ВЫЧИСЛИТЕЛЬНЫЕ МЕТОДЫ · ЛЕКЦИЯ 3

Основные численные методы

Ошибки, корни, линейные системы, интерполяция, дифференцирование и интегрирование

Лю Шисян

Кафедра теплофизики (Э6)
МГТУ им. Н.Э. Баумана
Москва / 2026

Место этой лекции в курсе

Лекция 3 — Основные численные методы

Численные методы позволяют получать приближённые решения математических задач с помощью конечной последовательности вычислений.

После лекции вы сможете:

- оценивать абсолютную и относительную погрешности;
- находить корни нелинейных уравнений;
- решать системы линейных алгебраических уравнений;
- выполнять интерполяцию, численное дифференцирование и интегрирование.

Теперь мы переходим от вычислительной среды к самим алгоритмам.

План

TOC

1

Численные вычисления и погрешности

Когда нужен численный метод?

Аналитическое решение

Решение можно представить в явном виде с помощью формулы.
Например:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

- результат получается напрямую;
- удобно исследовать влияние параметров;
- дополнительный алгоритм не требуется.

Численное решение

Для многих задач удобной аналитической формулы не существует.
Например:

$$\cos x - x = 0.$$

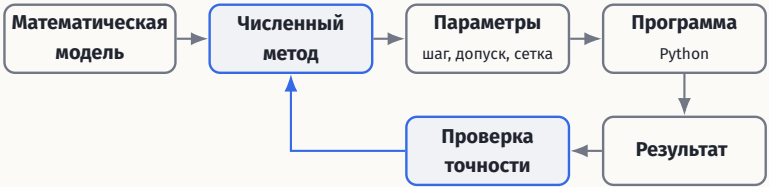
Решение строится последовательностью вычислений:

$$x_0 \to x_1 \to x_2 \to \cdots \to \tilde{x}.$$

Необходимо выбрать метод, критерий остановки и требуемую точность.

Численный метод заменяет недоступную формулу вычислительным алгоритмом.

Типичный вычислительный процесс



Почему нужна проверка?

Численный результат зависит не только от математической модели, но и от выбранного метода и его параметров. При недостаточной точности расчёт необходимо повторить.

Число на экране ещё не означает правильный результат.

Абсолютная и относительная погрешность

Пусть x — точное значение, а $\tilde{x}$ — его численное приближение.

Абсолютная погрешность

$$\Delta x = |x - \tilde{x}|.$$

Показывает абсолютное отклонение приближённого значения от точного.

Имеет те же единицы измерения, что и величина x.

Относительная погрешность

При $x \neq 0$:

$$\delta x = \frac{|x - \tilde{x}|}{|x|}.$$

В процентах:

$$\delta_{\%} = 100 \frac{|x - \tilde{x}|}{|x|} \%.$$

Она характеризует погрешность относительно масштаба величины.

Откуда возникает погрешность?

Модель и данные

Исходные данные могут быть неточными, а математическая модель — упрощать реальную систему.

реальность $\rightarrow$ модель

Аппроксимация

Точная математическая операция заменяется приближённой.
Например:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Округление

Компьютер хранит числа с конечной точностью.
Поэтому арифметические операции также выполняются приближённо.

Точность результата определяется всей цепочкой вычислений.

Ограниченная точность компьютера

Числа с плавающей точкой

Компьютер хранит вещественные числа с конечным количеством битов.
Поэтому многие десятичные числа представляются только приближённо.

```
x = 0.1 + 0.2

print(x)
# 0.30000000000000004

print(x == 0.3)
# False
```

Как сравнивать числа?

Для чисел типа `float` обычно проверяют не строгое равенство, а близость значений.

```
import numpy as np

np.isclose(0.1 + 0.2, 0.3)
# True
```

Для `float64`

Обычно доступно около 15–16 значащих десятичных цифр.

Погрешность округления является частью любого численного вычисления.

