



Системный анализ процессов переработки нефти и газа

Лекция 4

Введение в библиотеку SciPy.
Визуализация данных при помощи
библиотеки Matplotlib

Вячеслав Алексеевич Чузлов
к.т.н., доцент ОХИ ИШПР

19 февраля 2024 г.



СОДЕРЖАНИЕ

1. Физические константы
2. Интегрирование и обыкновенные дифференциальные уравнения
 - Определенные интегралы от одной переменной
 - Обыкновенные дифференциальные уравнения
3. Интерполяция и аппроксимация
 - Одномерная интерполяция
 - Нелинейная аппроксимация методом наименьших квадратов
4. Численные методы решения уравнений
5. Визуализация данных с помощью библиотеки Matplotlib
 - Настройка графика
 - Пределы осей координат
 - Метки на графиках
 - Диаграммы рассеяния
 - Гистограммы
 - Трехмерные графики

Введение

- SciPy является библиотекой модулей языка Python для научных расчетов, предоставляющей более специализированные функциональные возможности, чем общие структуры данных и математические алгоритмы, которые предлагает библиотека NumPy.
- Библиотека SciPy содержит модули для вычисления специальных функций, распространенных в научной и инженерной деятельности в целях оптимизации, интегрирования, интерполяции и обработки изображений.
- По аналогии с библиотекой NumPy большинство внутренних алгоритмов SciPy выполняется как заранее скомпилированный С-код, что обеспечивает высокую скорость вычислений.
- В дополнение библиотека SciPy является свободно распространяемым программным обеспечением с открытым кодом.



TOMSK
POLYTECHNIC
UNIVERSITY



ТОМСКИЙ
ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

Физические константы

Физические константы

- В библиотеке SciPy содержатся рекомендованные международным Комитетом по данным для науки и техники CODATA значения целого ряда физических констант (<https://physics.nist.gov/cuu/Constants/>).
- Значения констант вместе с единицами их измерения и допустимыми погрешностями хранятся в словаре `scipy.constants.physical_constants`, в котором ключами являются строки идентификации. Например:

```
>>> import scipy.constants as const
>>> const.physical_constants['Avogadro constant']
(6.02214076e+23, 'mol^-1', 0.0)
```

Специальные методы `value`, `unit` и `precision` возвращают соответствующие свойства констант:

```
>>> const.value('electron mass')
9.1093837015e-31
>>> const.unit('electron mass')
'kg'
>>> const.precision('electron mass')
3.0737534961217373e-10
```

Физические константы

Полный перечень всех констант с названиями приведен в официальной документации (<https://docs.scipy.org/doc/scipy/reference/constants.html>), наиболее распространенные из них представлены в таблице:

Имя константы	Переменная	Значение	Единицы измерения
'atomic mass constant'	m_u	1.6605390666e-27	кг
'Avogadro constant'	N_A	6.02214076e+23	1/моль
'Bohr magneton'		9.2740100783e-24	Дж/Тл
'Bohr radius'		5.29177210903e-11	м
'Boltzmann constant'	k	1.380649e-23	Дж/К
'electron mass'	m_e	9.1093837015e-31	кг
'elementary charge'	e	1.602176634e-19	Кл
'Faraday constant'		96485.33212	Кл/моль
'molar gas constant'	R	8.314462618	Дж/(моль · К)
'neutron mass'	m_n	1.67492749804e-27	кг
'Planck constant'	h	6.62607015e-34	Дж · с
'proton mass'	m_p	1.67262192369e-27	кг
'Rydberg constant'	Rydberg	10973731.56816	1/м
'speed of light in vacuum'	c	299792458.0	м/с

Физические константы

Значения некоторых самых часто используемых констант присвоены переменным в модуле `scipy.constants` (в единицах СИ), что делает возможным их прямой импорт:

```
>>> from scipy.constants import c, R, k
>>> # скорость света, универсальная газовая постоянная, постоянная Больцмана
>>> c, R, k
(299792458.0, 8.314462618, 1.380649e-23)
```

Пакет `scipy.constants` содержит определение полезных коэффициентов и методов преобразования, которые включают префиксы системы СИ:

```
>>> import scipy.constants as const
>>> const.atm # 1 атм в Па
101325.0
>>> const.bar # 1 бар в Па
100000.0
>>> const.torr # 1 торр в Па
133.32236842105263
>>> const.zero_Celsius # 0 °C в K
273.15
>>> const.micro # а также нано, пико, мега, гига и т.д.
1e-06
```



Интегрирование и обыкновенные дифференциальные уравнения

Определенные интегралы от одной переменной

- Наиболее распространенной программой численного интегрирования является `scipy.integrate.quad`, которая основана на библиотеке FORTRAN 77 QUADPACK.
- В данной программе используется методика адаптивной квадратуры для приближенного вычисления значения интеграла делением его области интегрирования на меньшие интервалы, которые выбираются итеративно для достижения допустимого предела погрешности.
- В наиболее простом виде метод принимает три аргумента: объект Python-функции, соответствующий подынтегральной функции, `func`, а также пределы интегрирования `a` и `b`.
- В тех случаях, когда подынтегральная функция представляет собой простое выражение, удобно использовать `lambda`-функции.

К примеру, для получения значения интеграла

$$\int_1^4 x^{-2} dx = \frac{3}{4}$$

в численном виде:

```
>>> from scipy.integrate import quad
>>> quad(lambda x: 1 / x ** 2, 1, 4)
(0.7500000000000002, 1.913234548258993e-09)
```

Определенные интегралы от одной переменной

- Метод quad возвращает два значения в виде кортежа – значение интеграла и оценку абсолютной погрешности полученного результата.
- Для вычисления несобственных интегралов используется специализированное значение np.inf:

```
>>> import numpy as np
>>> quad(lambda x: np.exp(-x ** 2), 0, np.inf)
(0.8862269254527579, 7.101318378329813e-09)
>>> np.sqrt(np.pi) / 2 # аналитическое решение
0.8862269254527579
```

- В тех случаях, когда подынтегральная функция представляет собой более сложное выражение необходимо выполнять явное определение Python-функции при помощи оператора def:

```
>>> def g(x):
...     if abs(x) < 0.5:
...         return -x
...     return x - np.sign(x)
...
>>> quad(g, -0.6, 0.8)
(-0.0600000000000002, 6.661338147750941e-17)
```

Определенные интегралы от одной переменной

- В методе quad предусмотрены опциональные аргументы `epsrel` и `epsabs`, позволяющие определить требуемую точность вычисления интеграла в виде относительной и абсолютной погрешности, соответственно.
- По умолчанию для этих параметров принято значение `1.49e-8`, однако расчет можно выполнить быстрее, если нужно получить менее точный результат. Рассмотрим для наглядности быстро изменяющуюся функцию:

$$f(x) = e^{-|x| \sin^2 x^2}$$

```
>>> def f(x):
...     return np.exp(-np.abs(x) * np.sin(x ** 2) ** 2)
...
>>> quad(f, -1, 2, epsabs=0.1)
(2.3015366025566437, 0.001973649019845005)
>>> quad(f, -1, 2, epsabs=1.49e-8) # значение по умолчанию
(2.3015386470012746, 8.388012012715896e-10)
```

- Параметр `epsabs` – это требуемая верхняя граница. В действительности точность результата может оказаться значительно лучше, чем эта оценка.

Определенные интегралы от одной переменной

- В тех случаях, когда подынтегральная функция принимает один или несколько параметров помимо своего основного аргумента, эти дополнительные параметры могут быть переданы в метод quad в виде кортежа через аргумент args. Например, определим следующий интеграл в численном выражении:

$$I_{n,m} = \int_{-\pi/2}^{\pi/2} \sin^n x \cos^m x dx$$

```
>>> def f(x, n, m):
...     return np.sin(x) ** n * np.cos(x) ** m
...
>>> n, m = 2, 1
>>> quad(f, -np.pi/2, np.pi/2, args=(n, m))
(0.6666666666666667, 1.6257070626918973e-13)
```

- Дополнительные параметры n и m передаются как аргументы подынтегральной функции после координаты, определяющей направление интегрирования по x .

Обыкновенные дифференциальные уравнения

Решение одного ОДУ первого порядка

При решении одного ОДУ первого порядка:

$$\frac{dy}{dt} = f(t, y)$$

метод `solve_ivp` принимает три аргумента: объект функции, возвращаемой dy/dt , начальный и конечный моменты времени для интегрирования и набор начальных условий y_0 .

В соответствии с механизмом решения ОДУ выбирается и возвращается последовательность подходящих моментов времени (точек), в которых выполняется интегрирование (если эти моменты времени не были заданы).

Например, рассмотрим дифференциальное уравнение первого порядка для описания скорости реакции $A \rightarrow B$ по концентрации компонента A:

$$\frac{d[A]}{dt} = -k [A]$$

Для данного примера можно вполне легко получить аналитическое решение:

$$[A] = [A]_0 e^{-kt}$$

где $[A]_0$ – начальная концентрация компонента A.

Обыкновенные дифференциальные уравнения

- Для численного решения данного уравнения при помощи функции `solve_ivp()` потребуется записать его в форме с единственной зависимой переменной $y(t) \equiv [A]$, являющейся функцией независимой переменной t (время эксперимента). В итоге получим:

$$\frac{dy}{dt} = -ky$$

- Необходимо определить функцию, возвращающую dy/dt , как $f(t, y)$ (в общем случае эта функция зависит и от t и от y), которая для рассматриваемого примера может быть записана следующим образом:

```
>>> def func(t, y):
...     return -k * y
```

- В данном случае порядок аргументов крайне важен. Начальный и конечный моменты времени t_span передаются в виде кортежа $t0, tf$, а начальные условия необходимо предать в виде массива, даже если, как и в нашем случае, передается всего одно значение:

```
>>> solution = solve_ivp(func, (t0, tf), [y0])
```

- Возвращаемый объект `solution` является экземпляром класса `OdeResult`, определяющий ряд важных свойств, таких как массивы `solution.t` для точек по времени, использованных при интегрировании и `solution.y` со значениями решений, полученных в этих точках по времени.

