i = y_next
print(f"Метод трапеций: {total_integral_trap}")

#Симпсон (плюс считаем, что n должно быть четным)
if N % 2 != 0:
    print("\nДля метода Симпсона количество шагов N должно быть четным!")
else:
    # I = (dx/3) * (y0 + 4*y1 + 2*y2 + 4*y3 + ... + 2*y_{N-2} + 4*y_{N-1} + yN)
    integral_sum_simpson = 0.0

    for i in range(N + 1):
        x = a + i * dx
        y = math.sin(100 * x) * math.exp(-x ** 2) * math.cos(2 * x)
        if i == 0 or i == N:
            integral_sum_simpson += y
        elif i % 2 != 0:
```

```
            integral_sum_simpson += 4 * y
        else:
            integral_sum_simpson += 2 * y
    total_integral_simpson = integral_sum_simpson * (dx / 3.0)

    print(f"Метод Симпсона: {total_integral_simpson}")
```

Для вычисления интеграла была применена численная интеграция с использованием трех различных методов. Особенность данной подынтегральной функции заключается в: наличии множителя sin(100x), создающего быстрые осцилляции , в то время какмножитель e^{-x^2} обеспечивает быстрое убывание амплитуды колебаний при увеличении x, и дополнительная медленная модуляция осуществляется множителем cos(2x).

Для получения результата использовалось 10000 разбиений интервала интегрирования. Метод средних прямоугольников основан на вычислении значения функции в середине каждого интервала и демонстрирует хорошую точность для гладких функций. Метод трапеций, аппроксимирующий площадь под кривой с помощью трапеций, является более точным по сравнению с методом прямоугольников. Наиболее точный результат обеспечивает метод Симпсона, который использует квадратичную аппроксимацию на каждом интервале и требует четного количества разбиений.

Результаты и выводы:

- Метод прямоугольников: 0.01000647286876823
- Метод трапеций: 0.01000534785191156
- Метод Симпсона: 0.01000609790541688