Аппроксимация и выбор шага h

Численная производная

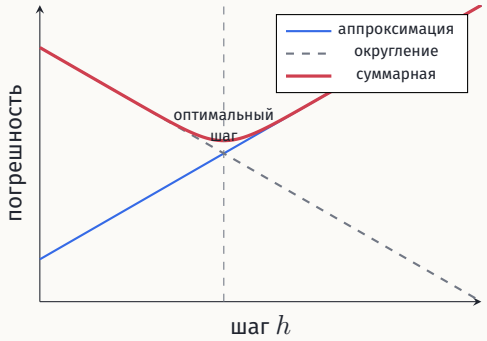
Производную можно приближённо вычислить с помощью конечной разности:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

При уменьшении h **погрешность аппроксимации уменьшается**.

Но слишком малый шаг тоже опасен

При очень малом h значения $f(x+h)$ и $f(x)$ становятся близкими.
При их вычитании возрастает влияние **погрешности округления**.



Слишком большой h даёт погрешность аппроксимации, а слишком малый — усиливает погрешность округления.

2

Корни нелинейных уравнений

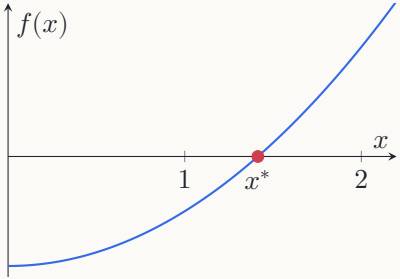
Задача поиска корня

Постановка задачи

Для заданной функции $f(x)$ необходимо найти такое значение x^*, что

$$\boxed{f(x^*) = 0}.$$

Такое значение x^* называется **корнем уравнения**.



Пример:

$$f(x) = x^2 - 2.$$

Тогда положительный корень:

$$x^* = \sqrt{2} \approx 1.4142.$$

Геометрически корень — это точка пересечения графика $f(x)$ с осью x.

Метод бисекции

Пусть функция непрерывна на отрезке $[a, b]$ и значения на концах имеют разные знаки:

$$f(a)\, f(b) < 0.$$

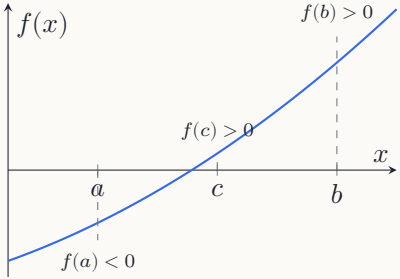
Тогда на интервале $[a, b]$ существует хотя бы один корень.

На каждом шаге берём середину интервала:

$$c = \frac{a + b}{2}.$$

Далее вычисляем $f(c)$ и оставляем ту половину, в которой сохраняется смена знака:

$$[a, b] \to [a, c] \quad \text{или} \quad [c, b].$$



Смысл метода

Интервал, содержащий корень, последовательно **делится пополам**, поэтому длина интервала постепенно уменьшается.

Метод бисекции на Python

```
def bisection(f, a, b, tol=1e-8):
    if f(a) * f(b) >= 0:
        raise ValueError("No sign change on [a, b]")

    while (b - a) / 2 > tol:
        c = (a + b) / 2
        if f(a) * f(c) <= 0:
            b = c
        else:
            a = c

    return (a + b) / 2


f = lambda x: x**2 - 2

root = bisection(f, 1.0, 2.0)

print(root)
print(f(root))
```

Преимущества

- простая реализация;
- понятный алгоритм;
- высокая надёжность.

Недостаток

Метод сходится **относительно медленно**, потому что на каждом шаге только делит интервал пополам.

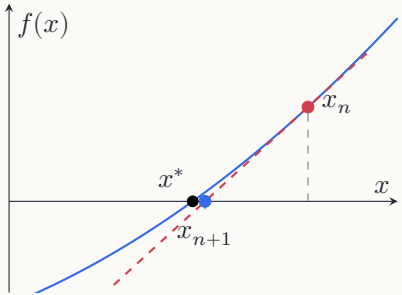
Бисекция — хороший первый выбор, если известен интервал со сменой знака.

Метод Ньютона: идея

Вместо деления интервала пополам строим **касательную** к графику функции в текущей точке x_n.
Точка пересечения касательной с осью x даёт новое приближение x_{n+1}.

Формула метода Ньютона:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



Смысл метода

Метод Ньютона использует локальную линейную аппроксимацию функции.