Обыкновенные дифференциальные уравнения

Пример программы, сравнивающей численное и аналитическое решение для реакции при $k = 0.2 \text{ c}^{-1}$ и $y(0) \equiv [A]_0 = 100$:

```
>>> import numpy as np
>>> from scipy.integrate import solve_ivp
>>> k = 0.2 # Константа скорости реакции первого порядка, 1 / с
>>> y0 = 100 # Начальное условие: y = 100 в момент времени t = 0
>>> t0, tf = 0, 20 # Начальная и конечная точки времени для интегрирования
>>> def func(t, y):
...     """Return dy/dt = f(t, y) at time t."""
...     return -k * y
...
>>> # Интегрирование дифференциального уравнения
>>> solution = solve_ivp(func, (t0, tf), [y0])
>>> t, y = solution.t, solution.y[0]
```

Обыкновенные дифференциальные уравнения

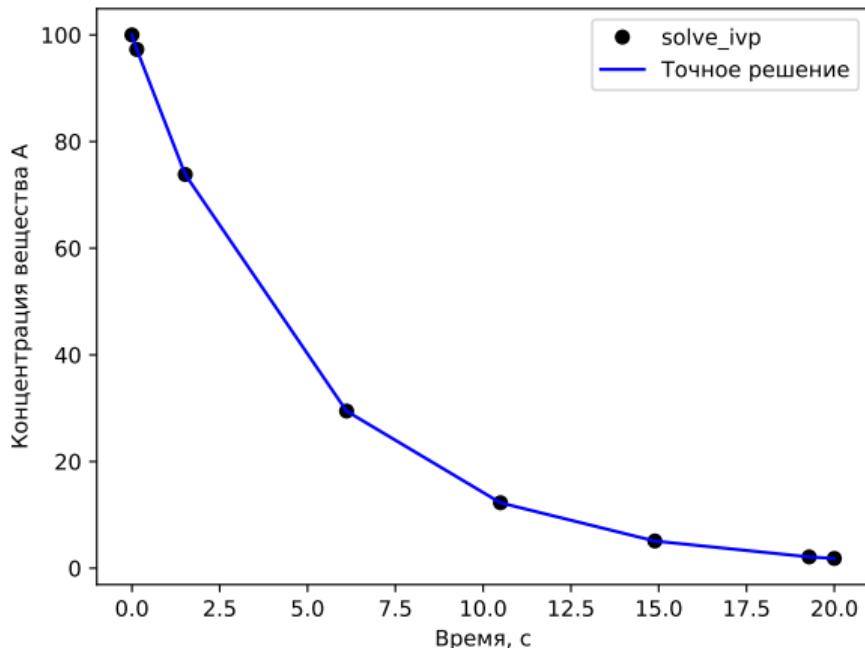


Рисунок 1 – Снижение концентрации реагента в реакции первого порядка:
аналитическое и численное решения

- Данный подход оправдан в тех случаях, когда требуется определить только конечную концентрацию реагента, однако для отслеживания изменения концентрации во времени с более высокой дискретностью можно передать специальную последовательность точек по времени в optionalном аргументе `t_eval`.

Обыкновенные дифференциальные уравнения

- Разобьем наш интервал интегрирования на 20 точек по времени:

```
>>> t0, tf = 0, 20
>>> t_eval = np.linspace(t0, tf, 20)
>>> solution = solve_ivp(func, (t0, tf), [y0], t_eval=t_eval)
>>> t, y = solution.t, solution.y[0]
```

- В optionalный аргумент `dense_output` можно передать значение `True` для определения объекта `OdeSolution` с атрибутом `sol` как одним из возвращаемых объектов.
- Данный функционал можно использовать для генерации значений решения в промежуточных точках по времени:

```
>>> # Начальная и конечные точки по времени интегрирования
>>> t0, tf = 0, 20
>>> # Интегрирования дифференциального уравнения
>>> solution = solve_ivp(func, (t0, tf), [y0], dense_output=True)
>>> t = np.linspace(t0, tf, 20)
>>> y = solution.sol(t)[0]
```

- Объект `solution.sol` можно вызывать: значение независимой переменной – времени – передается в него как аргумент, и возвращается массив решений в этот момент времени. В рассмотренном случае есть только одна зависимая переменная `y`, поэтому используется индекс `[0]`.

Обыкновенные дифференциальные уравнения

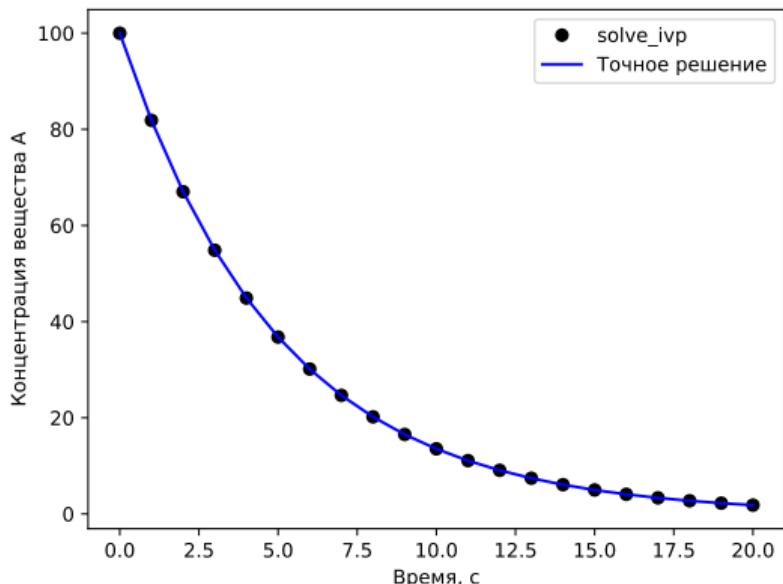


Рисунок 2 – Снижение концентрации реагента по времени: аналитическое и численное решение с использованием предварительно определенных точек

- Аналогично методу quad, дополнительные аргументы можно передать через параметр args.
- В рассмотренном выше примере константа скорости реакции k определена в глобальной области видимости, однако лучше передавать эту переменную явным образом:

```
>>> def func(t, y, k):  
...     return -k * y  
... 
```

- Дополнительные аргументы указываются в параметре args:
`>>> solution = solve_ivp(func, (t0, tf), [y0], args=(k,))`

Система взаимосвязанных ОДУ первого порядка

Решение системы ОДУ первого порядка

Метод `solve_ivp` также может быть использован для решения системы взаимосвязанных ОДУ первого порядка с несколькими зависимыми переменными $y_1(t), y_2(t), \dots, y_n(t)$:

$$\begin{cases} \frac{dy_1}{dt} = f_1(y_1, y_2, \dots, y_n; t) \\ \frac{dy_2}{dt} = f_2(y_1, y_2, \dots, y_n; t) \\ \dots \\ \frac{dy_n}{dt} = f_n(y_1, y_2, \dots, y_n; t) \end{cases}$$

```
>>> def func(t, y):
...     # y = [y1, y2, ... yn] - последовательность зависимых переменных
...     dy1dt = f1(y, t)
...     dy2dt = f2(y, t)
...     # ... и т.д.
...     # Возвращаются вычисленные производные в последовательности,
...     # например, в кортеже
...     return dy1dt, dy2dt, ... dyndt
```

Система взаимосвязанных ОДУ первого порядка

Для наглядной демонстрации рассмотрим следующую схему химических реакций:



с константами скоростей k_1 и k_2 . Уравнения, описывающие скорость изменения концентраций компонентов по времени, записываются следующим образом:

$$\begin{cases} \frac{d[A]}{dt} = -k_1 [A] \\ \frac{d[B]}{dt} = k_1 [A] - k_2 [B] \\ \frac{d[C]}{dt} = k_2 [B] \end{cases}$$

Для численного решения предположим $y_1 \equiv [A]$, $y_2 \equiv [B]$ и $y_3 \equiv [C]$:

$$\begin{cases} \frac{d[A]}{dt} = -k_1 y_1 \\ \frac{d[B]}{dt} = k_1 y_1 - k_2 y_2 \\ \frac{d[C]}{dt} = k_2 y_2 \end{cases}$$

Система взаимосвязанных ОДУ первого порядка

Зададимся значениями констант: $k_1 = 0.2 \text{ с}^{-1}$, $k_2 = 0.8 \text{ с}^{-1}$ и начальными условиями: $y_1(0) = 100$, $y_2(0) = 0$ $y_3(0) = 0$.

```
>>> import numpy as np
>>> from scipy.integrate import solve_ivp
>>> # константы скоростей и начальные условия
>>> k1, k2 = 0.2, 0.8
>>> a0, b0, c0 = 100, 0, 0
>>> t0, tf = 0, 20
>>> def func(t, y, k1, k2):
...     """Returns dy_i/dt = f(t, y_i) at time t."""
...     y1, y2, y3 = y
...     dy1dt = -k1 * y1
...     dy2dt = k1 * y1 - k2 * y2
...     dy3dt = k2 * y2
...     return dy1dt, dy2dt, dy3dt
...
>>> y0 = a0, b0, c0
```

Система взаимосвязанных ОДУ первого порядка

```
>>> solution = solve_ivp(func, (t0, tf), y0, dense_output=True, args=(k1, k2))
>>> t = np.linspace(t0, tf, 50)
>>> a, b, c = solution.sol(t)
>>> # аналитическое решение
>>> a_exact = a0 * np.exp(-k1 * t)
>>> b_exact = a0 * k1 / (k2 - k1) * (np.exp(-k1 * t) - np.exp(-k2 * t))
>>> c_exact = a0 - a_exact - b_exact
```