Метод Ньютона на Python

```
def newton(f, df, x0, tol=1e-8, max_iter=100):
    x = x0

    for _ in range(max_iter):
        x_new = x - f(x) / df(x)

        if abs(x_new - x) < tol:
            return x_new

        x = x_new

    raise RuntimeError("No convergence")

f = lambda x: x**2 - 2
df = lambda x: 2*x

root = newton(f, df, 1.5)

print(root)
print(f(root))
```

Преимущества

- очень быстрая сходимость;
- мало шагов при хорошем x_0.

Недостатки

- нужна производная $f'(x)$;
- возможна потеря сходимости;
- важен выбор начального приближения.

Ньютон быстрее бисекции, но чувствительнее к выбору начальной точки.

Бисекция или Ньютон?

	Бисекция	Ньютон
Начальные данные	интервал $[a, b]$	точка x_0
Требуется смена знака	да	нет
Требуется производная	нет	да
Сходимость	линейная	обычно квадратичная
Надёжность	высокая	зависит от x_0
Скорость	ниже	выше

Когда использовать бисекцию?

- есть интервал со сменой знака;
- важна надёжность;
- производная неизвестна.

Когда использовать Ньютон?

- производная доступна;
- есть хорошее начальное приближение;
- важна скорость сходимости.

Быстрый метод не всегда самый надёжный. На практике методы часто комбинируют.

3

Системы линейных уравнений

Система линейных уравнений

Матричная форма

Система n линейных уравнений записывается компактно:

$$\boxed{A\mathbf{x} = \mathbf{b}}$$

где

- A — матрица коэффициентов;
- $\mathbf{x}$ — неизвестный вектор;
- $\mathbf{b}$ — известная правая часть.

Например:

$$\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}.$$

Откуда возникает?

После дискретизации непрерывная физическая задача часто превращается в систему

$$A\mathbf{x} = \mathbf{b}$$

с большим числом неизвестных.

Решение линейных систем — одна из основных операций вычислительной физики.

Метод Гаусса

1. Прямой ход

Элементарными преобразованиями строк исключаем неизвестные и приводим систему к верхнетреугольному виду:

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right] \Downarrow \left[\begin{array}{ccc|c} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \end{array}\right].$$

2. Обратная подстановка

После получения треугольной системы неизвестные находятся снизу вверх:

$$x_3 \to x_2 \to x_1.$$

Например:

$$a_{33}x_3 = b_3$$

$$x_3 = \frac{b_3}{a_{33}}.$$

Затем найденное x_3 подставляется в предыдущее уравнение.

Метод Гаусса: исключение неизвестных $\to$ обратная подстановка.

Решение системы в NumPy

Решаем $A\mathbf{x} = \mathbf{b}$

```
import numpy as np

A = np.array([[2., 1.],
              [1., 3.]])
b = np.array([5., 6.])

x = np.linalg.solve(A, b)

print(x)    # [1.8 1.4]
```

Проверка

```
print(A @ x)     # [5. 6.]
```

Что делает `solve`?

Команда

`np.linalg.solve(A, b)`

непосредственно решает систему $A\mathbf{x} = \mathbf{b}$.

Явно вычислять A^{-1} для этого не требуется.

В NumPy линейную систему решаем напрямую, а результат обязательно проверяем.

4

Интерполяция и аппроксима-ция данных

Дискретные данные: какие задачи возникают?

Пусть функция известна только в наборе дискретных точек: $(x_1, y_1),\ (x_2, y_2),\ \dots,\ (x_n, y_n)$.
Необходимо построить приближённую зависимость $y \approx \tilde{f}(x)$.

Интерполяция

Построенная функция должна **точно проходить через заданные точки**.
Основная задача:

найти значения между известными узлами

Например:

$$T(10) = 320 \text{ К}, \qquad T(20) = 350 \text{ К}.$$

Как оценить $T(15)$?

Аппроксимация данных / fitting

Функция описывает **общую закономерность** в данных и не обязана проходить через каждую точку.
Основная задача:

определить модель или её параметры

Например:

$$y \approx ax + b.$$

По экспериментальным данным необходимо определить a и b.

Что такое интерполяция?