Система взаимосвязанных ОДУ первого порядка

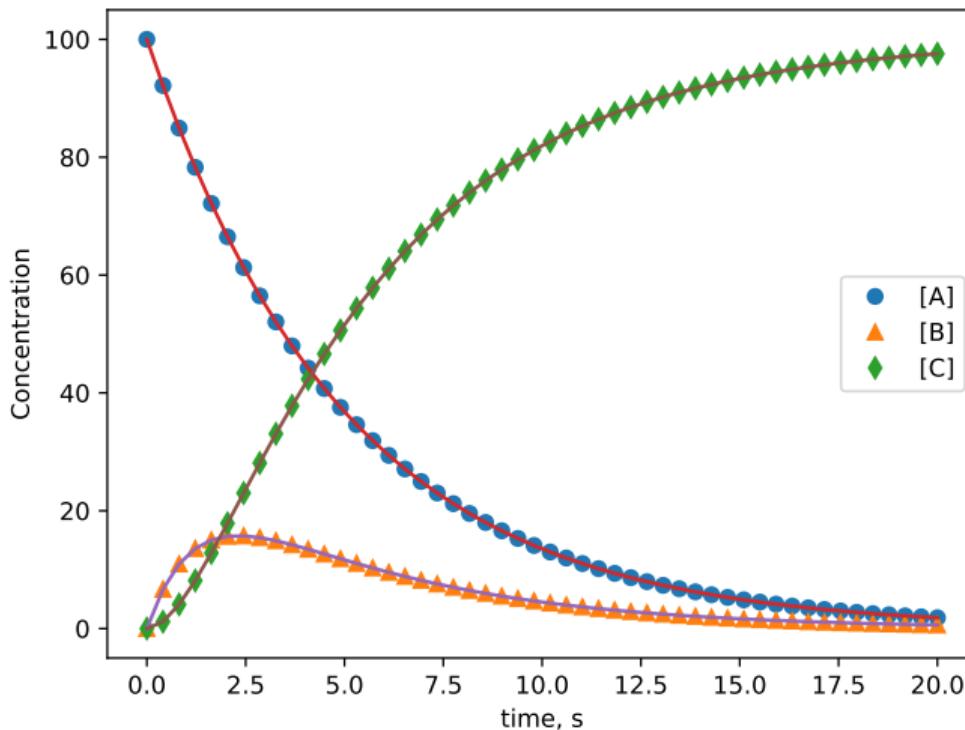


Рисунок 3 – Схема из двух взаимосвязанных химических реакций первого порядка: численное и аналитическое решения

TOMSK
POLYTECHNIC
UNIVERSITY



ТОМСКИЙ
ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

Интерполяция и аппроксимация

Одномерная интерполяция

- Метод `scipy.interpolate.interp1d` представляет метод одномерной интерполяции.
- Данный метод в качестве входных параметров принимает массивы точек `x` и `y`, а возвращает объект функции, который можно вызывать для получения интерполируемых значений в промежуточных точках `x`.
- По умолчанию используется линейная схема интерполяции, однако существует возможность применения и других вариантов.

kind	Описание
'linear'	Принятая по умолчанию линейная интерполяция, использующая при расчетах только значения из исходных данных, охватывающих требуемую точку
'nearest'	Привязка к ближайшей точке данных
'zero'	Сплайн нулевого порядка: интерполирует по последнему наблюдаемому значению при проходе по массиву данных
'slinear'	Интерполяция сплайном первого порядка (аналог 'linear')
'quadratic'	Интерполяция сплайном второго порядка
'cubic'	Интерполяция кубическим сплайном
'previous'	Используется предыдущая точка данных

Одномерная интерполяция

```
>>> import numpy as np
>>> from scipy.interpolate import interp1d
>>> a, nu, k = 10, 4, 2
>>> def func(x, a, nu, k):
...     return a * np.exp(-k * x) * np.cos(2 * np.pi * nu * x)
...
>>> xmax, nx = 0.5, 8
>>> x = np.linspace(0, xmax, nx)
>>> y = func(x, a, nu, k)
>>> nearest = interp1d(x, y, kind='nearest')
>>> linear = interp1d(x, y)
>>> cubic = interp1d(x, y, kind='cubic')
```

Одномерная интерполяция

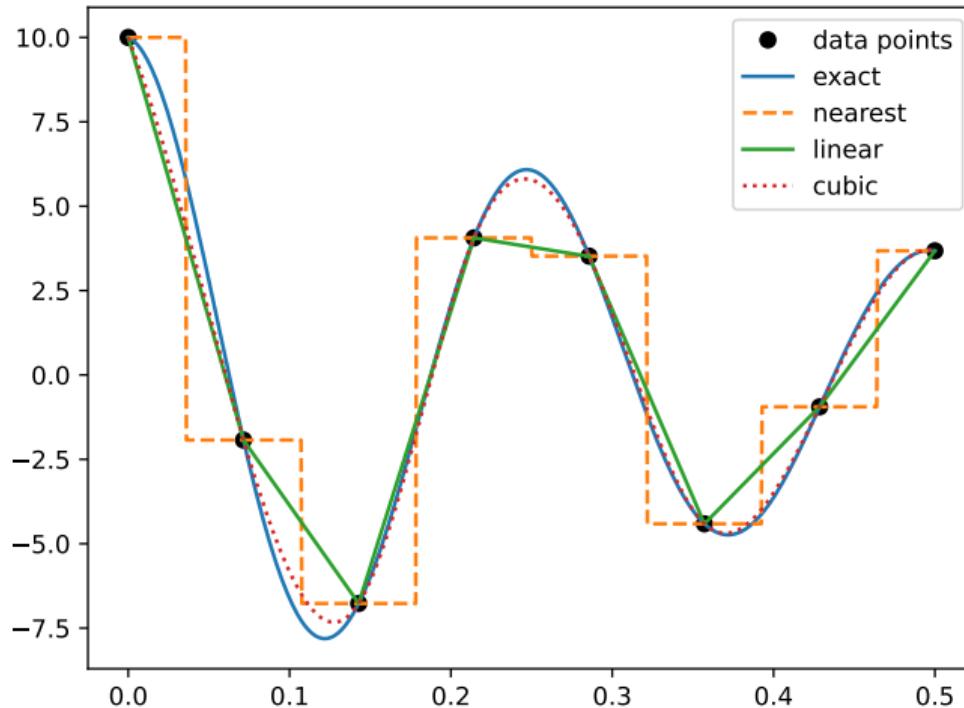


Рисунок 4 – Демонстрация различных методик интерполяции, доступных в методе `scipy.interpolate.interp1d`

Нелинейная аппроксимация методом наименьших квадратов

В библиотеке SciPy реализована нелинейная аппроксимация методом наименьших квадратов в виде `scipy.optimize.leastsq`. Базовая сигнатура вызова данного метода выглядит следующим образом:

```
>>> scipy.optimize.leastsq(func, x0, args=())
```

- Данный метод выполняет аппроксимацию последовательности точек данных у моделируемой функцией f , зависящей от одного или нескольких параметров аппроксимации.
- В функцию `leastsq` необходимо передать соответствующий объект функции `func`, которая возвращает разность между y и f (остатки).
- Также необходимо задать начальное приближение для x_0 .
- Если функция `func` требует дополнительные аргументы, то их можно передать в кортеже `args`. Для наглядности рассмотрим аппроксимацию искусственно зашумленной функции затухающего косинуса

$$f(t) = ae^{-t/\tau} \cos 2\pi\nu t$$

Нелинейная аппроксимация методом наименьших квадратов

```
>>> import numpy as np
>>> a, freq, tau = 10, 4, 0.5
>>> def f(t, a, freq, tau):
...     return a * np.exp(-t / tau) * np.cos(2 * np.pi * freq * t)
...
>>> tmax, dt = 1, 0.01
>>> t = np.arange(0, tmax, dt)
>>> y_exact = f(t, a, freq, tau)
>>> y = y_exact + np.random.randn(y_exact.shape[0]) * 2
```

Нелинейная аппроксимация методом наименьших квадратов

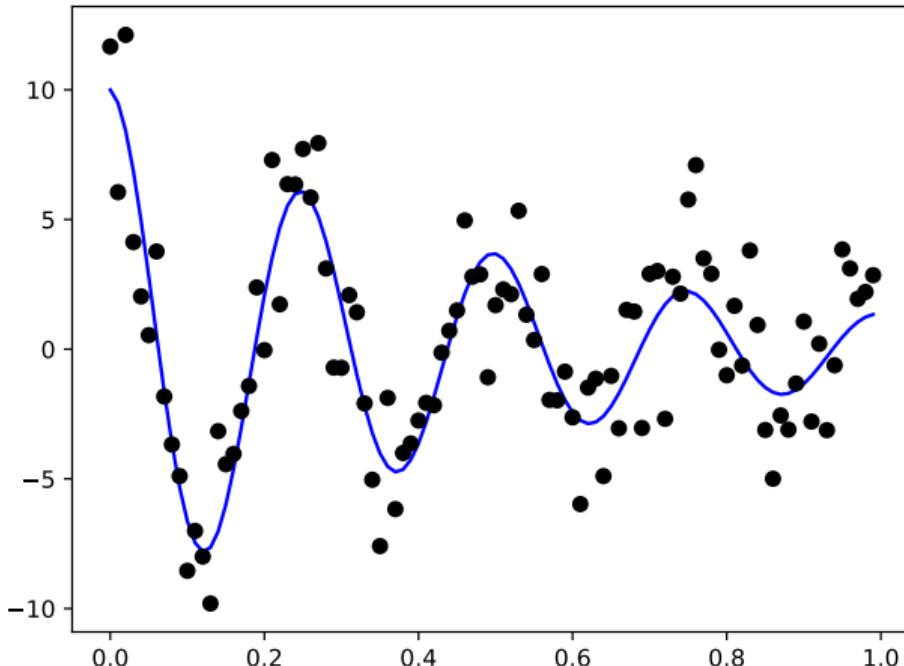


Рисунок 5 – Функция затухающего косинуса с искусственным зашумлением

Нелинейная аппроксимация методом наименьших квадратов

Задача аппроксимации зашумленного набора данных сводится к определению параметров a , $freq$ и τ (представим, что они нам неизвестны).

- Для решения этой задачи сначала необходимо определить функцию `residuals`:

```
>>> def residuals(params, y, t):
...     a, freq, tau = params
...     return y - f(t, a, freq, tau)
...
```

- Первый аргумент – последовательность параметров `params` которые для лучшей читаемости кода распаковываются в именованные переменные.
- Требуемые дополнительные аргументы: набор точек данных `y` и независимая переменная `t`.
- Далее необходимо задать начальные приближения для параметров и вызвать метод `leastsq`:

```
>>> from scipy.optimize import leastsq
>>> params0 = 5, 5, 1
>>> plsq = leastsq(residuals, params0, args=(y, t))
>>> plsq[0]
array([9.1920154, 4.0158567, 0.593839 ])
```

Действительные значения параметров a , $freq$, $\tau = 10$, 4 , 0.5 , таким образом, учитывая шум, добавленный к исходным данным, результат можно считать вполне приемлемым.

Нелинейная аппроксимация методом наименьших квадратов

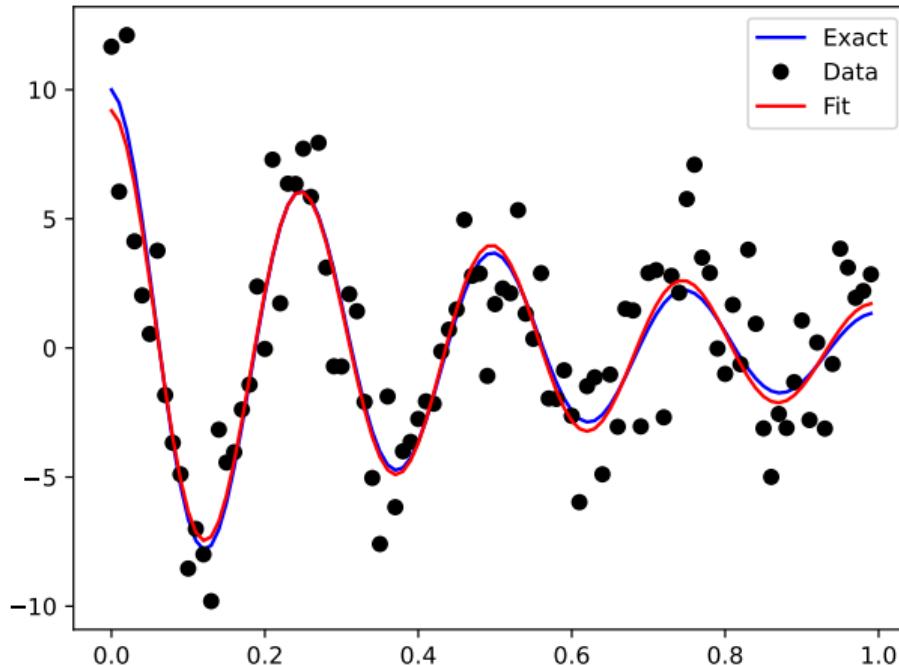


Рисунок 6 – Нелинейная аппроксимация методом наименьших квадратов для функции затухающего косинуса с зашумлением

Аппроксимация

Дана табличная зависимость энтропии вещества от температуры.

$T, \text{ К}$	298.15	400	500	600	700
$S, \text{ Дж}/(\text{моль} \cdot \text{К})$	49.15	51.36	53.30	54.97	56.40

Необходимо построить линейную, степенную и экспоненциальную аппроксимирующие функции и найти значение теплоемкости при температуре $T = 650 \text{ К}$.

- Линейная функция:

$$y = a_0 + a_1 \cdot x$$

где a_0 и a_1 – коэффициенты.

- Степенная функция:

$$y = a \cdot x^b$$

где a и b – коэффициенты.

- Экспоненциальная функция:

$$y = a \cdot e^{b \cdot x}$$

где a и b – коэффициенты.

Аппроксимация

```
1 from __future__ import annotations # для версии Python ниже 3.10
2 import numpy as np
3 from typing import Callable
4 from scipy.optimize import least_squares
5
6
7 def linear(x: float | np.ndarray,
8             params: tuple[float, float]) -> float | np.ndarray:
9     a0, a1 = params
10    return a0 + a1 * x
11
12 def power(x: float | np.ndarray,
13            params: tuple[float, float]) -> float | np.ndarray:
14    a, b = params
15    return a * x ** b
16
17 def exponent(x: float | np.ndarray,
18              params: tuple[float, float]) -> float | np.ndarray:
19    a, b = params
20    return a * np.exp(b * x)
21
```

Аппроксимация

```
22 def residuals(params: tuple[float, float], x: np.ndarray,
23                 y: np.ndarray, func: Callable) -> np.ndarray:
24     return y - func(x, params)
25
26
27 t = np.array([298.15, 400, 500, 600, 700])
28 s = np.array([49.15, 51.36, 53.30, 54.97, 56.40])
29 t_new = 650
30 x0 = 0.01, 0.01
31
32 results = least_squares(residuals, x0=x0, args=(t, s, linear))
33 linear_params, linear_cost = results.x, results.cost
34 print(linear_params, linear_cost)
35
36 results = least_squares(residuals, x0=x0, args=(t, s, power))
37 power_params, power_cost = results.x, results.cost
38 print(power_params, power_cost)
39
40 results = least_squares(residuals, x0=x0, args=(t, s, exponent))
41 exponent_params, exponent_cost = results.x, results.cost
42 print(exponent_params, exponent_cost)
```

Аппроксимация

Результаты расчетов:

- Линейная аппроксимация
 - | [4.40187563e+01 1.80478428e-02] 0.11106445922161706
- Степенная аппроксимация
 - | [19.46645965 0.16224222] 0.011577630390983832
- Экспоненциальная аппроксимация
 - | [4.47183499e+01 3.39118322e-04] 0.17227508882258857

Суммарная ошибка для степенной аппроксимации минимальна, поэтому выбираем данный вид функции для определения энтропии:

```
1 | s_new = power(t_new, power_params)
2 | print(s_new)
3 |
| 55.67500745784808
```

TOMSK
POLYTECHNIC
UNIVERSITY



ТОМСКИЙ
ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

Численные методы решения уравнений

Численные методы решения уравнений

Модуль `scipy.optimize` содержит несколько методов для определения корней уравнений с одной или несколькими неизвестными величинами: `brentq`, `brenth`, `ridder`, `bisect`.

- Каждый из указанных выше методов требует непрерывную функцию $f(x)$ и пару чисел, определяющих интервал поиска корней, т.е. значения a и b , такие, что корень находится в интервале $[a, b]$ и $f(a) \cdot f(b) < 0$.
- Наиболее распространенным методом поиска корней функции является `scipy.optimize.brentq`, являющийся реализацией метода Брента с обратной квадратичной экстраполяцией (`scipy.optimize.brenth` – аналогичный метод, использующий гиперболическую экстраполяцию). Рассмотрим в качестве демонстрации следующую функцию на интервале $-1 \leq x \leq 1$:

$$f(x) = \frac{1}{5} + x \cos\left(\frac{3}{x}\right)$$

- График этой функции (рисунок 7) показывает, что корень находится между -0.7 и -0.5 .

Численные методы решения уравнений

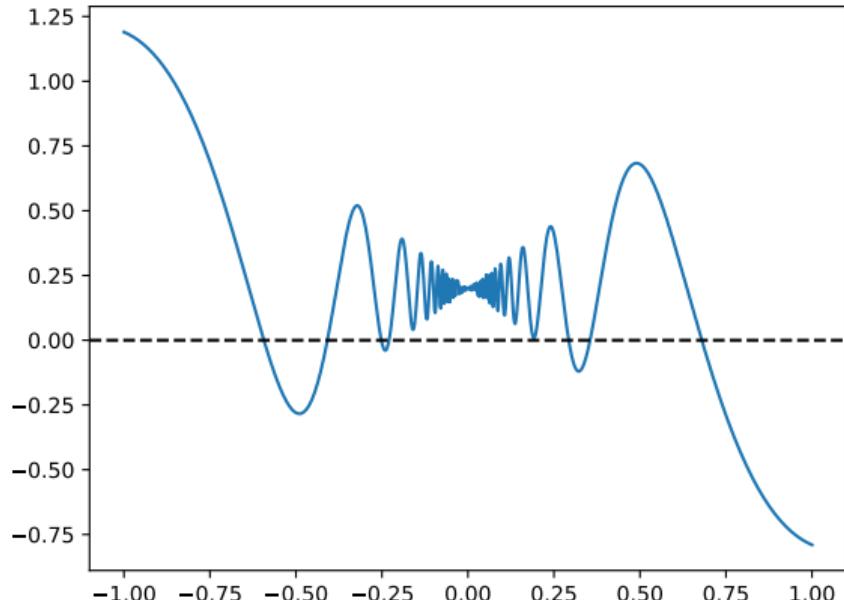


Рисунок 7 – График функции $f(x) = 1/5 + x \cos(3/x)$

```
1 import numpy as np
2 import scipy.optimize as opt
3
4
5 def f(x: float) -> float:
6     return 1 / 5 + x * np.cos(3 / x)
7
8
9 root = opt.brentq(f, -.7, -.5)
10 print(root)
11
12 -0.5933306271014237
```

Численные методы решения уравнений

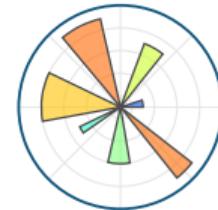
- Наиболее медленный, но весьма надежный **метод бисекций** реализован в `scipy.optimize.bisect`.
- Одним из быстрейших методов поиска корней является алгоритм Ньютона-Рафсона в тех случаях, когда можно найти определить первую производную $f'(x)$.
- В тех случаях, когда возможно написать исходный код для аналитического выражения первой производной $f'(x)$, этот код передается в метод `scipy.optimize.newton` в виде аргумента `fprime` вместе с начальной точкой x_0 , которая должна находиться как можно ближе к определяемому корню. В данном случае нет необходимости определять ограничивающий интервал для поиска корня.
- Если первую производную $f'(x)$ выразить аналитически невозможно, то в методе `newton` используется **алгоритм секущих**.



Визуализация данных с помощью библиотеки Matplotlib

Библиотека Matplotlib

- **Matplotlib** – библиотека для визуализации данных, основанная на массивах NumPy.
- Важнейшей возможностью данного пакета является хорошая совместимость с множеством операционных систем и графических прикладных частей.
- Полезным источником информации является галерея Matplotlib (<https://matplotlib.org/stable/gallery/index.html>). Галерея содержит большое количество миниатюр различных типов графиков, каждая из которых является ссылкой на страницу с фрагментом кода на языке Python, используемым для его генерации.