Узлы интерполяции

Пусть известны значения

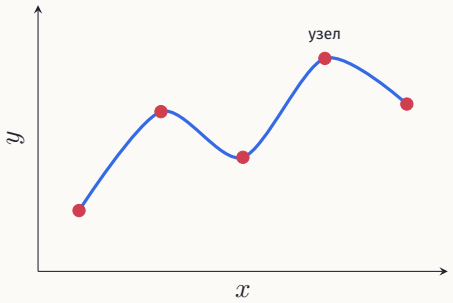
$$y_i = f(x_i).$$

Точки x_i называются **узлами интерполяции**.

Необходимо построить функцию $\tilde{f}(x)$, такую что

$$\tilde{f}(x_i) = y_i$$

для всех известных узлов.



Интерполяция используется, когда нужно оценить функцию между известными значениями:

$$x_i < x < x_{i+1}.$$

Линейная интерполяция

Между двумя соседними узлами

$$(x_0, y_0), \qquad (x_1, y_1)$$

функция заменяется прямой линией.

Для точки $x \in [x_0, x_1]$:

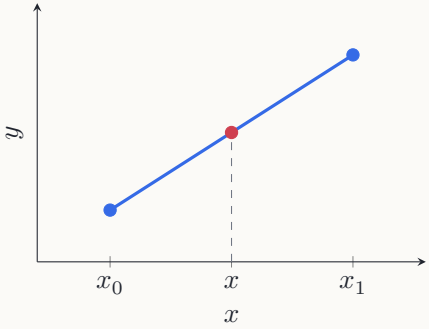
$$\boxed{y(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)}$$

Например:

$$T(10) = 320 \text{ К}, \qquad T(20) = 350 \text{ К}.$$

Тогда

$$T(15) = 335 \text{ К}.$$



Свойства

- простая и быстрая;
- использует соседние узлы.

Полиномы и кубические сплайны

Линейная интерполяция проста, но для гладких функций иногда требуется более гладкая кривая.

Глобальный полином

Через $n+1$ узлов можно построить полином степени не выше n:

$$P_n(x_i) = y_i.$$

Один полином используется для всей области.

Проблема

При большом числе узлов полином высокой степени может сильно осциллировать, особенно около границ интервала.

Кубический сплайн

Область разбивается на интервалы.
На каждом интервале строится кубический полином:

$$S_i(x) = a_i + b_i x + c_i x^2 + d_i x^3.$$

Соседние полиномы соединяются с согласованными производными.

Аппроксимация данных и fitting

Почему не всегда нужна интерполяция?

Экспериментальные и численные данные часто содержат шум.
В таком случае нет необходимости заставлять модель проходить точно через каждую точку.

Предположим, что модель имеет вид

$$\tilde{f}(x) = ax + b.$$

Необходимо подобрать параметры a и b.

Метод наименьших квадратов

Один из наиболее распространённых критериев:

$$S(a, b) = \sum_{i=1}^{n} [y_i - (ax_i + b)]^2 \to \min$$

Выбираются такие параметры a, b, при которых суммарное квадратичное отклонение минимально.

Fitting ищет закономерность в данных, а не пытается точно воспроизвести каждую точку.

Интерполяция и fitting в Python

```python
import numpy as np
from scipy.interpolate import CubicSpline

x = np.array([0., 1., 2., 3., 4.])
y = np.array([0.2, 1.1, 1.8, 3.2, 3.9])

x_new = np.linspace(0, 4, 200)

# Linear interpolation
y_lin = np.interp(x_new, x, y)

# Cubic spline
spline = CubicSpline(x, y)
y_spline = spline(x_new)

# Linear fitting: y = a*x + b
a, b = np.polyfit(x, y, 1)
y_fit = a * x_new + b
```

Интерполяция

`np.interp`

или

`CubicSpline`

дают функцию, проходящую через узлы.

Fitting

`np.polyfit`

подбирает параметры модели по всем данным.

$$y \approx ax + b.$$

5

Численное дифференцирование

Производная из дискретных данных

От определения к вычислению

Производная определяется как

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

Но в численных расчётах h всегда конечно. Поэтому производную заменяют **конечной разностью**.