Для импорта библиотеки Matplotlib приняты стандартные сокращения:

```
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
```

Используемая версия библиотеки - 3.8.2

Настройка стилей

Для того чтобы настроить стиль графиков, будем использовать команду `plt.style`. Ниже приведен пример выбора стиля `classic`, который обеспечит в создаваемых графиках классический стиль библиотеки Matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use('classic')
```

Для того, чтобы увидеть список доступных для использования стилей, необходимо выполнить следующую инструкцию:

```
>>> plt.style.available
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-nogrid', 'bmh',
'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale',
'seaborn-v0_8', 'seaborn-v0_8-bright', 'seaborn-v0_8-colorblind', 'seaborn-v0_8-dark',
'seaborn-v0_8-dark-palette', 'seaborn-v0_8-darkgrid', 'seaborn-v0_8-deep',
'seaborn-v0_8-muted', 'seaborn-v0_8-notebook', 'seaborn-v0_8-paper', 'seaborn-v0_8-pastel',
'seaborn-v0_8-poster', 'seaborn-v0_8-talk', 'seaborn-v0_8-ticks', 'seaborn-v0_8-white',
'seaborn-v0_8-whitegrid', 'tableau-colorblind10']
```

Построение графиков

Построение линейного графика можно выполнить при помощи следующих команд:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 x = np.linspace(0, 10, 100)
6 plt.plot(x, np.sin(x), '-')
7 plt.plot(x, np.cos(x), '--')
8 plt.show()
9
```

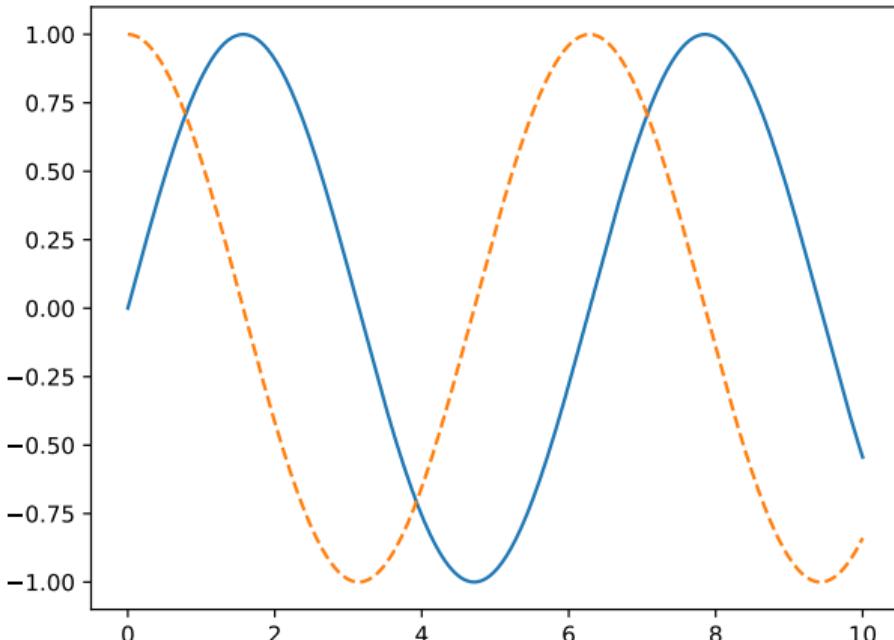


Рисунок 8 – Пример построения графика

Построение графиков

При необходимости совместить несколько графиков на одном рисунке можно воспользоваться функцией `subplots()`:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 x = np.linspace(0, 10, 100)
6
7 fig, ax = plt.subplots(
8     1, 2, figsize=(7, 3.)
9 )
10 ax[0].plot(x, np.sin(x), '-')
11 ax[1].plot(x, np.cos(x), '--')
12
13 plt.show()
14
```

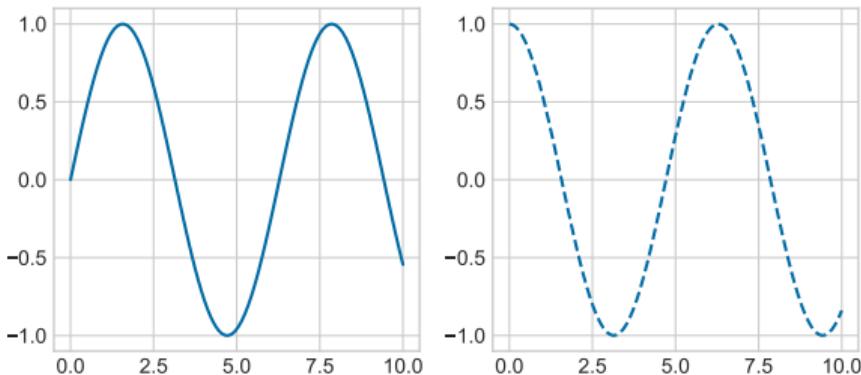


Рисунок 9 – Пример совмещения двух графиков на одном рисунке

Сохранение рисунков

Библиотека Matplotlib предоставляет возможность сохранять рисунки в файлы различных форматов. К примеру, можно сохранить предыдущий рисунок в формате PNG:

```
15 plt.savefig('myfig.png')
```

```
16
```

Метод `savefig()` устанавливает формат файла, исходя из расширения в заданном пользователем имени файла. Список поддерживаемых форматов:

```
17 plt.gcf().canvas.get_supported_filetypes()
```

```
18
```

```
{'eps': 'Encapsulated Postscript',
'jpg': 'Joint Photographic Experts Group',
'jpeg': 'Joint Photographic Experts Group',
'pdf': 'Portable Document Format',
'pgf': 'PGF code for LaTeX',
'png': 'Portable Network Graphics',
'ps': 'Postscript',
'raw': 'Raw RGBA bitmap',
'rgba': 'Raw RGBA bitmap',
'svg': 'Scalable Vector Graphics',
'svgz': 'Scalable Vector Graphics',
'tif': 'Tagged Image File Format',
'tiff': 'Tagged Image File Format'}
```



Простые линейные графики

После того, как созданы оси, можно приступить к построению графика данных, применив метод `ax.plot()`. Рассмотрим простую синусоиду (рисунок 10):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegr
6
7 fig = plt.figure()
8 ax = plt.axes()
9
10 x = np.linspace(0, 10, 1000)
11 ax.plot(x, np.sin(x))
12
13 plt.show()
14
```

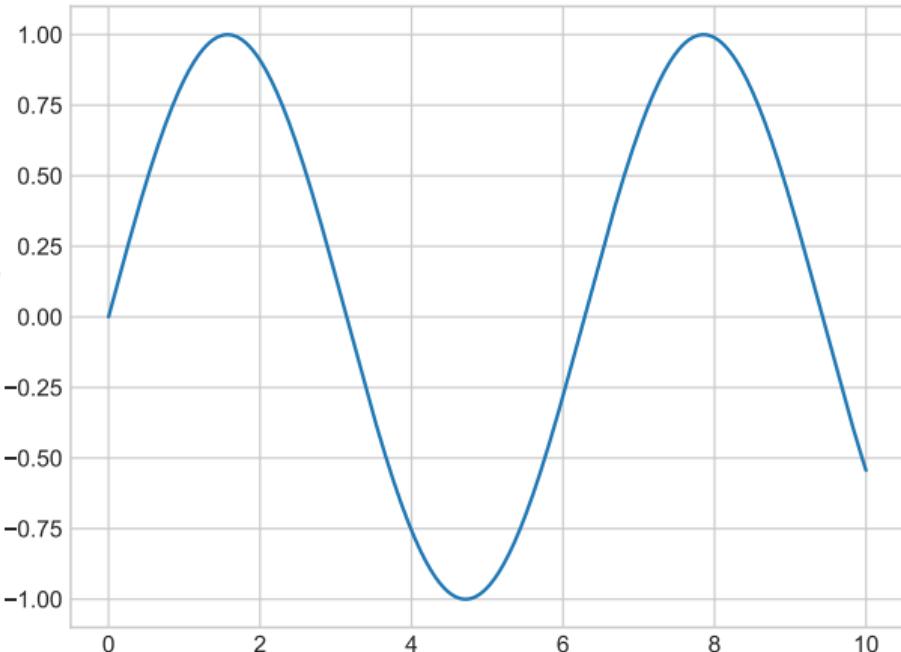


Рисунок 10 – График простой синусоиды



Простые линейные графики

При необходимости создать рисунок с несколькими линиями, можно применить метод `plot` несколько раз, воспользовавшись объектно-ориентированным интерфейсом (рисунок 11):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegr
6
7 fig = plt.figure()
8 ax = plt.axes()
9
10 x = np.linspace(0, 10, 1000)
11 ax.plot(x, np.sin(x))
12 ax.plot(x, np.cos(x))
13
14 plt.show()
```

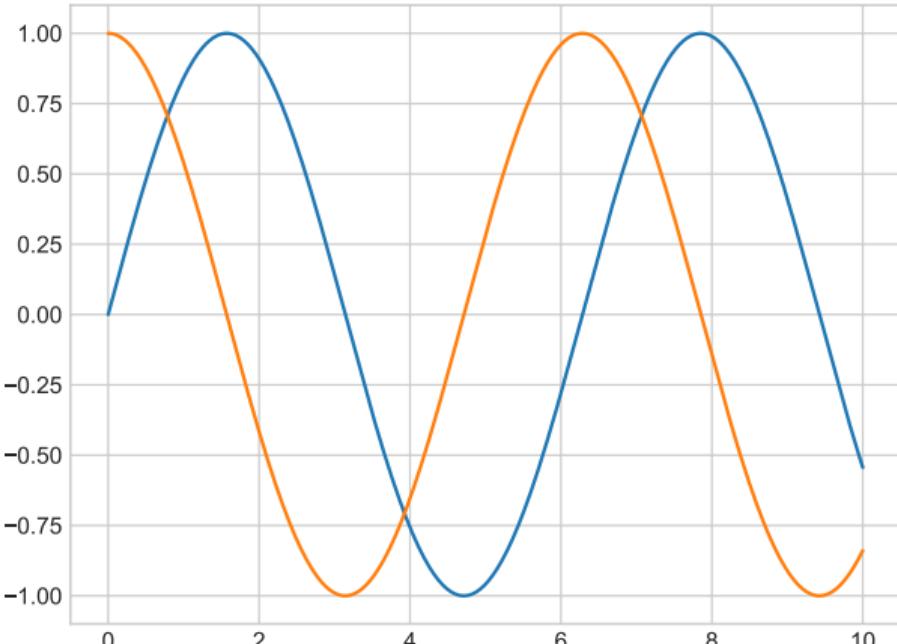


Рисунок 11 – График с несколькими линиями

Цвет и стиль линий

Функции plt.plot можно передавать дополнительные аргументы, позволяющие управлять цветами и стилями линий. Для настройки цвета используется ключевое слово color с соответствующим строковым аргументом, задающим почти любой цвет (рисунок 12):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 plt.style.use('seaborn-v0_8-whitegrid')
5 # цвет по названию
6 plt.plot(x, np.sin(x), color='blue')
7 # краткий код цвета (rgbcmuyk)
8 plt.plot(x, np.sin(x - 1), color='g')
9 # Шкала оттенков серого цвета в диапазоне от 0 до 1
10 plt.plot(x, np.sin(x - 2), color='0.75')
11 # 16-ричный код (RRGGBB от 00 до FF)
12 plt.plot(x, np.sin(x - 3), color="#FFDD44")
13 # Кортеж RGB, значения от 0 до 1
14 plt.plot(x, np.sin(x - 4), color=(1.0, 0.2, 0.3))
15 # Поддерживаются все имена цветов HTML
16 plt.plot(x, np.sin(x - 5), color='chartreuse')
17
18 plt.show()
```

Цвет и стиль линий

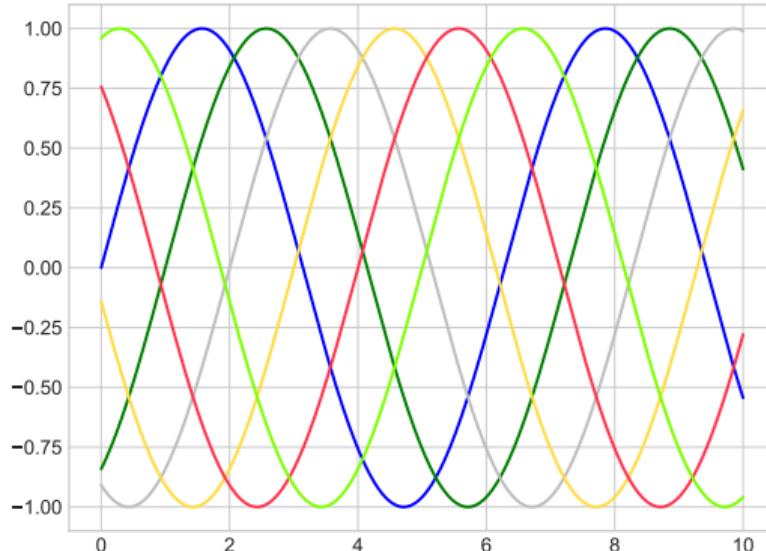


Рисунок 12 – Настройка цвета линий графика

- В том случае, если цвет не определен пользователем, выбор будет происходить автоматически по циклу из набора цветов по умолчанию, если график содержит несколько линий.

Цвет и стиль линий

Стиль линий настраивается при помощи ключевого аргумента `linestyle` (рисунок 13):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegrid')
6
7 plt.plot(x, x, linestyle='solid')
8 plt.plot(x, x + 1, linestyle='dashed')
9 plt.plot(x, x + 2, linestyle='dashdot')
10 plt.plot(x, x + 3, linestyle='dotted')
11 # разрешены следующие сокращения
12 plt.plot(x, x + 4, linestyle='-' ) # сплошная
13 plt.plot(x, x + 5, linestyle='--' ) # штриховая
14 plt.plot(x, x + 6, linestyle='-.-' ) # штрихпунктирная
15 plt.plot(x, x + 7, linestyle=':' ) # линия из точек
16
17 plt.show()
18
```

Цвет и стиль линий

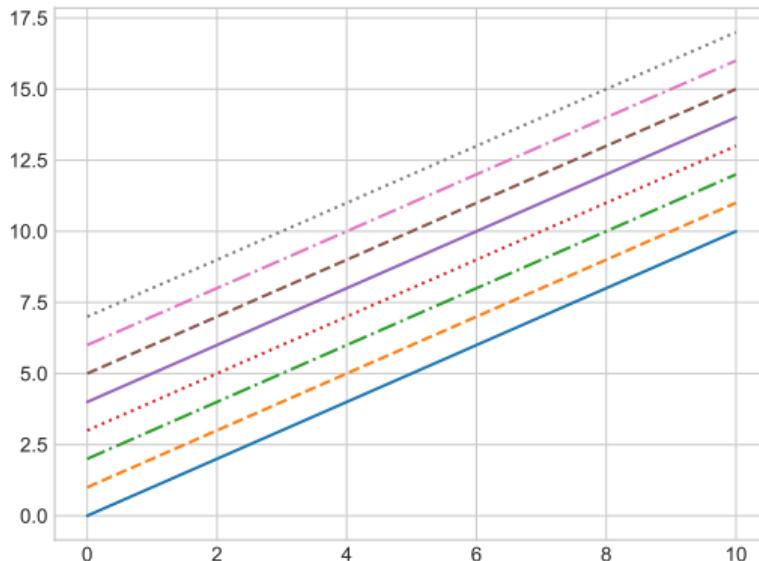


Рисунок 13 – Настройка стиля линий графика

В действительности существует множество других аргументов для более тонкой настройки внешнего вида графика. Обратитесь к документации функции `plt.plot()` для более подробного изучения.



Пределы осей координат

В библиотеке Matplotlib достаточно хорошо реализован автоматический подбор пределов осей координат, однако в некоторых случаях требуется более тонкая настройка. Наиболее легкий путь – использовать методы `plt.xlim()` и `plt.ylim()` (рисунок 14):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegrid')
6
7 x = np.linspace(0, 10, 1000)
8 plt.plot(x, np.sin(x))
9 plt.xlim(-1, 11)
10 plt.ylim(-1.5, 1.5)
11
12 plt.show()
13
```

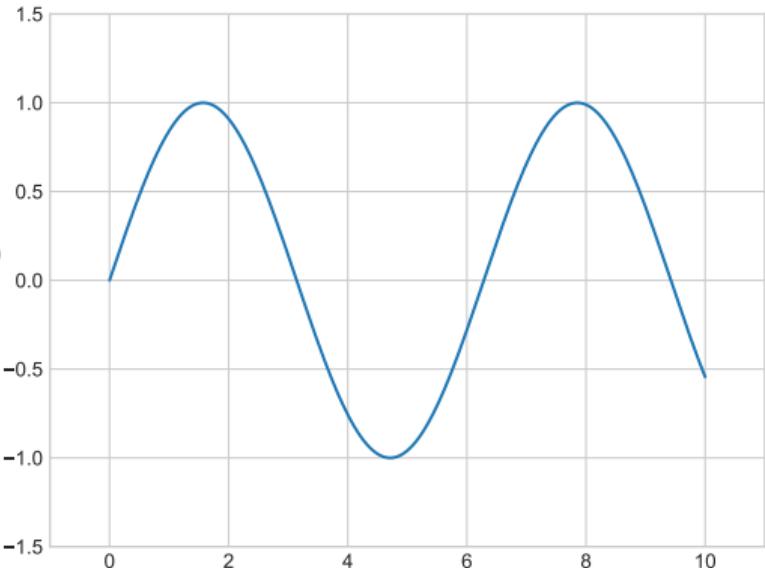


Рисунок 14 – Пример настройки пределов координатных осей



Пределы осей координат

Альтернативный способ – вызов метода `plt.axis()`. Данный метод дает возможность задавать пределы значений координатных осей путем передачи списка `[xmin, xmax, ymin, ymax]` (рисунок 15):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegrid')
6
7 x = np.linspace(0, 10, 1000)
8 plt.plot(x, np.sin(x))
9 plt.axis([0, 5, -1.1, 1.1])
10
11 plt.show()
12
```

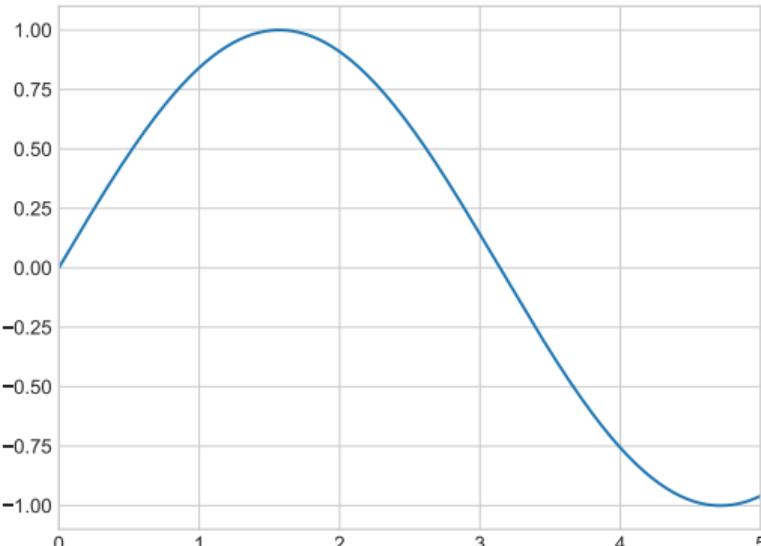


Рисунок 15 – Настройка пределов координатных осей посредством метода `plt.axis()`



Метки на графиках

Для быстрого определения названий и меток осей существуют специальные методы (рисунок 16):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegrid')
6
7 x = np.linspace(0, 10, 1000)
8 plt.plot(x, np.cos(x))
9
10 # График функции y = cos(x)
11 plt.title('График функции y = cos(x)')
12 plt.xlabel('x')
13 plt.ylabel('cos(x)')
14
15 plt.show()
16
```