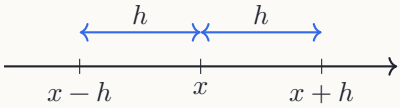
Разность вперёд

$$\boxed{f'(x) \approx \frac{f(x+h) - f(x)}{h}}$$

Центральная разность

Используем значения с двух сторон от точки:

$$\boxed{f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}}$$



Симметричная центральная разность обычно даёт более высокую точность при том же шаге h.

Откуда берётся порядок точности?

Разложение Тейлора

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \cdots$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \cdots$$

Разность вперёд

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{h}{2}f''(x) + \cdots$$

$$= f'(x) + \mathcal{O}(h)$$

Центральная разность

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{h^2}{6}f'''(x) + \cdots$$

$$= f'(x) + \mathcal{O}(h^2).$$

Порядок $\mathcal{O}(h^p)$ показывает, как быстро уменьшается погрешность при уменьшении шага.

Численная производная на Python

Для

$$f(x) = \sin x$$

точная производная известна:

$$f'(x) = \cos x.$$

Если h уменьшить в 10 раз

Теоретически:

$$\mathcal{O}(h) \Rightarrow E/10,$$

$$\mathcal{O}(h^2) \Rightarrow E/100.$$

```python
import numpy as np

f = np.sin

x = 1.0
h = 1e-3

df_forward = ( f(x + h) - f(x) ) / h
df_central = ( f(x + h) - f(x - h) ) / (2*h)

df_exact = np.cos(x)

err_forward = abs(df_forward - df_exact)
err_central = abs(df_central - df_exact)

print(df_forward, err_forward)
print(df_central, err_central)
```

6

Численное интегрирование

Интеграл как сумма простых площадей

Необходимо вычислить

$$I = \int_a^b f(x)\,dx$$

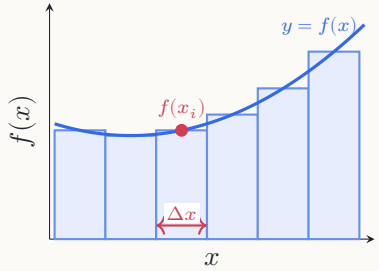
Разобьём интервал:

$$a = x_0 < x_1 < \cdots < x_N = b$$

$$\Delta x = \frac{b-a}{N}.$$

Основная идея:

площадь под кривой $\approx \sum$ простых площадей



Квадратурная формула

$$I \approx \sum_i f(x_i)\Delta x$$

Численное интегрирование превращает непрерывную площадь в конечную сумму.

Метод трапеций

На интервале

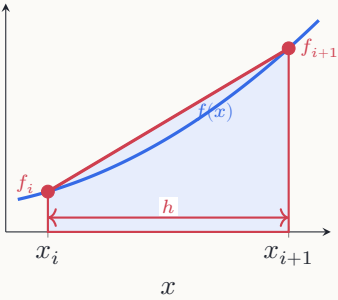
$$[x_i, x_{i+1}]$$

заменим функцию **прямой**, соединяющей две соседние точки. Тогда площадь под функцией приближённо равна площади трапеции:

$$\int_{x_i}^{x_{i+1}} f(x)\,dx \approx \frac{h}{2}\,[f_i + f_{i+1}]$$

Для равномерной сетки

$$h = \frac{b-a}{N}.$$



Формула

$$I_{\text{trap}} = h\left[\frac{f_0 + f_N}{2} + \sum_{i=1}^{N-1} f_i\right]$$

Метод Симпсона

На двух соседних интервалах $[x_0, x_2]$ заменим функцию **параболой**, проходящей через три узла:

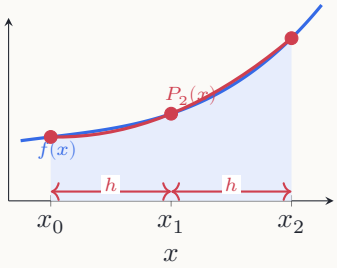
$$(x_0, f_0), \qquad (x_1, f_1), \qquad (x_2, f_2).$$

При одинаковом шаге $h = x_1 - x_0 = x_2 - x_1$ получаем:

$$\int_{x_0}^{x_2} f(x)\,dx \approx \frac{h}{3}\left[f_0 + 4f_1 + f_2\right]$$

Для составной формулы число интервалов должно быть чётным: $N = 2, 4, 6, \ldots$

Метод трапеций использует линейную, а метод Симпсона — квадратичную аппроксимацию.