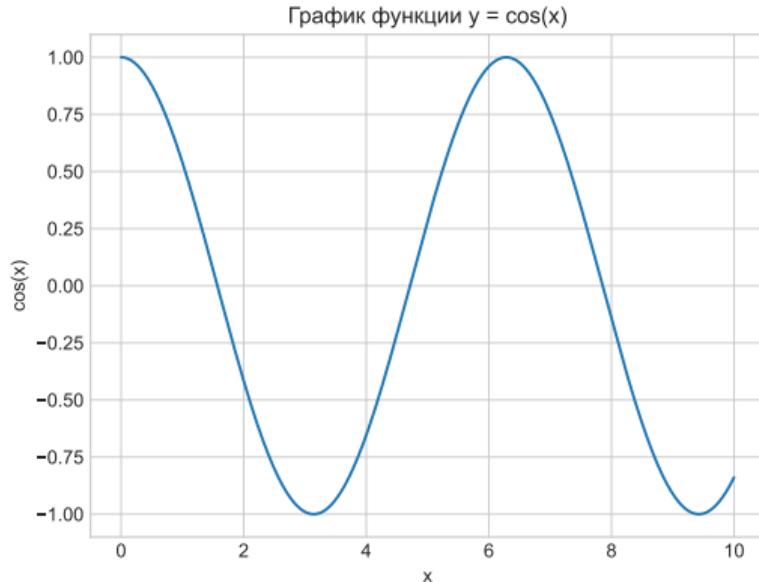


Рисунок 16 – Пример настройки заголовка графика и названий осей координат



Метки на графиках

Когда требуется отобразить несколько линий в одной координатной сетке, часто бывает необходимо создать легенду, на которой была бы отмечена каждая линия. Для быстрого решения данной задачи в Matplotlib есть метод `plt.legend()` (рисунок 17):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegrid')
6
7 x = np.linspace(0, 10, 1000)
8 plt.plot(x, np.cos(x), color='b',
9           linestyle='-', 
10          label='cos(x)')
11 plt.plot(x, np.sin(x), color='g',
12           linestyle=':', 
13          label='sin(x)')
14
15 plt.legend()
16 plt.show()
17
```

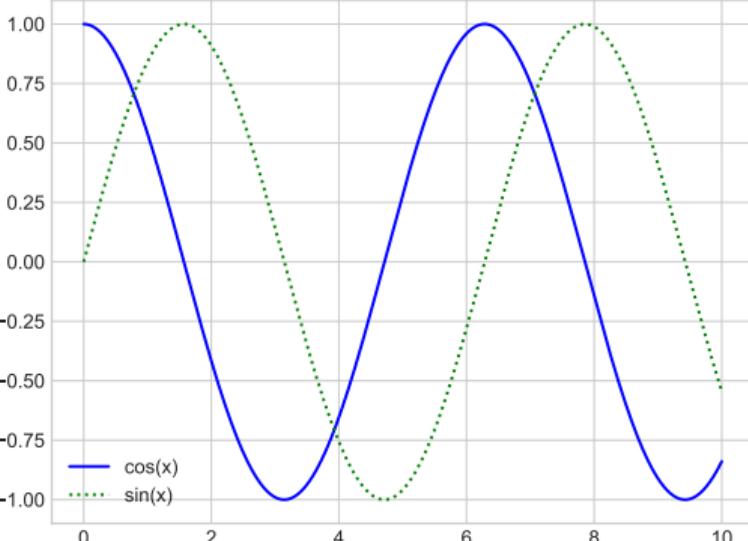


Рисунок 17 – Пример настройки легенды графика с помощью метода `plt.legend()`



Диаграммы рассеяния

Наряду с линейными графиками, еще одним распространенным типом является диаграмма рассеяния. В таких диаграммах точки не соединяются линиями, а представляются отдельно (рисунок 18):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegrid')
6
7 x = np.linspace(0, 10, 30)
8 y = np.sin(x)
9
10 plt.plot(x, y, 'o', color='black')
11 plt.show()
12
```

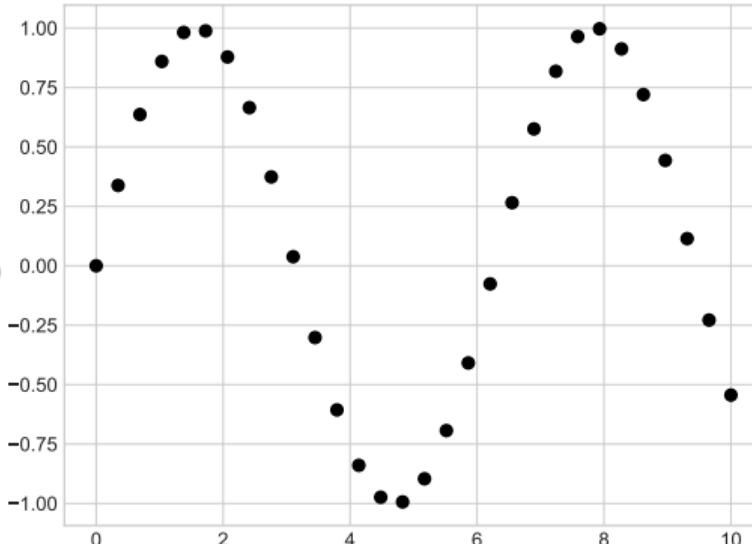


Рисунок 18 – Пример построения диаграммы рассеяния с помощью функции plt.plot()

Диаграммы рассеяния

Третий аргумент в вызове plt.plot() задает тип символа, используемого при построении графика. Для стилей маркеров существует собственный набор сокращенных строковых кодов (рисунок 19):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('tableau-colorblind10')
6
7 rng = np.random.RandomState(0)
8 for marker in ['o', '.', ',', 'x', '+', 'v', '^',
9                 '<', '>', 's', 'd']:
10     plt.plot(rng.rand(5), rng.rand(5), marker,
11               label=f"marker='{marker}'")
12
13 plt.legend()
14 plt.xlim(0, 1.8)
15
16 plt.show()
17
```

Диаграммы рассеяния

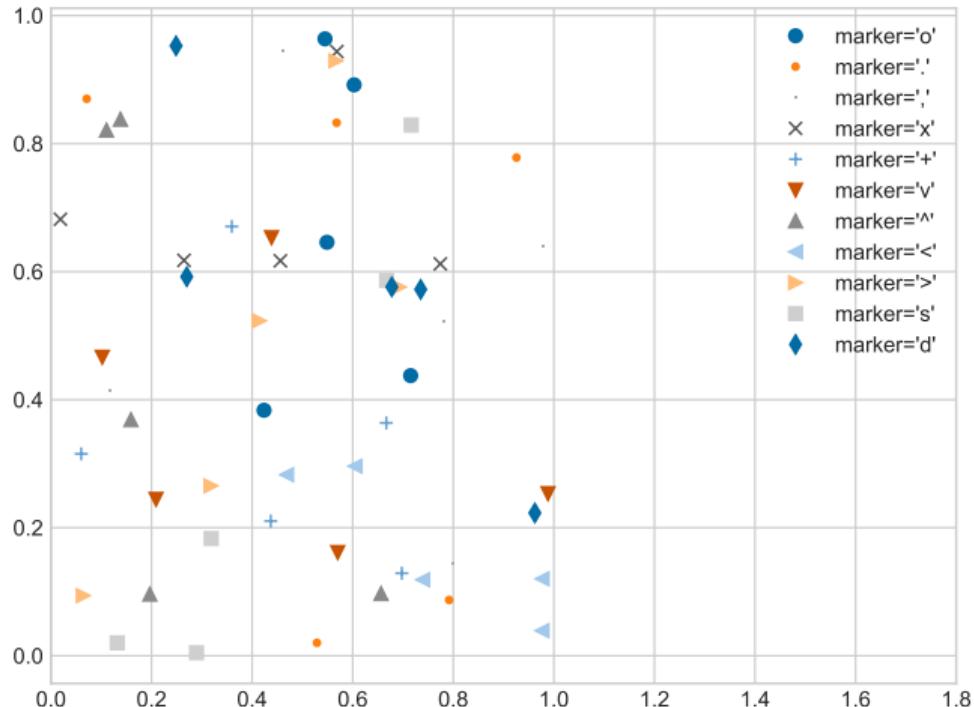


Рисунок 19 – Пример использования строковых кодов для маркеров диаграммы рассеяния



Диаграммы рассеяния

Символьные коды для маркеров разрешено использовать совместно с кодами линий и цветов, отображая на графике точки вместе с соединяющими их линиями (рисунок 20):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegrid')
6
7 x = np.linspace(0, 10, 30)
8 y = np.sin(x)
9
10 # штриховая линия (--),
11 # маркер ромба (d) черный цвет (k)
12 plt.plot(x, y, '--dk')
13 plt.show()
14
```

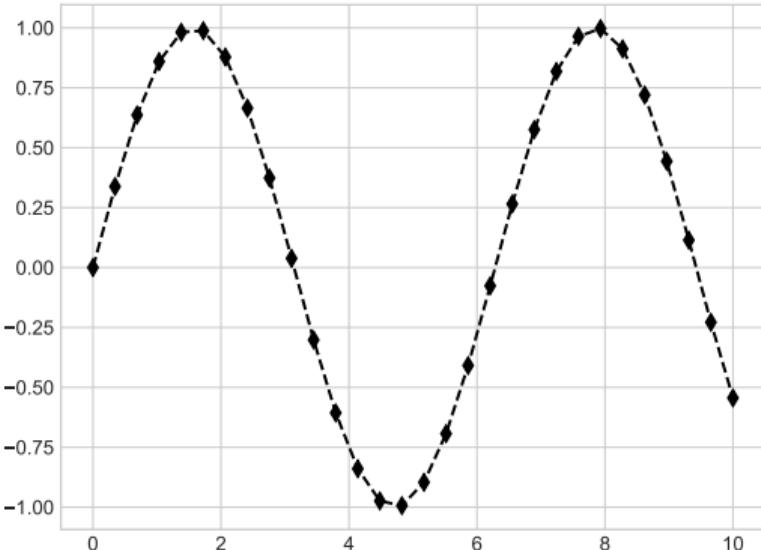


Рисунок 20 – Пример сочетания линий и маркеров точек



Диаграммы рассеяния

Используя дополнительные ключевые аргументы функции plt.plot() можно определять множество свойств линий и маркеров (рисунок 21):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegrid')
6
7 x = np.linspace(0, 10, 30)
8 y = np.sin(x)
9
10 plt.plot(x, y, '-p', color='gray',
11           markersize=14, linewidth=4,
12           markerfacecolor='white',
13           markeredgecolor='gray',
14           markeredgewidth=1)
15 plt.ylim(-1.2, 1.2)
16
17 plt.show()
```

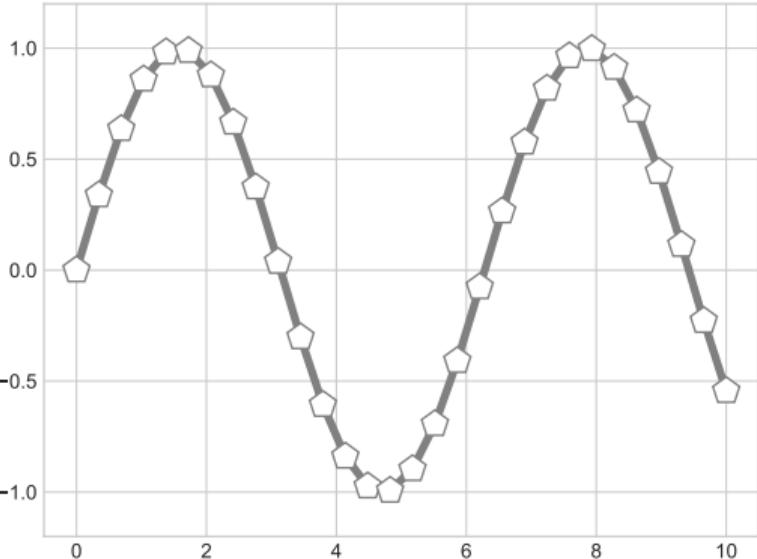


Рисунок 21 – Пример индивидуальной настройки вида линий и маркеров точек



Диаграммы рассеяния

Функция построения диаграмм `plt.scatter()`, в многом похожая на `plt.plot()`, обладает еще более расширенными возможностями (рисунок 22):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-whitegrid')
6
7 x = np.linspace(0, 10, 30)
8 y = np.sin(x)
9
10 plt.scatter(x, y, marker='o')
11
12 plt.show()
13
```

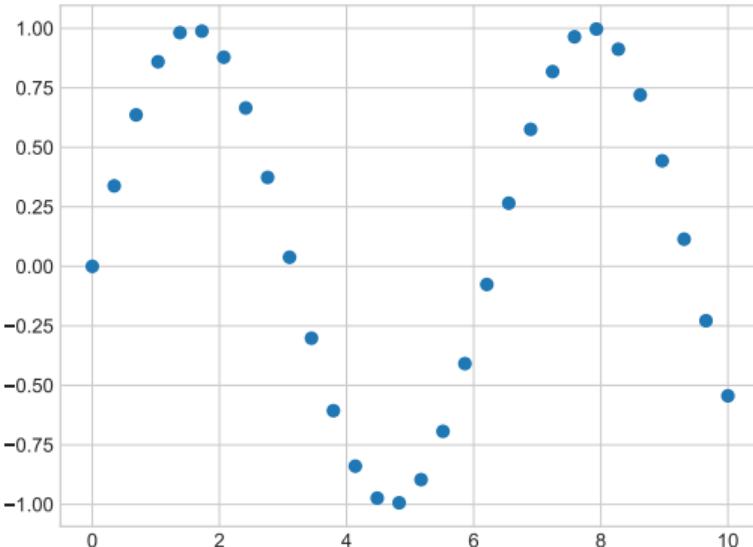


Рисунок 22 – Диаграмма рассеяния, построенная с использованием метода `plt.scatter()`



Диаграммы рассеяния

При помощи `plt.scatter()` можно создавать диаграммы рассеяния с индивидуальными свойствами каждой точки (такими как размер, заливка, цвет рамки и т.д.).

Создадим случайную диаграмму рассеяния с точками различных размеров и цветов.

Воспользуемся ключевым аргументом `alpha`, отвечающим за уровень прозрачности (рисунок 23):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 rng = np.random.RandomState(0)
4
5 x = rng.randn(100)
6 y = rng.randn(100)
7 colors = rng.rand(100)
8 sizes = 1000 * rng.rand(100)
9 plt.scatter(x, y, c=colors,
10             s=sizes, alpha=0.3,
11             cmap='viridis')
12 plt.colorbar() # цветовая шкала
13 plt.show()
14
```

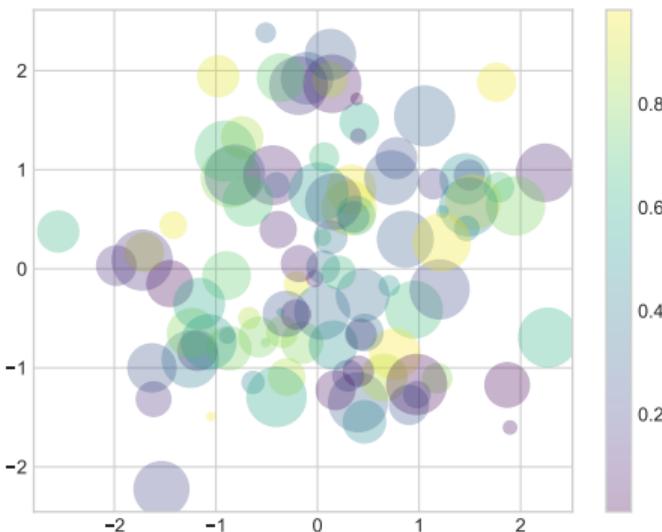


Рисунок 23 – Настройка размера, цвета и прозрачности точек диаграммы рассеяния при помощи функции `plt.scatter()`



Гистограммы

Функция `hist()` поддерживает множество параметров для настройки вычисления и отображения. Рассмотрим пример гистограммы с детальными пользовательскими настройками (рисунок 24):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 plt.style.use('seaborn-v0_8-white')
6 data = np.random.randn(1000)
7
8 plt.hist(data, bins=30, alpha=0.5,
9           color='steelblue',
10          edgecolor='none')
11
12 plt.show()
13
```

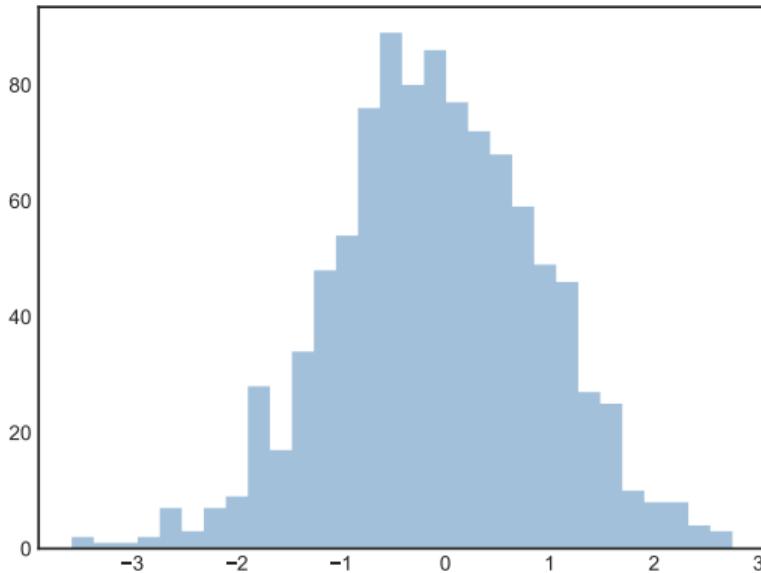


Рисунок 24 – Пример гистограммы с пользовательскими настройками

Гистограммы

Сравнение гистограмм нескольких распределений (рисунок 25):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 x1 = np.random.normal(0, 0.8, 1000)
6 x2 = np.random.normal(-2, 1, 1000)
7 x3 = np.random.normal(3, 2, 1000)
8
9 options = dict(alpha=0.3, bins=40)
10
11 plt.hist(x1, **options)
12 plt.hist(x2, **options)
13 plt.hist(x3, **options)
14
15 plt.show()
16
```

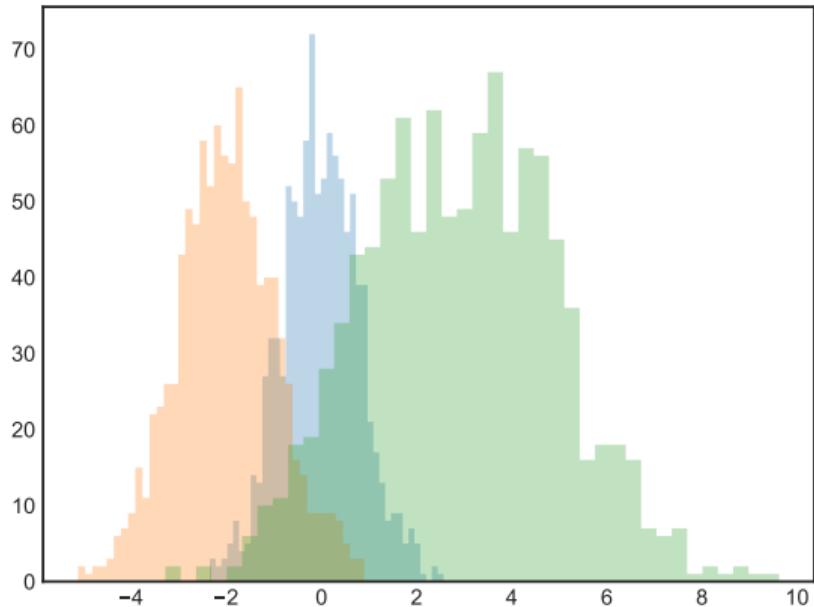


Рисунок 25 – Несколько гистограмм с наложением



Трехмерные графики

Модуль `mplplot3d` содержит инструменты для отображения трехмерных графиков. Функция `ax.contour3D()` ожидает данные в формате двумерных регулярных сеток (рисунок 26):

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def fun(x, y):
5     return np.sin(np.sqrt(x ** 2 + y ** 2))
6
7 x = np.linspace(-6, 6, 30)
8 y = np.linspace(-6, 6, 30)
9 x_mesh, y_mesh = np.meshgrid(x, y)
10 z_mesh = fun(x_mesh, y_mesh)
11
12 fig = plt.figure()
13 ax = plt.axes(projection='3d')
14 ax.contour3D(x_mesh, y_mesh, z_mesh,
15               50, cmap='cool')
16 ax.set_xlabel('x')
17 ax.set_ylabel('y')
18 ax.set_zlabel('z')
19 plt.show()
```

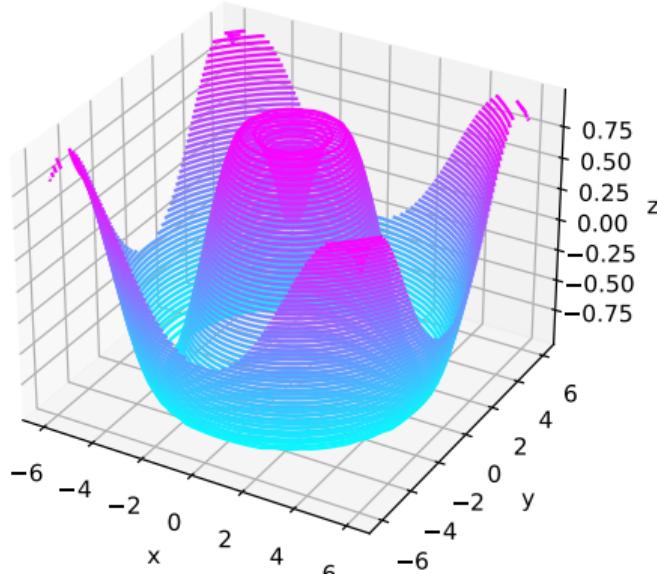


Рисунок 26 – Построение трехмерного графика при помощи функции `ax.contour3D()`



Контакты

Вячеслав Алексеевич Чузлов
к.т.н., доцент ОХИ ИШПР



Учебный корпус №2, ауд. 136



chuva@tpu.ru



+7-962-782-66-15

Благодарю за внимание!