Формула

$$I_S = \frac{h}{3}\left[f_0 + f_N + 4\sum_{\substack{i=1 \\ i \text{ нечёт.}}}^{N-1} f_i + 2\sum_{\substack{i=2 \\ i \text{ чёт.}}}^{N-2} f_i\right]$$

Численное интегрирование в Python

Рассмотрим функцию

$$f(x) = \sin x, \qquad \int_0^{\pi} \sin x \, dx = 2.$$

Метод трапеций

$$I_{\text{trap}} \approx 1.9998355, \qquad E_{\text{trap}} \approx 1.64 \times 10^{-4}$$

Метод Симпсона

$$I_{\text{S}} \approx 2.00000001, \qquad E_{\text{S}} \approx 1.08 \times 10^{-8}$$

На той же сетке ошибка Симпсона значительно меньше.

Используем $N = 100$ одинаковых интервалов:

```
import numpy as np
from scipy.integrate import simpson

N = 100

x = np.linspace(0, np.pi, N + 1)
y = np.sin(x)

I_trap = np.trapezoid(y, x)
I_simp = simpson(y, x=x)

I_exact = 2.0

print("Trapezoid:", I_trap)
print("Simpson:  ", I_simp)

print("Error trap:", abs(I_trap - I_exact))

print("Error simp:", abs(I_simp - I_exact))
```

Когда обычная сетка становится слишком дорогой?

От одного измерения к многим

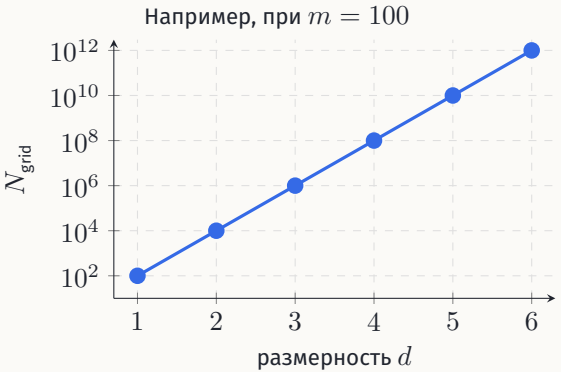
В одном измерении:

$$I = \int_a^b f(x)\,dx.$$

В вычислительной физике часто возникают многомерные интегралы:

$$I = \int f(x_1, \dots, x_d)\,dx_1 \cdots dx_d.$$

Если по каждой координате взять m узлов, то требуется $N_{\text{grid}} = m^d$ точек.



Регулярная сетка → N_{grid} быстро растёт с d → **Случайная выборка** метод Монте-Карло

7

Практика и итоги

Задания для самостоятельной работы

Задание 1 — функция

Исследуем

$$f(x) = x^3 - x - 1$$

1. Найдите корень на $[1, 2]$ методом **бисекции**.
2. Найдите тот же корень методом **Ньютона** при $x_0 = 1.5$.
3. Вычислите $f'(x^*)$ центральной разностью.
4. Вычислите
$$\int_1^2 f(x)\,dx$$
методом трапеций и Симпсона.

Задание 2 — дискретные данные

Пусть известны значения температуры:

x	0	1	2	3	4
T	300	315	327	346	358

1. Оцените $T(1.5)$ линейной интерполяцией.
2. Постройте кубический сплайн.
3. Выполните линейный fitting:
$$T(x) \approx ax + b.$$
4. Сравните интерполяцию и fitted-модель на одном графике.

Итоги лекции

Решение уравнений

- бисекция;
- метод Ньютона;
- метод Гаусса.

Работа с данными

- интерполяция;
- кубический сплайн;
- fitting.

Операции

- дифференцирование;
- интегрирование;
- оценка погрешности.

Но что делать, если неизвестна не одна величина, а целая функция $T(x,t)$?

$$\boxed{\frac{dy}{dt} = f(t,y)} \qquad \boxed{\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}}$$

Лекция 4 — Численное решение дифференциальных уравнений